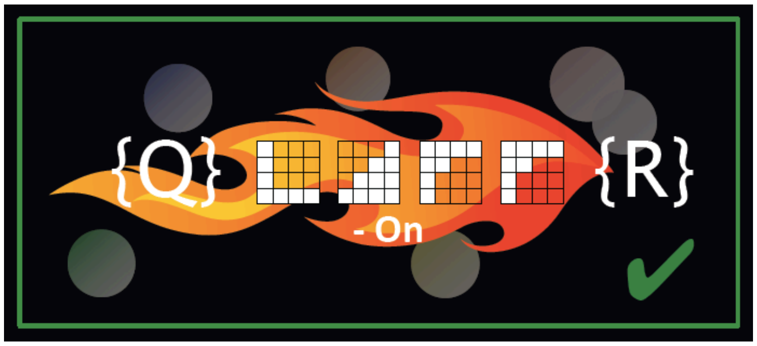


# LAFF-On

## Programming for Correctness

Margaret E. Myers

Robert A. van de Geijn

Release Date
Tuesday 13th October, 2020

**Library of Congress Cataloging-in-Publication Data** not yet available
Draft Edition, Tuesday 13th October, 2020
This "Draft Edition" allows this material to be used while we sort out through what mechanism we will publish the book.

# Contents

# Preface

☛ Click to view

In the 1972 ACM Turing Lecture "The Humble Programmer," Edsger W. Dijkstra suggested:

> "Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand. Argument three is essentially based on the following observation. If one first asks oneself what the structure of a convincing proof would be and, having found this, then constructs a program satisfying this proof's requirements, then these correctness concerns turn out to be a very effective heuristic guidance. By definition this approach is only applicable when we restrict ourselves to intellectually manageable programs, but it provides us with effective means for finding a satisfactory one among these."

This course introduces the basic tools that enable goal-oriented programming and demonstrates its practical application.

# Acknowledgments

These notes were written in support of *LAFF-On Programming for Correctness,* a Massive Open Online Course (MOOC) offered by edX, a non-profit founded by Harvard University and the Massachusetts Institute of Technology. It was funded in part by a generous gift from MathWorks and the National Science Foundation Award ACI-1550493 titled "Sustaining Innovation in the Linear Algebra Software Stack for Computational Chemistry and other Sciences"[1], which also supports our research on how to develop linear algebra software libraries. It is also supported in part by profits from the offering of Linear Algebra: Foundations and Frontiers (LAFF) on the edX platform under agreement with the Dean of Natural Sciences and the chairs of the Department of Computer Science and the Department of Statistics and Data Sciences.

The course was designed and developed by Dr. Maggie Myers and Prof. Robert van de Geijn. It is based on a Special Topics course offered by the Computer Science department titled "Programming for Correctness and Performance."

## The Team

**Dr. Maggie Myers** is a lecturer in the Department of Statistics and Data Sciences and a Research Scientist in the Institute for Computational Engineering and Sciences. She currently teaches undergraduate and graduate courses in Bayesian Statistics. Her research activities range from informal learning opportunities in mathematics education to formal derivation of linear algebra algorithms. Earlier in her career she was a senior research scientist with the Charles A. Dana Center and consultant to the Southwest Educational Development Lab (SEDL). Her partnerships (in marriage and research) with Robert have lasted for decades and seem to have survived the development of two MOOCs.

**Dr. Robert van de Geijn** is a professor of Computer Science and a core member of the Institute for Computational Engineering and Sciences. Prof. van de Geijn is a leading expert in the areas of high-performance computing, linear algebra libraries, parallel processing, and formal derivation of algorithms. He is the recipient of the 2007-2008 President's Associates Teaching Excellence Award from The University of Texas at Austin.

We gratefully acknowledge the technical support of Sejal Shah and Grace Kennedy, and early feedback on the materials from Dr. Devangi Parikh. A number of participants were instrumental as beta testers, forging ahead as soon as a new week was launched and giving us valuable feedback: Vincent DiCarlo, William Lucas, Richard Minke, and others. The "sizzle" video was created for us by Thomas Humphreys.

## Thank you all!

Finally, we would like to thank the participants for their enthusiasm and valuable comments. Their patience with our occasional shortcomings were most appreciated!

[1]*Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).*

Week 0

# Getting Started

## 0.1 Opening Remarks

### 0.1.1 Welcome to LAFF-On Programming for Correctness ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

### 0.1.2 Outline ☛ to edX

Each week is structured so that we give the outline for the week immediately after the “launch”:

### 0.1.3 What you will learn ☛ to edX

The third unit of the week informs you of what you will learn. This describes the knowledge and skills that you can expect to acquire. In addition, this provides an opportunity for you to self-assess upon completion of the week.

Upon completion of this week, you should be able to

- Recognize the structure of a typical week.
- Navigate the different components of LAFF-On.
- Activate MATLAB Online.
- Better understand what we expect you to know when you start and intend for you to know when you finish.

## 0.2 How to LAFF-On

### 0.2.1 What Should We Know? ☛ to edX

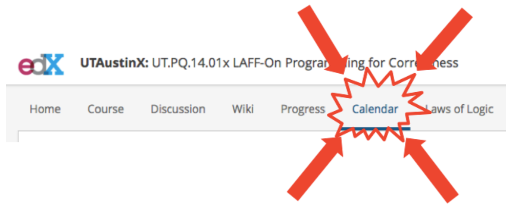
The material in this course is intended for learners who have already had an exposure to linear algebra, logic, and simple proofs. We do briefly visit these topics to ensure that we are all on the same page to allow us to communicate exactly and efficiently. For those who want to dig deeper, we try to give pointers to other materials.

For Part I, Foundation, you don't need to know any linear algebra. We use simple operations with one dimensional arrays for most of our motivating examples. The focus is on applying logic to reason about the correctness of programs and, ultimately, to allow you to systematically derive programs hand-in-hand with their proofs of correctness. We hope you will make Dijkstra proud!

For Part II, Application, we use progressively more complicated linear algebra operations to illustrate how to apply the ideas from Part I to practical situations. Abstracting away from the intricacies of indices is key to making this possible. The "slicing and dicing" that was a hallmark of our Linear Algebra: Foundations to Frontiers (LAFF) MOOC plays a central role there. Those not comfortable with thinking about vectors and matrices as subvectors and submatrices may want to explore Weeks 1, 3, 4, and 5 of LAFF. To what depth you would want to do so is up to you.

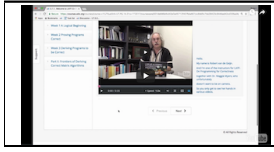
### 0.2.2 When to LAFF-On ☛ to edX

The beauty of an online course is that you get to study when you want, where you want. Still, deadlines tend to keep people moving forward. To strike a balance between flexibility and structure, we release the materials roughly one week at a time. There are no intermediate due dates but only work that is completed by the closing of the course will count towards the optional Verified Certificate. To help you structure your time, we give a suggested course calendar with proposed due dates. It can be found by clicking the "Calendar" tab of the navigation bar of the course on the edX platform:



Each Week is posted on a Tuesday at UTC 23:00. Please reference this schedule often as any official changes will appear here.

### 0.2.3 How to Navigate LAFF-On



☛ Watch Video on edX
☛ Watch Video on YouTube

### 0.2.4 Homework and LAFF-On ☛ to edX

When future weeks become available, you will notice that homework appears both in the notes and in the corresponding units on the edX platform. Most of the time, the questions will match exactly but sometimes they will be worded slightly differently.

Realize that the edX platform is ever evolving and that at some point we had to make a decision about what features we would embrace and what features did not fit our format so well. As a result, homework problems have frequently been (re)phrased in a way that fits both the platform and our course.

Some things you will notice:

- "Open" questions in the text are sometimes rephrased as multiple choice or "drag and drop" questions in the course on edX.
- Video answers appear as embedded YouTube, with the link to the end of the week where the same video, with captioning and optional download from an alternative source, can be found. This was because edX's video player could not (yet) be embedded in answers.

Please be patient with some of these decisions. Our course and the edX platform are both evolving, and sometimes we had to improvise.

### 0.2.5 Grading and LAFF-On ☛ to edX

How to grade the course was another decision that required compromise. Our fundamental assumption is that you are taking this course because you want to learn the material, and that the homework and other activities are mostly there to help you learn and self-assess. For this reason, for the homework, we

- Give you an infinite number of chances to get an answer right;
- Provide you with detailed answers;
- Allow you to view the answer any time you believe it will help you master the material efficiently;
- Include some homework that is ungraded to give those who want extra practice an opportunity to do so while allowing others to move on.

In other words, you get to use the homework in whatever way helps you learn best.

Don't forget to click on "Check" or you don't get credit for the exercise!

How your progress is measured is another interesting compromise. The homework for each of Weeks 1-5 is worth 20% of the total points in the course. Week 6 has ungraded homework, meant for those who *really* want to master the techniques. It is that week that will bring you to our frontier.

To view your progress, click on "Progress" in the edX navigation bar. If you find out that you missed a homework, scroll down the page, and you will be able to identify where to go to fix it. Don't be shy about correcting a missed answer. The primary goal is to learn.

Some of you will be disappointed that the course is not more rigorously graded, thereby (to you) diminishing the value of a certificate. The fact is that MOOCs are still evolving. People are experimenting with how to make them serve different audiences. In our case, we decided to focus on quality material first, with the assumption that for most participants the primary goal for taking the course is to learn.

Let's focus on what works, and please be patient with what doesn't!

### 0.2.6 Setting Up to LAFF-On ☛ to edX

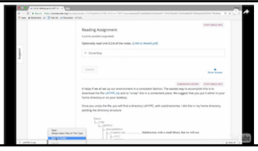
When we offered the course for the first time, we were still developing it as it was being offered. For this reason, in many of the homeworks you will find files that we suggest you download to specific places on your computer and/or to Matlab Online. For the sake of backward compatibility, we have left the video and information that was used the first year in place (below). If you follow those directions, all should be fine as well.
This time, we give you an alternative option for getting all the files. We have created a repository on github:☛ https://github.com/ULAFF/LAFFPfC. You can download the zip file or, which we recommend, you can "clone" the repository so that you can easily update files if necessary.
If you use a command window in Linux or a terminal window in Apple OS-X, you can execute

```
git clone https://github.com/ULAFF/LAFFPfC.git
```

in the directory where you want the directory LAFFPfC to exist. This will then create the directory structure, with contents, described in the video and in the text below the video.
Any time you feel the need to see if there are updates to the files, you can then execute

```
git pull
```

There are ways of doing all this on a Windows system as well, but we don't do Windows... So you will have to figure it out yourself.



☛ Watch Video on edX
☛ Watch Video on YouTube

It helps if we all set up our environment in a consistent fashion. The easiest way to accomplish this is to download the file ☛ LAFFPfC.zip and to "unzip" this in a convenient place. We suggest that you put it either in your home directory or on your desktop.

Once you unzip the file, you will find a directory LAFFPfC, with subdirectories. I did this in my home directory, yielding the directory structure in Figure 1.

## 0.3 Software to LAFF-On

### 0.3.1 Activating MATLAB Online ☛ to edX

Starting at the end of Week 3, we will implement derived to be correct programs using MATLAB.

MathWorks generously provides licenses and technical support for MATLAB Online for use in, and for the duration of, the course. Instructions on how to gain access to MATLAB Online for this course, please see this unit on the edX platform ☛ to edX. MATLAB Online is supported in Chrome (recommended for best experience), Firefox, and Safari.

### 0.3.2 MATLAB Basics ☛ to edX

Below you find a few short videos that introduce you to MATLAB. For a more comprehensive tutorial, you may want to visit ☛ MATLAB Tutorials at MathWorks and click "Launch Tutorial".

You need relatively little familiarity with MATLAB in order to learn what we want you to learn in this course. So, you could just skip these tutorials altogether, and come back to them if you find you want to know more about MATLAB and its programming language (M-script).

```
Users
└─ rvdg
   └─ LAFFPfC
      ├─ Assignments
      │  ├─ flameatlab ............ Subdirectory with a small library that we will use.
      │  │  ├─ util
      │  │  ├─ matvec
      │  │  ├─ matmat
      │  │  └─ vecvec
      │  │
      │  ├─ Week0 .................. Subdirectory for the assignments for Week 0.
      │  │  ├─ LaTeX ............... LaTeX related assignments.
      │  │  └─ matlab .............. MATLAB related assignments (for if you choose to use a desktop version
      │  │                           of MATLAB).
      │  ├─    ⋮
      │  └─ Week6 .................. Subdirectory for the assignments for Week 6.
      │
      ├─ Assignments.zip ........... File with assignments to upload to MATLAB Online.
      ├─ FLaTeX .................... Directory with support files for later exercises that use LaTeX.
      ├─ Notes ..................... We suggest you place the PDFs for the course notes here. (Download
      │  │                           from the ☛ Course Updates & News page.) .
      │  ├─ Week0.pdf
      │  ├─    ⋮
      │  └─ Week6.pdf
      ├─ Resources ................. Other resources.
      └─ Spark ..................... Local copy of Spark webpage.
```

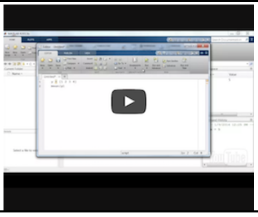
Figure 1: Directory structure for your LAFF-On materials. Items in **blue** will be placed into the materials by you. In this example, we placed `LAFFPfC` in the home directory `Users -> rvdg`. You may want to place it on your account's "Desktop" instead.

The following videos are somewhat dated and target the desktop version of MATLAB. You will want to view the videos on the edX platform instead.

**What is MATLAB?**

☛ Watch Video on edX
☛ Watch Video on YouTube

**The MATLAB Environment**



☛ Watch Video on edX
☛ Watch Video on YouTube

**MATLAB Variables**

```
>>x = 3
>> theta1 = 3.1415
```

```
x        3
theta1   3.1415
```

☛ Watch Video on edX
☛ Watch Video on YouTube

**MATLAB as a Calculator**

```
>>x = 1.5
>>dt = 0.1
>>f = (cos(x+dt)-2*cos(x)+cos(x-dt))/(2*dt)
>>f = cos(x+dt)-2*cos(x)+cos(x-dt)/2*dt
```

Calculate:

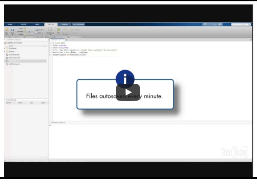
$$\frac{\cos(x+\Delta t)-2\cos(x)+\cos(x-\Delta t)}{2\Delta t}$$

where:

$x = 1.5$
$\Delta t = 0.1$

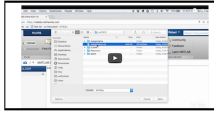
☛ Watch Video on edX
☛ Watch Video on YouTube

**Managing files with MATLAB Online**



☛ Watch Video on edX
☛ Watch Video on YouTube

### 0.3.3 Setting up MATLAB Online to LAFF-On ☛ to edX



☛ Watch Video on edX

### 0.3.4 MATLAB Live Script ☛ to edX

MATLAB Live Script is a relatively new feature of MATLAB that allows one to interleave text with executable code. It is nice for LAFF-On because we can insert explanations interleaved with coding exercises, thus enhancing the learning experience.

In this course, we will also use Live Script in an innovative way, by using the text boxes to insert the proof of correctness into the code itself. This will become clearer later in the course.

## 0.4 Typesetting LAFF-On

### 0.4.1 Typesetting mathematics ☛ to edX

The tool of choice for typesetting mathematics is a document typesetting system called LaTeX. We will use it to typeset many of our exercises in a way that captures how we want you to think about discovering algorithms hand-in-hand with their proofs of correctness.

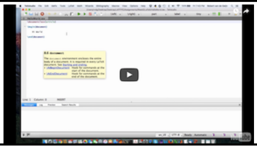
We recommend that you use TeXstudio:

> "TeXstudio an integrated writing environment for creating LaTeX documents."

It is the environment we typically use in the videos for this course and that we used to create the notes and many of the activities. You are, of course, free to use whatever such environment you prefer.

> When asking questions or posting comments on the discussion board for this course, you can also use LaTeX syntax to typeset mathematics. You place LaTeX math source between dollar signs ($) to do so.

### 0.4.2 Downloading and testing TeXstudio ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Since installing and use of TeXstudio may depend on what operating system you use, we suggest participants help each other by asking and answering questions about this on the discussion board for this unit.

You can download the open source TeXstudio software from ☛ `www.texstudio.org`.

- Open the file `Assignments/Week0/LaTeX/HelloWorld.tex`.
  - You may want to set up your computer so that the default application for "`.tex`" files is the TeXstudio application.
  - Alternatively, you can open the TeXstudio application and then click on

    ```
    File -> Open
    ```

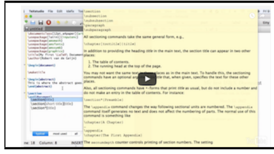
    choosing the file.
  - At this point, it is a good idea to click on

    ```
    Options -> Root Document -> Set Current Document as Explicit Root
    ```

    which will make the "HelloWorld.tex" file the root file for the "compilation" of the document. This is important when the root file itself includes other files in some hierarchical fashion, in future exercises.
- Once you have opened "`HelloWorld.tex`" Click on

  on the top bar. Eventually you will see a "Process exited normally" in the message box in the lower left-hand corner as well as the formatted text to the right.

### 0.4.3 LaTeX and TeXstudio Primer ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

The source for the LaTeX example in that video is found in

```
LAFFPfC -> Assignments -> Week0 -> LaTeX -> LaTeXPrimer.tex
```

We will use LaTeX in a very limited way. But you may want to become more familiar with this tool by following some tutorials that you can find on the internet. (Search "LaTeX tutorial").

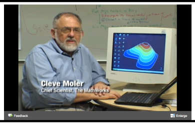
The following is probably a good place to start:

☛ LaTeX Tutorials - a Primer.

Whenever it talks about `latex myfile` and `xdvi myfile` instead just click on ▶.

## 0.5 Enrichments

### 0.5.1 The Origins of MATLAB ☛ to edX



☛ Watch Video on edX

(Or view at MathWorks.)

### 0.5.2 The Origins of LaTeX ☛ to edX

You may enjoy this interview with Dr. Leslie Lamport, who created LaTeX.
☛ `https://www.infoq.com/interviews/lamport-latex-paxos-tla`

## 0.6 Wrap Up

### 0.6.1 Additional Homework ☛ to edX

For a typical week, additional assignments may be given in this unit.

This week, we point you to other resources that delve deeper into some prerequisite knowledge. We provide limited overviews of logic (needed for the entire course) and linear algebra (needed to fully appreciate the second part of the course). One strategy is to get started with LAFF-On, and visit some of these resources as needed and desired.

### 0.6.2 Summary ☛ to edX

You will see that we develop a lot of the theory behind the various topics in linear algebra via a sequence of homework exercises. At the end of each week, we summarize theorems and insights for easy reference.

# Part I

# Foundation

Week 1

# A Logical Beginning

## 1.1 Opening Remarks

### 1.1.1 Launch ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

Consider the polynomial

$$y = p_1 + p_2 x + p_3 x^2 + \cdots + p_{n+1} x^n.$$

If the coefficients are stored in $(n+1) \times 1$ array `p` and `x` is the scalar, an efficient way of evaluating this polynomial is given by the MATLAB function

```
function y = EvalPolynomial( p, x )

  assert( size( p, 1 ) >= 1 && size( p, 2 ) == 1, ...
    'p must be a column vector with n+1 elements' )
  assert( isscalar( x), 'x must be a scalar' );

  n = size( p, 1 )-1;

  y = 0;
  for k=n+1:-1:1
    y = y * x + p(k);
  end
end
```

For those who aren't familiar with MATLAB:

- In MATLAB, all variables are multidimensional arrays (matrices, for simplicity). Here we assume the vector `p` is stored in an $(n+1) \times 1$ array (in other words, it is a column vector).
- `size( p )` extracts the row and column sizes of the matrix $p$.
- `for k=n+1:-1:1`
  means that you execute the statement between this line and the first `end` for values of `k` equal to `n+1`, `n`, through `1`, in that order.

For your coding pleasure, this exercise is also available as a MATLAB Live Script

```
LAFFPfC -> Assignments -> Week1 -> matlab -> EvalPolynomial.mlx.
```

- Decide if EvalPolynomial correctly computes the polynomial.
- Convince someone else of your conclusion.
- How sure are you?
- How long did it take you to convince yourself?
- How long did it take you to convince someone else?

We will not give you the answer now. By the end of Week 2, you will be able to determine if a program is correct, and by the end of Week 3 you will be able to derive it hand-in-hand with its proofs of correctness. You will then know that the implementation is correct before you ever execute it. Indeed, you will wonder why you are executing it at all, since you know it is correct!

### 1.1.2 Outline Week 1 ☛ to edX

### 1.1.3 What you will learn ☛ to edX

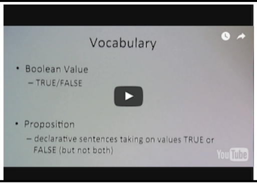
Before we begin programming for correctness, we need to start with an overview (or review) of logic since we want to create and clearly reason about correct code. Logic is essential to reasoning.

Upon completion of this week, you should be able to

- Translate between English statements and predicates that may involve quantifiers (in various ways).
- Write predicates that involve “for all”, “there exist”, “summation”, and other quantifiers.
- Prove using truth tables and equivalence style proofs.
- Prove using the Principle of Mathematical Induction.
- Evaluate and simplify predicates that involve quantifiers such as applying empty ranges, splitting ranges, and splitting off one term.
- Understand, recognize, and prove when one predicate is weaker or stronger than another predicate.
- Manipulate two predicates to expose that one is weaker than the other.

## 1.2 Review of Logic

### 1.2.1 Simple Propositions ☛ to edX


Vocabulary
- Boolean Value
  - TRUE/FALSE
- Proposition
  - declarative sentences taking on values TRUE or FALSE (but not both)

☛ Watch Video on edX
☛ Watch Video on YouTube

Translating from English to logical symbols is a foundational and challenging task. It is most valuable. Why? Writing English statements in symbolic form since they can be more concise and more precise. We put our thoughts into symbols that make reasoning easier to follow and judge for correctness.

Let's start by looking at some vocabulary and examples we run into in informal arguments. To begin, a *Boolean value* is one with two choices: TRUE or FALSE (YES or NO, 1 or 0). We will talk about *logical statements* that are declarative sentences taking on values TRUE or FALSE, known as *propositions*. For example,

- "Fivee is positive" ($5 > 0$) is a proposition that is TRUE.
- "The sum of five and three is two" or ($5+3=2$) is a proposition that is FALSE.

We often represent propositions by letters such as p, q, and r .

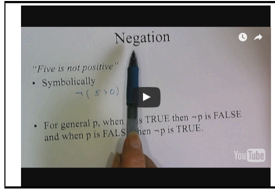
To understand what propositions are, answer the following problems.

**Homework 1.2.1.1** Decide whether or not the following are propositions:

1. Is it raining?
   (a) This is not a proposition.
   (b) This is a proposition and it evaluates to TRUE.
   (c) This is a proposition and it evaluates to FALSE.
2. Shut the window when it is raining!
   (a) This is not a proposition.
   (b) This is a proposition and it evaluates to TRUE.
   (c) This is a proposition and it evaluates to FALSE.
3. There is a number between 0 and 5 that is even.
   (a) This is not a proposition.
   (b) This is a proposition and it evaluates to TRUE.
   (c) This is a proposition and it evaluates to FALSE.
4. There is no number between 0 and 5 that is even.
   (a) This is not a proposition.
   (b) This is a proposition and it evaluates to TRUE.
   (c) This is a proposition and it evaluates to FALSE.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 1.2.2 Boolean operators ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Logical statements can be simple statements like our examples or formed from (functions of) operations with simple statements. These operations (known as Boolean operators) include:

- *not* (*negation*, $\neg$);
- *and* (*conjunction*, $\wedge$);
- *or* (*disjunction*, $\vee$);
- *implies* ( *implication*, $\Rightarrow$);
- *is equivalent to* ( *equivalence*, $\Leftrightarrow$).

For a list of logic symbols that includes LaTeX symbols, you may want to consult Wikipedia's ☛ List of logic symbols entry.

Let's define these operations by first looking at examples.

- "Five is positive" $(5 > 0)$ is a simple proposition that is always TRUE. We will represent it by $p$.
- "Five is not positive" $\neg(5 > 0)$ or $\neg p$ is a proposition that is always FALSE. We can represent it by $q$. If p represents the proposition "Five is positive" then we can represent "Five is not positive" with $\neg p$. When $p$ is TRUE then $\neg p$ is FALSE and when $p$ is FALSE then $\neg p$ is TRUE.
- "Five is positive and six is positive" is the result of two simple sentences connected with the Boolean operator and, $\wedge$. If $p$ is the logical statement "Five is positive" and $q$ is the statement that "Six is positive", then "Five is positive and six is positive" is written as p$\wedge$q. In order for the conjunction of logical expressions to be TRUE, both logical expressions must be TRUE. In this case, $p \wedge q$ is TRUE since both $p$ and $q$ are TRUE. .
- "Five is positive or six is positive" is the result of two simple sentences connected with the Boolean operator or, $\vee$. If $p$ is the logical statement "Five is positive" and $q$ is the statement "Six is positive", then "Five is positive or six is positive" is written as p$\vee$q. In order for the disjunction of logical expressions to be TRUE, at least one of logical expressions must be TRUE. In this case, $p \vee q$ is TRUE since both $p$ and $q$ are TRUE.
- "If five is non-negative then six is positive" is the result of two simple sentences connected with the Boolean implication operator, $\Rightarrow$. If $p$ is the logical statement "Five is non-negative" and $q$ is the statement "Six is positive", then "If five is non-negative then six is positive" is written as $p \Rightarrow$q. For general $p$ and $q$, in order for an implication to be TRUE, either $p$ is FALSE or both $p$ and $q$ are TRUE. The implication is FALSE if $p$ is TRUE but $q$ is FALSE. In this example, p$\Rightarrow$q is TRUE since both $p$ and $q$ are TRUE.
- "Five is non-negative is equivalent to six is positive" (or "Five is non-negative if and only if six is positive") is the result of two simple sentences connected with the Boolean equivalence operator, $\Leftrightarrow$. If $p$ is the logical statement "Five is non-negative" and $q$ is the statement "Six is positive", then "Five is non-negative iff six is positive" is given by p$\Leftrightarrow q$. For general $p$ and $q$, in order for an equivalence to be TRUE, either both $p$ and $q$ are TRUE or both $p$ and $q$ are FALSE.

The following table, known as a *truth table*, summarizes how the discussed operators evaluate.

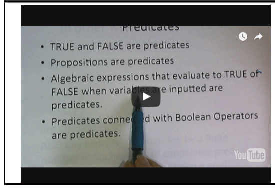
| | | not | and | or | implies | equivalent |
|---|---|---|---|---|---|---|
| $p$ | $q$ | $\neg p$ | $p \wedge q$ | p$\vee q$ | $p \Rightarrow q$ | $p \Leftrightarrow q$ |
| $T$ | $T$ | $F$ | $T$ | $T$ | $T$ | $T$ |
| $T$ | $F$ | $F$ | $F$ | $T$ | $F$ | $F$ |
| $F$ | $T$ | $T$ | $F$ | $T$ | $T$ | $F$ |
| $F$ | $F$ | $T$ | $F$ | $F$ | $T$ | $T$ |

**Homework 1.2.2.1** Complete the truth table for $(p \Rightarrow q) \wedge (q \Rightarrow r)$.

| $p$ | $q$ | $r$ | $p \Rightarrow q$ | $q \Rightarrow r$ | $(p \Rightarrow q) \wedge (q \Rightarrow r)$ |
|---|---|---|---|---|---|
| $T$ | $T$ | $T$ | | | |
| $T$ | $T$ | $F$ | | | |
| $T$ | $F$ | $T$ | | | |
| $T$ | $F$ | $F$ | | | |
| $F$ | $T$ | $T$ | | | |
| $F$ | $T$ | $F$ | | | |
| $F$ | $F$ | $T$ | | | |
| $F$ | $F$ | $F$ | | | |

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 1.2.3 Predicates ☛ to edX



Predicates

- TRUE and FALSE are predicates
- Propositions are predicates
- Algebraic expressions that evaluate to TRUE or FALSE when variables are inputted are predicates.
- Predicates connected with Boolean Operators are predicates.

☛ Watch Video on edX
☛ Watch Video on YouTube

**Propositions** are expressions that are either TRUE or FALSE. Sometimes, our logical expressions involve variables. A **predicate** is a logical expression that may be TRUE or FALSE depending on the values of the variables that appear in the expression.

- If a predicate is always TRUE no matter the values of its variables, it is known as a **tautology**.
- If it is always FALSE, it is called a **contradiction**.
- If it is sometimes TRUE and sometimes FALSE, it is known as a **contingency**.

Now, predicates could be constants– as in, they may not have variables involved. In particular, propositions are also predicates. So, for example, TRUE, FALSE, and $5 > 0$ are predicates. In addition, algebraic expressions that evaluate to TRUE or FALSE, depending on the input variables, are predicates. Furthermore, predicates connected with Boolean operators are also predicates.

Some examples:

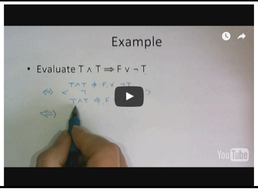
- Let $x$ be a variable that takes on integer values. "$x$ is positive" is a simple sentence that is TRUE or FALSE depending on the value that $x$ takes on so it is a contingency. It is not both when $x$ is specified. We can assign $p$ to represent this predicate.
- Let $x$ and $y$ be variables that take on integer values. "$x$ is positive and $y$ is positive" is the result of two simple sentences connected with the Boolean operator and, $\wedge$ . If $p$ is the logical statement that $x$ is positive and $q$ is the statement that $y$ is positive, then "$x$ is positive and $y$ is positive" is written as $p \wedge q$ . In order for the predicate to be TRUE, both simple predicates $p$ and $q$ must be TRUE. In this case, both $x$ and $y$ must be positive.
- Let $x$ and $y$ be variables that take on integer values. "$x$ is positive or $y$ is positive" is the result of two simple sentences connected with the Boolean operator or, $\vee$ . If $p$ is the logical statement that $x$ is positive and $q$ is the statement that "$y$ is positive, then $x$ is positive or $y$ is positive" is written as $p \vee q$ . In order for the logical expression to be TRUE, one of the simple sentences $p$ and $q$ must be TRUE (and both CAN be TRUE). In this case, both $x$ and $y$ must be positive.

**Homework 1.2.3.1** Let $x$ and $y$ be variables that take on integer values. Let $p$ be the statement "$x$ is positive" and $q$ be the statement "$y$ is positive". Determine the symbolic statements for the following predicates described using English. Mark all that are correct. (There may be multiple correct answers.)

1. Both $x$ and $y$ are positive.
   (a) $p \wedge q$
   (b) $p \vee q$
   (c) $\neg(p \wedge q)$
   (d) $p \Rightarrow q$
2. Either $x$ or $y$ is positive.
   (a) $p \wedge q$
   (b) $p \vee q$
   (c) $\neg(p \wedge q)$
   (d) $p \Rightarrow q$
3. $x$ is positive but $y$ is not.
   (a) $\neg q$
   (b) $\neg(p \wedge q)$
   (c) $\neg p \vee \neg q$
   (d) $p \wedge \neg q$
4. Either $x$ or $y$ is positive but not both are positive.
   (a) $p \wedge q$
   (b) $(p \vee q) \wedge \neg(p \wedge q)$
   (c) $\neg p \vee q$
   (d) $(p \wedge \neg q) \vee (\neg p \wedge q)$
5. $x$ is not positive and $y$ is not positive.
   (a) $\neg(p \wedge q)$
   (b) $\neg(p \vee q)$
   (c) $\neg p \wedge \neg q$
   (d) $\neg p \vee \neg q$
6. At least one of $x$ and $y$ is not positive.
   (a) $\neg(p \wedge q)$
   (b) $\neg(p \vee q)$
   (c) $\neg p \wedge \neg q$
   (d) $\neg p \vee \neg q$
7. Neither $x$ nor $y$ is positive.
   (a) $\neg p$
   (b) $\neg(p \wedge q)$
   (c) $\neg p \wedge \neg q$
   (d) $\neg p \vee \neg q$
8. It is not the case that both $x$ and $y$ are positive.
   (a) $\neg(p \wedge q)$
   (b) $\neg(p \vee q)$
   (c) $\neg p \wedge q$
   (d) $p \wedge \neg q$
9. Both $x$ and $y$ are not positive.
   (a) $\neg p$
   (b) $\neg(p \wedge q)$
   (c) $\neg p \wedge \neg q$
   (d) $\neg p \vee \neg q$
   (e) not clear
10. If x is positive then y is positive.
    (a) $p \wedge q$
    (b) $p \Rightarrow q$
    (c) $p \vee q$
    (d) $\neg(p \wedge q)$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 1.2.4 Precedence of Boolean operators ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

To avoid the unnecessary extra parentheses, there is a *precedence order* in which operations are evaluated:

- First negate: $\neg p \wedge q$ is the same as $(\neg p) \wedge q$.
- Second, evaluate $\wedge$: $\neg p \wedge q \Rightarrow r$ is the same as $((\neg p) \wedge q) \Rightarrow r$.
- Third, evaluate $\vee$: $\neg p \wedge q \vee r \Rightarrow s$ is the same as $(((\neg b) \wedge c) \vee r) \Rightarrow s$.
- Fourth, evaluate $\Rightarrow$: $t \Leftrightarrow \neg p \wedge q \vee r \Rightarrow s$ is the same as $t \Leftrightarrow ((((\neg b) \wedge c) \vee r) \Rightarrow s)$.
- Last evaluate $\Leftrightarrow$.

We will throw in extra parentheses if we think it makes the predicate clearer. So should you!

**Homework 1.2.4.1** Let $p = F$, $q = F$, and $r = F$. Evaluate

- $p \wedge q \Rightarrow r$
- $(p \wedge q) \Rightarrow r$
- $p \wedge (q \Rightarrow r)$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 1.2.4.2** Evaluate

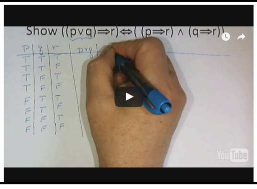
$$T \vee \neg T \wedge F \Rightarrow T \wedge \neg T \Leftrightarrow T \Rightarrow F.$$

a) $T$

b) $F$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 1.2.5 Proving using truth tables ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

As we mentioned previously, truth tables give the truth values of predicates under all combinations of the values of the components. We can prove that predicates are tautologies by examining all combinations of the input values. The following video illustrates how to construct a truth table for proving the contrapositive.

**Example 1.1** *Prove the contrapositive:* $(E1 \Rightarrow E2) \Leftrightarrow (\neg E2 \Rightarrow \neg E1)$ *with a truth table.*

*We start by setting up the truth table:*

| $E1$ | $E2$ | $\neg E1$ | $\neg E2$ | $(E1 \Rightarrow E2)$ | $(\neg E2 \Rightarrow \neg E1)$ | $(E1 \Rightarrow E2) \Leftrightarrow (\neg E2 \Rightarrow \neg E1)$ |
|---|---|---|---|---|---|---|
| $T$ | $T$ | | | | | |
| $T$ | $F$ | | | | | |
| $F$ | $T$ | | | | | |
| $F$ | $F$ | | | | | |

*and then systematically evaluate first the components* $\neg E1$, $\neg E2$:

| $E1$ | $E2$ | $\neg E1$ | $\neg E2$ | $(E1 \Rightarrow E2)$ | $(\neg E2 \Rightarrow \neg E1)$ | $(E1 \Rightarrow E2) \Leftrightarrow (\neg E2 \Rightarrow \neg E1)$ |
|---|---|---|---|---|---|---|
| $T$ | $T$ | $F$ | $F$ | | | |
| $T$ | $F$ | $F$ | $T$ | | | |
| $F$ | $T$ | $T$ | $F$ | | | |
| $F$ | $F$ | $T$ | $T$ | | | |

*and then* $(E1 \Rightarrow E2)$ *and* $(\neg E2 \Rightarrow \neg E1)$,

| $E1$ | $E2$ | $\neg E1$ | $\neg E2$ | $(E1 \Rightarrow E2)$ | $(\neg E2 \Rightarrow \neg E1)$ | $(E1 \Rightarrow E2) \Leftrightarrow (\neg E2 \Rightarrow \neg E1)$ |
|---|---|---|---|---|---|---|
| $T$ | $T$ | $F$ | $F$ | $T$ | $T$ | |
| $T$ | $F$ | $F$ | $T$ | $F$ | $F$ | |
| $F$ | $T$ | $T$ | $F$ | $T$ | $T$ | |
| $F$ | $F$ | $T$ | $T$ | $T$ | $T$ | |

,

*which then allows us to evaluate the desired result* $(E1 \Rightarrow E2) \Leftrightarrow (\neg E2 \Rightarrow \neg E1)$.

| $E1$ | $E2$ | $\neg E1$ | $\neg E2$ | $(E1 \Rightarrow E2)$ | $(\neg E2 \Rightarrow \neg E1)$ | $(E1 \Rightarrow E2) \Leftrightarrow (\neg E2 \Rightarrow \neg E1)$ |
|---|---|---|---|---|---|---|
| $T$ | $T$ | $F$ | $F$ | $T$ | $T$ | $T$ |
| $T$ | $F$ | $F$ | $T$ | $F$ | $F$ | $T$ |
| $F$ | $T$ | $T$ | $F$ | $T$ | $T$ | $T$ |
| $F$ | $F$ | $T$ | $T$ | $T$ | $T$ | $T$ |

**Homework 1.2.5.1** Use a truth table to prove the commutativity of the $\wedge$ operator:

$$(\mathrm{E1} \wedge \mathrm{E2}) \Leftrightarrow (\mathrm{E2} \wedge \mathrm{E1}).$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

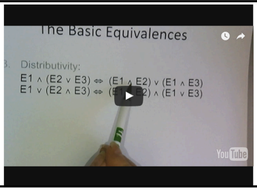
**Homework 1.2.5.2** Use a truth table to prove

$$p \wedge q \Rightarrow p.$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 1.3 Proof Techniques for LAFF-On

### 1.3.1 Basic Equivalences ☛ to edX


The Basic Equivalences
Distributivity:
E1 ∧ (E2 ∨ E3) ⇔ (E1 ∧ E2) ∨ (E1 ∧ E3)
E1 ∨ (E2 ∧ E3) ⇔ (E1 ∨ E2) ∧ (E1 ∨ E3)
YouTube

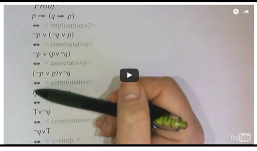
☛ Watch Video on edX
☛ Watch Video on YouTube

There are a variety of proof styles and techniques that one can use for predicate logic. We will use equivalence style proofs that start with a set of Basic Equivalences, given in Figure 1.1. It is these equivalences, and a few other laws introduced later, that we tend to use to justify our reasoning. Each can be easily proven via truth tables.

You should learn them — and their names — well. **The tab "Laws of Logic" in the LAFF-On navigation bar on the edX platform, is where you can also find Figure 1.1.**

### 1.3.2 Equivalence style proofs ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube

Let us illustrate what we mean by an equivalence style proof with an example:

**Example 1.2** *Prove the contrapositive: $(E1 \Rightarrow E2) \Leftrightarrow (\neg E2 \Rightarrow \neg E1)$ using only the other Basic Equivalences.*

---

**Proof:** *It often helps to start with the most complicated side:*

$$\begin{array}{ll} & \neg E2 \Rightarrow \neg E1 \\ \Leftrightarrow & < implication > \\ & \neg(\neg E2) \vee \neg E1 \\ \Leftrightarrow & < negation > \\ & E2 \vee \neg E1 \\ \Leftrightarrow & < commutativity > \\ & \neg E1 \vee E2 \\ \Leftrightarrow & < implication > \\ & E1 \Rightarrow E2 \end{array}$$

*Hence, by transitivity of equivalence, the two predicates are equal.*

---

Let E1 , E2 , and E3 be any propositions. Then

| | | | |
|---|---|---|---|
| Commutativity: | $(\mathrm{E1} \wedge \mathrm{E2})$ | $\Leftrightarrow$ | $(\mathrm{E2} \wedge \mathrm{E1})$ |
| | $(\mathrm{E1} \vee \mathrm{E2})$ | $\Leftrightarrow$ | $(\mathrm{E2} \vee \mathrm{E1})$ |
| | $(\mathrm{E1} \Leftrightarrow \mathrm{E2})$ | $\Leftrightarrow$ | $(\mathrm{E2} \Leftrightarrow \mathrm{E1})$ |
| Associativity: | $(\mathrm{E1} \wedge (\mathrm{E2} \wedge \mathrm{E3}))$ | $\Leftrightarrow$ | $((\mathrm{E1} \wedge \mathrm{E2}) \wedge \mathrm{E3})$ |
| | $(\mathrm{E1} \vee (\mathrm{E2} \vee \mathrm{E3}))$ | $\Leftrightarrow$ | $((\mathrm{E1} \vee \mathrm{E2}) \vee \mathrm{E3})$ |
| Distributivity: | $(\mathrm{E1} \wedge (\mathrm{E2} \vee \mathrm{E3}))$ | $\Leftrightarrow$ | $((\mathrm{E1} \wedge \mathrm{E2}) \vee (\mathrm{E1} \wedge \mathrm{E3}))$ |
| | $(\mathrm{E1} \vee (\mathrm{E2} \wedge \mathrm{E3}))$ | $\Leftrightarrow$ | $((\mathrm{E1} \vee \mathrm{E2}) \wedge (\mathrm{E1} \vee \mathrm{E3}))$ |
| De Morgan's laws: | $\neg(\mathrm{E1} \wedge \mathrm{E2})$ | $\Leftrightarrow$ | $(\neg\mathrm{E1} \vee \neg\mathrm{E2})$ |
| | $\neg(\mathrm{E1} \vee \mathrm{E2})$ | $\Leftrightarrow$ | $(\neg\mathrm{E1} \wedge \neg\mathrm{E2})$ |
| Negation: | $\neg(\neg\mathrm{E1})$ | $\Leftrightarrow$ | $\mathrm{E1}$ |
| Excluded Middle: | $(\mathrm{E1} \vee \neg\mathrm{E1})$ | $\Leftrightarrow$ | $T$ |
| Contradiction: | $(\mathrm{E1} \wedge \neg\mathrm{E1})$ | $\Leftrightarrow$ | $F$ |
| Implication: | $(\mathrm{E1} \Rightarrow \mathrm{E2})$ | $\Leftrightarrow$ | $(\neg\mathrm{E1} \vee \mathrm{E2})$ |
| Equivalence: | $(\mathrm{E1} \Leftrightarrow \mathrm{E2})$ | $\Leftrightarrow$ | $(\mathrm{E1} \Rightarrow \mathrm{E2}) \wedge (\mathrm{E2} \Rightarrow \mathrm{E1})$ |
| $\vee$-simplification: | $(\mathrm{E1} \vee \mathrm{E1})$ | $\Leftrightarrow$ | $\mathrm{E1}$ |
| | $(\mathrm{E1} \vee T)$ | $\Leftrightarrow$ | $T$ |
| | $(\mathrm{E1} \vee F)$ | $\Leftrightarrow$ | $\mathrm{E1}$ |
| | $(\mathrm{E1} \vee (\mathrm{E1} \wedge \mathrm{E2}))$ | $\Leftrightarrow$ | $\mathrm{E1}$ |
| $\wedge$-simplification: | $(\mathrm{E1} \wedge \mathrm{E1})$ | $\Leftrightarrow$ | $\mathrm{E1}$ |
| | $(\mathrm{E1} \wedge T)$ | $\Leftrightarrow$ | $\mathrm{E1}$ |
| | $(\mathrm{E1} \wedge F)$ | $\Leftrightarrow$ | $F$ |
| | $(\mathrm{E1} \wedge (\mathrm{E1} \vee \mathrm{E2}))$ | $\Leftrightarrow$ | $\mathrm{E1}$ |
| Identity: | $\mathrm{E1}$ | $\Leftrightarrow$ | $\mathrm{E1}$ |
| $\Rightarrow$-simplification: | $(\mathrm{E1} \Rightarrow \mathrm{E1})$ | $\Leftrightarrow$ | $T$ |
| | $(F \Rightarrow \mathrm{E1})$ | $\Leftrightarrow$ | $T$ |
| | $(\mathrm{E1} \Rightarrow T)$ | $\Leftrightarrow$ | $T$ |
| | $(T \Rightarrow E1)$ | $\Leftrightarrow$ | $E1$ |
| Contrapositive | $(\mathrm{E1} \Rightarrow \mathrm{E2})$ | $\Leftrightarrow$ | $(\neg\mathrm{E2} \Rightarrow \neg\mathrm{E1})$ |

Figure 1.1: Basic Equivalences.

What do we notice in this example: We created a sequence of equivalences $P_0 \Leftrightarrow P_1$, $P_1 \Leftrightarrow P_2$, $\ldots$, $P_{n-2} \Leftrightarrow P_{n-1}$, presented as

$$\begin{array}{ll} & P_0 \\ \Leftrightarrow & < \text{justification} > \\ & P_1 \\ \Leftrightarrow & < \text{justification} > \\ & \vdots \\ \Leftrightarrow & < \text{justification} > \\ & P_{n-1} \end{array}$$

By transitivity ($p \Leftrightarrow q$ and $q \Leftrightarrow r$ implies that $p \Leftrightarrow r$), we conclude that $P_0 \Leftrightarrow P_{n-1}$. Each step is typically an application of one or more Basic Equivalences or similar law.

In our proof above (and in the video), we started with the most complicated side and then reasoned towards the less complicated side. Alternatively, we can establish that a given predicate is a tautology by showing it is equivalent to TRUE. For example, we can alternatively prove the contrapositive with the following proof:

$$\begin{array}{ll} & (\text{E1} \Rightarrow \text{E2}) \Leftrightarrow (\neg\text{E2} \Rightarrow \neg\text{E1}) \\ \Leftrightarrow & < \text{implication} \times 2 > \\ & (\neg\text{E1} \vee \text{E2}) \Leftrightarrow (\neg(\neg\text{E2}) \vee \neg\text{E1}) \\ \Leftrightarrow & < \text{negation} > \\ & (\neg\text{E1} \vee \text{E2}) \Leftrightarrow (\text{E2} \vee \neg\text{E1}) \\ \Leftrightarrow & < \text{commutativity} > \\ & (\neg\text{E1} \vee \text{E2}) \Leftrightarrow (\neg\text{E1} \vee \text{E2}) \\ \Leftrightarrow & < \text{identity} > \\ & T \end{array}$$

Notice that there are other styles of proofs. Let's embrace equivalence style proofs so we are all on the same page. This will make it easier for us to communicate with each other.

The following two tautologies will become important in Unit 2.4.3 when we prove the If Theorem (which itself allows one to prove an "if" command correct):

**Homework 1.3.2.1** Prove that $(p \Rightarrow (q \wedge r)) \Leftrightarrow ((p \Rightarrow q) \wedge (p \Rightarrow r))$.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

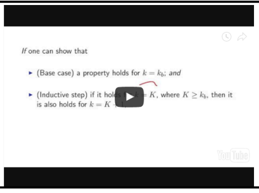
**Homework 1.3.2.2** Prove that $(p \Rightarrow (q \Rightarrow r)) \Leftrightarrow (p \wedge q \Rightarrow r)$.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 1.3.2.3** Prove that $(p \Rightarrow (q \vee r)) \Leftrightarrow ((p \Rightarrow q) \vee (p \Rightarrow r))$.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 1.3.3 (The Principle of Mathematical) Induction ☛ to edX



*If* one can show that

- (Base case) a property holds for $k = k_b$; *and*
- (Inductive step) if it holds f $k = K$, where $K \geq k_b$, then it is also holds for $k = K$

☛ Watch Video on edX
☛ Watch Video on YouTube

We will see starting in Week 2 that the Principle of Mathematical Induction (often simply called "induction") is perhaps the most fundamental of all mathematical theorems to programming. It is what will allow us to prove loops correct. (It also plays an important role in proving recursive functions correct, but we do not stress that in this course.)

The material in the remainder of this unit was duplicated from *Linear Algebra: Foundations to Frontiers – Notes to LAFF With*.

If we want to show something to be true for all integer values greater than or equal to $k_b$, then the **Principle of Mathematical Induction** (weak induction) says that *if* one can show that

- (Base case) a property holds for $k = k_b$; *and*
- (Inductive step) if it holds for $k = K$, where $K \geq k_b$, then it is also holds for $k = K + 1$,

*then* one can conclude that the property holds for all integers $k \geq k_b$. Often $k_b = 0$ or $k_b = 1$.

Here is Maggie's take on Induction, extending it beyond the proofs we do.

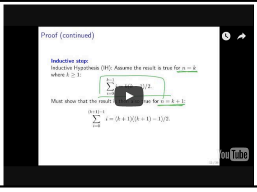
> In fact, if you want to prove something holds for all members of a set that can be defined inductively, then you would use mathematical induction. What does it mean for a set to be defined inductively? You may recall a set is a collection and as such the order of its members is not important. However, some sets do have a natural ordering that can be used to describe the membership. This is especially valuable when the set has an infinite number of members, for example, natural numbers. Sets for which the membership can be described by suggesting there is a first element (or small group of firsts) then from this first you can create another (or others) then more and more by applying a rule to get another element in the set are our focus here. If all elements (members) are in the set because they are either the first (basis) or can be constructed by applying the rule to the first (basis) a finite number of times, then the set can be inductively defined.
>
> So for us, the set of natural numbers is inductively defined. As a computer scientist you would say 0 is the first and the rule is to add one to get another element. So $0, 1, 2, 3, \ldots$ are members of the natural numbers. In this way, 10 is a member of natural numbers because you can find it by adding 1 to 0 ten times to get it.
>
> So, the Principle of Mathematical induction proves that something is true for all of the members of a set that can be defined inductively. If this set has an infinite number of members, you couldn't show it is true for each of them individually. The idea is if it is true for the first(s) and it is true for any next constructed member(s) no matter where you are in the list, it must be true for all. Why? Since we are proving things about natural numbers, the idea is if it is true for 0 and the next constructed, it must be true for 1 but then its true for 2, and then 3 and 4 and 5 and $\cdots$ and 10 and $\cdots$ and 10000 and 10001 , and so forth (all natural numbers). This is only because of the special ordering we can put on this set so we can know there is a next one for which it must be true. People often picture this rule by thinking of climbing a ladder or pushing down dominoes. If you know you started and you know wherever you are the next will follow, then you must make it through all (even if there are an infinite number).
>
> That is why to prove something using the Principle of Mathematical Induction you must show what you are proving holds at a start and then if it holds (assume it holds up to some point) then it holds for the next constructed element in the set. With these two parts shown, we know it must hold for all members of this inductively defined set.
>
> You can find many examples of how to prove using PMI as well as many examples of when and why this method of proof will fail all over the web. Notice it only works for statements about sets "that can

be defined inductively". Also notice subsets of natural numbers can often be defined inductively. For example, if I am a mathematician I may start counting at 1. Or I may decide that the statement holds for natural numbers $\geq 4$ so I start my base case at 4.

My last comment in this very long message is that this style of proof extends to other structures that can be defined inductively (such as trees or special graphs in CS).

If mathematical induction intimidates you, have a look at ☛ Puzzles and Paradoxes in Mathematical Induction, by Adam Bjorndahl.

### 1.3.4 The Principle of Mathematical Induction: Examples ☛ to edX



Proof (continued)

**Inductive step:**
Inductive Hypothesis (IH): Assume the result is true for $n = k$ where $k \geq 1$:

$$\sum_{i=0}^{k-1} i = k(k-1)/2.$$

Must show that the result is then also true for $n = k + 1$:

$$\sum_{i=0}^{(k+1)-1} i = (k+1)((k+1)-1)/2.$$

☛ Watch Video on edX
☛ Watch Video on YouTube

**Example 1.3**

$$\sum_{i=0}^{n-1} i = n(n-1)/2.$$

---

**Proof:**

**Base case:** $n = 1$. *For this case, we must show that* $\sum_{i=0}^{1-1} i = 1(0)/2$.

$\sum_{i=0}^{1-1} i$

$=$ *< Definition of summation>*

$0$

$=$ *< arithmetic>*

$1(0)/2$

*This proves the base case.*

**Inductive step:** *Inductive Hypothesis (IH): Assume that the result is true for* $n = k$ *where* $k \geq 1$*:*

$$\sum_{i=0}^{k-1} i = k(k-1)/2.$$

*We will show that the result is then also true for* $n = k+1$*:*

$$\sum_{i=0}^{(k+1)-1} i = (k+1)((k+1)-1)/2.$$

---

*Assume that* $k \geq 1$. *Then*

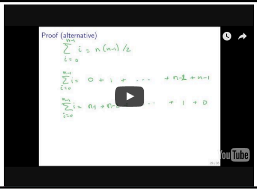
$$
\begin{array}{ll}
\sum_{i=0}^{(k+1)-1} i & \\
= & < arithmetic > \\
\sum_{i=0}^{k} i & \\
= & < split\ off\ last\ term > \\
\sum_{i=0}^{k-1} i + k & \\
= & < I.H. > \\
k(k-1)/2 + k. & \\
= & < algebra > \\
(k^2 - k)/2 + 2k/2. & \\
= & < algebra > \\
(k^2 + k)/2. & \\
= & < algebra > \\
(k+1)k/2. & \\
= & < arithmetic > \\
(k+1)((k+1)-1)/2. &
\end{array}
$$

*This proves the inductive step.*

**By the Principle of Mathematical Induction** *the result holds for all n.*

---

As we become more proficient, we will start combining steps. For now, we give lots of detail to make sure everyone stays on board.



☛ Watch Video on edX
☛ Watch Video on YouTube

There is an alternative proof for this result which does not involve mathematical induction. We give this proof now because it is a convenient way to re-derive the result should you need it in the future.

---

**Proof:**(alternative)

$$
\begin{array}{ccccccccccc}
\sum_{i=0}^{n-1} i & = & 0 & + & 1 & + & \cdots & + & (n-2) & + & (n-1) \\
\sum_{i=0}^{n-1} i & = & (n-1) & + & (n-2) & + & \cdots & + & 1 & + & 0 \\
\hline
2\sum_{i=0}^{n-1} i & = & \multicolumn{9}{c}{\underbrace{(n-1) + (n-1) + \cdots + (n-1) + (n-1)}_{n \text{ times the term } (n-1)}}
\end{array}
$$

so that $2\sum_{i=0}^{n-1} i = n(n-1)$. Hence $\sum_{i=0}^{n-1} i = n(n-1)/2$.

---

For those who don't like the "$\cdots$" in the above argument, notice that

$$\begin{array}{rcll}
2\sum_{i=0}^{n-1} i & = & \sum_{i=0}^{n-1} i + \sum_{j=0}^{n-1} j & < \text{algebra} > \\
& = & \sum_{i=0}^{n-1} i + \sum_{j=n-1}^{0} j & < \text{reverse the order of the summation} > \\
& = & \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} (n-i-1) & < \text{substituting } j = n-i-1 > \\
& = & \sum_{i=0}^{n-1} (i+n-i-1) & < \text{merge sums} > \\
& = & \sum_{i=0}^{n-1} (n-1) & < \text{algebra} > \\
& = & n(n-1) & < (n-1) \text{ is summed } n \text{ times} >.
\end{array}$$

Hence $\sum_{i=0}^{n-1} i = n(n-1)/2$.

---

**Homework 1.3.4.1** Let $n \geq 1$. Then $\sum_{i=1}^{n} i = n(n+1)/2$.

Always/Sometimes/Never

☛ SEE ANSWER

☛ DO EXERCISE ON edX

**Homework 1.3.4.2** Let $n \geq 1$. $\sum_{i=0}^{n-1} 1 = n$.

Always/Sometimes/Never

☛ SEE ANSWER

☛ DO EXERCISE ON edX

**Homework 1.3.4.3** Let $n \geq 1$ and $x \in \mathbb{R}^m$. Then

$$\sum_{i=0}^{n-1} x = \underbrace{x + x + \cdots + x}_{n \text{ times}} = nx$$

Always/Sometimes/Never

☛ SEE ANSWER

☛ DO EXERCISE ON edX

**Homework 1.3.4.4** Let $n \geq 1$. $\sum_{i=0}^{n-1} i^2 = (n-1)n(2n-1)/6$.

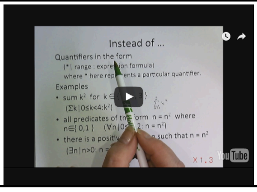
Always/Sometimes/Never

☛ SEE ANSWER

☛ DO EXERCISE ON edX

## 1.4 Quantified Expressions

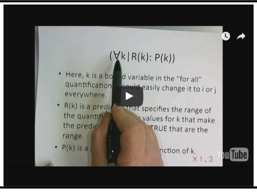
### 1.4.1 What is a quantifier ☛ to edX



☛ Watch Video on edX

☛ Watch Video on YouTube

### 1.4.2 The “for all” quantifier ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

To avoid having to explicitly list every term in the conjunction, one can instead use the “for all” quantifier:

$$P_0 \wedge P_1 \wedge \cdots \wedge P_{n-1} \quad \text{can be written as (is equivalent to)} \quad (\forall k \mid 0 \leq k < n : P_k).$$

This is read as “for all $k$ in the range $0 \leq k < n$ the expression $P_k$ holds” or, equivalently, “$P_k$ holds for all $k$ in the range $0 \leq k < n$.”

More generally, we may have a predicate that is a function of the integer variable $k$, $P(k)$, and a set of values of $k$ for which this expression evaluates to TRUE. This would be expressed as

$$(\forall k \mid R(k) : P(k)).$$

Here

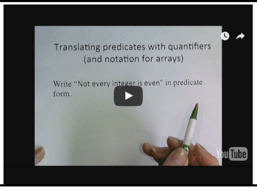
- $k$ is the *bound variable* in the “for all” quantification.
- $R(k)$ is a predicate that is a function of $k$ and specifies the *range* of the quantification. It is the values of $k$ for which $R(k)$ evaluates to TRUE that are in the range.
- $P(k)$ is the predicate that is a function of $k$.

If $S = \{k_0, k_1, k_2, \ldots\}$ is the set of all integers that satisfies $R(k)$, then

$$(\forall k \mid k \in S : P(k)) = (\forall k \mid R(k) : P(k)) = (P(k_0) \wedge P(k_1) \wedge \cdots).$$

In this course, the set denoted by $R(k)$ can be finite (e.g., $0 \leq k < 10$) or countably infinite (e.g., $0 \leq k$).

### 1.4.3 The “there exists” quantifier ☛ to edX


Translating predicates with quantifiers (and notation for arrays)

Write “Not every integer is even” in predicate form.

☛ Watch Video on edX
☛ Watch Video on YouTube

To avoid having to explicitly list every term in the disjunction, one can instead use the “there exists” quantifier:

$$P_0 \vee P_1 \vee \cdots \vee P_{n-1} \quad \text{can be written as (is equivalent to)} \quad (\exists k \mid 0 \leq k < n : P_k).$$

This is read as “there exists a $k$ in the range $0 \leq k < n$ such that the expression $P_k$ holds” or, equivalently, “$P_k$ holds for at least one $k$ in the range $0 \leq k < n$.”

More generally, in

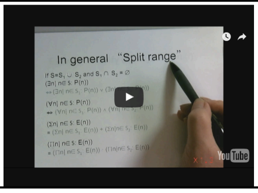
$$(\exists k \mid R(k) : P(k)).$$

- $k$ is the *bound variable* in the “there exists” quantification.
- $R(k)$ is a predicate that is a function of $k$ and specifies the *range* of the quantification. It is the values of $k$ for which $R(k)$ evaluates to TRUE that are in the range.
- $P(k)$ is the predicate that is a function of $k$.

If $S = \{k_0, k_1, k_2, \ldots\}$ is the set of all integers that satisfy $R(k)$, then

$$(\exists k \mid k \in S : P(k)) = (\exists k \mid R(k) : P(k)) = (P(k_0) \vee P(k_1) \vee \cdots).$$

### 1.4.4 Splitting the range ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Often in our discussions we will want to "split the range" of a quantifier.

**Example 1.4** Consider, an array $b$ with elements $b(0)$ through $b(n-1)$. The predicate

$$s = (\sum k \mid 0 \leq k < n : b(k))$$

expresses that variable $s$ equals the sum of the elements of $b$. Given an integer $i$ that satisfies $0 \leq i < n$, the given predicate is equivalent to

$$s = (\sum k \mid 0 \leq k < i : b(k)) + (\sum k \mid i \leq k < n : b(k)).$$

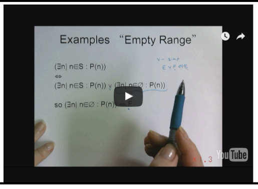
because

$$\begin{aligned}(\sum k \mid 0 \leq k < n : b(k)) &= \underbrace{b(0) + \cdots + b(i-1)}_{(\sum k \mid 0 \leq k < i : b(k))} + \underbrace{b(i) + \cdots + b(n-1)}_{(\sum k \mid i \leq k < n : b(k))} \\ &= (\sum k \mid 0 \leq k < i : b(k)) + (\sum k \mid i \leq k < n : b(k))\end{aligned}$$

Let us discuss the theory that underlies this. Partition $S$ into the subsets $S_0$ and $S_1$. Partitioning means that $S_0 \cup S_1 = S$ and $S_0 \cap S_1 = \emptyset$. Then

$$\begin{array}{rcl}(\forall k \mid k \in S : E(k)) & \Leftrightarrow & (\forall k \mid k \in S_0 : E(k)) \wedge (\forall k \mid k \in S_1 : E(k)) \\ (\exists k \mid k \in S : E(k)) & \Leftrightarrow & (\exists k \mid k \in S_0 : E(k)) \vee (\exists k \mid k \in S_1 : E(k)) \\ (\sum k \mid k \in S : E(k)) & = & (\sum k \mid k \in S_0 : E(k)) + (\sum k \mid k \in S_1 : E(k)) \\ (\prod k \mid k \in S : E(k)) & = & (\prod k \mid k \in S_0 : E(k)) \times (\prod k \mid k \in S_1 : E(k)).\end{array}$$

### 1.4.5 Quantifiers with special ranges ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

In our proofs of correctness, we will often want to split one component or term from a quantifier.

**Example 1.5** Splitting one term from a "for all" quantifier:

$$\begin{aligned}
& 1 \le k \wedge (\forall i \mid 1 \le i \le k : a(i) = b(i) + c(i)) \\
= \quad & 1 \le k \wedge (\forall i \mid 1 \le i < k : a(i) = b(i) + c(i)) \wedge \underbrace{(\forall i \mid k \le i \le k : a(i) = b(i) + c(i))}_{a(k) = b(k) + c(k)} \\
= \quad & 1 \le k \wedge (\forall i \mid 1 \le i < k : a(i) = b(i) + c(i)) \wedge a(k) = b(k) + c(k)
\end{aligned}$$

Splitting one term from a "there exists" quantifier:

$$\begin{aligned}
& 1 \le k \wedge (\exists i \mid 1 \le i \le k : a(i) = b(i) + c(i)) \\
= \quad & 1 \le k \wedge (\exists i \mid 1 \le i < k : a(i) = b(i) + c(i)) \vee \underbrace{(\exists i \mid k \le i \le k : a(i) = b(i) + c(i))}_{a(k) = b(k) + c(k)} \\
= \quad & 1 \le k \wedge (\exists i \mid 1 \le i < k : a(i) = b(i) + c(i)) \vee a(k) = b(k) + c(k)
\end{aligned}$$

Splitting one term from a summation:

$$\begin{aligned}
& 1 \le k \wedge (\textstyle\sum i \mid 1 \le i \le k : a(i)) \\
= \quad & 1 \le k \wedge (\textstyle\sum i \mid 1 \le i < k : a(i)) + \underbrace{(\textstyle\sum i \mid k \le i \le k : a(i))}_{a(k)} \\
= \quad & 1 \le k \wedge (\textstyle\sum i \mid 1 \le i < k : a(i)) + a(k)
\end{aligned}$$

What these examples illustrate is that when the range of a quantifier consists of a single point it simply evaluates to the expression (term) evaluated at that point.

In the last unit you may also have wondered what would happen if one of the two sets $S_0$ or $S_1$ is the empty set. Let us motivate this by revisiting Example 1.4:

**Example 1.6** Consider, an array $b$ with elements $b(0)$ through $b(n-1)$. The predicate

$$s = (\sum k \mid 0 \leq k < n : b(k))$$

expresses that variable $s$ equals the sum of the elements of $b$. Given an integer $i$ that satisfies $0 \leq i < n$, the given predicate is equivalent to

$$s = (\sum k \mid 0 \leq k < i : b(k)) + (\sum k \mid i \leq k < n : b(k)).$$

Let us examine the extreme cases:

$i = 0$**:** In this case, we find that

$$s = (\sum k \mid \underbrace{0 \leq k < 0}_{\text{empty!}} : b(k)) + (\sum k \mid 0 \leq k < n : b(k))$$

We notice that in this example, the sum over the empty range better be defined to equal zero!

$i = n$**:** In this case, we find that

$$s = (\sum k \mid 0 \leq k < n : b(k)) + (\sum k \mid \underbrace{n \leq k < n}_{\text{empty!}} : b(k))$$

We notice that, again, the sum over the empty range better be defined to equal zero!

Let us look at this more generally. Split $S = S_0 \cup S_1$ where $S_0 \cap S_1 = \varnothing$. Consider

$$(\sum k \mid k \in S : E(k)) = (\sum k \mid k \in S_0 : E(k)) + (\sum k \mid k \in S_1 : E(k)).$$

If $S_0 = \varnothing$, then $S_1 = S$ and

$$(\sum k \mid k \in S : E(k)) = (\sum k \mid k \in \varnothing : E(k)) + (\sum k \mid k \in S : E(k))$$

implies that

$$(\sum k \mid k \in \varnothing : E(k)) = 0.$$

Via similar reasoning one concludes that

$$\begin{array}{rcl} (\forall k \mid k \in \varnothing : P(k)) & \Leftrightarrow & T \\ (\exists k \mid k \in \varnothing : P(k)) & \Leftrightarrow & F \\ (\sum k \mid k \in \varnothing : E(k)) & = & 0 \\ (\prod k \mid k \in \varnothing : E(k)) & = & 1. \end{array}$$

### 1.4.6 Practice expressing statements as predicates ☛ to edX

**Homework 1.4.6.1** Let us consider a one dimensional array $b(1:n)$ (using Matlab notation), where $1 \leq n$. Let $j$ and $k$ be two integer variables satisfying $1 \leq j \leq k \leq n$. By $b(j:k)$ we mean the subarray of $b$ consisting of $b(j), b(j+1), \ldots b(k)$. The segment $b(j:k)$ is empty (contains no elements) if $j > k$.
Translate the following sentences into predicates.

1. All elements in the subarray $b(j:k)$ are positive.
2. No element in the subarray $b(j:k)$ is positive.
3. It is not the case that all elements in the subarray $b(j:k)$ are positive.
4. All elements in the subarray $b(j:k)$ are not positive.
5. Some element in the subarray $b(j:k)$ is positive.
6. There is an element in the subarray $b(j:k)$ that is positive.
7. At least one element in the subarray $b(j:k)$ is positive.
8. Some element in the subarray $b(j:k)$ is not positive.
9. Not all elements in the subarray $b(j:k)$ are positive.
10. It is not the case that there is an element in the subarray $b(j:k)$ that is positive.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 1.4.6.2** Translate the following sentence into a predicate: Exactly one element in the subarray $b(j:k)$ is positive.

1. $(\exists i \mid j \leq i \leq k : b(i) > 0 \wedge (\forall p \mid j \leq p \leq k \wedge p \neq i : \neg(b(p) > 0)))$
2. $(\exists i \mid j \leq i \leq k : b(i) > 0) \wedge (\forall p \mid j \leq p \leq k \wedge p \neq i : \neg(b(p) > 0))$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

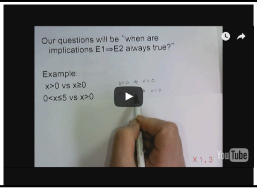
**Homework 1.4.6.3** Formalize the following English specifications. Be sure to introduce necessary restrictions.

1. Set $s$ equal to the sum of the elements of $b(j:k)$.
2. Set $M$ equal to the maximum value in $b(j:k)$.
3. Set $I$ equal to the index of a maximum value of $b(j:k)$.
4. Calculate $x$, the greatest power of 2 that is not greater than positive integer $n$.
5. Compute $c$, the number of zeroes in array $b(1:n)$.
6. Consider array of integers $b(1:n)$. Each of its subsegments $b(i:j)$ has a sum $S_{i,j} = (\sum \mid i \leq k \leq j : b(k))$. Compute $M$ equal to the maximum such sum.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 1.5 Weakening/strengthening

### 1.5.1 Weakening/strengthening laws ☛ to edX


Our questions will be "when are implications E1⇒E2 always true?"
Example:
x>0 vs x≥0
0<x≤5 vs x>0

☛ Watch Video on edX
☛ Watch Video on YouTube

When proving correctness of programs, the notion of one predicate being "stronger" or "weaker" than another predicate will play a central role. This will become clear starting in Week 2. In this unit, you will learn what it means for a predicate to be stronger or weaker and you will be equipped with some laws we call "Weakening/strenghtening" laws.

When two predicates E1 and E2 have the property that E1 $\Rightarrow$ E2, the predicate E2 is said to be *weaker* (less restrictive) than predicate E1. Equivalently, E1 is said to be *stronger* (more restrictive) than E2. Notice that this means that any predicate is simultaneously weaker and stronger than itself.

How do we most systematically show that $x \geq 3$ is weaker than $x = 5$ using what we have learned before?

$$\begin{aligned} & (x=5) \Rightarrow (x \geq 3) \\ \Leftrightarrow\ & < \text{algebra} > \\ & (x=5) \Rightarrow (x \geq 6) \vee (x=5) \vee (x=4) \vee (x=3) \end{aligned}$$

Now, at this point you may look at

$$(x=5) \Rightarrow (x \geq 6) \vee (x=5) \vee (x=4) \vee (x=3)$$

and say "Well, dah! Obviously this is TRUE". However, to show it rigorously, you have to continue the proof:

$$\begin{aligned} & (x=5) \Rightarrow (x \geq 6) \vee (x=5) \vee (x=4) \vee (x=3) \\ \Leftrightarrow\ & < \text{implication; commutivity} > \\ & \neg(x=5) \vee ((x=5) \vee (x \geq 6) \vee (x=4) \vee (x=3)) \ \Leftrightarrow < \text{associativity; excluded middle; associativity} > \\ & T \vee ((x \geq 6) \vee (x=4) \vee (x=3)) \\ \Leftrightarrow\ & < \vee\text{-simplification} > \\ & T \end{aligned}$$

At the end of the next unit, we will equip you with a few new laws of logic that allow you to essentially say "Well, dah" instead.

### 1.5.2 Weakening/strengthening exercises ☛ to edX

**Homework 1.5.2.1** For each of the following, if applicable, indicate which statement is TRUE (by examination):

1. (a) $0 \leq x \leq 10$ is weaker than $1 \leq x < 5$.
   (b) $0 \leq x \leq 10$ is stronger than $1 \leq x < 5$.
2. (a) $x = 5 \wedge y = 4$ is weaker than $y = 4$.
   (b) $x = 5 \wedge y = 4$ is stronger than $y = 4$.
3. (a) $x \leq 5 \vee y = 3$ is weaker than $x = 5 \wedge y = 4$.
   (b) $x \leq 5 \vee y = 3$ is stronger than $x = 5 \wedge y = 4$.
4. (a) $T$ is weaker than $F$.
   (b) $T$ is stronger than $F$.
5. (a) $(\forall i | 5 \leq i \leq 10 : b(i+1) < b(i))$ is weaker than $(\forall i | 7 \leq i \leq 10 : b(i+1) < b(i))$.
   (b) $(\forall i | 5 \leq i \leq 10 : b(i+1) < b(i))$ is stronger than $(\forall i | 7 \leq i \leq 10 : b(i+1) < b(i))$.
6. (a) $x \leq 1$ is weaker than $x \geq 5$.
   (b) $x \leq 1$ is stronger than $x \geq 5$.
7. (a) $x \leq 4$ is weaker than $5 > x$.
   (b) $x \leq 4$ is stronger than $5 > x$.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

There are a few situations that we will encounter later in the course where understanding how specific predicates are stronger than a slightly different predicate will be key to accurate reasoning. Let us first discuss these informally after which we leave the formal proof as an exercise.

- If the expression $E1 \wedge E2$ is *true* then obviously the expression E1 is *true*. So, $E1 \wedge E2$ is a stronger predicate (is more restrictive) than E1.
- If the expression E1 is *true* then obviously the expression $E1 \vee E3$ is *true*. So, E1 is a stronger predicate (is more restrictive) than $E1 \vee E3$.
- If the expression $E1 \wedge E2$ is *true* then obviously the expression E1 is *true* and hence $E1 \vee E3$ is true. So, $E1 \wedge E2$ is a stronger predicate (is more restrictive) than $E1 \vee E3$.

While this is obvious, one really should prove it:

**Homework 1.5.2.2** Use the Basic Equivalences to prove the following. (Do NOT use the weakening/strengthening laws given in Figure 1.2, which we will discuss later.)

1. $E1 \wedge E2 \Rightarrow E1$
2. $E1 \Rightarrow E1 \vee E3$
3. $E1 \wedge E2 \Rightarrow E1 \vee E3$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

| | | | | | |
|---|---|---|---|---|---|
| Weakening/ Strengthening: | $((E1 \wedge E2)$ | $\Rightarrow$ | $E1)$ | $\Leftrightarrow$ | $T$ |
| | $(E1$ | $\Rightarrow$ | $(E1 \vee E3))$ | $\Leftrightarrow$ | $T$ |
| | $((E1 \wedge E2)$ | $\Rightarrow$ | $(E1 \vee E3))$ | $\Leftrightarrow$ | $T$ |

Figure 1.2: Weaking/strengthening laws.

**Homework 1.5.2.3** Use the Basic Equivalences and/or the results from Homework 1.5.2.2 to prove that

$$E1 \wedge E2 \Rightarrow (E1 \vee E3) \wedge E2.$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

The insights from these last two homeworks will become powerful tools as we prove programs correct. Together we will call them *Weakening/Strengthening Laws*. They are summarized in Figure 1.2. These form a second set of useful tautologies.

**Homework 1.5.2.4** In Figure 1.2 we present three Weakening/Strengening Laws. This exercise shows that if you only decide to remember one, it should be the last one.

1. Show that $(E1 \wedge E2) \Rightarrow E1$ is a special case of $(E1 \wedge E2) \Rightarrow (E1 \vee E3)$.
2. Show that $E1 \Rightarrow (E1 \vee E3)$ is a special case of $(E1 \wedge E2) \Rightarrow (E1 \vee E3)$.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

What you will find later is that it is $(E1 \wedge E2) \Rightarrow (E1 \vee E3)$ becomes our tool of choice in many proofs.

**Homework 1.5.2.5** For each of the following predicates pairs from Homework 1.5.2.1 use an equivalence style proof, the Basic Logic Equivalences, and the Weakening/strengthening laws to prove which predicate is weaker:

1. $0 \le x \le 10$ and $1 \le x < 5$.
2. $x = 5 \wedge y = 4$ and $y = 4$.
3. $x \le 5 \vee y = 3$ and $x = 5 \wedge y = 4$.
4. $T$ and $F$.
5. $(\forall i | 5 \le i \le 10 : b(i+1) < b(i))$ and $(\forall i | 7 \le i \le 10 : b(i+1) < b(i))$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 1.6 Enrichment

### 1.6.1 The Humble Programmer – Edsger W. Dijkstra ☛ to edX

Edsger W. Dijkstra was one of the most influential computer scientists. His pioneering and visionary work greatly influenced the material that underlies this course. You should start your journey by reading his ACM Turing Award acceptance speech.

☛ ACM Turing Lecture 1972: “The Humble Programmer” by Edsger W. Dijkstra

A partial tape recording of the lecture can be found on

☛ YouTube

Unfortunately, it seems like some tapes were overwritten with music...

Notice that some find the title of Dijkstra’s Turing Award acceptance speech amusing. Here is an interesting quote from Alan Kay:

> “I don’t know how many of you have ever met Dijkstra, but you probably know that arrogance in computer science is measured in nano-Dijkstras.”

### 1.6.2 Typesetting proofs with LaTeX ☛ to edX

$p \Rightarrow (q \Rightarrow p)$
$\Leftrightarrow$ < implication >
$\neg p \vee (q \Rightarrow p)$
$\Leftrightarrow$ < implication >
$\neg p \vee (\neg q \vee p)$

☛ Watch Video on edX
☛ Watch Video on YouTube

### 1.6.3 More on logic ☛ to edX

We have tried to give just enough logic background for the course to be self-contained. Here we list some resources if you want to go beyond.

Some places where you can learn more about logic:

#### FREE!

Dr. Elaine Rich and Prof. Alan Cline created an online introductory course on logic for use at The University of Texas at Austin titled “Fundamentals of Reasoning”. They have ported this course to edX ”Edge”, which is a platform where unofficial courses are offered. They have graciously made it possible for you to try the first few chapters of this course, which should give you a solid background in logic. You can try this course by going to

☛ https://edge.edx.org/courses/course-v1:UT+101+2017/about

(You will have to register for edge.edx.org and sign in.)

#### Some books to try! (Not free)

- David Gries, The Science of Programming (Monographs in Computer Science), Springer, 1987.
- David Gries and Fred B. Schneider, A Logical Approach to Discrete Math (Texts and Monographs in Computer Science), Springer, 1994.

## 1.7 Wrapup

### 1.7.1 Additional exercises ☛ to edX

There are no additional exercises this week. Skip and go on!

### 1.7.2 Summary ☛ to edX

#### Boolean operators

- *not* (*negation*, $\neg$);
- *and* (*conjunction*, $\wedge$);
- *or* (*disjunction*, $\vee$);
- *implies* ( *implication*, $\Rightarrow$);
- *is equivalent to* ( *equivalence*, $\Leftrightarrow$).

For a list of logic symbols that includes LaTeX symbols, you may want to consult Wikipedia's ☛ List of logic symbols entry (`https://en.wikipedia.org/wiki/List_of_logic_symbols`).

| | | not | and | or | implies | equivalent |
|---|---|---|---|---|---|---|
| $p$ | $q$ | $\neg p$ | $p \wedge q$ | p$\vee q$ | $p \Rightarrow q$ | $p \Leftrightarrow q$ |
| $T$ | $T$ | $F$ | $T$ | $T$ | $T$ | $T$ |
| $T$ | $F$ | $F$ | $F$ | $T$ | $F$ | $F$ |
| $F$ | $T$ | $T$ | $F$ | $T$ | $T$ | $F$ |
| $F$ | $F$ | $T$ | $F$ | $F$ | $T$ | $T$ |

#### Precedence of Boolean operators

- First negate: $\neg p \wedge q$ is the same as $(\neg p) \wedge q$.
- Second, evaluate $\wedge$: $\neg p \wedge q \Rightarrow r$ is the same as $((\neg p) \wedge q) \Rightarrow r$.
- Third, evaluate $\vee$: $\neg p \wedge q \vee r \Rightarrow s$ is the same as $(((\neg p) \wedge q) \vee r) \Rightarrow s$.
- Fourth, evaluate $\Rightarrow$: $t \Leftrightarrow \neg p \wedge q \vee r \Rightarrow s$ is the same as $t \Leftrightarrow ((((\neg p) \wedge q) \vee r) \Rightarrow s)$.
- Last evaluate $\Leftrightarrow$.

#### Predicates, tautologies, contradications

A **predicate** is a logical statement that may be TRUE or FALSE depending on the values of the variables that appear in the statement.

A **tautology** is a predicate that evaluates to TRUE regardless of the choice of the variables in the predicate.

A **contradiction** is a predicate that evaluates to FALSE regardless of the choice of the variables in the predicate.

#### Basic Equivalences

The basic equivalences can be found in Figure 1.1 and by visiting the "Laws of Logic" tab in the LAFF-On edX course navigation bar.

## The Principle of Mathematical Induction

The **Principle of Mathematical Induction** (weak induction) says that *if* one can show that

- (Base case) a property holds for $k = k_b$; *and*
- (Inductive step) if it holds for $k = K$, where $K \geq k_b$, then it is also holds for $k = K + 1$,

*then* one can conclude that the property holds for all integers $k \geq k_b$. Often $k_b = 0$ or $k_b = 1$.

## Quantifiers

**For all ...** $(\forall i \mid R(i) : P(i))$ stands for "For all $i$ that satisfy the predicate $R(i)$ the predicate $P(i)$ holds."

**There exists ...** $(\exists i \mid R(i) : P(i))$ stands for "There exists an $i$ that satisfies the predicate $R(i)$ for which the predicate $P(i)$ holds."

**Sum ...** $(\sum i \mid R(i) : E(i))$ stands for "Sum expressions $E(i)$ for all $i$ that satisfy the predicate $R(i)$." More traditionally this is denoted by $\sum_{R(i)} E(i)$.

**Product ...** $(\prod i \mid R(i) : E(i))$ stands for "Multiply expressions $E(i)$ for all $i$ that satisfy the predicate $R(i)$." More traditionally this is denoted by $\prod_{R(i)} E(i)$.

## Splitting the range

Partition $S$ into the subsets $S_0$ and $S_1$. Partitioning means that $S_0 \cup S_1 = S$ and $S_0 \cap S_1 = \emptyset$. Then

$$\begin{array}{rcl}
(\forall k \mid k \in S : E(k)) & \Leftrightarrow & (\forall k \mid k \in S_0 : E(k)) \wedge (\forall k \mid k \in S_1 : E(k)) \\
(\exists k \mid k \in S : E(k)) & \Leftrightarrow & (\exists k \mid k \in S_0 : E(k)) \vee (\exists k \mid k \in S_1 : E(k)) \\
(\sum k \mid k \in S : E(k)) & = & (\sum k \mid k \in S_0 : E(k)) + (\sum k \mid k \in S_1 : E(k)) \\
(\prod k \mid k \in S : E(k)) & = & (\prod k \mid k \in S_0 : E(k)) \times (\prod k \mid k \in S_1 : E(k))
\end{array}$$

## One point rule

$$\begin{array}{rcl}
(\forall k \mid k \in \{i\} : P(k)) & \Leftrightarrow & P(i) \\
(\exists k \mid k \in \{i\} : P(k)) & \Leftrightarrow & P(i) \\
(\sum k \mid k \in \{i\} : E(k)) & = & E(i) \\
(\prod k \mid k \in \{i\} : E(k)) & = & E(i).
\end{array}$$

## Empty range

$$\begin{array}{rcl}
(\forall k \mid k \in \emptyset : P(k)) & \Leftrightarrow & T \\
(\exists k \mid k \in \emptyset : P(k)) & \Leftrightarrow & F \\
(\sum k \mid k \in \emptyset : E(k)) & = & 0 \\
(\prod k \mid k \in \emptyset : E(k)) & = & 1.
\end{array}$$

## Weakening/strengthening laws

The basic equivalences can be found in Figure 1.2 and by visiting the "Laws of Logic" tab in the LAFF-On edX course navigation bar.

Week 2

# Proving Programs Correct

## 2.1 Opening Remarks

### 2.1.1 Launch ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

In Week 1, you equipped yourself with a number of tools that you will now employ to prove program segments correct.

In this launch, we want you to stretch yourself a bit and puzzle out how to add assertions about the state of variables, in the form of predicates, to a code segment. Each assertion says something about what you expect to be true at a given point in the code. With this, you can then argue about the correctness of the code.

Let us consider the simple loop given in *pseudocode*, that sums the elements of an array, $b$, that has $n = \text{size}(b)$ elements, where the indexing into the array starts at zero, meaning that the elements of $b$ are referenced as $b(0)$ through $b(n-1)$. The result is accumulated in variable $s$.

```
$s := 0$
$k := 0$
**while** $k < n$ **do**
  $s := s + b(k)$
  $k := k + 1$
**endwhile**
```

(Let's not get distracted by the fact that this loop would often be written as a "for" loop. It is much easier to reason about the correctness of a **while** loop.)

How might we reason about the correctness of such a code segment? We might start by identifying a predicate that describes what must be TRUE about the variables before the code segment. This is called the *precondition*. In the below annotated code we assume that $n$ equals the size of array $b$ and the precondition asserts that before the code segment starts this size is nonnegative. We also want to add a predicate at the end of the code segment that describes the desired state of the variables upon completion. This is called the *postcondition*, and in the below annotated code it says that upon completion $s$ equals the sum of the elements of $b$:

```
{ $0 \leq n$ }
$s := 0$
$k := 0$
while $k < n$ do
    $s := s + b(k)$
    $k := k + 1$
endwhile
{ $s := (\sum i \mid 0 \leq i < n : b(i))$ }
```

The code segment is correct if, starting in a state that satisfies the precondition, the code segment completes (in a finite amount of time to avoid, for example, infinite loops) in a state that satisfies the postcondition.

To reason about the correctness of the code segment, we now ask you to insert more assertions in form of predicates.

**Homework 2.1.1.1** Consider again the algorithm that sums the elements of array $b$, now given in Figure 2.1. Place the following assertions in the correct place (the blank boxes) in the algorithm:

1. $\{ s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge k < n \}$
2. $\{ s = 0 \wedge 0 = k \leq n \}$
3. $\{ s = 0 \wedge 0 \leq n \}$
4. $\{ s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge \neg(k < n) \}$
5. $\{ s = (\sum i \mid 0 \leq i < k+1 : b(i)) \wedge 0 \leq k < n \}$
6. $\{ s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \}$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

In the following video, Robert misspeaks at 1:00. "...is k less than n. If not, then we enter the loop", should be "...is k less than n. If so, then we enter the loop".

☛ Watch Video on edX
☛ Watch Video on YouTube

What we notice is how assertions about the state of (the values in) the variables are given as predicates. Each assertion indicates that the values of the variables at that point in the code should be such that the predicate evaluates to TRUE. If a predicate does not evaluate to TRUE, then there is something unexpected because either the code is wrong or the predicate does not correctly describe what should be TRUE at that point in the code.

Since there is at least one assertion after each command, one can reason about whether the program is correct by reasoning about the correctness of each individual command along the way.

```
{ $0 \leq n$ }
$s := 0$

$k := 0$

{ $s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n$ }
**while** $k < n$ **do**

    $s := s + b(k)$

    $k := k + 1$

**endwhile**

{ $s = (\sum i \mid 0 \leq i < n : b(i))$ }
```

Figure 2.1: Partially annotated algorithm for computing the sum of the elements of array *b*.

**Homework 2.1.1.2** Take the solution for the last homework and use it to convince someone (possibly yourself) that the code segment is correct.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

Convincing someone is not the same as proving the code segment correct. How often have you convinced yourself that your code is correct only to later discover a bug? In this week, you will learn how to formally prove the correctness of a program segment. This is a first step towards systematically deriving a program to be correct hand-in-hand with its proof of correctness, which will be the topic of Week 3.

### 2.1.2 Outline Week 2 ☛ to edX

### 2.1.3 What you will learn ☛ to edX

Today, common practice is to show a program correct through testing. The issue with this is that while testing can uncover errors, it cannot prove a program correct. After all, you may not have tested the cases for which the program fails. This week equips you with the tools to provide definitive proofs of correctness. This is a first step towards systematic derivation of correct programs.

Proving programs correct is foundational. It was studied by many, including several Turing Award winners. In this week, we share our understanding of their work.

Upon completion of this week, you should be able to

- Annotate a program with assertions.
- Create and apply Hoare triples to reason about correctness of short program segments involving skip, abort, and assignments.
- Evaluate the weakest precondition.
- Prove the correctness of a Hoare triple using the weakest precondition.
- Prove the correctness of various assignment commands.
- Prove correctness of an **if** command via the If Theorem.
- Prove correctness of an iterative command via the While Theorem.

## 2.2 Tools for Reasoning About Correctness

### 2.2.1 The Hoare triple ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

The launch for this week motivates the notion of a Hoare triple, named after Sir Charles Antony Richard Hoare (Sir Tony Hoare): Given predicates $Q$ and $R$ and command $S$, the Hoare triple

$$\{Q\}S\{R\}$$

*holds* (evaluates to TRUE) if the command $S$ when started in a state for which $Q$ evaluates to TRUE completes *in a finite amount of time* in a state for which $R$ evaluates to TRUE. For this triple, $Q$ is the precondition and $R$ is the postcondition.

In other words, if $Q$ and $R$ describe the states in which we expect to be before and after the command, it only executes as expected if the triple holds. This will give us a formal vehicle for reasoning about the correctness of commands in a code segment.

We have encountered a number of Hoare triples in the launch for this week, when we annotated code with assertions. There we saw

| |
|---|
| $\{\ Q \qquad\qquad\qquad \}$ |
| $S$ |
| $\{\ R \qquad\qquad\qquad \}$ |

or just

$$\begin{array}{l}\{Q\}\\ S\\ \{R\}\end{array}$$

or, if space on the page is an issue, $\{Q\}S\{R\}$.

### 2.2.2 The weakest precondition ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

While the Hoare triple formalizes how to assert that a code segment is correct, it is the definition of a *weakest precondition* that allows one to *prove* that a code segment is correct. Importantly, it also facilitates *goal-oriented programming*, as we will see starting in Week 3.

Before doing the following homework, you may want to review Section 1.5.1.

**Homework 2.2.2.1** By examination, decide whether the following Hoare triples hold (evaluate to TRUE) and which of the predicates are stronger/weaker:

1. $\{x > 4\} y := x + 1 \{y > 5\}$ TRUE/FALSE
2. $\{x = 10\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x = 10$ is stronger than $x > 4$ TRUE/FALSE
3. $\{x = 5\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is weaker than $x = 5$ TRUE/FALSE
4. $\{x \geq 5\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is at least as weak as $x \geq 5$ TRUE/FALSE
5. $\{x = 4\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is weaker than $x = 4$ TRUE/FALSE
6. $\{x \geq 4\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is weaker than $x \geq 4$ TRUE/FALSE
7. $\{x > 4\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is at least as weak as $x > 4$ TRUE/FALSE

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 2.2.2.2** What do you notice about the relationship between the preconditions, $P$, that make the Hoare triple $\{P\} y := x + 1 \{y > 5\}$ TRUE and the predicate $x > 4$? Choose the correct answer:

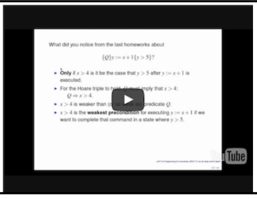
a) Of all $P$ for which the Hoare triple holds (evaluates to TRUE), $x > 4$ is the weakest.

b) Of all $P$ for which the Hoare triple holds (evaluates to TRUE), $x > 4$ is the strongest.

c) Of all $P$ for which the Hoare triple holds, $y > 5$ is the weakest.

d) No obvious relation.

☛ SEE ANSWER
☛ DO EXERCISE ON edX



What did you notice from the last homeworks about

$$\{Q\} y := x + 1 \{y > 5\}?$$

- **Only** if $x > 4$ is it be the case that $y > 5$ after $y := x + 1$ is executed.
- For the Hoare triple to hold, $Q$ must imply that $x > 4$:
  $Q \Rightarrow x > 4$.
- $x > 4$ is weaker than (or as weak as) predicate $Q$.
- $x > 4$ is the **weakest precondition** for executing $y := x + 1$ if we want to complete that command in a state where $y > 5$.

☛ Watch Video on edX
☛ Watch Video on YouTube

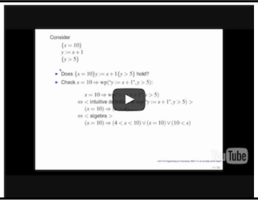
This motivates the fundamental question

"What is the set of *all* states such that the execution of $y := x + 1$ results in a state where $y > 5$?"

or, equivalently,

"What predicate describes the set of *all* states such that the execution of the command $y := x + 1$ in a state described by that predicate will leave us in a state where $y > 5$ (after a finite amount of time)?"

### 2.2.3 Proving the correctness of a Hoare triple ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Let us look at this more generally. Consider the command $S$ and let us assume that we would like to check whether the code segment

$\{Q\}$
S
$\{R\}$

is correct. Now, what if we had this function, $\mathrm{wp}(\text{“}S\text{”}, R)$, that returns a predicate that describes *all* states for which executing $S$ leaves the variables (in a finite amount of time) in a state where $R$ is true?

**Definition 2.1** *Given command S and postcondition R, the weakest precondition (which describes the set of all states for which execution of command S completes in a finite amount of time is a state described by R) is given by* $\mathrm{wp}(\text{“}S\text{”}, R)$.

We then know that

$\{Q\}$
$S$
$\{R\}$

holds (is correct) if and only if

$\{Q\}$
$\{\mathrm{wp}(\text{“}S\text{”}, R)\}$
$S$
$\{R\}$

is correct. Why? Because only if we are in a state where $\mathrm{wp}(\text{“}S\text{”}, R)$ holds will $S$ complete (in a finite amount of time) in a state where $R$ is TRUE.

What does this mean? If $Q$ is true, it must imply that $\mathrm{wp}(\text{“}S\text{”}, R)$ is true. Thus, what this in turn means is that the code segment $\{Q\}S\{R\}$ is correct if and only if

$$Q \Rightarrow \mathrm{wp}(\text{“}S\text{”}, R).$$

In other words, $\mathrm{wp}(\text{“}S\text{”}, R)$ must be *at least as weak as* the predicate $Q$. This function that transforms $R$ into a new predicate is known as the *weakest precondition* function.

Notice

- Any $Q$ such that $\{Q\}S\{R\}$ is TRUE is a precondition so that command $S$ completes in a finite amount of time in a state described by predicate $R$.
- If $P$ satisfies $Q \Rightarrow P$ then $P$ is at least as weak as $Q$: $P$ is *no more restrictive* than predicate $Q$.
- The *weakest* (*least restrictive*) predicate $P$ such that $\{P\}S\{R\}$ is TRUE describes the set of *all* states in which statement $S$ can be executed so that it completes, in a finite amout of time, in a state where $R$ is TRUE.
- The wp is the function that takes the command $S$ and the postcondition $R$ as inputs, and returns a predicate that describes the set of all states for which executing $S$ completes (in a finite amount of time) in a state that satisfies the postcondition $R$.

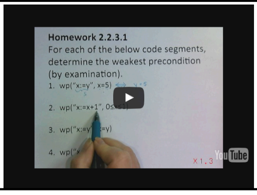
It will take a bit of practice to fully understand and appreciate this.

**Homework 2.2.3.1** For each of the below code segments, determine the weakest precondition (by examination):

1. $\text{wp}(\text{“}x := y\text{”}, x = 5) =$
2. $\text{wp}(\text{“}x := x+1\text{”}, 0 \le x \le 1) =$
3. $\text{wp}(\text{“}x := y\text{”}, x = y) =$
4. $\text{wp}(\text{“}x := 4\text{”}, x = 5) =$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

In the next subsections, we use wp to precisely define commands in our pseudo language.

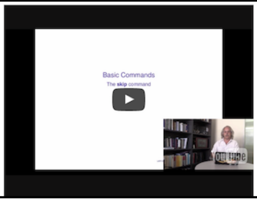

Homework 2.2.3.1
For each of the below code segments, determine the weakest precondition (by examination).

☛ Watch Video on edX
☛ Watch Video on YouTube

## 2.3 Basic Commands

In this section, we look at some simple commands that we encounter in our pseudo code and define the weakest precondition for these.

### 2.3.1 The skip command ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 2.3.1.1** Consider the **skip** command, which simply doesn't do anything:

$\{Q : ?\}$
**skip**
$\{R : x > 4\}$

From what state $Q$ will the command **skip** finish (in a finite amount of time) in a state where $x > 4$ is TRUE? In other words,

$\text{wp}(\text{“}\textbf{skip}\text{”}, x > 4) =$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

$\{wp(\text{“}\textbf{skip}\text{”}, x > 4) = \quad \}$
**skip**
$\{x > 4\}$

Give the weakest predicate [illegible] command **skip** will finish (in a finite amount of time) in a [illegible] 4 is TRUE.

$wp(\text{“}\textbf{skip}\text{”}, x > 4) =$

☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 2.3.1.2** Building on the intuition from the last homework, give $wp(\text{“}\textbf{skip}\text{”}, R) =$ for an arbitrary post-condition $R$.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

$\{wp(\text{“}S\text{”}, R) = \quad \}$
**skip**
$\{R\}$

Give the weakest predicate such that the command **skip** will finish (in a finite amount of time) in a state where $R$ is TRUE.

$wp(\text{“}\textbf{skip}\text{”}, R) =$

☛ Watch Video on edX
☛ Watch Video on YouTube

### 2.3.2 The abort command ☛ to edX

Specification of **abort** command

- The **abort** command stops the execution of the program.

☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 2.3.2.1** Consider the **abort** command, which aborts (which means execution does not reach the point in the program after the **abort** command).

$\{Q : ?\}$
**abort**
$\{R : x > 4\}$

From what state $Q$ will the command **abort** finish (in a finite amount of time) in a state where $x > 4$ is $T$? In other words, evaluate

$$wp(\text{“}\textbf{abort}\text{”}, x > 4) =$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

$\{wp(\text{“}\textbf{abort}\text{”}, x > 4) = F\}$
**abort**
$\{x > 4\}$

Give the weakest predicate such that the command **abort** will finish (in a finite amount of time) in a state where $x > 4$ is TRUE.

$wp(\text{“}\textbf{abort}\text{”}, x > 4) = F$

☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 2.3.2.2** Building on the intuition from the last homework, evaluate

$$wp(\text{“}\mathbf{abort}\text{”}, R) =$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

$\{\text{wp}(\text{“}\mathbf{abort}\text{”}, R) = F\}$
**abort**
$\{R\}$

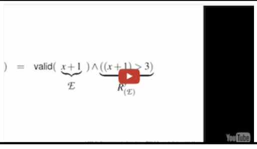
Give the weakest predicate such that the command **abort** will finish (in a finite amount of time) in a state where $R$ is TRUE.

$\text{wp}(\text{“}\mathbf{abort}\text{”}, R) = F$

- What states satisfy that $F$ is TRUE? No states.
- For no states does the **abort** command finish (in a finite amount of time) in a state where $R$ is TRUE (because it never reaches the point in the program immediately after the **abort** command).

☛ Watch Video on edX
☛ Watch Video on YouTube

## 2.3.3 Assignment to a simple variable ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Consider the code segment

$$\{Q : x = 5\}$$
$$S : y := x + 1$$
$$\{R : y > 3\}$$

It is easy to informally argue that if $x = 5$ then then the assignment $y := x + 1$ leaves one in a state where $y > 3$.

To be able to *prove* this systematically, we need to be able to compute wp(“$y := x + 1$”, $y > 3$), which answers the question of how to describe *all* states such that executing $y := x + 1$ leaves us in a state where $y > 3$ holds. More importantly, we need to expose a systematic way of taking an arbitrary postcondition $R$, and transforming it into the weakest precondition for an arbitrary assignment.

Going back to our example, notice that after the assignment the predicate $R : y > 3$ holds. The expression $x + 1$ was just assigned to $y$. Thus, only if $(x + 1) > 3$ before the assignment will the assignment leave the program in a state where $y > 3$.

This example leads us to the following definitions:

- The predicate valid($\mathcal{E}$) that returns TRUE if and only if the expression $\mathcal{E}$ is a valid expression; and
- The predicate $R^x_{(\mathcal{E})}$ that equals the predicate $R$ with all *free* occurrences of variable $x$ replaced by the expression $(\mathcal{E})$. A free occurrence of a variable in a predicate is any occurrence of that variable that is not a variable for a quantifier. In other words, a free occurrence of a variable in a predicate is any occurrence of that variable that is not a bound (dummy) variable for a quantifier. The parentheses make sure that the order of operations is properly preserved.

With this, the weakest precondition for assignment of the expression $\mathcal{E}$ to a variable $x$ is given by

$$\text{wp}(\text{“}x := \mathcal{E}\text{”}, R) = \text{valid}(\mathcal{E}) \wedge R^x_{(\mathcal{E})}.$$

Thus, in our example,

$$\text{wp}(\text{“}\underbrace{y := x+1}_{y := \mathcal{E}}\text{”}, \underbrace{y > 3}_{R}) \;=\; \text{valid}(\underbrace{x+1}_{\mathcal{E}}) \wedge \underbrace{((x+1) > 3)}_{R^y_{(\mathcal{E})}}$$

$$
\begin{aligned}
&= T \wedge (x > 2)\\
&= (x > 2).
\end{aligned}
$$

Often, we will skip including valid$(\mathcal{E})$ when $\mathcal{E}$ is obviously a valid expression.

For the exercises in this section, let us revisit the example from the launch and prove the various assignments correct:

**Example 2.2** *Prove the following code segment correct:*

| |
|---|
| $\{\ Q : 0 \leq n\ \}$ |
| $S : s := 0$ |
| $\{\ R : s = 0 \wedge 0 \leq n\ \}$ |

---

**Proof:** We need to prove that $Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)$:

$$
\begin{aligned}
& Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)\\
\Leftrightarrow\ & < \text{Instantiate } S \text{ and } R >\\
& Q \Rightarrow \text{wp}(\text{“}s := 0\text{”}, s = 0 \wedge 0 \leq n)\\
\Leftrightarrow\ & < \text{Definition of wp}(:=) >\\
& Q \Rightarrow (s = 0 \wedge 0 \leq n)^{s}_{(0)}\\
\Leftrightarrow\ & < \text{Instantiate } Q\text{; definition of } R^{s}_{(\mathcal{E})} >\\
& (0 \leq n) \Rightarrow (0 = 0 \wedge 0 \leq n)\\
\Leftrightarrow\ & < \text{identity; commutativity; } \wedge\text{-simplification; } \Rightarrow\text{-simplification} >\\
& T
\end{aligned}
$$

where we skip valid$(\mathcal{E})$ since 0 is obviously a valid expression.

---

In our answers, we wait as long as possible with instantiating $Q$, to save ourselves from having to repeatedly write $0 \leq n$ in each step. This can save you a lot of time, effort, and opportunities for accidently introducing a “typo”.

Another way of organizing the thought process and proof for the last example is to insert wp(“$S$”$, R$) where it should hold in the code segment:

| |
|---|
| $\{\ Q : 0 \leq n\ \}$ |
| $\{\ \text{wp}(\text{“}S\text{”}, R) : (0 = 0 \wedge 0 \leq n)\ \}$ |
| $S : s := 0$ |
| $\{\ R : s = 0 \wedge 0 \leq n\ \}$ |

from which we then conclude that $Q \Rightarrow (0 = 0 \wedge 0 \leq n)$ must be shown to be TRUE since $Q$ describes what initially must be TRUE and wp(“$S$”$, R$) : $(0 = 0 \wedge 0 \leq n)$ describes what must be TRUE if command $S$ is to leave the variables in a state where $R$ is TRUE.

**Homework 2.3.3.1** Prove the following code segment correct:

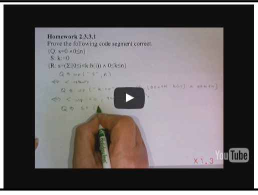
| |
|---|
| $\{\ Q : s = 0 \wedge 0 \le n\ \}$ |
| $S : k := 0$ |
| $\{\ R : s = (\sum i \mid 0 \le i < k : b(i)) \wedge 0 \le k \le n\ \}$ |

Answer We need to prove that $Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)$:

$$
\begin{array}{ll}
 & Q \Rightarrow \text{wp}(\text{“}S\text{”}, R) \\
\Leftrightarrow & < \text{instantiate } S \text{ and } R > \\
 & Q \Rightarrow \text{wp}(\text{“}k := 0\text{”}, s = (\sum i \mid 0 \le i < k : b(i)) \wedge 0 \le k \le n) \\
\Leftrightarrow & < \text{definition of wp}(:=) > \\
 & Q \Rightarrow (s = (\sum i \mid 0 \le i < k : b(i)) \wedge 0 \le k \le n)^k_{(0)} \\
\Leftrightarrow & < \text{definition of } R^k_{(\mathcal{E})} > \\
 & Q \Rightarrow (s = (\sum i \mid 0 \le i < 0 : b(i)) \wedge 0 \le 0 \le n) \\
\Leftrightarrow & < \text{instantiate } Q\text{; sum over empty range; algebra} > \\
 & (s = 0 \wedge 0 \le n) \Rightarrow (s = 0 \wedge 0 \le n) \\
\Leftrightarrow & < \Rightarrow\text{-simplification} > \\
 & T
\end{array}
$$

where we skip valid$(\mathcal{E})$ since it is obviously a valid expression.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

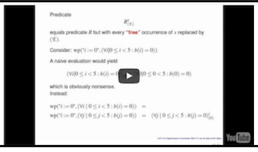


☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 2.3.3.2** Prove the following code segment correct:

| |
|---|
| $\{\ Q : s = (\sum i \mid 0 \le i < k : b(i)) \wedge 0 \le k \le n \wedge k < n\ \}$ |
| $S : s := s + b(k)$ |
| $\{\ R : s = (\sum i \mid 0 \le i < k+1 : b(i)) \wedge 0 \le k < n\ \}$ |

☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Now, consider the following problem:

$$\text{wp}(\text{“}i := 0\text{”}, (\forall i | 0 \leq i < 5 : b(i) = 0))$$

A naive evaluation would yield

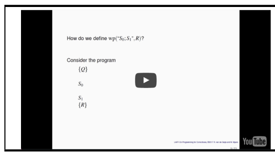
$$(\forall i | 0 \leq i < 5 : b(i) = 0)^{i}_{(0)} = (\forall 0 | 0 \leq 0 < 5 : b(0) = 0)$$

which is obviously nonsense. The problem here is that in the predicate variabe $i$ is a “bound variable” (dummy variable) of the quantifier. Since it is a bound variable, it can be replaced by any other variable that does not occur in the expression. If one replaces it with, for example, $j$, then

$$\begin{aligned} \text{wp}(\text{“}i := 0\text{”}, (\forall i \mid 0 \leq i < 5 : b(i) = 0)) &= (\forall j \mid 0 \leq j < 5 : b(j) = 0)^{i}_{(0)} \\ &= (\forall j \mid 0 \leq j < 5 : b(j) = 0), \end{aligned}$$

which does makes sense. The point: If a variable that is bound to a quantification in $R$ appears in the assignment, then the bound variable should be replaced with something that does not appear in the assignment before performing the textual substitution.

### 2.3.4 Composition ☛ to edX



How do we define wp(“$S_0;S_1$”, $R$)?

Consider the program

$\{Q\}$
$S_0$
$S_1$
$\{R\}$

LAFF-On Programming for Correctness, ©2017 R. van de Geijn and M. Myers

☛ Watch Video on edX
☛ Watch Video on YouTube

Obviously, a program that consists of a single command, $S$, is not a very interesting program, especially since the only commands we have introduced so far are **skip**, **abort**, and simple assignment. We will want to compose multiple commands.

Consider the program that consists of two arbitrary commands

$S_0$
$S_1$

which we can also write more compactly as $S_0;S_1$, using the semi-colon to separate the commands being composed. We would like to determine

$$\text{wp}(\text{“}S_0;S_1\text{”}, R).$$

Let us think about this, working backwards. The states from which executing $S_1$ leaves one in a state where $R$ is TRUE is given by wp(“$S_1$”, $R$). Thus, after executing $S_0$ one must be in a state where wp(“$S_1$”, $R$) is TRUE:

$\{Q\}$
$S_0$
$\{\text{wp}(\text{“}S_1\text{”}, R)\}$
$S_1$
$\{R\}$

The predicate that describes the states from which executing $S_0$ leaves one in a state where wp(“$S_1$”, $R$) is TRUE is described by wp(“$S_0$”, wp(“$S_1$”, $R$)). In other words, for the composed command $S_0;S_1$ to complete (in a finite amount of time) in a state where $R$ is TRUE, one must start in a state, $P$, that implies that wp(“$S_0$”, wp(“$S_1$”, $R$)) is TRUE:

$\{Q\}$
$\{\text{wp}(\text{“}S_0\text{”}, \text{wp}(\text{“}S_1\text{”}, R))\}$
$S_0$
$\{\text{wp}(\text{“}S_1\text{”}, R)\}$
$S_1$
$\{R\}$

We conclude that

$$\text{wp}(\text{“}S_0;S_1\text{”}, R) = \text{wp}(\text{“}S_0\text{”}, \text{wp}(\text{“}S_1\text{”}, R)).$$

Obviously, this generalizes to composition of $n$ commands:

$$\begin{array}{rcl} \text{wp}(\text{“}S_0;S_1;\cdots;S_{n-1}\text{”}, R) & = & \text{wp}(\text{“}S_0\text{”}, \text{wp}(\text{“}S_1;\cdots;S_{n-1}\text{”}, R)) \\ & \vdots & \quad\vdots \\ & = & \text{wp}(\text{“}S_0\text{”}, \text{wp}(\text{“}S_1\text{”}, \text{wp}(\cdots, \text{wp}(\text{“}S_{n-1}\text{”}, R)\cdots))). \end{array}$$

**Homework 2.3.4.1** Compute

1. $\text{wp}(\text{“}i := i-1\text{”}, i \geq 0)$
2. $\text{wp}(\text{“}i := i+1\text{”}, i = j)$
3. $\text{wp}(\text{“}i := i+1; j := j+i\text{”}, i = j)$
4. $\text{wp}(\text{“}i := 2i+1; j := j+i\text{”}, i = j)$
5. $\text{wp}(\text{“}j := j+i; i := 2i+1\text{”}, i = j)$
6. $\text{wp}(\text{“}t := i; i := j; j := t\text{”}, i = \widehat{i} \wedge j = \widehat{j})$
7. $\text{wp}(\text{“}i := 0; s := 0\text{”}, 0 \leq i < n \wedge s = (\sum j | 0 \leq j < i : b(j)))$.
8. $\text{wp}(\text{“}s := s + b(i); i := i+1\text{”}, 0 \leq i \leq n \wedge s = (\sum j | 0 \leq j < i : b(j)))$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 2.3.4.2** As part of the launch, you informally argued the correctness of the code segment

| |
|---|
| $\{\ Q:\ 0 \leq n \ \}$ |
| $S_0:\ s := 0$ |
| $S_1:\ k := 0$ |
| $\{\ R:\ (s = (\sum i \mid 0 \leq i < k : b(i))) \wedge (0 \leq k \leq n)\ \}$ |

where array $b$ has size $n$ with $0 \leq n$. **Prove** this code segment correct. (In the “Wrap Up” you find another exercise related to the correctness of the program in the launch.)

☛ SEE ANSWER
☛ DO EXERCISE ON edX

In our assertions, we will often use $\widehat{\ }$ to indicate a variable that is introduced to denote the “original contents” of a variable without the $\widehat{\ }$. For example, in the below homework $\widehat{x}$ and $\widehat{y}$ are used to denote the original values of variables $x$ and $y$.

**Homework 2.3.4.3** Prove the following code segment, which swaps the values of variables $x$ and $y$, correct.

| |
|---|
| $\{\ Q:\ (x = \widehat{x}) \wedge (y = \widehat{y})\ \}$ |
| $S_0:\ t := x$ |
| $S_1:\ x := y$ |
| $S_2:\ y := t$ |
| $\{\ R:\ (x = \widehat{y}) \wedge (y = \widehat{x})\ \}$ |

☛ SEE ANSWER
☛ DO EXERCISE ON edX

You may ask yourself "But how do I know if a code segment is *not* correct?"
Let's look at the above example, but with the incorrect implementation:

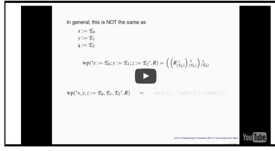
| |
|---|
| $\{\ Q:\ (x = \widehat{x}) \wedge (y = \widehat{y})\ \}$ |
| $S_0:\ y := x$ |
| $S_1:\ x := y$ |
| $\{\ R:\ (x = \widehat{y}) \wedge (y = \widehat{x})\ \}$ |

In this case,

$$\begin{aligned} \text{wp}(\text{"}S_0; S_1\text{"}, R) &= \text{wp}(\text{"}S_0\text{"}, \text{wp}(\text{"}x := y\text{"}, (x = \widehat{y}) \wedge (y = \widehat{x}))) \\ &= \text{wp}(\text{"}y := x\text{"}, (y = \widehat{y}) \wedge (y = \widehat{x})) \\ &= (x = \widehat{y}) \wedge (x = \widehat{x}) \end{aligned}$$

and $(x = \widehat{x}) \wedge (y = \widehat{y})$ does not, in general, imply $(x = \widehat{y}) \wedge (x = \widehat{x}))$. In other words, something is wrong!

## 2.3.5 Simultaneous assignment ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

The assignment statement discussed in the last unit assigns an expression to a single variable. It is often convenient to instead simultaneously assign multiple expressions to multiple, distinct variables as in

$$x, y, z := \mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2,$$

where $x, y, z$ are three distinct variables and $\mathcal{E}_0$, $\mathcal{E}_1$, and $\mathcal{E}_2$ are distinct expressions. For arbitrary postcondition $R$, the weakest precondition of this command is defined by

$$\text{wp}(\text{"}x, y, z := \mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2\text{"}, R) = R^{x,y,z}_{(\mathcal{E}_0),(\mathcal{E}_1),(\mathcal{E}_2)} \ ,$$

where $R^{x,y,z}_{(\mathcal{E}_0),(\mathcal{E}_1),(\mathcal{E}_2)}$ denotes simultaneous substitution of $(\mathcal{E}_0)$ for every free occurence of $x$, $(\mathcal{E}_1)$ for every free occurrence of $y$, and $(\mathcal{E}_2)$ for every free occurence of $z$.

**Homework 2.3.5.1** Evaluate

1. wp(“$i := i+1; j := 2i$”, $2i = j$)
2. wp(“$j := 2i; i := i+1$”, $2i = j$)
3. wp(“$i, j := i+1, 2i$”, $2i = j$)

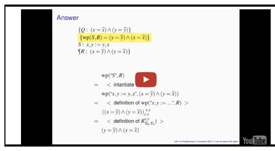
☛ SEE ANSWER
☛ DO EXERCISE ON edX

One reason for using simultaneous assignment is that it simplifies (shortens) the proof, as is illustrated in the following homework that achieves the same computation as was encountered in Homework 2.3.4.3:

**Homework 2.3.5.2** Prove the following code segment correct. It swaps the values of variables $x$ and $y$.

$$\begin{array}{l}\{Q:\ (x = \widehat{x}) \wedge (y = \widehat{y})\} \\ S:\ x, y := y, x \\ \{R:\ (x = \widehat{y}) \wedge (y = \widehat{x})\}\end{array}$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX


Answer

$\{Q:\ (x = \widehat{x}) \wedge (y = \widehat{y})\}$
$\{\mathrm{wp}(S,R) = (y = \widehat{y}) \wedge (x = \widehat{x})\}$
$S:\ x, y := y, x$
$\{R:\ (x = \widehat{y}) \wedge (y = \widehat{x})\}$

$$\begin{array}{cl} & \mathrm{wp}(\text{“}S\text{”}, R) \\ = & <\ \text{intantiate} \\ & \mathrm{wp}(\text{“}x,y := y,x\text{”}, (x = \widehat{y}) \wedge (y = \widehat{x})) \\ = & <\ \text{definition of } \mathrm{wp}(\text{“}x,y := \ldots\text{”}, R)\ > \\ & ((x = \widehat{y}) \wedge (y = \widehat{x}))_{y,x}^{x,y} \\ = & <\ \text{definition of } R_{\mathcal{E}_0,\mathcal{E}_1}^{x,y})\ > \\ & (y = \widehat{y}) \wedge (x = \widehat{x}) \end{array}$$

☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 2.3.5.3** Evaluate

1. wp(“$i := 2i+j; j := i+2j+4$”, $i = j$)
2. wp(“$j := i+2j+4; i := 2i+j$”, $i = j$)
3. wp(“$i, j := 2i+j, i+2j+4$”, $i = j$)

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 2.3.6 Assignment to an array element ☛ to edX

Example

$\{Q: (\forall i \mid 0 \le i < k : x(i) = y(i)) \wedge 0 \le k < n\}$
$\{\mathrm{wp}(\text{“}S\text{”}, R) : \qquad \}$
$S : x(k) := y(k)$
$\{R : (\forall i \mid 0 \le i \le k : x(i) = y(i)$ [illegible] $n\}$

☛ Watch Video on edX
☛ Watch Video on YouTube

Assignment to an array element is a bit more complicated. In order to define the weakest precondition of an assignment like $b(i) := \mathcal{E}$, where $\mathcal{E}$ is an expression, we need to introduce a new notation

$$(b; i : \mathcal{E})$$

which equals a copy of array $b$, but with the $i$th entry set to the result of evaluating expression $\mathcal{E}$. With this new notation/function, the assignment

$b(i) := \mathcal{E}$

can be thought of as the reassignment of the entire array

$b := (b; i : \mathcal{E})$

where $(b; i : \mathcal{E})$ denotes the array $b$ with the contents of only the $i$th element replaced with the value of expression $\mathcal{E}$. (There are examples in the video for this unit.) This allows us to define

$$\text{wp}(\text{“}b(i) = \mathcal{E}\text{”}, R) = \text{wp}(\text{“}b = (b; i : \mathcal{E})\text{”}, R) = R^{b}_{(b;i:\mathcal{E})}$$

much like assignment to a simple variable. Now, strictly speaking, it needs to be also asserted that $i$ is in the correct range and that $\mathcal{E}$ is a valid expression:

$$\text{wp}(\text{“}b(i) = \mathcal{E}\text{”}, R) = ((0 \le i < n) \wedge \text{valid}(\mathcal{E}) \wedge R^{b}_{(b;i;e)}).$$

Often, we will implicitly assume the fact that $i$ is in the range of the array and that $\mathcal{E}$ is a valid expression.

Let us check whether this has the desired effect by considering the following code segment that starts with all elements of array $b$ equaling the corresponding elements of array $\widehat{b}$ and then assigns $\mathcal{E}$ to $b(i)$, finishing with the assertion that all entries in $b$ equal those in $\widehat{b}$, except for the $i$th one, which equals $\mathcal{E}$. In other words, we check whether the following Hoare triple holds:

$\{Q : 0 \le i < n \wedge (\forall k \mid 0 \le k < n : b(k) = \widehat{b}(k))\}$
$b(i) := \mathcal{E}$
$\{R : 0 \le i < n \wedge (\forall k \mid 0 \le k < n \wedge k \ne i : b(k) = \widehat{b}(k)) \wedge b(i) = \mathcal{E}\}$

$Q \Rightarrow \text{wp}(\text{“}b(i) := \mathcal{E}\text{”}, R)$

$\Leftrightarrow$ < equivalent definition of := >

$Q \Rightarrow \text{wp}(\text{“}b := (b; i : \mathcal{E})\text{”}, R)$

$\Leftrightarrow$ < definition of weakest precondition of := >

$Q \Rightarrow R^{b}_{(b;i:\mathcal{E})}$

$\Leftrightarrow$ < instantiate $R$ >

$Q \Rightarrow (0 \le i < n \wedge (\forall k \mid 0 \le k < n \wedge k \ne i : b(k) = \widehat{b}(k)) \wedge b(i) = \mathcal{E})^{b}_{(b;i:\mathcal{E})}$

$\Leftrightarrow$ < definition of $R^{x}_{(\mathcal{E})}$ >

$Q \Rightarrow (0 \le i < n \wedge (\forall k \mid 0 \le k < n \wedge k \ne i : (b; i : \mathcal{E})(k) = \widehat{b}(k)) \wedge (b; i : \mathcal{E})(i) = \mathcal{E})$

$\Leftrightarrow$ < definition of $(b; i : \mathcal{E})$ >

$Q \Rightarrow (0 \le i < n \wedge (\forall k \mid 0 \le k < n \wedge k \ne i : b(k) = \widehat{b}(k)) \wedge \mathcal{E} = \mathcal{E})$

$\Leftrightarrow$ < identity >

$Q \Rightarrow (0 \le i < n \wedge (\forall k \mid 0 \le k < n \wedge k \ne i : b(k) = \widehat{b}(k)) \wedge T)$

$\Leftrightarrow$ < instantiate $Q$ >

$(0 \le i < n \wedge (\forall k \mid 0 \le k < n : b(k) = \widehat{b}(k)))$
$\quad \Rightarrow (0 \le i < n \wedge (\forall k \mid 0 \le k < n \wedge k \ne i : b(k) = \widehat{b}(k)) \wedge T)$

$\Leftrightarrow$ < split range; $\wedge$-simplification >

$(0 \le i < n \wedge (\forall k \mid 0 \le k < n \wedge k \ne i : b(k) = \widehat{b}(k)) \wedge b(i) = \widehat{b}(i))$
$\quad \Rightarrow 0 \le i < n \wedge (\forall k \mid 0 \le k < n \wedge k \ne i : b(k) = \widehat{b}(k))$

$\Leftrightarrow$ < weakening/strengthening >

$T$

In other words, the Hoare triple holds and the assignment to an array operates as desired.

Notice that we aren't consistent about using $i$ versus $k$ for the "bound variable" in a quantifier. This may cause some confusion. On the other hand, this will happen in practice, so you might as well get used to it.

*If* $i$ is not in the range of any quantifier in predicate $R$, then

$$\text{wp}(\text{“}b(i) = \mathcal{E}\text{”}, R) = R^{b(i)}_{(\mathcal{E})}$$

by which we mean that $(\mathcal{E})$ can simply be substituted in for each occurrence of $b(i)$.

**Homework 2.3.6.1** Prove the correctness of the following code segment. It might be part of a loop that scales the elements of array $b$ by scalar $\alpha \neq 0$. The array $\widehat{b}$ is introduced to refer to the original contents of $b$ and should not be used in actual computation. You may skip checking if the expression being assigned is valid (since they clearly are).

$$\left\{\; Q : (\forall i \mid 0 \leq i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge (\forall i \mid k \leq i < n : b(i) = \widehat{b}(i)) \wedge (0 \leq k < n) \;\right\}$$

$S : b(k) := \alpha \times b(k)$

$$\left\{\; R : (\forall i \mid 0 \leq i < k+1 : b(i) = \alpha \times \widehat{b}(i)) \wedge (\forall i \mid k+1 \leq i < n : b(i) = \widehat{b}(i)) \wedge (0 \leq k \leq n) \;\right\}$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

Like for assignment to simple variables, one can perform simultaneous assignments to array elements. Like for simple variables, the array elements to which we assign with a simultaneous assignment should be distinct.

**Homework 2.3.6.2** Consider the following code segment that swaps the contents of $b(i)$ and $b(j)$.

$$\left\{\; Q : (\forall k \mid 0 \leq k < n : b(k) = \widehat{b}(k)) \wedge (0 \leq i < n) \wedge (0 \leq j < n) \wedge i \neq j \;\right\}$$

$b(i), b(j) := b(j), b(i)$

$$\left\{\; \begin{array}{l} \{R :\; (\forall k \mid (0 \leq k < n) \wedge (k \neq i) \wedge (k \neq j) : b(k) = \widehat{b}(k)) \\ \qquad \wedge\, (b(i) = \widehat{b}(j)) \wedge (b(j) = \widehat{b}(i)) \wedge (0 \leq i < n) \wedge (0 \leq j < n)\} \end{array} \;\right\}$$

Prove it correct.

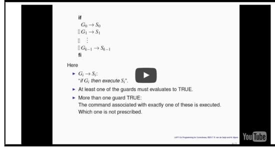
☛ SEE ANSWER
☛ DO EXERCISE ON edX

In the above examples, the code would be correct *even* if $i = j$. Still, it is better to handle the case where $i = j$ separately in a program, to avoid inadvertent errors.

In Weeks 4-6 we will see many instances of simultaneous assignment to one and two dimensional arrays. You will find out that we will always use the splitting of the range to isolate those elements that are being updated, so that we never need the notation $(b; i : \mathcal{E})$.

## 2.4 The If Command

### 2.4.1 Specification ☛ to edX



```
if
   $G_0 \rightarrow S_0$
$\square$ $G_1 \rightarrow S_1$
$\square$   ⋮
$\square$ $G_{k-1} \rightarrow S_{k-1}$
fi
```

Here

- $G_i \rightarrow S_i$:
  "*if $G_i$ then execute $S_i$*".
- At least one of the guards must evaluates to TRUE.
- More than one guard TRUE:
  The command associated with exactly one of these is executed.
  Which one is not prescribed.

LAFF-On Programming for Correctness, ©2017 R. van de Geijn and M. Myers

☛ Watch Video on edX
☛ Watch Video on YouTube

We are now ready to discuss the **if** command. It takes the following form:

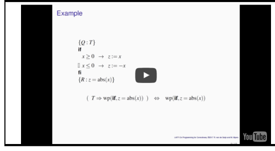
```
if
   $G_0 \rightarrow S_0$
$\square$ $G_1 \rightarrow S_1$
      ⋮
$\square$ $G_{k-1} \rightarrow S_{k-1}$
fi
```

Here

- $\square$ is a separator to make the statement easier to parse.
- Each $G_i \rightarrow S_i$ should be interpreted as "*if $G_i$ then execute $S_i$* and jump to immediately after the **fi**". The predicate $G_i$ is the *guard* for command $S_i$ and $G_i \rightarrow S_i$ a *guarded command*.
- When the **if** command is reached, it must be the case that at least one of the guards evaluates to TRUE.
- If more than one guard evaluates to true, the command associated with exactly one of these is executed. Which one is not prescribed.

This last bullet means that **our programs can be nondeterministic.** That is, two guards may evaluate to TRUE and it may not be predetermined which of the guarded commands is therefore executed. Regardless of the command that is chosen, $R$ must be satisfied upon completion by a correct **if** command.

### 2.4.2 wp("**if**", $R$) ☛ to edX



Example

```
{$Q : T$}
if
   $x \geq 0 \rightarrow z := x$
$\square$ $x \leq 0 \rightarrow z := -x$
fi
{$R : z = \text{abs}(x)$}
```

$$( T \Rightarrow \text{wp}(\textbf{if}, z = \text{abs}(x)) ) \Leftrightarrow \text{wp}(\textbf{if}, z = \text{abs}(x))$$

LAFF-On Programming for Correctness, ©2017 R. van de Geijn and M. Myers

☛ Watch Video on edX
☛ Watch Video on YouTube

Let us annotate this code segment some more, on our way to motivating how wp("**if**", $R$) is defined:

```
{wp("if", $R$)}
if
   $G_0 \rightarrow$ {wp("$S_0$", $R$)}$S_0${$R$}
$\square$ $G_1 \rightarrow$ {wp("$S_1$", $R$)}$S_1${$R$}
   ⋮
$\square$ $G_{k-1} \rightarrow$ {wp("$S_{k-1}$", $R$)}$S_{k-1}${$R$}
fi
{$R$}
```

What does this mean? By examining this annotated code we can justify the following definition of the weakest precondition for this generic **if** command:

$$\text{wp}(\text{"}\mathbf{if}\text{"}, R) = \overbrace{(G_0 \vee G_1 \vee \cdots \vee G_{k-1})}^{\text{at least one guard must evaluate to TRUE}} \wedge \overbrace{(G_0 \Rightarrow \text{wp}(\text{"}S_0\text{"}, R))}^{\text{if } G_i \text{ then } S_i \textit{ may} \text{ execute, in which case it must start in a state that guarantees completion in a state where } R \text{ holds.}} \\ \wedge (G_1 \Rightarrow \text{wp}(\text{"}S_1\text{"}, R)) \\ \vdots \qquad \vdots \\ \wedge (G_{k-1} \Rightarrow \text{wp}(\text{"}S_{k-1}\text{"}, R)). \tag{2.1}$$

Here we say "if $G_i$ then $S_i$ *may* execute," because in the case where multiple guards are TRUE, only one command $S_i$ is executed. Since we don't know which one, they *all* must have the property that they leave the program in a state where $R$ is TRUE. Implicit in this definition is that each of the guards $G_i$ must be well-defined, since otherwise the **if** statement will abort. Equivalently, we can state (2.3) as

$$\text{wp}(\text{"}\mathbf{if}\text{"}, R) = \overbrace{(\exists i | 0 \leq i < k : G_i)}^{\text{at least one guard must evaluate to TRUE}} \wedge \overbrace{(\forall i | 0 \leq i < k : G_i \Rightarrow \text{wp}(\text{"}S_i\text{"}, R))}^{\text{if } G_i \text{ then } S_i \textit{ may} \text{ execute, in which case it must start in a state that guarantees completion in a state where } R \text{ holds.}}. \tag{2.2}$$

**Example 2.3** Consider the following code segment.

```
if
   $x \geq 0$  →  $z := x$
 ▯ $x \leq 0$  →  $z := -x$
fi
```

Prove that regardless of the original value of scalar $x$, it sets $z$ to the absolute value of $x$.

Notice that this program is not deterministic in the sense that if $x = 0$ either $z := x$ or $z := -x$ may be executed.

---

**Proof:** We must prove that

$$T \Rightarrow \text{wp}(\text{"}\mathbf{if}\text{"}, z = \text{abs}(x))$$

---

Notice that proving that $T \Rightarrow p$ is TRUE is equivalent to proving that $p$ is TRUE.

$$
\begin{array}{ll}
 & \mathrm{wp}(\text{“}\textbf{if}\text{”}, z = \mathrm{abs}(x)) \\
\Leftrightarrow & < \text{Definition of } \textbf{if} > \\
 & \underbrace{(x \geq 0 \vee x \leq 0)}_{G_0 \vee G_1} \quad \wedge \quad \underbrace{(x \geq 0 \Rightarrow \mathrm{wp}(\text{“}z := x\text{”}, z = \mathrm{abs}(x))}_{G_0 \Rightarrow \mathrm{wp}(S_0, R)} \\
 & \qquad\qquad\qquad\quad \wedge \quad \underbrace{(x \leq 0 \Rightarrow \mathrm{wp}(\text{“}z := -x\text{”}, z = \mathrm{abs}(x))}_{G_1 \Rightarrow \mathrm{wp}(\text{“}S_1\text{”}, R)} \\
\Leftrightarrow & < \text{definition of } := > \\
 & (x \geq 0 \vee x \leq 0) \wedge (x \geq 0 \Rightarrow x = \mathrm{abs}(x)) \wedge (x \leq 0 \Rightarrow -x = \mathrm{abs}(x)) \\
\Leftrightarrow & < \text{algebra} > \\
 & T \wedge T \wedge T \\
\Leftrightarrow & < \wedge \text{ simplification } \times 2 > \\
 & T
\end{array}
$$

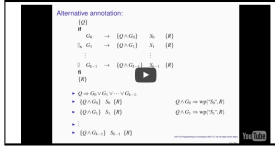
**Homework 2.4.2.1** In the above example, we use an intuitive understanding of the abs() function. We can refine this by recognizing that $z = \mathrm{abs}(x)$ is equivalent to $(x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x)$ so that the code segment becomes

| |
|---|
| $\{\ T \qquad \}$ |
| **if** |
| $\quad x \geq 0 \rightarrow z := x$ |
| $\quad x \leq 0 \rightarrow z := -x$ |
| **fi** |
| $\{\ (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x) \quad \}$ |

Prove this code segment correct.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 2.4.3 The If Theorem ☛ to edX


Alternative annotation:

☛ Watch Video on edX
☛ Watch Video on YouTube

Let us now revisit the annotated **if** command

```
{$Q$}
**if**
  $G_0 \rightarrow \{Q \wedge G_0\} S_0 \{R\}$
□ $G_1 \rightarrow \{Q \wedge G_1\} S_1 \{R\}$
   ⋮
□ $G_{k-1} \rightarrow \{Q \wedge G_{k-1}\} S_{k-1} \{R\}$
**fi**
{$R$}
```

Assuming all guards are well-defined, we notice that we can break up the proof of correctness into parts:

- We know that $Q$ must imply that at least one of the guards is TRUE:
$$Q \Rightarrow (G_0 \vee G_1 \vee \cdots \vee G_{k-1}).$$
- For each of the commands $S_i$ we must establish that
$$\{Q \wedge G_i\} S_i \{R\}$$
holds or, equivalently, that
$$(Q \wedge G_i) \Rightarrow \text{wp}(S_i, R).$$

These observations can be stated as a theorem:

**Theorem 2.4 (If Theorem)** *The annotated code segment*

```
{$Q$}
**if**
□  $G_0 \rightarrow S_0$
   ⋮
□  $G_{K-1} \rightarrow S_{K-1}$
**fi**
{$Q$}
```

*is correct if and only if*

- $Q \Rightarrow (\exists i | 0 \leq i < K : G_i)$ *and*
- $(\forall i | 0 \leq i < K : Q \wedge G_i \Rightarrow \text{wp}(S_i, R))$

*In other words, under these conditions* $Q \Rightarrow \text{wp}(\text{“}\textbf{if}\text{”}, R)$.

Recall from Unit 1.3.2 that in Homeworks 1.3.2.1 and 1.3.2.2 we proved

1. $(p \Rightarrow (q \wedge r)) \Leftrightarrow ((p \Rightarrow q) \wedge (p \Rightarrow r))$.
2. $(p \Rightarrow (q \Rightarrow r)) \Leftrightarrow (p \wedge q \Rightarrow r)$.

---

**Proof:** (If Theorem) The proof uses the above insights:

$$Q \Rightarrow \text{wp}(\text{“}\textbf{if}\text{”}, R)$$
$$\Leftrightarrow < \text{definition of wp( “}\textbf{if}\text{”, R )} >$$
$$Q \Rightarrow (G_0 \vee \cdots \vee G_{K-1}) \wedge (G_0 \Rightarrow \text{wp}(S_0, R)) \wedge \cdots \wedge (G_{K-1} \Rightarrow \text{wp}(S_{K-1}, R))$$
$$\Leftrightarrow < p \Rightarrow q \wedge r \Leftrightarrow (p \Rightarrow q) \wedge (p \Rightarrow r), \text{ several times} >$$
$$(Q \Rightarrow G_0 \vee \cdots \vee G_{K-1}) \wedge (Q \Rightarrow (G_0 \Rightarrow \text{wp}(S_0, R))) \wedge \cdots \wedge (Q \Rightarrow (G_{K-1} \Rightarrow \text{wp}(S_{K-1}, R)))$$
$$\Leftrightarrow < (p \Rightarrow (q \Rightarrow r)) \Leftrightarrow ((p \wedge q) \Rightarrow r), \text{ several times} >$$
$$(Q \Rightarrow G_0 \vee \cdots \vee G_{K-1}) \wedge (Q \wedge G_0 \Rightarrow \text{wp}(S_0, R)) \wedge \cdots \wedge (Q \wedge G_{K-1} \Rightarrow \text{wp}(S_{K-1}, R))$$

---

The proof is completed by noting that to prove a conjunction to be TRUE, all you have to do is to prove each sub-predicate in the conjunction to be TRUE.

---

The If Theorem allows us to break $Q \Rightarrow \text{wp}(\text{"}\textbf{if}\text{"}, R)$ into smaller, more manageable, pieces. Let us demonstrate this by revisiting Homework 2.4.2.1:

| |
|---|
| $\{\ T \qquad \}$ |
| **if** |
| $\quad x \geq 0 \rightarrow z := x$ |
| $\quad x \leq 0 \rightarrow z := -x$ |
| **fi** |
| $\{\ (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x) \qquad \}$ |

To prove the correctness of this code segment, we employ the If Theorem and check

- $Q \Rightarrow G_0 \vee \cdots \vee G_{k-1}$:

$$\begin{aligned} & Q \Rightarrow G_0 \vee G_1 \\ \Leftrightarrow\ & < \text{instantiate} > \\ & T \Rightarrow x \geq 0 \vee x \leq 0 \\ \Leftrightarrow\ & < \text{algebra; } \Rightarrow\text{-simplification} > \\ & T \end{aligned}$$

- $Q \wedge G_0 \Rightarrow \text{wp}(S_0, R)$:

$$\begin{aligned} & Q \wedge G_0 \Rightarrow \text{wp}(S_0, R) \\ \Leftrightarrow\ & < \text{instantiate} > \\ & T \wedge x \geq 0 \Rightarrow \text{wp}(\text{"}z := x\text{"}, (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x)) \\ \Leftrightarrow\ & < \wedge\text{-simplification, definition of } := > \\ & x \geq 0 \Rightarrow (x \geq 0 \wedge x = x) \vee (x \leq 0 \wedge x = -x) \\ \Leftrightarrow\ & < \text{algebra } \times 2 > \\ & x \geq 0 \Rightarrow (x \geq 0 \wedge T) \vee (x \leq 0 \wedge x = 0) \\ \Leftrightarrow\ & < \wedge\text{-simplification; algebra} > \\ & x \geq 0 \Rightarrow x \geq 0 \vee x = 0 \\ \Leftrightarrow\ & < \text{weakening/strengthening} > \\ & T \end{aligned}$$

- $Q \wedge G_1 \Rightarrow \text{wp}(\text{"}S_1\text{"}, R)$: See Homework 2.4.3.1.

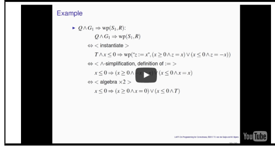
---

**Homework 2.4.3.1** Complete the proof of the correctness of

| | |
|---|---|
| $\{\ T$ | $\}$ |
| **if** | |
| $\quad x \geq 0 \rightarrow z := x$ | |
| $\quad x \leq 0 \rightarrow z := -x$ | |
| **fi** | |
| $\{\ (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x)$ | $\}$ |

from the last example.

☛ SEE ANSWER
☛ DO EXERCISE ON edX


Example

- $Q \wedge G_1 \Rightarrow \text{wp}(S_1, R)$:

$$\begin{aligned}
& Q \wedge G_1 \Rightarrow \text{wp}(S_1, R) \\
\Leftrightarrow\ & < \text{instantiate} > \\
& T \wedge x \leq 0 \Rightarrow \text{wp}(\text{“}z := x\text{”}, (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x)) \\
\Leftrightarrow\ & < \wedge\text{-simplification, definition of} := > \\
& x \leq 0 \Rightarrow (x \geq 0 \wedge -x = x) \vee (x \leq 0 \wedge x = x) \\
\Leftrightarrow\ & < \text{algebra} \times 2 > \\
& x \leq 0 \Rightarrow (x \geq 0 \wedge x = 0) \vee (x \leq 0 \wedge T)
\end{aligned}$$

LAFF-On Programming for Correctness, ©2017 R. van de Geijn and M. Myers

☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 2.4.3.2** The following code segment sets $m$ to the maximum of $x$ and $y$. Use the If Theorem to prove it correct.

```
{$Q : T$}
if
  $x \geq y \quad \rightarrow \quad m := x$
$\square$ $x \leq y \quad \rightarrow \quad m := y$
fi
{$R : (x \geq y \wedge m = x) \vee (x \leq y \wedge m = y)$}
```

☛ SEE ANSWER
☛ DO EXERCISE ON edX

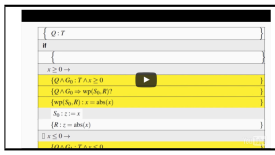
**Homework 2.4.3.3** The following code segment might be part of a loop that computes $m$, the minimum value in array $b$:

```
{$Q : (\forall j \mid 1 \leq j < i : m \leq b(j)) \wedge 0 \leq i < n$}
if
  $b(i) \geq m \quad \rightarrow$ skip
$\square$ $b(i) \leq m \quad \rightarrow \quad m := b(i)$
fi
$i := i + 1$
{$R : (\forall j \mid 1 \leq j < i : m \leq b(j)) \wedge 0 \leq i \leq n$}
```

Prove it correct.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 2.4.4 A worksheet for proving an if command correct ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Once again, let's revisit

```
{$Q : T$}
if
   $x \geq 0 \;\rightarrow\; z := x$
$\square$ $x \leq 0 \;\rightarrow\; z := -x$
fi
{$R : z = \text{abs}(x)$}
```

Provided $Q \Rightarrow (x \geq 0) \vee (x \leq 0)$, we can view proving this correct as annotating the program: Starting with what we know, given in Figure 2.2, we can fill in $Q \wedge G_0$ and $Q \wedge G_1$, $R$, and $\text{wp}(\text{“}S_0\text{”}, R)$ and $\text{wp}(\text{“}S_1\text{”}, R)$, yielding Figure 2.3. This is a matter of inserting known predicates and evaluating the wp operator. What is left now is to check whether

- $Q \Rightarrow G_0 \vee G_1 :$ ;
- $Q \wedge G_0 \Rightarrow \text{wp}(\text{“}S_0\text{”}, R)$ ; and
- $Q \wedge G_1 \Rightarrow \text{wp}(\text{“}S_1\text{”}, R)$ .

which, after proving these parts either by examination or via formal proof, leaves us with the completed “worksheet” in Figure 2.4.

What this “worksheet” does is on the one hand illustrate more visually how the **if** Theorem relates to systematic reasoning about the correctness of an **if** command and on the other hand structures this reasoning.

### 2.4.5 The if-then-else command ☛ to edX

In practice, languages usually support a variant of the **if** command that we will call the **if-then-else** command, specified by

```
if $G$
   $S_0$
else
   $S_1$
fi
```

It is equivalent to

```
if
     $G \rightarrow S_0$
$\square$ $\neg G \rightarrow S_1$
fi
```

The important observation is that then automatically $Q \Rightarrow G \vee \neg G$ since $G \vee \neg G$ simplifies to TRUE by law of excluded middle. As a result, that part of the If Theorem needs not be checked for the **if-then-else** command.

The worksheet for the **if-then-else** command is given in Figure 2.5.

```
{ $Q$ : }
if
    {$Q \Rightarrow G_0 \vee G_1$ : }
        $G_0 \rightarrow$
    {$Q \wedge G_0$ : }
    { $Q \wedge G_0$ $\Rightarrow$ wp(“$S_0$”, $R$) ? }
    {wp(“$S_0$”, $R$) : }
    $S_0$ :
    {$R$ : }
▯       $G_1 \rightarrow$
    {$Q \wedge G_1$ : }
    { $Q \wedge G_1$ $\Rightarrow$ wp(“$S_1$”, $R$) ? }
    {wp(“$S_1$”, $R$) : }
    $S_1$ :
    {$R$ : }
fi
{ $R$ : }
```

Figure 2.2: Recipe for checking the correctness of the **if** command.

```
$\{\; Q : T \;\}$
if
    $\{Q \Rightarrow G_0 \vee G_1 : \quad\}$
  $x \geq 0 \rightarrow$
    $\{Q \wedge G_0 : T \wedge x \geq 0 \quad\}$
    $\left\{ \boxed{Q \wedge G_0} \Rightarrow \boxed{\mathrm{wp}(``S_0", R)} : \right\}$
    $\{\mathrm{wp}(``S_0", R) : x = \mathrm{abs}(x) \quad\}$
    $S_0 : z := x$
    $\{R : z = \mathrm{abs}(x) \quad\}$
$\square$ $x \leq 0 \rightarrow$
    $\{Q \wedge G_1 : T \wedge x \leq 0 \quad\}$
    $\left\{ \boxed{Q \wedge G_1} \Rightarrow \boxed{\mathrm{wp}(``S_1", R)} : \right\}$
    $\{\mathrm{wp}(``S_1", R) : -x = \mathrm{abs}(x) \quad\}$
    $S_1 : z := -x$
    $\{R : z = \mathrm{abs}(x) \quad\}$
fi
$\{\; R : z = \mathrm{abs}(x) \;\}$
```

Figure 2.3: Recipe for checking the correctness of the **if** command after filling in various parts.

| |
|---|
| $\left\{ \; Q : T \qquad \right\}$ |
| **if** |
| $\{Q \Rightarrow G_0 \vee G_1 : \checkmark \qquad \}$ |
| $x \geq 0 \rightarrow$ |
| $\{Q \wedge G_0 : T \wedge x \geq 0 \qquad \}$ |
| $\left\{ \boxed{Q \wedge G_0} \Rightarrow \boxed{\text{wp}(\text{“}S_0\text{”}, R)} : \checkmark \qquad \right\}$ |
| $\{\text{wp}(\text{“}S_0\text{”}, R) : x = \text{abs}(x) \qquad \}$ |
| $S_0 : z := x$ |
| $\{R : z = \text{abs}(x) \qquad \}$ |
| $\square \; x \leq 0 \rightarrow$ |
| $\{Q \wedge G_1 : T \wedge x \leq 0 \qquad \}$ |
| $\left\{ \boxed{Q \wedge G_1} \Rightarrow \boxed{\text{wp}(\text{“}S_1\text{”}, R)} : \checkmark \qquad \right\}$ |
| $\{\text{wp}(\text{“}S_1\text{”}, R) : -x = \text{abs}(x) \qquad \}$ |
| $S_1 : z := -x$ |
| $\{R : z = \text{abs}(x) \qquad \}$ |
| **fi** |
| $\left\{ \; R : z = \text{abs}(x) \qquad \right\}$ |

Figure 2.4: Completed worksheet for checking the correctness of the **if** command.

| |
|---|
| $\{\ Q:\ \}$ |
| **if** $G$ |
| $\{G \wedge Q:\ \}$ |
| $\{G \wedge Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, R)?\ \}$ |
| $\{\text{wp}(\text{“}S_0\text{”}, R):\ \}$ |
| $S_0:$ |
| $\{R:\ \}$ |
| **else** |
| $\{\neg G \wedge Q:\ \}$ |
| $\{\neg G \wedge Q \Rightarrow \text{wp}(\text{“}S_1\text{”}, R)?\ \}$ |
| $\{\text{wp}(\text{“}S_1\text{”}, R):\ \}$ |
| $S_1:$ |
| $\{R:\ \}$ |
| **fi** |
| $\{\ R:\ \}$ |

Figure 2.5: If-then-else worksheet.

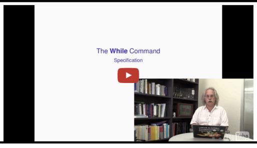
**Homework 2.4.5.1** The following code segment sets $m$ to the maximum of $x$ and $y$ with an if-then-else command. Use Figure 2.5 to prove it correct.

```
{$Q : T$}
**if** $x \geq y$
  $m := x$
**else**
  $m := y$
**fi**
{$R : (x \geq y \wedge m = x) \vee (\neg(x \geq y) \wedge m = y)$}
```

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 2.5 The While Command

### 2.5.1 Specification ☛ to edX



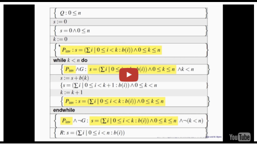
☛ Watch Video on edX
☛ Watch Video on YouTube

In the launch for Week 2, we introduced the concept of a **while** command (loop):

```
**while** $G$ **do**
   $S$
**endwhile**
```

and discussed various annotations for an example. The loop continues to iterate until $G$ evaluates to FALSE, at which point control proceeds to the first statement after **endwhile**.

### 2.5.2 Correctness ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Consider again the example from the launch for this week, given in Figure 2.6. The predicate highlighted in yellow, known as the loop invariant for this program, is fundamental to the proof of correctness of the loop, as we will now see.

Inspired by the example, let us annotate the generic **while** loop:

```
{ $Q$ }
{ $P_{\rm inv}$ }
while $G$ do
    { $P_{\rm inv} \wedge G$ }
    $S$
    { $P_{\rm inv}$ }
endwhile
{ $P_{\rm inv} \wedge \neg G$ }
{ $R$ }
```

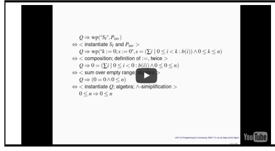
in preparation for a theorem regarding its correctness. Here $P_{\rm inv}$ represents the *loop invariant* for the loop. We notice the following:

- $P_{\rm inv}$ holds before the loop.
- When the loop is entered the first time, no computation is performed between the assertion $P_{\rm inv}$ and the assertion $P_{\rm inv} \wedge G$ at the top of the loop body. Thus, $P_{\rm inv}$ is true at the top of the loop body as well.
- *If* $\{P_{\rm inv} \wedge G\}S\{P_{\rm inv}\}$ holds, then we know that $P_{\rm inv}$ holds at the bottom of the loop body at the end of the first iteration.
- We conclude that $P_{\rm inv} \wedge G$ is TRUE before the executions of the loop body and $P_{\rm inv}$ is true after execution of the loop body for every iteration.
- **If** the loop guard evaluate to FALSE, **then** the loop stops executing, leaving the program in a state where $P_{\rm inv}$ is still TRUE and $G$ evaluates to FALSE, right after the loop.
- Now, *if* $P_{\rm inv} \wedge \neg G$ implies $R$, then we can conclude that $\{Q\}$**while**$\{R\}$ is TRUE, meaning that the loop correctly computes $R$ if entered in a state where $Q$ is TRUE **provided** it can be shown that the loop terminates in a finite amount of time.

The reader will notice that this is merely an application of the Principle of Mathematical Induction:

- The base case is that $P_{\rm inv}$ holds before the loop starts.
- The inductive step is the assumption that $P_{\rm inv} \wedge G$ holds at the top of the loop body (for the $k$th iteration) and the proof that shows $P_{\rm inv}$ holds at the bottom of the loop body (and hence at the top of the loop body in the next iteration).
- By the Principle of Mathematical Induction, it holds at the top and the bottom of the loop for every iteration.

## 2.5.3 The While Theorem ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

The observations in the last unit motivate the following theorem:

**Theorem 2.5 (While Theorem)**

```
$\{Q\}$
$\{P_{\rm inv}\}$
while $G$ do
    $\{P_{\rm inv} \wedge G\}$
    $S$
    $\{P_{\rm inv}\}$
endwhile
$\{P_{\rm inv} \wedge \neg G\}$
$\{R\}$
```

*Partial correctness:*

- $Q \Rightarrow P_{\rm inv}$;
- $\{P_{\rm inv} \wedge G\}S\{P_{\rm inv}\}$; *and*
- $P_{\rm inv} \wedge \neg G \Rightarrow R$

*Complete correctness, in addition, for some bound function $t$:*

- $P_{\rm inv} \wedge G \Rightarrow t \geq 0$ *and*
- $\{P_{\rm inv} \wedge G\}t' := t; S\{t < t'\}$

**Example 2.6** *Prove the program in Figure 2.6 correct.*

- *Prove that the loop invariant is TRUE before the loop. In other words, show that the initialization $\alpha := 0; k := 0$ puts the program in a state where $P_{\rm inv}$ is TRUE. This was established in Homework 2.3.4.2. Here we repeat the answer to that exercise:*

$$
\begin{aligned}
& Q \Rightarrow \mathrm{wp}(\text{“}S_I\text{”}, P_{\rm inv}) \\
\Leftrightarrow\ & < \textit{instantiate } S_I \textit{ and } P_{\rm inv} > \\
& Q \Rightarrow \mathrm{wp}(\text{“}k := 0; s := 0\text{”}, s = (\textstyle\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n) \\
\Leftrightarrow\ & < \textit{composition; definition of } :=\textit{, twice} > \\
& Q \Rightarrow 0 = (\textstyle\sum i \mid 0 \leq i < 0 : b(i)) \wedge 0 \leq 0 \leq n) \\
\Leftrightarrow\ & < \textit{sum over empty range; algebra} > \\
& Q \Rightarrow (0 = 0 \wedge 0 \leq n) \\
\Leftrightarrow\ & < \textit{instantiate } Q\textit{; algebra; } \wedge\textit{-simplification} > \\
& 0 \leq n \Rightarrow 0 \leq n \\
\Leftrightarrow\ & < \Rightarrow\textit{-simplification} > \\
& T
\end{aligned}
$$

- *Prove that if the loop invariant holds at the beginning of an iteration (and hence the guard holds), then it holds again at the end of that iteration: $P_{\rm inv} \wedge G \Rightarrow \mathrm{wp}(\text{“}S\text{”}, P_{\rm inv})$. This was shown in Homework 2.7.1.1. Here we repeat:*

$$P_{\text{inv}} \wedge G \Rightarrow \text{wp}(\text{“}S\text{”}; P_{\text{inv}})$$
$\Leftrightarrow <$ *instantiate* $>$
$$(P_{\text{inv}} \wedge G) \Rightarrow \text{wp}(\text{“}s := s + b(k); k := k+1\text{”}, s = (\sum i \mid 0 \leq i < k : b(i)) \\ \wedge 0 \leq k \leq n)$$
$\Leftrightarrow <$ *composition; definition of* $:= >$
$$(P_{\text{inv}} \wedge G) \Rightarrow \text{wp}(\text{“}s := s + b(k)\text{”}, s = (\sum i \mid 0 \leq i < k+1 : b(i)) \\ \wedge 0 \leq k+1 \leq n)$$
$\Leftrightarrow <$ *definition of* $:= >$
$$(P_{\text{inv}} \wedge G) \Rightarrow s + b(k) = (\sum i \mid 0 \leq i < k+1 : b(i)) \wedge (0 \leq k+1 \leq n)$$
$\Leftrightarrow <$ *instantiate; algebra; split range* $>$
$$(s = (\sum k | 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge k < n) \\ \Rightarrow (s + b(k) = (\sum i \mid 0 \leq i < k : b(i)) + b(k) \wedge 0 \leq k+1 \leq n)$$
$\Leftrightarrow <$ *algebra* $>$
$$(s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k < n) \\ \Rightarrow (s = (\sum i | 0 \leq i < k : b(i)) \wedge -1 \leq k < n)$$
$\Leftrightarrow <$ *algebra* $>$
$$(s = (\sum j | 1 \leq j < i : b(j)) \wedge 0 \leq k < n) \\ \Rightarrow (s = (\sum j \mid 0 \leq i < k : b(i)) \wedge (-1 = k \vee 0 \leq k < n))$$
$\Leftrightarrow < \wedge$*-distributivity; weakening/strengthening* $>$
$$T$$

- *Prove that the fact that after the last iteration the loop invariant holds and the loop guard is FALSE implies that the desired result has been computed:* $(P_{\text{inv}} \wedge \neg(G)) \Rightarrow R$

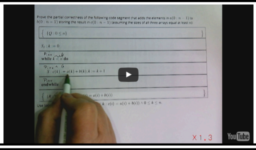
$$P_{\text{inv}} \wedge \neg G \Rightarrow R$$
$\Leftrightarrow <$ *instantiate* $>$
$$(s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge \neg(k < n)) \Rightarrow R)$$
$\Leftrightarrow <$ *algebra* $>$
$$(s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge k \geq n) \Rightarrow R)$$
$\Leftrightarrow <$ *algebra; instantiate R* $>$
$$(s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k = n) \Rightarrow (s = (\sum i \mid 0 \leq i < n : b(i)))$$
$\Leftrightarrow <$ *substitute* $k = n >$
$$(s = (\sum i \mid 0 \leq i < n : b(i)) \wedge 0 \leq k = n) \Rightarrow (s = (\sum i \mid 0 \leq i < n : b(i))$$
$\Leftrightarrow <$ *weakening/strengthening* $>$
$$T$$

**Homework 2.5.3.1** Prove the partial correctness of the following code segment that adds the elements in $a(0:n-1)$ to $b(0:n-1)$ storing the result in $c(0:n-1)$ (assuming the sizes of all three arrays equal at least $n$):

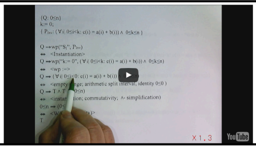
```
{ $\{Q : 0 \leq n\}$ }
$S_I : k := 0$
while $k < n$ do
    $S : c(k) := a(k) + b(k); k := k + 1$
endwhile
{ $(R : \forall i \mid 0 \leq i < n : c(i) = a(i) + b(i))$ }
```

Use loop invariant $P_{\text{inv}} : (\forall i \mid 0 \leq i < k : c(i) = a(i) + b(i)) \wedge 0 \leq k \leq n$.

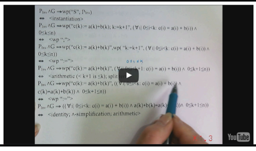
☛ SEE ANSWER
☛ DO EXERCISE ON edX



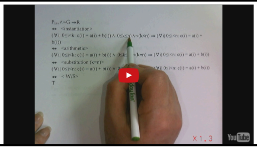
☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube

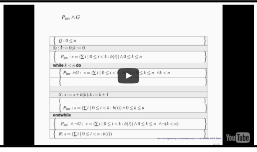


☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube

### 2.5.4 Total correctness ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Notice that in the example in the last unit we proved that **if** the loop completes, **then** we can conclude that it computes the correct value. But how do we know it completes? For the example that computes the sum of the elements of $b$ we can informally reason that

- Initially $k = 0$;
- Each time the loop body is executed, $k$ is incremented; and
- The loop only continues to execute as long as $k < n$.

Hence, we can conclude that eventually the loop must terminate.

Let us formalize this. We define $t$, *a function of some or all the variables encountered in the loop*, to be the *bound function* for the loop if $t$ satisfies

- $P_{\text{inv}} \wedge G \Rightarrow (t \geq 0)$; and
- $\{P_{\text{inv}} \wedge G\}\ t' := t; S\ \{t < t'\}$.

The first condition says that the bound function is bounded below by zero. The second condition means that each time through the loop $t$ decreases in value.

Bounding $t \geq 0$ is merely a choice. The lower bound could be any finite value. Strictly speaking $t < t'$ is not enough: $t$ must decrease every time through the iteration by at least a positive constant, since otherwise it could converge to zero without ever becoming less than zero.
The point is that the two conditions together force the number of iterations that are executed to be finite, which means the loop completes in a finite amount of time. Why? The bound function can't be bounded below and decreased infinitely often by at least a positive constant. Thus, the loop must terminate.

The concept of the bound function and how to find it seems to give the novice trouble. Here is what you usually do:

- Identify a variable that is systematically only incremented or only decremented every time an iteration of the loop is executed. Often, this is the loop index.
- Create a function of that variable that has the property that
  - The value of that function decreases every time an iteration of the loop is executed.
  - The value remains nonnegative as long as the loop continues to execute.

In Example 2.6, we notice that $k$ increases every time an iteration of the loop is executed. We also notice that it is bounded by $n$ because the loop guard is $k < n$. So, we can take the function $t$ to equal $(n-k)$. It decreases every time an iteration of the loop is executed and it is bounded below: Then

- $P_{\text{inv}} \wedge G \Rightarrow (t \geq 0)$:

$$\begin{array}{ll} & P_{\text{inv}} \wedge (k < n) \Rightarrow ((n-k) \geq 0) \\ \Leftrightarrow & < \text{algebra} > \\ & P_{\text{inv}} \wedge (k < n) \Rightarrow (k \leq n) \\ \Leftrightarrow & < \text{algebra} > \\ & P_{\text{inv}} \wedge (k < n) \Rightarrow ((k < n) \vee (k = n)) \\ \Leftrightarrow & < \text{weakening/strengthening} > \\ & T \end{array}$$

  Notice that we chose $t = n-k$ because then the guard $G : k < n$ can be rewritten as $n-k > 0$ and hence $n-k \geq 0$. Alternatively, we could have chosen $t = n-k-1$ in which case $n-k > 0$ implies that $n-k-1 \geq 0$ (since $n-k$ only takes on integer values). Here one could have chosen instead $t : 123n - 123k + 172$ is also a bound function. But there is an obvious benefit to keeping this function simple.

- $\{P_{\rm inv} \wedge G\} t' := t; S\{t < t'\}$:

$$
\begin{array}{l}
\quad \{P_{\rm inv} \wedge G\} t' := t; S\{t < t'\} \\
\Leftrightarrow < \text{definition} > \\
\quad (P_{\rm inv} \wedge G) \Rightarrow \text{wp}(\text{``}t' := t; S\text{''}, t < t') \\
\Leftrightarrow < \text{Instantiate} > \\
\quad (P_{\rm inv} \wedge G) \Rightarrow \text{wp}(\text{``}t' := n-k; s = s + b(k); k := k+1\text{''}, n-k < t') \\
\Leftrightarrow < \text{definition of} := > \\
\quad (P_{\rm inv} \wedge G) \Rightarrow \text{wp}(\text{``}t' := n-k; s = s + b(i)\text{''}, n-(k+1) < t') \\
\Leftrightarrow < \text{definition of} := > \\
\quad (P_{\rm inv} \wedge G) \Rightarrow \text{wp}(\text{``}t' := n-k\text{''}, n-(k+1) < t') \\
\Leftrightarrow < \text{definition of} := > \\
\quad (P_{\rm inv} \wedge G) \Rightarrow (n-(k+1) < n-k) \\
\Leftrightarrow < \text{algebra} > \\
\quad (P_{\rm inv} \wedge G \Rightarrow (-1 < 0) \\
\Leftrightarrow < \text{algebra} > \\
\quad (P_{\rm inv} \wedge G) \Rightarrow T \\
\Leftrightarrow < \Rightarrow\text{-simplification} > \\
\quad T
\end{array}
$$

Notice that we never had to instantiate $P_{\rm inv} \wedge G$. This is often the case and can save you a lot of writing.

The discussion so far leads us to the **checklist for a while command** in Figure 2.7.

Some of you may have noticed that the condition for total correctness regarding bound function $t$ decreasing at every step is not quite good enough if the amount by which it decreases is not, for example, an integer. The definition was proposed by people who focus on "discrete" problems.
A better definition of a bound function is $t$ such that $\{P_{\rm inv} \wedge G\}\ t' := t; S\ \{t' - t \geq 1\}$. What this avoids is the possibility that $t' - t$ becomes infinitesimally small, in which case the lower bound may never be reached.

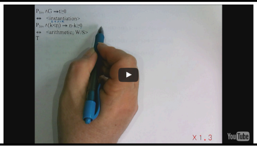
**Homework 2.5.4.1** In Homework 2.5.3.1 you proved the partial correctness of the following code segment that adds the elements in $a(0:n-1)$ to $b(0:n-1)$ storing the result in $c(0:n-1)$ (assuming the sizes of all three arrays equal at least $n$).

```
{ $\{Q : 0 \leq n\}$ }
$S_I : k := 0$
{ $\{P_{\text{inv}} : (\forall i \mid 0 \leq i < k : c(i) = a(i) + b(i)) \wedge 0 \leq k \leq n\}$ }
{ $\{t : n - k\}$ }
while $k < n$ do
    $S : c(k) := a(k) + b(k); k := k + 1$
endwhile
{ $(R : \forall i \mid 0 \leq i < n : c(i) = a(i) + b(i))$ }
```

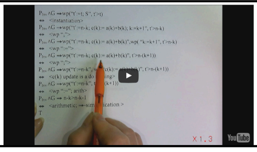
Prove in addition the total correctness of this code segment.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

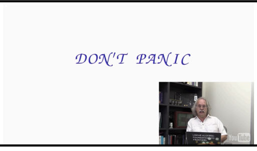


☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube

### 2.5.5 Don't Panic ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

As it says in the Hitchhiker's Guide to the Galaxy:

> "... the Hitchhiker's Guide to the Galaxy itself has outsold the Encyclopedia Galactica because it is slightly cheaper, and because it has the words 'DON'T PANIC' in large, friendly letters on the cover."

What you now start to appreciate is the line from Dijkstra's ☛ "The Humble Programmer":

> "But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden."

In the next week and subsequent weeks you will find out about the next observation in that essay:

> "On the contrary: the programmer should let correctness proof and program grow hand in hand."

We hope that by the end of the course you will find achieving this doable and valuable, motivated by important vector and matrix computation.

## 2.6 Enrichment

### 2.6.1 The do command ☛ to edX

A generization of the **while** command is the **do** command which is given by

```
do
   $G_0 \rightarrow S_0$
$\square$ $G_1 \rightarrow S_1$
        $\vdots$
$\square$ $G_{K-1} \rightarrow S_{K-1}$
od
```

Like for the **if** command, each $G_i \rightarrow S_i$ is a guarded command, where $S_i$ is a candidate for execution if the guard $G_i$ evaluates to TRUE. If $G_i$ is the only guard that evaluates to TRUE, then $S_i$ is executed. If multiple guards evaluate to TRUE, then exactly one of the corresponding commands is executed. The loop continues to iterate until none of the guards are `true`, at which point control proceeds to the first statement after **do**.

Let us assume that the **do** loop can be annotated as in Figure 2.8, in preparation for a theorem regarding the correctness of a **do** loop. Here $P_{inv}$ again denotes a *loop invariant* for the loop. To analyze the correctness of the loop $\{Q\}$**do** $\{R\}$ we notice the following:

- $P_{\text{inv}}$ holds before the loop starts (at 2.),
- When the loop is entered the first time, no computation is performed before 3. is reached. Thus, $P_{\text{inv}}$ is true there too.
- The guarded commands in the loop body are such that regardless of which guard evaluates to TRUE and is chosen, the execution of the corresponding command leaves the program again in a state where $P_{\text{inv}}$ is TRUE.
- As a result, $P_{\text{inv}}$ is TRUE before the executions of the loop body (at 3.) and after execution of the loop body (at 5.) for every iteration.
- **If** ever none of the loop guards evaluate to TRUE, **then** the loop stops executing, leaving the program in a state where $P_{\text{inv}}$ is still TRUE and $G_0 \vee \cdots \vee G_{K-1}$ evaluates to FALSE, at 6.
- Now, *if* $P_{\text{inv}}$ implies $R$, then we can conclude that $\{Q\}$**do** $\{R\}$ is TRUE, meaning that this code segment is correct.

#### Partial correctness

The observations in the last unit motivate the following theorem:

**Theorem 2.7 (do Theorem, Partial Correctness)**

```
$\{Q\}$
$\{P_{\rm inv}\}$
do
  $G_0 \rightarrow \{P_{\rm inv} \wedge G_0\}\ S_0 \{P_{\rm inv}\}$
▯ $G_1 \rightarrow \{P_{\rm inv} \wedge G_1\}\ S_1 \{P_{\rm inv}\}$
      $\vdots$
▯ $G_{K-1} \rightarrow \{P_{\rm inv} \wedge G_{K-1}\}\ S_{K-1} \{P_{\rm inv}\}$
od
$\{P_{\rm inv} \wedge \neg(G_0 \vee \cdots \vee G_{K-1})\}$
$\{R\}$
```

*If predicate $P_{\rm inv}$ satisfies*

- $Q \Rightarrow P_{\rm inv}$;
- $P_{\rm inv} \wedge G_i \Rightarrow \text{wp}(\text{“}S_i\text{”}, P_{\rm inv})$, *for* $i = 0, \ldots, K-1$; *and*
- $(P_{\rm inv} \wedge \neg(G_0 \vee \cdots \vee G_{K-1})) \Rightarrow R$

*then the code segment on the left correctly computes a state where R is TRUE* **if** *the loop terminates.*

### Total correctness

Notice that in the example in the last unit we proved that **if** the loop completes, **then** we can conclude that it computes the correct value. But how do we know it completes? For this example we can informally reason

- Initially $k = 0$.
- Each time the loop body is executed, $k$ is incremented.
- The loop only continues to execute as long as $k < n$.

Hence, eventually the loop must terminate.

Let us formalize this. We define $t$, a function of some or all the variables encountered in the loop, to be the *bound function* for the loop if $t$ satisfies

- $P_{\rm inv} \wedge (G_0 \vee \cdots \vee G_{K-1}) \Rightarrow (t \geq 0)$; and
- $\{P_{\rm inv} \wedge G_i\}\ t' := t; S_i\ \{t < t'\}$, for $i = 0, \ldots, K-1$.

The first condition says that the bound function is bounded below by zero. The second condition means that each time through the loop $t$ decreases in value.

The discussion leads us to the **checklist for a do loop** in Figure 2.9.

### 2.6.2 Desirable properties of a language ☛ to edX

We keep repeating this, but let's once again consider

$$\text{wp}(\text{“}S\text{”}, R)$$

and its interpretation:

> $\text{wp}(\text{“}S\text{”}, R)$ denotes the weakest predicate so that execution of $S$ started in a state that satisfies this predicate is guaranteed to terminate in a finite amount of time in a state that satisfies $R$.

Another way of saying this is

> $\text{wp}(\text{“}S\text{”}, R)$ denotes the set of states so that if the execution of $S$ is started in any of these states, it is guaranteed to terminate in a finite amount of time in a state such that $R$ is TRUE.

Try to internalize these interpretations.

We will now reason that for a language with reasonable semantics, commands in that language should obey the following properties. Afterwards, we will take these as axioms.

**Law of Excluded Miracle.** What if the predicate $R$ is the state described by $F$ (FALSE)? Let's plug this into the second interpretation:

> $\text{wp}(\text{“}S\text{”}, F)$ denotes the set of states so that if the execution of $S$ is started in any of these states, it is guaranteed to terminate in a finite amount of time in a state such that FALSE is TRUE.

Now, obviously there is no state that has this property. The predicate that describes "no states" is $F$ (FALSE). We conclude that

$$\text{wp}(\text{“}S\text{”}, F) = F$$

for all reasonably defined commands $S$. This is known as the *Law of Excluded Miracle*.

**Law of Distributivity of Conjunction.** Next, let us consider an arbitrary command $S$ and postconditions $Q$ and $R$. Then

$$\text{wp}(\text{“}S\text{”}, Q) \wedge \text{wp}(\text{“}S\text{”}, R) = \text{wp}(\text{“}S\text{”}, Q \wedge R)$$

Why is this?

First, notice that we use $=$ in the above statement because we think of wp as a function that transforms input to output, and hence the output of the left-hand side equals the output of the right-hand side. But we can also think of the left-hand side and the right-hand side as a predicate that describes a set of states, in which case the above becomes

$$\text{wp}(\text{“}S\text{”}, Q) \wedge \text{wp}(\text{“}S\text{”}, R) \Leftrightarrow \text{wp}(\text{“}S\text{”}, Q \wedge R).$$

To prove this to be a tautology, we need to show that an arbitrary state in the set described by $\text{wp}(\text{“}S\text{”}, Q) \wedge \text{wp}(\text{“}S\text{”}, R)$ is also a state in the set described by $\text{wp}(\text{“}S\text{”}, Q \wedge R)$, and visa versa.

$\underline{\text{wp}(\text{“}S\text{”}, Q) \wedge \text{wp}(\text{“}S\text{”}, R) \Rightarrow \text{wp}(\text{“}S\text{”}, Q \wedge R)}$**:** If $s$ satisfies $\text{wp}(\text{“}S\text{”}, Q) \wedge \text{wp}(\text{“}S\text{”}, R)$ then it satisfies $\text{wp}(\text{“}S\text{”}, Q)$ and hence it has the property that if $S$ is executed with state $s$ then it will complete (in a finite amount of time) in a state where $Q$ is true. Similarly, $s$ also satisfies $\text{wp}(\text{“}S\text{”}, R)$ and hence it has the property that if $S$ is executed with state $s$ then it will complete (in a finite amount of time) in a state where $R$ is true. Thus, $s$ has the property that if $S$ is executed with state $s$ then it will complete (in a finite amount of time) in a state where $Q$ and $R$ are true. This shows that $s$ also satisfies $\text{wp}(\text{“}S\text{”}, Q \wedge R)$.

$\underline{\text{wp}(\text{“}S\text{”}, Q) \wedge \text{wp}(\text{“}S\text{”}, R) \Leftarrow \text{wp}(\text{“}S\text{”}, Q \wedge R)}$**:** If $s$ satisfies $\text{wp}(\text{“}S\text{”}, Q \wedge R)$ then it has the property that if $S$ is executed with state $s$ then it will complete (in a finite amount of time) in a state where $Q \wedge R$ is true. But that means it completes (in a finite amount of time) in a state where $Q$ is true and hence $s$ also satisfies $\text{wp}(\text{“}S\text{”}, Q)$. Similarly, we can argue that it also satisfies $\text{wp}(\text{“}S\text{”}, R)$. We conclude that it satisfies $\text{wp}(\text{“}S\text{”}, Q) \wedge \text{wp}(\text{“}S\text{”}, R)$.

**Law of Monotonicity.** The Law of Monotonicity is given by

> If $Q \Rightarrow R$ then $\text{wp}(\text{“}S\text{”}, Q) \Rightarrow \text{wp}(\text{“}S\text{”}, R)$.

Here is the way we will reason that a statement $S$ in a reasonable language has this property.

- The definition of wp means the following Hoare triple (annotated code segment) evaluates to TRUE (is correct):

  $\{\text{wp}(\text{“}S\text{”}, Q)\}$
  $S$
  $\{Q\}$

- The fact that $Q \Rightarrow R$ means that the following annotated code segment is also correct:

  $\{\text{wp}(\text{“}S\text{”}, Q)\}$
  $S$
  $\{Q\}$
  $\{R\}$

- Hence the Hoare triple

$\{\text{wp}(\text{“}S\text{”}, Q)\}$
$S$
$\{R\}$

evaluates to TRUE.

- But a Hoare triple $\{P\}S\{R\}$ only evaluates to TRUE if its precondition, $P$, implies the weakest precondition $\text{wp}(\text{“}S\text{”}, R)$.
- Hence $\text{wp}(\text{“}S\text{”}, Q) \Rightarrow \text{wp}(\text{“}S\text{”}, R)$.

**Law of Distributivity of Disjunction.** Finally, we discuss Distributivity of Disjunction

$$(\text{wp}(\text{“}S\text{”}, Q) \vee \text{wp}(\text{“}S\text{”}, R)) \Rightarrow \text{wp}(\text{“}S\text{”}, Q \vee R)$$

The following exercise prepares us for the reasoning behind this axiom:

**Homework 2.6.2.1** Prove that $((p \Rightarrow r) \wedge (q \Rightarrow r)) \Leftrightarrow ((p \vee q) \Rightarrow r)$.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

Here is the way we will reason that a statement $S$ in a reasonable language obeys Distributivity of Disjunction:

- The definition of wp means the following Hoare triple (annotated code segment) evaluates to TRUE (is correct):

  $\{\text{wp}(\text{“}S\text{”}, Q)\}$
  $S$
  $\{Q\}$

- We know from the Weakening/Strengthening Laws that $Q \Rightarrow Q \vee R$ and hence

  $\{\text{wp}(\text{“}S\text{”}, Q)\}$
  $S$
  $\{Q\}$
  $\{Q \vee R\}$

- Hence we conclude that the Hoare triple

  $\{\text{wp}(\text{“}S\text{”}, Q)\}$
  $S$
  $\{Q \vee R\}$

  evaluates to $T$.

- But a Hoare triple only evaluates to TRUE if its precondition implies the weakest precondition.
- Hence,
  $$\text{wp}(\text{“}S\text{”}, Q) \Rightarrow \text{wp}(\text{“}S\text{”}, Q \vee R).$$
- Similarly, we can conclude that
  $$\text{wp}(\text{“}S\text{”}, R) \Rightarrow \text{wp}(\text{“}S\text{”}, Q \vee R).$$
- In other words,
  $$(\text{wp}(\text{“}S\text{”}, Q) \Rightarrow \text{wp}(\text{“}S\text{”}, Q \vee R)) \wedge (\text{wp}(\text{“}S\text{”}, R) \Rightarrow \text{wp}(\text{“}S\text{”}, Q \vee R))$$
- By the last homework, this is equivalent to
  $$(\text{wp}(\text{“}S\text{”}, Q) \vee \text{wp}(\text{“}S\text{”}, R) \Rightarrow \text{wp}(\text{“}S\text{”}, Q \vee R)).$$

**Homework 2.6.2.2** Prove that the **skip** command satisfies the Laws of Excluded Miracle, Distributivity of Conjunction, Monotonicity, and Distributed Disjunction.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 2.6.2.3** Prove that the **abort** command satisfies the Laws of Excluded Miracle, Distributivity of Conjunction, Monotonicity, and Distributed Disjunction.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 2.6.2.4** Prove that the composition of commands satisfies the Laws of Excluded Miracle, Distributivity of Conjunction, Monotonicity, and Distributed Disjunction.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 2.6.2.5** Prove that the **if** statement

```
if
    $G_0 \rightarrow S_0$
▯   $G_1 \rightarrow S_1$
fi
```

satisfies the Laws of Excluded Micacle, Distributivity of Conjunction, Monotonicity, and Distributed Disjunction.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 2.6.3 A conversation with Sir Tony Hoare ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

## 2.7 Wrap Up

### 2.7.1 Additional exercises ☛ to edX

**Homework 2.7.1.1** Consider an array $b$ of size $n$ with $0 \leq n$, a scalar variable $s$, and the code segment

| |
|---|
| $\{\ Q:\ (s = (\sum i \mid 0 \leq i < k : b(i))) \wedge (0 \leq k < n)\ \}$ |
| $S_0:\ s := s + b(k)$ |
| $S_1:\ k := k + 1$ |
| $\{\ R:\ (s = (\sum i \mid 0 \leq i < k : b(i))) \wedge (0 \leq k \leq n)\ \}$ |

which may be part of a program that sums the entries in array $b$. Prove this code segment correct.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

**Homework 2.7.1.2** Consider an array $b$ with $n$ elements ($0 \leq n$), a scalar variable $s$, and the code segment

$$\{Q:\ (s = (\sum i \mid 0 \leq i < k : b(i)) \wedge (0 \leq k < n))\}$$
$$S:\ s, k := s + b(k), k + 1$$
$$\{R:\ (s = (\sum i \mid 0 \leq i < k : b(i)) \wedge (0 \leq k \leq n))\}$$

This code segment may be part of a program that sums the entries in array $b$. Prove this code segment correct.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

**Homework 2.7.1.3** Prove the correctness of the following code segment. It swaps the contents of $b(i)$ and $b(j)$. You may skip checking if the expressions being assigned are valid (since they clearly are).

| |
|---|
| $\{\ Q: (\forall k \mid 0 \leq k < n : b(k) = \widehat{b}(k)) \wedge (0 \leq i < n) \wedge (0 \leq j < n)\ \}$ |
| $t := b(i)$ |
| $b(i) := b(j)$ |
| $b(j) := t$ |
| $\left\{ \begin{array}{l} R:\ (\forall k \mid (0 \leq k < n) \wedge (k \neq i) \wedge (k \neq j) : b(k) = \widehat{b}(k)) \\ \qquad \wedge (b(i) = \widehat{b}(j)) \wedge (b(j) = \widehat{b}(i)) \wedge (0 \leq i < n) \wedge (0 \leq j < n) \end{array} \right\}$ |

☛ SEE ANSWER

☛ DO EXERCISE ON edX

**Homework 2.7.1.4** Prove the following code segment correct:

```
$\{(\forall j | 0 \leq j < i : m \geq b(j))\}$
**if**
   $b(i) \leq m \rightarrow$ **skip**
$\square$ $b(i) \geq m \rightarrow m := b(i)$
**fi**
$i := i + 1$
$\{(\forall j | 0 \leq j < i : m \geq b(j))\}$
```

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 2.7.1.5** The greatest common divisor (gcd) of two positive integers, $x$ and $y$, is defined to be the largest integer $k$ that evenly divides both $x$ and $y$. Let $\text{gcd}(x,y)$ be the function that returns this integer. A property of this function is that if $x < y$ then $\text{gcd}(x, y-x) = \text{gcd}(x,y)$ and if $y < x$ then $\text{gcd}(x-y, y) = \text{gcd}(x,y)$. Obviously, if $x = y$ then $x = y = \text{gcd}(x,y)$.
Prove the partial correctness of the following program for computing $\text{gcd}(x,y)$, returning the result in updated variables $x$ and $y$. (If you feel energetic, prove complete correctness!)

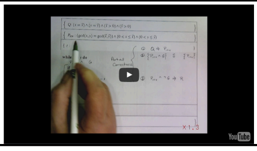
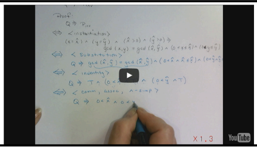
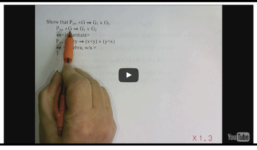
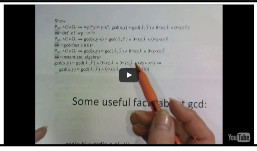
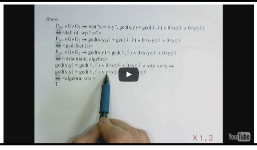
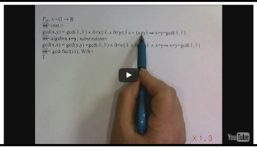
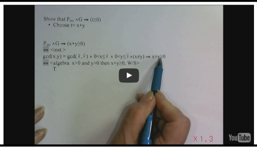
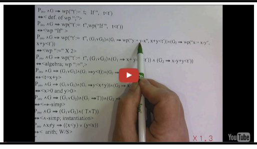
```
$\{ Q: (x = \widehat{x}) \wedge (y = \widehat{y}) \wedge (\widehat{x} > 0) \wedge (\widehat{y} > 0) \}$
$\{ P_{\text{inv}}: (\text{gcd}(x,y) = \text{gcd}(\widehat{x},\widehat{y})) \wedge (0 < x \leq \widehat{x}) \wedge (0 < y \leq \widehat{y}) \}$
$\{ t: \text{abs}(x+y) \}$
**while** $x \neq y$ **do**
   **if**
      $x < y \longrightarrow y := y - x$
   $\square$ $y < x \longrightarrow x := x - y$
   **fi**
**endwhile**
$\{ x = y = \text{gcd}(\widehat{x},\widehat{y}) \}$
```

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Videos of the answer (9 videos)**

Examples

☛ Watch Video on edX
☛ Watch Video on YouTube

| | | |
|---|---|---|
| Overview |  | ☛ Watch Video on edX<br>☛ Watch Video on YouTube |
| $Q \Rightarrow P_{\rm inv}$ |  | ☛ Watch Video on edX<br>☛ Watch Video on YouTube |
| $P_{\rm inv} \wedge G \Rightarrow B_0 \vee B_1$ |  | ☛ Watch Video on edX<br>☛ Watch Video on YouTube |
| $P_{\rm inv} \wedge G \wedge B_0$<br>$\Rightarrow \mathrm{wp}(“S_0”, P_{\rm inv})$ |  | ☛ Watch Video on edX<br>☛ Watch Video on YouTube |
| $P_{\rm inv} \wedge G \wedge B_1$<br>$\Rightarrow \mathrm{wp}(“S_1”, P_{\rm inv})$ |  | ☛ Watch Video on edX<br>☛ Watch Video on YouTube |
| $P_{\rm inv} \wedge \neg G \Rightarrow R$ |  | ☛ Watch Video on edX<br>☛ Watch Video on YouTube |
| Bound function is bounded below |  | ☛ Watch Video on edX<br>☛ Watch Video on YouTube |
| Bound function decreases |  | ☛ Watch Video on edX<br>☛ Watch Video on YouTube |

### 2.7.2 Summary ☛ to edX

#### Hoare triple

Given predicates $Q$ and $R$ and command $S$, the Hoare triple

$$\{Q\}S\{R\}$$

*holds* (evaluates to TRUE) if the command $S$ when started in a state for which $Q$ evaluates to TRUE completes *in a finite amount of time* in a state for which $R$ evaluates to TRUE. For this triple, $Q$ is the precondition and $R$ is the postcondition.

#### Weakest precondition

Given command $S$ and postcondition $R$, the *weakest precondition* (which describes the set of all states for which execution of command $S$ completes in a finite amount of time is a state described by $R$) is given by $\text{wp}(\text{“}S\text{”}, R)$.

#### Proving a program segment correct

The Hoare triple $\{Q\}S\{R\}$ holds (evaluates to TRUE) if and only if $Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)$.

In other words, the annotated program $\{Q\}S\{R\}$ is correct if and only if the Hoare triple $\{Q\}S\{R\}$ holds, and if and only if $Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)$.

#### The skip command

$$\text{wp}(\text{“}\textbf{skip}\text{”}, R) = R.$$

#### The abort command

$$\text{wp}(\text{“}\textbf{abort}\text{”}, R) = F.$$

#### Textual substitution

$$R^{x}_{(\mathcal{E})}$$

equals the predicate $R$ with all free occurrences of $x$ replaced by $(\mathcal{E})$.

#### Simple assignment

$$\text{wp}(\text{“}x := \mathcal{E}\text{”}, R) = \text{valid}(\mathcal{E}) \wedge R^{x}_{(\mathcal{E})}.$$

#### Composition

$$\text{wp}(\text{“}S_0; S_1\text{”}, R) = \text{wp}(\text{“}S_0\text{”}, \text{wp}(\text{“}S_1\text{”}, R)).$$

#### Simultaneous assignment

$$\text{wp}(\text{“}x, y := \mathcal{E}_{\S}, \mathcal{E}_{\dagger}\text{”}, R) = \text{valid}(\mathcal{E}) \wedge R^{x,y}_{(\mathcal{E}_{\S}, \mathcal{E}_{y})}.$$

**Assignment to an array element**

$(b;i:\mathcal{E})$ equals the array $b$ with the element indexed by $i$ replaced with the result of evaluating the expression $\mathcal{E}$.

$$\mathrm{wp}(\text{“}b(i)=\mathcal{E}\text{”},R)=\mathrm{wp}(\text{“}b=(b;i:\mathcal{E})\text{”},R)=R^{b}_{(b;i:\mathcal{E})}$$

**The If command**

```
if
  $G_0 \rightarrow S_0$
▯ $G_1 \rightarrow S_1$
    $\vdots$
▯ $G_{k-1} \rightarrow S_{K-1}$
fi
```

$$\begin{array}{rcll}
\mathrm{wp}(\text{“}\mathbf{if}\text{”},R)= & \overbrace{(G_0\vee G_1\vee\cdots\vee G_{K-1})}^{\text{at least one guard must evaluate to TRUE}} & \wedge & \overbrace{(G_0\Rightarrow \mathrm{wp}(\text{“}S_0\text{”},R))}^{\text{if } G_i \text{ then } S_i \text{ may execute, in which case it must start in a state that guarantees completion in a state where } R \text{ holds.}} \\
 & & \wedge & (G_1\Rightarrow \mathrm{wp}(\text{“}S_1\text{”},R)) \\
 & & \vdots & \vdots \\
 & & \wedge & (G_{K-1}\Rightarrow \mathrm{wp}(S_{K-1},R)).
\end{array} \tag{2.3}$$

**If Theorem**

The annotated code segment

```
{$Q$}
if
▯  $G_0 \rightarrow S_0$
  $\vdots$
▯  $G_{K-1} \rightarrow S_{K-1}$
fi
{$Q$}
```

is correct if and only if

- $Q\Rightarrow(\exists i|0\le i<K:G_i)$ and
- $(\forall i|0\le i<K:Q\wedge G_i\Rightarrow \mathrm{wp}(S_i,R))$

In other words, under these conditions $Q\Rightarrow\mathrm{wp}(\text{“}\mathbf{if}\text{”},R)$.

**Recipe for checking correctness of if command**

| |
|---|
| $\left\{ \; Q : \right.$ $\left. \right\}$ |
| **if** |
| $\{Q \Rightarrow G_0 \vee G_1 :$ $\}$ |
| $G_0 \rightarrow$ |
| $\{Q \wedge G_0 :$ $\}$ |
| $\left\{ \; Q \wedge G_0 \;\; \Rightarrow \;\; \text{wp}(\text{“}S_0\text{”}, R) \; ? \right.$ $\left. \right\}$ |
| $\{\text{wp}(\text{“}S_0\text{”}, R) :$ $\}$ |
| $S_0 :$ |
| $\{R :$ $\}$ |
| $\square$ $G_1 \rightarrow$ |
| $\{Q \wedge G_1 :$ $\}$ |
| $\left\{ \; Q \wedge G_1 \;\; \Rightarrow \;\; \text{wp}(\text{“}S_1\text{”}, R) \; ? \right.$ $\left. \right\}$ |
| $\{\text{wp}(\text{“}S_1\text{”}, R) :$ $\}$ |
| $S_1 :$ |
| $\{R :$ $\}$ |
| **fi** |
| $\left\{ \; R : \right.$ $\left. \right\}$ |

**if-then-else command**

```
if $G$
  $S_0$
else
  $S_1$
fi
```

### Recipe for checking correctness of if-then-else command

| |
|---|
| $\{\ Q : \qquad \}$ |
| **if** $G$ |
| $\{ G \wedge Q : \qquad \}$ |
| $\{ G \wedge Q \Rightarrow \mathrm{wp}(\text{“}S_0\text{”}, R)? \qquad \}$ |
| $\{ \mathrm{wp}(\text{“}S_0\text{”}, R) : \qquad \}$ |
| $S_0$ : |
| $\{ R : \qquad \}$ |
| **else** |
| $\{ \neg G \wedge Q : \qquad \}$ |
| $\{ \neg G \wedge Q \Rightarrow \mathrm{wp}(\text{“}S_1\text{”}, R)? \qquad \}$ |
| $\{ \mathrm{wp}(\text{“}S_1\text{”}, R) : \qquad \}$ |
| $S_1$ : |
| $\{ R : \qquad \}$ |
| **fi** |
| $\{\ R : \qquad \}$ |

### The while command

```
while $G$ do
    $S$
endwhile
```

**The While Theorem**

| | |
|---|---|
| $\{Q\}$<br>$\{P_{\rm inv}\}$<br>**while** $G$ **do**<br>    $\{P_{\rm inv} \wedge G\}$<br>    $S$<br>    $\{P_{\rm inv}\}$<br>**endwhile**<br>$\{P_{\rm inv} \wedge \neg G\}$<br>$\{R\}$ | Partial correctness:<br>• $Q \Rightarrow P_{\rm inv}$;<br>• $\{P_{\rm inv} \wedge G\} S \{P_{\rm inv}\}$; and<br>• $P_{\rm inv} \wedge \neg G \Rightarrow R$<br>Complete correctness, in addition, for some bound function $t$:<br>• $P_{\rm inv} \wedge G \Rightarrow t \geq 0$ and<br>• $\{P_{\rm inv} \wedge G\} t' := t; S \{t < t'\}$ |

```
{ $Q : 0 \leq n$ }
$s := 0$
{ $s = 0 \wedge 0 \leq n$ }
$k := 0$
{ $P_{\text{inv}} : s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n$ }
while $k < n$ do
    { $P_{\text{inv}} \wedge G : s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge k < n$ }
    $s := s + b(k)$
    { $s = (\sum i \mid 0 \leq i < k+1 : b(i)) \wedge 0 \leq k < n$ }
    $k := k + 1$
    { $P_{\text{inv}} : s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n$ }
endwhile
{ $P_{\text{inv}} \wedge \neg G : s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge \neg(k < n)$ }
{ $R$: $s = (\sum i \mid 0 \leq i < n : b(i))$ }
```

Figure 2.6: Example from the launch.

**Checklist for a while command**. Consider

```
{$Q$}
while G do
   $S$
endwhile
{$R$}
```

with loop invariant $P_{\text{inv}}$ and bound function $t$:

- Show that $P_{\text{inv}}$ holds before the loop execution begins: $Q \Rightarrow P_{\text{inv}}$.
- Show that if $P_{\text{inv}}$ and $G$ are TRUE at the beginning of an iteration, then $P_{\text{inv}}$ is again TRUE at the end of the iteration:
  $\{P_{\text{inv}} \wedge G\} S \{P_{\text{inv}}\}$ or, equivalently, $P_{\text{inv}} \wedge G \Rightarrow \text{wp}(S, P_{\text{inv}})$.
- Show that if $P_{\text{inv}}$ is TRUE and the loop guard is FALSE, then $R$ holds: $P_{\text{inv}} \wedge \neg G \Rightarrow R$.
- To prove that the loop completes, show that for some bound function $t$
  - $P_{\text{inv}} \wedge G \Rightarrow (t \geq 0)$ and
  - $\{P_{\text{inv}} \wedge G\} t' := t; S_i \{t < t'\}$.

We will annotate a typical loop like

```
{$Q$}
{$P_{\text{inv}}$ :< loop invariant >}
{$t$ :< bound function >}
while G do
   $S$
endwhile
{$R$}
```

Figure 2.7: Check list for a **while** command.

$$
\begin{array}{ll}
1. & \{Q\} \\
2. & \{P_{\text{inv}}\} \\
 & \textbf{do} \\
3. & \{P_{\text{inv}}\} \\
4. & \left.\begin{array}{llll}
\hline
G_0 & \rightarrow \{P_{\text{inv}} \wedge G_0\} & S_0 & \{P_{\text{inv}}\} \\
\square\, G_1 & \rightarrow \{P_{\text{inv}} \wedge G_1)\} & S_1 & \{P_{\text{inv}}\} \\
 & \vdots & & \\
\square\, G_{K-1} & \rightarrow \{P_{\text{inv}} \wedge G_{K-1})\} & S_{K-1} & \{P_{\text{inv}}\} \\
\hline
\end{array}\right\} \text{Loop body} \\
5. & \{P_{\text{inv}}\} \\
 & \textbf{od} \\
6. & \{P_{\text{inv}}\} \wedge \neg(G_0 \vee \cdots \vee G_{K-1}) \\
7. & \{R\}
\end{array}
$$

Figure 2.8: Annotated **do** loop.

**Checklist for a do loop** of the form

```
$\{Q\}$
**do**
  $G_0 \to S_0$
$\square$ $G_1 \to S_1$
      $\vdots$
$\square$ $G_{K-1} \to S_{K-1}$
**od**
$\{R\}$
```

with loop invariant $P_{\text{inv}}$ and bound function $t$:

- Show that $P_{\text{inv}}$ holds before the loop execution begins.
- Show that $\{P_{\text{inv}} \wedge G_i\} S_i \{P_{\text{inv}}\}$ holds for $i = 0, \dots, K-1$.
- Show that $P_{\text{inv}} \wedge \neg(G_0 \vee \cdots \vee G_{K-1}) \Rightarrow R$ holds.
- Show that $P_{\text{inv}} \wedge (G_0 \vee \cdots \vee G_{K-1}) \Rightarrow (t \geq 0)$.
- Show that $\{P_{\text{inv}} \wedge G_i\} t' := t; S_i \{t < t'\}$ holds for $i = 0, \dots, K-1$.

We will annotate a typical loop like

```
$\{Q\}$
$\{P_{\text{inv}} :<$ loop invariant $>\}$
$\{t :<$ bound function $>\}$
**do**
  $G_0 \to S_0$
$\square$ $G_1 \to S_1$
      $\vdots$
$\square$ $G_{K-1} \to S_{K-1}$
**od**
$\{R\}$
```

Figure 2.9: Check list for **do** command.

Week 3

# Deriving Programs to be Correct

## 3.1 Opening Remarks ☛ to edX

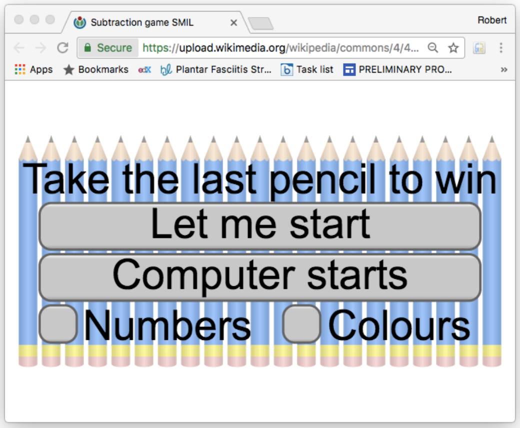
### 3.1.1 Launch ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

To get you in the right mind frame, we challenge you to develop a winning strategy for a very simple version of the ancient game of Nim.

You start with an arbitrary number of pencils. (or sticks or stones or objects). Two players take turns removing up to three pencils. The person who takes the last pencil(s) wins.

We found the following version of Nim online at wikimedia.org:

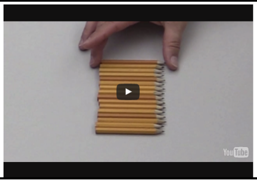

Subtraction game SMIL
Robert
Secure https://upload.wikimedia.org/wikipedia/commons/4/4...
Apps Bookmarks edX Plantar Fasciitis Str... Task list PRELIMINARY PRO...
Take the last pencil to win
Let me start
Computer starts
Numbers
Colours

Play Nim.

Play it, where you make the first move (otherwise, the computer will win!).

- What is the winning strategy?

- How did you figure this out?
- When did you notice that you were going to win or lose?
- Could you figure this out earlier to help guide your moves?
- Did you find that you used a backward analysis?
- How does this relate to goal-oriented programming?
- Can you find a “winning” invariant?



☛ Watch Video on edX
☛ Watch Video on YouTube

**Another algorithm for summing the elements of an array.** The purpose of the first launch in this section, the game of Nim, was to have you notice that sometimes working linearly from beginning to end is not the best way to approach a problem. A goal-oriented approach, by which we mean starting from where you want to end up, leads to an easier development of a solution.

This suggests that maybe the systematic development of a correct program segment should start with the goal: what is to be computed. Here we challenge you to revisit summing the elements in an array. This was also the goal of the simple loop from the launch of Week 2 where we summed from the first to the last entries in the array. This time, instead of giving you the program segment to prove correct, we challenge you to try to derive another algorithm that instead sums the values in the array starting at the last element and ending at the first. This builds on what you learned in Week 2, yet turns the process around. Think of how assertions might guide you toward commands. You will notice that the algorithm you will derive is similar yet not the same algorithm as you saw before, in Week 2, since the invariant is different.

It has been suggested that you may find it helpful to re-watch the video in the launch of Week 2 that explains the previous algorithm. Then you may want to look at this homework side by side with the complete annotated code for that previous algorithm (which is shown at 2:24 of the video).

In the below homework, you may want to consider these questions to help guide you:

- In what order do you insert the components?
- What do you insert first? Is it the loop guard? Why?
- What do you need to know before you find the loop guard?
- What do you need to know to find the initialization?
- What do you need to know to find the command in the loop?
- Can you suggest what might be a good flow to create assertions and commands for determining the code segment?
- Does it require starting at the top and working you way down?
- Did you get it right the first time?
- Before you checked, how confident were you that your pseudo code was correct?

**Homework 3.1.1.1** Consider the problem of summing the elements of array $b$, as outlined in Figure 3.1. Place the following assertions in the correct place (the blank boxes) in the algorithm. Some need to be inserted in multiple places. Some are just there to confuse you (and not be used)

1. $s = (\sum i \mid k \leq i < n : b(i)) \wedge 0 \leq k \leq n$
2. $k < n$
3. $0 \leq k$
4. $0 < k$
5. $s := 0$
6. $k = 0$
7. $k := n$
8. $s := s + b(k)$
9. $s := s + b(k-1)$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

- In what order did you insert the components?
- Did you get it right the first time?
- Before you checked, how confident were you that your pseudo code was correct?

☛ Watch Video on edX
☛ Watch Video on YouTube

```
{ $0 \le n$ }



{ }
while        do
    { }
    $k := k - 1$

    { }
endwhile
{ }
{ $s = (\sum i \mid 0 \le i < n : b(i))$ }
```

Figure 3.1: Partially annotated alternative algorithm for computing the sum of the elements of array *b*.

### 3.1.2 Outline Week 3 ☛ to edX

### 3.1.3 What you will learn ☛ to edX

This week begins our journey to develop programs that are correct. Starting with the goal and annotating the precondition and postcondition, we uncover strategies to find loop-based algorithms by first systematically deriving loop invariants which then guide the development of the rest of the commands in the loop.

Upon completion of this week, you should be able to

- Use Hoare triples and weakest precondition to determine appropriate assignment commands.
- Reason and apply goal-oriented programming to develop short program segments involving skip, assignments, and conditional branching.
- Develop loop-based algorithms using goal-oriented programming techniques given invariants.
- Determine various invariants for loops traversing one-dimensional arrays.
- Implement your first program.

## 3.2 Developing Simple Commands ☛ to edX

### 3.2.1 The skip command ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

Consider the code segment

| | |
|---|---|
| $\{\ Q$ | $\}$ |
| $S$ | |
| $\{\ R$ | $\}$ |

where the purpose of the game is to determine command $S$. If $Q \Rightarrow R$ (in other words, $R$ is weaker than $Q$), then replacing $S$ with the **skip** command makes the code segment correct, since then $Q \Rightarrow \mathrm{wp}(\mathbf{skip}, R)$ is equivalent to $Q \Rightarrow R$, which we assumed evaluates to TRUE.

Consider the following to-be-derived code segment that increases counter $c$ by one if $x = 0$. The precondition indicates that $x \neq 0$. (This may be encountered, for example, as part of an **if** command.)

| | |
|---|---|
| $\{Q : (x \neq 0) \wedge (c = \widehat{c})$ | $\}$ |
| $\{Q \Rightarrow \mathrm{wp}(“S”, R)?$ | $\}$ |
| $\{\mathrm{wp}(“S”, R) :$ | $\}$ |
| $S :$ | |
| $\{R : (x = 0 \wedge c = \widehat{c} + 1) \vee (x \neq 0 \wedge c = \widehat{c})$ | $\}$ |

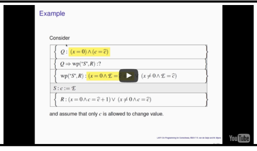
How do we systematically arrive upon the fact that $S$ can be chosen to be the **skip** command? We check whether $Q \Rightarrow R$:

$$\underbrace{(x \neq 0) \wedge (c = \widehat{c})}_{Q} \Rightarrow \underbrace{(x = 0 \wedge c = \widehat{c} + 1) \vee (x \neq 0 \wedge c = \widehat{c})}_{R} \ ?$$

By weakening/strengthening (after commuting the disjunction), this implication is TRUE. Hence, $S$ can be taken to be the **skip** command:

| | |
|---|---|
| $\{Q : (x \neq 0) \wedge (c = \widehat{c})$ | $\}$ |
| $\{Q \Rightarrow \mathrm{wp}(“S”, R)?$ YES! | $\}$ |
| $\{\mathrm{wp}(“S”, R) : (x = 0 \wedge c = \widehat{c} + 1) \vee (x \neq 0 \wedge c = \widehat{c})$ | $\}$ |
| $S :$ **skip** | |
| $\{R : (x = 0 \wedge c = \widehat{c} + 1) \vee (x \neq 0 \wedge c = \widehat{c})$ | $\}$ |

### 3.2.2 Assignment to simple variables ☛ to edX



Example

Consider

| | |
|---|---|
| $\{\ Q : (x = 0) \wedge (c = \widehat{c})$ | $\}$ |
| $\{\ Q \Rightarrow \mathrm{wp}(“S”, R) :?$ | $\}$ |
| $\{\ \mathrm{wp}(“S”, R) : (x = 0 \wedge \mathcal{E} = \widehat{c} + 1) \vee (x \neq 0 \wedge \mathcal{E} = \widehat{c})$ | $\}$ |
| $S : c := \mathcal{E}$ | |
| $\{\ R : (x = 0 \wedge c = \widehat{c} + 1) \vee (x \neq 0 \wedge c = \widehat{c})$ | $\}$ |

and assume that only $c$ is allowed to change value.

☛ Watch Video on edX
☛ Watch Video on YouTube

Consider the following to-be-derived code segment that increases counter $c$ by one if $x = 0$. The precondition indicates that $x = 0$. We assume that $x$ cannot change its value. (Again, this may be encountered, for example, as part of an **if** command.)

| |
|---|
| $\{Q : (x = 0) \wedge (c = \widehat{c}) \}$ |
| $\{Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)? \}$ |
| $\{\text{wp}(\text{“}S\text{”}, R) : \}$ |
| $S :$ |
| $\{R : (x = 0 \wedge c = \widehat{c}+1) \vee (x \neq 0 \wedge c = \widehat{c}) \}$ |

The question is how to systematically determine command $S$.

We first will want to check whether the precondition implies the postcondition, in which case $S$ can (and should) equal **skip**. The unfortunate truth is that it doesn't. We will comment further on this later in this unit.

Next, we notice that the problem specification said that $x$ cannot change its value. By convention, $\widehat{c}$ cannot change value either, because it is a "dummy variable" that is introduced to indicate the "original contents of $c$." Thus, we conclude that an expression, $\mathcal{E}$ must be assigned to $c$:

| |
|---|
| $\{Q : (x = 0) \wedge (c = \widehat{c}) \}$ |
| $\{Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)? \}$ |
| $\{\text{wp}(\text{“}S\text{”}, R) : \}$ |
| $S : c := \mathcal{E}$ |
| $\{R : (x = 0 \wedge c = \widehat{c}+1) \vee (x \neq 0 \wedge c = \widehat{c}) \}$ |

where $\mathcal{E}$ is the expression that is to be determined. Now, we can further annotate this code segment with wp("$c := \mathcal{E}$",$R$)

| |
|---|
| $\{Q : (x = 0) \wedge (c = \widehat{c}) \}$ |
| $\{Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)? \}$ |
| $\left\{ \begin{array}{l} \text{wp}(\text{“}S\text{”}, R) : \text{wp}(\text{“}c := \mathcal{E}\text{”}, (x = 0 \wedge c = \widehat{c}+1) \vee (x \neq 0 \wedge c = \widehat{c})) \\ \Leftrightarrow < \text{ definition of } := \ > \\ \quad (x = 0 \wedge \mathcal{E} = \widehat{c}+1) \vee (x \neq 0 \wedge \mathcal{E} = \widehat{c}) \end{array} \right\}$ |
| $S : c := \mathcal{E}$ |
| $\{R : (x = 0 \wedge c = \widehat{c}+1) \vee (x \neq 0 \wedge c = \widehat{c}) \}$ |

and now we notice that $\mathcal{E}$ must be chosen so that $Q \Rightarrow \text{wp}(\text{“}c := \mathcal{E}\text{”}, R)$:

$$\underbrace{(x = 0) \wedge (c = \widehat{c})}_{Q} \Rightarrow \underbrace{(x = 0 \wedge \mathcal{E} = \widehat{c}+1) \vee (x \neq 0 \wedge \mathcal{E} = \widehat{c})}_{\text{wp}(\text{“}c := \mathcal{E}\text{”}, R)},$$

This guides us to choose $\mathcal{E}$ to equal $c + 1$ since then

$$\underbrace{(x = 0) \wedge (c = \widehat{c})}_{Q} \Rightarrow \underbrace{(x = 0 \wedge c+1 = \widehat{c}+1) \vee (x \neq 0 \wedge c+1 = \widehat{c})}_{\text{wp}(\text{“}c := c+1\text{”}, R)},$$

which we notice is true by weakening/strengthening after subtracting $+1$ on both sides of $(c + 1 = \widehat{c} + 1)$. Hence we conclude that the derived-to-be-correct code segment is given by

| |
|---|
| $\{Q : (x = 0) \wedge (c = \widehat{c}) \}$ |
| $\{Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)?$ YES! $\}$ |
| $\left\{ \begin{array}{l} \text{wp}(\text{“}S\text{”}, R) : \text{wp}(\text{“}c := c+1\text{”}, (x = 0 \wedge c = \widehat{c}+1) \vee (x \neq 0 \wedge c = \widehat{c})) \\ \Leftrightarrow < \text{ definition of } := \; > \\ \quad (x = 0 \wedge c+1 = \widehat{c}+1) \vee (x \neq 0 \wedge c+1 = \widehat{c}) \end{array} \right\}$ |
| $S : c := c+1$ |
| $\{R : (x = 0 \wedge c = \widehat{c}+1) \vee (x \neq 0 \wedge c = \widehat{c}) \}$ |

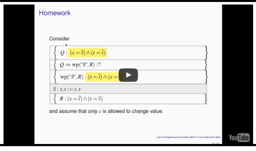
Notice that the precondition and postcondition for the command $S$ prescribe what command $S$ needs to be.

Let us briefly return to the question of how to determine that the **skip** command is *not* a correct choice for command $S$. This is actually the wrong question. The right question is “What if we had tried to derive a command of the form $c := \mathcal{E}$ for $S$ in the previous unit?” The answer is that the expression $\mathcal{E}$ would have turned out to equal $c$. In other words, $S$ would have been the assignment $c := c$. But instead of assigning $c$ to $c$, we might as well then insert the **skip** command.

**Homework 3.2.2.1** Systematically derive the expressions $\mathcal{E}_0$ and $\mathcal{E}_1$ that make the following code segment correct:

| |
|---|
| $\{ \; Q : (s = \widehat{s}) \wedge (t = \widehat{t}) \; \}$ |
| $s, t := \mathcal{E}_0, \mathcal{E}_1$ |
| $\{ \; R : (s = \widehat{t}) \wedge (t = \widehat{s}) \; \}$ |

☛ SEE ANSWER
☛ DO EXERCISE ON edX


Homework

Consider

and assume that only $c$ is allowed to change value.

☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 3.2.2.2** Find the missing assignment to make the following program segment correct. It may be part of a program that sets variable fac equal to $(n-1)! = (n-1) \times (n-2) \times \cdots \times 2 \times 1$. **(Check your results!)**

```
{ Q : $0 < n$ }
$i, \text{fac} := n, ?$
{ R : $1 \leq i \leq n \wedge \text{fac} = (\prod j \mid i \leq j < n : j)$ }
```

a) $i, \text{fac} := n, 0$

b) $i, \text{fac} := n, 1$

c) $i, \text{fac} := n, n$

d) cannot be made to be correct

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 3.2.2.3** Find the missing assignment to make the following program segment correct. It may be part of a program that coverts a binary representations (stored in array $b$) into a decimal number stored in $y$. **(Check your results!)**

```
{ Q : $y = (\sum j \mid i \leq j < n : b(j) \times 2^j)$ }
$y := ?$
$i := i - 1$
{ R : $y = (\sum j \mid i \leq j < n : b(j) \times 2^j)$ }
```

a) $y := b(i) \times 2^i + y$

b) $y := b(i-1) \times 2^{i-1} + y$

c) $y := b(i+1) \times 2^{i+1} + y$

d) cannot be made to be correct

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 3.2.3 Careful! ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

Consider the code segment

| |
|---|
| $\{\ T \qquad \}$ |
| $k := k+1$ |
| $y := \mathcal{E}$ |
| $\{\ y = x(k) \qquad \}$ |

where $\mathcal{E}$ is an expression to be determined so that the Hoare triple holds. Let us annotate:

| |
|---|
| $\{\ T \qquad \}$ |
| $\{\ \text{wp}(\text{“}k := k+1; y := \mathcal{E}\text{”}, y = x(k)) : \mathcal{E} = x(k+1) \qquad \}$ |
| $k := k+1$ |
| $\{\ \text{wp}(\text{“}y := \mathcal{E}\text{”}, y = x(k)) : \mathcal{E} = x(k) \qquad \}$ |
| $y := \mathcal{E}$ |
| $\{\ y = x(k) \qquad \}$ |

We would conclude that we should choose expression $\mathcal{E}$ to equal $x(k+1)$ because then

$$T \Rightarrow x(k+1) = x(k+1).$$

However, if we then insert $y := x(k+1)$ we notice that

| |
|---|
| $\{\ T \qquad \}$ |
| $\{\ \text{wp}(\text{“}k := k+1; y := x(k+1)\text{”}, y = x(k)) : x(k+2) = x(k+1) \qquad \}$ |
| $k := k+1$ |
| $\{\ \text{wp}(\text{“}y := \mathcal{E}\text{”}, y = x(k)) : x(k+1) = x(k) \qquad \}$ |
| $y := x(k+1)$ |
| $\{\ y = x(k) \qquad \}$ |

which is NOT a correct code segment because

$$T \Rightarrow x(k+2) = x(k+1)$$

evaluates to TRUE only if $x(k+2) = x(k+1)$.

What went wrong? The problem is that $k$ appears in the expression $\mathcal{E}$, and $k$ is changed prior to the assignment $y := x(k+1)$. The way around this is to capture that $\mathcal{E}$ is a function of $k$:

| |
|---|
| $\{\ T \qquad \}$ |
| $k := k+1$ |
| $y := \mathcal{E}(k)$ |
| $\{\ y = x(k) \qquad \}$ |

where $\mathcal{E}(k)$ represents an expression that *may* depend on $k$ and that is to be determined so that the Hoare triple holds. Let us annotate:

| |
|---|
| $\{\ T \qquad \}$ |
| $\{\ \mathrm{wp}(\text{“}k := k+1; y := \mathcal{E}(k)\text{”}, y = x(k)) : \mathcal{E}(k+1) = x(k+1) \quad \}$ |
| $k := k+1$ |
| $\{\ \mathrm{wp}(\text{“}y := \mathcal{E}(k)\text{”}, y = x(k)) : \mathcal{E}(k) = x(k) \quad \}$ |
| $y := \mathcal{E}(k)$ |
| $\{\ y = x(k) \quad \}$ |

We conclude that we should choose $\mathcal{E}(k+1)$ to equal $x(k+1)$ and hence $\mathcal{E}(k)$ as $x(k)$. Now let's check:

| |
|---|
| $\{\ T \qquad \}$ |
| $\{\ \mathrm{wp}(\text{“}k := k+1; y := x(k)\text{”}, y = x(k)) : x(k+1) = x(k+1) \quad \}$ |
| $k := k+1$ |
| $\{\ \mathrm{wp}(\text{“}y := x(k)\text{”}, y = x(k)) : x(k) = x(k) \quad \}$ |
| $y := x(k)$ |
| $\{\ y = x(k) \quad \}$ |

which IS a correct code segment because

$$T \Rightarrow x(k+1) = x(k+1)$$

evaluates to TRUE.

We conclude that given Hoare triple

| |
|---|
| $\{\ Q \qquad \}$ |
| $k := \cdots$ |
| $y := ?$ |
| $\{\ R \qquad \}$ |

where ? is to be determined, one should use $\mathcal{E}(k)$ if in the Hoare triple $k$ appears on the left of an assignment statement prior to the assignment to $y$. This generalizes if there are more assignments that precede the assignment to be determined. Each variable that appears on the left of an assignment prior to the assignment to $y$ should appear as a parameter for expression $\mathcal{E}$.

**Homework 3.2.3.1** Find the missing assignment to make the following program segment correct. **(Check your results!)**

```
{ $Q : 0 < i < n$ }
$i, j := i+1, ?$
{ $R : j = n - i \wedge 0 \leq j < n$ }
```

a) $i, j := i+1, n-i$

b) $i, j := i+1, n-i+1$

c) $i, j := i+1, n-i-1$

d) cannot be made to be correct

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 3.2.3.2** Find the missing assignment to make the following program segment correct. **(Check your results!)**

```
{ $Q : 0 < i < n$ }
$i := i+1$
$j :=?$
{ $R : j = n - i \wedge 0 \leq j < n$ }
```

a) $j := n-i$

b) $j := n-i+1$

c) $j := n-i-1$

d) cannot be made to be correct

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 3.2.3.3** Find the missing assignment to make the following program segment correct. **(Check your results!)**

| |
|---|
| $\{\ Q : 0 < i < n\ \}$ |
| $j := ?$ |
| $i := i + 1$ |
| $\{\ R : j = n - i \wedge 0 \leq j < n\ \}$ |

a) $j := n - i$

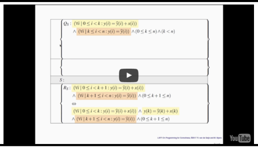
b) $j := n - i + 1$

c) $j := n - i - 1$

d) cannot be made to be correct

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 3.2.4 Assignment to array elements ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Assignment to array elements can in principle be made similarly systematic. It is the manipulation of the precondition and postcondition that becomes a bit trickier. The reason is that, as we noticed last week when proving code segments that involved assignment to arrays correct, "textual substitution" is greatly simplified if the array element being updated does not appear in a quantifier. For this reason, the range of the quantifier is split to expose that particular element.

Let us consider an example where array $x$ is added to array $y$, given in Figure 3.2, where command $S$ is to be determined. (How to get to this point will be discussed in the later section on how to systematically develop a loop.)

Recognizing that $(0 \leq k \leq n) \wedge (k < n)$ is equivalent to $(0 \leq k < n)$ we focus on

| |
|---|
| $\left\{ \begin{array}{l} Q_S : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \\ \quad \wedge\ (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge (0 \leq k < n) \end{array} \right\}$ |
| S: |
| $\left\{ \begin{array}{l} R_S : (\forall i \mid 0 \leq i < k+1 : y(i) = \widehat{y}(i) + x(i)) \\ \quad \wedge\ (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge (0 \leq k+1 \leq n) \end{array} \right\}$ |

In reasoning about what $S$ should be, we notice the difference in ranges in the precondition and postcondition. To reconcile these, we split ranges both in the precondition and the postcondition:

| |
|---|
| $\{\ Q : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le n)\ \}$ |
| $k := 0$ |
| $\{\ P_{\text{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n)\ \}$ |
| **while** $k < n$ **do** |
| $\{\ P_{\text{inv}} \wedge G : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i))$ <br> $\wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n) \wedge (k < n)\ \}$ |
| S: |
| $\{\ \text{wp}(\text{“}k := k+1\text{”}, P_{\text{inv}}) : (\forall i \mid 0 \le i < k+1 : y(i) = \widehat{y}(i) + x(i))$ <br> $\wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k+1 \le n)\ \}$ |
| $k := k+1$ |
| $\{\ P_{\text{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n)\ \}$ |
| **endwhile** |
| $\{\ P_{\text{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n) \wedge \neg(k < n)\ \}$ |
| $\{\ (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i) + x(i))\ \}$ |

Figure 3.2: Add the values in array $x$ to the corresponding values in array $y$. It is implicitly assumed that the values in $x$ do not change. Array $\widehat{y}$ is introduced to be able to refer to the original contents of $y$. It is implicitly assumed that the arrays are of size $n$.

$$\left\{\begin{array}{l} Q_S : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \\ \qquad \wedge \underbrace{y(k) = \widehat{y}(k) \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i))}_{(\forall i \mid k \le i < n : y(i) = \widehat{y}(i))} \wedge (0 \le k < n) \end{array}\right\}$$

S:

$$\left\{\begin{array}{l} R_S : \overbrace{(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i) \wedge (y(k) = \widehat{y}(k) + x(k))}^{(\forall i \mid 0 \le i < k+1 : y(i) = \widehat{y}(i) + x(i))} \\ \qquad \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k+1 \le n) \end{array}\right\}$$

We then notice that it is $y(k)$ that changes value, so that $S$ should have the form

$$S : \; y(k) := \mathcal{E},$$

where $\mathcal{E}$ is some expression. If we also simplify $0 \le k+1 \le n$ we find that

$$\left\{\begin{array}{l} Q_S : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \\ \qquad \wedge (y(k) = \widehat{y}(k)) \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k < n) \end{array}\right\}$$

$S : \; y(k) := \mathcal{E}$

$$\left\{\begin{array}{l} R_S : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \\ \qquad \wedge (y(k) = \widehat{y}(k) + x(k)) \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (-1 \le k < n) \end{array}\right\}$$

Now, we can insert wp(“$y(k) := \mathcal{E}$”,$R_S$):

$$\left\{\begin{array}{l} Q_S : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \\ \qquad \wedge (y(k) = \widehat{y}(k)) \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k < n) \end{array}\right\}$$

$\{Q_S \Rightarrow \text{wp}(\text{“}S\text{”}, R_S)? \}$

$$\left\{\begin{array}{l} \text{wp}(\text{“}y(k) := \mathcal{E}\text{”}, R_S) : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \\ \qquad \wedge (\mathcal{E} = \widehat{y}(k) + x(k)) \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \\ \qquad \wedge (-1 \le k < n) \end{array}\right\}$$

$S : \; y(k) := \mathcal{E}$

$$\left\{\begin{array}{l} R_S : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \\ \qquad \wedge (y(k) = \widehat{y}(k) + x(k)) \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (-1 \le k < n). \end{array}\right\}$$

We notice that if we choose $\mathcal{E}$ to equal $y(k) + x(k)$, then

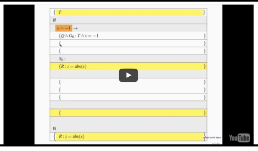
$$\underbrace{y(k) + x(k)}_{\mathcal{E}} = \widehat{y}(k) + x(k)$$

and, after cancelling $x(k)$ on both sides, by weakening/strengthening $Q_S \Rightarrow \text{wp}(\text{“}y(k) := y(k) + x(k)\text{”}, R_S)$ becomes TRUE.

| |
|---|
| $\left\{ Q_S : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (y(k) = \widehat{y}(k)) \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k < n) \right\}$ |
| $\{Q_S \Rightarrow \text{wp}(\text{“}S\text{”}, R_S)$? YES! $\}$ |
| $\left\{ \text{wp}(\text{“}y(k) := y(k) + x(k)\text{”}, R_S) : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (y(k) + x(k) = \widehat{y}(k) + x(k)) \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (-1 \le k < n) \right\}$ |
| $S: \; y(k) := y(k) + x(k)$ |
| $\left\{ R_S : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (y(k) = \widehat{y}(k) + x(k)) \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (-1 \le k < n). \right\}$ |

## 3.3 Developing the if Command ☛ to edX

### 3.3.1 A general strategy ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Consider a prototypical **if** command that has been annotated:

| |
|---|
| $\{ \; Q :$ $\}$ |
| **if** |
| $G_0 \rightarrow$ |
| $\{G_0 \wedge Q :$ $\}$ |
| $\{G_0 \wedge Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, R)$? $\}$ |
| $\{\text{wp}(\text{“}S_0\text{”}, R) :$ $\}$ |
| $S_0 :$ |
| $\{R :$ $\}$ |
| $[\,] \; G_1 \rightarrow$ |
| $\{(G_1 \wedge Q) :$ $\}$ |
| $\{G_1 \wedge Q \Rightarrow \text{wp}(\text{“}S_1\text{”}, R)$? $\}$ |
| $\{\text{wp}(\text{“}S_1\text{”}, R) :$ $\}$ |
| $S_1 :$ |
| $\{R :$ $\}$ |
| $[\,] \; \ldots$ |
| **fi** |
| $\{ \; R :$ $\}$ |

What do we know about the guards? Recall from the If Theorem that it must be the case that

$$Q \Rightarrow G_0 \vee G_1 \vee \cdots \vee G_{K-1},$$

where $K$ equals the number of guarded commands in the **if** command. A straightforward strategy for determining the guards is to keep adding new guards until $Q \Rightarrow G_0 \vee G_1 \vee \cdots \vee G_{K-1}$. Every time a new guard $G_i$ is chosen, we focus on the (hopefully) simpler subproblem of determining command $S_i$ that makes the code segment

| |
|---|
| $\{G_i \wedge Q) :$ $\}$ |
| $\{G_i \wedge Q \Rightarrow \text{wp}(\text{“}S_i\text{”}, R)?$ $\}$ |
| $\{\text{wp}(\text{“}S_i\text{”}, R) :$ $\}$ |
| $S_i :$ |
| $\{R :$ $\}$ |

correct.

Let us illustrate this with the following example that sets variable $z$ to the absolute value of $x$, where for now we assume you understand the intuitive meaning of the absolute value function, abs(). Later we will more precisely define it and see how this helps us find guards even more systematically.

Let's guess a loop guard, $G_0 : x = -1$ and place it in the **if** command recipe. After developing command $S_0$, we would arrive at

| |
|---|
| $\{\ T$ $\}$ |
| **if** |
| $x = -1 \rightarrow$ |
| $\{Q \wedge G_0 : T \wedge x = -1$ $\}$ |
| $\{T \wedge x = -1 \Rightarrow \text{wp}(\text{“}S_0\text{”}, R)?$ YES! $\}$ |
| $\{\text{wp}(\text{“}S_0\text{”}, R) : 1 = \text{abs}(x)$ $\}$ |
| $S_0 : z = 1$ |
| $\{R : z = \text{abs}(x)$ $\}$ |
| **fi** |
| $\{\ R : z = \text{abs}(x)$ $\}$ |

We then check whether $Q \Rightarrow G_0$, which instantiates to $T \Rightarrow x = -1$ and conclude we need more guards. So, we add the guards $G_1 : x = 1$, $G_2 : x = 0$ to arrive at

| |
|---|
| $\{ \; T \quad \}$ |
| **if** |
| $x = -1 \rightarrow$ |
| $\{Q \wedge G_0 : T \wedge x = -1 \quad \}$ |
| $\{T \wedge x = -1 \Rightarrow \text{wp}(\text{“}S_0\text{”}, R)?$ YES! $\}$ |
| $\{\text{wp}(\text{“}S_0\text{”}, R) : 1 = \text{abs}(x) \quad \}$ |
| $S_0 : z = 1$ |
| $\{R : z = \text{abs}(x) \quad \}$ |
| $x = 1 \rightarrow$ |
| $\{Q \wedge G_1 : T \wedge x = 1 \quad \}$ |
| $\{T \wedge x = 1 \Rightarrow \text{wp}(\text{“}S_1\text{”}, R)?$ YES! $\}$ |
| $\{\text{wp}(\text{“}S_1\text{”}, R) : 1 = \text{abs}(x) \quad \}$ |
| $S_1 : z = 1$ |
| $\{R : z = \text{abs}(x) \quad \}$ |
| $x = 0 \rightarrow$ |
| $\{Q \wedge G_2 : T \wedge x = 0 \quad \}$ |
| $\{T \wedge x = 0 \Rightarrow \text{wp}(\text{“}S_2\text{”}, R)?$ YES! $\}$ |
| $\{\text{wp}(\text{“}S_2\text{”}, R) : 0 = \text{abs}(x) \quad \}$ |
| $S_2 : z = 0$ |
| $\{R : z = \text{abs}(x) \quad \}$ |
| **fi** |
| $\left\{ \; R : z = \text{abs}(x) \quad \right\}$ |

Now, obviously we would have to create an infinite number of guards if we kept going this way. We need to be smarter.

The key is to be more explicit about expressing the postcondition $R : z = \text{abs}(x)$. Instead, let's use the definition of the absolute value:

- If $x$ is greater than or equal to zero, $\text{abs}(x) = x$.
- If $x$ is less than or equal to zero, $\text{abs}(x) = -x$.

This can be expressed as the predicate

$$R : (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x).$$

Notice that it is fine to have the conditions $x \geq 0$ and $x \leq 0$ overlap, as long as the **if** command is derived to be correct. We also notice that the guards are easily identified in the postcondition: $x \geq 0$ and $x \leq 0$.

After deriving $S_0$ and $S_1$ using the techniques from earlier this week, the **if** command recipe yields

| |
|---|
| $\{\ T \qquad \}$ |
| **if** |
| $x \geq 0 \rightarrow$ |
| $\{Q \wedge G_0 : T \wedge x \geq 0 \qquad \}$ |
| $\{T \wedge \geq 0 \Rightarrow \text{wp}(\text{“}S_0\text{”}, R)\text{? YES!} \qquad \}$ |
| $\{\text{wp}(\text{“}S_0\text{”}, R) : (x \geq 0 \wedge x = x) \vee (x \leq 0 \wedge x = -x) \qquad \}$ |
| $S_0 : z = x$ |
| $\{R : (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x) \qquad \}$ |
| $x \leq 0 \rightarrow$ |
| $\{Q \wedge G_0 : T \wedge x \leq 0 \qquad \}$ |
| $\{T \wedge \geq 0 \Rightarrow \text{wp}(\text{“}S_1\text{”}, R)\text{? YES!} \qquad \}$ |
| $\{\text{wp}(\text{“}S_1\text{”}, R) : (x \geq 0 \wedge -x = x) \vee (x \leq 0 \wedge -x = -x) \qquad \}$ |
| $S_1 : z = -x$ |
| $\{R : (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x) \qquad \}$ |
| **fi** |
| $\{\ R : (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x) \qquad \}$ |

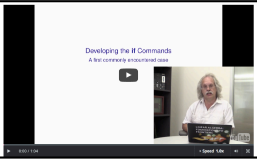
The details of how to systematicallly derive $S_i$ for the code segment, given $Q$, $G_i$, and $R$:

| |
|---|
| $\{G_i \wedge Q) \qquad \}$ |
| $\{G_i \wedge Q \Rightarrow \text{wp}(\text{“}S_i\text{”}, R)? \qquad \}$ |
| $\{\text{wp}(\text{“}S_i\text{”}, R) : \qquad \}$ |
| $S_i$ |
| $\{R \qquad \}$ |

build upon what we know about how to derive simple commands. Still, it quickly becomes messy when the postcondition is nontrivial. For this reason, we don't focus on the general case in this course, treating a simpler (but common) case instead, in the next unit.

One strategy, obviously, is to guess and to then check correctness. This is a perfectly reasonable strategy. Often, the annotations guide one towards an educated guess.

### 3.3.2 A commonly encountered case ☛ to edX



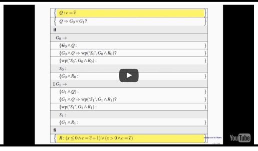
☛ Watch Video on edX
☛ Watch Video on YouTube

The postcondition for an **if** command can often be chosen or manipulated into a convenient format that makes the identification of the guards and the derivation of the guarded commands simpler. In this unit we discuss one such case.

**Homework 3.3.2.1** Identify for each of the operations on the left the corresponding predicate that best expresses it on the right.

| | |
|---|---|
| 1. $z = \text{abs}(x)$ | a. $(x \le 0 \wedge c = \widehat{c}+1) \vee (x > 0 \wedge c = \widehat{c})$ |
| 2. $z = \min(x,y)$ | b. $(x \le y \wedge z = y) \vee (x \ge y \wedge z = x)$ |
| 3. $z = \max(x,y)$ | c. $(x \le y \wedge z = x) \vee (x \ge y \wedge z = y)$ |
| 4. $z = \text{abs}(x-y)$ | d. $(x \ge 0 \wedge z = x) \vee (x \le 0 \wedge z = -x)$ |
| 5. Increment $c$ by one if $x \le 0$ | e. $(x \ge y \wedge z = x-y) \vee (y \ge x \wedge z = y-x)$ |

☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

What the examples from the last homework have in common is that the predicate that describes the operation to be computed can be written as a disjunction of conjunctions: $R$ has the form

$$(G_0 \wedge R_0) \vee (G_1 \wedge R_1) \vee \cdots \vee (G_{K-1} \wedge R_{K-1}).$$

Now, each component in the disjunction is by itself stronger than $R$:

$$(G_j \wedge R_j) \Rightarrow (G_0 \wedge R_0) \vee (G_1 \wedge R_1) \vee \cdots \vee (G_{K-1} \wedge R_{K-1})$$

by weakening/strengthening. What this suggests is the specialized worksheet for this case given in Figure 3.3. Deriving each guarded command $S_j$ now comes down to deriving the code segments

| |
|---|
| $\{G_j \wedge Q :$ $\}$ |
| $\{G_j \wedge Q \Rightarrow \text{wp}(\text{“}S_j\text{”}, G_j \wedge R_j)?$ $\}$ |
| $\{\text{wp}(\text{“}S_j\text{”}, G_j \wedge R_j) :$ $\}$ |
| $S_j$ |
| $\{G_j \wedge R_j :$ $\}$ |

in that worksheet.

**Example 3.1** *Consider the Hoare triple*

| |
|---|
| $\left\{ \; Q : c = \widehat{c} \right\}$ $\}$ |
| $S :$ |
| $\left\{ \; R : (x \le 0 \wedge c = \widehat{c}+1) \vee (x > 0 \wedge c = \widehat{c}) \right.$ $\}$ |

*for the command $S$ that increments $c$ if $x \le 0$. Here, as usual, $\widehat{c}$ is a “dummy variable” introduced to denote the contents of $c$ at the beginning of the code segment. We assume that the contents of variable $x$ will not be changed.*

*We use the insights in this unit to derive $S$. We verify that*

$$(\underbrace{x \le 0}_{G_0} \wedge \underbrace{c = \widehat{c}+1}_{R_0}) \vee (\underbrace{x > 0}_{G_1} \wedge \underbrace{c = \widehat{c}}_{R_1}).$$

| |
|---|
| $\left\{ Q : \right\}$ |
| $\left\{ Q \Rightarrow G_0 \vee G_1 \vee \cdots \vee G_{K-1}? \right\}$ |
| **if** |
| $G_0 \rightarrow$ |
| $\{G_0 \wedge Q : \}$ |
| $\{G_0 \wedge Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0)? \}$ |
| $\{\text{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0) : \}$ |
| $S_0 :$ |
| $\{G_0 \wedge R_0 : \}$ |
| $\square\, G_1 \rightarrow$ |
| $\{G_1 \wedge Q) : \}$ |
| $\{G_1 \wedge Q \Rightarrow \text{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1)? \}$ |
| $\{\text{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1) : \}$ |
| $S_1$ |
| $\{G_1 \wedge R_1 : \}$ |
| $\square \cdots$ |
| $\{\}$ |
| $\square\, G_{K-1} \rightarrow$ |
| $\{G_{K-1} \wedge Q) : \}$ |
| $\{G_{K-1} \wedge Q \Rightarrow \text{wp}(\text{“}S_{K-1}\text{”}, G_{K-1} \wedge R_{K-1})? \}$ |
| $\{\text{wp}(\text{“}S_{K-1}\text{”}, G_{K-1} \wedge R_{K-1}) : \}$ |
| $S_{K-1} :$ |
| $\{G_{K-1} \wedge R_{K-1} : \}$ |
| **fi** |
| $\left\{ R : (G_0 \wedge R_0) \vee (G_1 \wedge R_1) \vee \cdots \vee (G_{K-1} \wedge R_{K-1}) \right\}$ |

Figure 3.3: Worksheet for the case where $R$ has the form $(G_0 \wedge R_0) \vee (G_1 \wedge R_1) \vee \cdots \vee (G_{K-1} \wedge R_{K-1})$.

| |
|---|
| $\left\{ \; Q : c = \widehat{c} \right.$ $\}$ |
| $\left\{ \; Q \Rightarrow x \leq 0 \vee x > 0\text{? YES!} \right.$ $\}$ |
| **if** |
| $x \leq 0 \rightarrow$ |
| $\{ G_0 \wedge Q : x \leq 0 \wedge c = \widehat{c}$ $\}$ |
| $\{ G_0 \wedge Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0)?$ $\}$ |
| $\{ \text{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0) : x \leq 0 \wedge \mathcal{E}_0 = \widehat{c} + 1$ $\}$ |
| $S_0 : c = \mathcal{E}_0$ |
| $\{ G_0 \wedge R_0 : x \leq 0 \wedge c = \widehat{c} + 1$ $\}$ |
| ▯$x > 0 \rightarrow$ |
| $\{ G_1 \wedge Q : x > 0 \wedge c = \widehat{c}$ $\}$ |
| $\{ G_1 \wedge Q \Rightarrow \text{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1)?$ $\}$ |
| $\{ \text{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1) : x > 0 \wedge \mathcal{E}_1 = \widehat{c}$ $\}$ |
| $S_1 : c = \mathcal{E}_1$ |
| $\{ G_1 \wedge R_1 : x > 0 \wedge c = \widehat{c}$ $\}$ |
| **fi** |
| $\left\{ \; R : (x \leq 0 \wedge c = \widehat{c} + 1) \vee (x > 0 \wedge c = \widehat{c}) \right.$ $\}$ |

Figure 3.4: Partially completed worksheet for computing $(x \leq 0 \wedge c = \widehat{c} + 1) \vee (x > 0 \wedge c = \widehat{c})$.

| |
|---|
| $\{\ Q : c = \widehat{c}\ \}$ |
| $\{\ Q \Rightarrow x \leq 0 \vee x > 0$? YES! $\}$ |
| **if** |
| $x \leq 0 \rightarrow$ |
| $\{G_0 \wedge Q : x \leq 0 \wedge c = \widehat{c}\ \}$ |
| $\{G_0 \wedge Q \Rightarrow \mathrm{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0)$? YES! $\}$ |
| $\{\mathrm{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0) : x \leq 0 \wedge c + 1 = \widehat{c} + 1\ \}$ |
| $S_0 : c = c + 1$ |
| $\{G_0 \wedge R_0 : x \leq 0 \wedge c = \widehat{c} + 1\ \}$ |
| $\square\, x > 0 \rightarrow$ |
| $\{G_1 \wedge Q : x > 0 \wedge c = \widehat{c}\ \}$ |
| $\{G_1 \wedge Q \Rightarrow \mathrm{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1)$? YES! $\}$ |
| $\{\mathrm{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1) : x > 0 \wedge c = \widehat{c}\ \}$ |
| $S_1$ : **skip** |
| $\{G_1 \wedge R_1 : x > 0 \wedge c = \widehat{c}\ \}$ |
| **fi** |
| $\{\ R : (x \leq 0 \wedge c = \widehat{c} + 1) \vee (x > 0 \wedge c = \widehat{c})\ \}$ |

Figure 3.5: Completed worksheet for computing $(x \leq 0 \wedge c = \widehat{c} + 1) \vee (x > 0 \wedge c = \widehat{c})$.

*Thus, the postcondition has the desired format. Also $Q \rightarrow G_0 \vee G_1$ since $G_0 \vee G_1 \Leftrightarrow T$. This allows us to fill out many of the expressions in the worksheet, yielding Figure 3.4. There, we also indicate that $S_0$ and $S_1$ update $c$ with expressions $\mathcal{E}_0$ and $\mathcal{E}_1$, respectively. From the highlighted fields we deduce that $\mathcal{E}_0$ should equal $c+1$ (so that $S_0$ equals $c := c+1$) and that $\mathcal{E}_1$ should equal $c$ (so that $S_1$ should equal $c := c$ or, equivalently,* **skip***). This is summarized in Figure 3.5.*

**Homework 3.3.2.2** Use Figure 3.6 to develop a code segment that computes $z = \min(x, y)$:

|   |
|---|
| $\{\ Q : T \}$ |
| $S$ |
| $\{\ R : (x \leq y \wedge z = x) \vee (x \geq y \wedge z = y) \}$ |

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 3.3.2.3** Use Figure 3.6 to develop a code segment that computes $z = \text{abs}(x - y)$:

|   |
|---|
| $\{\ Q : T \}$ |
| $S$ |
| $\{\ R : (x \geq y \wedge z = x - y) \vee (y \geq x \wedge z = y - x) \}$ |

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 3.4 Developing a While Command ☛ to edX

### 3.4.1 A worksheet for the while command ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

In Week 2 Section 2.5.1, we discussed how to prove a **while** loop correct. Let us now turn to how to *derive* a typical **while** loop.

Consider the prototypical loop

where command $S_I$ initializes variables (we call this the initialization step), $G$ is the loop guard, and $S$ is the loop body.

The key is to turn the proof of correctness for a loop into a worksheet much like we created a worksheet for the **if** command. This worksheet was hinted at in the launch for this week and is given by

| |
|---|
| $\{\ Q :$ $\}$ |
| $\{\ Q \Rightarrow G_0 \vee G_1?$ $\}$ |
| **if** |
| $G_0 \rightarrow$ |
| $\{G_0 \wedge Q :$ $\}$ |
| $\{G_0 \wedge Q \Rightarrow \mathrm{wp}(``S_0", G_0 \wedge R_0)?$ $\}$ |
| $\{\mathrm{wp}(``S_0", G_0 \wedge R_0) :$ $\}$ |
| $S_0 :$ |
| $\{G_0 \wedge R_0 :$ $\}$ |
| $\square\, G_1 \rightarrow$ |
| $\{G_1 \wedge Q) :$ $\}$ |
| $\{G_1 \wedge Q \Rightarrow \mathrm{wp}(``S_1", G_1 \wedge R_1)?$ $\}$ |
| $\{\mathrm{wp}(``S_1", G_1 \wedge R_1) :$ $\}$ |
| $S_1 :$ |
| $\{G_1 \wedge R_1 :$ $\}$ |
| **fi** |
| $\{\ R :$ $\}$ |

Figure 3.6: Worksheet for the case where $R$ has the form $(G_0 \wedge R_0) \vee (G_1 \wedge R_1)$.

```
$S_I$
**while** $G$ **do**
    $S$
**endwhile**
```

| |
|---|
| $\{ \; Q: \hfill \}$ |
| $S_I$ |
| $\{ \; P_{\rm inv} \hfill \}$ |
| **while** $\; G \;$ **do** |
| $\quad \{ P_{\rm inv} \; \wedge \; G \hfill \}$ |
| $\quad \{ {\rm wp}(\text{"}S\text{"}, P_{\rm inv}) \hfill \}$ |
| $\quad S$ |
| $\quad \{ P_{\rm inv} \hfill \}$ |
| **endwhile** |
| $\{ \; P_{\rm inv} \; \wedge \; \neg( \; G \; ) \hfill \}$ |
| $\{ \; R \hfill \}$ |

Now, *if* we can *a priori* determine what $P_{\rm inv}$ is, then everything else in this loop is predetermined:

- We know that $(P_{\rm inv} \wedge \neg G) \Rightarrow R$ must be *true*. Thus, given $P_{\rm inv}$ and $R$ we must determine a condition $G$ that makes this *true*. In other words, to determine $G$ we focus on

| |
|---|
| $\{ \; P_{\rm inv} \; \wedge \; \neg( \; G \; ) \hfill \}$ |
| $\{ \; R \hfill \}$ |

  A careful reexamination of how to prove this *given* $P_{\rm inv}$, $R$, *and* $G$ in Week 2 tells us that weakening/strengthening laws are usually provide insight into how to choose $G$.

- Given $Q$ and $P_{\rm inv}$ we can deploy techniques discussed earlier in this week to derive the initialization command $S_I$. For this, we focus on deriving $S_I$ so that

| |
|---|
| $\{ \; Q: \hfill \}$ |
| $S_I$ |
| $\{ \; P_{\rm inv} \hfill \}$ |

  holds. This is a smaller subproblem and hopefully easier to tackle.

- Similarly, given $P_{\rm inv}$ and $G$ we can deploy techniques previously discussed in this week to derive command $S$. For this, we focus on deriving $S$ so that

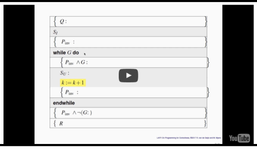
| |
|---|
| $\{ \; P_{\rm inv} \wedge G: \hfill \}$ |
| $S$ |
| $\{ \; P_{\rm inv} \hfill \}$ |

holds. Now, the naive solution would be to choose $S$ : **skip** since

| |
|---|
| $\{ \; P_{\rm inv} \wedge G : \;\}$ |
| $S$ : **skip** |
| $\{ \; P_{\rm inv} \;\}$ |

can be easily shown to always hold. However, we need to make progress towards completion since otherwise the loop will never stop. It is this that forces $S$ to be a more complicated command.

## 3.4.2 Progress towards completion ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

In this first part of the course, we mainly focus on algorithms that traverse one dimensional arrays in a systematic way: either from the first element to the last or from the last element to the first. Which of these two cases applies can be determined from the loop invariant. Because of this restriction, we can refine the worksheet further. Assuming the variable that keeps track of where in the array we are is given by $k$, if the algorithm traverses from the first element to the last, the worksheet becomes

| |
|---|
| $\{ \; Q \;\}$ |
| $S_I$ |
| $\{ \; P_{\rm inv} \;\}$ |
| **while** $G$ **do** |
| $\quad \{ P_{\rm inv} \wedge G \}$ |
| $\quad \{ \mathrm{wp}(\text{“}S_U; k := k+1\text{”}, P_{\rm inv}) \}$ |
| $\quad S_U$ |
| $\quad \{ \mathrm{wp}(\text{“}k := k+1\text{”}, P_{\rm inv}) \}$ |
| $\quad k := k+1$ |
| $\quad \{ P_{\rm inv} \}$ |
| **endwhile** |
| $\{ \; P_{\rm inv} \wedge \neg( G ) \;\}$ |
| $\{ \; R \;\}$ |

or

| |
|---|
| $\{ \; Q \; \}$ |
| $S_I$ |
| $\{ \; P_{\mathrm{inv}} \; \}$ |
| **while** $G$ **do** |
| $\quad \{ P_{\mathrm{inv}} \; \wedge \; G \; \}$ |
| $\quad \{ \mathrm{wp}(\text{“}k := k+1; S_U\text{”}, P_{\mathrm{inv}}) \; \}$ |
| $\quad k := k+1$ |
| $\quad \{ \mathrm{wp}(\text{“}S_U\text{”}, P_{\mathrm{inv}}) \; \}$ |
| $\quad S_U$ |
| $\quad \{ P_{\mathrm{inv}} \; \}$ |
| **endwhile** |
| $\{ \; P_{\mathrm{inv}} \; \wedge \; \neg( \; G \; ) \; \}$ |
| $\{ \; R \; \}$ |

Here $S_U$ is the command that *updates* the pertinent variables. If on the other hand the algorithm traverses the array from last to first (as in the example in the launch), the worksheet is given by

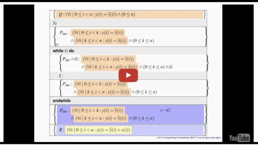
| |
|---|
| $\{ \; Q \; \}$ |
| $S_I$ |
| $\{ \; P_{\mathrm{inv}} \; \}$ |
| **while** $G$ **do** |
| $\quad \{ P_{\mathrm{inv}} \; \wedge \; G \; \}$ |
| $\quad \{ \mathrm{wp}(\text{“}S_U; k := k-1\text{”}, P_{\mathrm{inv}}) \; \}$ |
| $\quad S_U$ |
| $\quad \{ \mathrm{wp}(\text{“}k := k-1\text{”}, P_{\mathrm{inv}}) \; \}$ |
| $\quad k := k-1$ |
| $\quad \{ P_{\mathrm{inv}} \; \}$ |
| **endwhile** |
| $\{ \; P_{\mathrm{inv}} \; \wedge \; \neg( \; G \; ) \; \}$ |
| $\{ \; R \; \}$ |

or

| |
|---|
| $\{\ Q \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}$ |
| $S_I$ |
| $\{\ P_{\rm inv} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}$ |
| **while** $G$ **do** |
| $\quad \{P_{\rm inv} \ \wedge \ G \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}$ |
| $\quad \{{\rm wp}(\text{“}k := k-1; S_U\text{”}, P_{\rm inv}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}$ |
| $\quad k := k-1$ |
| $\quad \{{\rm wp}(\text{“}S_U\text{”}, P_{\rm inv}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}$ |
| $\quad S_U$ |
| $\quad \{P_{\rm inv} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}$ |
| **endwhile** |
| $\{\ P_{\rm inv} \ \wedge \ \neg(\ G\ ) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}$ |
| $\{\ R \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}$ |

Notice that if $k$ is part of the loop guard, it can be chosen to be part of the bound function $t$ that is used to prove complete correctness, which then guarantees completion of the loop. For this reason, we don't bother with proving complete correctness, since we know loops of the above structure will complete.

### 3.4.3 *A priori* determination of loop invariants ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

The insights in the last two units lead us to the conclusion that we need to systematically derive $P_{\rm inv}$ from precondition $Q$ and postcondition $R$. If we can do that, then we have a systematic way of deriving loop-based algorithms since the other parts of the loop with systematically fall in place.

Let us go back and consider a loop that adds the contents of array $x$ to those in array $y$ in Figure 3.2, but let's pretend we don't know what the loop should be. Let's choose to progress through the arrays from first to last element. We will implicitly assume that the arrays are of size $n$ with $0 \leq n$. The precondition then becomes

$$Q : \ (\forall i \mid 0 \leq i < n : y(i) = \widehat{y}(i)) \ \wedge 0 \leq n$$

while the postcondition is given by

$$R : \ (\forall i \mid 0 \leq i < n : y(i) = \widehat{y}(i) + x(i)) \ .$$

It is from these two expressions that we now want to derive possible loop invariants.

In the worksheets discussed in the last unit, $k$ becomes the index that keeps track of where in the arrays we are. It tells us what parts of the array have been processed. To describe this with a predicate, we take quantifiers in the precondition and postcondition, and split the ranges using this index: the precondition

$$Q : \ (\forall i \mid 0 \leq i < n : y(i) = \widehat{y}(i)) \ \wedge 0 \leq n$$

is split to yield

$$(\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i)) \ \wedge \ (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \ \wedge 0 \leq k \leq n$$

and the postcondition

$$R: \ (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i) + x(i))$$

is split to yield

$$(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n.$$

This **systematic** splitting of the ranges is the first step.

Now, as computation proceeds, we should make progress towards the result, which is given by the postcondition. How can we construct a predicate that describes such progress? In other words, how can we extract a loop invariant from the above information? **A loop invariant should capture the partial progress towards the final result.**

Let us consider possibilities derived from the precondition and postcondition, calling these candidates for loop invariants (Candidates A through D):

| Candidate | Loop invariant | Invariant |
|---|---|---|
| A | $(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n)$ | |
| B | $(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) \wedge (0 \le k \le n))$ | 1 |
| C | $(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge (0 \le k \le n)$ | 2 |
| D | $(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge (0 \le k \le n))$ | |

Notice:

- Candidate A describes a loop invariant that says that as the loop progresses, $y$ always contains its original contents. Therefore, upon completing the loop, it will still contain its original contents. Obviously there is no loop guard such that $P_{\text{inv}} \wedge \neg G \Rightarrow R$. It is not a valid loop invariant for this computation.

- Candidate B describes a loop invariant that says that as the loop progresses, the first $k$ elements of $y$ have been updated with the final result, while the last $n-k$ elements have not yet been updated. We will later argue that there is a $G$ such that $P_{\text{inv}} \wedge \neg G \Rightarrow R$ and there is an initialization $S_I$ such that $\{Q\}S_I\{P_{\text{inv}}\}$ holds. It will result in a correct algorithm that we will call *(algorithmic) Variant 1*, corresponding to the given *(loop) Invariant 1*.

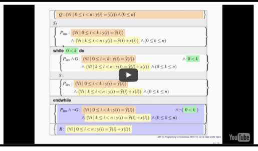
- Candidate C describes a loop invariant that says that as the loop progresses, the last $n-k$ elements of $y$ have been updated with the final result, while the first $k$ elements have not yet been updated. Again, we will later argue that there is a $G$ such that $P_{\text{inv}} \wedge \neg G \Rightarrow R$ and there is an initialization $S_I$ such that $\{Q\}S_I\{P_{\text{inv}}\}$ holds. It will result in a correct algorithm that we will call Variant 2, corresponding to the given Invariant 2.

- Candidate D describes a loop invariant that says that as the loop progresses, $y$ contains its final contents. Obviously, this means that before the loop starts, $y$ must already have been initialized with the final contents. Given that we intend for $S_I$ to only perform simple initializations, it is not a valid loop invariant for this computation.

We have **systematically** identified two viable loop invariants!

### 3.4.4 Deriving the loop guard and initialization command ☛ to edX

IMPORTANT. In the videos of Units 3.4.4 and 3.4.5, we discuss how to derive the loop for the case where the algorithm marches through the vectors from the last element to the first element. We consider this to be the case corresponding to (Loop) Invariant 2. At the end, in Homework 3.4.5.1, we have an exercise on the edX platform that asks you to derive the algorithm that marches through the vectors from first to last (Invariant 1).
Here in the book in the same units, we discuss how to derive the loop for the case where the algorithm marches through the vectors from the first element to the last element. In other words, Units 3.4.4 and 3.4.5 of the book are a very thorough answer to Homework 3.4.5.1.
We suggest that you watch the videos, do Homework 3.4.5.1 in the online version of Unit 3.4.5, and then go through the materials in the book. Alternatively, go through Units 3.4.4 and 3.4.5 in the book, do Homework 3.4.5.1 in the book, and then go back and watch the videos.



☛ Watch Video on edX
☛ Watch Video on YouTube

The discussion in the last unit left us with two viable loop invariants for updating an array $y$ by adding the elements of $x$ to each of its elements. Let us pick Invariant 1 and use it to derive what remains of the loop. For now, we only know the precondition $Q$, the postcondition $R$, and the loop invariant $P_{\text{inv}}$, as shown in Figure 3.7. What is left to be determined are the loop guard $G$, the initialization command $S_I$, how to update the loop index $k$, and the *update* command $S_U$.

We start by **systematically** determining the loop guard. What we know is that

$$P_{\text{inv}} \wedge \neg G : \; (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \wedge \neg(G)$$

must imply

$$R : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i) + x(i)).$$

By examination, we deduce that choosing $G$ as $k < n$ has the desired property because then $P_{\text{inv}} \wedge \neg G$ becomes

$$(\forall i \mid 0 \le i < \; n \; : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid \; n \; \le i < n : y(i) = \widehat{y}(i)) \wedge \; k = n,$$

which implies the postcondition by weakening/strengthening.

Next, we **systematically** derive the initialization, which must make the Hoare triple

$$\left\{ \; Q : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le n \; \right\}$$
$$\left\{ \; \text{wp}(\text{“}S_I\text{”}, P_{\text{inv}}) : \; \right\}$$
$$S_I :$$
$$\left\{ \; P_{\text{inv}} : \; (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \; \right\}$$

hold. Notice that if $S_I$ is chosen to set $k := 0$, then the first quantifier in the loop invariant has an empty range and equals TRUE, meaning that the precondition implies that $P_{\text{inv}}$ holds. More precisely:

| |
|---|
| $\left\{ Q : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le n \right\}$ |
| $\left\{ \mathrm{wp}(``S_I", P_{\mathrm{inv}}) : \right\}$ |
| $S_I :$ |
| $\left\{ P_{\mathrm{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \right\}$ |
| **while** $G$ **do** |
| $\left\{ P_{\mathrm{inv}} \wedge G : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \quad \wedge \quad G \right\}$ |
| $\left\{ \mathrm{wp}(``S_U; k := \mathcal{E}(k)", P_{\mathrm{inv}}) : \right\}$ |
| $S_U :$ |
| $\left\{ \mathrm{wp}(``k := \mathcal{E}(k)", P_{\mathrm{inv}}) : \right\}$ |
| $k := \mathcal{E}(k)$ |
| $\left\{ P_{\mathrm{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \right\}$ |
| **endwhile** |
| $\left\{ P_{\mathrm{inv}} \wedge \neg G : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \wedge \neg(G) \right\}$ |
| $\left\{ R : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i) + x(i)) \right\}$ |

Figure 3.7: Recipe for adding vector $x$ to vector $y$ after filling in Invariant 1.

$$\left\{ Q: (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le n \right\}$$

$$\left\{ \begin{array}{l} \mathrm{wp}(``S_I", P_{\mathrm{inv}}): \overbrace{(\forall i \mid 0 \le i < 0 : y(i) = \widehat{y}(i) + x(i))}^{T} \\ \qquad\qquad\qquad \wedge\ (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le 0 \le n \end{array} \right\}$$

$$S_I : k := 0$$

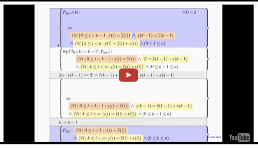
$$\left\{ P_{\mathrm{inv}}: (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \right\}$$

holds.

Finally, we notice that initially $k = 0$ and eventually $k = n$. Thus, it makes sense to increment $k := k + 1$ in the loop body. With these insights, the worksheet is further filled out as illustrated in Figure 3.8.

### 3.4.5 Deriving the loop body ☛ to edX

IMPORTANT. In the videos of Units 3.4.4 and 3.4.5, we discuss how to derive the loop for the case where the algorithm marches through the vectors from the last element to the first element. We consider this to be the case corresponding to (Loop) Invariant 2. At the end, in Homework 3.4.5.1, we have an exercise on the edX platform that asks you to derive the algorithm that marches through the vectors from first to last (Invariant 1).
Here in the book in the same units, we discuss how to derive the loop for the case where the algorithm marches through the vectors from the first element to the last element. In other words, Units 3.4.4 and 3.4.5 of the book are a very thorough answer to Homework 3.4.5.1.
We suggest that you watch the videos, do Homework 3.4.5.1 in the online version of Unit 3.4.5, and then go through the materials in the book. Alternatively, go through Units 3.4.4 and 3.4.5 in the book, do Homework 3.4.5.1 in the book, and then go back and watch the videos.



☛ Watch Video on edX
☛ Watch Video on YouTube

We now focus on the loop body:

$$\left\{ P_{\mathrm{inv}} \wedge G: (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \wedge k < n \right\}$$

$$\{ \mathrm{wp}(``S_U; k := k + 1", P_{\mathrm{inv}}): \}$$

$$S_U :$$

$$\{ \mathrm{wp}(``k := k + 1", P_{\mathrm{inv}}): \}$$

$$k := k + 1$$

$$\left\{ P_{\mathrm{inv}}: (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \right\}$$

The insight we have is that the range of the first quantifier in $P_{\mathrm{inv}}$ expanded by one element in the current iteration, and it is that element that is updated. For this reason, we split the range of that quantifier:

$\{\ Q : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le n\ \}$

$\left\{ \begin{array}{l} \mathrm{wp}(\text{“}S_I\text{”}, P_{\rm inv}) : \quad (\forall i \mid 0 \le i < 0 : y(i) = \widehat{y}(i) + x(i)) \\ \quad \wedge\ (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le 0 \le n \end{array} \right\}$

$S_I : k := 0$

$\left\{ \begin{array}{l} P_{\rm inv} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \\ \quad \wedge 0 \le k \le n \end{array} \right\}$

**while** $k < n$ **do**

$\left\{ \begin{array}{l} P_{\rm inv} \wedge G : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \quad \wedge \quad k < n \\ \quad \wedge 0 \le k \le n \end{array} \right\}$

$\left\{ \mathrm{wp}(\text{“}S_U; k := k+1\text{”}, P_{\rm inv}) : \right\}$

$S_U :$

$\left\{ \mathrm{wp}(\text{“}k := k+1\text{”}, P_{\rm inv}) : \right\}$

$k := k + 1$

$\left\{ \begin{array}{l} P_{\rm inv} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \\ \quad \wedge 0 \le k \le n \end{array} \right\}$

**endwhile**

$\{\ P_{\rm inv} \wedge \neg G : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid n \le i < n : y(i) = \widehat{y}(i)) \wedge k = n\ \}$

$\{\ R : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i) + x(i))\ \}$

Figure 3.8: Recipe for adding vector $x$ to vector $y$ after determining Invariant 1, the loop guard $G$, the initialization command $S_I$, and the update of the loop index $k := k + 1$.

| |
|---|
| $\left\{ P_{\mathrm{inv}} \wedge G : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n \wedge k < n \right\}$ |
| $\{\mathrm{wp}(\text{“}S_U; k := k+1\text{”}, P_{\mathrm{inv}}) : \}$ |
| $S_U$ : |
| $\{\mathrm{wp}(\text{“}k := k+1\text{”}, P_{\mathrm{inv}}) : \}$ |
| $k := k+1$ |
| $\left\{ \begin{array}{l} P_{\mathrm{inv}} : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n \\ \quad = (\forall i \mid 0 \leq i < k-1 : y(i) = \widehat{y}(i) + x(i)) \wedge y(k-1) = \widehat{y}(k-1) + x(k-1) \\ \qquad \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n. \end{array} \right\}$ |

We can then compute

$$\begin{array}{rl} \mathrm{wp}(\text{“}k := k+1\text{”}, P_{\mathrm{inv}}) = & (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge y(k) = \widehat{y}(k) + x(k) \\ & \wedge (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k+1 \leq n. \end{array}$$

We enter this into the loop body:

| |
|---|
| $\left\{ P_{\mathrm{inv}} \wedge G : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n \wedge k < n \right\}$ |
| $\{\mathrm{wp}(\text{“}S_U; k := k+1\text{”}, P_{\mathrm{inv}}) : \}$ |
| $S_U$ : |
| $\left\{ \begin{array}{l} \mathrm{wp}(\text{“}k := k+1\text{”}, P_{\mathrm{inv}}) : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge y(k) = \widehat{y}(k) + x(k) \\ \qquad \wedge (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k+1 \leq n. \end{array} \right\}$ |
| $k := k+1$ |
| $\left\{ \begin{array}{l} P_{\mathrm{inv}} : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n \\ \quad = (\forall i \mid 0 \leq i < k-1 : y(i) = \widehat{y}(i) + x(i)) \wedge y(k-1) = \widehat{y}(k-1) + x(k-1) \\ \qquad \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n. \end{array} \right\}$ |

Similarly, we can rewrite $P_{\mathrm{inv}} \wedge G$ to expose the same term in the quantifier:

$$(\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n \wedge k < n$$

becomes

$$\begin{array}{rl} & (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge y(k) = \widehat{y}(k) \\ \wedge & (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k < n, \end{array}$$

which we can enter in the derivation of the loop body:

$$\left\{\begin{array}{l} P_{\rm inv} \wedge G : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n \wedge k < n \\ \quad = (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge y(k) = \widehat{y}(k) \\ \qquad \wedge (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k < n. \end{array}\right\}$$

$$\{ {\rm wp}(\text{“}S_U ; k := k+1\text{”}, P_{\rm inv}) : \qquad \}$$

$$S_U :$$

$$\left\{\begin{array}{l} {\rm wp}(\text{“}k := k+1\text{”}, P_{\rm inv}) : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge y(k) = \widehat{y}(k) + x(k) \\ \qquad \wedge (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k+1 \leq n. \end{array}\right\}$$

$$k := k+1$$

$$\left\{\begin{array}{l} P_{\rm inv} : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n \\ \quad = (\forall i \mid 0 \leq i < k-1 : y(i) = \widehat{y}(i) + x(i)) \wedge y(k-1) = \widehat{y}(k-1) + x(k-1) \\ \qquad \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n. \end{array}\right\}$$

It now becomes obvious that $y(k)$ must be updated as part of update command $S_U$:

$$\left\{\begin{array}{l} P_{\rm inv} \wedge G : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n \wedge k < n \\ \quad = (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge y(k) = \widehat{y}(k) \\ \qquad \wedge (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k < n. \end{array}\right\}$$

$$\{ {\rm wp}(\text{“}S_U ; k := k+1\text{”}, P_{\rm inv}) : \qquad \}$$

$$S_U : y(k) := \mathcal{E}$$

$$\left\{\begin{array}{l} {\rm wp}(\text{“}k := k+1\text{”}, P_{\rm inv}) : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge y(k) = \widehat{y}(k) + x(k) \\ \qquad \wedge (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k+1 \leq n. \end{array}\right\}$$

where $\mathcal{E}$ is an expression that is to be determined. We then compute ${\rm wp}(\text{“}S_U ; k := k+1\text{”}, P_{\rm inv})$:

$$\left\{\begin{array}{l} P_{\rm inv} \wedge G : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k \leq n \wedge k < n \\ \quad = (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge y(k) = \widehat{y}(k) \\ \qquad \wedge (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k < n. \end{array}\right\}$$

$$\left\{\begin{array}{l} {\rm wp}(\text{“}S_U ; k := k+1\text{”}, P_{\rm inv}) : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge \mathcal{E} = \widehat{y}(k) + x(k) \\ \qquad \wedge (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k+1 \leq n. \end{array}\right\}$$

$$S_U : y(k) := \mathcal{E}$$

$$\left\{\begin{array}{l} {\rm wp}(\text{“}k := k+1\text{”}, P_{\rm inv}) : (\forall i \mid 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \wedge y(k) = \widehat{y}(k) + x(k) \\ \qquad \wedge (\forall i \mid k+1 \leq i < n : y(i) = \widehat{y}(i)) \wedge 0 \leq k+1 \leq n. \end{array}\right\}$$

and we notice that $\mathcal{E} = y(k) + x(k)$. The fully annotated worksheet is given in Figure 3.9.

**Homework 3.4.5.1** Derive the algorithm for adding array $x$ to array $y$ corresponding to Invariant 2 using the worksheet in Figure 3.10.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

$$\left\{\ Q : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le n \ \right\}$$

$$\left\{\begin{array}{l} \mathrm{wp}(\text{“}S_I\text{”}, P_{\mathrm{inv}}) : \ (\forall i \mid 0 \le i < 0 : y(i) = \widehat{y}(i) + x(i)) \ \wedge \ (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \\ \qquad \wedge \ 0 \le 0 \le n \end{array}\right\}$$

$S_I : k := 0$

$$\left\{\ P_{\mathrm{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \ \right\}$$

**while** $k < n$ **do**

$$\left\{\begin{array}{l} P_{\mathrm{inv}} \wedge G : \ (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \wedge k < n \\ \quad = \ (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \ \wedge \ y(k) = \widehat{y}(k) \\ \qquad \wedge \ (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \ \wedge \ 0 \le k < n. \end{array}\right\}$$

$$\left\{\begin{array}{l} \mathrm{wp}(\text{“}S_U ; k := k+1\text{”}, P_{\mathrm{inv}}) : \\ \quad (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \ \wedge \ y(k) + x(k) = \widehat{y}(k) + x(k) \\ \quad \wedge \ (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \ \wedge \ 0 \le k+1 \le n. \end{array}\right\}$$

$S_U : y(k) := y(k) + x(k)$

$$\left\{\begin{array}{l} \mathrm{wp}(\text{“}k := k+1\text{”}, P_{\mathrm{inv}}) : \ (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \ \wedge \ y(k) = \widehat{y}(k) + x(k) \\ \qquad \wedge \ (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \ \wedge \ 0 \le k+1 \le n. \end{array}\right\}$$

$k := k + 1$

$$\left\{\begin{array}{l} P_{\mathrm{inv}} : \ (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \\ \quad = \ (\forall i \mid 0 \le i < k-1 : y(i) = \widehat{y}(i) + x(i)) \ \wedge \ y(k-1) = \widehat{y}(k-1) + x(k-1) \\ \qquad \wedge \ (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \ \wedge \ 0 \le k \le n. \end{array}\right\}$$

**endwhile**

$$\left\{\ P_{\mathrm{inv}} \wedge \neg G : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i) + x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le k \le n \wedge \neg(k < n) \ \right\}$$

$$\left\{\ R : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i) + x(i)) \ \right\}$$

Figure 3.9: Completed worksheet for adding vector $x$ to vector $y$ (Variant 1).

| |
|---|
| $\{\; Q : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le n \;\}$ |
| $\left\{\; \text{wp}(\text{“}S_I\text{”}, P_{\text{inv}}) : \;\right\}$ |
| |
| $\{\; P_{\text{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \;\}$ |
| **while** **do** |
| $\left\{\; P_{\text{inv}} \wedge G : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \wedge \;\right\}$ |
| $\left\{\; \text{wp}(\text{“}S_U; k := k-1\text{”}, P_{\text{inv}}) : \;\right\}$ |
| $S_U$ : |
| $\left\{\; \text{wp}(\text{“}k := k-1\text{”}, P_{\text{inv}}) : \;\right\}$ |
| $k := k - 1$ |
| $\{\; P_{\text{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \;\}$ |
| **endwhile** |
| $\{\; P_{\text{inv}} \wedge \neg G : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \wedge \neg(\quad) \;\}$ |
| $\{\; R : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i) + x(i)) \;\}$ |

Figure 3.10: Blank worksheet for adding vector $x$ to vector $y$ (Variant 2).

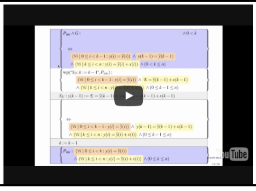
In Summary:

- Given a precondition and postcondition, we can derive loop invariants.
- Given a loop invariant and the postcondition, we can derive the loop guard.
- Given the precondition and a loop invariant, we can derive the initialization command.
- Given a loop invariant and an update to the loop index, we can derive the loop body.

All steps are *prescribed* by the precondition and the postcondition!

## 3.5 Examples ☛ to edX

### 3.5.1 Evaluating a polynomial ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

In this section, we look at a really nice example through a sequence of exercises. It may be a trivial example at first, too similar to previous examples and homeworks. But there is an interesting twist!

This exercise prepares you for the first programming exercise, in the next unit. Since we will use MATLAB to implement the algorithm, we switch to assuming that "indexing starts at one", meaning that the elements of an array $p$ are accessed as $p(1)$, $p(2)$, $\cdots$, $p(n)$.

Now, consider the evaluation of a polynomial of degree $n-1$ with coefficients stored in array $p$ of size $n$:

$$\begin{aligned} y &= p(1) + p(2)x + p(3)x^2 + \cdots + p(n)x^{n-1} \\ &= (\sum i \mid 1 \le i \le n : p(i)x^{i-1}). \end{aligned}$$

Since there are $n$ coefficients, the last term is $p(n)x^{n-1}$ and a typical term is $p(i)x^{i-1}$.

Now, let's do what we did before, and partition the quantifier:

$$\begin{aligned} y &= \underbrace{p(1) + p(2)x + \cdots + p(k-1)x^{k-2}}_{(\sum i \mid 1 \le i < k : p(i)x^{i-1})} + \underbrace{p(k)x^{k-1} + p(k+1)x^k + \cdots + p(n)x^{n-1}}_{(\sum i \mid k \le i \le n : p(i)x^{i-1})} \wedge 1 \le k \le n+1 \\ &= (\sum i \mid 1 \le i < k : p(i)x^{i-1}) + (\sum i \mid k \le i \le n : p(i)x^{i-1}) \wedge 1 \le k \le n+1 \end{aligned}$$

From this, we can derive two obvious loop invariants:

Invariant 1: $y = (\sum i \mid 1 \le i < k : p(i)x^{i-1}) \wedge 1 \le k \le n+1,$

which leads to an (algorithmic) Variant 1 that updates something like

$$\begin{aligned} &y := y + p(k) \times x^{k-1} \\ &k := k+1 \end{aligned}$$

in the loop body and

Invariant 2: $y = (\sum i \mid k \le i \le n : p(i)x^{i-1}) \wedge 1 \le k \le n+1$

which leads to Variant 2 that updates something like

$$k := k-1$$
$$y := y + p(k) \times x^{k-1}$$

in the loop body. (We don't guarantee this is exactly correct. Remember that Dijkstra may be watching over your shoulder, so you would want to derive the update.)

The problem with the two variants mentioned above is that in each iteration of the loop, $x^{k-1}$ must be evaluated, which could be expensive. If you have some experience programming, you may observe that it could be good to have a variable, $z$, that holds $x^{k-1}$ every time through the loop. But we don't just guess at such things. We derive loop invariants. How do we systematically derive these loop invariants? Observe that

$$\begin{aligned} y &= (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) + (\sum i \mid k \leq i \leq n : p(i)x^{i-1}) \wedge 1 \leq k \leq n+1 \\ &= (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) + (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \times x^{k-1} \wedge 1 \leq k \leq n+1 \\ &= (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) + (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \times z \wedge z = x^{k-1} \wedge 1 \leq k \leq n+1 \end{aligned}$$

which then yields two loop invariants

Invariant 3: $y = (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) \wedge z = x^{k-1} \wedge 1 \leq k \leq n+1,$

which leads to Variant 3 that updates something like

$$y := y + p(k) \times z$$
$$k := k+1$$
$$z := z \times x$$

in the loop body and

Invariant 4: $y = (\sum i \mid k \leq i \leq n : p(i)x^{i-1}) \wedge z = x^{k-1} \wedge 1 \leq k \leq n+1,$

which leads to Variant 4 that updates something like

$$k := k-1$$
$$y := y + p(k) \times z$$
$$z := z/x$$

Again, we are doing this "quick and dirty". You will want to derive the details for the algorithms!

Now there is one more invariant hidden in the expression

$$y = (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) + (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \times z \wedge z = x^{k-1} \wedge 1 \leq k \leq n+1,$$

namely

Invariant 5: $y = (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \wedge 1 \leq k \leq n+1.$

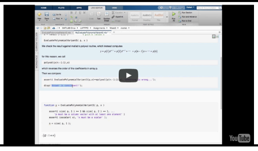
It is this loop invariant that we allude at in the video and that we want you to derive as algorithmic Variant 5 and that you will implement as function `EvaluatePolynomialVariant5`, in the next unit.

### 3.5.2 At last, you write your first code! ☛ to edX

"... in order to drive home the message that this introductory programming course is primarily a course in formal mathematics, we see to it that the programming language in question has not been implemented on campus so that students are protected from the temptation to test their programs." – Edsger W. Dijkstra ☛ EWD1036

In this course, we have yet to program a single line of real code! The reason is that we don't believe one should write programs unless one has been equipped to derive a program to be correct. And now you are ready!

The MATLAB Live Script mentioned in the next video can be found in Unit 3.5.2 on the edX platform. Follow the directions in that unit for uploading it to your MATLAB Online account.


☛ Watch Video on edX
☛ Watch Video on YouTube

In the following exercises, you will develop the loop guard, the initialization, and the loop body of Variant 5 corresponding to Invariant 5

Invariant 5: $y = (\sum i \mid k \le i \le n : p(i)x^{i-k}) \wedge 1 \le k \le n+1.$

for computing the polynomial

$$\begin{aligned} y &= p(1) + p(2)x + p(3)x^2 + \cdots + p(n)x^{n-1} \\ &= (\sum i \mid 1 \le i \le n : p(i)x^{i-1}). \end{aligned}$$

**Homework 3.5.2.1** At the end of the last video, you were asked to derive the loop guard $G$ from

$$\left\{ P_{\text{inv}} \wedge \neg G : y = (\sum i \mid k \le i \le n : p(i)x^{i-k}) \wedge 1 \le k \le n+1 \wedge \neg G \right\}$$

$$\left\{ R : (\sum i \mid 1 \le i \le n : p(i)x^{i-1}) \right\}$$

Indicate which of the following is a correct loop guard $G$ (there may be more than one...)

a) $1 < k$.

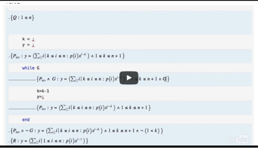
b) $k < n$.

c) $k \neq 1$.

d) $k \neq n$.

Enter a correct loop guard in the Live Script.

☛ SEE ANSWER
☛ DO EXERCISE ON edX


☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 3.5.2.2** At the end of the last video, you were asked to derive the initialization command

$$\begin{aligned} &k = \\ &y = \end{aligned}$$

to make

|  |
|---|
| $\{ \; Q : 0 < n \; \}$ |
| $k :=$ |
| $y :=$ |
| $\{ \; P_{\text{inv}} : y = (\sum i \mid k \leq i \leq n : p(i) x^{i-k}) \wedge 1 \leq k \leq n+1 \; \}$ |

correct. Indicate which of the following is a correct initialization.
(There may be more than one...)

a) $k := n$
   $y := p(n)$

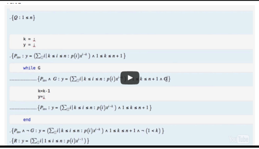
b) $k := n+1$
   $y := 0$

c) $k := n$
   $y := 1$

d) $k := 0$
   $y := 0$

Enter a correct initialization command in the Live Script.

☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 3.5.2.3** At the end of the last video, you were asked to derive the commands in the loop body

$$k = k - 1$$
$$y =$$

to make

| |
|---|
| $\{P_{\text{inv}} \wedge G : y = (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \wedge 1 \leq k \leq n+1 \wedge 1 < k \quad \}$ |
| $k := k - 1$ |
| $y :=???$ |
| $\{P_{\text{inv}} : y = (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \wedge 1 \leq k \leq n+1 \quad \}$ |

correct. Indicate which of the following is the correct choice for updating $y$. Hint: derive it systematically!

a) $k := k - 1$
$y := p(k) \times x^{k-1} + y$

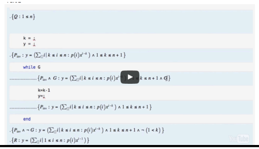
b) $k := k - 1$
$y := p(k) + y \times x$

c) $k := k - 1$
$y := y + p(k-2) \times x^{k-1}$

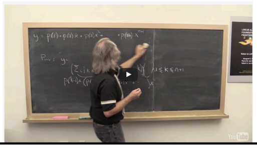
d) $k := k - 1$
$y := y + p(k) \times x$

**Enter a correct update to $y$ in the Live Script and enjoy getting the right answer the first time!**

☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube

With these homeworks you have discovered what is known in the United States as Horner's rule for evaluating a polynomial. It is what in practice is done to efficiently evaluate a polynomial. You may want to read the ☛ Wikipedia entry for Horner's method (Horner's rule) (search for "Horner's method").

## 3.6 Enrichment ☛ to edX

### 3.6.1 A conversation with Prof. David Gries ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

### 3.6.2 Dafny: a language and program verifier for functional correctness

"Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs."

We believe you will enjoy and appreciate learning about Microsoft's Dafny language and program verifier. The following more advanced example for that project is closely related to what you learned this Week: ☛ The Verification Corner: Loop Invariants. This may lead you to investigate this project more [LINK].

## 3.7 Wrap Up ☛ to edX

### 3.7.1 Additional exercises ☛ to edX

In the second part of the course, we will focus on operations from linear algebra, first with vectors (stored in one dimensional arrays) and later with matrices (stored in two dimensional arrays). To get this started, we now apply what we have learned to the dot product (also known as the inner product) operations.

The dot product of vectors $x$ and $y$ of size $n$ is often written as $x^T y$, which links the operation to that of computing the matrix-matrix multiplication (product) of a row vector $x^T$ with a column vector $y$, viewed as $1 \times n$ and $n \times 1$ matrices, respectively. If you are a bit fuzzy on this operation, you may want to visit units 1.4.3 and 1.6.1-1.6.3 of our MOOC titled "Linear Algebra: Foundations to Frontiers" (LAFF) offered on edX.

The operation is defined by

$$x^T y = (\sum i \mid 1 \leq i \leq n : x(i) \times y(i)),$$

where we start indexing at one, since we will want to implement it again with MATLAB. (In Week 4, we will briefly revert back to starting our indexing at zero as we introduce a notation that avoids indexing altogether.) If the result is stored in variable $d$, the postcondition is given by

$$d = (\sum i \mid 1 \leq i \leq n : x(i) \times y(i)).$$

Notice that

$$d = (\sum i \mid 1 \leq i < k : x(i) \times y(i)) + (\sum i \mid k \leq i \leq n : x(i) \times y(i)) \wedge 1 \leq k \leq n+1$$

and also

$$d = (\sum i \mid 1 \leq i \leq k : x(i) \times y(i)) + (\sum i \mid k < i \leq n : x(i) \times y(i)) \wedge 0 \leq k \leq n.$$

The subtle difference is to which of the two parts the element indexed by $k$ belongs.

**Homework 3.7.1.1** Identify which of the following are valid loop invariants for deriving a loop that computes the dot product:

a) $d = (\sum i \mid 1 \leq i < k : x(i) \times y(i)) \wedge 1 \leq k \leq n+1$.

b) $d = (\sum i \mid k \leq i \leq n : x(i) \times y(i)) \wedge 1 \leq k \leq n+1$.

c) $d = (\sum i \mid 1 \leq i \leq k : x(i) \times y(i)) \wedge 0 \leq k \leq n$.

d) $d = (\sum i \mid k < i \leq n : x(i) \times y(i)) \wedge 0 \leq k \leq n$.

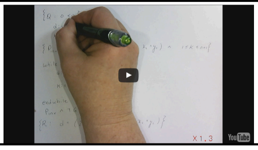
☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 3.7.1.2** For the loop invariant

$$d = (\sum i \mid 1 \leq i < k : x(i) \times y(i)) \wedge 1 \leq k \leq n+1$$

derive a correct program for computing $x^T y$. You will want to use the worksheet in Figure 3.11 for this exercise.

☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 3.7.1.3** Implement the program from the last exercise using the Live Script in

```
LAFFPfC −> Assignments −> Week3 −> matlab −> DotVariant1.mlx.
```

For additional instructions, see Homework 3.5.3.3 on the edX platform.
Make sure you get the right answer the first time!

☛ SEE ANSWER
☛ DO EXERCISE ON edX

There are actually eight algorithms that could result from the loop invariants in Homework 3.7.1.1: For each of the four invariants, you can update (increment or decrement) $k$ before updating $d$ or after updating $d$:

$k :=???$
$d :=???$

or

$d :=???$
$k :=???$

It is actually instructional to derive all eight. For some programs, the update to $d$ involves $x(k)$ and $y(k)$ and for others it involves $x(k-1)$ and $y(k-1)$ or $x(k+1)$ and $y(k+1)$. You can actually predict from the loop invariant which of these will happen. In Weeks 4-6 we abstract away from indexing details, and these distinctions disappear.

| |
|---|
| $\{$ $Q :$ $\}$ |
| $\{$ $\mathrm{wp}(\text{“}S_I\text{”}, P_{\mathrm{inv}}) :$ $\}$ |
| $S_I :$ |
| $\{$ $P_{\mathrm{inv}} :$ $\}$ |
| **while** **do** |
| $\{$ $P_{\mathrm{inv}} \wedge G :$ $\wedge$ $\}$ |
| $\{$ $\mathrm{wp}(\text{“}S_U ; k := k \quad 1\text{”}, P_{\mathrm{inv}}) :$ $\}$ |
| $S_U :$ |
| $\{$ $\mathrm{wp}(\text{“}k := k \quad 1\text{”}, P_{\mathrm{inv}}) :$ $\}$ |
| $k := k \quad 1$ |
| $\{$ $P_{\mathrm{inv}} :$ $\}$ |
| **endwhile** |
| $\{$ $P_{\mathrm{inv}} \wedge \neg G :$ $\wedge\ \neg( \qquad )$ $\}$ |
| $\{$ $R :$ $\}$ |

Figure 3.11: Blank worksheet.

**Homework 3.7.1.4** For the loop invariant

$$d = (\sum i \mid k < i \leq n : x(i) \times y(i)) \wedge 0 \leq k \leq n$$

derive a correct program for computing $x^T y$. You will want to use the worksheet in Figure 3.11 for this exercise.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 3.7.1.5** The prefix operation takes an array $x(0:n-1)$ and overwrites it so that $x(k)$ equals the sum of all previous elements in the array, including $x(k)$:

$$(\forall i \mid 0 \leq i < n : x(i) = (\sum j \mid 0 \leq j \leq i : \widehat{x}(j))).$$

The precondition would be

$$(\forall i \mid 0 \leq i < n : x(i) = \widehat{x}(i)) \wedge 1 \leq n.$$

Use the worksheet in Figure 3.11 to derive a correct loop for this operation, leveraging the loop invariant

$$P_{\text{inv}} : (\forall i \mid 0 \leq i < k : x(i) = (\sum j \mid 0 \leq j \leq i : \widehat{x}(j))) \wedge (\forall i \mid k \leq i < n : x(i) = \widehat{x}(i)) \wedge 1 \leq k \leq n.$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 3.7.1.6** To compare and contrast the effort needed to derive an algorithm vs. the effort required to prove an algorithm correct, consider the same prefix operation but for the case where the indexing of the array starts with 1:

$$(\forall i \mid 1 \leq i \leq n : x(i) = (\sum j \mid 1 \leq j \leq i : \widehat{x}(j))).$$

The precondition would now be

$$(\forall i \mid 1 \leq i \leq n : x(i) = \widehat{x}(i)) \wedge 1 \leq n.$$

Prove the following program correct:

```
{$Q : (\forall i \mid 1 \leq i \leq n : x(i) = \widehat{x}(i)) \wedge 1 \leq n$}
$k := 1$
{$P_{\text{inv}} : (\forall i \mid 1 \leq i \leq k : x(i) = (\sum j \mid 1 \leq j \leq i : \widehat{x}(j))) \wedge (\forall i \mid k < i \leq n : x(i) = \widehat{x}(i))$
    $\wedge 1 \leq k \leq n$}
while $k < n$
  $x(k+1) := x(k+1) + x(k)$
  $k := k+1$
endwhile
{$R : (\forall i \mid 1 \leq i \leq n : x(i) = (\sum j \mid 1 \leq j \leq i : \widehat{x}(j)))$}
```

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 3.7.2 Summary ☛ to edX

#### Developing an arbitrary command

Choose $S$ to satisfy the following worksheet:

| |
|---|
| $\{Q: \qquad \}$ |
| $\{Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)? \qquad \}$ |
| $\{\text{wp}(\text{“}S\text{”}, R): \qquad \}$ |
| $S:$ |
| $\{R: \qquad \}$ |

#### Developing the skip command

Check if $Q \Rightarrow R$:

| |
|---|
| $\{Q: \qquad \}$ |
| $\{Q \Rightarrow R? \qquad \}$ |
| $S:$ **skip** |
| $\{R: \qquad \}$ |

#### Developing a simple assignment

Choose $\mathcal{E}$ to satisfy the following worksheet:

| |
|---|
| $\{Q: \qquad \}$ |
| $\{Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)? \qquad \}$ |
| $\{\text{wp}(\text{“}S\text{”}, R): \qquad \}$ |
| $S: x := \mathcal{E}$ |
| $\{R: \qquad \}$ |

#### Developing a sequence of assignments

Choose $\mathcal{E}$, $\mathcal{E}(x)$, $\mathcal{E}(x,y)$, etc., to satisfy the following worksheet:

| |
|---|
| $\{Q: \qquad \}$ |
| $\{Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)? \qquad \}$ |
| $\{\text{wp}(\text{“}S_0; S_1; S_2\text{”}, R): \qquad \}$ |
| $S_0: x := \mathcal{E}$ |
| $S_1: y := \mathcal{E}(x)$ |
| $S_2: z := \mathcal{E}(x,y)$ |
| $\{R: \qquad \}$ |

### Developing an assignment to an array element

Isolate (by splitting the range) what is different in quantifiers that occur in the precondition and postcondition. Compare and contrast.

### Developing an if command

Choose $G_0$, $G_1$, etc., until $Q \Rightarrow G_0 \wedge G_1 \wedge \cdots$. For each $G_i$ develop

| |
|---|
| $\{ G_i \wedge Q :$ $\}$ |
| $\{ G_i \wedge Q \Rightarrow \text{wp}(\text{“}S_i\text{”}, R)?$ $\}$ |
| $\{ \text{wp}(\text{“}S_i\text{”}, R) :$ $\}$ |
| $S_i :$ |
| $\{ R :$ $\}$ |

Complete worksheet:

| |
|---|
| $\left\{ \; Q : \right.$ $\left. \right\}$ |
| $\left\{ \; Q \Rightarrow G_0 \vee G_1 \vee \cdots ? \right.$ $\left. \right\}$ |
| **if** |
| $G_0 \rightarrow$ |
| $\{ G_0 \wedge Q :$ $\}$ |
| $\{ G_0 \wedge Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, R)?$ $\}$ |
| $\{ \text{wp}(\text{“}S_0\text{”}, R) :$ $\}$ |
| $S_0 :$ |
| $\{ R :$ $\}$ |
| $\square \; G_1 \rightarrow$ |
| $\{ G_1 \wedge Q) :$ $\}$ |
| $\{ G_1 \wedge Q \Rightarrow \text{wp}(\text{“}S_1\text{”}, R)?$ $\}$ |
| $\{ \text{wp}(\text{“}S_1\text{”}, R) :$ $\}$ |
| $S_1 :$ |
| $\{ R :$ $\}$ |
| $\square \; \cdots$ |
| **fi** |
| $\{ \; R :$ $\}$ |

### Developing an if command (special case)

| |
|---|
| $\left\{ Q : \right\}$ |
| $\left\{ Q \Rightarrow G_0 \vee G_1 ? \right\}$ |
| **if** |
| $G_0 \rightarrow$ |
| $\left\{ G_0 \wedge Q : \right\}$ |
| $\{ G_0 \wedge Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0) ? \}$ |
| $\left\{ \text{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0) : \right\}$ |
| $S_0 :$ |
| $\left\{ G_0 \wedge R_0 : \right\}$ |
| ▯ $G_1 \rightarrow$ |
| $\left\{ G_1 \wedge Q) : \right\}$ |
| $\{ G_1 \wedge Q \Rightarrow \text{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1) ? \}$ |
| $\left\{ \text{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1) : \right\}$ |
| $S_1 :$ |
| $\left\{ G_1 \wedge R_1 : \right\}$ |
| **fi** |
| $\left\{ R : (G_0 \wedge R_0) \vee (G_1 \wedge R_1) \right\}$ |

### Developing a while loop

For a loop over an array or multiple arrays, specified by postcondition $R$ with one or more quantifiers:

- Determine a logical loop index.
- Split the quantifier(s) in $R$ using the loop index.
- Determine one or more loop invariants from the resulting predicate by answering the question “What constitutes partial progress towards the result.”
- Pick a loop invariant that results.
- Determine the loop guard.
- Determine an initialization step.
- Determine whether to increment or decrement the loop counter in the loop body and whether to do so at the top or bottom of the loop body.
- Derive the remainder of the loop body.

Assuming the loop index is updated at the bottom of the loop body, use the worksheet given on the next page.

```
$\{$ $Q:$ $\}$
$\{$ wp(“$S_I$”, $P_{\rm inv}$) : $\}$
$S_I$
$\{$ $P_{\rm inv}$ : $\}$
**while** **do**
    $\{$ $P_{\rm inv} \land G$ :
          $\land$ $\}$
    $\{$ wp(“$S_U$; $k := k$ 1”, $P_{\rm inv}$) : $\}$
    $S_U$ :
    $\{$ wp(“$k := k$ 1”, $P_{\rm inv}$) : $\}$
    $k := k$ 1
    $\{$ $P_{\rm inv}$ : $\}$
**endwhile**
$\{$ $P_{\rm inv} \land \neg G$ : $\land$ $\neg($ $)$ $\}$
$\{$ $R$ : $\}$
```

### 3.7.3 Why Dijkstra received the ACM Turing Award ☛ to edX

Dijkstra received the ACM Turing Award in 1972

> "For fundamental contributions to programming as a high, intellectual challenge; for eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; for illuminating perception of problems at the foundations of program design."

What you hopefully have noticed in this week is that the thought process behind developing different parts of a program can be made remarkably systematic. Key is thinking about programs themselves in a structured way (which is why the advent of "structured programming" in the late 1950s was so important). In this course, we then structure goal-oriented programming with "worksheets" for different occasions.

# Part II

# Application

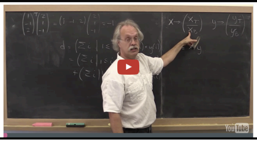
# Week 4

# Matrix-Vector Operations

## 4.1 Opening Remarks ☛ to edX

### 4.1.1 Launch ☛ to edX

"How do we convince people that in programming simplicity and clarity –in short: what mathematicians call "elegance"– are not a dispensable luxury, but a crucial matter that decides between success and failure?" – Dijkstra ☛ EWD648 (1972)



☛ Watch Video on edX
☛ Watch Video on YouTube

In the first part of the course, we used operations over one-dimensional arrays for many of our examples. We now revisit some of these operations as we introduce new notation, developed as part of our FLAME project, that will make derivation of algorithms, in our experience, less cumbersome. The true power of the notation will be seen when we discuss operations with matrices, later this week and in the remainder of the course.

Let us consider the dot product discussed in Section 3.5: Given vectors $x$ and $y$ of size $n \geq 0$, compute

$$\begin{aligned} d &= x^T y \\ &= (\sum i \mid 0 \leq i < n : x(i) \times y(i)) \end{aligned}$$

An annotated algorithm that resulted from the techniques we discussed is given in Figure 4.1.

What do we notice? The vectors (arrays) $x$ and $y$ are inherently partitioned into two subvectors (subarrays):

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) = \left(\begin{array}{c} x(0) \\ \vdots \\ x(k-1) \\ \hline x(k) \\ \vdots \\ x(n-1) \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} y(0) \\ \vdots \\ y(k-1) \\ \hline y(k) \\ \vdots \\ y(n-1) \end{array}\right).$$

Here the $T$ refers to the *T*op part of the vectors and the $B$ refers to the *B*ottom part of the vectors. We pronounce $x_T$ and $x_B$ as "x-top" and "x-bottom", respectively.

```
$\{\ (0 \le n)\ \}$
$k := 0$
$d := 0$
$\{\ d = (\forall i \mid 0 \le i < k : x(i) \times y(i)) \wedge (0 \le k \le n)\ \}$
while $k < n$ do
    $\{\ d = (\forall i \mid 0 \le i < k : x(i) \times y(i)) \wedge (0 \le k \le n) \wedge (k < n)\ \}$
    $d := d + x(k) \times y(k)$
    $k := k + 1$
    $\{\ d = (\forall i \mid 0 \le i < k : x(i) \times y(i)) \wedge (0 \le k \le n)\ \}$
endwhile
$\{\ d = (\forall i \mid 0 \le i < k : x(i) \times y(i)) \wedge (0 \le k \le n) \wedge \neg(k < n)\ \}$
$\{\ d = (\sum i \mid 0 \le i < n : x(i) \times y(i))\ \}$
```

Figure 4.1: Annotated algorithm for computing $d = x^T y$.

With this, we can express the quantifiers involved in the derivation of the loop invariant more concisely:

$$\overbrace{d = \overbrace{(\sum i \mid 0 \le i < k : x(i) \times y(k))}^{x_T^T y_T} + \overbrace{(\sum i \mid k \le i < n : x(i) \times y(k))}^{x_B^T y_B}}^{\overbrace{d = \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)^T \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)}^{d = x^T y}}$$

Now the condition $(0 \le k \le n)$ can be made implicit, since the subvectors $x_T$, $x_B$, $y_T$, and $y_B$ are inherently subvectors of $x$ and $y$.

**Homework 4.1.1.1** Match the predicate on the left with the corresponding predicate on the right:

| | |
|---|---|
| (1) $d = (\sum i \mid 0 \le i < n : x(i) \times y(i))$ | (a) $d = x_B^T y_B$ |
| (2) $d = (\sum i \mid 0 \le i < k : x(i) \times y(i))$ | (b) $d = x^T y$ |
| $+ (\sum i \mid k \le i < n : x(i) \times y(i)) \wedge 0 \le k \le n$ | |
| (3) $d = (\sum i \mid 0 \le i < k : x(i) \times y(i)) \wedge 0 \le k \le n$ | (c) $d = x_T^T y_T + x_B^T y_B$ |
| (4) $d = (\sum i \mid k \le i < n : x(i) \times y(i)) \wedge 0 \le k \le n$ | (d) $d = x_T^T y_T$ |

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.1.1.2** In the annotated algorithm for computing $d = x^T y$ in Figure 4.1, place the following expressions where they can replace the quantifiers. Use your intuition!

- $d = x^T y$
- $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ and $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$, where $x_T$ and $y_T$ have no elements
- $d = x_T^T y_T$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

The problem is that even if this new notation captures how we think of the vectors more concisely, it is not clear how the indexing with $k$ and the update $d := d + x(k) \times y(k)$ interacts with this notation. Furthermore, we would like to move on to algorithms over two dimensional arrays. These are the topics we explore this week.

☛ Watch Video on edX
☛ Watch Video on YouTube

### 4.1.2 Outline Week 4 ☛ to edX

### 4.1.3 What you will learn ☛ to edX

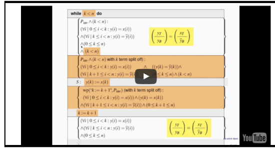
This is it! We enter the frontier. Reasoning starting with the goal is definitely the way to go. The problem you may have noticed in Week 3 is that indices get in the way. Before, indices were where you would make many of your programming errors. Now, you still find that it is easy to make similar mistakes in the quantifiers. So, how do we get rid of those pesky indices? The answer is to think in terms of "parts and whole," representing progress through arrays and loops utilizing regions instead of quantifiers. Through abstraction and special notation, we hide indices and expose strategies for finding invariants. As before, the invariants then guide the development of loop-based algorithms. We call this the FLAME approach.

Upon completion of this week, you should be able to

- Abstract away from indices.
- Use FLAME notation to express the goal (postcondition), find the Partitioned Matrix Expression (PME), determine invariants corresponding to a PME, and, with a worksheet, derive loop-based algorithms involving traversing in one dimension.
- Use the Spark webpage to translate from algorithms to MATLAB code.
- Utilize Live Script to embed proofs of correctness in the MATLAB code.
- Recognize the value of having a family of algorithms that compute the same operation.

## 4.2 A Farewell to Indices ☛ to edX

### 4.2.1 More notation ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

In the first part of the course, we used operations over one dimensional arrays for many of our examples. We now revisit one of the simplest such operations as we introduce new notation that will make derivation of algorithms, in our experience, less cumbersome. The true power of the notation will be seen when we discuss operations with matrices, later this week and in the remainder of the course.

Let us, again, consider the copying of vector $x$ into vector y. The annotated algorithm that resulted from the techniques we discussed in Part I is given in Figure 4.2.

What do we notice? The vectors (arrays) $x$ and $y$ (and $\widehat{y}$) are inherently partitioned into two subvectors (subarrays):

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) = \left(\begin{array}{c} x(0) \\ \vdots \\ x(k-1) \\ \hline x(k) \\ \vdots \\ x(n-1) \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} y(0) \\ \vdots \\ y(k-1) \\ \hline y(k) \\ \vdots \\ y(n-1) \end{array}\right).$$

Here the subscript $T$ refers to the $T$op part of vectors and the subscript $B$ refers to the $B$ottom part of vectors.

With this, we can express the loop invariant more concisely:

$$\overbrace{\overbrace{(\forall i \mid 0 \le i < k : y(i) = x(i))}^{y_T = x_T} \;\wedge\; \overbrace{(\forall i \mid k \le i < n : y(i) = \widehat{y}(i))}^{y_B = \widehat{y}_B}}^{\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right)}$$

Now the constraint $0 \le k \le n$ can be made implicit, since the subvectors $x_T$, $x_B$, $y_T$, and $y_B$ are inherently subvectors of $x$ and $y$. Also, since we now hide the indexing into the arrays, it does not matter whether the arrays start with $x(0)$ and $y(0)$ or $x(1)$ and $y(1)$.

We will use the functions $m(A)$ and $n(A)$ to extract the number of rows and number of columns in matrix $A$, respectively.

Thus, $n = m(x) = m(y)$ in this example, viewing the vectors as column vectors and hence the special case of a matrix with $m(x) = m(y)$ elements.

Next, let us further annotate the algorithm with some of the details of the derivation/proof of the loop body, as given on Page 183, where we already add some equivalent annotations using the new notation, consistent with what you discovered in the launch for this week.

What one notices there (and in all of the examples in Part I that dealt with vectors) is that, when the loop proceeds through arrays from first to last element, inherently the first element of the bottom part is involved in computation and inherently a term related to that element has to be split from a quantifier as part of the proof. This suggests the following additional notation:

$$\left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right) = \left(\begin{array}{c} x(0) \\ \vdots \\ x(k-1) \\ \hline x(k) \\ x(k+1) \\ \vdots \\ x(n-1) \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} y(0) \\ \vdots \\ y(k-1) \\ \hline y(k) \\ y(k+1) \\ \vdots \\ y(n-1) \end{array}\right),$$

where $\chi_1$ (Greek lower case letter "chi") and $\psi_1$ (Greek lower case letter "psi") now represent the top elements of $x_B$ and $y_B$, respectively.

**Homework 4.2.1.1** In the annotated algorithm on Page 183 for copying vector x into vector y insert the expressions where they make sense. Use your intuition!

a) $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$

b) $m(y_T) < m(y)$ (three places).

c) $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$

d) $\left(\begin{array}{c} y_0 \\ \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \widehat{\psi}_1 \\ \widehat{y}_2 \end{array}\right).$

e) $\left(\begin{array}{c} y_0 \\ \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \chi_1 \\ \widehat{y}_2 \end{array}\right)$

f) $\psi_1 := \chi_1$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

☛ Watch Video on edX
☛ Watch Video on YouTube

Once one understands how the two notations are related, the annotated algorithm using only the suggested new notation can be given more concisely, as in Figure 4.3.

After one then removes the annotations that are part of the proof, one is left with the algorithm.

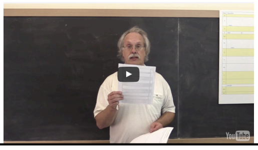
```
$x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ and $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$
    where $x_T$ and $y_T$ are empty
while $m(y_T) < m(y)$ do
    $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$
        where $\chi_1$ and $\psi_1$ are scalars
    $\psi_1 := \chi_1$
    $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$
endwhile
```

This notation, which hides the details of indexing, is commonly known as the FLAME notation.

**Homework 4.2.1.2** Modify the annotated algorithm in Figure 4.3 "in the obvious way" so that it copies $x$ into $y$ from the last element to the first element. You may want to do so by printing Figure 4.4.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 4.2.2 Deriving algorithms with the FLAME notation ☛ to edX

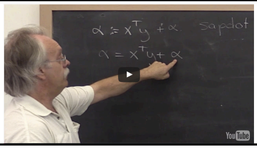


☛ Watch Video on edX
☛ Watch Video on YouTube

Let us now revisit the computation of the dot product of vector $x$ with vector $y$, and, using the FLAME notation, systematically derive the algorithms to be correct. We change the operation slightly, computing $\alpha := x^T y + \alpha$ instead. Notice this is a slight generalization since $\alpha$ can always be initialized to zero before executing the algorithm to compute $\alpha := x^T y$. You may want to print the ☛ blank worksheet so that you write the steps down as the explanation progresses for better understanding.

We use the lower case Greek letter $\alpha$ (alpha) because we mostly reserve lower case Roman letters to denote vectors and lower case Greek letters to denote scalars, as an added visual cue.

### Step 1. Precondition and postcondition



☛ Watch Video on edX
☛ Watch Video on YouTube

We start with the precondition

$$\alpha = \widehat{\alpha},$$

where (as before) the $\widehat{\alpha}$ is used to denote the original contents of variable $\alpha$. Implicit is the fact that the two vectors are of equal size: $0 \leq m(x) = m(y)$. The postcondition is given by

$$\alpha = x^T y + \widehat{\alpha}.$$

### Step 2. *A priori* determination of loop invariants

☛ Watch Video on edX
☛ Watch Video on YouTube

We by now know that one tends to move through vectors (arrays) systematically, from top to bottom or bottom to top. At an intermediate point, one reasons about regions. So, at the top of the loop body, the vectors are in a state where they have been partitioned as

$$x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \quad \text{and} \quad y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right).$$

From the earlier units in this week, we recognize this as helping us when splitting the range of the quantifier.

If we substitute this into the postcondition

$$\alpha = x^T y + \widehat{\alpha}$$

we get

$$\alpha = \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)^T \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) + \widehat{\alpha}$$

and, by applying what we know about the dot product of partitioned vectors, we find that

$$\alpha = x_T^T y_T + x_B^T y_B + \widehat{\alpha}. \tag{4.1}$$

(If you are not comfortable with computing the dot product of partitioned vectors, we suggest you visit ☛ Units 1.6.1-1.6.3 of LAFF.)

We call the expression in (**??**) the Partioned Matrix Expression (PME), where here a vector can be viewed as a special case of a matrix. It is a recursive definition of the operation, in terms of the partitioned operands.

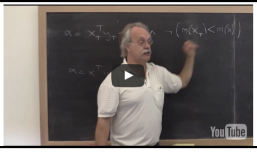
This then allows us to come up with loop invariants much like we did when we were explicitly indexing into arrays:

**Invariant 1:** $\alpha = x_T^T y_T + \widehat{\alpha}$.
(The dot product has proceeded to where $\alpha$ has been updated with the dot product of $x_T$ and $y_T$, traversing the vectors from first to last element.)

**Invariant 2:** $\alpha = x_B^T y_B + \widehat{\alpha}$.
(The dot product has proceeded to where $\alpha$ has been updated with the dot product of $x_B$ and $y_B$, traversing the vectors from last to first element.)

For the remainder of the discussion, let us pick Invariant 1. This leads to the partial "worksheet" in Figure 4.5.

**Step 3: Determine the loop guard**

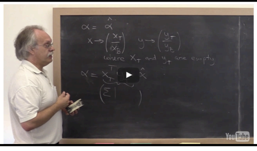

☛ Watch Video on edX
☛ Watch Video on YouTube

We know that $(P_{\rm inv} \wedge \neg G) \Rightarrow R$ has to hold. Given the loop invariant we chose, we find that

$$(\alpha = x_T^T y_T + \widehat{\alpha} \wedge \neg G) \Rightarrow (\alpha = x^T y + \widehat{\alpha})$$

must hold. The choice $G : m(x_T) < m(x)$ has the desired property (since the fact that $x_T$ is a subvector of $x$ implicitly means that $0 \leq m(x_T) \leq m(x)$ is TRUE and $m(y_T) = m(x_T)$ by design).

**Step 4: Determine the initialization**


☛ Watch Video on edX
☛ Watch Video on YouTube

The initialization step partitions the vectors, which takes the place of the initialization of a loop index. The precondition is given by

$$\alpha = \widehat{\alpha}.$$

The initialization inherently has the form

$$x \rightarrow \left( \begin{array}{c} x_T \\ \hline x_B \end{array} \right), \; y \rightarrow \left( \begin{array}{c} y_T \\ \hline y_B \end{array} \right)$$

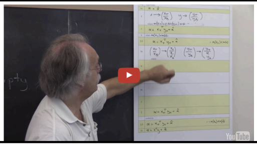
and has to leave the so exposed subvectors in a state where the loop invariant holds

$$\alpha = x_T^T y_T + \widehat{\alpha}$$

preferably without performing any real computation. Now, we notice that if $x_T$ and $y_T$ are empty, then $x_T^T y_T$ equals zero (summation over the empty range!) and hence we conclude that after the initialization the loop invariant holds. Obviously, here we are building on the intuition that we developed in Week 3. We could formally derive the initialization.

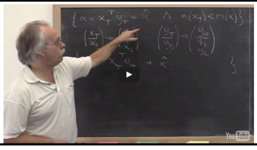
This yields the worksheet as given in Figure 4.6.

**Step 5: Progressing through the vectors**


☛ Watch Video on edX
☛ Watch Video on YouTube

Because $x_T$ is initially empty (and hence so is $y_T$) and the loop guard tells us that eventually $x_T$ must be all of $x$, we conclude that every time through the loop the top element of the bottom subvector should be moved to the bottom of the top subvector, for each of the vectors $x$ and $y$. This leads us to the repartitionings in Step 5a and 5b of Figure 4.7.

### Step 6: State before the update



☛ Watch Video on edX
☛ Watch Video on YouTube

After Step 5a, the loop invariant is still true. We can use this to determine the values of the exposed parts of the subvectors:

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$$

means that

$$\alpha = x_T^T y_T + \widehat{\alpha}$$

is now expressed as

$$\alpha = x_0^T y_0 + \widehat{\alpha} .$$

This is sort of like splitting the range. Here

$$\alpha = x_T^T y_T + \widehat{\alpha}$$

represents

$$\alpha = (\sum \mid 0 \le i < k : x(i) \times y(i)) + \widehat{\alpha}$$

which can also be written as

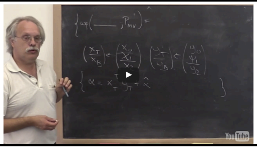
$$\alpha = (\sum \mid 0 \le i < k : x(i) \times y(i)) + \underbrace{(\sum \mid k \le i < n : 0)}_{0} + \widehat{\alpha}$$

where the second quantifier captured that for future iterations nothing has been added yet (hence the 0). This in turn can then be expressed as

$$\alpha = (\sum i \mid 0 \le i < k : x(i) \times y(i)) + \underbrace{(\sum i \mid i = k : 0)}_{0} + \underbrace{(\sum i \mid k < i < n : 0)}_{0} + \widehat{\alpha}$$

at which point the implicit "splitting of the range" becomes apparent.

### Step 7: State after the update



☛ Watch Video on edX
☛ Watch Video on YouTube

Similarly, in Step 7 we need to fill in

$$\text{wp}(\text{``}\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right); \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)\text{''}, \alpha = x_T^T y_T + \widehat{\alpha}):$$

$$\alpha = \left(\begin{array}{c} x_0 \\ \chi_1 \end{array}\right)^T \left(\begin{array}{c} y_0 \\ \psi_1 \end{array}\right) + \widehat{\alpha}$$

$$\Leftrightarrow$$

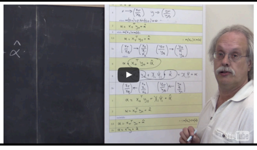
$$\alpha = \alpha = x_0^T y_0 + \chi_1 \times \psi_1 + \widehat{\alpha}$$

which captures that

$$\alpha = (\Sigma i \mid 0 \leq i < k : x(i) \times y(i)) + x(k) \times y(k) + (\Sigma i \mid k < i < n : 0) + \widehat{\alpha}$$

The worksheet is now as given in Figure 4.8.

## Step 8: Update



☛ Watch Video on edX
☛ Watch Video on YouTube

One now examines

| |
|---|
| $\{ \alpha = x_0^T y_0 + \widehat{\alpha} \}$ |
| |
| $\{ \alpha = x_0^T y_0 + \chi_1 \times \psi_1 + \widehat{\alpha} \}$ |

to conclude that the assignment

$$\alpha := \chi_1 \times \psi_1 + \alpha$$

does the trick. This leaves us with the completed worksheet in Figure 4.9.

This is like comparing the state in Step 6, expressed with indices,

$$\alpha = (\Sigma i \mid 0 \leq i < k : x(i) \times y(i)) + \underbrace{(\Sigma i \mid i = k : 0)}_{0} + \underbrace{(\Sigma i \mid k < i < n : 0)}_{0} + \widehat{\alpha}$$

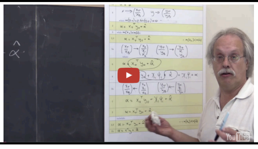
to the state in Step 7, expressed with indices:

$$\alpha = (\Sigma i \mid 0 \leq i < k : x(i) \times y(i)) + x(k) \times y(k) + \underbrace{(\Sigma i \mid k < i < n : 0)}_{0} + \widehat{\alpha}$$

to conclude that we should update

$$\alpha := x(k) \times y(k) + \alpha.$$

### Algorithm



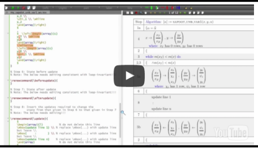
☛ Watch Video on edX
☛ Watch Video on YouTube

Finally, erasing all annotations, we are left with the correct algorithm, given in Figure 4.10.

**Homework 4.2.2.1** Use the ☛ blank worksheet to derive Variant 2 for computing the "sapdot" operation $\alpha := x^T y + \alpha$, the algorithm corresponding to Invariant 2. (In theory, this worksheet is also in `LAFFPfC/Resources/BlankWorksheet.pdf`. In practice, you may want to put a copy there yourself, since the one that is there is not quite the same.)

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 4.2.3 Typesetting algorithms with FLAME notation and LaTeX ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

One drawback of how we present our algorithms with the FLAME notation is that it takes considerably more room and time to write them down. To overcome this, we suggest using LaTeX. In Week 0 you presumably installed a tool, TeXstudio, for writing LaTeX documents. (Alternatively, use ShareLaTeX.com online.)

**Homework 4.2.3.1** Follow the instructions in the video to duplicate Figure 4.9.

- Download `sapdot_unb_var1_ws.tex` and place it in `LAFFPfC/Assignments/Week4/LaTeX/`.
- Also download `color_flatex.tex` and place it in the same directory.
- If you use `sharelatex.com` you will want to upload these files there.
- Use the mentioned "Spark" webpage. You may want to bookmark it!

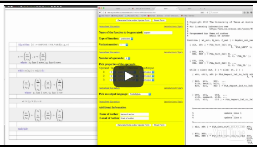
It seems like the Spark webpage sometimes doesn't update correctly. You may have to close the window and restart at that point. You may also have to click "Reset Form" and then reload... It is a quick and dirty implementation. Dijkstra would not approve!

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.2.3.2** Typeset the algorithm from Homework 4.2.2.1. Download `sapdot_unb_var2_ws.tex` and place it in `LAFFPfC/Assignments/Week4/LaTeX/`. Make sure you "march" through the vectors in the correct direction!

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 4.2.4 Representing (FLAME) algorithms in code ☛ to edX

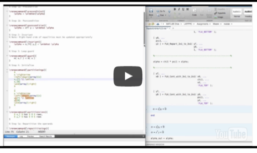


☛ Watch Video on edX
☛ Watch Video on YouTube

We now have a new notation for expressing algorithms. The problem is that a correct algorithm still has to be translated into correct code. Programming bugs can be easily introduced as part of that translation. To overcome this, we will use an ☛ Application Programming Interface (API) that allows the code to closely resemble the algorithm.

**Homework 4.2.4.1** Place the Live Script `SapdotUnbVar1LS.mlx` in directory `LAFFPfC/Week4/matlab/` and follow the instructions in the video to translate the algorithm in Figure 4.10 into code using the FLAME@lab API.

☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

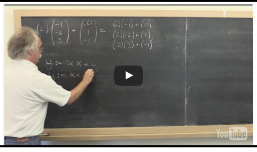
**Homework 4.2.4.2** Translate the algorithm from Homework 4.2.2.1 into code starting with the Live Script in

`Assignments/Week4/matlab/Sapdot_unb_var2_LS.mlx`

Make sure you "march" through the vectors in the correct order.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 4.2.5 The AXPY operation ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

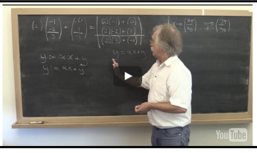
There is a important operations that we will encounter in the remainder of this course: The AXPY ( a lpha times x p lus y ) operation defined by

$$y := \alpha x + y .$$

It is an operation supported by the BLAS, discussed in an enrichment in this week.

**Homework 4.2.5.1** Derive a Partitioned Matrix Expression (PME) for computing $y := \alpha x + y$.

☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 4.2.5.2** Derive two loop invariants (the first for marching through the vector $x$ from first to last element and the second for marching from last to first element) for the AXPY operation $y := \alpha x + y$.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 4.2.5.3** Using the Worksheet derive the algorithm, Variant 1, corresponding to Invariant 1.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.2.5.4** Typeset the worksheet and algorithm corresponding to Invariant 1. For this, start by downloading axpy_unb_var1_ws.tex into `Assignments/Week4/LaTeX/`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.2.5.5** Download the Live Script AxpyUnbVar1LS.mlx into `Assignments/Week4/matlab/` and use it to translate Variant 1 into code. If you feel energetic, add the annotations that record the proof of correctness.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

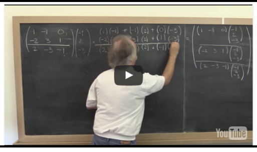
**Homework 4.2.5.6** If you feel energetic, repeat the last homeworks with Invariant 2.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 4.3 Algorithms over two-dimensional arrays (matrices) ☛ to edX

### 4.3.1 Some algorithms for matrix-vector multiplication ☛ to edX

If you find yourself a bit rusty on the details of how matrix-vector multiplication works, you will want to check out ☛ Week 3 of LAFF. (Indeed, you may want to start in Week 2, since linear transformations may be even more of a mystery.)



☛ Watch Video on edX
☛ Watch Video on YouTube

### Step 1: Precondition and postcondition

Let us consider simple matrix-vector multiplication: $y := Ax + y$. (It is often convenient to add to the result vector rather than just computing $Ax$. You can always set $y$ to the zero vector before computing $y := Ax + y$.) The precondition is that

$$y = \widehat{y}$$

(and a number of terms regarding the relative sizes of $A$, $x$, and $y$, which we typically keep in mind but don't state explicitly). The postcondition is

$$y = Ax + \widehat{y}.$$

### Step 2: Deriving loop invariants

When discussing operations with vectors, the choice of how to partition the operands (vectors) was pretty clear. There is only one way to partition these. For matrices, tracking through the matrix leads to three options:

1. Partition the matrix into a top part and bottom part:
$$A \rightarrow \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right);$$
2. Partition the matrix into a left part and right part:
$$A \rightarrow \left( \begin{array}{c|c} A_L & A_R \end{array} \right).$$
Here we pronounce $A_L$ and $A_R$ as "A left" and "A right".
3. Partition the matrix into quadrants:
$$A \rightarrow \left( \begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right).$$

We will see uses for the last option later this week. Let's focus on the first one.

For operations with vectors, we partitioned the vectors and then substituted the result into the precondition and postcondition. It is actually the substitution into the postcondition that is usually most telling. Let us partition

$$A \rightarrow \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right)$$

and let us substitute this into the postcondition $y = Ax + \widehat{y}$:

$$y = \underbrace{\left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) x}_{\left( \begin{array}{c} A_T x \\ \hline A_B x \end{array} \right)} + \widehat{y}.$$

This tells us that in order to add the result of $Ax$ to $\widehat{y}$, producing $y$, we need to also partition $y$ and $\widehat{y}$:

$$\left( \begin{array}{c} y_T \\ \hline y_B \end{array} \right) = \left( \begin{array}{c} A_T x \\ \hline A_B x \end{array} \right) + \left( \begin{array}{c} \widehat{y}_T \\ \hline \widehat{y}_B \end{array} \right)$$

which leaves us with

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_T x + \widehat{y}_T \\ \hline A_B x + \widehat{y}_B \end{array}\right).$$

This expresses the desired final result in terms of the partitioned matrices. Some of you may recognize it also as a recursive definition of matrix-vector multiplication. Hidden in our notation are nested quantifiers. We have dubbed this expression the *Partitioned Matrix Expression* (PME) for this operation.

A loop invariant represents a partial computation toward the final result. It is from the PME that we can now identify loop invariants:

**Invariant 1:** $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_T x + \widehat{y}_T \\ \hline A_B x + \widehat{y}_B \end{array}\right)$. Notice that we "struck out" $A_B x$, which is still vaguely visible in light gray. This represents computation that will happen in future iterations of the loop.

**Invariant 2:** $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_T x + \widehat{y}_T \\ \hline A_B x + \widehat{y}_B \end{array}\right)$. Here, we "struck out" $A_T x$.

#### Steps 3-8: Deriving variants

Hint: Matrix $A$ will be repartitioned $\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline a_1^T \\ A_2 \end{array}\right)$. Here $a_1^T$ represents the top row of $A_B$. We use lower case Roman letters to denote *column* vectors. The $^T$ indicates it is a transposed column vector (making it a row) and here it is really part of a variable or label that denotes the row, $a_1^T$, rather than that it indicates the transposition operation. This is sometimes a cause for confusion. Experience tells us that you will get used to it!

**Homework 4.3.1.1** Derive and typeset the worksheet and algorithm corresponding to Invariant 1. A partial LaTeX document can be found in

`Assigments/Week4/LaTeX/gemv_unb_var1.tex.`

What operation do you recognize in the update step of the loop body?

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.1.2** Translate the algorithm from the last homework into code starting with the Live Script in

`Assignments/Week4/matlab/gemv_unb_var1_LS.mlx`

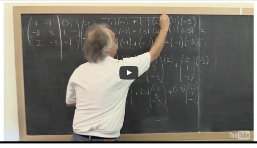
☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.1.3** If you feel energetic, derive the worksheet and algorithm for Invariant 2, and implement it starting with the Live Script in

`Assignments/Week4/matlab/gemv_unb_var2_LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 4.3.2 But you get so much more... ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

A second set of loop invariants results from partitioning $A \rightarrow \left( A_L \,\middle|\, A_R \right)$. If we substitute this into the postcondition $y = Ax + \widehat{y}$, we notice that $x$ must also be "conformally" partitioned:

$$y = \underbrace{\left( A_L \,\middle|\, A_R \right) \left( \begin{array}{c} x_T \\ \hline x_B \end{array} \right)}_{A_L x_T + A_R x_B} + \widehat{y},$$

By "conformally" we mean that the operation is well-defined. In this case this means that the number of columns in $A_L$ matches the number of elements in $x_T$. This tells us that this time $y$ needs not be partitioned. Again, this expresses the desired final result in terms of the partitioned matrix and vectors. Some of you may recognize it also as a recursive definition of matrix-vector multiplication. Again, hidden in our notation are nested quantifiers. It is an alternative PME for this operation.

A loop invariant represents a partial computation toward the final result. It is from the PME that we can now identify loop invariants:

**Invariant 3:** $y = A_L x_T + A_R x_B + \widehat{y}$. We "struck out" $A_R x_B$, which is still vaguely visible in light gray. This represents computation that will happen in future iterations of the loop.

**Invariant 4:** $y = A_L x_T + A_R x_B + \widehat{y}$. Here, we "struck out" $A_L x_T$.

#### Steps 3-8: Deriving variants

Hint: Matrix $A$ will be repartitioned $\left( A_L \,\middle|\, A_R \right) \rightarrow \left( A_0 \,\middle|\, a_1 \,\middle|\, A_2 \right)$. Here $a_1$ represents the left-most column of $A_R$.

**Homework 4.3.2.1** Derive and typeset the worksheet and algorithm corresponding to Invariant 3. A partial LaTeX document can be found in

`Assigments/Week4/LaTeX/gemv_unb_var3.tex.`

What operation do you recognize in the update step of the loop body?

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.2.2** Translate the algorithm from the last homework into code starting with the Live Script in

`Assignments/Week4/matlab/gemv_unb_var3_LS.mlx`

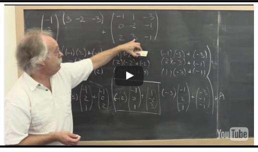
☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.2.3** If you feel energetic, derive the worksheet and algorithm for Invariant 4, and implement it starting with the Live Script in

`Assignments/Week4/matlab/gemv_unb_var4_LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 4.3.3 The rank-1 update ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Another important operation in linear algebra is the rank-1 update: $A := A + xy^T$, where $A$ is a matrix and $x$ and $y$ are vectors. The reason for the name is that $xy^T$ is a matrix of rank at most one and it updates $A$. (For the linear algebra inclined: Each column is a multiple of the vector $x$ and each row is a multiple of the row vector $y^T$. Hence there is at most one linearly independent column which means the rank is at most one.) In the BLAS alphabet soup that is used to name operations, this is known as a ge neral r ank-1 update ( GER ).

**Homework 4.3.3.1** Derive two different PMEs for this operation. Hint: Partition $A$ in two different ways.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.3.2** Derive and typeset the worksheet and algorithm corresponding to Invariant 1. A partial LaTeX document can be found in

`Assigments/Week4/LaTeX/gemr_unb_var1.tex.`

What operation do you recognize in the update step of the loop body?

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.3.3** Translate the algorithm from the last homework into code starting with the Live Script in

`Assignments/Week4/matlab/ger_unb_var1_LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.3.4** If you feel energetic, derive the worksheet and algorithm for Invariant 2, and implement it starting with the Live Script in

`Assignments/Week4/matlab/ger_unb_var2_LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.3.5** Derive and typeset the worksheet and algorithm corresponding to Invariant 3. A partial LaTeX document can be found in

`Assigments/Week4/LaTeX/ger_unb_var3.tex.`

What operation do you recognize in the update step of the loop body?

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.3.6** Translate the algorithm from the last homework into code starting with the Live Script in

`Assignments/Week4/matlab/ger_unb_var3_LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.3.7** If you feel energetic, derive the worksheet and algorithm for Invariant 4, and implement it starting with the Live Script in

`Assignments/Week4/matlab/ger_unb_var4_LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 4.3.4 Why do we want multiple algorithms? ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

Notice that you derived the algorithm for computing matrix-vector multiplication without first trying a few problems to get the feel for the computation. That is goal-oriented programming: You derive from specification. For matrix-vector multiplication, this yielded four algorithms, two of which are summarized in Figure 4.11. The FLAME notation allows one to easily compare and contrast these algorithms.

**Homework 4.3.4.1** Use GEMV_UNB_VAR1 to compute

$$\begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix}\begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix}+\begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix}=$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 4.3.4.2** Use GEMV_UNB_VAR3 to compute

$$\begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} =$$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

Now, MATLAB (as do many programming languages and software libraries used in high-performance scientific computing) stores matrices in *column-major* order, which means that a matrix

$$\begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix}$$

is stored in memory by stacking columns:

$$\begin{array}{|r|} 1 \\ -2 \\ -1 \\ \hline -1 \\ 2 \\ 1 \\ \hline 2 \\ 0 \\ -2 \\ \hline \end{array}$$

Computation tends to be more efficient if one accesses memory contiguously. This means that an algorithm that accesses $A$ by columns often computes the answer faster than one that accesses $A$ by rows. Other languages store matrices in *row-major* order, which stores matrices by rows. In that case the algorithm that accesses the matrix by rows is typically more efficient (will complete faster).

**Homework 4.3.4.3** Which algorithm is likely more efficient when the matrix is stored in column-major order, `gemv_unb_var1` or `gemv_unb_var3`?
Which algorithm is likely more efficient when the matrix is stored in row-major order, `gemv_unb_var1` or `gemv_unb_var3`?

☛ SEE ANSWER
☛ DO EXERCISE ON edX

Why finding multiple algorithms is important will become progressively clearer as Weeks 5 and 6 unfold, especially when you spend some time with the enrichments.

## 4.4 Enrichment ☛ to edX

### 4.4.1 Related reading ☛ to edX

The Basic Linear Algebra Subprograms (BLAS) are an interface to linear algebra operations that are commonly used in scientific computing libraries. The operations identified for this interface inlude some we have already discussed: the dot product, the ”axpy” operation, matrix-vector multiplication and rank-1 update. Others you will learn about in future weeks.

An overview of the BLAS can be found in

> Robert van de Geijn and Kazushige Goto, ”BLAS (Basic Linear Algebra Subprograms)”, *Encyclopedia of Parallel Computing*, Part 2, Pages 157-164. 2011.

If you don't have access to this through your university's library, read a draft of this article available from the LAFF-On unit on edX.

The original papers, cited in that article, are

> C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, ”Basic Linear Algebra Subprograms for Fortran Usage,” *ACM Transactions on Mathematical Software*, 5 (1979) 305-325.

> J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, ”An Extended Set of FORTRAN Basic Linear Algebra Subprograms,” *ACM Transactions on Mathematical Software*, 14 (1988) 1-17.

> J. J. Dongarra, J. Du Croz, S. Hammarling, and I Duff, ”A Set of Level 3 Basic Linear Algebra Subprograms,” *ACM Transactions on Mathematical Software*, 16 (1990) 1-17.

The style of coding that we use is at the core of our FLAME project and was first published in

> John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn, ”FLAME: Formal Linear Algebra Methods Environment,” *ACM Transactions on Mathematical Software*, 27 (2001) 422-455.

> Paolo Bientinesi, Enrique S. Quintana-Orti, and Robert A. van de Geijn, ”Representing linear algebra algorithms in code: the FLAME application program interfaces,” ACM Transactions on Mathematical Software, 31 (2005) 27-59.

These last two papers can be accessed for free by visiting `http://www.cs.utexas.edu/~flame/web/publications.html`.

## 4.5 Wrap Up ☛ to edX

### 4.5.1 Additional exercises ☛ to edX

In this unit, we give a large number of vector-vector (level-1 BLAS) and matrix-vector (level-2 BLAS) operations for which you may want to try deriving algorithms. For the matrix-vector operations that involve symmetric or triangular matrices, you may want to wait until after you have finished Section 5.2.

#### Level-1 BLAS (vector-vector) operations

SAPDOT.

Earlier this week, you already derived algorithms for the SAPDOT (**s**calar **a**lpha **p**lus **dot** product) operation:

$$\alpha := x^T y + \alpha,$$

where $x$ and $y$ are column vectors. For completeness, you would want to derive algorithms for the cases where

- $x$ and $y$ are both row vectors,
- $x$ is a row vector and $y$ is a column vector, and
- $x$ is a column vector and $y$ is a row vector.

We might want to name the resulting algorithms and functions SAPDOT_CC_UNB_VARX( X, Y, ALPHA), SAPDOT_CR_UNB_VARX( x, y, alpha), SAPDOT_RC_UNB_VARX( x, y, alpha), and SAPDOT_RR_UNB_VARX( x, y, alpha), where CC, CR, RC, and RR indicate whether $x$ and $y$ are **c**olumn vector and/or **r**ow vectors. The X in VARX would then be replaced with a variant number.

AXPY.

Earlier this week, you also already derived algorithms for the AXPY (**al**pha times **x** **pl**us **y**) operation:

$$y := \alpha x + y,$$

where $x$ and $y$ are column vectors. For completeness, you would want to derive algorithms for the cases where

- $x$ and $y$ are both row vectors,
- $x$ is a row vector and $y$ is a column vector, and
- $x$ is a column vector and $y$ is a row vector.

We might want to name the resulting algorithms and functions
AXPY_CC_UNB_VARX(ALPHA, X, Y), AXPY_CR_UNB_VARX(ALPHA, X, Y),
AXPY_RC_UNB_VARX(ALPHA, X, Y), and AXPY_RR_UNB_VARX(ALPHA, X, Y).

COPY.

The COPY operation copies vector $x$ into vector $y$:

$$y := x.$$

In the obvious way this might yield
functions COPY_CC_UNB_VARX(X, Y), COPY_CR_UNB_VARX(X, Y),
COPY_RC_UNB_VARX(X, Y), and COPY_RR_UNB_VARX(X, Y).

SWAP.

The SWAP operation swaps the contents of vectors $x$ and $y$:

$$x, y := y, x.$$

In the obvious way this might yield
functions SWAP_CC_UNB_VARX(X, Y), SWAP_CR_UNB_VARX(X, Y),
SWAP_RC_UNB_VARX(X, Y), and SWAP_RR_UNB_VARX(X, Y).

**fused** SAPDOT/AXPY.

An operation that is at the core of what turns out to be the best implementation for symmetric matrix-vector multiplication (mentioned below and the topic of Sections 5.1 and 5.2 in Week 5) is a fused SAPDOT/AXPY operation:

$$\left\{ \begin{array}{lcl} y & := & \alpha x + y \\ \beta & := & x^T z + \beta \end{array} \right.$$

where $x$, $y$, and $z$ are vectors, and $\alpha$ and $\beta$ are scalars. The goal is to find a variant that "marches" through the elements of $x$ only once. (Why will become clear in Week 5.) This might yield the function SAPDOT_AXPY_UNB_VARX(ALPHA, X, Y, Z, BETA ).

### Level-2 BLAS (matrix-vector) operations

These operations involve a matrix and one or more vectors. This time, you can assume that the vectors are column vectors. The reason? If they are not, you can start by copying them into a column vector and/or finish by copying the result from a column vector. Since that requires $O(n)$ memory operations (where $n$ is the size of the vector) and the matrix-vector operation typically requires $O(mn)$ floating point and memory operations, the cost of the copy is not significant.

GEMV.

Earlier this week, you already derived algorithms for the GEMV (**ge**neral **m**atrix **v**ector multiplication) operation:

$$y := Ax + y,$$

where $x$ and $y$ are column vectors and $A$ is a matrix of appropriate size. This is a special case of the operation that is part of the BLAS, which includes all of the following operations:

$$\begin{aligned} y &:= \alpha A x + \beta y \\ y &:= \alpha A^T x + \beta y. \end{aligned}$$

(Actually, it includes even more if the matrix and vectors can be complex valued). The key is that matrix $A$ *not* be explicitly transposed because of the memory operations and/or extra space that would require. We suggest you ignore $\alpha$ and $\beta$. This then yields the algorithms/functions GEMV_N_UNB_VARX(A, X, Y) and GEMV_T_UNB_VARX(A, X, Y), where the N and T indicate whether $A$ is **n**ot transposed or **t**ransposed.

GER.

Earlier this week, you already derived algorithms for the GER (**ge**neral **r**ank-one) update :

$$A := xy^T + A,$$

where $x$ and $y$ are column vectors and $A$ is a matrix of appropriate size. This yields the algorithms/function GER_UNB_VARX(A, X, Y).

SYMV.

Week 5 starts with two sections on symmetric matrix-vector multiplication, $y := Ax + y$ where $A$ is symmetric and therefore only stored in the lower triangular part of $A$. Obviously, the matrix could instead be stored in the upper trianglar part of $A$. The key is that the symmetric matrix $A$ should *not* be explicitly formed. Instead, one computes with the data where it is stored. . This then yields the algorithms/functions SYMV_L_UNB_VARX(A, X, Y) and SYMV_U_UNB_VARX(A, X, Y), where the L and U indicate whether $A$ is stored in the **l**ower or **u**pper triangular part of $A$. You will find out that there are eight algorithmic variants for each of these cases.

You could try to derive them before you start Week 5! (We recommend you move on and then return to the remainder of these operations if you want more practice.)

SYR.

If matrix $A$ is symmetric, then so is the result of what is known as a symmetric rank-1 update (SYR): $A := xx^T + A$. In this case, only the lower or upper triangular part of $A$ needs to be stored and updated. This then yields the algorithms/functions SYR_L_UNB_VARX( X, A) and SYR_U_UNB_VARX( X, A), where the L and U indicate whether $A$ is stored in the **l**ower or **u**pper triangular part of $A$.

SYR2.

Similarly, if matrix $A$ is symmetric, then so is the result of what is known as a symmetric rank-2 update (SYR2): $A := xy^T + yx^T + A$. Again, only the lower or upper triangular part of $A$ needs to be stored and updated. This then yields the algorithms/functions SYR2_L_UNB_VARX( X, Y, A) and SYR2_U_UNB_VARX( X, Y, A), where the L and U indicate whether $A$ is stored in the **l**ower or **u**pper triangular part of $A$.

TRMV.

Another special case of matrix-vector multiplication is given by $y := Ay$, where $A$ is (lower or upper) triangular. It turns out that the output can overwrite the input vector $y$ if the computation is carefully ordered. Actually, there is a whole family of triangular matrix-vector multiplications:

$$\begin{array}{rcl} y & := & Ly \\ y & := & L^T y \\ y & := & Uy \\ y & := & U^T y \end{array}$$

where $L$ is a lower triangular (possibly implicitly unit lower triangular) and $U$ is an upper triangular (possibly implicitly unit upper triangular). Here unit triangular means the diagonal elements of the matrix are implicitly ones (unit). Implicit means the ones are not stored. This then yields the algorithms/functions

- TRMV_LNN_UNB_VARX(L, Y) where LNN stands for **l**ower triangular, **n**o transpose, **n**on unit diagonal,
- TRMV_LNU_UNB_VARX(L, Y) where LNU stands for **l**ower triangular, **n**o transpose, **u**nit diagonal,
- TRMV_LTN_UNB_VARX(L, Y) where LTN stands for **l**ower triangular, **t**ranspose, **n**on unit diagonal,
- TRMV_LTU_UNB_VARX(L, Y) where LTU stands for **l**ower triangular, **t**ranspose, **u**nit diagonal,
- TRMV_UNN_UNB_VARX(L, Y) where UNN stands for **u**pper triangular, **n**o transpose, **n**on unit diagonal,
- TRMV_UNU_UNB_VARX(L, Y) where UNU stands for **u**pper triangular, **n**o transpose, **u**nit diagonal,
- TRMV_UTN_UNB_VARX(L, Y) where UTN stands for **u**pper triangular, **t**ranspose, **n**on unit diagonal,
- TRMV_UTU_UNB_VARX(L, Y) where UTU stands for **u**pper triangular, **t**ranspose, **u**nit diagonal.

(You may want to wait until after Week 6 to do this operation.)

TRSV.

The final matrix-vector operation solves $Ax = y$ where $A$ is triangular, and the solution $x$ overwrites $y$. We discuss this operation in detail in Week 6.

### 4.5.2 Summary ☛ to edX

As discussed in the "opener" of this week, algorithms over two dimensional arrays (matrices) are complicated by the fact that they typically require double nested loops. This makes specifying the operation trickier and, if the algorithm uses explicit indexing, the opportunity for error greater, even if one uses the techniques we discussed in Part I of the course.

We are going to see how the FLAME notation comes to the rescue. But, in order to be able to hide the details of indices, we need to review (briefly) how operations that involve matrices and vectors act when those matrices and vectors are partitioned into submatrices and subvectors.

The good news is that we only need to be able to reason about matrices and vectors that are partitioned into two parts (like the Top and Bottom we saw in the last section for vectors) or quadrants. What you need to know is summarized in Figure 4.12.

We previously discussed the convention that we usually use

- Lower case Greek letters for scalars.
- Lower case Roman letters for (column) vectors.

We now add the convention that we will use

- Upper case Roman letters for matrices.

```
$\{ (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le n) \}$
$k := 0$
$\{ (\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n) \}$
while $k < n$ do
    $\{ (\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n) \wedge (k < n) \}$
    $y(k) := x(k)$
    $k := k + 1$
    $\{ (\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n) \}$
endwhile
$\{ (\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n) \wedge \neg(k < n) \}$
$\{ (\forall i \mid 0 \le i < n : y(i) = x(i)) \}$
```

Figure 4.2: Annotated algorithm for copying $x$ into $y$, from first element to last element.

```
{ $(\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le n)$    $y = \widehat{y}$ }
$k := 0$    $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ and $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$ where $x_T$ and $y_T$ are empty
{ $(\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge$
  $(\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n)$    $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right)$ }
while    $k < n$    do
    { $P_{\rm inv} \wedge (k < n)$ :
      $(\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge$
      $(\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n)$  $\wedge (k < n)$    $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \wedge$ }

    { $P_{\rm inv} \wedge (k < n)$ with $k$ term split off) :
      $(\forall i \mid 0 \le i < k : y(i) = x(i))$  $\wedge$  $((y(k) = \widehat{y}(k)) \wedge$
      $(\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i))$  $\wedge$  $(0 \le k \le n) \wedge (k < n)$ }
    $S$ :  $y(k) := x(k)$
    { wp("$k := k+1$", $P_{\rm inv}$) (with $k$ term split off) :
      $(\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge (y(k) = x(k))$
      $\wedge(\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k+1 \le n)$ }
    $k := k+1$
    { $(\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge$
      $(\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n)$    $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right)$ }
endwhile
{ $(\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge$
  $(\forall i \mid k \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k \le n)$    $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \wedge \neg($          $)$
  $\wedge \neg(k < n)$ }
{ $(\forall i \mid 0 \le i < n : y(i) = x(i))$    $y = x$ }
```

```
$\{ \; y = \widehat{y} \; \}$
$x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ and $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$
    where $x_T$ and $y_T$ are empty
$\left\{ \; \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \; \right\}$
while $m(y_T) < m(y)$ do
    $\left\{ \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \wedge (m(y_T) < m(y)) \right\}$
    $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$
        where $\chi_1$ and $\psi_1$ are scalars
    $\left\{ \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right)$ with split range: $\left(\begin{array}{c} y_0 \\ \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \widehat{\psi}_1 \\ \widehat{y}_2 \end{array}\right) \right\}$
    $\psi_1 := \chi_1$
    $\left\{ \text{wp(“} \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) := \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right); \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) := \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right) \text{”}, \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right)) : \left(\begin{array}{c} y_0 \\ \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \chi_1 \\ \widehat{y}_2 \end{array}\right) \right\}$
    $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$
    $\left\{ \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \right\}$
endwhile
$\left\{ \; \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \wedge \neg(m(y_T) < m(y)) \; \right\}$
$\{ \; y = x \; \}$
```

Figure 4.3: Annotated algorithm for copying $x$ into $y$ from the first element to the last element, using the FLAME notation.

| |
|---|
| $\{ \; y = \widehat{y} \; \}$ |
| where |
| $\{ \quad \}$ |
| **while** **do** |
| $\{ \quad \}$ |
| where |
| $\{ \quad \}$ |
| |
| $\{ \quad \}$ |
| |
| $\{ \quad \}$ |
| **endwhile** |
| $\{ \quad \}$ |
| $\{ \; y = x \; \}$ |

Figure 4.4: Blank worksheet for Homework 4.2.1.2.

| Step | Algorithm: $\alpha := x^T y + \alpha$ |
|---|---|
| 1a | $\{\alpha = \widehat{\alpha}$ $\}$ |
| 4 | where |
| 2 | $\{\alpha = x_T^T y_T + \widehat{\alpha}$ $\}$ |
| 3 | while do |
| 2,3 | $\{$ $\alpha = x_T^T y_T + \widehat{\alpha} \wedge$ $\}$ |
| 5a | where |
| 6 | $\{$ $\}$ |
| 8 | |
| 7 | $\{$ $\}$ |
| 5b | |
| 2 | $\{$ $\alpha = x_T^T y_T + \widehat{\alpha}$ $\}$ |
| | endwhile |
| 2,3 | $\{\alpha = x_T^T y_T + \widehat{\alpha} \wedge \neg($ $)$ $\}$ |
| 1b | $\{\alpha = x^T y + \widehat{\alpha}$ $\}$ |

Figure 4.5: Partial derivation of algorithm for computing $\alpha := x^T y + \alpha$ after Step 2.

| Step | Algorithm: $\alpha := x^T y + \alpha$ |
|---|---|
| 1a | $\{ \alpha = \widehat{\alpha} \}$ |
| 4 | $x \rightarrow \left( \begin{array}{c} x_T \\ \hline x_B \end{array} \right)$, $y \rightarrow \left( \begin{array}{c} y_T \\ \hline y_B \end{array} \right)$ where $x_T$ has 0 rows, $y_T$ has 0 rows |
| 2 | $\{ \alpha = x_T^T y_T + \widehat{\alpha} \}$ |
| 3 | while $m(x_T) < m(x)$ do |
| 2,3 | $\{ \alpha = x_T^T y_T + \widehat{\alpha} \wedge m(x_T) < m(x) \}$ |
| 5a | where |
| 6 | $\{ \}$ |
| 8 | |
| 7 | $\{ \}$ |
| 5b | |
| 2 | $\{ \alpha = x_T^T y_T + \widehat{\alpha} \}$ |
| | endwhile |
| 2,3 | $\{ \alpha = x_T^T y_T + \widehat{\alpha} \wedge \neg (m(x_T) < m(x)) \}$ |
| 1b | $\{ \alpha = x^T y + \widehat{\alpha} \}$ |

Figure 4.6: Partial derivation of algorithm for computing $\alpha := x^T y + \alpha$ after Step 4.

| Step | Algorithm: $\alpha := x^T y + \alpha$ |
|---|---|
| 1a | $\{\alpha = \widehat{\alpha}$ $\}$ |
| 4 | $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$ where $x_T$ has 0 rows, $y_T$ has 0 rows |
| 2 | $\{\alpha = x_T^T y_T + \widehat{\alpha}$ $\}$ |
| 3 | while $m(x_T) < m(x)$ do |
| 2,3 | $\{\ \alpha = x_T^T y_T + \widehat{\alpha} \wedge m(x_T) < m(x)$ $\}$ |
| 5a | $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$ where $\chi_1$ has 1 element, $\psi_1$ has 1 element |
| 6 | $\{$ $\}$ |
| 8 | |
| 7 | $\{$ $\}$ |
| 5b | $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$ |
| 2 | $\{\ \alpha = x_T^T y_T + \widehat{\alpha}$ $\}$ |
| | endwhile |
| 2,3 | $\{\alpha = x_T^T y_T + \widehat{\alpha} \wedge \neg(m(x_T) < m(x))$ $\}$ |
| 1b | $\{\alpha = x^T y + \widehat{\alpha}$ $\}$ |

Figure 4.7: Partial derivation of algorithm for computing $\alpha := x^T y + \alpha$ after Step 5.

| Step | Algorithm: $\alpha := x^T y + \alpha$ |
|---|---|
| 1a | $\{\alpha = \widehat{\alpha}\}$ |
| 4 | $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right), y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$ where $x_T$ has 0 rows, $y_T$ has 0 rows |
| 2 | $\{\alpha = x_T^T y_T + \widehat{\alpha}\}$ |
| 3 | while $m(x_T) < m(x)$ do |
| 2,3 | $\{\alpha = x_T^T y_T + \widehat{\alpha} \wedge m(x_T) < m(x)\}$ |
| 5a | $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$ where $\chi_1$ has 1 element, $\psi_1$ has 1 element |
| 6 | $\{\alpha = x_0^T y_0 + \widehat{\alpha}\}$ |
| 8 | |
| 7 | $\{\alpha = x_0^T y_0 + \chi_1 \times \psi_1 + \widehat{\alpha}\}$ |
| 5b | $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$ |
| 2 | $\{\alpha = x_T^T y_T + \widehat{\alpha}\}$ |
| | endwhile |
| 2,3 | $\{\alpha = x_T^T y_T + \widehat{\alpha} \wedge \neg(m(x_T) < m(x))\}$ |
| 1b | $\{\alpha = x^T y + \widehat{\alpha}\}$ |

Figure 4.8: Partial derivation of algorithm for computing $\alpha := x^T y + \alpha$ after Step 7.

| Step | Algorithm: $\alpha := x^T y + \alpha$ |
|---|---|
| 1a | $\{\alpha = \widehat{\alpha}$ $\}$ |
| 4 | $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$ where $x_T$ has 0 rows, $y_T$ has 0 rows |
| 2 | $\{\alpha = x_T^T y_T + \widehat{\alpha}$ $\}$ |
| 3 | while $m(x_T) < m(x)$ do |
| 2,3 | $\{$ $\alpha = x_T^T y_T + \widehat{\alpha} \wedge m(x_T) < m(x)$ $\}$ |
| 5a | $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$ where $\chi_1$ has 1 element, $\psi_1$ has 1 element |
| 6 | $\{$ $\alpha = x_0^T y_0 + \widehat{\alpha}$ $\}$ |
| 8 | $\alpha := \chi_1 \times \psi_1 + \alpha$ |
| 7 | $\{$ $\alpha = x_0^T y_0 + \chi_1 \times \psi_1 + \widehat{\alpha}$ $\}$ |
| 5b | $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$ |
| 2 | $\{$ $\alpha = x_T^T y_T + \widehat{\alpha}$ $\}$ |
| | endwhile |
| 2,3 | $\{\alpha = x_T^T y_T + \widehat{\alpha} \wedge \neg(m(x_T) < m(x))$ $\}$ |
| 1b | $\{\alpha = x^T y + \widehat{\alpha}$ $\}$ |

Figure 4.9: Completed worksheet for computing $\alpha := x^T y + \alpha$.

```
Algorithm: $\alpha := x^T y + \alpha$
$x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ , $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$
  where $x_T$ has 0 rows, $y_T$ has 0 rows
while $m(x_T) < m(x)$ do
    $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$ , $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$
      where $\chi_1$ has 1 element, $\psi_1$ has 1 element

    $\alpha := \chi_1 \times \psi_1 + \alpha$

    $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$ , $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$
endwhile
```

Figure 4.10: Algorithm for computing $\alpha := x^T y + \alpha$ from first element to last element.

```
Algorithm: $y := \text{GEMV\_UNB\_VAR}1(A, x, y)$
$A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$
  where $A_T$ has 0 rows, $y_T$ has 0 rows
while $m(A_T) < m(A)$ do
    $\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline a_1^T \\ A_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$
      where $a_1$ has 1 row, $\psi_1$ has 1 row
    $\psi_1 := a_1^T x + \psi_1$
    $\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ a_1^T \\ \hline A_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$
endwhile
```

```
Algorithm: $y := \text{GEMV\_UNB\_VAR}3(A, x, y)$
$A \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$
  where $A_L$ has 0 columns, $x_T$ has 0 rows
while $n(A_L) < n(A)$ do
    $\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_0 & a_1 & A_2 \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$
      where $a_1$ has 1 column, $\chi_1$ has 1 row
    $y := \chi_1 a_1 + y$
    $A \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_0 & a_1 & A_2 \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$
endwhile
```

Figure 4.11: Variants 1 (top) and 3 (bottom) for computing $y := Ax + y$ in FLAME notation.

Consider vectors $x$ and $y$ that are partitioned like $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ and $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$, and matrices $A$, $B$, and $C$ that have been partitioned like $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$, $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$, and $C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$. Then, provided the sizes of the matrices and vectors are conformal (sizes match correctly),

$$x^T = \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)^T = \left(\begin{array}{c|c} x_T^T & x_B^T \end{array}\right) \qquad A^T = \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)^T = \left(\begin{array}{c|c} A_T^T & A_B^T \end{array}\right)$$

$$B^T = \left(\begin{array}{c|c} B_L & B_R \end{array}\right)^T = \left(\begin{array}{c} B_L^T \\ \hline B_R^T \end{array}\right) \qquad C^T = \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)^T = \left(\begin{array}{c|c} C_{TL}^T & C_{BL}^T \\ \hline C_{TR}^T & C_{BR}^T \end{array}\right)$$

$$Ax = \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) x = \left(\begin{array}{c} A_T x \\ \hline A_B x \end{array}\right) \qquad x^T B = x^T \left(\begin{array}{c|c} B_L & B_R \end{array}\right) = \left(\begin{array}{c|c} x^T B_L & x^T B_R \end{array}\right)$$

$$Bx = \left(\begin{array}{c|c} B_L & B_R \end{array}\right) \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) = B_L x_T + B_R x_B \qquad x^T A = \left(\begin{array}{c|c} x_T^T & x_B^T \end{array}\right) \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) = x_T^T A_T + x_B^T B_B$$

$$Cx = \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) = \left(\begin{array}{c} C_{TL} x_T + C_{TR} x_B \\ \hline C_{BL} x_T + C_{BR} x_B \end{array}\right)$$

$$x^T C = \left(\begin{array}{c|c} x_T^T & x_B^T \end{array}\right) \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c|c} x_T^T C_{TL} + x_B^T C_{BL} & x_T^T C_{TR} + x_B^T C_{BR} \end{array}\right)$$

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) y^T = \left(\begin{array}{c} x_T y^T \\ \hline x_B y^T \end{array}\right) \qquad x \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)^T = \left(\begin{array}{c|c} x y_T^T & x y_B^T \end{array}\right)$$

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)^T = \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \left(\begin{array}{c|c} y_T^T & y_B^T \end{array}\right) = \left(\begin{array}{c|c} x_T y_T^T & x_T y_B^T \\ \hline x_B y_T^T & x_B y_B^T \end{array}\right)$$

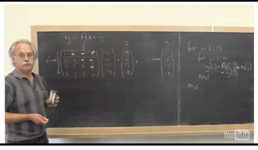
Figure 4.12: Summary of partitioned matrix-vector multiplication and rank-1 update (outer product).

Week 5

# Matrix-Matrix Operations

## 5.1 Opening Remarks ☛ to edX

### 5.1.1 Launch ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

In order to truly appreciate what the FLAME notation and API bring to the table, it helps to look at a programming problem that on the surface seems straightforward, but turns out to be trickier than expected. When programming with indices, coming up with *an* algorithm turns out to be relatively simple. But, when the goal is to, for example, access memory in a favorable pattern, finding an appropriate algorithm is sometimes more difficult.

In this launch, you experience this by executing algorithms from last week by hand. Then, you examine how these algorithms can be implemented with for loops and indices. The constraint that the matrices are symmetric is then added into the mix. Finally, you are asked to find an algorithm that takes advantage of symmetry in storage yet accesses the elements of the matrix in a beneficial order. The expectation is that this will be a considerable challenge.

**Homework 5.1.1.1** Compute

$$\begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} =$$

using algorithmic Variant 1 given in Figure 5.1.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

In Figure 5.1 we show Variant 1 for $y := Ax + y$ in FLAME notation and below it, in Figure 5.2, a more traditional implementation in MATLAB. To understand it easily, we use the convention that the index $i$ is used to keep track of the current row. In the algorithm expressed with FLAME notation this would be $a_1^T$. The $j$ index is then used for the loop that updates

$$\psi_1 := a_1^T x + \psi_1,$$

which you hopefully recognize as a dot product (or, more precisely, a sapdot) operation.

```
Algorithm: $y := \text{GEMV\_UNB\_VAR1}(A, x, y)$
$A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$
    where $A_T$ has 0 rows, $y_T$ has 0 elements
while $m(A_T) < m(A)$ do
    $\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline a_1^T \\ A_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$
        where $a_1^T$ is a row, $\psi_1$ is a scalar

    $\psi_1 := a_1^T x + \psi_1$

    $\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ a_1^T \\ \hline A_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$
endwhile
```

Figure 5.1: Variant 1 for computing $y := Ax + y$ in FLAME notation.

```matlab
function [ y_out ] = MatVec1( A, x, y )
% Compute y := A x + y

% Extract the row and column size of A
[ m, n ] = size( A );

% (Strictly speaking you should check that x is a vector size n and y is a
% vector of size m...)

% Copy y into y_out
y_out = y;

% Compute y_out = A * x + y_out
for i = 1:m
    for j=1:n
        y_out( i ) = A( i,j ) * x( j ) + y_out( i );
    end
end

end
```

LAFFPfC/Assignments/Week5/matlab/MatVec1.m

Figure 5.2: Function that computes $y := Ax + y$, returning the result in vector `y_out`.

.

```
function [ y_out ] = MatVec1( A, x, y )
% Compute y := A x + y

% Extract the row and column size of A
[ m, n ] = size( A );

% (Strictly speaking you should check that x
     is a vector size n and y is a
% vector of size m...)

% Copy y into y_out
y_out = y;

% Compute y_out = A * x + y_out
for i = 1:m
    for j=1:n
        y_out( i ) = A( i,j ) * x( j ) +
    y_out( i );
    end
end

end
```

LAFFPfC/Assignments/Week5/matlab/MatVec1.m

```
function [ y_out ] = SymMatVec1( A, x, y )
% Compute y := A x + y, assuming A is
    symmetric and stored in lower
% triangular part of array A.

% Extract the row and column size of A
[ m, n ] = size( A );

% (Strictly speaking you should check that m
     = n, x is a vector size n and y is a
% vector of size n...)

% Copy y into y_out
y_out = y;

% Compute y_out = A * x + y_out
for i = 1:n
    for j=1:i
        y_out( i ) = A( i,j ) * x( j ) +
    y_out( i );
    end
    for j=i+1:n
        y_out( i ) = A( j,i ) * x( j ) +
    y_out( i );
    end
end

end
```

LAFFPfC/Assignments/Week5/matlab/SymMatVec1.m

Figure 5.3: Functions that compute $y := Ax + y$, returning the result in vector `y_out`. On the right, matrix $A$ is assumed to be symmetric and only stored in the lower triangular part of array `A`.

**Homework 5.1.1.2** Download the Live Script `MatVec1LS.mlx` into `Assignments/Week5/matlab/` and follow the directions in it to execute function `MatVec1`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

Now, if $m = n$ then matrix $A$ is square and if the elements indexed with $i, j$ and $j, i$ are equal ($A(i, j) = A(j, i)$) then it is said to be a symmetric matrix.

**Homework 5.1.1.3** Knowing that the matrix is symmetric, compute

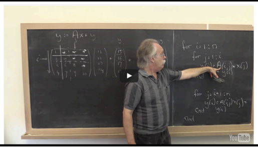
$$\begin{pmatrix} 1 & \star & \star \\ -2 & 2 & \star \\ -1 & 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} =$$

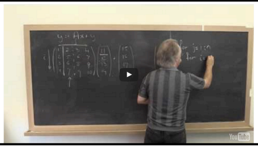
using algorithmic Variant 1 given in Figure 5.1.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.1.1.4** Download the Live Script `SymVec1LS.mlx` into `Assignments/Week5/matlab/` and follow the directions in it to change the given function to only compute with the lower triangular part of the matrix.

☛ SEE ANSWER
☛ DO EXERCISE ON edX


☛ Watch Video on edX
☛ Watch Video on YouTube


☛ Watch Video on edX
☛ Watch Video on YouTube

Now, MATLAB stores matrices in *column-major* order, which means that a matrix

$$\begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix}$$

is stored in memory by stacking columns:

$$\begin{array}{|r|} \hline 1 \\ -2 \\ -1 \\ \hline -1 \\ 2 \\ 1 \\ \hline 2 \\ 0 \\ -2 \\ \hline \end{array}$$

Computation tends to be more efficient if one accesses memory contiguously. This means that an algorithm that accesses $A$ by columns often computes the answer faster than one that accesses $A$ by rows.

In a linear algebra course you should have learned that,

$$\begin{aligned} \begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} &= (2) \begin{pmatrix} 1 \\ -2 \\ -1 \end{pmatrix} + (-1) \begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix} + (1) \begin{pmatrix} 2 \\ 0 \\ -2 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} \\ &= \left[ \left[ \left[ \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} + (2) \begin{pmatrix} 1 \\ -2 \\ -1 \end{pmatrix} \right] + (-1) \begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix} \right] + (1) \begin{pmatrix} 2 \\ 0 \\ -2 \end{pmatrix} \right], \end{aligned}$$

which is exactly how Variant 3 for computing $y := Ax + y$, given in Figure 5.4, proceeds. It also means that the implementation in Figure 5.2 can be rewritten as the one in Figure 5.5. The two implementations in Figures 5.2 and 5.5 differ only in the order of the loops indexed by `i` and `j`.

```
Algorithm: $y := \text{GEMV\_UNB\_VAR3}(A, x, y)$

$A \rightarrow \left( A_L \,\middle|\, A_R \right)$, $x \rightarrow \left( \begin{array}{c} x_T \\ \hline x_B \end{array} \right)$
  where $A_L$ has 0 columns, $x_T$ has 0 rows
while $n(A_L) < n(A)$ do
    $\left( A_L \,\middle|\, A_R \right) \rightarrow \left( A_0 \,\middle|\, a_1 \;\; A_2 \right)$, $\left( \begin{array}{c} x_T \\ \hline x_B \end{array} \right) \rightarrow \left( \begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array} \right)$
      where $a_1$ has 1 column, $\chi_1$ has 1 row

    $y := \chi_1 a_1 + y$

    $A \rightarrow \left( A_L \,\middle|\, A_R \right) \leftarrow \left( A_0 \;\; a_1 \,\middle|\, A_2 \right)$, $\left( \begin{array}{c} x_T \\ \hline x_B \end{array} \right) \leftarrow \left( \begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array} \right)$
endwhile
```

Figure 5.4: Variant 3 for computing $y := Ax + y$ in FLAME notation.

```
function [ y_out ] = MatVec3( A, x, y )
% Compute y := A x + y

% Extract the row and column size of A
[ m, n ] = size( A );

% (Strictly speaking you should check that x is a vector size n and y is a
% vector of size m...)

% Copy y into y_out
y_out = y;

% Compute y_out = A * x + y_out
for j = 1:n
    for i=1:m
        y_out( i ) = A( i,j ) * x( j ) + y_out( i );
    end
end

end
```

LAFFPfC/Assignments/Week5/matlab/MatVec3.m

Figure 5.5: Function that computes $y := Ax + y$, returning the result in vector `y_out`.

.

**Homework 5.1.1.5** Knowing that the matrix is symmetric, compute

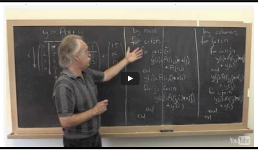
$$\begin{pmatrix} 1 & \star & \star \\ 2 & -2 & \star \\ -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} =$$

using algorithmic Variant 3 given in Figure 5.4.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.1.1.6** Download the Live Script ☛ `SymVec3LS.mlx` into `Assignments/Week5/matlab/` and follow the directions in it to change the given function to only compute with the lower triangular part of the matrix.

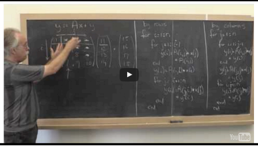
☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 5.1.1.7** Which algorithm for computing $y := Ax + y$ casts more computation in terms of the columns of the stored matrix (and is therefore probably higher performing)?

☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Now we get to two exercises that we believe demonstrate the value of our notation and systematic derivation of algorithms. They are surprisingly hard, even for an experts. Don't be disappointed if you can't work it out! The answer comes later in the week.

**Homework 5.1.1.8 (Challenge)** Download the Live Script `SymMatVecByColumnsLS.mlx` into `Assignments/Week5/matlab/` and follow the directions in it to change the given function to only compute with the lower triangular part of the matrix **and** only access the matrix by columns. (Not sort-of-kind-of as in `SymMatVec3.mlx`.)

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.1.1.9 (Challenge)** Find someone who knows a little (or a lot) about linear algebra and convince this person that the answer to the last exercise is correct. Alternatively, if you did not manage to come up with an answer for the last exercise, look at the answer to that exercise and convince yourself it is correct.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

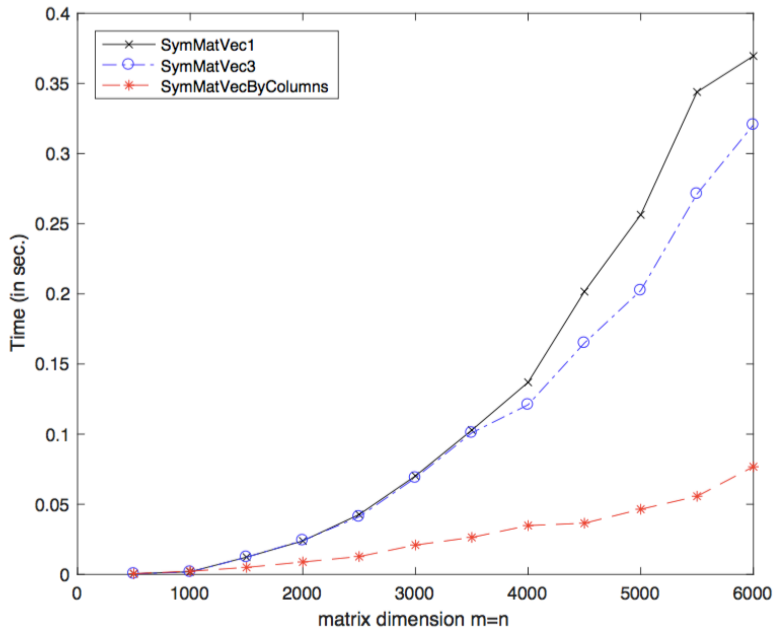
The point of these last two exercises is:

- It is difficult to find algorithms with specific (performance) properties even for relatively simple operations. The problem: the traditional implementation involves a double nested loop, which makes the application of what you learned in Week 3 bothersome.
- It is still difficult to give a convincing argument that even a relatively simple algorithm is correct, even after you have completed Week 2. The problem: proving a double loop correct.

One could ask "But isn't having any algorithm to compute the result good enough?" The graph in Figure 5.6 illustrates the difference in performance of the different implementations (coded in C). The implementation that corresponds to `SymMatVecByColumns` is roughly five times faster than the other implementations. It demonstrates there is a definite performance gain that results from picking the right algorithm.

What you will find next is that the combination of our new notation and the application of systematic derivation provides the solution, in Unit 5.2.6.

While we discuss efficiency here, implementing the algorithms as we do in MATLAB generally means they don't execute particularly efficiently. If you execute `A * x` in MATLAB, this is typically translated into a call to a high-performance implementation. But implementing it yourself in MATLAB, as loops or with our FLAME API, is not particularly efficient. We do it to illustrate algorithms. One would want to implement these same algorithms in a language that enables high-performance, like C. We have a FLAME API for C as well.



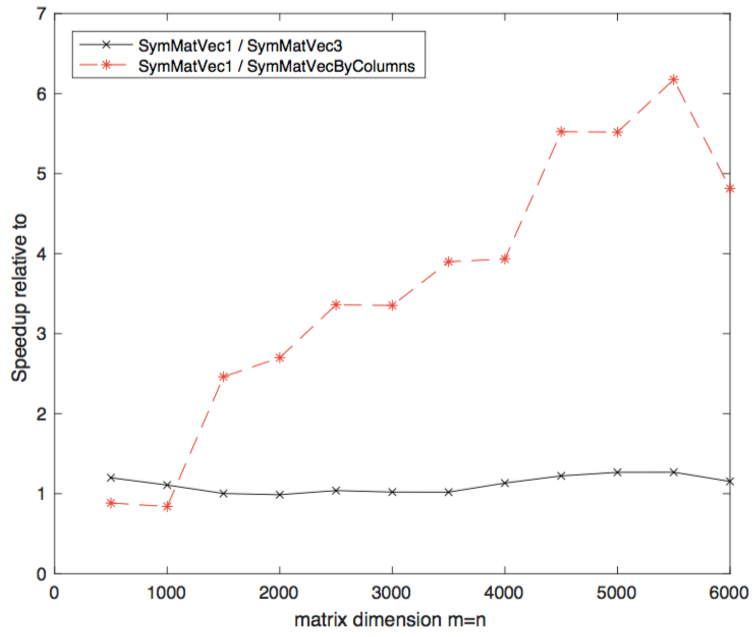


Figure 5.6: Execution time (top) and speedup (bottom) as a function of matrix size for the different implementations of symmetric matrix-vector multiplication.

### 5.1.2 Outline Week 5 ☛ to edX

### 5.1.3 What you will learn ☛ to edX

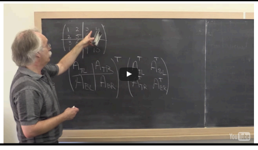
While the FLAME notation has allowed us to abstract away from those nasty indices, so far the operations used to illustrate this have been simple enough that they could have been derived with the techniques from Week 3. We now see that the FLAME notation facilitates the derivation of families of algorithms for progressively more complicated operations with matrices and vectors, yielding some algorithms that are not easily found without it.

Upon completion of this week, you should be able to

- Multiply with partitioned matrices to take advantage of special structure.
- Derive partitioned matrix expressions for matrix-vector and matrix-matrix operations.
- Recognize that a particular operation can have several PMEs each with multiple loop invariants.
- Enumerate candidate loop invariants for matrix operations from their PMEs and eliminate loop invariants that do not show promise.
- Accomplish a complete derivation and implementation of an algorithm.

## 5.2 Partitioning matrices into quadrants ☛ to edX

### 5.2.1 Background ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Consider the matrix-vector operation $Ax$ where $A$ and $x$ are of appropriate sizes so that this multiplication makes sense. Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad \text{and} \quad x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right).$$

Then

$$Ax = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + A_{TR}x_B \\ \hline A_{BL}x_T + A_{BR}x_B \end{array}\right)$$

provided $x_T$ and $x_B$ have the appropriate size for the subexpressions to be well-defined.

Now, if $A$ is symmetric, then $A = A^T$. For the partitioned matrix this means that

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)^T = \left(\begin{array}{c|c} A_{TL}^T & A_{BL}^T \\ \hline A_{TR}^T & A_{BR}^T \end{array}\right)$$

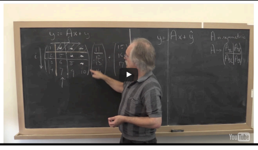
If $A_{TL}$ is square (and hence so is $A_{BR}$ since $A$ itself is), then we conclude that

- $A_{TL}^T = A_{TL}$ and hence $A_{TL}$ is symmetric.
- $A_{BR}^T = A_{BR}$ and hence $A_{BR}$ is symmetric.
- $A_{TR} = A_{BL}^T$ and $A_{BL} = A_{TR}^T$. Thus, if $A_{TR}$ is not stored, one can compute with $A_{BL}^T$ instead. Notice that one need not explicitly transpose the matrix. In MATLAB the command $A' * x$ will compute $A^T x$.

Hence, for a partitioned symmetric matrix where $A_{TL}$ is square, one can compute with $\left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$ if $A_{TR}$ is not available (e.g., is not stored) or $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{TR}^T & A_{BR} \end{array}\right)$ if $A_{BL}$ is not available (e.g., is not stored). In the first case,

$$Ax = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) = \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B \\ \hline A_{BL}x_T + A_{BR}x_B \end{array}\right).$$

### 5.2.2 Example: Deriving algorithms for symmetric matrix-vector multiplication ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

The operation we wish to implement is mathematically given by $y := Ax + y$, where $A$ is a symmetric matrix (and hence square) and only the lower triangular part of matrix $A$ can be accessed, because (for example) the strictly upper triangular part is not stored.

**Step 1: Precondition and postcondition**

We are going to implicitly remember that $A$ is symmetric and only the lower triangular part of the matrix is stored. So, in the postcondition we simply state that $y = Ax + \widehat{y}$ is to be computed.

**Step 2: Deriving loop invariants**

Since matrix $A$ is symmetric, we want to partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

where $A_{TL}$ is square since then, because of the symmetry of $A$, we know that

- $A_{TL}$ and $A_{BR}$ are symmetric,
- $A_{TR} = A_{BL}^T$, and
- if we partition

  $$x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \quad \text{and} \quad y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$$

  then entering the partitioned matrix and vectors into the postcondition $y = Ax + \widehat{y}$ yields

  $$\begin{aligned} \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) &= \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) + \left(\begin{array}{c} \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right) \\ &= \left(\begin{array}{c} A_{TL}x_T + A_{TR}x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right) \\ &= \left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right) \quad \text{since } A_{TR} \text{ is not to be used.} \end{aligned}$$

This last observation gives us our PME for this operation:

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right).$$

**Homework 5.2.2.1** Below on the left you find four loop invariants for computing $y := Ax + y$ where $A$ has no special structure. On the right you find four loop invariants for computing $y := Ax + y$ when $A$ is symmetric and stored in the lower triangular part of $A$. Match the loop invariants on the right to the loop invariants on the left that you would expect maintain the same values in $y$ before and after each iteration of the loop. (In the video, we mentioned asking you to find two invariants. We think you can handle finding these four!)

(1) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} \widehat{y}_T \\ \hline A_B x + \widehat{y}_B \end{array}\right)$

(2) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} A_T x + \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right)$

(3) $y = A_L x_T + \widehat{y}$

(4) $y = A_R x_B + \widehat{y}$

(a) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BR} x_B + \widehat{y}_B \end{array}\right)$

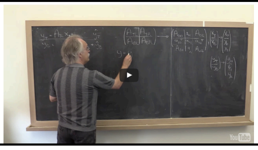
(b) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} \widehat{y}_T \\ \hline A_{BL} x_T + A_{BR} x_B + \widehat{y}_B \end{array}\right)$

(c) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{lr} A_{TL} x_T + & \widehat{y}_T \\ \hline A_{BL} x_T + & \widehat{y}_B \end{array}\right)$

(d) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} A_{TL} x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right)$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube

Now, how do we come up with possible loop invariants? Each term in the PME is either included or not. This gives one a table of **candidate** loop invariants, given in Figure 5.7. But not all of these candidates will lead to a valid algorithm. In particular, any valid algorithm must include exactly one of the terms $A_{TL}x_T$ or $A_{BR}x_B$. The reasons?

- Since $A_{TL}$ and $A_{BR}$ must be square submatrices, when the loop completes one of them must be the entire matrix $A$ while the other matrix is empty. But that means that one of the two terms must be included in the loop invariant, since otherwise the loop invariant, together with the loop guard becoming false, will not imply the postcondition.

- If both $A_{TL}x_T$ and $A_{BR}x_B$ are in the loop invariant, there is no simple initialization step that places the variables in a state where the loop invariant is TRUE. Why? Because if one of the two matrices $A_{TL}$ and $A_{BR}$ is empty, then the other one is the whole matrix $A$, and hence the final result must be computed as part of the initialization step.

We conclude that **exactly one** of the terms $A_{TL}x_T$ and $A_{BR}x_B$ can and must appear in the loop invariant, leaving us with the loop invariants tabulated in Figure 5.8.

| | $A_{TL}x_T$ | $A_{BL}^T x_B$ | $A_{BL}x_T$ | $A_{BR}x_B$ | $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) =$ | |
|---|---|---|---|---|---|---|
| A | No | No | No | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| B | Yes | No | No | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| C | No | Yes | No | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| D | Yes | Yes | No | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| E | No | No | Yes | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| F | Yes | No | Yes | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| G | No | Yes | Yes | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| H | Yes | Yes | Yes | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| I | No | No | No | Yes | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| J | Yes | No | No | Yes | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| K | No | Yes | No | Yes | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| L | Yes | Yes | No | Yes | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| M | No | No | Yes | Yes | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| N | Yes | No | Yes | Yes | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| O | No | Yes | Yes | Yes | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |
| P | Yes | Yes | Yes | Yes | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | |

Figure 5.7: Candidates for loop-invariants for $y := Ax + y$ where $A$ is symmetric and only its lower triangular part is stored.

$$\text{PME: } \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right).$$

| $A_{TL}x_T$ | $A_{BL}^T x_B$ | $A_{BL}x_T$ | $A_{BR}x_B$ | $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) =$ | Invariant # |
|---|---|---|---|---|---|
| Yes | No | No | No | $\left(\begin{array}{c} A_{TL}x_T + \textcolor{lightgray}{A_{BL}^T x_B +} \widehat{y}_T \\ \hline \textcolor{lightgray}{A_{BL}x_T + A_{BR}x_B +} \widehat{y}_B \end{array}\right)$ | 1 |
| Yes | Yes | No | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline \textcolor{lightgray}{A_{BL}x_T + A_{BR}x_B +} \widehat{y}_B \end{array}\right)$ | 2 |
| Yes | No | Yes | No | $\left(\begin{array}{c} A_{TL}x_T + \textcolor{lightgray}{A_{BL}^T x_B +} \widehat{y}_T \\ \hline A_{BL}x_T + \textcolor{lightgray}{A_{BR}x_B +} \widehat{y}_B \end{array}\right)$ | 3 |
| Yes | Yes | Yes | No | $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + \textcolor{lightgray}{A_{BR}x_B +} \widehat{y}_B \end{array}\right)$ | 4 |
| No | Yes | Yes | Yes | $\left(\begin{array}{c} \textcolor{lightgray}{A_{TL}x_T +} A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | 5 |
| No | Yes | No | Yes | $\left(\begin{array}{c} \textcolor{lightgray}{A_{TL}x_T +} A_{BL}^T x_B + \widehat{y}_T \\ \hline \textcolor{lightgray}{A_{BL}x_T +} A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | 6 |
| No | No | Yes | Yes | $\left(\begin{array}{c} \textcolor{lightgray}{A_{TL}x_T + A_{BL}^T x_B +} \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | 7 |
| No | No | No | Yes | $\left(\begin{array}{c} \textcolor{lightgray}{A_{TL}x_T + A_{BL}^T x_B +} \widehat{y}_T \\ \hline \textcolor{lightgray}{A_{BL}x_T +} A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | 8 |

Figure 5.8: Loop-invariants for $y := Ax + y$ where $A$ is symmetric and only its lower triangular part is stored.

### 5.2.3 One complete derivation ☛ to edX

In this unit, we continue the derivation started in Unit 5.2.2, with the loop invariant

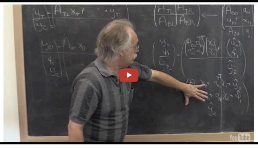
$$\text{Invariant 1:} \quad \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right).$$

**Homework 5.2.3.1** You may want to derive the algorithm corresponding to Invariant 1 yourself, consulting the video if you get stuck. Some resources:

- The ☛ blank worksheet.
- Download ☛ `symv_unb_var1_ws.tex` and place it in `LAFFPfC/Assignments/Week5/LaTeX/`. You will need ☛ `color_flatex.tex` as well in that directory.
- The ☛ Spark webpage.

Alternatively, you may want to download the completed worksheet (with intermediate steps later in the PDF) ☛ `symv_unb_var1_ws_answer.pdf` and/or its source ☛ `symv_unb_var1_ws_answer.tex`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

**Step 3: Determining the loop-guard.**

The condition

$$P_{\text{inv}} \wedge \neg G \quad \equiv \quad \left( \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right) \right) \wedge \neg G$$

must imply that

$$R : y = Ax + \widehat{y}$$

holds. The loop guard $m(A_{TL}) < m(A)$ has the desired property.

**Step 4: Initialization.**

When we derived the PME in Step 2, we decided to partition the matrices like

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right), \quad \text{and} \quad y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right).$$

The question now is how to choose the sizes of the submatrices and vectors so that the precondition

$$y = \widehat{y}$$

implies that the loop invariant

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right)$$

holds after the initialization (and before the loop commences). The initialization

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right), y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$$

where $A_{TL}$ is $0 \times 0$, and $x_T$ and $y_T$ have 0 elements, has the desired property.

**Step 5: Progressing through the matrix and vectors.**

We now note that, as part of the computation, $A_{TL}$, $x_T$ and $y_T$ start by containing no elements and must ultimately equal all of $A$, $x$ and $y$, respectively. Thus, as part of the loop in Step 5a, the top elements of $x_B$ and $y_B$ are exposed by

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right).$$

They are added to $x_T$ and $y_T$ with

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right).$$

Similarly, rows of $A$ are exposed

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ A_{20} & a_{21} & A_{22} \end{array}\right)$$

and "moved", in Step 5b,

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & a_{01} & A_{02} \\ a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right).$$

**Step 6: Determining the state after repartitioning.**

This is where things become less than straight forward. The repartitionings in Step 5a do not change the contents of $y$: it is an "indexing" operation. We can thus ask ourselves the question of what the contents of $y$ are in terms of the newly exposed parts of $A$, $x$, and $y$. We can derive this state, $P_{\text{before}}$, via *textual substitution*: The repartitionings in Step 5a imply that

$$\begin{array}{c|c} A_{TL} = A_{00} & A_{TR} = \left(\begin{array}{c|c} a_{01} & A_{02} \end{array}\right) \\ \hline A_{BL} = \left(\begin{array}{c} a_{10}^T \\ \hline A_{20} \end{array}\right) & A_{BR} = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \end{array}, \quad \begin{array}{c} x_T = x_0 \\ \hline x_B = \left(\begin{array}{c} \chi_1 \\ \hline x_2 \end{array}\right) \end{array}, \quad \text{and} \quad \begin{array}{c} y_T = y_0 \\ \hline y_B = \left(\begin{array}{c} \psi_1 \\ \hline y_2 \end{array}\right) \end{array}.$$

If we substitute the expressions on the right of the equalities into the loop invariant, we find that

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right)$$

becomes

$$\left(\begin{array}{c} y_0 \\ \hline \left(\begin{array}{c} \psi_1 \\ \hline y_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} A_{00}x_0 + \widehat{y}_0 \\ \hline \left(\begin{array}{c} \widehat{\psi}_1 \\ \hline \widehat{y}_2 \end{array}\right) \end{array}\right)$$

and hence

$$\left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} A_{00}x_0 + \widehat{y}_0 \\ \hline \widehat{\psi}_1 \\ \widehat{y}_2 \end{array}\right)$$

**Step 7: Determining the state after moving the thick lines.**

The movement of the thick lines in Step 5b means that now

$$\begin{array}{c|c} A_{TL} = \left(\begin{array}{c|c} A_{00} & a_{01} \\ \hline a_{10}^T & \alpha_{11} \end{array}\right) & A_{TR} = \left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \end{array}\right) \\ \hline A_{BL} = \left(\begin{array}{c|c} A_{20} & a_{21} \end{array}\right) & A_{BR} = A_{22} \end{array}, \quad \begin{array}{c} x_T = \left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right) \\ \hline x_B = x_2 \end{array}, \quad \text{and} \quad \begin{array}{c} y_T = \left(\begin{array}{c} y_0 \\ \hline \psi_1 \end{array}\right) \\ \hline y_B = y_2 \end{array}.$$

If we substitute the expressions on the right of the equalities into the loop invariant we find that

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right)$$

becomes

$$\left(\begin{array}{c} \left(\begin{array}{c} y_0 \\ \hline \psi_1 \end{array}\right) \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c|c} A_{00} & (a_{10}^T)^T \\ \hline a_{10}^T & \alpha_{11} \end{array}\right) \left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right) + \left(\begin{array}{c} \widehat{y}_0 \\ \hline \widehat{\psi}_1 \end{array}\right) \\ \hline \widehat{y}_2 \end{array}\right),$$

where we recognize that due to symmetry $a_{01} = (a_{10}^T)^T$ and hence .

$$\left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} A_{00}x_0 + (a_{10}^T)^T\chi_1 + \widehat{y}_0 \\ a_{10}^T x_0 + \alpha_{11}\chi_1 + \widehat{\psi}_1 \\ \hline \widehat{y}_2 \end{array}\right)$$

**Step 8: Determining the update.**

Comparing the contents in Step 6 and Step 7 now tells us that the state of $y$ must change from

$$\left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} A_{00}x_0 + \widehat{y}_0 \\ \hline \widehat{\psi}_1 \\ \widehat{y}_2 \end{array}\right)$$

to

$$\left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} A_{00}x_0 + (a_{10}^T)^T\chi_1 + \widehat{y}_0 \\ a_{10}^T x_0 + \alpha_{11}\chi_1 + \widehat{\psi}_1 \\ \hline \widehat{y}_2 \end{array}\right),$$

which can be accomplished by updating

$$\begin{aligned} y_0 &:= \chi_1 (a_{10}^T)^T + y_0 \\ \psi_1 &:= a_{10}^T x_0 + \alpha_{11}\chi_1 + \psi_1. \end{aligned}$$

### 5.2.4 Other variants ☛ to edX

It is important to build fluency and contrast a number of different algorithmic variants so you can discover patterns. So, please take time for the next homework!

**Homework 5.2.4.1** Derive algorithms for Variants 2-8, corresponding to the loop invariants in Figure 5.8. (If you don't have time to do all, then we suggest you do at least Variants 2-4 and Variant 8). Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `symv_unb_var2_ws.tex`, ☛ `symv_unb_var3_ws.tex`, ☛ `symv_unb_var4_ws.tex`, ☛ `symv_unb_var5_ws.tex`, ☛ `symv_unb_var6_ws.tex`, ☛ `symv_unb_var7_ws.tex`, ☛ `symv_unb_var8_ws.tex`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.2.4.2** Match the loop invariant (on the left) to the "update" in the loop body (on the right):

Invariant 1: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(a) $y_0 := \chi_1 a_{01} + y_0$
$\psi_1 := \alpha_{11}\chi_1 + \psi_1$
$y_2 := \chi_1 a_{21} + y_2$

Invariant 2: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(b) $\psi_1 := \alpha_{11}\chi_1 + a_{21}^T x_2 + \psi_1$
$y_2 := \chi_1 a_{21} + y_2$

Invariant 3: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(c) $y_0 := \chi_1 (a_{10}^T)^T + y_0$
$\psi_1 := \alpha_{11}\chi_1 + \psi_1$
$y_2 := \chi_1 a_{21} + y_2$

Invariant 4: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(d) $\psi_1 := a_{10}^T x_0 + \alpha_{11}\chi_1 + a_{21}^T x_2 + \psi_1$

Invariant 8: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(e) $y_0 := \chi_1 (a_{10}^T)^T + y_0$
$\psi_1 := a_{10}^T x_0 + \alpha_{11}\chi_1 + \psi_1$

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.2.4.3** Derive algorithms for Variants 2-8, corresponding to the loop invariants in Figure 5.8. (If you don't have time to do all, then we suggest you do at least Variants 2-4 and Variant 8). Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `symv_unb_var2_ws.tex`, ☛ `symv_unb_var3_ws.tex`, ☛ `symv_unb_var4_ws.tex`, ☛ `symv_unb_var5_ws.tex`, ☛ `symv_unb_var6_ws.tex`, ☛ `symv_unb_var7_ws.tex`, ☛ `symv_unb_var8_ws.tex`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 5.2.5 Visualizing the different algorithms ☛ to edX

Let us reexamine the symmetric matrix-vector multiplication

$$y := Ax + y,$$

where only the lower triangular part of symmetric matrix A is stored.

The PME for this operation is

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right).$$

Consider

$$A = \begin{pmatrix} \alpha_{0,0} & \star & \star & \star \\ \alpha_{1,0} & \alpha_{1,1} & \star & \star \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \star \\ \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} \end{pmatrix}, \quad x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \chi_2 \\ \chi_3 \end{pmatrix}, \quad \text{and} \quad y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix}.$$

Then *all* the calculations that need to be performed are given by

$$\begin{array}{lllll} \alpha_{0,0}\chi_0 & +\alpha_{1,0}\chi_1 & +\alpha_{2,0}\chi_2 & +\alpha_{3,0}\chi_3 & +\widehat{\psi}_0 \\ \alpha_{1,0}\chi_0 & +\alpha_{1,1}\chi_1 & +\alpha_{2,1}\chi_2 & +\alpha_{3,1}\chi_3 & +\widehat{\psi}_1 \\ \alpha_{2,0}\chi_0 & +\alpha_{2,1}\chi_1 & +\alpha_{2,2}\chi_2 & +\alpha_{3,2}\chi_3 & +\widehat{\psi}_2 \\ \alpha_{3,0}\chi_0 & +\alpha_{3,1}\chi_1 & +\alpha_{3,2}\chi_2 & +\alpha_{3,3}\chi_3 & +\widehat{\psi}_3 \end{array}$$

Now, consider again the PME, color coded for the different parts of the matrix

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right).$$

Let us consider what computations this represents when $A_{TL}$ is $2 \times 2$ for our $4 \times 4$ example:

$$\begin{array}{lllll} \alpha_{0,0}\chi_0 & +\alpha_{1,0}\chi_1 & +\alpha_{2,0}\chi_2 & +\alpha_{3,0}\chi_3 & +\widehat{\psi}_0 \\ \alpha_{1,0}\chi_0 & +\alpha_{1,1}\chi_1 & +\alpha_{2,1}\chi_2 & +\alpha_{3,1}\chi_3 & +\widehat{\psi}_1 \\ \alpha_{2,0}\chi_0 & +\alpha_{2,1}\chi_1 & +\alpha_{2,2}\chi_2 & +\alpha_{3,2}\chi_3 & +\widehat{\psi}_2 \\ \alpha_{3,0}\chi_0 & +\alpha_{3,1}\chi_1 & +\alpha_{3,2}\chi_2 & +\alpha_{3,3}\chi_3 & +\widehat{\psi}_3 \end{array}$$

With this color coding, how the different algorithms perform computation is illustrated in Figure 5.9.

### 5.2.6 Which variant? ☛ to edX

Figure 5.10 summarizes all eight loop invariants for computing $y := Ax + y$ for the case where $A$ is symmetric and stored in the lower triangular part of the matrix. In this figure, the algorithms corresponding to Invariants 1-4 move through matrix $A$ from the top-left to bottom-right while the algorithms corresponding to Invariants 5-8 move through matrix $A$ from the bottom-right to top-left. To the right of the invariants is the update to $y$ that is in the loop body of the resulting algorithm. Interestingly, for each algorithmic variant that moves through the matrix from top-left to bottom-right, there is a corresponding variant that moves from the bottom-right to the top-left **that results in the same update to vector $y$.**

There is a clear link between the two loop invariants that yield the same update, if you look at how each pair differs and how the differences relate to the PME. In one of the enrichments, we point you to recent research that explains what you observe.

**Homework 5.2.6.1** We now return to the launch for this week and the question of how to find an algorithm for computing $y := Ax + y$, where $A$ is symmetric and stored only in the lower triangular part of $A$. Consult Figure 5.10 to answer the question of which invariant(s) yield an algorithm that accesses the matrix by columns.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

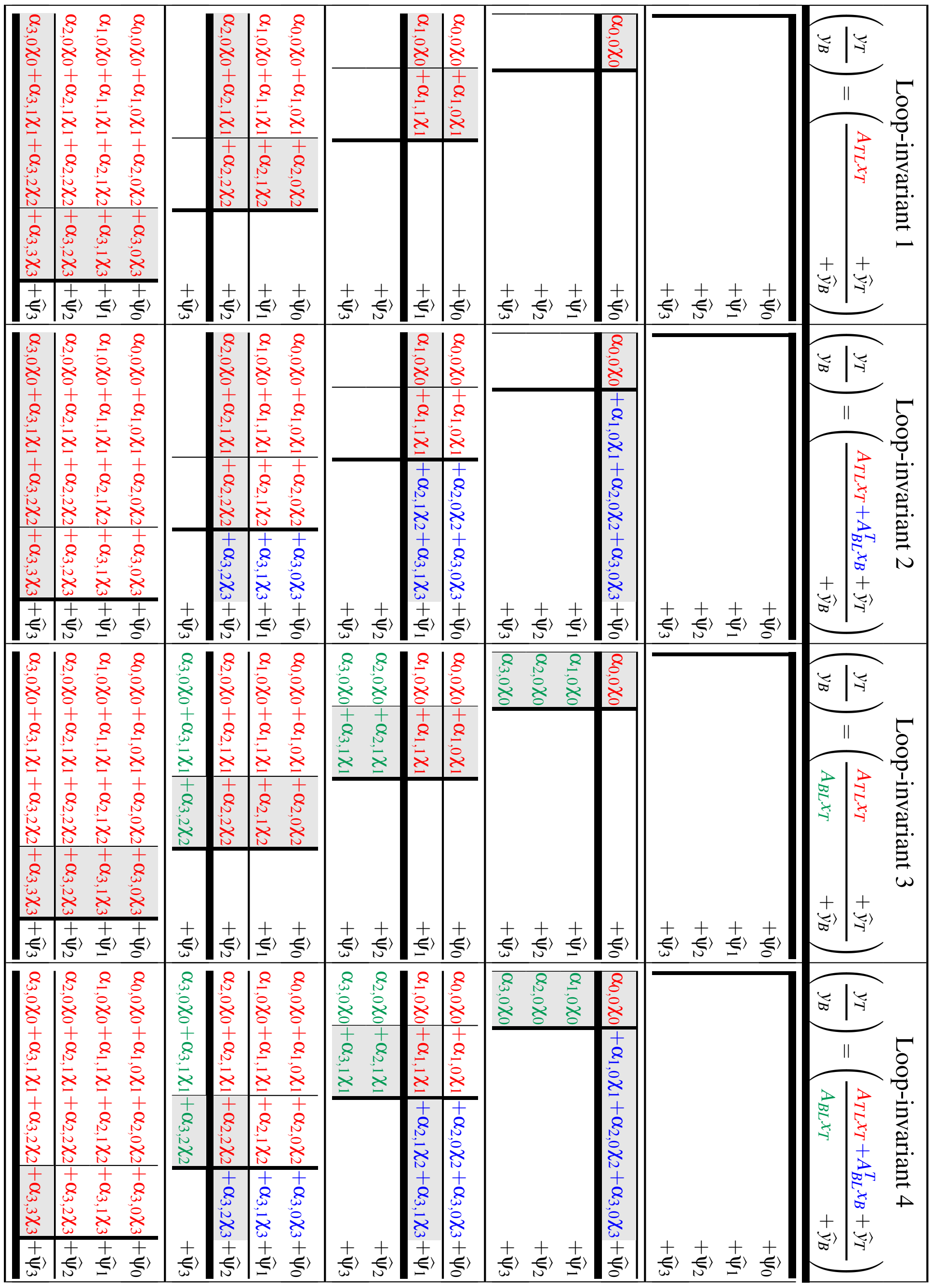


Figure 5.9: Illustration of how computation proceeds when computing $y := Ax + y$ where $A$ is symmetric and stored in the lower triangular part of $A$. The shaded region shows the computation that is performed in the indicated iteration.

| Loop invariants | | Update |
|---|---|---|
| Invariant 1: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | Invariant 5: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | $y_0 := \chi_1 (a_{10}^T)^T + y_0$ <br> $\psi_1 := a_{10}^T x_0 + \alpha_{11}\chi_1 + \psi_1$ |
| Invariant 2: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | Invariant 6: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | $\psi_1 := a_{10}^T x_0 + \alpha_{11}\chi_1 + a_{21}^T x_2 + \psi_1$ |
| Invariant 3: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | Invariant 7: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | $y_0 := \chi_1 (a_{10}^T)^T + y_0$ <br> $\psi_1 := \alpha_{11}\chi_1 + \psi_1$ <br> $y_2 := \chi_1 a_{21} + y_2$ |
| Invariant 4: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | Invariant 8: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ | $\psi_1 := \alpha_{11}\chi_1 + a_{21}^T x_2 + \psi_1$ <br> $y_2 := \chi_1 a_{21} + y_2$ |

Figure 5.10: Summary of loop invariants for computing $y := Ax + y$, where $A$ is symmetric and stored in the lower triangular part of the matrix. To the right is the update to $y$ in the derived loop corresponding to the invariants.

## 5.3 Matrix-matrix multiplication ☛ to edX

### 5.3.1 Background ☛ to edX

For details on why this operation is defined the way it is and practice with this operation, you may want to consult Weeks 4-5 of Linear Algebra: Foundations to Frontiers (LAFF). Here we give the briefest of reviews.

Given matrices $C$, $A$, and $B$ of sizes $m \times n$, $m \times k$, and $k \times n$, respectively, view these matrices as the two-dimensional arrays that represent them:

$$C = \left(\begin{array}{cccc} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array}\right), A = \left(\begin{array}{cccc} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{array}\right),$$

and

$$B = \left(\begin{array}{cccc} \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,n-1} \\ \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \beta_{k-1,0} & \beta_{k-1,1} & \cdots & \beta_{k-1,n-1} \end{array}\right).$$

Then the result of computing $C := AB$ sets

$$\gamma_{i,j} := \sum_{p=0}^{k-1} \alpha_{i,p} \times \beta_{p,j} \qquad (5.1)$$

for all $0 \le i < m$ and $0 \le j < n$. In the notation from Weeks 1-3 this is given as

$$(\forall i \mid 0 \le i < m : (\forall j \mid 0 \le j < n : \gamma_{i,j} = (\sum p \mid 0 \le p < k : \alpha_{i,p} \times \beta_{p,j}))),$$

which gives some idea of how messy postconditions and loop invariants for this operation might become using that notation.

Now, if one partitions matrices $C$, $A$, and $B$ into submatrices:

$$C = \left( \begin{array}{cccc} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \vdots & \vdots & & \vdots \\ C_{M-1,0} & C_{M-1,1} & \cdots & C_{M-1,N-1} \end{array} \right), A = \left( \begin{array}{cccc} A_{0,0} & A_{0,1} & \cdots & A_{0,K-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,K-1} \\ \vdots & \vdots & & \vdots \\ A_{M-1,0} & A_{M-1,1} & \cdots & A_{M-1,K-1} \end{array} \right),$$

and

$$B = \left( \begin{array}{cccc} B_{0,0} & B_{0,1} & \cdots & B_{0,N-1} \\ B_{1,0} & B_{1,1} & \cdots & B_{1,N-1} \\ \vdots & \vdots & & \vdots \\ B_{K-1,0} & B_{K-1,1} & \cdots & B_{K-1,N-1} \end{array} \right),$$

where $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ are $m_i \times n_j$, $m_i \times k_p$, and $k_p \times n_j$, respectively, then

$$C_{i,j} := \sum_{p=0}^{k-1} A_{i,p} B_{p,j}.$$

The computation with submatrices (blocks) mirrors the computation with the scalars in Equation 5.1:

$$C_{i,j} := \sum_{p=0}^{k-1} A_{i,p} B_{p,j} \quad \text{versus} \quad \gamma_{i,j} := \sum_{p=0}^{k-1} \alpha_{i,p} \beta_{p,j}.$$

Thus, to remember how to multiply with partitioned matrices, all you have to do is to remember how to multiply with matrix elements *except* that $A_{i,p} \times B_{p,j}$ does not necessarily commute. We will often talk about the constraint on how matrix sizes must match up by saying that the matrices are partitioned *conformally*.

There are special cases of this that will be encountered in the subsequent discussions:

$$\begin{aligned} \left( \begin{array}{c|c} A_L & A_R \end{array} \right) \left( \begin{array}{c} B_T \\ \hline B_B \end{array} \right) &= A_L B_T + A_R B_B, \\ \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) B &= \left( \begin{array}{c} A_T B \\ \hline A_B B \end{array} \right), \quad \text{and} \\ A \left( \begin{array}{c|c} B_L & B_R \end{array} \right) &= \left( \begin{array}{c|c} A B_L & A B_R \end{array} \right). \end{aligned}$$

## 5.3.2 Matrix-matrix multiplication by columns ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

To arrive at a first PME for computing $C := AB + C$, we partition matrix $B$ by columns:

$$B \rightarrow \left( B_L \,\middle|\, B_R \right).$$

After placing this in the postcondition $C = AB + \widehat{C}$, we notice that $C$ must be conformally partitioned, yielding

$$\left( C_L \,\middle|\, C_R \right) = A \left( B_L \,\middle|\, B_R \right) + \left( \widehat{C}_L \,\middle|\, \widehat{C}_R \right).$$

But what we learned in the last unit is that then

$$\left( C_L \,\middle|\, C_R \right) = \left( AB_L + \widehat{C}_L \,\middle|\, AB_R + \widehat{C}_R \right).$$

This is the sought-after PME:

- PME 1: $\left( C_L \,\middle|\, C_R \right) = \left( AB_L + \widehat{C}_L \,\middle|\, AB_R + \widehat{C}_R \right)$.

**Homework 5.3.2.1** Identify two loop invariants from PME 1.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.3.2.2** Derive Variant 1, the algorithm corresponding to Invariant 1, in the answer to the last homework. Assume the algorithm "marches" through the matrix one row or column at a time (meaning you are to derive an unblocked algorithm).
Some resources:

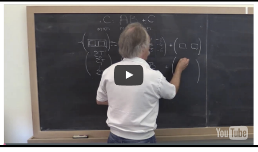
- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `gemm_unb_var1_ws.tex`
- ☛ `GemmUnbVar1LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.3.2.3** If you feel energetic, repeat the last homework for Invariant 2.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 5.3.3 Matrix-matrix multiplication by rows ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

To arrive at a second PME (PME 2) for computing $C := AB + C$, we partition matrix $A$ by rows:

$$A \rightarrow \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right).$$

After placing this in the postcondition $C = AB + \widehat{C}$, we notice that $C$ must be conformally partitioned.

**Homework 5.3.3.1** Identify a second PME (PME 2) that corresponds to the case where $A$ is partitioned by rows.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 5.3.3.2** Identify two loop invariants from this second PME (PME 2). Label these Invariant 3 and Invariant 4.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.3.3.3** Derive Variant 3, the algorithm corresponding to Invariant 3, in the answer to the last homework. Assume the algorithm "marches" through the matrix one row or column at a time (meaning you are to derive an unblocked algorithm).
Some resources:

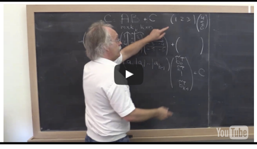
- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `gemm_unb_var3_ws.tex`
- ☛ `GemmUnbVar3LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.3.3.4** If you feel energetic, repeat the last homework for Invariant 4,

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 5.3.4 Matrix-matrix multiplication via rank-1 updates ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

To arrive at the third PME for computing $C := AB + C$, we partition matrix $A$ by columns:

$$A \rightarrow \left( \begin{array}{c|c} A_L & A_R \end{array} \right).$$

After placing this in the postcondition $C = AB + \widehat{C}$, what other matrix must be conformally patitioned?

**Homework 5.3.4.1** Identify a third PME that corresponds to the case where $A$ is partitioned by columns.
☛ SEE ANSWER
☛ DO EXERCISE ON edX

☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 5.3.4.2** Identify two loop invariants from PME 3. Label these Invariant 5 and Invariant 6.
☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.3.4.3** Derive Variant 5, the algorithm corresponding to Invariant 5, in the answer to the last homework. Assume the algorithm "marches" through the matrix one row or column at a time (meaning you are to derive an unblocked algorithm).
Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `gemm_unb_var5_ws.tex`
- ☛ `GemmUnbVar5LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.3.4.4** If you feel energetic, repeat the last homework for Invariant 6.
☛ SEE ANSWER
☛ DO EXERCISE ON edX

☛ Watch Video on edX
☛ Watch Video on YouTube

### 5.3.5 Blocked algorithms ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

In the discussions so far, we always advanced the algorithm one row and/or column at a time:

| Variant | Step 5a | Step 5b |
|---|---|---|
| 1 | $\left( B_L \,\Big\vert\, B_R \right) \rightarrow \left( B_0 \,\Big\vert\, b_1 \;\; B_2 \right), \cdots$ | $\left( B_L \,\Big\vert\, B_R \right) \leftarrow \left( B_0 \;\; b_1 \,\Big\vert\, B_2 \right), \cdots$ |
| 2 | $\left( B_L \,\Big\vert\, B_R \right) \rightarrow \left( B_0 \;\; b_1 \,\Big\vert\, B_2 \right), \cdots$ | $\left( B_L \,\Big\vert\, B_R \right) \leftarrow \left( B_0 \,\Big\vert\, b_1 \;\; B_2 \right), \cdots$ |
| 3 | $\left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \rightarrow \left( \begin{array}{c} A_0 \\ \hline a_1^T \\ A_2 \end{array} \right), \cdots$ | $\left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \leftarrow \left( \begin{array}{c} A_0 \\ a_1^T \\ \hline A_2 \end{array} \right), \cdots$ |
| 4 | $\left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \rightarrow \left( \begin{array}{c} A_0 \\ a_1^T \\ \hline A_2 \end{array} \right), \cdots$ | $\left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \leftarrow \left( \begin{array}{c} A_0 \\ \hline a_1^T \\ A_2 \end{array} \right), \cdots$ |
| 5 | $\left( A_L \,\Big\vert\, A_R \right) \rightarrow \left( A_0 \,\Big\vert\, a_1 \;\; A_2 \right), \cdots$ | $\left( A_L \,\Big\vert\, A_R \right) \leftarrow \left( A_0 \;\; a_1 \,\Big\vert\, A_2 \right), \cdots$ |
| 6 | $\left( A_L \,\Big\vert\, A_R \right) \rightarrow \left( A_0 \;\; a_1 \,\Big\vert\, A_2 \right), \cdots$ | $\left( A_L \,\Big\vert\, A_R \right) \leftarrow \left( A_0 \,\Big\vert\, a_1 \;\; A_2 \right), \cdots$ |

As will become clear in the enrichment for this week, exposing a block of columns or rows allows one to "block" for performance:

| Variant | Step 5a | Step 5b |
|---|---|---|
| 1 | $\left( B_L \,\Big\vert\, B_R \right) \rightarrow \left( B_0 \,\Big\vert\, B_1 \;\; B_2 \right), \cdots$ | $\left( B_L \,\Big\vert\, B_R \right) \leftarrow \left( B_0 \;\; B_1 \,\Big\vert\, B_2 \right), \cdots$ |
| 2 | $\left( B_L \,\Big\vert\, B_R \right) \rightarrow \left( B_0 \;\; B_1 \,\Big\vert\, B_2 \right), \cdots$ | $\left( B_L \,\Big\vert\, B_R \right) \leftarrow \left( B_0 \,\Big\vert\, B_1 \;\; B_2 \right), \cdots$ |
| 3 | $\left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \rightarrow \left( \begin{array}{c} A_0 \\ \hline A_1 \\ A_2 \end{array} \right), \cdots$ | $\left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \leftarrow \left( \begin{array}{c} A_0 \\ A_1 \\ \hline A_2 \end{array} \right), \cdots$ |
| 4 | $\left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \rightarrow \left( \begin{array}{c} A_0 \\ A_1 \\ \hline A_2 \end{array} \right), \cdots$ | $\left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \leftarrow \left( \begin{array}{c} A_0 \\ \hline A_1 \\ A_2 \end{array} \right), \cdots$ |
| 5 | $\left( A_L \,\Big\vert\, A_R \right) \rightarrow \left( A_0 \,\Big\vert\, A_1 \;\; A_2 \right), \cdots$ | $\left( A_L \,\Big\vert\, A_R \right) \leftarrow \left( A_0 \;\; A_1 \,\Big\vert\, A_2 \right), \cdots$ |
| 6 | $\left( A_L \,\Big\vert\, A_R \right) \rightarrow \left( A_0 \;\; A_1 \,\Big\vert\, A_2 \right), \cdots$ | $\left( A_L \,\Big\vert\, A_R \right) \leftarrow \left( A_0 \,\Big\vert\, A_1 \;\; A_2 \right), \cdots$ |

Such algorithms are usually referred to as "blocked algorithms," explaining why we referred to previous algorithms encountered in the course as "unblocked algorithms."

**Homework 5.3.5.1** Derive Variants 1, 3, and 5, the algorithms corresponding to Invariant 1, 3, and 5.
Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `gemm_blk_var1_ws.tex`, ☛ `gemm_blk_var3_ws.tex` ☛ `gemm_blk_var5_ws.tex`
- ☛ `GemmBlkVar1LS.mlx` , ☛ `GemmBlkVar3LS.mlx` , ☛ `GemmBlkVar5LS.mlx`

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 5.3.5.2** If you feel energetic, also derive Blocked Variants 2, 4, and 6.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 5.4 Symmetric matrix-matrix multiplication ☛ to edX

### 5.4.1 Background ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

(Throughout: notice the parallel between this material and that for symmetric matrix-vector multiplication.)

Consider the matrix-matrix operation $AB$ where $A$ and $B$ are of appropriate sizes so that this multiplication makes sense. Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right).$$

Then

$$AB = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + A_{TR}B_B \\ \hline A_{BL}B_T + A_{BR}B_B \end{array}\right)$$

provided $B_T$ and $B_B$ have the appropriate size for the subexpressions to be well-defined.

Recall from Unit 5.2.1 that if $A$ is symmetric, then $A = A^T$. For the partitioned matrix this means that

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)^T = \left(\begin{array}{c|c} A_{TL}^T & A_{BL}^T \\ \hline A_{TR}^T & A_{BR}^T \end{array}\right)$$

If $A_{TL}$ is square (and hence so is $A_{BR}$ since $A$ itself is), then we conclude that

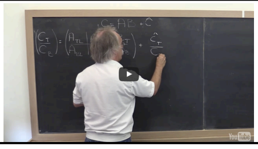
- $A_{TL}^T = A_{TL}$ and hence $A_{TL}$ is symmetric.
- $A_{BR}^T = A_{BR}$ and hence $A_{BR}$ is symmetric.

- $A_{TR} = A_{BL}^T$ and $A_{BL} = A_{TR}^T$.

Thus, for a partitioned symmetric matrix where $A_{TL}$ is square, one can compute with $\left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$ if $A_{TR}$ is not available (e.g., is not stored) or $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{TR}^T & A_{BR} \end{array}\right)$ if $A_{BL}$ is not available (e.g., is not stored). In the first case,

$$AB = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) = \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right) \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B \\ \hline A_{BL}B_T + A_{BR}B_B \end{array}\right).$$

### 5.4.2 Deriving the first PME and corresponding loop invariants ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

The operation we wish to implement is mathematically given by $C := AB + C$, where $A$ is a symmetric matrix (and hence square) and only the lower triangular part of matrix $A$ can be accessed, because (for example) the strictly upper triangular part is not stored.

#### Step 1: Precondition and postcondition

We are going to implicitly remember that $A$ is symmetric and only the lower triangular part of the matrix is stored. So, in the postcondition we simply state that $C = AB + \widehat{C}$ is TRUE.

#### Step 2: Deriving loop-invariants

Since matrix $A$ is symmetric, we want to partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

where $A_{TL}$ is square because then, because of the symmetry of $A$, we know that

- $A_{TL}$ and $A_{BR}$ are symmetric,
- $A_{TR} = A_{BL}^T$, and
- if we partition

$$B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \quad \text{and} \quad C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$$

then entering the partitioned matrices into the postcondition $C = AB + \widehat{C}$ yields

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) + \left(\begin{array}{c} \widehat{C}_T \\ \hline \widehat{C}_B \end{array}\right)$$

$$= \left( \begin{array}{c} A_{TL}B_T + A_{TR}B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array} \right)$$

$$= \left( \begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array} \right) \quad \text{(since } A_{TR} \text{ is not to be used).}$$

This last observation gives us our first PME for this operation:

$$\text{PME 1:} \quad \left( \begin{array}{c} C_T \\ \hline C_B \end{array} \right) = \left( \begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array} \right).$$

**Homework 5.4.2.1** Create a table of all loop invariants for PME 1, disgarding those for which there is no viable loop guard or initialization command. You may want to start with Figure 5.11. The gray text there will help you decide what to include in the loop invariant.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

### 5.4.3 Deriving unblocked algorithms corresponding to PME 1 ☛ to edX

In this unit, we work out the details for Invariant 4, yielding unblocked Variant 4.

#### Step 3: Determining the loop-guard.

The condition

$$P_{\text{inv}} \wedge \neg G \quad \equiv \quad \left( \begin{array}{c} C_T \\ \hline C_B \end{array} \right) = \left( \begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + \widehat{C}_B \end{array} \right) \wedge \neg G$$

must imply that

$$R : C = AB + \widehat{C}$$

holds.

We can choose $G$ as $m(A_{TL}) < m(A)$ or, equivalently because the partitioning of the matrices must be conformal, $m(C_T) < m(C)$ or $m(B_T) < m(B)$.

#### Step 4: Initialization.

When we derived the PME in Step 2, we decided to partition the matrices like

$$A \rightarrow \left( \begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right), \quad B \rightarrow \left( \begin{array}{c} B_T \\ \hline B_B \end{array} \right), \quad \text{and} \quad C \rightarrow \left( \begin{array}{c} C_T \\ \hline C_B \end{array} \right).$$

The question now is how to choose the sizes of the submatrices and vectors so that the precondition

$$C = \widehat{C}$$

implies that the loop-invariant

$$\left( \begin{array}{c} C_T \\ \hline C_B \end{array} \right) = \left( \begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + \widehat{C}_B \end{array} \right)$$

$$\text{PME: } \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array}\right).$$

| $A_{TL}B_T$ | $A_{BL}^T B_B$ | $A_{BL}B_T$ | $A_{BR}B_B$ | $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) =$ | |
|---|---|---|---|---|---|
| Yes | No | No | No | $\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array}\right)$ | 1 |
| Yes | Yes | No | No | $\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array}\right)$ | 2 |
| Yes | No | Yes | No | $\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array}\right)$ | 3 |
| Yes | Yes | Yes | No | $\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array}\right)$ | 4 |
| No | Yes | Yes | Yes | $\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array}\right)$ | 5 |
| No | Yes | No | Yes | $\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array}\right)$ | 6 |
| No | No | Yes | Yes | $\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array}\right)$ | 7 |
| No | No | No | Yes | $\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + A_{BR}B_B + \widehat{C}_B \end{array}\right)$ | 8 |

Figure 5.11: Table for Homework 5.4.2.1, in which to identify loop-invariants for $C := AB + C$ where $A$ is symmetric and only its lower triangular part is stored.

holds after the initialization (and before the loop commences).

This leads us to the initialization step

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right), C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$$

where $A_{TL}$ is $0 \times 0$ and $B_T$ and $C_T$ have no rows.

### Step 5: Progressing through the matrices.

We now note that, as part of the computation, $A_{TL}$, $B_T$ and $C_T$ start by containing no elements and must ultimately equal all of $A$, $B$ and $C$, respectively. Thus, as part of the loop, rows must be taken from $B_B$ and $C_B$ and must be added to $B_T$ and $C_T$, respectively, and the quadrant $A_{TL}$ must be expanded every time the loop executes:

$$\text{Step 5a:} \quad \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ B_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline c_1^T \\ C_2 \end{array}\right)$$

and

$$\text{Step 5b:} \quad \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & a_{01} & A_{02} \\ a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ b_1^T \\ \hline B_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ c_1^T \\ \hline C_2 \end{array}\right).$$

### Step 6: Determining the state after repartitioning.

This is where things become less straightforward. The repartitionings in Step 5a do not change the contents of $C$: it is an "indexing" operation. We can thus ask ourselves the question of what the contents of $C$ are in terms of the newly exposed parts of $A$, $B$, and $C$. We can derive this state, $P_{\text{before}}$, via *textual substitution*: The repartitionings in Step 5a imply that

$$\begin{array}{c|c} A_{TL} = A_{00} & A_{TR} = \begin{pmatrix} a_{01} & A_{02} \end{pmatrix} \\ \hline A_{BL} = \begin{pmatrix} a_{10}^T \\ A_{20} \end{pmatrix} & A_{BR} = \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{20} \end{pmatrix} \end{array}, \quad \begin{array}{c} B_T = B_0 \\ \hline B_B = \begin{pmatrix} b_1^T \\ B_2 \end{pmatrix} \end{array}, \quad \text{and} \quad \begin{array}{c} C_T = C_0 \\ \hline C_B = \begin{pmatrix} c_{10}^T \\ C_{20} \end{pmatrix} \end{array}.$$

If we substitute the expressions on the right of the equalities into the loop-invariant we find that

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + \widehat{C}_B \end{array}\right)$$

becomes

$$\left(\begin{array}{c} C_0 \\ \hline \begin{pmatrix} c_1^T \\ \hline C_2 \end{pmatrix} \end{array}\right) = \left(\begin{array}{c} (A_{00})(B_0) + \begin{pmatrix} a_{10}^T \\ A_{20} \end{pmatrix}^T \begin{pmatrix} b_1^T \\ B_2 \end{pmatrix} + \widehat{C}_0 \\ \hline \begin{pmatrix} a_{10}^T \\ A_{20} \end{pmatrix} B_0 + \begin{pmatrix} \widehat{c}_1^T \\ \widehat{C}_2 \end{pmatrix} \end{array}\right)$$

and hence

$$\left(\begin{array}{c} C_0 \\ \hline c_1^T \\ C_2 \end{array}\right) = \left(\begin{array}{c} A_{00}B_0 + (a_{10}^T)^T b_1^T + A_{20}^T B_2 + \widehat{C}_0 \\ \hline a_{10}^T B_0 + \widehat{c}_1^T \\ A_{20}^T B_0 + \widehat{C}_2 \end{array}\right)$$

**Step 7: Determining the state after moving the lines.**

The movement of the lines in Step 5b means that now

$$\left(\begin{array}{c|c} A_{TL} = \begin{pmatrix} A_{00} & a_{01} \\ a_{10}^T & \alpha_{11} \end{pmatrix} & A_{TR} = \begin{pmatrix} A_{02} \\ a_{12}^T \end{pmatrix} \\ \hline A_{BL} = \begin{pmatrix} A_{20} & a_{21} \end{pmatrix} & A_{BR} = A_{22} \end{array}\right), \quad \left(\begin{array}{c} B_T = \begin{pmatrix} B_0 \\ b_1^T \end{pmatrix} \\ \hline B_B = B_2 \end{array}\right), \quad \text{and} \quad \left(\begin{array}{c} C_T = \begin{pmatrix} C_0 \\ c_1^T \end{pmatrix} \\ \hline C_B = C_2 \end{array}\right).$$

If we substitute the expressions on the right of the equalities into the loop-invariant we find that

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T \\ \hline A_{BL}B_T + \widehat{C}_B \end{array}\right)$$

becomes

$$\left(\begin{array}{c} \begin{pmatrix} C_0 \\ c_1^T \end{pmatrix} \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} \begin{pmatrix} A_{00} & (a_{10}^T)^T \\ a_{10}^T & \alpha_{11} \end{pmatrix} \begin{pmatrix} B_0 \\ b_1^T \end{pmatrix} + \begin{pmatrix} A_{20} & a_{21} \end{pmatrix}^T B_{22} + \begin{pmatrix} \widehat{C}_0 \\ \widehat{c}_1^T \end{pmatrix} \\ \hline \begin{pmatrix} A_{20} & a_{21} \end{pmatrix} \begin{pmatrix} B_0 \\ b_1^T \end{pmatrix} + \widehat{C}_2 \end{array}\right)$$

where we recognize that due to symmetry $a_{01} = (a_{10}^T)^T$ and hence

$$\left(\begin{array}{c} C_0 \\ c_1^T \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} A_{00}B_0 + (a_{10}^T)^T b_1^T + A_{20}^T B_2 + \widehat{C}_0 \\ a_{10}^T B_0 + \alpha_{11} b_1^T + a_{21}^T B_2 + \widehat{c}_1^T \\ \hline A_{20}B_0 + a_{21} b_1^T + \qquad \widehat{C}_2 \end{array}\right)$$

**Step 8: Determining the update.**

Comparing the contents in Step 6 and Step 7 now tells us that the state of $C$ must change from

$$\left(\begin{array}{c} C_0 \\ \hline c_1^T \\ C_2 \end{array}\right) = \left(\begin{array}{c} A_{00}B_0 + (a_{10}^T)^T b_1^T + A_{20}^T B_2 + \widehat{C}_0 \\ \hline a_{10}^T B_0 + \widehat{c}_1^T \\ A_{20}^T B_0 + \widehat{C}_2 \end{array}\right)$$

to

$$\left(\begin{array}{c} C_0 \\ c_1^T \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} A_{00}B_0 + (a_{10}^T)^T b_1^T + A_{20}^T B_2 + \widehat{C}_0 \\ a_{10}^T B_0 + \alpha_{11} b_1^T + a_{21}^T B_2 + \widehat{c}_1^T \\ \hline A_{20}B_0 + a_{21} b_1^T + \qquad \widehat{C}_2 \end{array}\right),$$

which can be accomplished by updating

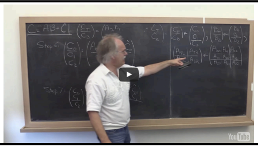
$$\begin{aligned} c_1^T &:= \alpha_{11} b_1^T + a_{12}^T B_2 + c_1^T \\ C_2 &:= a_{21} b_1^T + C_2 \end{aligned}$$

**Homework 5.4.3.1** Derive as many unblocked algorithmic variants as you find useful.
Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `symm_l_unb_var1_ws.tex`, ☛ `symm_l_unb_var2_ws.tex`, ☛ `symm_l_unb_var3_ws.tex`, ☛ `symm_l_unb_var4_ws.tex`, ☛ `symm_l_unb_var5_ws.tex`, ☛ `symm_l_unb_var6_ws.tex`, ☛ `symm_l_unb_var7_ws.tex`, ☛ `symm_l_unb_var8_ws.tex`.
- ☛ `SymmLUnbVar1LS.mlx`, ☛ `SymmLUnbVar2LS.mlx`, ☛ `SymmLUnbVar3LS.mlx`, ☛ `SymmLUnbVar4LS.mlx`, ☛ `SymmLUnbVar5LS.mlx`, ☛ `SymmLUnbVar6LS.mlx`, ☛ `SymmLUnbVar7LS.mlx`, ☛ `SymmLUnbVar8LS.mlx`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 5.4.4 Blocked Algorithms ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

We now discuss how to derive a blocked algorithm for symmetric matrix-matrix multiplication. Such an algorithm casts most computation in terms of matrix-matrix multiplication. If the matrix-matrix multiplication achieves high performance, then so does the blocked symmetric matrix-matrix multiplication (for large problem sizes).

#### Step 1: Precondition and postcondition

**Same as for the unblocked algorithms!**

We are going to implicitly remember that $A$ is symmetric and only the lower triangular part of the matrix is stored. So, in the postcondition we simply state that $C = AB + \widehat{C}$ is to be computed.

#### Step 2: Deriving loop-invariants

**Same as for the unblocked algorithms!**

We continue with Invariant 1: $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + \widehat{C}_T \\ \hline \widehat{C}_B \end{array}\right)$.

#### Step 3: Determining the loop-guard.

**Same as for the unblocked variants!**

#### Step 4: Initialization.

**Same as for the unblocked variants!**

This leads us to the initialization step

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right), C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$$

where $A_{TL}$ is $0 \times 0$ and $B_T$ and $C_T$ have no rows.

**Step 5: Progressing through the matrices.**

We now note that, as part of the computation, $A_{TL}$, $B_T$ and $C_T$ start by containing no elements and must ultimately equal all of $A$, $B$ and $C$, respectively. Thus, as part of the loop, rows must be taken from $B_B$ and $C_B$ and must be added to $B_T$ and $C_T$, respectively, and the quadrant $A_{TL}$ must be expanded every time the loop executes:

Step 5a: $$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ B_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ C_2 \end{array}\right)$$

and

Step 5b: $$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ B_1 \\ \hline B_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ C_1 \\ \hline C_2 \end{array}\right).$$

**Step 6: Determining the state after repartitioning.**

This is where things become again less straightforward. The repartitionings in Step 5a do not change the contents of $C$: it is an "indexing" operation. We can thus ask ourselves the question of what the contents of $C$ are in terms of the newly exposed parts of $A$, $B$, and $C$. We can derive this state, $P_{\text{before}}$, via *textual substitution*: The repartitionings in Step 5a imply that

$$\begin{array}{c|c} A_{TL} = A_{00} & A_{TR} = \left( A_{01} \;\; A_{02} \right) \\ \hline A_{BL} = \left(\begin{array}{c} A_{10} \\ A_{20} \end{array}\right) & A_{BR} = \left(\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{20} \end{array}\right) \end{array}, \quad \begin{array}{c} B_T = B_0 \\ \hline B_B = \left(\begin{array}{c} B_1 \\ B_2 \end{array}\right) \end{array}, \quad \text{and} \quad \begin{array}{c} C_T = C_0 \\ \hline C_B = \left(\begin{array}{c} C_{10} \\ C_{20} \end{array}\right) \end{array}.$$

If we substitute the expressions on the right of the equalities into the loop-invariant we find that

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + \widehat{C}_T \\ \hline \widehat{C}_B \end{array}\right)$$

becomes

$$\left(\begin{array}{c} C_0 \\ \hline \left(\begin{array}{c} C_1 \\ \hline C_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} (A_{00})(B_0) + \widehat{C}_0 \\ \hline \left(\begin{array}{c} \widehat{C}_1 \\ \widehat{C}_2 \end{array}\right) \end{array}\right)$$

and hence

$$\left(\begin{array}{c} C_0 \\ \hline C_1 \\ C_2 \end{array}\right) = \left(\begin{array}{c} A_{00}B_0 + \widehat{C}_0 \\ \hline \widehat{C}_1 \\ \widehat{C}_2 \end{array}\right)$$

**Step 7: Determining the state after moving the thick lines.**

The movement of the thick lines in Step 5b means that now

$$\begin{array}{c|c} A_{TL} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} & A_{TR} = \begin{pmatrix} A_{02} \\ A_{12} \end{pmatrix} \\ \hline A_{BL} = \begin{pmatrix} A_{20} & a_{21} \end{pmatrix} & A_{BR} = A_{22} \end{array}, \quad \begin{array}{c} B_T = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix} \\ \hline B_B = B_2 \end{array}, \quad \text{and} \quad \begin{array}{c} C_T = \begin{pmatrix} C_0 \\ C_1 \end{pmatrix} \\ \hline C_B = C_2 \end{array}.$$

If we substitute the expressions on the right of the equalities into the loop-invariant we find that

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + \widehat{C}_T \\ \hline \widehat{C}_B \end{array}\right)$$

becomes

$$\left(\begin{array}{c} \begin{pmatrix} C_0 \\ C_1 \end{pmatrix} \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} \begin{pmatrix} A_{00} & A_{10}^T \\ A_{10} & A_{11} \end{pmatrix}\begin{pmatrix} B_0 \\ B_1 \end{pmatrix} + \begin{pmatrix} \widehat{C}_0 \\ \widehat{C}_1 \end{pmatrix} \\ \hline \widehat{C}_2 \end{array}\right)$$

where we recognize that due to symmetry $A_{01} = A_{10}^T$ and hence

$$\left(\begin{array}{c} C_0 \\ C_1 \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} A_{00}B_0 + A_{10}^T B_1 + \widehat{C}_0 \\ A_{10}B_0 + A_{11}B_1 + \widehat{C}_1 \\ \hline \widehat{C}_2 \end{array}\right)$$

**Step 8: Determining the update.**

Comparing the contents in Step 6 and Step 7 now tells us that the state of $C$ must change from

$$\left(\begin{array}{c} C_0 \\ \hline C_1 \\ C_2 \end{array}\right) = \left(\begin{array}{c} A_{00}B_0 + \widehat{C}_0 \\ \hline \widehat{C}_1 \\ \widehat{C}_2 \end{array}\right)$$

to

$$\left(\begin{array}{c} C_0 \\ C_1 \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} A_{00}B_0 + A_{10}^T B_1 + \widehat{C}_0 \\ A_{10}B_0 + A_{11}B_1 + \widehat{C}_1 \\ \hline \widehat{C}_2 \end{array}\right),$$

which can be accomplished by updating

$$\begin{array}{rcl} C_0 & := & A_{10}^T B_1 + C_0 \\ C_1 & := & A_{10}B_0 + A_{11}B_1 + C_1 \end{array}$$

**Discussion**

☛ Watch Video on edX
☛ Watch Video on YouTube

Let us discuss the update step

$$\begin{array}{rcl} C_0 & := & \phantom{A_{10}B_0 + {}} A_{10}^T B_1 + C_0 \\ C_1 & := & A_{10}B_0 + A_{11}B_1 + C_1 \end{array}$$

and decompose the second assignment into two steps:

$$\begin{array}{rcl} C_0 & := & A_{10}^T B_1 + C_0 \\ C_1 & := & A_{10}B_0 + C_1 \\ C_1 & := & A_{11}B_1 + C_1 \end{array}$$

Let's assume that matrices $C$ and $B$ are $m \times n$ so that $A$ is $m \times m$, and that the algorithm uses a block size of $n_b$. For convenience, assume $m$ is an integer multiple of $n_b$: let $m = Mn_b$. We are going to analyze how much time is spent in each of the assignments.

Assume $A_{00}$ is $(Jn_b) \times (Jn_b)$ in size, for $0 \leq J < M-1$. Then, counting each multiply and each add as a floating point operation (flop):

- $C_0 := A_{10}^T B_1 + C_0$: This is a matrix-matrix multiply involving $(Jn_b) \times n_b$ matrix $A_{10}^T$ and $n_b \times n$ matrix $B_1$, for
$$2Jn_b^2 n \text{ flops.}$$
- $C_1 := A_{10}B_0 + C_1$: This is a matrix-matrix multiply involving $n_b \times (Jn_b)$ matrix $A_{10}$ and $Jn_b \times n$ matrix $B_0$, for
$$2Jn_b^2 n \text{ flops.}$$
- $C_1 := A_{11}B_1 + C_1$: This is a symmetric matrix-matrix multiply involving $n_b \times n_b$ matrix $A_{11}$ and $n_b \times n$ matrix $B_1$, for
$$2n_b^2 n \text{ flops.}$$

If we aggregate this over all iterations, we get

- All $C_0 := A_{10}^T B_1 + C_0$:
$$\sum_{J=0}^{M-1} 2Jn_b^2 n \text{ flops} \approx 2\frac{M^2}{2} n_b^2 n \text{ flops} = (Mn_b)^2 n \text{ flops} = m^2 n \text{ flops.}$$
- All $C_1 := A_{10}B_0 + C_1$: This is a matrix-matrix multiply involving $n_b \times (Jn_b)$ matrix $A_{10}$ and $Jn_b \times n$ matrix $B_0$, for
$$\sum_{J=0}^{M-1} 2Jn_b^2 n \text{ flops} \approx 2\frac{M^2}{2} n_b^2 n \text{ flops} = (Mn_b)^2 n \text{ flops} = m^2 n \text{ flops.}$$
- All $C_1 := A_{11}B_1 + C_1$: This is a symmetric matrix-matrix multiply involving $n_b \times n_b$ matrix $A_{11}$ and $n_b \times n$ matrix $B_1$, for
$$\sum_{J=0}^{M-1} 2n_b^2 n \text{ flops} = 2Mn_b^2 n \text{ flops} = 2n_b mn \text{ flops.}$$

The point: If $n_b$ is much smaller than $m$, then most computation is being performed in the general matrix-matrix multiplications

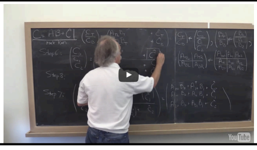
$$\begin{array}{rcl} C_0 & := & A_{10}^T B_1 + C_0 \\ C_1 & := & A_{10}B_0 + C_1 \end{array}$$

and a relatively small amount in the symmetric matrix-matrix multiplication

$$C_1 \; := \; A_{11}B_1 + C_1$$

Thus, one can use a less efficient implementation for this subproblem (for example, using an unblocked algorithm). Alternatively, since $A_{11}$ is relatively small, one can create a temporary matrix $T$ in which to copy $A_{11}$ with its upper triangular part explicitly copied as well, so that a general matrix-matrix multiplication can also be used for this subproblem.

### 5.4.5 Other blocked algorithms ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 5.4.5.1** Derive as many blocked algorithmic variants as you find useful.
Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `symm_l_blk_var1_ws.tex`, ☛ `symm_l_blk_var2_ws.tex`, ☛ `symm_l_blk_var3_ws.tex`, ☛ `symm_l_blk_var4_ws.tex`, ☛ `symm_l_blk_var5_ws.tex`, ☛ `symm_l_blk_var6_ws.tex`, ☛ `symm_l_blk_var7_ws.tex`, ☛ `symm_l_blk_var8_ws.tex`.
- ☛ `SymmLBlkVar1LS.mlx`, ☛ `SymmLBlkVar2LS.mlx`, ☛ `SymmLBlkVar3LS.mlx`, ☛ `SymmLBlkVar4LS.mlx`,
(The rest of these are not yet available.)
☛ `SymmLBlkVar5LS.mlx`, ☛ `SymmLBlkVar6LS.mlx`, ☛ `SymmLBlkVar7LS.mlx`, ☛ `SymmLBlkVar8LS.mlx`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 5.4.6 A second PME ☛ to edX

There is a second PME for this operation. Partition $C \rightarrow \left( C_L \middle| C_R \right)$ and Partition $B \rightarrow \left( B_L \middle| B_R \right)$. Then, entering this in the postcondition $C = AB + \widehat{C}$, we find that

$$\left( C_L \middle| C_R \right) = A \left( B_L \middle| B_R \right) + \left( \widehat{C}_L \middle| \widehat{C}_R \right)$$

yielding the second PME

$$\text{PME 2:} \quad \left( C_L \middle| C_R \right) = \left( AB_L + \widehat{C}_L \middle| AB_R + \widehat{C}_R \right).$$

Notice that this is identical to PME 1 for general matrix-matrix multiplication in Unit 5.3.2.

The astute reader will recognize that the update for the resulting variants cast computation in terms of a symmetric matrix-vector multiply

$$c_1 := Ab_1 + c_1$$

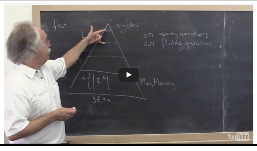
for the unblocked algorithms and the symmetric matrix-matrix multiply

$$C_1 := AB_1 + C_1$$

for the blocked algorithms.

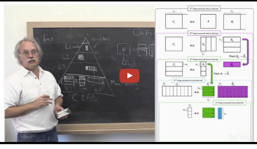
## 5.5 Enrichment ☛ to edX

### 5.5.1 The memory hierarchy ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

### 5.5.2 The GotoBLAS matrix-matrix multiplication algorithm ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

A number of recent papers on matrix-matrix multiplication are listed below.

- Kazushige Goto, Robert A. van de Geijn. "Anatomy of high-performance matrix multiplication." *ACM Transactions on Mathematical Software (TOMS)*, 2008.
  This paper on the GotoBLAS approach for implementing matrix-matrix multiplication is probably the most frequently cited recent paper on high-performance matrix-matrix multiplication. It was written to be understandable by expert and novice alike.

- Field G. Van Zee, Robert A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality." *ACM Transactions on Mathematical Software (TOMS)*, 2015.
  In this paper, the implementation of the GotoBLAS approach is refined, exposing more loops around a "micro-kernel" so that less code needs to be highly optimized.

These papers can be accessed for free from

http://www.cs.utexas.edu/ flame/web/FLAMEPublications.html

(Journal papers #11 and #39.)

We can list more reading material upon request.

### 5.5.3 The PME and loop invariants say it all! ☛ to edX

In Unit 5.2.6, Figure 5.10, you may have noticed a pattern between the PME and the two loop invariants that yielded the same update in the loop. One of those loop invariants yields an algorithm that marches through the matrix from top-left to bottom-right while the other one marches from bottom-right to top-left.

Reversing the order in which a loop index changes (e.g. from incrementing to decrementing or vise versa) is known as loop reversal. It is only valid under some circumstances. What Figure 5.10 suggests is that there may be a relation between the PME and a loop invariant that tells us conditions under which it is legal to reverse.

How the PME and loop invariants give insight into, for example, opportunities for parallelism is first discussed in

- Tze Meng Low, Robert A. van de Geijn, Field G. Van Zee.
  "Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications."
  *PPoPP '05: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

Look for Conference Publication #8 at

http://www.cs.utexas.edu/ flame/web/FLAMEPublications.html

for free access. This then led to a more complete treatment in the dissertation

- Tze Meng Low.
  *A Calculus of Loop Invariants for Dense Linear Algebra Optimization.*
  Ph.D. Dissertation. The University of Texas at Austin, Department of Computer Science. December 2013.
  This work shows how an important complication for compilers, the *phase ordering problem* can be side-stepped by looking at the PME and loop invariants.

This dissertation is also available from the same webpage.

We believe you now have the background to understand these works.

## 5.6 Wrap Up ☛ to edX

### 5.6.1 Additional exercises ☛ to edX

#### Level-3 BLAS (matrix-matrix) operations

For extra practice, the level-3 BLAS (matrix-matrix) operations are a good source. These operations involve two or more matrices and are special cases of matrix-matrix multiplication.

GEMM.

Earlier this week, you already derived algorithms for the GEMM (**ge**neral **m**atrix **m**atrix multiplication operation:

$$C := AB + C,$$

where $A$, $B$, and $C$ are all matrices with appropriate sizes. This is a special case of the operation that is part of the BLAS, which includes all of the following operations:

$$\begin{aligned} C &:= \alpha AB + \beta C \\ C &:= \alpha A^T B + \beta C \\ C &:= \alpha AB^T + \beta C \\ C &:= \alpha A^T B^T + \beta C \end{aligned}$$

(Actually, it includes even more if the matrices are complex valued). The key is that matrices $A$ and $B$ are *not* to be explicitly transposed because of the memory operations and/or extra space that would require. We suggest you ignore $\alpha$ and $\beta$. This then yields the unblocked algorithms/functions

- GEMM_NN_UNB_VARX(A, B, C) (**n**o transpose $A$, **n**o transpose B),
- GEMM_TN_UNB_VARX(A, B, C) (**t**ranspose $A$, **n**o transpose B),
- GEMM_NT_UNB_VARX(A, B, C) (**n**o transpose $A$, **t**ranspose B), and
- GEMM_TT_UNB_VARX(A, B, C) (**t**ranspose $A$, **t**ranspose B).

as well as the blocked algorithms/functions

- GEMM_NN_BLK_VARX(A, B, C) (**n**o transpose $A$, **n**o transpose B),
- GEMM_TN_BLK_VARX(A, B, C) (**t**ranspose $A$, **n**o transpose B),
- GEMM_NT_BLK_VARX(A, B, C) (**n**o transpose $A$, **t**ranspose B), and
- GEMM_TT_BLK_VARX(A, B, C) (**t**ranspose $A$, **t**ranspose B).

SYMM.

Earlier this week we discussed $C := AB + C$ where $A$ is symmetric and therefore only stored in the lower triangular part of $A$. Obviously, the matrix could instead be stored in the upper triangular part of $A$. In addition, the symmetric matrix could be on the right of matrix $B$, as in $C := BA + C$. This then yields the unblocked algorithms/functions

- SYMM_LL_UNB_VARX(A, B, C) (**l**eft, **l**ower triangle stored),
- SYMM_LU_UNB_VARX(A, B, C) (**l**eft, **u**pper triangle stored),
- SYMM_RL_UNB_VARX(A, B, C) (**r**ight, **l**ower triangle stored), and
- SYMM_RU_UNB_VARX(A, B, C) (**r**ight, **u**pper triangle stored).

and blocked algorithms/functions

- SYMM_LL_BLK_VARX(A, B, C) (**l**eft, **l**ower triangle stored),
- SYMM_LU_BLK_VARX(A, B, C) (**l**eft, **u**pper triangle stored),
- SYMM_RL_BLK_VARX(A, B, C) (**r**ight, **l**ower triangle stored), and
- SYMM_RU_BLK_VARX(A, B, C) (**r**ight, **u**pper triangle stored).

SYRK.

If matrix $C$ is symmetric, then so is the result of what is known as a symmetric rank-k update (SYRK): $C := AA^T + C$. In this case, only the lower or upper triangular part of $C$ needs to be stored and updated. Alternatively, the rank-k update can compute with the transpose of $A$ yielding $C := A^T A + C$. The resulting unblocked algorithms/functions are then

- SYRK_LN_UNB_VARX(A, B, C) (**l**ower triangle stored, **n**o transpose),
- SYRK_LT_UNB_VARX(A, B, C) (**l**ower triangle stored, **t**ranspose),
- SYRK_UN_UNB_VARX(A, B, C) (**u**pper triangle stored, **n**o transpose),
- SYRK_UT_UNB_VARX(A, B, C) (**u**pper triangle stored, **t**ranspose),

while the blocked algorithms/functions are

- SYRK_LN_BLK_VARX(A, C) (**l**ower triangle stored, **n**o transpose),
- SYRK_LT_BLK_VARX(A, C) (**l**ower triangle stored, **t**ranspose),
- SYRK_UN_BLK_VARX(A, C) (**u**pper triangle stored, **n**o transpose),
- SYRK_UT_BLK_VARX(A, C) (**u**pper triangle stored, **t**ranspose),

SYR2K.

Similarly, if matrix $C$ is symmetric, then so is the result of what is known as a symmetric rank-2k update (SYR2K): $C := AB^T + BA^T C$. In this case, only the lower or upper triangular part of $C$ needs to be stored and updated. Alternatively, the rank-2k update can compute with the transposes of $A$ and $B$, yielding $C := A^T B + B^T A + C$. The resulting unblocked algorithms/functions are then

- SYR2K_LN_UNB_VARX(A, B, C) (**l**ower triangle stored, **n**o transpose),
- SYR2K_LT_UNB_VARX(A, B, C) (**l**ower triangle stored, **t**ranspose),
- SYR2K_UN_UNB_VARX(A, B, C) (**u**pper triangle stored, **n**o transpose),
- SYR2K_UT_UNB_VARX(A, B, C) (**u**pper triangle stored, **t**ranspose),

and the blocked ones are

- SYR2K_LN_BLK_VARX(A, B, C) (**l**ower triangle stored, **n**o transpose),
- SYR2K_LT_BLK_VARX(A, B, C) (**l**ower triangle stored, **t**ranspose),
- SYR2K_UN_BLK_VARX(A, B, C) (**u**pper triangle stored, **n**o transpose),
- SYR2K_UT_BLK_VARX(A, B, C) (**u**pper triangle stored, **t**ranspose).

You may want to consider deriving algorithms for this operation after Week 6, since there is similarity with operations discussed there.

TRMM.

Another special case of matrix-matrix multiplication is given by $B := AB$, where $A$ is (lower or upper) triangular. It turns out that the output can overwrite the input matrix $B$ if the computation is carefully ordered. Alternatively, the triangular matrix can be to the right of $B$. Finally, $A$ can be optionally transposed and/or have a unit or nonunit diagonal.

$$\begin{array}{rcl} B & := & LB \\ B & := & L^T B \\ B & := & UB \\ B & := & U^T B \end{array}$$

where $L$ is a lower triangular (possibly implicitly unit lower triangular) matrix and $U$ is an upper triangular (possibly implicitly unit upper triangular) matrix. This then yields the algorithms/functions

- TRMM_LLNN_UNB_VARX(L, B) where LLNN stands for **l**eft, **l**ower triangular, **n**o transpose, **n**on unit diagonal,
- TRMM_RLNN_UNB_VARX(L, B) where LLNN stands for **r**ight, **l**ower triangular, **n**o transpose, **n**on unit diagonal,
- and so forth.

TRSM.

The final matrix-matrix operation solves $AX = B$ where $A$ is triangular, and the solution $X$ overwrites $B$. This is known as **tr**iangular **s**olve with **m**ultiple right-hand sides. We discuss this operation in detail in Week 6.

### 5.6.2 Summary ☛ to edX

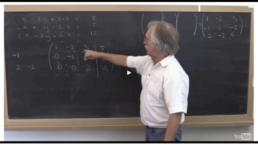
In this week, we applied the techniques to more advanced problems. Not much to summarize!

Week 6

# Advanced Matrix Operations

## 6.1 Opening Remarks ☛ to edX

### 6.1.1 Launch ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

### 6.1.2 Outline Week 6 ☛ to edX

### 6.1.3 What you will learn ☛ to edX

The final week shows the power of the FLAME notation. It demonstrates that advanced matrix operations can be tackled using this approach by examining LU factorization to discover unblocked and blocked algorithms used to solve systems of equations.

Upon completion of this week, you should be able to

- Derive and implement the unblocked and blocked algorithms for LU factorization and related operations.
- Select a loop invariant so that the algorithm developed with it will have certain desired properties, such as accessing matrices by columns.
- Recognize practical applications and extend the ideas of goal-oriented program to new situations.
- Discover the many directions of research that what you have learned enable.

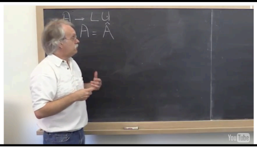
## 6.2 LU Factorization ☛ to edX

### 6.2.1 Background ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

### 6.2.2 From specification to the PME ☛ to edX

#### Step 1: The precondition and postcondition



☛ Watch Video on edX
☛ Watch Video on YouTube

Those who feel they need a refresher on Gaussian elimination and how it relates to the LU factorization may want to visit Week 6 of Linear Algebra: Foundations to Frontiers (LAFF). The materials for this course are available on ☛ edX or from ☛ `www.ulaff.net`

In the launch we argued that Gaussian elimination can be reformulated as the computation of the LU factorization of a given square matrix $A$:

$$A = LU,$$

where $L$ is a unit lower triangular matrix and $U$ is an upper triangular matrix. (This factorization does not always exist. We will revisit this issue later.) As part of the computation, the original matrix $A$ is typically overwritten with $L$ and $U$. More precisely,

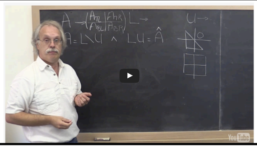
- The strictly lower triangular part of $L$ overwrites the strictly lower triangular part of $A$.
- The diagonal of $L$ is known to be all ones, and hence needs not to be stored.
- The upper triangular matrix $U$ overwrites the upper triangular part of $A$.

The precondition is given by $A = \widehat{A}$, where we implicitly assume that $A$ is square and that the LU factorization exists, and the postcondition is given by

$$A = L \backslash U \wedge LU = \widehat{A},$$

where the notation $L \backslash U$ is used to indicate that $A$ is overwritten with $L$ and $U$ as described earlier.

#### Partitioning the matrices



☛ Watch Video on edX
☛ Watch Video on YouTube

What we have learned so far is that a matrix is often partitioned into quadrants if it has special structure. For example, when a matrix, $A$, was symmetric and stored in the lower triangular part of the array, we partitioned it as

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right),$$

where $A_{TL}$ is square (and hence so is $A_{BR}$). This then meant that

$$A = \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

because of symmetry.

In fact, we were partitioning $A$ in this way also because of the triangular nature of what was stored. When matrices are triangular, one will similarly want to partition them into quadrants, this time to expose a submatrix of zeroes. Partition into lower triangular matrix $L$ and upper triangular matrix $U$ like

$$L \rightarrow \left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right) \quad \text{and} \quad U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline U_{BL} & U_{BR} \end{array}\right),$$

where

- $L_{TL}$ and $L_{BR}$ are square and hence are unit lower triangular.
- $U_{TL}$ and $U_{BR}$ are square and hence are upper triangular.

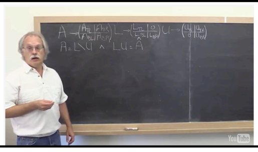
Then

$$L = \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \quad \text{and} \quad U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right),$$

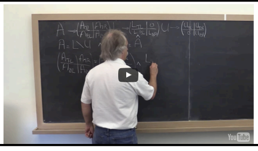
where 0 denotes a zero matrix of appropriate size.

**Homework 6.2.2.1** Derive the PME from the postcondition and how matrices $L$ and $U$ inherently need to be partitioned.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

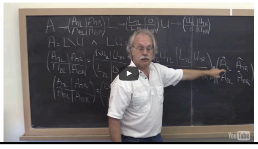


☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube

### 6.2.3 Unblocked Variant 1 ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 6.2.3.1** Derive and implement the unblocked algorithm that corresponds to

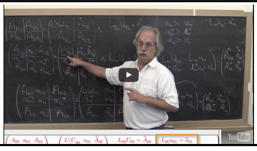
$$\text{Invariant 1}: \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge L_{TL}U_{TL} = \widehat{A}_{TL}.$$

If you get stuck, there are hints in the below videos.
Some resources:

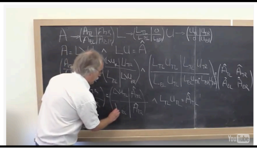
- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `LU_unb_var1_ws.tex`.
  This worksheet is already partially filled out (through Step 5).
- If you end up using the first video that follows this homework as a hint, then you may want to continue after watching that with the following worksheet: ☛ `LU_unb_var1_ws_step6.tex`.
  This worksheet is filled out through Step 6.
- ☛ `LUUnbVar1LS.mlx`.
  **Note:** for the implementation, you don't need to include L and U as parameters. They were "temporaries" in the derivation, but don't show up in the actual implementation. This same comment holds for all implementations of LU factorization.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

☛ Watch Video on edX
☛ Watch Video on YouTube


☛ Watch Video on edX
☛ Watch Video on YouTube

## 6.2.4 More loop invariants ☛ to edX


☛ Watch Video on edX
☛ Watch Video on YouTube

In the video, we suggest the second loop invariant

$$\text{Invariant 2}: \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge \begin{array}{c} L_{TL}U_{TL} = \widehat{A}_{TL} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} \end{array}$$

where

$$\begin{array}{c} L_{TL}U_{TL} = \widehat{A}_{TL} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} \end{array}$$

captures the constraint

$$L_{TL}U_{TL} = \widehat{A}_{TL} \wedge L_{BL}U_{TL} = \widehat{A}_{BL}.$$

**Homework 6.2.4.1** Derive and implement the unblocked algorithm that corresponds to

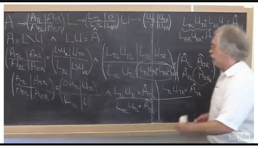
$$\text{Invariant 2}: \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge \begin{array}{c} L_{TL}U_{TL} = \widehat{A}_{TL} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL}. \end{array}$$

Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `LU_unb_var2_ws.tex`.
  This worksheet is already partially filled out (through Step 5).
- ☛ `LUUnbVar2LS.mlx`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 6.2.4.2** Identify three additional loop invariants (Invariants 3-5) for computing the LU factorization.
☛ SEE ANSWER
☛ DO EXERCISE ON edX



☛ Watch Video on edX
☛ Watch Video on YouTube

**Homework 6.2.4.3** Derive and implement the unblocked algorithms that correspond to Invariants 3-5 from the last homework. If you have limited time, then derive at least the algorithm corresponding to Invariant 5.
Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `LU_unb_var3_ws.tex`, ☛ `LU_unb_var4_ws.tex`, ☛ `LU_unb_var5_ws.tex`.
  These worksheets are already partially filled out (through Step 5).
- ☛ `LUUnbVar3LS.mlx`, ☛ `LUUnbVar4LS.mlx`, ☛ `LUUnbVar5LS.mlx`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 6.2.5 Blocked algorithms ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

The loop invariants for LU factorization are given in Figure 6.12.

**Homework 6.2.5.1** Derive and implement the blocked algorithm that corresponds to

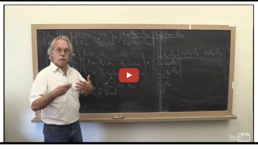
Invariant 5 :

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & \widehat{U}_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right) \wedge \begin{array}{c|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL}. & \end{array}$$

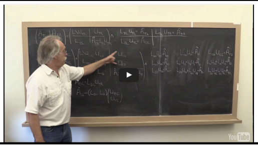
If you need hints along the way, you may want to watch the videos that follow this homework.
Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `LU_blk_var5_ws.tex`.
- ☛ `LUBlkVar5LS.mlx`.

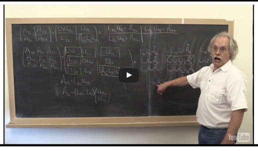
☛ SEE ANSWER
☛ DO EXERCISE ON edX



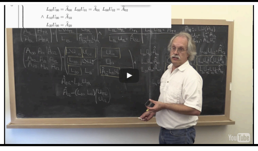
☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube



☛ Watch Video on edX
☛ Watch Video on YouTube

| | |
|---|---|
| Invariant 1 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right) \quad \wedge \; L_{TL}U_{TL} = \widehat{A}_{TL}$ |
| Invariant 2 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right) \quad \wedge \; \begin{array}{c} L_{TL}U_{TL} = \widehat{A}_{TL} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL}. \end{array}$ |
| Invariant 3 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & U_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right) \quad \wedge \; \begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \end{array}$ |
| Invariant 4 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right) \quad \wedge \; \begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}$ |
| Invariant 5 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right) \wedge \; \begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}$ |

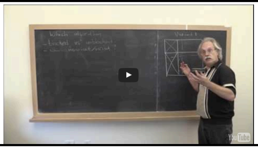
Figure 6.1: The five loop invariants for computing the LU factorization.

**Homework 6.2.5.2** Derive and implement the blocked algorithms that correspond to Invariants 1-4. If you have limited time, then you may want to focus on Invariant 2. A table of all loop invariants was given in the solution for Homework 6.2.4.2.
Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `LU_blk_var1_ws.tex`, ☛ `LU_blk_var2_ws.tex`, ☛ `LU_blk_var3_ws.tex`, ☛ `LU_blk_var4_ws.tex`.
- ☛ `LUBlkVar1LS.mlx`, ☛ `LUBlkVar2LS.mlx`, ☛ `LUBlkVar3LS.mlx`, ☛ `LUBlkVar4LS.mlx`.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 6.2.6 Which variant to pick ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

| | | |
|---|---|---|
| Invariant 1 : (Bordered algorithm) | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right)$ | $\wedge\ L_{TL}U_{TL} = \widehat{A}_{TL}$ |
| Invariant 2 : (Left-looking algorithm) | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right)$ | $\wedge\ \begin{array}{c} L_{TL}U_{TL} = \widehat{A}_{TL} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL}. \end{array}$ |
| Invariant 3 : (Up-looking algorithm) | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & U_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right)$ | $\wedge\ \begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \end{array}$ |
| Invariant 4 : (Crout Variant) | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right)$ | $\wedge\ \begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}$ |
| Invariant 5 : (Right-looking algorithm) | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right)$ | $\wedge\ \begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}$ |

Figure 6.2: The five loop invariants for computing the LU factorization. Commonly used names are given in parentheses. The Right-looking algorithm is also often called Classical Gaussian Elimination.

### 6.2.7 LU factorization with pivoting ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

☛ Watch Video on edX
☛ Watch Video on YouTube

For details on LU factorization with pivoting, we refer the interested reader to

Robert A. van de Geijn and Enrique S. Quintana-Ortí
The Science of Programming Matrix Computations
PDF for free at www.lulu.com, 2008

## 6.3 Related Operations

### 6.3.1 Triangular solve ☛ to edX

☛ Watch Video on edX
☛ Watch Video on YouTube

Computing the LU factorization of a matrix is a first step towards solving the linear system

$$Ax = y,$$

where $A$ is a square matrix and vector $y$ is the right-hand side of the system. These are the given inputs. The vector $x$ is to be computed.

The idea is as follows: if $A = LU$, then

$$Ax = y$$

is equivalent to

$$(LU)x = y,$$

which is in turn equivalent to

$$L(Ux) = y.$$

Now, we can introduce a temporary vector $z = Ux$ so that $Lz = y$. Then $Ax = y$ can be solved via the steps

- Compute the LU factorization of $A$ so that $A = LU$.
- Solve $Lz = y$. Typically the result overwrites the input vector $y$.
- Solve $Ux = y$, where $y$ now contains the solution to $Lz = y$. Typically $x$ overwrites the input vector $y$.

These operations are known as triangular solves (TRSV in BLAS terminology).

The operation that solves $Lx = y$ was already encountered when we discussed LU factorization: Unblocked Variants 1 and 2 included the update

$$a_{01} := L_{00}^{-1} a_{01}.$$

One can (and should) implement this as the unit lower triangular solve $Lx = y$, where $L$ is the submatrix $L_{00}$, $y$ is $a_{01}$, and $x$ overwrites $a_{01}$. Why should you implement it this way? Because inverting an $n \times n$ matrix requires $n^3$ floating point operations while solving with $L$ requires only $n^2$ operations.

Another operation that we encountered, in Unblocked Variants 1 and 3, was

$$a_{10}^T := a_{10}^T U_{00}^{-1}.$$

This can be reformulated as

$$(a_{01}^T)^T := U_{00}^{-T} (a_{01}^T)^T$$

where $U_{00}^{-T}$ equals the inverse of $U^T$. (In a linear algebra course you may have learned that $(A^{-1})^T = (A^T)^{-1}$ can be denoted by $A^{-T}$ since the order of applying the transposition and the inversion doesn't matter.) We can then recognize this as solving the triangular system

$$U^T x = y$$

where $U$ equals $U_{00}$, $y$ equals $(a_{10}^T)^T$ and the solution $x$ overwrites $a_{10}^T$ (storing it as a row in $A$.)

This now suggests an entire class of triangular solve operations:

$$\begin{aligned} L\ x &= y & (6.1)\\ L^T x &= y & (6.2)\\ U\ x &= y & (6.3)\\ U^T x &= y, & (6.4) \end{aligned}$$

where $L$ can be a lower or unit lower triangular matrix, $U$ can be an upper or unit upper triangular matrix, $x$ overwrites $y$, and $y$ can be stored as a row or a column.

**Homework 6.3.1.1** Derive the PME for solving $Lx = y$, overwriting $y$ with the solution, where $L$ is **unit lower triangular** and $y$ is stored as a column vector. Next, derive and implement unblocked algorithms. You will want to find an algorithm that accesses $L$ by columns. Can you already tell from the loop invariant which algorithm will have that property, so that you only have to derive one?
We use `trsv_lnu` and `TrsvLNU` to indicate that the **tr**iangular **s**ol**v**e involves the **l**ower triangular part of the matrix, that the matrix stored there is **n**ot to be transposed, and that matrix is a **u**nit triangular matrix.
Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `trsv_lnu_unb_var1_ws.tex`, ☛ `trsv_lnu_unb_var2_ws.tex`
- ☛ `TrsvLNUUnbVar1LS.mlx`, ☛ `TrsvLNUUnbVar2LS.mlx`
  **Note:** for the implementation, you don't need to include $x$ as a parameter. It was a "temporary" in the derivation, but doesn't show up in the actual implementation. This same comment holds for all implementations of triangular solve.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

**Homework 6.3.1.2** Derive the PME for solving $Ux = y$, overwriting $y$ with the solution, where $U$ is upper triangular and $y$ is stored as a column vector. In what direction should you march through the matrix and vectors? (The PME should tell you.) Derive and implement unblocked algorithms. You will want to find an algorithm that accesses $U$ by columns. Can you already tell from the loop invariant which algorithm will have that property, so that you only have to derive one?
We use `trsv_unn` and `TrsvUNN` to indicate that the **tr**iangular **s**ol**v**e involves the **u**pper triangular part of the matrix, that the matrix stored there is **n**ot to be transposed, and that matrix is **n**ot a unit triangular matrix.
Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `trsv_unn_unb_var1_ws.tex`, ☛ `trsv_unn_unb_var2_ws.tex`
- ☛ `TrsvUNNUnbVar1LS.mlx`, ☛ `TrsvUNNUnbVar2LS.mlx`
  **Note:** for the implementation, you don't need to include $x$ as a parameter. It was a "temporary" in the derivation, but doesn't show up in the actual implementation. This same comment holds for all implementations of triangular solve.

☛ SEE ANSWER
☛ DO EXERCISE ON edX

### 6.3.2 Triangular solve with multiple right-hand sides ☛ to edX

(Caption will be displayed when you start playing the video.)

☛ Watch Video on edX
☛ Watch Video on YouTube

Consider the set of linear equations

$$\begin{aligned} Ax_0 &= b_0 \\ Ax_1 &= b_1 \\ \vdots &\quad \vdots \\ Ax_{n-1} &= b_{n-1}, \end{aligned}$$

where $A$ is given as are the right-hand sides, $b_0, b_1, \cdots, b_{n-1}$. This can be reformulated as

$$\left( \begin{array}{c|c|c|c} Ax_0 & Ax_1 & \cdots & Ax_{n-1} \end{array} \right) = \left( \begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1} \end{array} \right)$$

or, equivalently,

$$AX = B,$$

where $X = \left( \begin{array}{c|c|c|c} x_0 & x_1 & \cdots & x_{n-1} \end{array} \right)$ and $B = \left( \begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1} \end{array} \right)$. Thus, if $A$ and $B$ are given, then solving $AX = B$ for matrix $X$ can be viewed as solving a *linear system with multiple right-hand sides*. If matrix $A$ is triangular, then this becomes a triangular solve with multiple right-hand sides (TRSM in BLAS terminology).

Consider the case where $A$ is replaced by the lower triangular matrix $L$. Then the postcondition becomes

$$B = X \wedge LX = \widehat{B},$$

which captures that $B$ can be overwritten with the solution. Partitioning $X$ and $B$ by columns yields the PME

$$\left( \begin{array}{c|c} B_L & B_R \end{array} \right) = \left( \begin{array}{c|c} X_L & X_R \end{array} \right) \wedge \begin{array}{c|c} LX_L = \widehat{B}_L & LX_R = \widehat{B}_R \end{array}.$$

It is not hard to see that this PME yields algorithms that compute $X$ one or more columns at a time (depending on whether an unblocked or blocked algorithm is derived).

**Homework 6.3.2.1** Derive an alternative PME that corresponds to partitioning $L$ into quadrants. You are on your own: derive the invariants, the algorithms, the implementations. If you still have energy left after that, do the same for solving $UX = B$ or $L^T X = B$ ore $U^T X = B$ (without explicitly transposing $L$ or $U$). You are now an expert!

☛ SEE ANSWER
☛ DO EXERCISE ON edX

## 6.4 Enrichment ☛ to edX

### 6.4.1 At the frontier and beyond...

For those who yearn for more, further readings can be found at

☛ http://shpc.ices.utexas.edu/publications.html.

Most can be accessed for free by clicking on the title of the paper on that webpage.

When an undergraduate or graduate student shows interest in pursuing research with us, we ask them to read

> Robert A. van de Geijn and Enrique S. Quintana-Orti.
> The Science of Programming Matrix Computations.
> www.lulu.com, 2008.

and some of the materials mentioned in Unit 5.5.2. After this, if they are still interested, they may be ready to pursue research. Now, this course provides that background.

As one learns about formal derivation of algorithms, one starts to wonder...

- Can the techniques yield production quality libraries?
- Since scientific computing typically involves real valued data, stored in finite precision in the memory of a computer, what does correct mean anyway?
- The examples in the second part of this course seem to focus on dense matrix operations (dense matrices don't have enough known zeros to warrant taking advantage of them). Does the methodology apply to all such operations? Does it apply to operations with sparse matrices (which do have enough known zeros to warrant taking advantage of them)?
- Does it apply to operations that don't involve matrices or multidimensional arrays?
- Why haven't software engineers been programming using APIs like FLAME@lab all along?
- As you were doing the various homework assignments, you may have noticed how systematic the approach is. Can it be automated? Spark on steriods?

In this week's enrichments, we address some of these questions. But what we really hope is that you now see opportunities to take what you have learned in new directions.

### 6.4.2 Practical libraries ☛ to edX

In this course, we illustrate how algorithms can be represented in code using the FLAME@lab API for MATLAB. To attain high performance, one would need to instead use a language like C.

An overview of how the techniques have been implmented in a practical library can be found in

> Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Orti, Gregorio Quintana-Orti.
> The libflame Library for Dense Matrix Computations.
> IEEE Computing in Science and Engineering, Vol. 11, No 6, November/December 2009.

A complete reference for the `libflame` library that has resulted is given in

> Field G. Van Zee.
> `libflame`: The Complete Reference.
> `www.lulu.com`, 2009.

We are funded by the National Science Foundation's Software Infrastructure for Sustained Innovation (SI2) program (most currently by Award ACI-1550493, which also partially funds this MOOC) to create a new linear algebra software stack. In addition, we receive funding from a number of partner companies, listed at ☛ http://shpc.ices.utexas.edu/sponsors.html.

A video of Robert implementing the Cholesky factorization, an operation that is closely related to LU factorization, in C that illustrates the performance benefits of blocked algorithms can be found at

☛ http://www.cs.utexas.edu/users/flame/Movies.html#Chol

**(If you would like to try your hand at this, be warned that at this time we need to still check if the given links still work. In particular, the link to the GotoBLAS2 is out of date.)**

### 6.4.3 Correctness in the presence of roundoff error ☛ to edX

The problem with correctness for matrix algorithms is that in practice the data is real valued (stored as floating point numbers) and that the computation uses floating point arithmetic that introduces roundoff error when real valued numbers are approximated.

A notion of correctness in the presence of roundoff error is that of numerical stability. A computer implementation of an algorithm (which incurs roundoff error) for computing a result is said to be numerically stable if it computes the exact result for a slightly changed (perturbed) input. The idea is that if the problem is such that a small change in the input yields a large change in the output, then the problem is inherently difficult to solve accurately and is then said to be ill-conditioned. It is only if the computed result cannot be linked to a slightly changed problem that the implementation has a serious concern. In this case, the implementation and/or algorithm is not correct in the presence of roundoff error.

Let us illustrate this by considering the problem of solving $Ax = y$ where $A$ and $y$ are given and $x$ is to be computed. When we solve this on a computer, we expect an inexact result to be computed because of roundoff error that accumulates as the computation proceeds. Let's call this result $\tilde{x}$. Then $A\tilde{x} = \tilde{y}$, some alternative right-hand side. Now, it is easy to show that $A(\tilde{x} - x) = \tilde{y} - y$. Rather than focusing on whether $\tilde{x}$ is "close to" $x$ (which it won't be if the problem of solving *this* linear system is ill-conditioned), we instead ask whether $\tilde{x}$ is the solution to a nearby problem: Is $\tilde{y}$ "close to" $y$? In practice the matrix and the right-hand side often involve measured data that is inexact. So, we would expect that $y$ has some error in it anyway: even $x$ then only solves a nearby problem. Thus, computing the solution to a nearby problem is the best we can hope for. If the problem is badly behaved, then a small change in the input ($\tilde{y}$) could yield a large change in the output ($\tilde{x}$) and hence one can't hope for a good solution, regardless of how "correct" our implementation is.

To prove a matrix algorithm correct thus means that one must not only find a correct algorithm in the absence of roundoff error, but one must also then prove that the algorithm is numerically stable. Proving that an algorithm is numerically stable typically employs a technique known as "backward error analysis." Fortunately, we have shown that for many matrix computations backward error analysis can be systematic much like the derivation of matrix algorithms was systematic. This involves what we call the "error worksheet." Details can be found in the journal paper

> Paolo Bientinesi, Robert A. van de Geijn.
> Goal-Oriented and Modular Stability Analysis.
> Journal on Matrix Analysis and Applications Volume 32 Issue 1, February 2011.

The material in that paper is based on the dissertation

> Paolo Bientinesi.
> ☛ Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms.
> Ph.D. Dissertation. The University of Texas at Austin, Department of Computer Sciences. Aug. 2006.

If you are interested, we recommend you instead read the technical report

> Paolo Bientinesi and Robert A. van de Geijn.
> ☛ The Science of Deriving Stability Analyses.
> FLAME Working Note #33. Aachen Institute for Computational Engineering Sciences, RWTH Aachen.
> TR AICES-2008-2. November 2008.

which is very much like the journal paper, but includes exercises for the reader.

### 6.4.4 Beyond dense linear algebra ☛ to edX

Computations with matrices and vectors are of great importance to scientific computing and domains like machine learning. While operations with dense matrices are important, so-called sparse matrix computations are more common.

In preliminary work, we have extended the FLAME methodology to so-called Krylov subspace methods (including the Conjugate Gradient method):

> Victor Eijkhout, Paolo Bientinesi, Robert van de Geijn.
> Toward Mechanical Derivation of Krylov Solver Libraries.
> Procedia Computer Science, 1(1) 1805-1813, 2010 (Proceedings of ICCS2010.)

For those who know about such methods, the trick is to view the vectors that are produced by the iteration as the columns of a matrix, as was advocated by Alston Householder in the 1960s. That matrix is dense, allowing a variation of the FLAME methodology to be applied.

In

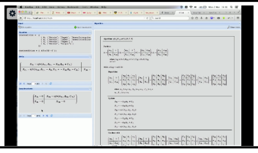
> Tze Meng Low.
> ☛ *A Calculus of Loop Invariants for Dense Linear Algebra Optimization.*
> Ph.D. Dissertation. The University of Texas at Austin, Department of Computer Science. December 2013.

it is suggested that the FLAME methodology can be extended to the class of Primitive Recursive Functions (PRFs). Those familiar with recursion will have noticed that the PME is invariably a recursive definition of the operation and that the FLAME methodology transforms this definition into a family of loop-based algorithms. This makes the operations to which we have applied the FLAME methodology members of the class of PRFs.

### 6.4.5 When the worksheet does not yield algorithms for matrix operations ☛ to edX

There are matrix operations for which the FLAME methodology will not yield loop-based algorithms. (Or, at least, not in a straightforward fashion.) It is well-known that, in general, there is no formula for solving the algebraic eigenvalue problem for larger matrices. (Given a matrix of size at least $5 \times 5$ there is no formula for finding the eigenvalues, except for special matrices.) Thus, we cannot expect a loop to be derived, via the FLAME methodology or otherwise, that computes these in a finite number of iterations.

### 6.4.6 If it is so systematic, can't we get a computer to do it? ☛ to edX



☛ Watch Video on edX
☛ Watch Video on YouTube

Computer scientists make knowledge systematic with automation as the ultimate goal. For many matrix computations, programming with the FLAME methodology is systematic. Can it be automated?

The above video by Prof. Paolo Bientinesi demonstrates the Cl1ck tool mentioned later in this unit. (Unfortunately, the interface that is used in that video is not publicly available. Fortunately, Cl1ck has been rewritten in Python.)

The FLAME methodology was first published in

> John A. Gunnels, Fred G. Gustavson, Greg M. Henry, Robert A. van de Geijn.
> FLAME: Formal Linear Algebra Methods Environment.
> ACM Transactions on Mathematical Software (TOMS), 2001

and the dissertation

> John A. Gunnels.
> ☛ A Systematic Approach to the Design and Analysis of Linear Algebra Algorithms.
> Ph.D. Dissertation. FLAME Working Note #6, The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2001-44. Nov. 2001.

"The Worksheet" to which you were introduced in Weeks 4-6 was first published in

> Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Orti, Robert A. van de Geijn.
> The science of deriving dense linear algebra algorithms.
> ACM Transactions on Mathematical Software (TOMS), 2005.

What the worksheet made obvious was that, on the one hand, the methodology had pedagogical potential and, on the other hand, it could be automated. A prototype was first published in

Paolo Bientinesi, Sergey Kolos, and Robert A. van de Geijn.
Automatic Derivation of Linear Algebra Algorithms with Application to Control Theory.
PARA 2004, LNCS 3732, pp. 385–394, 2005.

and became a major part of the dissertation

Paolo Bientinesi.
☛ Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms.
Ph.D. Dissertation. The University of Texas at Austin, Department of Computer Sciences. Aug. 2006.

More recently, how to automatically generate PMEs is described in

Diego Fabregat-Traver and Paolo Bientinesi.
☛ Knowledge-Based Automatic Generation of Partitioned Matrix Expressions.
Computer Algebra in Scientific Computing, Lecture Notes in Computer Science, Volume 6885, pp. 144-157, Springer, 2011.
Technical report: ☛ http://hpac.rwth-aachen.de/aices/preprint/documents/AICES-2011-01-03.pdf.

and how to then derive invariants in

Diego Fabregat-Traver and Paolo Bientinesi.
☛ Automatic Generation of Loop-Invariants for Matrix Operations.
Computational Science and its Applications, International Conference, pp. 82-92, IEEE Computer Society, 2011.
Technical report: ☛ http://hpac.rwth-aachen.de/aices/preprint/documents/AICES-2011-02-01.pdf

The complete automation of the methodology is the subject of Chapter 4 of

Diego Fabregat-Traver.
Knowledge-Based Automatic Generation of Linear Algebra Algorithms and Code.
Dissertation. RWTH Aachen, April 2014.
Technical report: ☛ http://arxiv.org/abs/1404.3406

The Cl1ck tool that resulted from this work can be downloaded from

☛ https://github.com/dfabregat/Cl1ck.

There you will also find instructions and examples.

## 6.5 Wrap Up ☛ to edX

### 6.5.1 Additional exercises ☛ to edX

In this unit, we list a number of operations that are similar to those found in this week and for which you may want to try and derive algorithms.

#### Factorization operations

This week started with a discussion of the LU factorization. Here are a number of related operations.

**Variations on LU factorization.** While the LU factorization captures the computations performed by Gaussian elimination, some obvious variations of the theme come to mind:

- $A \rightarrow LU$ where $L$ is lower triangular and $U$ is unit upper triangular,
- One could compute $A \rightarrow UL$ where $L$ is lower triangular and $U$ is unit upper triangular, and
- One could compute $A \rightarrow UL$ where $L$ is unit lower triangular and $U$ is upper triangular.

(Here the $\rightarrow$ denotes that the factorization is computed.)

**Unfactoring the LU factorization.** Also of interest might be the operations $A := LU$ or $A := UL$ that reverse the factorizations discussed above.

**LU factorization with pivoting.** It is well-known that in practice one should swap rows as one computes the LU factorization. This is discussed, for example, in Week 7 of Linear Algebra: Foundations to Frontiers (LAFF) and is also a standard topic in courses on numerical algorithms or numerical linear algebra. The derivation is a bit tricky... Robert has done it, but it is not cleanly written up anywhere. Hopefully he will do so soon!

**Reduction to row-echelon form.** For the insider, reduction to row-echelon form can be thought of as LU factorization with complete pivoting. Another project Robert should take on sometime...

**Cholesky factorization.** In a number of the enrichments, the Cholesky factoriziation is mentioned. Given symmetric positive definite (SPD) matrix $A$, it computes the factorization $A \rightarrow LL^T$ where $L$ is a lower triangular matrix. (If the diagonal entries are forced to be positive, the factorization is unique). The Cholesky factor $L$, overwrites the lower triangular part of $A$ (if it was that part of the symmetric matrix that was originally stored). It should be relatively straightforwardard to derive the three loop invariants for this operation and to then derive the unblocked and blocked variants.

There are some obvious variations on this theme: $A \rightarrow UU^T$, $A \rightarrow L^T L$ and $A \rightarrow U^T U$.

**The $LDL^T$ factorization.** Given a symmetric (but not necessarily positive definite) matrix, under mild conditions one can compute the $LDL^T$ factorization, $A \rightarrow LDL^T$, where $L$ is unit lower triangular and $D$ is diagonal. Obvious variations on the these include $A \rightarrow U^T DU$, where $U$ is unit upper triangular.

**The QR factorization.** Given matrix $A$, computing an orthogonal basis for its column space can be formulated as the factorization $A \rightarrow QR$ where the columns of $Q$ form an orthonormal basis for the column space of $A$ and $R$ is upper triangular. The postcondition for this operation becomes $A = QR \wedge Q^T Q = I$. From this, it is relatively easy to derive what are known as the classic Gram-Schmidt method and the Modified Gram-Schmidt method.

Trickier to derive is Householder QR factorization. Robert has notes, somewhere...

## Solving or multiplying with triangular matrices

**TRSV (triangular solve).**

In Section 6.3, we discussed a number of operations related to the solution of a triangular system of linear equations. Much like there were a large number such operations for the closely related triangular matrix-vector multiplication operation (TRMV), there are a large number of triangular solve operations. As for TRMV, the input can be overwritten with the output:

$$\begin{array}{rcll} y & := & L^{-1}y & \text{or, equivalently, solve } Lx = y \text{ overwriting } y \text{ with } x \\ y & := & L^{-T}y & \text{or, equivalently, solve } L^T x = y \text{ overwriting } y \text{ with } x \\ y & := & U^{-1}y & \text{or, equivalently, solve } Ux = y \text{ overwriting } y \text{ with } x \\ y & := & U^{-T}y & \text{or, equivalently, solve } U^T x = y \text{ overwriting } y \text{ with } x, \end{array}$$

where $L$ is a lower triangular (possibly implicitly unit lower triangular) matrix and $U$ is an upper triangular (possibly implicitly unit upper triangular) matrix. This then yields the algorithms/functions

- TRSV_LNN_UNB_VARX(L, Y) where LNN stands for **l**ower triangular, **n**o transpose, **n**on unit diagonal,
- TRSV_LNU_UNB_VARX(L, Y) where LNU stands for **l**ower triangular, **n**o transpose, **u**nit diagonal,
- TRSV_LTN_UNB_VARX(L, Y) where LTN stands for **l**ower triangular, **t**ranspose, **n**on unit diagonal,
- TRSV_LTU_UNB_VARX(L, Y) where LTU stands for **l**ower triangular, **t**ranspose, **u**nit diagonal,

- TRSV_UNN_UNB_VARX(L, Y) where UNN stands for **u**pper triangular, **n**o transpose, **n**on unit diagonal,
- TRSV_UNU_UNB_VARX(L, Y) where UNU stands for **u**pper triangular, **n**o transpose, **u**nit diagonal,
- TRSV_UTN_UNB_VARX(L, Y) where UTN stands for **u**pper triangular, **t**ranspose, **n**on unit diagonal,
- TRSV_UTU_UNB_VARX(L, Y) where UTU stands for **u**pper triangular, **t**ranspose, **u**nit diagonal.

TRSM **(triangular solves with multiple right-hand sides).**
Similarly, there are many cases of triangular solve with multiple right-hand sides (TRSM):

| | |
|---|---|
| $B := L^{-1}B$ | or, equivalently, solve $LX = B$ overwriting $B$ with $X$ |
| $B := L^{-T}B$ | or, equivalently, solve $L^TX = B$ overwriting $B$ with $X$ |
| $B := BL^{-1}$ | or, equivalently, solve $XL = B$ overwriting $B$ with $X$ |
| $B := BL^{-T}$ | or, equivalently, solve $XL^T = B$ overwriting $B$ with $X$ |
| $B := U^{-1}B$ | or, equivalently, solve $UX = B$ overwriting $B$ with $X$ |
| $B := U^{-T}B$ | or, equivalently, solve $U^TX = B$ overwriting $B$ with $X$ |
| $B := BU^{-1}$ | or, equivalently, solve $XU = B$ overwriting $B$ with $X$ |
| $B := BU^{-T}$ | or, equivalently, solve $XU^T = B$ overwriting $B$ with $X$ |

where $L$ is a lower triangular (possibly implicitly unit lower triangular) matrix and $U$ is an upper triangular (possibly implicitly unit upper triangular) matrix. This then yields the algorithms/functions

- TRSM_LLNN_UNB_VARX(L, B) where LLNN stands for **l**eft, **l**ower triangular, **n**o transpose, **n**on unit diagonal,
- TRSM_RLNN_UNB_VARX(L, B) where LLNN stands for **r**ight, **l**ower triangular, **n**o transpose, **n**on unit diagonal,
- and so forth.

**Two-sided triangular solve.** Given square matrix $A$, the algebraic eigenvalue problem computes eigenpairs $(\lambda, x)$ so that $Ax = \lambda x$. If $A$ is symmetric, this becomes a symmetric eigenproblem. For reasons that go beyond this course, using the FLAME methodology for computing this solution is inherently not possible. But there are steps along the way to which it does apply.

Here we discuss an operation that reduces the generalized positive definite eigenvalue problem $Ax = \lambda Bx$ to a symmetric eigenvalue problem. Typically, matrix $B$ is symmetric positive definite (SPD) and $A$ is symmetric. In this case, we can use the Cholesky factorization (discussed above) to factor $B \rightarrow LL^T$, where $L$ is lower triangular. Now $Ax = \lambda LL^Tx$ and hence $L^{-1}AL^{-T}(L^Tx) = \lambda(L^Tx)$, which means we can instead solve the symmetric eigenproblem $Cy = \lambda y$, where $C = L^{-1}AL^{-T}$ and $y = L^Tx$.

The operation that, given symmetric matrix $A$ and lower triangular matrix $L$, computes $A := L^{-1}AL^{-T}$ is called the two-sided triangular solve operation. There are obviously a number of related operations:

$$\begin{array}{rcl} A & := & L^{-1}AL^{-T} \\ A & := & L^{-T}AL^{-1} \\ A & := & U^{-1}AU^{-T} \\ A & := & U^{-T}AU^{-1}, \end{array}$$

where $L$ is a lower triangular matrix and $U$ is an upper triangular matrix. (This time, these triangular matrices are never unit triangular matrices, except by sheer coincidence.) Resulting algorithms/functions are

- TRSM2SIDED_LN_UNB_VARX(L, B) where LN stands for **l**ower triangular, **n**o transpose,

- TRSM2SIDED_LT_UNB_VARX(L, B) where LT stands for **l**ower triangular, **t**transpose,
- TRSM2SIDED_UN_UNB_VARX(L, B) where UN stands for **u**pper triangular, **n**o transpose,
- TRSM2SIDED_UT_UNB_VARX(L, B) where UT stands for **u**pper triangular, **t**transpose.

This operation and the closely related two-sided triangular matrix-matrix multiplication operation, are discussed in

> Jack Poulson, Robert van de Geijn, and Jeffrey Bennighof.
> ☛ Parallel Algorithms for Reducing the Generalized Hermitian-Definite Eigenvalue Problem.
> FLAME Working Note #56. The University of Texas at Austin, Department of Computer Science. Technical Report TR-11-05. Feb. 2011.

**Two-sided triangular matrix-matrix multiplication.** The operation that, given symmetric matrix $A$ and lower triangular matrix $L$, computes $A := LAL^T$ is called the two-sided triangular matrix-matrix multiplication operation. There are obviously a number of related operations:

$$\begin{aligned} A &:= LAL^T \\ A &:= L^TAL \\ A &:= UAU^T \\ A &:= U^TAU, \end{aligned}$$

where $L$ is a lower triangular matrix and $U$ is an upper triangular matrix. (Again, these triangular matrices are never unit triangular matrices, except by sheer coincidence.) Resulting algorithms/functions are

- TRMM2SIDED_LN_UNB_VARX(L, B) where LN stands for **l**ower triangular, **n**o transpose,
- TRMM2SIDED_LT_UNB_VARX(L, B) where LT stands for **l**ower triangular, **t**transpose,
- TRMM2SIDED_UN_UNB_VARX(L, B) where UN stands for **u**pper triangular, **n**o transpose,
- TRMM2SIDED_UT_UNB_VARX(L, B) where UT stands for **u**pper triangular, **t**transpose.

## Inversion of matrices

Generally speaking, it is ill-advised to explicitly invert a matrix. Still, on occasion there is reason to do so.

**Inverting a triangular matrix.** Let us consider inverting lower triangular matrix $L$. It is well-known that its inverse is itself lower triangular.

Notice that

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)^{-1} = \left(\begin{array}{c|c} L_{TL}^{-1} & 0 \\ \hline -L_{BR}^{-1}L_{BL}L_{TL} & L_{BR}^{-1} \end{array}\right),$$

which is one way to express the PME. (You can check this by multiplying out $LL^{-1}$ with the partitioned matrices, to find this yields the identity.) This approach is employed in

> Paolo Bientinesi, Brian Gunter, Robert A. van de Geijn.
> Families of algorithms related to the inversion of a Symmetric Positive Definite matrix.
> ACM Transactions on Mathematical Software (TOMS), 2008.

in which you will also find derivations for many related operations.

Alternatively, one can view this as solving $LX = I$ where $I$ is the identity and lower triangular matrix $X$ overwrites $L$. This yields one PME. A second one results from instead considering $XL = I$.

Obviously, these observations can be extended to the computation of the inverse of an upper triangular matrix $U$.

**Inverting a symmetric positive definite (SPD) matrix** The paper

> Paolo Bientinesi, Brian Gunter, Robert A. van de Geijn. Families of algorithms related to the inversion of a Symmetric Positive Definite matrix.
> ACM Transactions on Mathematical Software (TOMS), 2008.

as the title suggests, also discusses the inversion of SPD matrices.

**Inverting a general (square) matrix.** The first journal paper that used what we now call the FLAME notation dealt with the high-performance parallelization of a general matrix (and includes pivoting):

> Xiaobai Sun, Enrique S. Quintana, Gregorio Quintana, and Robert van de Geijn.
> A Note on Parallel Matrix Inversion.
> SIAM Journal on Scientific Computing, Vol. 22, No. 5, pp. 1762–1771, 2001.

At the time, we did not yet explicitly use the FLAME methodology to derive algorithms.

If one were to derive algorithms for computing $A^{-1}$, one might use the observation that

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)^{-1} = \text{some nasty expession}$$

from which then loop invariants can be derived. Alternatively, one could consider

$$AX = I$$

and

$$XA = I$$

to derive PMEs and loop invariants.

Additing pivoting would make for a really interesting exercise...

## Operations encountered in control theory

We have also applied the FLAME methodology to two fundamental operations encountered in control theory: solving the triangular Sylvester equation and its symmetric counterpart, the triangular Lyapunov equation.

**The triangular Sylvester equation.** The continuous Sylvester equation is given by

$$AX + XB + Y = 0,$$

where $A$, $B$, and $Y$ are given, and $X$ is to be computed, overwriting $Y$. This problem is related to the eigenvalue problem and hence, as a general rule, the FLAME methodology does not apply. However, after initial manipulation, the problem becomes

$$LX + XU + C = 0,$$

where $L$ is lower triangular and $U$ is upper triangular. A variation of this operation was one of the first more complicated case studies pursued after the FLAME methodology was first unveiled, in

> Enrique S. Quintana-Orti, Robert A. van de Geijn.
> Formal derivation of algorithms: The triangular Sylvester equation.
> ACM Transactions on Mathematical Software (TOMS), 2003.

**The triangular Lyapunov equation** The continuous Lyapunov equation is given by

$$AX + XA^T + Y = 0,$$

where $A$ and symmetric $Y$ are given, and $X$ is to be computed, overwriting $Y$. Again, this problem is related to the eigenvalue problem and hence, as a general rule, the FLAME methodology does not apply. However, after initial manipulation the problem becomes

$$LX + XL^T + C = 0,$$

where $L$ is lower triangular and $C$ is still SPD. This operation is used as an example in

> Paolo Bientinesi, John Gunnels, Maggie Myers, Enrique Quintana-Ortí, Tyler Rhodes, Robert van de Geijn, and Field Van Zee.
> Deriving Linear Algebra Libraries
> Formal Aspects of Computing, November 2013, Volume 25, Issue 6, pp 933-–945.

# Answers

**Homework 1.2.1.1** Decide whether or not the following are propositions:

1. Is it raining?
   (a) This is not a proposition.
   (b) This is a proposition and it evaluates to TRUE.
   (c) This is a proposition and it evaluates to FALSE.

   **Answer:** This is not a proposition since it is a question, not a declarative statement.

2. Shut the window when it is raining!
   (a) This is not a proposition.
   (b) This is a proposition and it evaluates to TRUE.
   (c) This is a proposition and it evaluates to FALSE.

   **Answer:** This is not a proposition since it is a command, not a declarative statement.

3. There is a number between 0 and 5 that is even.
   (a) This is not a proposition.
   (b) This is a proposition and it evaluates to TRUE.
   (c) This is a proposition and it evaluates to FALSE.

   **Answer:** This is a proposition. It is a declarative statement that evalues to TRUE.

4. There is no number between 0 and 5 that is even.
   (a) This is not a proposition.
   (b) This is a proposition and it evaluates to TRUE.
   (c) This is a proposition and it evaluates to FALSE.

   **Answer:** This a proposition. It is a declarative statement evaluates to FALSE.

☛ BACK TO TEXT

**Homework 1.2.2.1** Complete the truth table for $(p \Rightarrow q) \wedge (q \Rightarrow r)$.

| $p$ | $q$ | $r$ | $p \Rightarrow q$ | $q \Rightarrow r$ | $(p \Rightarrow q) \wedge (q \Rightarrow r)$ |
|---|---|---|---|---|---|
| $T$ | $T$ | $T$ | $T$ | $T$ | $T$ |
| $T$ | $T$ | $F$ | $T$ | $F$ | $F$ |
| $T$ | $F$ | $T$ | $F$ | $T$ | $F$ |
| $T$ | $F$ | $F$ | $F$ | $T$ | $F$ |
| $F$ | $T$ | $T$ | $T$ | $T$ | $T$ |
| $F$ | $T$ | $F$ | $T$ | $F$ | $F$ |
| $F$ | $F$ | $T$ | $T$ | $T$ | $T$ |
| $F$ | $F$ | $F$ | $T$ | $T$ | $T$ |

☛ BACK TO TEXT

**Homework 1.2.3.1** Let $x$ and $y$ be variables that take on integer values. Let $p$ be the statement "$x$ is positive" and $q$ be the statement "$y$ is positive". Determine the symbolic statements for the following predicates described using English. Mark all that are correct. (There may be multiple correct answers.)

1. Both $x$ and $y$ are positive.
   (a) $p \wedge q$
   (b) $p \vee q$
   (c) $\neg(p \wedge q)$
   (d) $p \Rightarrow q$

   **Answer:** (a) $p \wedge q$; We could rephrase this sentence as X is positive and Y is positive.

2. Either $x$ or $y$ is positive.
   (a) $p \wedge q$
   (b) $p \vee q$
   (c) $\neg(p \wedge q)$
   (d) $p \Rightarrow q$

   **Answer:** (b) $p \vee q$; We could rephrase this sentence as $x$ is positive or $y$ is positive.

3. $x$ is positive but $y$ is not.
   (a) $\neg q$
   (b) $\neg(p \wedge q)$
   (c) $\neg p \vee \neg q$
   (d) $p \wedge \neg q$

   **Answer:** (d) $p \wedge \neg(q)$; We could rephrase this sentence as $x$ is positive and not ($y$ is positive).

4. Either $x$ or $y$ is positive but not both are positive.
   (a) $p \wedge q$
   (b) $(p \vee q) \wedge \neg(p \wedge q)$
   (c) $\neg p \vee q$
   (d) $(p \wedge \neg q) \vee (\neg p \wedge q)$

   **Answer:** Both (b) and (d). We could rephrase this sentence as $x$ is positive or $y$ is positive and not both $x$ is positive and $y$ is positive. This we could also rephrase as exactly one of $x$ and $y$ is positive giving us (d). This Boolean operator is often called the exclusive or and sometime denoted as XOR. Notice this means that the expression in (b) is equivalent to the expression in (d).

5. $x$ is not positive and $y$ is not positive.
   (a) $\neg(p \wedge q)$
   (b) $\neg(p \vee q)$
   (c) $\neg p \wedge \neg q$
   (d) $\neg p \vee \neg q$

   **Answer:** c) Self-explanatory!

6. At least one of $x$ and $y$ is not positive.
   (a) $\neg(p \wedge q)$
   (b) $\neg(p \vee q)$
   (c) $\neg p \wedge \neg q$
   (d) $\neg p \vee \neg q$

   **Answer:** (d) $\neg(p) \vee \neg(q)$ and (a) $\neg(p \wedge q)$. We could rephrase this sentence as "$x$ is not positive or $y$ is not positive" which justifies (d). However, we could also rephrase this as "It is not the case that $x$ and $y$ are both positive". Notice this means that the expression in (a) is equivalent to the expression in (d).

7. Neither $x$ nor $y$ is positive.
   (a) $\neg p$
   (b) $\neg(p \wedge q)$
   (c) $\neg p \wedge \neg q$
   (d) $\neg p \vee \neg q$

   **Answer:** (c) $\neg(p) \wedge \neg(q)$. We could rephrase this sentence as "$x$ is not positive and $y$ is not positive".

8. It is not the case that both $x$ and $y$ are positive.
   (a) $\neg(p \wedge q)$
   (b) $\neg(p \vee q)$
   (c) $\neg p \wedge q$
   (d) $p \wedge \neg q$

   **Answer:** (a) We could rephrase this as "Not ($x$ is positive and $y$ is positive)". We could also rephrase this sentence as "$x$ is not positive or $y$ is not positive".

9. Both $x$ and $y$ are not positive.
   (a) $\neg p$
   (b) $\neg(p \wedge q)$
   (c) $\neg p \wedge \neg q$
   (d) $\neg p \vee \neg q$
   (e) not clear

   **Answer:** (e) The most likely answer is $\neg(p) \wedge \neg(q)$; We could rephrase this sentence as "$x$ is not positive and $y$ is not positive". However, sometimes in English we confuse this with it is not the case that both are positive. It is often difficult to determine where the parentheses are in English statements. Notice that we could also rephrase this as "neither $x$ nor $y$ are positive". So, we declare it "not clear".

10. If x is positive then y is positive.
    (a) $p \wedge q$
    (b) $p \Rightarrow q$
    (c) $p \vee q$

☛ BACK TO TEXT

**Homework 1.2.4.1** Let $p = F$, $q = F$, and $r = F$. Evaluate

- $p \wedge q \Rightarrow r$

  **Answer:** $\underbrace{\underbrace{F \wedge F}_{F} \Rightarrow F}_{T}$

- $(p \wedge q) \Rightarrow r$

  **Answer:** $\underbrace{\underbrace{(F \wedge F)}_{F} \Rightarrow F}_{T}$

- $p \wedge (q \Rightarrow r)$

  **Answer:** $\underbrace{F \wedge \underbrace{(F \Rightarrow F)}_{T}}_{F}$

☛ BACK TO TEXT

**Homework 1.2.4.2** Evaluate

$$T \vee \neg T \wedge F \Rightarrow T \wedge \neg T \Leftrightarrow T \Rightarrow F.$$

a) $T$

b) $F$

**Answer:**

$$\underbrace{\underbrace{\underbrace{T \vee \underbrace{\underbrace{\neg T}_{F} \wedge F}_{F}}_{T} \Rightarrow \underbrace{T \wedge \underbrace{\neg T}_{F}}_{F}}_{F} \Leftrightarrow \underbrace{T \Rightarrow F}_{F}}_{T}$$

☛ BACK TO TEXT

**Homework 1.2.5.1** Use a truth table to prove the commutativity of the $\wedge$ operator:

$$(\text{E1} \wedge \text{E2}) \Leftrightarrow (\text{E2} \wedge \text{E1}).$$

**Answer:**

| E1 | E2 | $(\text{E1} \wedge \text{E2})$ | $(\text{E2} \wedge \text{E1})$ | $(\text{E1} \wedge \text{E2}) \Leftrightarrow (\text{E2} \wedge \text{E1})$ |
|---|---|---|---|---|
| $T$ | $T$ | $T$ | $T$ | $T$ |
| $T$ | $F$ | $F$ | $F$ | $T$ |
| $F$ | $T$ | $F$ | $F$ | $T$ |
| $F$ | $F$ | $F$ | $F$ | $T$ |

☛ BACK TO TEXT

**Homework 1.2.5.2** Use a truth table to prove

$$p \wedge q \Rightarrow p.$$

**Answer:**

| $p$ | $q$ | $(p \wedge q)$ | $(p \wedge q) \Rightarrow p$ |
|---|---|---|---|
| $T$ | $T$ | $T$ | $T$ |
| $T$ | $F$ | $F$ | $T$ |
| $F$ | $T$ | $F$ | $T$ |
| $F$ | $F$ | $F$ | $T$ |

☛ BACK TO TEXT

**Homework 1.3.2.1** Prove that $(p \Rightarrow (q \wedge r)) \Leftrightarrow ((p \Rightarrow q) \wedge (p \Rightarrow r))$.

**Answer:**

$$\begin{array}{l} \quad (p \Rightarrow (q \wedge r)) \\ \Leftrightarrow \ < \text{implication} > \\ \quad \neg p \vee (q \wedge r) \\ \Leftrightarrow \ < \text{distributivity} > \\ \quad (\neg p \vee q) \wedge (\neg p \vee r) \\ \Leftrightarrow \ < \text{implication } \times 2 > \\ \quad (p \Rightarrow q) \wedge (p \Rightarrow r) \end{array}$$

☛ BACK TO TEXT

**Homework 1.3.2.2** Prove that $(p \Rightarrow (q \Rightarrow r)) \Leftrightarrow (p \wedge q \Rightarrow r)$.

**Answer:**

$$\begin{array}{l} \quad p \Rightarrow (q \Rightarrow r) \\ \Leftrightarrow < \text{implication} > \\ \quad \neg p \vee (q \Rightarrow r) \\ \Leftrightarrow < \text{implication} > \\ \quad \neg p \vee \neg q \vee r \\ \Leftrightarrow < \text{associativity} > \\ \quad (\neg p \vee \neg q) \vee r \\ \Leftrightarrow < \text{De Morgan's} > \\ \quad \neg(p \wedge q) \vee r \\ \Leftrightarrow < \text{implication} > \\ \quad p \wedge q \Rightarrow r \end{array}$$

☛ BACK TO TEXT

**Homework 1.3.2.3** Prove that $(p \Rightarrow (q \vee r)) \Leftrightarrow ((p \Rightarrow q) \vee (p \Rightarrow r))$.

**Answer:**

$$\begin{array}{l} \quad (p \Rightarrow (q \vee r) \\ \Leftrightarrow < \text{implication} > \\ \quad \neg p \vee (q \vee r) \\ \Leftrightarrow < \vee\text{-simplification} > \\ \quad \neg p \vee \neg p \vee (q \vee r)) \\ \Leftrightarrow < \text{commutativity; associativity} \ > \\ \quad (\neg p \vee q) \vee (\neg p \vee r) \\ \Leftrightarrow < \text{implication} \times 2 > \\ \quad (p \Rightarrow q) \vee (p \Rightarrow r) \end{array}$$

☛ BACK TO TEXT

**Homework 1.3.4.1** Let $n \geq 1$. Then $\sum_{i=1}^{n} i = n(n+1)/2$.

Always/Sometimes/Never

**Answer:**

**VideoInfoTwoThreeTwoOne**

We can prove this in three different ways:

1. By mathematical induction, carefully mimicing the proof that $\sum_{i=0}^{n-1} i = (n-1)n/2$; or
2. Using a trick similar to the one used in the alternative proof given for $\sum_{i=0}^{n-1} i = (n-1)n/2$; or
3. Using the fact that $\sum_{i=0}^{n-1} i = n(n-1)/2$.

☛ BACK TO TEXT

**Homework 1.3.4.2** Let $n \geq 1$. $\sum_{i=0}^{n-1} 1 = n$.

Always/Sometimes/Never

**Answer:** Always.

**Base case:** $n = 1$. For this case, we must show that $\sum_{i=0}^{1-1} 1 = 1$.

$\sum_{i=0}^{1-1} 1$

$=$ (Definition of summation)

$1$

This proves the base case.

**Inductive step:** Inductive Hypothesis (IH): Assume that the result is true for $n = k$ where $k \geq 1$:

$$\sum_{i=0}^{k-1} 1 = k.$$

We will show that the result is then also true for $n = k+1$:

$$\sum_{i=0}^{(k+1)-1} 1 = (k+1).$$

Assume that $k \geq 1$. Then

$\sum_{i=0}^{(k+1)-1} 1$

$=$ (arithmetic)

$\sum_{i=0}^{k} 1$

$=$ (split off last term)

$\left(\sum_{i=0}^{k-1} 1\right) + 1$

$=$ (I.H.)

$k+1$.

This proves the inductive step.

**By the Principle of Mathematical Induction** the result holds for all $n$.

☛ BACK TO TEXT

**Homework 1.3.4.3** Let $n \geq 1$ and $x \in \mathbb{R}^m$. Then

$$\sum_{i=0}^{n-1} x = \underbrace{x + x + \cdots + x}_{n \text{ times}} = nx$$

Always/Sometimes/Never

**Answer:** Always.

$$\sum_{i=0}^{n-1} x = \left(\sum_{i=0}^{n-1} 1\right) x = nx.$$

However, we want you to prove this with mathematical induction:

**Base case:** $n = 1$. For this case, we must show that $\sum_{i=0}^{1-1} x = x$.

$\sum_{i=0}^{1-1} x$

$=$ $\quad < \text{Definition of summation} >$

$x$

This proves the base case.

**Inductive step:** Inductive Hypothesis (IH): Assume that the result is true for $n = k$ where $k \geq 1$:

$$\sum_{i=0}^{k-1} x = kx.$$

We will show that the result is then also true for $n = k+1$:

$$\sum_{i=0}^{(k+1)-1} x = (k+1)x.$$

Assume that $k \geq 1$. Then

$\sum_{i=0}^{(k+1)-1} x$

$=$ $\quad < \text{arithmetic} >$

$\sum_{i=0}^{k} x$

$=$ $\quad < \text{split off last term} >$

$\sum_{i=0}^{k-1} x + x$

$=$ $\quad < \text{I.H.} >$

$kx + x$

$=$ $\quad < \text{algebra} >$

$(k+1)x$

This proves the inductive step.

**By the Principle of Mathematical Induction** the result holds for all $n$.

☛ BACK TO TEXT

**Homework 1.3.4.4** Let $n \geq 1$. $\sum_{i=0}^{n-1} i^2 = (n-1)n(2n-1)/6$.

Always/Sometimes/Never

**Answer: VideoInfoTwoThreeTwoFour**

Always

**Base case:** $n = 1$. For this case, we must show that $\sum_{i=0}^{1-1} i^2 = (1-1)(1)(2(1)-1)/6$. But $\sum_{i=0}^{1-1} i^2 = 0 = (1-1)(1)(2(1)-1)/6$. This proves the base case.

**Inductive step:** Inductive Hypothesis (IH): Assume that the result is true for $n = k$ where $k \geq 1$:

$$\sum_{i=0}^{k-1} i^2 = (k-1)k(2k-1)/6.$$

We will show that the result is then also true for $n = k+1$:

$$\sum_{i=0}^{(k+1)-1} i^2 = ((k+1)-1)(k+1)(2(k+1)-1)/6 = (k)(k+1)(2k+1)/6.$$

Assume that $k \geq 1$. Then

$\sum_{i=0}^{(k+1)-1} i^2$

$=$ (arithmetic)

$\sum_{i=0}^{k} i^2$

$=$ (split off last term)

$\sum_{i=0}^{k-1} i^2 + k^2$

$=$ (I.H.)

$(k-1)k(2k-1)/6 + k^2$

$=$ (algebra)

$[(k-1)k(2k-1) + 6k^2]/6.$

Now,

$$(k)(k+1)(2k+1) = (k^2+k)(2k+1) = 2k^3 + 2k^2 + k^2 + k = 2k^3 + 3k^2 + k$$

and

$$(k-1)k(2k-1) + 6k^2 = (k^2-k)(2k-1) + 6k^2 = 2k^3 - 2k^2 - k^2 + k + 6k^2 = 2k^3 + 3k^2 + k.$$

Hence

$$\sum_{i=0}^{(k+1)-1} i^2 = (k)(k+1)(2k+1)/6$$

This proves the inductive step.

**By the Principle of Mathematical Induction** the result holds for all $n$.

☛ BACK TO TEXT

**Homework 1.4.6.1** Let us consider a one dimensional array $b(1:n)$ (using Matlab notation), where $1 \leq n$. Let $j$ and $k$ be two integer variables satisfying $1 \leq j \leq k \leq n$. By $b(j:k)$ we mean the subarray of $b$ consisting of $b(j), b(j+1), \ldots b(k)$. The segment $b(j:k)$ is empty (contains no elements) if $j > k$.

Translate the following sentences into predicates.

1. All elements in the subarray $b(j:k)$ are positive.

   **Answer:** $(\forall i \mid j \leq i \leq k : b(i) > 0)$

2. No element in the subarray $b(j:k)$ is positive.

   **Answer:** Some correct translations into logic notation:

- Literal translation: (there does not exist an element in the subarray that is positive) $\neg(\exists i \mid j \leq i \leq k : b(i) > 0)$.
- All elements in the subarray are not postive: $(\forall i \mid j \leq i \leq k : \neg(b(i) > 0))$ or $(\forall i \mid j \leq i \leq k : b(i) \leq 0)$.

3. It is not the case that all elements in the subarray $b(j : k)$ are positive.

   **Answer:** Some correct translations into logic notation:

   - Literal translation:$(\forall i \mid j \leq i \leq k : b(i) > 0)$.
   - There exists an element in the subarray that is not positive: $(\exists i \mid j \leq i \leq k : \neg(b(i) > 0))$ or $(\exists i \mid j \leq i \leq k : b(i) \leq 0)$.

4. All elements in the subarray $b(j : k)$ are not positive.

   **Answer:** Some correct translations into logic notation:

   - Literal translation:$(\forall i \mid j \leq i \leq k :\neq (b(i) > 0))$ or $(\forall i \mid j \leq i \leq k : b(i) \leq 0)$.
   - There does not exist an element in the subarray that is positive: $\neg(\exists i \mid j \leq i \leq k : b(i) > 0)$.

5. Some element in the subarray $b(j : k)$ is positive.

   **Answer:** Some correct translations into logic notation:

   - Literal translation: (there exists an element in the subarray that is positive) $(\exists i \mid j \leq i \leq k : b(i) > 0)$.

   Notice that the English sentence is somewhat ambiguous: is exactly one element positive or at least one element?

6. There is an element in the subarray $b(j : k)$ that is positive.

   **Answer:** Some correct translations into logic notation:

   - Literal translation: $(\exists i \mid j \leq i \leq k : b(i) > 0)$.

   Again, the English sentence is somewhat ambiguous.

7. At least one element in the subarray $b(j : k)$ is positive.

   **Answer:** Some correct translations into logic notation:

   - Literal translation: $(\exists i \mid j \leq i \leq k : b(i) > 0)$.
   - Not all elements in the subarray are not positive: $\neg(\forall i \mid j \leq i \leq k : \neg(b(i) > 0))$ or $\neg(\forall i \mid j \leq i \leq k : b(i) \leq 0)$.

   Now the English sentence is not ambiguous.

8. Some element in the subarray $b(j : k)$ is not positive.

   **Answer:** Some correct translations into logic notation:

   - Literal translation: $(\exists i \mid j \leq i \leq k : \neg(b(i) > 0))$ or $(\exists i \mid j \leq i \leq k : b(i) \leq 0)$.
   - Not all elements in the subarray are positive: $\neg(\forall i \mid j \leq i \leq k : b(i) > 0)$.

   Again, the English sentence is somewhat ambiguous.

9. Not all elements in the subarray $b(j : k)$ are positive.

   **Answer:** Some correct translations into logic notation:

- Literal translation: $\neg(\forall i \mid j \le i \le k : b(i) > 0)$.
- There exists at least one element in the subarray that is not positive: $(\exists i \mid j \le i \le k : \neg(b(i) > 0))$ or $(\exists i \mid j \le i \le k : b(i) \le 0)$.

10. It is not the case that there is an element in the subarray $b(j:k)$ that is positive.

    **Answer:** Some correct translations into logic notation:

    - Literal translation: $\neg(\exists i \mid j \le i \le k : b(i) > 0)$.
    - All elements in the subarray are not positive: $(\forall i \mid j \le i \le k : \neg(b(i) > 0))$ or $(\forall i \mid j \le i \le k : b(i) \le 0)$.

☛ BACK TO TEXT

**Homework 1.4.6.2** Translate the following sentence into a predicate: Exactly one element in the subarray $b(j:k)$ is positive.

1. $(\exists i \mid j \le i \le k : b(i) > 0 \wedge (\forall p \mid j \le p \le k \wedge p \ne i : \neg(b(p) > 0)))$
2. $(\exists i \mid j \le i \le k : b(i) > 0) \wedge (\forall p \mid j \le p \le k \wedge p \ne i : \neg(b(p) > 0))$

**Answer:** There exists an element in the subarray $b(j:k)$ that is positive and all other elements in that subarray are not positive: $(\exists i \mid j \le i \le k : b(i) > 0 \wedge (\forall p \mid j \le p \le k \wedge p \ne i : \neg(b(i) > 0)))$. Notice that the quantifiers have to be "nested"

☛ BACK TO TEXT

**Homework 1.4.6.3** Formalize the following English specifications. Be sure to introduce necessary restrictions.

1. Set $s$ equal to the sum of the elements of $b(j:k)$.

   **Answer:**

   $$s = (\sum i \mid j \le i \le k : b(i))$$

   What if $j:k$ is empty? This should return zero, which this properly captures.

2. Set $M$ equal to the maximum value in $b(j:k)$.

   **Answer:**

   $$(\exists i \mid j \le i \le k : M = b(i)) \wedge (\forall i \mid j \le i \le k : b(i) \le M)$$

   What if $j:k$ is empty? Then the $\exists$ clause evaluates to $F$ and hence the whole thing evaluates to false. This means that the program should not complete.

3. Set $I$ equal to the index of a maximum value of $b(j:k)$.

   **Answer:**

   $$(j \le I \le k) \wedge (\forall i \mid j \le i \le k : b(i) \le b(I))$$

   What if $j:k$ is empty? Then the $(j \le U \le k)$ clause evaluates to $F$ and hence the whole thing evaluates to false. This means that the program should not complete.

4. Calculate $x$, the greatest power of 2 that is not greater than positive integer $n$.

   **Answer:**

$$1 \leq x \leq n \wedge (\exists i \mid 0 \leq i : x = 2^i \wedge \frac{n}{2} < x \leq n)$$

   or

$$1 \leq x \leq n \wedge (\exists i \mid 0 \leq i : x = 2^i) \wedge \neg(\exists i \mid x < 2^i \leq n)$$

5. Compute $c$, the number of zeroes in array $b(1:n)$.

   **Answer:**

$$(\sum i \mid 1 \leq i \leq n \wedge b(i) = 0 : 1)$$

6. Consider array of integers $b(1:n)$. Each of its subsegments $b(i:j)$ has a sum $S_{i,j} = (\sum \mid i \leq k \leq j : b(k))$. Compute $M$ equal to the maximum such sum.

   **Answer:**

$$(\exists i,j \mid 1 \leq i \leq j \leq n : M = S_{i,j}) \wedge \neg(\exists i,j \mid 1 \leq i \leq j \leq n : M < S_{i,j})$$

   or

$$(\exists i,j \mid 1 \leq i \leq j \leq n : M = S_{i,j}) \wedge (\forall i,j \mid 1 \leq i \leq j \leq n : M \geq S_{i,j})$$

☛ BACK TO TEXT

**Homework 1.5.2.1** For each of the following, if applicable, indicate which statement is TRUE (by examination):

1. (a) $0 \leq x \leq 10$ is weaker than $1 \leq x < 5$.
   (b) $0 \leq x \leq 10$ is stronger than $1 \leq x < 5$.

   **Answer:** By examination, $0 \leq x \leq 10$ is weaker than $1 \leq x < 5$. Your reason could be "If $x$ is between one and five (inclusive one and five) then it is certainly between zero and ten (inclusive zero and ten)."

2. (a) $x = 5 \wedge y = 4$ is weaker than $y = 4$.
   (b) $x = 5 \wedge y = 4$ is stronger than $y = 4$.

   **Answer:** By examination, $y = 4$ is weaker. If $x$ equals 5 and $y$ equals 4, then $y$ equals 4. Another way of thinking about this is "$x = 5$ AND $y = 4$ is more restrictive on what values $x$ and $y$ can take on than when we only require $y$ to equal 4."

3. (a) $x \leq 5 \vee y = 3$ is weaker than $x = 5 \wedge y = 4$.
   (b) $x \leq 5 \vee y = 3$ is stronger than $x = 5 \wedge y = 4$.

   **Answer:** By examination, $x \leq 5 \vee y = 3$ is weaker. If $x$ equals 5 and $y$ equals 4, then $x$ equals 5 must be TRUE and hence $x$ equal to 5 OR $y$ equal to 3 must be TRUE.

4. (a) $T$ is weaker than $F$.
   (b) $T$ is stronger than $F$.

   **Answer:** By examination, $T$ is weaker. "If FALSE is TRUE then certainly TRUE is TRUE. Or, "The set of states that satisfies the condition where FALSE is TRUE is included in the set of states where TRUE is TRUE." (After all, the set of states for which FALSE is TRUE is empty, and all elements in an empty set are in any other set, including the set of states where TRUE is TRUE.)

5. (a) $(\forall i | 5 \leq i \leq 10 : b(i+1) < b(i))$ is weaker than $(\forall i | 7 \leq i \leq 10 : b(i+1) < b(i))$.

   (b) $(\forall i | 5 \leq i \leq 10 : b(i+1) < b(i))$ is stronger than $(\forall i | 7 \leq i \leq 10 : b(i+1) < b(i))$.

   **Answer:** By examination, $(\forall i | 5 \leq i \leq 10 : b(i+1) < b(i))$ is stronger. $(\forall i | 5 \leq i \leq 10 : b(i+1) < b(i))$ means $b(5) > b(6) > b(7) > b(8) > b(9) > b(10) > b(11)$ which implies that $b(7) > b(8) > b(9) > b(10) > b(11)$.

6. (a) $x \leq 1$ is weaker than $x \geq 5$.

   (b) $x \leq 1$ is stronger than $x \geq 5$.

   **Answer:** Neither of these implies the other, so neither is weaker than the other. How do we argue (indeed *prove*) this? $x = 1$ satisfies $x \leq 1$ but not $x \geq 5$. So, $x \leq 1 \Rightarrow x \geq 5$ is not TRUE. Similarly, $x = 5$ satisfies $x \geq 5$ but not $x \leq 1$. So, $x \geq 5 \Rightarrow x \leq 1$ is not TRUE.

7. (a) $x \leq 4$ is weaker than $5 > x$.

   (b) $x \leq 4$ is stronger than $5 > x$.

   **Answer:** This is a trick question. These two expressions are equivalent. We have decided that any predicate is simultaneously stronger and weaker than itself. So both are TRUE.

☛ BACK TO TEXT

**Homework 1.5.2.2** Use the Basic Equivalences to prove the following. (Do NOT use the weakening/strengthening laws given in Figure 1.2, which we will discuss later.)

1. $\text{E1} \land \text{E2} \Rightarrow \text{E1}$

   **Answer:** Now, we noticed that in the last proof there was a point at which we would have liked to have said "well, dah, of course $b$ **and** $c$ implies $c$." This exercise shows that particular insight in isolation. Again, we use an "equivalence style" of proof:

$$\begin{array}{ll} & \text{E1} \land \text{E2} \Rightarrow \text{E1} \\ \Leftrightarrow & < \text{implication} > \\ & \neg(\text{E1} \land \text{E2}) \lor \text{E1} \\ \Leftrightarrow & < \text{DeMorgan's} > \\ & (\neg\text{E1} \lor \neg\text{E2}) \lor \text{E1} \\ \Leftrightarrow & < \text{commutativity; associativity; commutativity} > \\ & \neg\text{E2} \lor (\text{E1} \lor \neg\text{E1}) \\ \Leftrightarrow & < \text{excluded middle} > \\ & \neg\text{E2} \lor T \\ \Leftrightarrow & < \lor\text{-simplification} > \\ & T \end{array}$$

2. $\text{E1} \Rightarrow \text{E1} \lor \text{E3}$

   **Answer:** Here is another one of those "well, dah" problems. Let's prove it:

$$\begin{aligned}
& \mathrm{E1} \Rightarrow \mathrm{E1} \vee \mathrm{E3} \\
\Leftrightarrow & < \text{implication} > \\
& \neg \mathrm{E1} \vee (\mathrm{E1} \vee \mathrm{E3}) \\
\Leftrightarrow & < \text{associativity; commutativity} > \\
& (\mathrm{E1} \vee \neg \mathrm{E1}) \vee \mathrm{E3} \\
\Leftrightarrow & < \text{excluded middle} > \\
& T \vee \mathrm{E3} \\
\Leftrightarrow & < \text{commutativity; } \vee\text{-simplification} > \\
& T
\end{aligned}$$

3. $\mathrm{E1} \wedge \mathrm{E2} \Rightarrow \mathrm{E1} \vee \mathrm{E3}$ **Answer:** This one generalized the last two results.

$$\begin{aligned}
& \mathrm{E1} \wedge \mathrm{E2} \Rightarrow \mathrm{E1} \vee \mathrm{E3} \\
\Leftrightarrow & < \text{implication} > \\
& \neg(\mathrm{E1} \wedge \mathrm{E2}) \vee (\mathrm{E1} \vee \mathrm{E3}) \\
\Leftrightarrow & < \text{DeMorgan's} > \\
& (\neg \mathrm{E1} \vee \neg \mathrm{E2}) \vee (\mathrm{E1} \vee \mathrm{E3}) \\
\Leftrightarrow & < \text{commutativity; associativity; commutativity } \times 2 > \\
& \neg \mathrm{E2} \vee \mathrm{E3} \vee (\mathrm{E1} \vee \neg \mathrm{E1}) \\
\Leftrightarrow & < \text{excluded middle} > \\
& \neg \mathrm{E2} \vee \mathrm{E3} \vee T \\
\Leftrightarrow & < \vee\text{-simplification } \times 2 > \\
& T
\end{aligned}$$

☛ BACK TO TEXT

**Homework 1.5.2.3** Use the Basic Equivalences and/or the results from Homework 1.5.2.2 to prove that

$$\mathrm{E1} \wedge \mathrm{E2} \Rightarrow (\mathrm{E1} \vee \mathrm{E3}) \wedge \mathrm{E2}.$$

**Answer:**

$$\begin{aligned}
& \mathrm{E1} \wedge \mathrm{E2} \Rightarrow (\mathrm{E1} \vee \mathrm{E3}) \wedge \mathrm{E2} \\
\Leftrightarrow & < \wedge \text{ distributivity} > \\
& \mathrm{E1} \wedge \mathrm{E2} \Rightarrow (\mathrm{E1} \wedge \mathrm{E2}) \vee (\mathrm{E3} \wedge \mathrm{E2}) \\
\Leftrightarrow & < \text{Homework 1.5.2.2 } \textbf{??} > \\
& T
\end{aligned}$$

☛ BACK TO TEXT

**Homework 1.5.2.4** In Figure 1.2 we present three Weakening/Strengening Laws. This exercise shows that if you only decide to remember one, it should be the last one.

1. Show that $(\mathrm{E1} \wedge \mathrm{E2}) \Rightarrow \mathrm{E1}$ is a special case of $(\mathrm{E1} \wedge \mathrm{E2}) \Rightarrow (\mathrm{E1} \vee \mathrm{E3})$.

   **Answer:** $(\mathrm{E1} \wedge \mathrm{E2}) \Rightarrow (\mathrm{E1} \vee \mathrm{E3})$ is *true* for all expressions E1, E2, and E3. If you choose $\mathrm{E3} = F$ then you get $(\mathrm{E1} \wedge \mathrm{E2}) \Rightarrow \mathrm{E1}$

2. Show that E1 $\Rightarrow$ (E1 $\vee$ E3 ) is a special case of (E1 $\wedge$ E2 ) $\Rightarrow$ (E1 $\vee$ E3 ).

   **Answer:** (E1 $\wedge$ E2 ) $\Rightarrow$ (E1 $\vee$ E3 ) is *true* for all expressions E1 , E2 , and E3 . If you choose E2 $= T$ then you get E1 $\Rightarrow$ (E1 $\vee$ E3 )

☛ BACK TO TEXT

**Homework 1.5.2.5** For each of the following predicates pairs from Homework 1.5.2.1 use an equivalence style proof, the Basic Logic Equivalences, and the Weakening/strengthening laws to prove which predicate is weaker:

1. $0 \leq x \leq 10$ and $1 \leq x < 5$.

   **Answer:** By examination, $0 \leq x \leq 10$ is weaker. Proof:

$$
\begin{array}{ll}
 & (1 \leq x < 5) \Rightarrow (0 \leq x \leq 10) \\
\Leftrightarrow & < \text{algebra} > \\
 & (1 \leq x < 5) \Rightarrow (0 = x \vee 1 \leq x < 5 \vee 5 \leq x \leq 10) \\
\Leftrightarrow & < \text{weakening/strengening law} > \\
 & T
\end{array}
$$

2. $x = 5 \wedge y = 4$ and $y = 4$.

   **Answer:** By examination, $y = 4$ is weaker. Proof:

$$
\begin{array}{ll}
 & (x = 5 \wedge y = 4) \Rightarrow (y = 4) \\
\Leftrightarrow & < \text{weakening/strengening law} > \\
 & T
\end{array}
$$

3. $x \leq 5 \vee y = 3$ and $x = 5 \wedge y = 4$.

   **Answer:** By examination, $x \leq 5 \vee y = 3$ is weaker. Proof:

$$
\begin{array}{ll}
 & (x = 5 \wedge y = 4) \Rightarrow (x \leq 5 \vee y = 3) \\
\Leftrightarrow & < \text{algebra} > \\
 & (x = 5 \wedge y = 4) \Rightarrow (x < 5 \vee x = 5 \vee y = 3) \\
\Leftrightarrow & < \text{weakening/strengening law} > \\
 & T
\end{array}
$$

4. $T$ and $F$. **Answer:** By examination, $T$ is weaker. Proof:

$$
\begin{array}{ll}
 & F \Rightarrow T \\
\Leftrightarrow & < \wedge\text{-simplication} > \\
 & F \wedge T \Rightarrow T \\
\Leftrightarrow & < \text{weakening/strengening law} > \\
 & T
\end{array}
$$

5. $(\forall i|5 \leq i \leq 10 : b(i+1) < b(i))$ and $(\forall i|7 \leq i \leq 10 : b(i+1) < b(i))$ **Answer:** By examination, $(\forall i|5 \leq i \leq 10 : b(i+1) < b(i))$ is stronger. Proof:

$$(\forall i|5 \leq i \leq 10 : b(i+1) < b(i)) \Rightarrow (\forall i|7 \leq i \leq 10 : b(i+1) < b(i))$$

$\Leftrightarrow <$ split range $>$

$$(\forall i|5 \leq i \leq 6 : b(i+1) < b(i)) \wedge (\forall i|7 \leq i \leq 10 : b(i+1) < b(i)) \Rightarrow (\forall i|7 \leq i \leq 10 : b(i+1) < b(i))$$

$\Leftrightarrow <$ weakening/strengthening law $>$

$T$

☛ BACK TO TEXT

Figure 2.3: Annotated algorithm for computing the sum of the elements of array $b$.

**Homework 2.1.1.1** Consider again the algorithm that sums the elements of array $b$, now given in Figure 2.1. Place the following assertions in the correct place (the blank boxes) in the algorithm:

1. $\{ s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge k < n\}$
2. $\{ s = 0 \wedge 0 = k \leq n\}$
3. $\{ s = 0 \wedge 0 \leq n\}$
4. $\{ s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge \neg(k < n)\}$
5. $\{ s = (\sum i \mid 0 \leq i < k+1 : b(i)) \wedge 0 \leq k < n\}$
6. $\{ s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n\}$

**Answer:**

☛ BACK TO TEXT

**Homework 2.1.1.2** Take the solution for the last homework and use it to convince someone (possibly yourself) that the code segment is correct.

☛ BACK TO TEXT

5.2

**Homework 2.2.2.1** By examination, decide whether the following Hoare triples hold (evaluate to TRUE) and which of the predicates are stronger/weaker:

1. $\{x > 4\} y := x + 1 \{y > 5\}$ TRUE/FALSE

   **Answer: TRUE.**
   Clearly, $y$ will end up satisfying $\{y > 5\}$. This is a correct code segment.

2. $\{x = 10\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x = 10$ is stronger than $x > 4$ TRUE/FALSE

   **Answer: TRUE.**
   In this case, $y$ will end up equaling $y = 11$ and hence $\{y > 5\}$ holds after $y := x + 1$ is executed. In other words, the predicate $\{x = 10\} y := x + 1 \{y > 5\}$ evaluates to *true*. It is a correct code segment.

   Since $(x = 10) \Rightarrow (x > 4)$ the second statement is TRUE.

3. $\{x = 5\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is weaker than $x = 5$ TRUE/FALSE

   **Answer: TRUE.**
   Clearly, $y$ will end up equaling $y = 6$ and hence $\{y > 5\}$ holds after $y := x + 1$ is executed. The predicate $\{x = 5\} y := x + 1 \{y > 5\}$ evaluates to *true*. It is a correct code segment.

   Since $(x = 5) \Rightarrow (x > 4)$ the second statement is TRUE.

4. $\{x \geq 5\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is at least as weak as $x \geq 5$ TRUE/FALSE

   **Answer: TRUE.**
   Clearly, $y$ will end up satisfying $\{y > 5\}$. Again, the predicate $\{x = 5\} y := x + 1 \{y > 5\}$ evaluates to *true*. It is a correct code segment.

   Since $(x \geq 5) \Rightarrow (x > 4)$ the second statement is TRUE.

5. $\{x = 4\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is weaker than $x = 4$ TRUE/FALSE

   **Answer: FALSE.**
   Clearly, $y$ will end up equaling $y = 5$ and hence $\{y > 5\}$ does not holds after $y := x + 1$ is executed. The predicate $\{x = 4\} y := x + 1 \{y > 5\}$ evaluates to FALSE. It is a incorrect code segment.

   Since $x = 4$ does not imply that $x > 4$, the second statement is also FALSE.

6. $\{x \geq 4\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is weaker than $x \geq 4$ TRUE/FALSE

   **Answer: FALSE.**
   Clearly, $y$ will end up satisfying $\{y \geq 5\}$. This means $y > 5$ is not necessarily TRUE and hence this is an incorrect code segment.

   Since $x \geq 4$ does not imply that $x > 4$, the second statement is also FALSE.

7. $\{x > 4\} y := x + 1 \{y > 5\}$ TRUE/FALSE
   $x > 4$ is at least as weak as $x > 4$ TRUE/FALSE

   **Answer: TRUE**

☛ BACK TO TEXT

**Homework 2.2.2.2** What do you notice about the relationship between the preconditions, $P$, that make the Hoare triple $\{P\} y := x + 1 \{y > 5\}$ TRUE and the predicate $x > 4$? Choose the correct answer:

a) Of all $P$ for which the Hoare triple holds (evaluates to TRUE), $x > 4$ is the weakest.

b) Of all $P$ for which the Hoare triple holds (evaluates to TRUE), $x > 4$ is the strongest.

c) Of all $P$ for which the Hoare triple holds, $y > 5$ is the weakest.

d) No obvious relation.

**Answer:** a)
By examination:

- $\{P\} y := x + 1 \{y > 5\}$ evaluates to TRUE only when $P \Rightarrow (x > 4)$.
- Equivalently, the code segment

  $$\begin{array}{l} \{P\} \\ y := x + 1 \\ \{y > 5\} \end{array}$$

  is correct only when $x > 4$ is at least as weak as $P$.

- Yet another way of saying this: The program must be in a state where $x > 4$ before executing $y := x+1$ if it is to complete in a state where $y > 5$.
- And one more way: The set of all states (of variables) for which $y := x+1$ completes in a state where $y > 5$ is described by the predicate $x > 4$.

☛ BACK TO TEXT

**Homework 2.2.3.1** For each of the below code segments, determine the weakest precondition (by examination):

1. wp("$x := y$",$x = 5$) =

   **Answer:** $y = 5)$

2. wp("$x := x+1$",$0 \leq x \leq 1$) =

   **Answer:** $-1 \leq x \leq 0$

3. wp("$x := y$",$x = y$) =

   **Answer:** $T$

4. wp("$x := 4$",$x = 5$) =

   **Answer:** $F$

☛ BACK TO TEXT

**Homework 2.3.1.1** Consider the **skip** command, which simply doesn't do anything:

$\{Q : ?\}$
**skip**
$\{R : x > 4\}$

From what state $Q$ will the command **skip** finish (in a finite amount of time) in a state where $x > 4$ is TRUE? In other words,

wp("**skip**",$x > 4$) $= x > 4$

**Answer:** Obviously, $x > 4$ better be TRUE before the **skip** command is executed.

☛ BACK TO TEXT

**Homework 2.3.1.2** Building on the intuition from the last homework, give $wp($"**skip**"$,R) = R$ for an arbitrary post-condition $R$. **Answer:**

$$\text{wp}(\text{"}\textbf{skip}\text{"}, R) = R$$

for any predicate $R$: The set of all states such that executing **skip** completes (in a finite amount of time) in a state where $R$ is true is described by the predicate $R$ since what is true after the command is what is true before the command.

☛ BACK TO TEXT

**Homework 2.3.2.1** Consider the **abort** command, which aborts (which means execution does not reach the point in the program after the **abort** command).

$\{Q : ?\}$
**abort**
$\{R : x > 4\}$

From what state $Q$ will the command **abort** finish (in a finite amount of time) in a state where $x > 4$ is $T$? In other words, evaluate

$wp(\text{“}\textbf{abort}\text{”}, x > 4) = \mathit{FALSE}$

**Answer:** There is no state of the variables before the **abort** command such that it finishes in a state where $x > 4$. How do you describe the empty set of states? It is the set of states for which FALSE holds. (FALSE is never TRUE and hence there is no state for which this holds.)

☛ BACK TO TEXT

**Homework 2.3.2.2** Building on the intuition from the last homework, evaluate

$wp(\text{“}\textbf{abort}\text{”}, R) = \mathit{FALSE}$

**Answer:**

$$\text{wp}(\text{“}\textbf{abort}\text{”}, R) = \mathit{FALSE}$$

for any predicate $R$: The set of all states such that executing **abort** completes (in a finite amount of time) in a state where $R$ is true is empty.

☛ BACK TO TEXT

**Homework 2.3.3.1** Prove the following code segment correct:

| |
|---|
| $\{\ Q : s = 0 \wedge 0 \le n \ \}$ |
| $S : k := 0$ |
| $\{\ R : s = (\sum i \mid 0 \le i < k : b(i)) \wedge 0 \le k \le n \ \}$ |

Answer We need to prove that $Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)$:

$$\begin{array}{ll} & Q \Rightarrow \text{wp}(\text{“}S\text{”}, R) \\ \Leftrightarrow & < \text{instantiate } S \text{ and } R > \\ & Q \Rightarrow \text{wp}(\text{“}k := 0\text{”}, s = (\sum i \mid 0 \le i < k : b(i)) \wedge 0 \le k \le n) \\ \Leftrightarrow & < \text{definition of wp}(:=) > \\ & Q \Rightarrow (s = (\sum i \mid 0 \le i < k : b(i)) \wedge 0 \le k \le n)^{k}_{(0)} \\ \Leftrightarrow & < \text{definition of } R^{k}_{(\mathcal{E})} > \\ & Q \Rightarrow (s = (\sum i \mid 0 \le i < 0 : b(i)) \wedge 0 \le 0 \le n) \\ \Leftrightarrow & < \text{instantiate } Q\text{; sum over empty range; algebra} > \\ & (s = 0 \wedge 0 \le n) \Rightarrow (s = 0 \wedge 0 \le n) \\ \Leftrightarrow & < \Rightarrow\text{-simplification} > \\ & T \end{array}$$

where we skip valid($\mathcal{E}$) since it is obviously a valid expression.

☛ BACK TO TEXT

**Homework 2.3.3.2** Prove the following code segment correct:

| |
|---|
| $\{ \; Q : s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge k < n \; \}$ |
| $S : s := s + b(k)$ |
| $\{ \; R : s = (\sum i \mid 0 \leq i < k+1 : b(i)) \wedge 0 \leq k < n \; \}$ |

**Answer:** We need to prove that $Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)$:

$$
\begin{aligned}
& Q \Rightarrow \text{wp}(\text{“}S\text{”}, R) \\
\Leftrightarrow\ & < \text{Instantiate } S \text{ and } R > \\
& Q \Rightarrow \text{wp}(\text{“}s := s + b(k)\text{”}, s = (\sum i \mid 0 \leq i < k+1 : b(i)) \wedge 0 \leq k < n) \\
\Leftrightarrow\ & < \text{Definition of wp}(:=) > \\
& Q \Rightarrow (s = (\sum i \mid 0 \leq i < k+1 : b(i)) \wedge 0 \leq k < n)^{s}_{(s+b(k))} \\
\Leftrightarrow\ & < \text{Definition of } R^{s}_{(\mathcal{E})} > \\
& Q \Rightarrow ((s + b(k)) = (\sum \mid 0 \leq i < k+1 : b(i)) \wedge 0 \leq k < n) \\
\Leftrightarrow\ & < \text{Split range} > \\
& Q \Rightarrow (s + b(k) = (\sum \mid 0 \leq i < k : b(i)) + b(k) \wedge 0 \leq k < n) \\
\Leftrightarrow\ & < \text{Instantiate } Q\text{; algebra; } > \\
& (s = (\sum i \mid 0 \leq i < k : b(i)) \wedge 0 \leq k \leq n \wedge k < n) \Rightarrow \\
& \qquad (s = (\sum \mid 0 \leq i < k : b(i)) \wedge 0 \leq k < n) \\
\Leftrightarrow\ & < \text{Algebra} > \\
& T
\end{aligned}
$$

where we skip valid($\mathcal{E}$) since it is obviously a valid expression

☛ BACK TO TEXT

**Homework 2.3.4.1** Compute

1. $\text{wp}(\text{“}i := i - 1\text{”}, i \geq 0)$

   **Answer:**

$$
\begin{aligned}
& \text{wp}(\text{“}i := i - 1\text{”}, i \geq 0) \\
\Leftrightarrow\ & < \text{Definition of} := > \\
& i - 1 \geq 0 \\
\Leftrightarrow\ & < \text{algebra} > \\
& i \geq 1
\end{aligned}
$$

2. $\text{wp}(\text{“}i := i + 1\text{”}, i = j)$ **Answer:**

$$
\begin{aligned}
& \text{wp}(\text{“}i := i+1\text{”}, i = j) \\
\Leftrightarrow\ & <\text{Definition of} := > \\
& i+1 = j \\
\Leftrightarrow\ & <\text{algebra}> \\
& i = j-1
\end{aligned}
$$

3. $\text{wp}(\text{“}i := i+1; j := j+i\text{”}, i = j)$

   **Answer:**

$$
\begin{aligned}
& \text{wp}(\text{“}i := i+1; j := j+i\text{”}, i = j) \\
\Leftrightarrow\ & <\text{Definition of} := > \\
& \text{wp}(\text{“}i := i+1\text{”}, i = j+i) \\
\Leftrightarrow\ & <\text{Definition of} := > \\
& i+1 = j+i+1 \\
\Leftrightarrow\ & <\text{algebra}> \\
& 0 = j
\end{aligned}
$$

   or

$$
\begin{aligned}
& \text{wp}(\text{“}i := i+1; j := j+i\text{”}, i = j) \\
\Leftrightarrow\ & <\text{Definition of} := > \\
& \text{wp}(\text{“}i := i+1\text{”}, i = j+i) \\
\Leftrightarrow\ & <\text{algebra}> \\
& \text{wp}(\text{“}i := i+1\text{”}, 0 = j) \\
\Leftrightarrow\ & <\text{Definition of} := > \\
& 0 = j
\end{aligned}
$$

4. $\text{wp}(\text{“}i := 2i+1; j := j+i\text{”}, i = j)$

   **Answer:**

$$
\begin{aligned}
& \text{wp}(\text{“}i := 2i+1; j := j+i\text{”}, i = j) \\
\Leftrightarrow\ & <\text{Definition of} := > \\
& \text{wp}(\text{“}i := 2i+1\text{”}, i = j+i) \\
\Leftrightarrow\ & <\text{algebra}> \\
& \text{wp}(\text{“}i := 2i+1\text{”}, 0 = j) \\
\Leftrightarrow\ & <\text{Definition of} := > \\
& 0 = j
\end{aligned}
$$

5. $\text{wp}(\text{“}j := j+i; i := 2i+1\text{”}, i = j)$

**Answer:**

$$\begin{aligned}
& \text{wp}(``j := j+i; i := 2i+1", i = j) \\
\Leftrightarrow\ & < \text{Definition of} := > \\
& \text{wp}(``j := j+i", 2i+1 = j) \\
\Leftrightarrow\ & < \text{Definition of} := > \\
& 2i+1 = j+i \\
\Leftrightarrow\ & < \text{algebra} > \\
& i = j-1
\end{aligned}$$

6. wp(“$t := i; i := j; j := t$”, $i = \widehat{i} \wedge j = \widehat{j}$)

**Answer:**

$$\begin{aligned}
& \text{wp}(``t := i; i := j; j := t", i = \widehat{i} \wedge j = \widehat{j}) \\
\Leftrightarrow\ & < \text{Definition of} := > \\
& \text{wp}(``t := i; i := j", i = \widehat{i} \wedge t = \widehat{j}) \\
\Leftrightarrow\ & < \text{Definition of} := > \\
& \text{wp}(``t := i", j = \widehat{i} \wedge t = \widehat{j}) \\
\Leftrightarrow\ & < \text{Definition of} := > \\
& j = \widehat{i} \wedge i = \widehat{j}
\end{aligned}$$

7. wp(“$i := 0; s := 0$”, $0 \le i < n \wedge s = (\sum j | 0 \le j < i : b(j))$).

**Answer:**

$$\begin{aligned}
& \text{wp}(``i := 0; s := 0", 0 \le i < n \wedge s = (\textstyle\sum j | 0 \le j < i : b(j))) \\
\Leftrightarrow\ & < \text{Definition of} := > \\
& \text{wp}(``i := 0", 0 \le i < n \wedge 0 = (\textstyle\sum j | 0 \le j < i : b(j))) \\
\Leftrightarrow\ & < \text{Definition of} := > \\
& 0 \le 0 < n \wedge 0 = (\textstyle\sum j | 0 \le j < 0 : b(j)) \\
\Leftrightarrow\ & < \text{Algebra; empty range} > \\
& 0 < n \wedge 0 = 0 \\
\Leftrightarrow\ & < \text{Algebra} > \\
& 0 < n \wedge T \\
\Leftrightarrow\ & < \wedge\text{-simplification} > \\
& 0 < n
\end{aligned}$$

8. wp(“$s := s + b(i); i := i+1$”, $0 \le i \le n \wedge s = (\sum j | 0 \le j < i : b(j))$)

**Answer:**

$$\begin{aligned}
& \text{wp}(\text{“}s := s + b(i); i := i+1\text{”}, 0 \le i \le n \wedge s = (\textstyle\sum j | 0 \le j < i : b(j))) \\
\Leftrightarrow\ & < \text{Definition of } := > \\
& \text{wp}(\text{“}s := s + b(i)\text{”}, 0 \le i+1 \le n \wedge s = (\textstyle\sum j | 0 \le j < i+1 : b(j))) \\
\Leftrightarrow\ & < \text{Definition of } := > \\
& 0 \le i+1 \le n \wedge s + b(i) = (\textstyle\sum j | 0 \le j < i+1 : b(j)) \\
\Leftrightarrow\ & < \text{Split range} > \\
& 0 \le i+1 \le n \wedge s + b(i) = (\textstyle\sum j | 0 \le j < i : b(j)) + b(i) \\
\Leftrightarrow\ & < \text{Algebra} > \\
& 0 \le i+1 \le n \wedge s = (\textstyle\sum j | 0 \le j < i : b(j)) \\
\Leftrightarrow\ & < \text{Algebra} > \\
& -1 \le i < n \wedge s = (\textstyle\sum j | 0 \le j < i : b(j))
\end{aligned}$$

☛ BACK TO TEXT

**Homework 2.3.4.2** As part of the launch, you informally argued the correctness of the code segment

| |
|---|
| $\{\ Q:\ 0 \le n \ \}$ |
| $S_0:\ s := 0$ |
| $S_1:\ k := 0$ |
| $\{\ R:\ (s = (\sum i \mid 0 \le i < k : b(i))) \wedge (0 \le k \le n) \ \}$ |

where array $b$ has size $n$ with $0 \le n$. **Prove** this code segment correct. (In the “Wrap Up” you find another exercise related to the correctness of the program in the launch.)

**Answer:**

$$\begin{aligned}
& Q \Rightarrow \text{wp}(\text{“}S_0; S_1\text{”}, R) \\
\Leftrightarrow\ & < \text{instantiate} > \\
& Q \Rightarrow \text{wp}(\text{“}s := 0; k := 0\text{”}, (s = (\textstyle\sum i \mid 0 \le i < k : b(i))) \wedge (0 \le k \le n)) \\
\Leftrightarrow\ & < \text{definition of } :=\text{, twice} > \\
& Q \Rightarrow (0 = (\textstyle\sum i \mid 0 \le i < 0 : b(i))) \wedge (0 \le 0n \le n) \\
\Leftrightarrow\ & < \text{sum over empty range; algebra} > \\
& Q \Rightarrow (0 = 0 \wedge 0 \le n) \\
\Leftrightarrow\ & < \text{instantiate } Q\text{; algebra; } \wedge\text{-simplification} > \\
& (0 \le n) \Rightarrow (0 \le n) \\
\Leftrightarrow\ & < \Rightarrow\text{-simplification} > \\
& T
\end{aligned}$$

☛ BACK TO TEXT

**Homework 2.3.4.3** Consider an array $b$ of size $n$ with $0 \le n$, a scalar variable $s$, and the code segment

| |
|---|
| $\{ \; Q: \; (s = (\sum i \mid 0 \le i < k : b(i))) \wedge (0 \le k < n) \; \}$ |
| $S_0: \; s := s + b(k)$ |
| $S_1: \; k := k + 1$ |
| $\{ \; R: \; (s = (\sum i \mid 0 \le i < k : b(i))) \wedge (0 \le k \le n) \; \}$ |

which may be part of a program that sums the entries in array $b$. Prove this code segment correct.

**Answer:**

$Q \Rightarrow \text{wp}(\text{“}S_0; S_1'', R)$

$\Leftrightarrow <$ Composition of commands $>$

$Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, \text{wp}(\text{“}S_1\text{”}, R))$

$\Leftrightarrow <$ Instantiate $S_1$ $>$

$Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, \text{wp}(k := k+1, R))$

$\Leftrightarrow <$ Instantiate $R$; def. of wp$(:=, R)$ $>$

$Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (0 \le k \le n)))^k_{(k+1)})$

$\Leftrightarrow <$ Instantiate $S_0$; def. of $R^k_{(\mathcal{E})}$ $>$

$Q \Rightarrow \text{wp}(s := s + b(k), ((s = (\sum i \mid 0 \le i < k+1 : b(i)) \wedge (0 \le k+1 \le n)))$

$\Leftrightarrow <$ def. of wp$(:=, R)$ $>$

$Q \Rightarrow ((s = (\sum i \mid 0 \le i < k+1 : b(i)) \wedge (0 \le k+1 \le n)))^s_{(s+b(k))}$

$\Leftrightarrow <$ def. of $R^s_{(\mathcal{E})}$ $>$

$Q \Rightarrow ((s + b(k) = (\sum i \mid 0 \le i < k+1 : b(i)) \wedge (0 \le k+1 \le n)))$

$\Leftrightarrow <$ Split range $>$

$Q \Rightarrow ((s + b(k) = (\sum i \mid 0 \le i < k : b(i)) + b(k)) \wedge (0 \le k+1 \le n))$

$\Leftrightarrow <$ Instantiate $Q$; algebra; algebra $>$

$((s = (\sum i \mid 0 \le i < k : b(i))) \wedge (0 \le k < n)) \Rightarrow ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (-1 \le k < n)))$

$\Leftrightarrow <$ algebra $>$

$((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (0 \le k < n)))$

$\quad \Rightarrow ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge ((0 \le k < n) \vee (-1 = k))))$

$\Leftrightarrow <$ Weakening/strengthening $>$

$T$

☛ BACK TO TEXT

**Homework 2.3.4.4** Prove the following code segment, which swaps the values of variables $x$ and $y$, correct.

| |
|---|
| $\{ \; Q: \; (x = \widehat{x}) \wedge (y = \widehat{y}) \; \}$ |
| $S_0: \; t := x$ |
| $S_1: \; x := y$ |
| $S_2: \; y := t$ |
| $\{ \; R: \; (x = \widehat{y}) \wedge (y = \widehat{x}) \; \}$ |

**Answer:**

$$
\begin{array}{ll}
& Q \Rightarrow \text{wp}(\text{``}S_0; S_1; S''_2, R) \\
\Leftrightarrow & < \text{Composition of commands} > \\
& Q \Rightarrow \text{wp}(\text{``}S\text{''}_0, \text{wp}(\text{``}S\text{''}_1, \text{wp}(\text{``}S\text{''}_2, R))) \\
\Leftrightarrow & < \text{Instantiate } S_2 \text{ and } R > \\
& Q \Rightarrow \text{wp}(\text{``}S\text{''}_0, \text{wp}(\text{``}S\text{''}_1, \text{wp}(y := t, (x = \widehat{y}) \wedge (y = \widehat{x})))) \\
\Leftrightarrow & < \text{Definition of wp}(:=,) > \\
& Q \Rightarrow \text{wp}(\text{``}S\text{''}_0, \text{wp}(\text{``}S\text{''}_1, ((x = \widehat{y}) \wedge (y = \widehat{x}))^{y}_{(t)}))) \\
\Leftrightarrow & < \text{Instantiate } S_1, \text{ definition of } R^{y}_{(t)} > \\
& Q \Rightarrow \text{wp}(\text{``}S\text{''}_0, \text{wp}(x := y, ((x = \widehat{y}) \wedge (t = \widehat{x})))) \\
\Leftrightarrow & < \text{Definition of wp}(:=,) > \\
& Q \Rightarrow \text{wp}(\text{``}S\text{''}_0, ((x = \widehat{y}) \wedge (t = \widehat{x}))^{x}_{(y)}) \\
\Leftrightarrow & < \text{Instantiate } S_0, \text{ definition of } R^{x}_{(y)} > \\
& Q \Rightarrow \text{wp}(t := x, ((y = \widehat{y}) \wedge (t = \widehat{x}))) \\
\Leftrightarrow & < \text{Definition of wp}(:=,) > \\
& Q \Rightarrow ((y = \widehat{y}) \wedge (t = \widehat{x}))^{t}_{(x)} \\
\Leftrightarrow & < \text{Instantiate } Q, \text{ definition of } R^{t}_{(x)} > \\
& ((x = \widehat{x}) \wedge (y = \widehat{y})) \Rightarrow ((y = \widehat{y}) \wedge (x = \widehat{x})) \\
\Leftrightarrow & < \wedge\text{-commutivity; } \Rightarrow\text{-simplification} > \\
& T
\end{array}
$$

**Answer:** An alternative way to prove this is to annotate the code:

| |
|---|
| $\{\ Q: \ (x = \widehat{x}) \wedge (y = \widehat{y}) \ \}$ |
| $\{\ \text{wp}(\text{``}S\text{''}_0, \text{wp}(\text{``}S\text{''}_1, \text{wp}(S_2, R))) = (y = \widehat{y}) \wedge (x = \widehat{x}) \ \}$ |
| $S_0: \ t := x$ |
| $\{\ \text{wp}(\text{``}S\text{''}_1, \text{wp}(S_1, R)) = (y = \widehat{y}) \wedge (t = \widehat{x}) \ \}$ |
| $S_1: \ x := y$ |
| $\{\ \text{wp}(\text{``}S\text{''}_2, R) = (x = \widehat{y}) \wedge (t = \widehat{x}) \ \}$ |
| $S_2: \ y := t$ |
| $\{\ R: \ (x = \widehat{y}) \wedge (y = \widehat{x}) \ \}$ |

after which we are left with proving that $Q \Rightarrow (y = \widehat{y}) \wedge (x = \widehat{x})$, which is trivial.

☛ BACK TO TEXT

**Homework 2.3.5.1** Evaluate

1. $\text{wp}(\text{``}i := i + 1; j := 2i\text{''}, 2i = j)$

   **Answer:**

$$
\begin{aligned}
& \text{wp}(\text{“}i := i+1; j := 2i\text{”}, 2i = j) \\
\Leftrightarrow & < \text{definition of } := > \\
& \text{wp}(\text{“}i := i+1\text{”}, (2i = j)^{j}_{(2i)}) \\
\Leftrightarrow & < \text{definition of } R^{x}_{(\mathcal{E})} > \\
& \text{wp}(\text{“}i := i+1\text{”}, (2i = 2i)) \\
\Leftrightarrow & < \text{algebra} > \\
& \text{wp}(\text{“}i := i+1\text{”}, T) \\
\Leftrightarrow & < \text{definition of } := > \\
& (T)^{i}_{(i+1)} \\
\Leftrightarrow & < \text{definition of } R^{x}_{(\mathcal{E})} > \\
& T
\end{aligned}
$$

or, quicker,

$$
\begin{aligned}
& \text{wp}(\text{“}i := i+1; j := 2i\text{”}, 2i = j) \\
\Leftrightarrow & < \text{definition of } := > \\
& \text{wp}(\text{“}i := i+1\text{”}, (2i = 2i)) \\
\Leftrightarrow & < \text{algebra} > \\
& \text{wp}(\text{“}i := i+1\text{”}, T) \\
\Leftrightarrow & < \text{definition of } := > \\
& T
\end{aligned}
$$

2. $\text{wp}(\text{“}j := 2i; i := i+1\text{”}, 2i = j)$

   **Answer:**

$$
\begin{aligned}
& \mathrm{wp}(\text{“}j := 2i; i := i+1\text{”}, 2i = j) \\
\Leftrightarrow\ & < \text{definition of } := > \\
& \mathrm{wp}(\text{“}j := 2i\text{”}, (2i = j)^{i}_{(i+1)}) \\
\Leftrightarrow\ & < \text{definition of } R^{x}_{(\mathcal{E})} > \\
& \mathrm{wp}(\text{“}j := 2i\text{”}, (2(i+1) = j)) \\
\Leftrightarrow\ & < \text{algebra; definition of } := > \\
& (2i+2 = j)^{j}_{(2i)} \\
\Leftrightarrow\ & < \text{definition of } R^{x}_{(\mathcal{E})} > \\
& 2i+2 = 2i \\
\Leftrightarrow\ & < \text{algebra} > \\
& F
\end{aligned}
$$

or, quicker,

$$
\begin{aligned}
& \mathrm{wp}(\text{“}j := 2i; i := i+1\text{”}, 2i = j) \\
\Leftrightarrow\ & < \text{definition of } := > \\
& \mathrm{wp}(\text{“}j := 2i\text{”}, (2(i+1) = j)) \\
\Leftrightarrow\ & < \text{algebra; definition of } := > \\
& 2i+2 = 2i \\
\Leftrightarrow\ & < \text{algebra} > \\
& F
\end{aligned}
$$

3. $\mathrm{wp}(\text{“}i, j := i+1, 2i\text{”}, 2i = j)$

**Answer:**

$$
\begin{array}{l}
\quad \mathrm{wp}(\text{“}i,j := i+1, 2i\text{”}, 2i = j) \\
\Leftrightarrow \; < \text{definition of} := > \\
\quad (2i = j)^{i,j}_{(i+1),(2i)}) \\
\Leftrightarrow \; < \text{definition of } R^{x}_{(\mathcal{E})} > \\
\quad 2(i+1) = 2i \\
\Leftrightarrow \; < \text{algebra} > \\
\quad 2 = 0 \\
\Leftrightarrow \; < \text{algebra} > \\
\quad F
\end{array}
$$

or, quicker,

$$
\begin{array}{l}
\quad \mathrm{wp}(\text{“}i,j := i+1, 2i\text{”}, 2i = j) \\
\Leftrightarrow \; < \text{definition of} := > \\
\quad (2(i+1) = 2i) \\
\Leftrightarrow \; < \text{algebra} > \\
\quad F
\end{array}
$$

☛ BACK TO TEXT

**Homework 2.3.5.2** Prove the following code segment correct. It swaps the values of variables $x$ and $y$.

$\{Q: \; (x = \widehat{x}) \wedge (y = \widehat{y})\}$
$S: \; x, y := y, x$
$\{R: \; (x = \widehat{y}) \wedge (y = \widehat{x})\}$

**Answer:**

$$
\begin{array}{l}
\quad Q \Rightarrow \mathrm{wp}(\text{“}S\text{”}, R) \\
\Leftrightarrow \; < \text{Instantiate } S \text{ and } R > \\
\quad Q \Rightarrow \mathrm{wp}(\text{“}x, y := y, x'', (x = \widehat{y}) \wedge (y = \widehat{x})) \\
\Leftrightarrow \; < \text{Definition of wp}(:=,) > \\
\quad Q \Rightarrow ((x = \widehat{y}) \wedge (y = \widehat{x}))^{x,y}_{(y),(x)} \\
\Leftrightarrow \; < \text{Instantiate } Q, \text{ definition of } R^{x,y}_{(y),(x)} > \\
\quad ((x = \widehat{x}) \wedge (y = \widehat{y})) \Rightarrow ((y = \widehat{y}) \wedge (x = \widehat{x})) \\
\Leftrightarrow \; < \wedge\text{-commutivity; } \Rightarrow\text{-simplification} > \\
\quad T
\end{array}
$$

☛ BACK TO TEXT

**Homework 2.3.5.3** Evaluate

1. wp(“$i := 2i + j; j := i + 2j + 4$”, $i = j$)

   **Answer:**

$$\begin{aligned}
& \text{wp}(\text{“}i := 2i + j; j := i + 2j + 4\text{”}, i = j) \\
\Leftrightarrow & < \text{definition of} := > \\
& \text{wp}(\text{“}i := 2i + j\text{”}, i = i + 2j + 4) \\
\Leftrightarrow & < \text{definition of} := > \\
& 2i + j = 2i + j + 2j + 4 \\
\Leftrightarrow & < \text{algebra} > \\
& 0 = 2j + 4 \\
\Leftrightarrow & < \text{algebra} > \\
& j = -2
\end{aligned}$$

2. wp(“$j := i + 2j + 4; i := 2i + j$”, $i = j$)

   **Answer:**

$$\begin{aligned}
& \text{wp}(\text{“}j := i + 2j + 4; i := 2i + j\text{”}, i = j) \\
\Leftrightarrow & < \text{definition of} := > \\
& \text{wp}(\text{“}j := i + 2j + 4\text{”}, 2i + j = j) \\
\Leftrightarrow & < \text{definition of} := > \\
& 2i + (i + 2j + 4) = i + 2j + 4 \\
\Leftrightarrow & < \text{algebra} > \\
& 2i = 0 \\
\Leftrightarrow & < \text{algebra} > \\
& i = 0
\end{aligned}$$

3. wp(“$i, j := 2i + j, i + 2j + 4$”, $i = j$)

   **Answer:**

$$\begin{aligned}
& \text{wp}(\text{“}i, j := 2i + j, i + 2j + 4\text{”}, i = j) \\
\Leftrightarrow & < \text{definition of} := > \\
& 2i + j = i + 2j + 4 \\
\Leftrightarrow & < \text{algebra} > \\
& i = j + 4
\end{aligned}$$

☛ BACK TO TEXT

**Homework 2.3.6.1** Prove the correctness of the following code segment. It might be part of a loop that scales the elements of array $b$ by scalar $\alpha \neq 0$. The array $\widehat{b}$ is introduced to refer to the original contents of $b$ and should not be used in actual computation. You may skip checking if the expression being assigned is valid (since they clearly are).

|  |
|---|
| $\{\ Q : (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge (\forall i \mid k \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k < n)\ \}$ |
| $S : b(k) := \alpha \times b(k)$ |
| $\{\ R : (\forall i \mid 0 \le i < k+1 : b(i) = \alpha \times \widehat{b}(i)) \wedge (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n)\ \}$ |

**Answer:**

$$
\begin{array}{ll}
 & Q \Rightarrow \text{wp}(\text{“}S\text{”}, R) \\
\Leftrightarrow & < \text{instantiate } S \text{ and } R > \\
 & Q \Rightarrow \text{wp}(\text{“}b(k) := \alpha \times b(k)\text{”}, \quad (\forall i \mid 0 \le i < k+1 : b(i) = \alpha \times \widehat{b}(i)) \\
 & \qquad\qquad \wedge \ (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \\
\Leftrightarrow & < \text{split range} > \\
 & Q \Rightarrow \text{wp}(\text{“}b(k) := \alpha \times b(k)\text{”}, \quad (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge b(k) = \alpha \times \widehat{b}(k) \\
 & \qquad\qquad \wedge \ (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n)) \\
\Leftrightarrow & < \text{definition of } := (k \text{ is not in range of quantifier}) > \\
 & Q \Rightarrow \quad (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge \alpha \times b(k) = \alpha \times \widehat{b}(k) \\
 & \qquad \wedge \ (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \\
\Leftrightarrow & < \text{algebra} > \\
 & Q \Rightarrow \quad (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \\
 & \qquad \wedge \ b(k) = \widehat{b}(k) \wedge (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \\
\Leftrightarrow & < \text{split range} > \\
 & Q \Rightarrow (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge (\forall i \mid k \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \\
\Leftrightarrow & < \text{instantiate } Q > \\
 & (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge (\forall i \mid k \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k < n) \Rightarrow \\
 & \qquad (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge (\forall i \mid k \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \\
\Leftrightarrow & < \text{weakening/strengthening } (0 \le k < n \text{ is stronger than } 0 \le k \le n) > \\
 & T
\end{array}
$$

Alternative proof: arraycolsep=1.4pt

$$\left\{\begin{array}{l} Q: \ (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge (\forall i \mid k \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k < n) \\ \quad \Leftrightarrow < \text{ split range } > \\ \quad (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge b(k) = \widehat{b}(k) \\ \quad \wedge (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \end{array}\right\}$$

$$\left\{ Q \Rightarrow \text{wp}(\text{“}S\text{”}, R)? \quad \text{YES!} \right\}$$

$$\left\{\begin{array}{l} \text{wp}(\text{“}S\text{”}, R): \ (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge \alpha \times b(k) = \alpha \times \widehat{b}(k) \\ \qquad \wedge (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \\ \qquad \Leftrightarrow < \text{ algebra } > \\ \qquad (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge b(k) = \widehat{b}(k) \\ \qquad \wedge (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \end{array}\right\}$$

$S: b(k) := \alpha \times b(k)$

$$\left\{\begin{array}{l} R: \ (\forall i \mid 0 \le i < k+1 : b(i) = \alpha \times \widehat{b}(i)) \wedge (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \\ \quad \Leftrightarrow < \text{ split range } > \\ \quad (\forall i \mid 0 \le i < k : b(i) = \alpha \times \widehat{b}(i)) \wedge b(k) = \alpha \times \widehat{b}(k) \\ \quad \wedge (\forall i \mid k+1 \le i < n : b(i) = \widehat{b}(i)) \wedge (0 \le k \le n) \end{array}\right\}$$

☛ BACK TO TEXT

**Homework 2.3.6.2** Consider the following code segment that swaps the contents of $b(i)$ and $b(j)$.

$$\left\{ Q: (\forall k \mid 0 \le k < n : b(k) = \widehat{b}(k)) \wedge (0 \le i < n) \wedge (0 \le j < n) \wedge i \ne j \right\}$$

$b(i), b(j) := b(j), b(i)$

$$\left\{\begin{array}{l} \{R: \ (\forall k \mid (0 \le k < n) \wedge (k \ne i) \wedge (k \ne j) : b(k) = \widehat{b}(k)) \\ \qquad \wedge (b(i) = \widehat{b}(j)) \wedge (b(j) = \widehat{b}(i)) \wedge (0 \le i < n) \wedge (0 \le j < n)\} \end{array}\right\}$$

Prove it correct.

**Answer:**

$$
\begin{array}{l}
\quad Q \Rightarrow \mathrm{wp}(\text{“}S\text{”}, R) \\
\Leftrightarrow \ < \text{instantiate } S \text{ and } R > \\
\quad Q \Rightarrow \mathrm{wp}(\text{“}b(i), b(j) := b(j), b(i)\text{”},\ (\forall k \mid (0 \le k < n) \wedge (k \neq i) \wedge (k \neq j) : b(k) = \widehat{b}(k)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge (b(i) = \widehat{b}(j)) \wedge (b(j) = \widehat{b}(i)) \wedge (0 \le i < n) \wedge (0 \le j < n))\} \\
\Leftrightarrow \ < \text{simultaneous assignment to array} > \\
\quad Q \Rightarrow \ (\forall k \mid (0 \le k < n) \wedge (k \neq i) \wedge (k \neq j) : b(k) = \widehat{b}(k)) \\
\qquad\quad \wedge (b(j) = \widehat{b}(j)) \wedge (b(i) = \widehat{b}(i)) \wedge (0 \le i < n) \wedge (0 \le j < n)\} \\
\Leftrightarrow \ < \text{instantiate } Q\text{; split range} > \\
\quad (\forall k \mid 0 \le k < n : b(k) = \widehat{b}(k)) \wedge (0 \le i < n) \wedge 0 \le j < n) \wedge i \neq j \\
\qquad \Rightarrow \ (\forall k \mid (0 \le k < n) : b(k) = \widehat{b}(k)) \wedge (0 \le i < n) \wedge (0 \le j < n)\} \\
\Leftrightarrow \ < \text{weakening/strengthening} > \\
\quad T
\end{array}
$$

☛ BACK TO TEXT

**Homework 2.4.2.1** In the above example, we use an intuitive understanding of the abs() function. We can refine this by recognizing that $z = \mathrm{abs}(x)$ is equivalent to $(x \ge 0 \wedge z = x) \vee (x \le 0 \wedge z = -x)$ so that the code segment becomes

| |
|---|
| $\{\ T \qquad\qquad\qquad\qquad\qquad\qquad \}$ |
| **if** |
| $\quad x \ge 0 \rightarrow z := x$ |
| $\quad x \le 0 \rightarrow z := -x$ |
| **fi** |
| $\left\{\ (x \ge 0 \wedge z = x) \vee (x \le 0 \wedge z = -x) \qquad\qquad \right\}$ |

Prove this code segment correct.

**Answer:**

$$
\begin{array}{ll}
 & T \Rightarrow \text{wp}(\text{“}\textbf{if}\text{”}, R) \\
\Leftrightarrow & < \Rightarrow\text{-simplification} > \\
 & \text{wp}(\text{“}\textbf{if}\text{”}, R) \\
\Leftrightarrow & < \text{definition of wp(“}\textbf{if}\text{”, )} > \\
 & (x \geq 0 \vee x \leq 0) \;\wedge\; (x \geq 0 \Rightarrow \text{wp}(\text{“}z := x\text{”}, (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x))) \\
 & \qquad\qquad\qquad\;\; \wedge\; (x \leq 0 \Rightarrow \text{wp}(\text{“}z := -x\text{”}, (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x))) \\
\Leftrightarrow & < \text{definition of} := > \\
 & (x \geq 0 \vee x \leq 0) \;\wedge\; (x \geq 0 \Rightarrow (x \geq 0 \wedge x = x) \vee (x \leq 0 \wedge x = -x)) \\
 & \qquad\qquad\qquad\;\; \wedge\; (x \leq 0 \Rightarrow (x \geq 0 \wedge -x = x) \vee (x \leq 0 \wedge -x = -x)) \\
\Leftrightarrow & < \text{algebra} \times 5 > \\
 & T \wedge (x \geq 0 \Rightarrow (x \geq 0 \wedge T) \vee (x \leq 0 \wedge x = 0)) \wedge (x \leq 0 \Rightarrow (x \geq 0 \wedge x = 0) \vee (x \leq 0 \wedge T)) \\
\Leftrightarrow & < \text{algebra} \times 2\text{; } \wedge\text{-simplification} \times 3 > \\
 & (x \geq 0 \Rightarrow x \geq 0 \vee x = 0) \wedge (x \leq 0 \Rightarrow x = 0 \vee x \leq 0) \\
\Leftrightarrow & < \text{weakening/strengthening; commutativity; weakening/strengthening} > \\
 & T \wedge T \\
\Leftrightarrow & < \wedge\text{-simplification} > \\
 & T
\end{array}
$$

☛ BACK TO TEXT

**Homework 2.4.3.1** Complete the proof of the correctness of

| |
|---|
| $\{\; T \;\}$ |
| **if** |
| $\quad x \geq 0 \rightarrow z := x$ |
| $\quad x \leq 0 \rightarrow z := -x$ |
| **fi** |
| $\{\; (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x) \;\}$ |

from the last example.

**Answer:**

$Q \wedge G_1 \Rightarrow \text{wp}(\text{“}S_1\text{”}, R)$:

$$\begin{array}{ll} & Q \wedge G_1 \Rightarrow \text{wp}(\text{“}S_1\text{”}, R) \\ \Leftrightarrow & < \text{instantiate} > \\ & T \wedge x \leq 0 \Rightarrow \text{wp}(\text{“}z := -x\text{”}, (x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x)) \\ \Leftrightarrow & < \wedge\text{-simplification, definition of} := > \\ & x \leq 0 \Rightarrow (x \geq 0 \wedge -x = x) \vee (x \leq 0 \wedge -x = -x) \\ \Leftrightarrow & < \text{algebra } \times 2 > \\ & x \leq 0 \Rightarrow (x \geq 0 \wedge x = 0) \vee (x \leq 0 \wedge T) \\ \Leftrightarrow & < \wedge\text{-simplification; algebra} > \\ & x \leq 0 \Rightarrow x = 0 \vee x \leq 0 \\ \Leftrightarrow & < \text{weakening/strengthening} > \\ & T \end{array}$$

☛ BACK TO TEXT

**Homework 2.4.3.2** The following code segment sets $m$ to the maximum of $x$ and $y$. Use the If Theorem to prove it correct.

```
{Q : T}
if
   x ≥ y  →  m := x
[] x ≤ y  →  m := y
fi
{R : (x ≥ y ∧ m = x) ∨ (x ≤ y ∧ m = y)}
```

**Answer:**

- $Q \Rightarrow G_0 \vee \cdots \vee G_{k-1}$:

$$\begin{array}{ll} & Q \Rightarrow G_0 \vee G_1 \\ \Leftrightarrow & < \text{instantiate} > \\ & T \Rightarrow x \geq y \vee x \leq y \\ \Leftrightarrow & < \Rightarrow\text{-simplification; algebra} > \\ & T \end{array}$$

- $Q \wedge G_0 \Rightarrow \text{wp}(S_0, R)$:

$$\begin{aligned}
& Q \wedge G_0 \Rightarrow \text{wp}(S_0, R) \\
\Leftrightarrow\ & < \text{instantiate} > \\
& T \wedge x \geq y \Rightarrow \text{wp}(\text{“}m := x\text{”}, (x \geq y \wedge m = x) \vee (x \leq y \wedge m = y)) \\
\Leftrightarrow\ & < \wedge\text{-simplification, definition of } := > \\
& x \geq y \Rightarrow (x \geq y \wedge x = x) \vee (x \leq y \wedge x = y) \\
\Leftrightarrow\ & < \text{algebra } \times 2 > \\
& x \geq y \Rightarrow (x \geq y \wedge T) \vee x = y \\
\Leftrightarrow\ & < \wedge\text{-simplification} > \\
& x \geq y \Rightarrow x \geq y \vee x = y \\
\Leftrightarrow\ & < \text{weakening/strengthening} > \\
& T
\end{aligned}$$

- $Q \wedge G_1 \Rightarrow \text{wp}(\text{“}S_1\text{”}, R)$:

$$
\begin{array}{ll}
& Q \wedge G_1 \Rightarrow \text{wp}(\text{“}S_1\text{”}, R) \\
\Leftrightarrow & < \text{instantiate} > \\
& T \wedge x \leq y \Rightarrow \text{wp}(\text{“}m := y\text{”}, (x \geq y \wedge m = x) \vee (x \leq y \wedge m = y)) \\
\Leftrightarrow & < \wedge\text{-simplification, definition of } := > \\
& x \leq y \Rightarrow (x \geq y \wedge y = x) \vee (x \leq y \wedge y = y) \\
\Leftrightarrow & < \text{algebra } \times 2 > \\
& x \leq y \Rightarrow x = y \vee (x \leq y \wedge T) \\
\Leftrightarrow & < \wedge\text{-simplification} > \\
& x \leq y \Rightarrow x = y \vee x \leq y \\
\Leftrightarrow & < \text{weakening/strengthening} > \\
& T
\end{array}
$$

☛ BACK TO TEXT

**Homework 2.4.3.3** The following code segment might be part of a loop that computes $m$, the minimum value in array $b$:

```
{Q : (∀j | 1 ≤ j < i : m ≤ b(j)) ∧ 0 ≤ i < n}
if
  b(i) ≥ m  →  skip
▯ b(i) ≤ m  →  m := b(i)
fi
i := i + 1
{R : (∀j | 1 ≤ j < i : m ≤ b(j)) ∧ 0 ≤ i ≤ n}
```

Prove it correct. **Answer:**

First, let's bring it down to only having to prove the **if** command correct:

```
{Q : (∀j | 1 ≤ j < i : m ≤ b(j)) ∧ 0 ≤ i < n}
if
  b(i) ≥ m  →  skip
▯ b(i) ≤ m  →  m := b(i)
fi
{R′ : (∀j | 1 ≤ j < i + 1 : m ≤ b(j)) ∧ 0 ≤ i + 1 ≤ n}
i := i + 1
{R : (∀j | 1 ≤ j < i : m ≤ b(j)) ∧ 0 ≤ i ≤ n}
```

(This is a matter of substituting $i+1$ for $i$.)

Now, to check the correctness of the **if** command, we check

- $Q \Rightarrow G_0 \vee \cdots \vee G_{K-1}$:

$Q \Rightarrow (G_0 \vee G_1)$

$\Leftrightarrow <$ instantiate $>$

$Q \Rightarrow (b(i) \geq m \vee b(i) \leq m)$

$\Leftrightarrow <$ algebra $>$

$Q \Rightarrow T$

$\Leftrightarrow < \Rightarrow$-simplification $>$

$T$

(Notice: we did not instantiate $Q$ initially, and found out that it needed not be instantiated. This saves a lot of writing of long expressions.)

- $Q \wedge G_0 \Rightarrow \mathrm{wp}(S_0, R')$:

$Q \wedge G_0 \Rightarrow \mathrm{wp}(S_0, R')$

$\Leftrightarrow <$ instantiate $>$

$Q \wedge G_0 \Rightarrow \mathrm{wp}(\mathbf{skip}, (\forall j \mid 1 \leq j < i+1 : m \leq b(j)) \wedge 0 \leq i+1 \leq n)$

$\Leftrightarrow <$ definition **skip**; algebra $>$

$Q \wedge G_0 \Rightarrow (\forall j \mid 1 \leq j < i+1 : m \leq b(j)) \wedge -1 \leq i < n$

$\Leftrightarrow <$ instantiate $>$

$(\forall j \mid 1 \leq j < i : m \leq b(j)) \wedge 0 \leq i < n \wedge b(i) \geq m \Rightarrow (\forall j \mid 1 \leq j < i+1 : m \leq b(j)) \wedge -1 \leq i < n$

$\Leftrightarrow <$ algebra; split range $>$

$(\forall j \mid 1 \leq j < i+1 : m \leq b(j)) \wedge 0 \leq i < n \Rightarrow (\forall j \mid 1 \leq j < i+1 : m \leq b(j)) \wedge -1 \leq i < n$

$\Leftrightarrow <$ weakening strengthening $(0 \leq i < n \Rightarrow -1 \leq i < n) >$ $T$

(Notice how we delayed instantiation. In the last step, we use the fact that $(\mathrm{E2} \Rightarrow \mathrm{E3}) \Rightarrow (\mathrm{E1} \wedge \mathrm{E2} \Rightarrow \mathrm{E1} \wedge \mathrm{E3})$.)

- $Q \wedge G_1 \Rightarrow \text{wp}(\text{“}S_1\text{”}, R')$:

$$\begin{aligned}
& Q \wedge G_1 \Rightarrow \text{wp}(\text{“}S_1\text{”}, R') \\
\Leftrightarrow\ & < \text{instantiate} > \\
& Q \wedge G_1 \Rightarrow \text{wp}(m := b(i), (\forall j \mid 1 \le j < i+1 : m \le b(j)) \wedge 0 \le i+1 \le n) \\
\Leftrightarrow\ & < \text{definition wp}(:=,); \text{ algebra} > \\
& Q \wedge G_1 \Rightarrow (\forall j \mid 1 \le j < i+1 : b(i) \le b(j)) \wedge 0 \le i+1 \le n \\
\Leftrightarrow\ & < \text{instantiate} > \\
& (\forall j \mid 1 \le j < i : m \le b(j)) \wedge 0 \le i < n \wedge b(i) \le m \\
& \quad \Rightarrow (\forall j \mid 1 \le j < i+1 : b(i) \le b(j)) \wedge 0 \le i+1 \le n \\
\Leftrightarrow\ & < \text{algebra} > \\
& (\forall j \mid 1 \le j < i : b(i) \le b(j)) \wedge (\forall j \mid 1 \le j < i : m \le b(j)) \wedge 0 \le i < n \wedge b(i) \le m \\
& \quad \Rightarrow (\forall j \mid 1 \le j < i+1 : b(i) \le b(j)) \wedge 0 \le i+1 \le n \\
\Leftrightarrow\ & < \text{more algebra} > \\
& (\forall j \mid 1 \le j < i : b(i) \le b(j)) \wedge b(i) \le b(i) \wedge (\forall j \mid 1 \le j < i : m \le b(j)) \wedge 0 \le i < n \wedge b(i) \le m \\
& \quad \Rightarrow (\forall j \mid 1 \le j < i+1 : b(i) \le b(j)) \wedge 0 \le i+1 \le n \\
\Leftrightarrow\ & < \text{split range} > \\
& (\forall j \mid 1 \le j < i+1 : b(i) \le b(j)) \wedge (\forall j \mid 1 \le j < i : m \le b(j)) \wedge 0 \le i < n \wedge b(i) \le m \\
& \quad \Rightarrow (\forall j \mid 1 \le j < i+1 : b(i) \le b(j)) \wedge 0 \le i+1 \le n \\
\Leftrightarrow\ & < \text{weakening strengthening } (0 \le i < n \Rightarrow -1 \le i < n) > \\
& T
\end{aligned}$$

(Notice how we delayed instantiation. In the last step, we again use the fact that (E2 $\Rightarrow$ E3) $\Rightarrow$ (E1 $\wedge$ E2 $\Rightarrow$ E1 $\wedge$ E3).)

☛ BACK TO TEXT

**Homework 2.4.3.4** Prove the following code segment correct:

```
{$(\forall j | 0 \le j < i : m \ge b(j))$}
if
  $b(i) \le m$  $\rightarrow$  skip
▯ $b(i) \ge m$  $\rightarrow$  $m := b(i)$
fi
$i := i+1$
{$(\forall j | 0 \le j < i : m \ge b(j))$}
```

☛ BACK TO TEXT

**Homework 2.4.5.1** The following code segment sets $m$ to the maximum of $x$ and $y$ with an if-then-else command. Use Figure 2.5 to prove it correct.

```
{$Q : T$}
if $x \ge y$
```

```
  $m := x$
else
  $m := y$
fi
$\{R : (x \geq y \wedge m = x) \vee (\neg(x \geq y) \wedge m = y)\}$
```

**Answer:**

```
$\{ \; Q : T \; \}$
if $x \geq y$
    $\{ G \wedge Q : x \geq y \wedge T \}$
    $\{ G \wedge Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, R)$? YES! $\}$
    $\{ \text{wp}(\text{“}S_0\text{”}, R) : (x \geq y \wedge x = x) \vee (x < y \wedge x = y) \}$
    $S_0 : m := x$
    $\{ R : (x \geq y \wedge m = x) \vee (x < y \wedge m = y) \}$
else
    $\{ \neg G \wedge Q : \neg(x \geq y) \}$
    $\{ \neg G \wedge Q \Rightarrow \text{wp}(\text{“}S_1\text{”}, R)$? YES! $\}$
    $\{ \text{wp}(\text{“}S_1\text{”}, R) : (x \geq y \wedge y = x) \vee (x < y \wedge y = y) \}$
    $S_1 : m := y$
    $\{ R : (x \geq y \wedge m = x) \vee (x < y \wedge m = y) \}$
fi
$\{ \; R : (x \geq y \wedge m = x) \vee (\neg(x \geq y) \wedge m = y) \; \}$
```

- $Q \Rightarrow G \vee \neg G$: trivially true. Indeed, this never needs to be checked for an **if-then-else** command.

- $Q \wedge G \Rightarrow \text{wp}(S_0, R)$:

$Q \wedge G \Rightarrow \text{wp}(S_0, R)$

$\Leftrightarrow <$ instantiate $>$

$T \wedge x \geq y \Rightarrow \text{wp}(\text{“}m := x\text{”}, (x \geq y \wedge m = x) \vee (x < y \wedge m = y))$

$\Leftrightarrow <$ $\wedge$-simplification, definition of $:=$ $>$

$x \geq y \Rightarrow (x \geq y \wedge x = x) \vee (x < y \wedge x = y)$

$\Leftrightarrow <$ algebra $\times 2$ $>$

$x \geq y \Rightarrow (x \geq y \wedge T) \vee F$

$\Leftrightarrow <$ $\wedge$-simplification; $\vee$-simplification $>$

$x \geq y \Rightarrow x \geq y$

$\Leftrightarrow <$ $\Rightarrow$-simplification $>$

$T$

- $Q \wedge \neg G \Rightarrow \text{wp}(S_1, R)$:

$$\begin{array}{l} \quad Q \wedge \neg G \Rightarrow \text{wp}(S_1, R) \\ \Leftrightarrow \ < \text{instantiate} > \\ \quad T \wedge x < y \Rightarrow \text{wp}(\text{“}m := y\text{”}, (x \geq y \wedge m = x) \vee (x < y \wedge m = y)) \\ \Leftrightarrow \ < \wedge\text{-simplification, definition of } := > \\ \quad x < y \Rightarrow (x \geq y \wedge y = x) \vee (x < y \wedge y = y) \\ \Leftrightarrow \ < \text{algebra } \times 2 > \\ \quad x < y \Rightarrow x = y \vee (x < y \wedge T) \\ \Leftrightarrow \ < \wedge\text{-simplification} > \\ \quad x < y \Rightarrow x = y \vee x < y \\ \Leftrightarrow \ < \text{weakening/strengthening} > \\ \quad T \end{array}$$

☛ BACK TO TEXT

**Homework 2.5.3.1** Prove the partial correctness of the following code segment that adds the elements in $a(0:n-1)$ to $b(0:n-1)$ storing the result in $c(0:n-1)$ (assuming the sizes of all three arrays equal at least $n$):

```
{ {Q : 0 ≤ n} }
S_I : k := 0
while k < n do
    S : c(k) := a(k) + b(k); k := k + 1
endwhile
{ (R : ∀i | 0 ≤ i < n : c(i) = a(i) + b(i)) }
```

Use loop invariant $P_{\text{inv}} : (\forall i \mid 0 \leq i < k : c(i) = a(i) + b(i)) \wedge 0 \leq k \leq n$.

☛ BACK TO TEXT

**Homework 2.5.4.1** In Homework 2.5.3.1 you proved the partial correctness of the following code segment that adds the elements in $a(0:n-1)$ to $b(0:n-1)$ storing the result in $c(0:n-1)$ (assuming the sizes of all three arrays equal at least $n$).

```
{ {Q : 0 ≤ n} }
S_I : k := 0
{ {P_inv : (∀i | 0 ≤ i < k : c(i) = a(i) + b(i)) ∧ 0 ≤ k ≤ n} }
{ {t : n − k} }
while k < n do
    S : c(k) := a(k) + b(k); k := k + 1
endwhile
{ (R : ∀i | 0 ≤ i < n : c(i) = a(i) + b(i)) }
```

Prove in addition the total correctness of this code segment.

☛ BACK TO TEXT

**Homework 2.6.2.1** Prove that $((p \Rightarrow r) \wedge (q \Rightarrow r)) \Leftrightarrow ((p \vee q) \Rightarrow r)$.

**Answer:**

$$
\begin{array}{ll}
 & (p \Rightarrow r) \wedge (q \Rightarrow r) \\
\Leftrightarrow & < \text{Implication} > \\
 & (\neg p \vee r) \wedge (\neg q \vee r) \\
\Leftrightarrow & < \text{Distributivity of} > \\
 & (\neg p \wedge \neg q) \vee r \\
\Leftrightarrow & < \text{De Morgan's} > \\
 & \neg(p \vee q) \vee r \\
\Leftrightarrow & < \text{Implication} > \\
 & p \vee q \Rightarrow r
\end{array}
$$

☛ BACK TO TEXT

**Homework 2.6.2.2** Prove that the **skip** command satisfies the Laws of Excluded Miracle, Distributivity of Conjunction, Monotonicity, and Distributed Disjunction.

**Answer:**

**Excluded Miracle:** Since wp(“**skip**”, $R$) $= R$ clearly wp(**skip**, $F$) $= F$.

**Distributivity of Conjunction:** wp(“**skip**”$Q$) $\wedge$ $wp$(“**skip**”$R$) $= Q \wedge R =$ wp(“**skip**”$Q \wedge R$).

**Monotonicity:** Assume $Q \Rightarrow R$. Since wp(“**skip**”$Q$) $= Q$ and wp(“**skip**”$R$) $= R$ clearly wp(“**skip**”$Q$) $\Rightarrow$ $wp$(“**skip**”$R$).

**Distributivity of Disjunction:** wp(“**skip**”$Q$) $\vee$ wp(“**skip**”$R$) $= Q \vee R =$ wp(“**skip**”$Q \vee R$). Hence wp(“**skip**”$Q$) $\vee$ wp(“**skip**”$R$) $\Rightarrow$ wp(“**skip**”$Q \vee R$)

☛ BACK TO TEXT

**Homework 2.6.2.3** Prove that the **abort** command satisfies the Laws of Excluded Miracle, Distributivity of Conjunction, Monotonicity, and Distributed Disjunction.

**Answer:**

**Excluded Miracle:** Since wp(“**abort**”$R$) $= F$ clearly wp(“**abort**”$F$) $= F$.

**Distributivity of Conjunction:** wp(“**abort**”$Q$) $\wedge$ $wp$(“**abort**”$R$) $= F \wedge F = F =$ wp(“**abort**”$Q \wedge R$).

**Monotonicity:** Assume $Q \Rightarrow R$. Since wp(“**abort**”$Q$) $= F$ clearly wp(“**abort**”$Q$) $\Rightarrow$ $wp$(“**abort**”$R$) (independent of the fact that $Q \Rightarrow R$ for this command).

**Distributivity of Disjunction:** wp(“**abort**”$Q$) $\vee$ wp(“**abort**”$R$) $= F \vee F = F$. Hence wp(“**abort**”$Q$) $\vee$ wp(“**abort**”$R$) $\Rightarrow$ wp(“**abort**”$Q \vee R$).

☛ BACK TO TEXT

**Homework 2.6.2.4** Prove that the composition of commands satisfies the Laws of Excluded Miracle, Distributivity of Conjunction, Monotonicity, and Distributed Disjunction.

**Answer:**

**Excluded Miracle:** $\text{wp}(\text{``}S_0;S_1\text{''},F) = \text{wp}(\text{``}S_0\text{''},\text{wp}(\text{``}S_1\text{''},F))$
$= \text{wp}(\text{``}S_0\text{''},F) = F.$

**Distributivity of Conjunction:** $\text{wp}(\text{``}S_0;S_1\text{''},Q) \wedge wp(\text{``}S_0;S_1\text{''},R)$
$= \text{wp}(\text{``}S_0\text{''},\text{wp}(\text{``}S_1\text{''},Q)) \wedge \text{wp}(\text{``}S_0\text{''},\text{wp}(\text{``}S_1\text{''},R))$
$= \text{wp}(\text{``}S_0\text{''},\text{wp}(\text{``}S_1\text{''},Q \wedge R)) = \text{wp}(\text{``}S_0;S_1\text{''},Q \wedge R).$

**Monotonicity:** Assume $Q \Rightarrow R$. Then $\text{wp}(\text{``}S_0;S_1\text{''},Q) = \text{wp}(\text{````}S_0\text{''},\text{wp}(\text{``}S_1\text{''},Q)) \Rightarrow \text{wp}(\text{``}S_0\text{''},\text{wp}(\text{``}S_1\text{''},R)) = \text{wp}(\text{``}S_0;S_1\text{''},R).$

**Distributivity of Disjunction:** $\text{wp}(\text{``}S_0;S_1\text{''},Q) \vee \text{wp}(\text{``}S_0;S_1\text{''},R)$
$= \text{wp}(\text{``}S_0\text{''},\text{wp}(\text{``}S_1\text{''},Q)) \vee \text{wp}(\text{``}S_0\text{''},\text{wp}(\text{``}S_1\text{''},R)) \Rightarrow \text{wp}(\text{``}S_0\text{''},\text{wp}(\text{``}S_1\text{''},Q) \vee \text{wp}(\text{``}S_1\text{''},R))$
$\Rightarrow \text{wp}(\text{``}S_0\text{''},\text{wp}(\text{``}S_1\text{''},Q \vee R))$
$= \text{wp}(\text{``}S_0;S_1\text{''},Q \vee R).$

☛ BACK TO TEXT

**Homework 2.6.2.5** Prove that the **if** statement

```
if
   $G_0 \rightarrow S_0$
[] $G_1 \rightarrow S_1$
fi
```

satisfies the Laws of Excluded Micacle, Distributivity of Conjunction, Monotonicity, and Distributed Disjunction.

**Answer:**

**Excluded Miracle:**

$$\begin{array}{ll}
 & \text{wp}(\text{``}\textbf{if}\text{''},F) \\
\Leftrightarrow & < \text{Def. of wp}(\text{``}\textbf{if}\text{''},Q) > \\
 & (G_0 \vee G_1) \wedge (G_0 \Rightarrow \text{wp}(\text{``}S_0\text{''},F)) \wedge (G_1 \Rightarrow \text{wp}(\text{``}S_1\text{''},F)) \\
\Leftrightarrow & < \text{Excluded Miracle} > \\
 & (G_0 \vee G_1) \wedge (G_0 \Rightarrow F) \wedge (G_1 \Rightarrow F) \\
\Leftrightarrow & < \text{Implication} > \\
 & (G_0 \vee G_1) \wedge (\neg G_0 \vee F) \wedge (\neg G_1 \vee F) \\
\Leftrightarrow & < \vee\text{-simplification} > \\
 & (G_0 \vee G_1) \wedge \neg G_0 \wedge \neg G_1 \\
\Leftrightarrow & < \text{DeMorgan's Law} > \\
 & (G_0 \vee G_1) \wedge \neg(G_0 \vee G_1) \\
\Leftrightarrow & < \text{contradiction} > \\
 & F
\end{array}$$

**Distributivity of Conjunction:**

$$
\begin{array}{ll}
& \mathrm{wp}(\text{``}\mathbf{if}\text{''}, Q) \wedge \mathrm{wp}(\text{``}\mathbf{if}\text{''}, R) \\
\Leftrightarrow & < \text{definition of } \mathrm{wp}(\text{``}\mathbf{if}\text{''},) > \\
& (G_0 \vee G_1) \wedge (G_0 \Rightarrow \mathrm{wp}(\text{``}S_0\text{''}, Q)) \\
& \wedge (G_1 \Rightarrow \mathrm{wp}(\text{``}S_1\text{''}, Q)) \wedge (G_0 \vee G_1) \wedge (G_0 \Rightarrow \mathrm{wp}(\text{``}S_0\text{''}, R)) \wedge (G_1 \Rightarrow \mathrm{wp}(\text{``}S_1\text{''}, R)) \\
\Leftrightarrow & < \text{commutativity; } \wedge\text{-simplification; implication} > \\
& (G_0 \vee G_1) \wedge (\neg G_0 \vee \mathrm{wp}(\text{``}S_0\text{''}, Q)) \wedge (\neg G_0 \vee \mathrm{wp}(\text{``}S_0\text{''}, R)) \wedge (\neg G_1 \vee \mathrm{wp}(\text{``}S_1\text{''}, Q)) \wedge (\neg G_1 \vee \mathrm{wp}(\text{``}S_1\text{''}, R)) \\
\Leftrightarrow & < \vee\text{-distributivity} > \\
& (G_0 \vee G_1) \wedge (\neg G_0 \vee (\mathrm{wp}(\text{``}S_0\text{''}, Q) \wedge \mathrm{wp}(\text{``}S_0\text{''}, R))) \wedge (\neg G_1 \vee (\mathrm{wp}(\text{``}S_1\text{''}, Q) \wedge \mathrm{wp}(\text{``}S_1\text{''}, R))) \\
\Leftrightarrow & < \text{distributivity of conjunction} > \\
& (G_0 \vee G_1) \wedge (\neg G_0 \vee \mathrm{wp}(\text{``}S_0\text{''}, Q \wedge R)) \wedge (\neg G_1 \vee \mathrm{wp}(\text{``}S_1\text{''}, Q \wedge R)) \\
\Leftrightarrow & < \text{implication} > \\
& (G_0 \vee G_1) \wedge (G_0 \Rightarrow \mathrm{wp}(\text{``}S_0\text{''}, Q \wedge R)) \wedge (G_1 \Rightarrow \mathrm{wp}(\text{``}S_1\text{''}, Q \wedge R)) \\
\Leftrightarrow & < \text{definition of } \mathrm{wp}(\text{``}\mathbf{if}\text{''},) > \\
& \mathrm{wp}(\text{``}\mathbf{if}\text{''}, Q \wedge R)
\end{array}
$$

**Monotonicity:** (Proof provided by course participant Rhitik Bhatt. Not a strict equivalence style proof, but it is how Maggie and I would have done it too.)

To prove: $(Q \Rightarrow R) \Rightarrow (wp(\text{``}\mathbf{if}\text{''}, Q) \Rightarrow wp(\text{``}\mathbf{if}\text{''}, R))$. Let's assume $Q \Rightarrow R$ is true. Then we must prove that

$$(wp(\text{``}\mathbf{if}\text{''}, Q) \Rightarrow wp(\text{``}\mathbf{if}\text{''}, R))$$

is TRUE:

$$
\begin{array}{ll}
& \mathrm{wp}(\text{``}\mathbf{if}\text{''}, Q) \\
\Leftrightarrow & < \text{definition of } \mathrm{wp}(\text{``}\mathbf{if}\text{''}) > \\
& (G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(S_0, Q)) \wedge (G_1 \Rightarrow wp(S_1, Q)) \\
\Rightarrow & < \ \mathrm{wp}(S_0, Q) \Rightarrow wp(S_0, R), wp(S_1, Q) \Rightarrow wp(S_1, R) \quad > \\
& (G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(S_0, Q) \Rightarrow wp(S_0, R)) \wedge (G_1 \Rightarrow wp(S_1, Q) \Rightarrow wp(S_1, R)) \\
\\
Rightarrow & < \quad (E1 \Rightarrow E2 \Rightarrow E3) \Rightarrow (E1 \Rightarrow E3) \quad > \\
& (G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(S_0, R)) \wedge (G_1 \Rightarrow wp(S_1, R)) \\
\Leftrightarrow & < \text{definition of } \mathrm{wp}(\text{``}\mathbf{if}\text{''}) > \\
& \mathrm{wp}(\text{``}\mathbf{if}\text{''}, R)
\end{array}
$$

**Distributivity of Disjunction:** (Proof provided by course participant Rhitik Bhatt. Not a strict equivalence style proof, but it is how Maggie and I would have done it too.)

To Prove : wp(“**if**”, $Q) \vee wp($“**if**”, $R)) \Rightarrow wp($“**if**”, $Q \vee R)$.

$$
\begin{array}{l}
\quad \text{wp(“\textbf{if}”}, Q) \vee wp(\text{“\textbf{if}”}, R) \\
\Leftrightarrow < \text{Definition of wp(“\textbf{if}”)} > \\
\quad\ ((G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, Q)) \wedge (G_1 \Rightarrow wp(\text{“}S_1\text{”}, Q))) \\
\quad \vee ((G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, R)) \wedge (G_1 \Rightarrow wp(\text{“}S_1\text{”}, R))) \\
\Leftrightarrow < \vee\text{-distributivity} > \\
\quad ((G_0 \vee G_1) \vee ((G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, R)) \wedge (G_1 \Rightarrow wp(\text{“}S_1\text{”}, R))) \\
\quad \wedge ((G_0 \Rightarrow wp(\text{“}S_0\text{”}, Q)) \vee ((G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, R)) \wedge (G_1 \Rightarrow wp(\text{“}S_1\text{”}, R)))) \\
\quad \wedge ((G_1 \Rightarrow wp(\text{“}S_1\text{”}, Q)) \vee ((G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, R)) \wedge (G_1 \Rightarrow wp(\text{“}S_1\text{”}, R)))) \\
\Leftrightarrow < \vee\text{-simplification, } \vee\text{-distributivity} > \\
\quad (G_0 \vee G_1) \\
\quad \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, Q) \vee (G_0 \vee G_1)) \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, Q) \vee G_0 \Rightarrow wp(\text{“}S_0\text{”}, R)) \\
\qquad\qquad\qquad\qquad\qquad \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, Q) \vee G_1 \Rightarrow wp(\text{“}S_1\text{”}, R) \\
\quad \wedge (G_1 \Rightarrow wp(\text{“}S_1\text{”}, Q) \vee (G_0 \vee G_1)) \wedge (G_1 \Rightarrow wp(\text{“}S_1\text{”}, Q) \vee G_0 \Rightarrow wp(\text{“}S_0\text{”}, R)) \\
\qquad\qquad\qquad\qquad\qquad \wedge (G_1 \Rightarrow wp(\text{“}S_1\text{”}, Q) \vee G_1 \Rightarrow wp(\text{“}S_1\text{”}, R) \\
\quad (G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, Q) \vee G_0 \Rightarrow wp(\text{“}S_0\text{”}, R)) \wedge ((G_1 \Rightarrow wp(\text{“}S_1\text{”}, Q)) \vee (G_1 \Rightarrow wp(\text{“}S_1\text{”}, R))) \\
\Leftrightarrow < ((A \Rightarrow B) \vee (A \Rightarrow C)) \Leftrightarrow A \Rightarrow B \vee C > \\
\quad (G_0 \vee G_1) \wedge (G_0 \Rightarrow wp(\text{“}S_0\text{”}, Q) \vee wp(\text{“}S_0\text{”}, R)) \wedge ((G_1 \Rightarrow wp(\text{“}S_1\text{”}, Q)) \vee wp(\text{“}S_1\text{”}, R))) \\
\quad (G_0 \vee G_1) \wedge ((G_0 \Rightarrow wp(\text{“}S\text{”}, Q \vee R)) \wedge ((G_1 \Rightarrow wp(\text{“}S\text{”}, Q \vee R)) \\
\Leftrightarrow < \text{definition of wp(if)} > \\
\quad \text{wp(“\textbf{if}”}, Q \vee R)
\end{array}
$$

☛ BACK TO TEXT

**Homework 2.7.1.1** Consider an array $b$ of size $n$ with $0 \leq n$, a scalar variable $s$, and the code segment

| |
|---|
| $\{\ Q:\ (s = (\sum i \mid 0 \leq i < k : b(i))) \wedge (0 \leq k < n)\ \}$ |
| $S_0:\ s := s + b(k)$ |
| $S_1:\ k := k + 1$ |
| $\{\ R:\ (s = (\sum i \mid 0 \leq i < k : b(i))) \wedge (0 \leq k \leq n)\ \}$ |

which may be part of a program that sums the entries in array $b$. Prove this code segment correct.

**Answer:**

$Q \Rightarrow \mathrm{wp}(\text{“}S_0; S_1'', R)$

$\Leftrightarrow <$ Composition of commands $>$

$Q \Rightarrow \mathrm{wp}(\text{“}S_0\text{”}, \mathrm{wp}(\text{“}S_1\text{”}, R))$

$\Leftrightarrow <$ Instantiate $S_1$ $>$

$Q \Rightarrow \mathrm{wp}(\text{“}S_0\text{”}, \mathrm{wp}(k := k+1, R))$

$\Leftrightarrow <$ Instantiate $R$; def. of $\mathrm{wp}(:=, R)$ $>$

$Q \Rightarrow \mathrm{wp}(\text{“}S_0\text{”}, ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (0 \le k \le n)))^{k}_{(k+1)})$

$\Leftrightarrow <$ Instantiate $S_0$; def. of $R^{k}_{(\mathcal{E})}$ $>$

$Q \Rightarrow \mathrm{wp}(s := s + b(k), ((s = (\sum i \mid 0 \le i < k+1 : b(i)) \wedge (0 \le k+1 \le n)))$

$\Leftrightarrow <$ def. of $\mathrm{wp}(:=, R)$ $>$

$Q \Rightarrow ((s = (\sum i \mid 0 \le i < k+1 : b(i)) \wedge (0 \le k+1 \le n)))^{s}_{(s+b(k))}$

$\Leftrightarrow <$ def. of $R^{s}_{(\mathcal{E})}$ $>$

$Q \Rightarrow ((s + b(k) = (\sum i \mid 0 \le i < k+1 : b(i)) \wedge (0 \le k+1 \le n)))$

$\Leftrightarrow <$ Split range $>$

$Q \Rightarrow ((s + b(k) = (\sum i \mid 0 \le i < k : b(i)) + b(k)) \wedge (0 \le k+1 \le n))$

$\Leftrightarrow <$ Instantiate $Q$; algebra; algebra $>$

$((s = (\sum i \mid 0 \le i < k : b(i))) \wedge (0 \le k < n)) \Rightarrow ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (-1 \le k < n)))$

$\Leftrightarrow <$ algebra $>$

$((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (0 \le k < n)))$

$\quad \Rightarrow ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge ((0 \le k < n) \vee (-1 = k))))$

$\Leftrightarrow <$ Weakening/strengthening $>$

$T$

☛ BACK TO TEXT

**Homework 2.7.1.2** Consider an array $b$ with $n$ elements $(0 \le n)$, a scalar variable $s$, and the code segment

$\{Q: \ (s = (\sum i \mid 0 \le i < k : b(i)) \wedge (0 \le k < n))\}$
$S: \ s, k := s + b(k), k+1$
$\{R: \ (s = (\sum i \mid 0 \le i < k : b(i)) \wedge (0 \le k \le n))\}$

This code segment may be part of a program that sums the entries in array $b$. Prove this code segment correct.

**Answer:**

$$
\begin{array}{l}
\quad Q \Rightarrow \text{wp}(\text{“}S\text{”}, R) \\
\Leftrightarrow \ < \text{Instantiate } S > \\
\quad Q \Rightarrow \text{wp}(\text{“}s,k := s+b(k), k+1\text{''}, R) \\
\Leftrightarrow \ < \text{Instantiate } R\text{; def. of wp}(:=,R) > \\
\quad Q \Rightarrow ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (0 \le k \le n)))^{s,k}_{(s+b(k)),(k+1)} \\
\Leftrightarrow \ < \text{definition of } R^{s,k}_{(\mathcal{E}_0),(\mathcal{E}_1)} > \\
\quad Q \Rightarrow ((s+b(k) = (\sum i \mid 0 \le i < k+1 : b(i)) \wedge (0 \le k+1 \le n))) \\
\Leftrightarrow \ < \text{Split range} > \\
\quad Q \Rightarrow ((s+b(k) = (\sum i \mid 0 \le i < k : b(i)) + b(k)) \wedge (0 \le k+1 \le n)) \\
\Leftrightarrow \ < \text{Instantiate } Q\text{; algebra; algebra} > \\
\quad ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (0 \le k < n))) \Rightarrow ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (-1 \le k < n))) \\
\Leftrightarrow \ < \text{algebra} > \\
\quad ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge (0 \le k < n))) \\
\qquad \Rightarrow ((s = (\sum i \mid 0 \le i < k : b(i)) \wedge ((0 \le k < n) \vee (-1 = k)))) \\
\Leftrightarrow \ < \text{Weakening/strengthening} > \\
\quad T
\end{array}
$$

☛ BACK TO TEXT

**Homework 2.7.1.3** Prove the correctness of the following code segment. It swaps the contents of $b(i)$ and $b(j)$. You may skip checking if the expressions being assigned are valid (since they clearly are).

| |
|---|
| $\{ \; Q : (\forall k \mid 0 \le k < n : b(k) = \widehat{b}(k)) \wedge (0 \le i < n) \wedge (0 \le j < n) \; \}$ |
| $t := b(i)$ |
| $b(i) := b(j)$ |
| $b(j) := t$ |
| $\left\{ \begin{array}{l} R: \; (\forall k \mid (0 \le k < n) \wedge (k \ne i) \wedge (k \ne j) : b(k) = \widehat{b}(k)) \\ \qquad \wedge (b(i) = \widehat{b}(j)) \wedge (b(j) = \widehat{b}(i)) \wedge (0 \le i < n) \wedge (0 \le j < n) \end{array} \right\}$ |

**Answer:**

$Q \Rightarrow \text{wp}(\text{“}t := b(i); b(i) := b(j); b(j) := t\text{”}, R)$

$\Leftrightarrow$ < instantiate >

$Q \Rightarrow \text{wp}(\text{“}t := b(i); b(i) := b(j); b(j) := t\text{”},$
$\quad (\forall k \mid (0 \le k < n) \wedge (k \ne i) \wedge (k \ne j) : b(k) = \widehat{b}(k))$
$\quad\quad \wedge (b(i) = \widehat{b}(j)) \wedge (b(j) = \widehat{b}(i)) \wedge (0 \le i < n) \wedge (0 \le j < n)\ )$

$\Leftrightarrow$ < definition of := >

$Q \Rightarrow \text{wp}(\text{“}t := b(i); b(i) := b(j)\text{”},$
$\quad (\forall k \mid (0 \le k < n) \wedge (k \ne i) \wedge (k \ne j) : b(k) = \widehat{b}(k))$
$\quad\quad \wedge (b(i) = \widehat{b}(j)) \wedge (t = \widehat{b}(i)) \wedge (0 \le i < n) \wedge (0 \le j < n)\ )$

$\Leftrightarrow$ < definition of := >

$Q \Rightarrow \text{wp}(\text{“}t := b(i)\text{”},$
$\quad (\forall k \mid (0 \le k < n) \wedge (k \ne i) \wedge (k \ne j) : b(k) = \widehat{b}(k))$
$\quad\quad \wedge (b(j) = \widehat{b}(j)) \wedge (t = \widehat{b}(i)) \wedge (0 \le i < n) \wedge (0 \le j < n)\ )$

$\Leftrightarrow$ < definition of := >

$Q \Rightarrow (\forall k \mid (0 \le k < n) \wedge (k \ne i) \wedge (k \ne j) : b(k) = \widehat{b}(k))$
$\quad \wedge (b(j) = \widehat{b}(j)) \wedge (b(i) = \widehat{b}(i)) \wedge (0 \le i < n) \wedge (0 \le j < n)\ )$

$\Leftrightarrow$ < split range (in reverse) >

$Q \Rightarrow (\forall k \mid (0 \le k < n) : b(k) = \widehat{b}(k)) \wedge (0 \le i < n) \wedge (0 \le j < n)\ )$

$\Leftrightarrow$ < right-hand side of $\Rightarrow$ is $Q$ >

$Q \Rightarrow Q$

$\Leftrightarrow$ < $\Rightarrow$-simplification >

$T$

☛ BACK TO TEXT

**Homework 2.7.1.4** Prove the following code segment correct:

```
{(∀j|0 ≤ j < i : m ≥ b(j))}
if
  b(i) ≤ m  →  skip
▯ b(i) ≥ m  →  m := b(i)
fi
i := i+1
{(∀j|0 ≤ j < i : m ≥ b(j))}
```

☛ BACK TO TEXT

**Homework 2.7.1.5** The greatest common divisor (gcd) of two positive integers, $x$ and $y$, is defined to be the largest integer $k$ that evenly divides both $x$ and $y$. Let $\text{gcd}(x, y)$ be the function that returns this integer. A property of this function is that if $x < y$ then $\gcd(x, y-x) = \gcd(x, y)$ and if $y < x$ then $\gcd(x-y, y) = \gcd(x, y)$. Obviously, if $x = y$ then $x = y = \gcd(x, y)$.

Prove the partial correctness of the following program for computing $\gcd(x, y)$, returning the result in updated variables $x$ and $y$. (If you feel energetic, prove complete correctness!)

```
{ Q: $(x = \widehat{x}) \wedge (y = \widehat{y}) \wedge (\widehat{x} > 0) \wedge (\widehat{y} > 0)$ }
{ $P_{\text{inv}} : (\gcd(x,y) = \gcd(\widehat{x},\widehat{y})) \wedge (0 < x \le \widehat{x}) \wedge (0 < y \le \widehat{y})$ }
{ $t : \text{abs}(x + y)$ }
while $x \neq y$ do
    if
        $x < y \longrightarrow y := y - x$
      ▯ $y < x \longrightarrow x := x - y$
    fi
endwhile
{ $x = y = \gcd(\widehat{x},\widehat{y})$ }
```

**Answer:** Here is a more fully annotated algorithm:

```
{ $(x = \widehat{x}) \wedge (y = \widehat{y}) \wedge (\widehat{x} > 0) \wedge (\widehat{y} > 0)$ }
{ $(\gcd(x,y) = \gcd(\widehat{x},\widehat{y})) \wedge (0 < x \le \widehat{x}) \wedge (0 < y \le \widehat{y})$ }
while $x \neq y$ do
    {$(\gcd(x,y) = \gcd(\widehat{x},\widehat{y})) \wedge (0 < x \le \widehat{x}) \wedge (0 < y \le \widehat{y}) \wedge (x \neq y)$ }
    if
        $x < y \longrightarrow y := y - x$
      ▯ $y < x \longrightarrow x := x - y$
    fi
    {$(\gcd(x,y) = \gcd(\widehat{x},\widehat{y})) \wedge (0 < x \le \widehat{x}) \wedge (0 < y \le \widehat{y})$ }
endwhile
{ $(\gcd(x,y) = \gcd(\widehat{x},\widehat{y})) \wedge (0 < x \le \widehat{x}) \wedge (0 < y \le \widehat{y}) \wedge \neg(x \neq y)$ }
{ $x = y = \gcd(\widehat{x},\widehat{y})$ }
```

☛ BACK TO TEXT

**Homework 3.1.1.1** Consider the problem of summing the elements of array $b$, as outlined in Figure 3.1. Place the following assertions in the correct place (the blank boxes) in the algorithm. Some need to be inserted in multiple places. Some are just there to confuse you (and not be used)

1. $s = (\sum i \mid k \leq i < n : b(i)) \wedge 0 \leq k \leq n$
2. $k < n$
3. $0 \leq k$
4. $0 < k$
5. $s := 0$
6. $k = 0$
7. $k := n$
8. $s := s + b(k)$
9. $s := s + b(k-1)$

**Answer:**

```
{ $0 \leq n$ }
$s := 0$
$k := n$
{ $s = (\sum i \mid k \leq i < n : b(i)) \wedge 0 \leq k \leq n$ }
while $0 < k$ do
    { $s = (\sum i \mid k \leq i < n : b(i)) \wedge 0 \leq k \leq n \quad \wedge \quad 0 < k$ }
    $k := k - 1$
    $s := s + b(k)$
    { $s = (\sum i \mid k \leq i < n : b(i)) \wedge 0 \leq k \leq n$ }
endwhile
{ $s = (\sum i \mid k \leq i < n : b(i)) \wedge 0 \leq k \leq n \quad \wedge \quad \neg( \ 0 < k \ )$ }
{ $s = (\sum i \mid 0 \leq i < n : b(i))$ }
```

☛ BACK TO TEXT

**Homework 3.2.2.1** Systematically derive the expressions $\mathcal{E}_0$ and $\mathcal{E}_1$ that make the following code segment correct:

```
{ $Q : (s = \widehat{s}) \wedge (t = \widehat{t})$ }
$s, t := \mathcal{E}_0, \mathcal{E}_1$
{ $R : (s = \widehat{t}) \wedge (t = \widehat{s})$ }
```

☛ BACK TO TEXT

**Homework 3.2.2.2** Find the missing assignment to make the following program segment correct. It may be part of a program that sets variable fac equal to $(n-1)! = (n-1) \times (n-2) \times \cdots \times 2 \times 1$. **(Check your results!)**

|  |
|---|
| $\{\; Q : 0 < n \;\}$ |
| $i, \text{fac} := n, ?$ |
| $\{\; R : 1 \leq i \leq n \wedge \text{fac} = (\prod j \mid i \leq j < n : j) \;\}$ |

a) $i, \text{fac} := n, 0$

b) $i, \text{fac} := n, 1$

c) $i, \text{fac} := n, n$

d) cannot be made to be correct

**Answer:** b)

|  |
|---|
| $\{\; Q : 0 < n \;\}$ |
| $\{\; \text{wp}(\text{“}i, \text{fac} := n, \mathcal{E}\text{”}, R) : 1 \leq n \leq n \wedge \mathcal{E} = (\prod j \mid n \leq j < n : j) \;\}$ |
| $i, \text{fac} := n, \mathcal{E}$ |
| $\{\; R : 1 \leq i \leq n \wedge \text{fac} = (\prod j \mid i \leq j < n : j) \;\}$ |

From this we conclude that $\mathcal{E}$ should be chosen to equal 1 since the product over the empty range equals 1.

Check:

$$\begin{aligned}
& 0 < n \Rightarrow \text{wp}(\text{“}i, \text{fac} := n, 1\text{”}, R) \\
\Leftrightarrow & < \text{definition of} := > \\
& 0 < n \Rightarrow 1 \leq n \leq n \wedge 1 = (\prod j \mid n \leq j < n : j) \\
\Leftrightarrow & < \text{algebra; product over empty range} > \\
& 1 \leq n \Rightarrow 1 \leq n \wedge 1 = 1 \\
\Leftrightarrow & < \text{algebra; } \wedge\text{-simplification; } \Rightarrow\text{-simplification} > \\
& T
\end{aligned}$$

☛ BACK TO TEXT

**Homework 3.2.2.3** Find the missing assignment to make the following program segment correct. It may be part of a program that coverts a binary representations (stored in array $b$) into a decimal number stored in $y$. **(Check your results!)**

|  |
|---|
| $\{\; Q : y = (\sum j \mid i \leq j < n : b(j) \times 2^j) \;\}$ |
| $y :=?$ |
| $i := i - 1$ |
| $\{\; R : y = (\sum j \mid i \leq j < n : b(j) \times 2^j) \;\}$ |

a) $y := b(i) \times 2^i + y$

b) $y := b(i-1) \times 2^{i-1} + y$

c) $y := b(i+1) \times 2^{i+1} + y$

d) cannot be made to be correct

**Answer:** b)

$\{\ Q : y = (\sum j \mid i \leq j < n : b(j) \times 2^j)\ \}$

$\{\ \text{wp}(``y := \mathcal{E}; i := i-1", R) : \quad \mathcal{E} = (\sum j \mid i-1 \leq j < n : b(j) \times 2^j)$
$\Leftrightarrow$
$\mathcal{E} = b(i-1) \times 2^{i-1} + (\sum j \mid i \leq j < n : b(j) \times 2^j)\ \}$

$y := \mathcal{E}$

$\{\ \text{wp}(``i := i-1", R) : y = (\sum j \mid i-1 \leq j < n : b(j) \times 2^j)\ \}$

$i := i-1$

$\{\ R : y = (\sum j \mid i \leq j < n : b(j) \times 2^j)\ \}$

From this we conclude that $\mathcal{E}$ should be chosen to equal $b(i-1) \times 2^{i-1} + y$.

Check:

$$
\begin{aligned}
& Q \Rightarrow \text{wp}(``y := b(i-1) \times 2^{i-1} + y; i := i-1", R) \\
\Leftrightarrow\ & < \text{instantiate R} > \\
& Q \Rightarrow \text{wp}(``y := b(i-1) \times 2^{i-1} + y; i := i-1", y = (\sum j \mid i \leq j < n : b(j) \times 2^j)) \\
\Leftrightarrow\ & < \text{definition of} := > \\
& Q \Rightarrow \text{wp}(``y := b(i-1) \times 2^{i-1} + y", y = (\sum j \mid i-1 \leq j < n : b(j) \times 2^j)) \\
\Leftrightarrow\ & < \text{definition of} := > \\
& Q \Rightarrow b(i-1) \times 2^{i-1} + y = (\sum j \mid i-1 \leq j < n : b(j) \times 2^j) \\
\Leftrightarrow\ & < \text{split range} > \\
& Q \Rightarrow b(i-1) \times 2^{i-1} + y = b(i-1) \times 2^{i-1} + (\sum j \mid i \leq j < n : b(j) \times 2^j) \\
\Leftrightarrow\ & < \text{instantiate } Q\text{; algebra} > \\
& y = (\sum j \mid i \leq j < n : b(j) \times 2^j) \Rightarrow y = (\sum j \mid i \leq j < n : b(j) \times 2^j) \\
\Leftrightarrow\ & < \Rightarrow\text{-simplification} > \\
& T
\end{aligned}
$$

☛ BACK TO TEXT

**Homework 3.2.3.1** Find the missing assignment to make the following program segment correct. **(Check your results!)**

$\{\ Q : 0 < i < n\ \}$

$i, j := i+1, ?$

$\{\ R : j = n - i \wedge 0 \leq j < n\ \}$

a) $i, j := i+1, n-i$

b) $i, j := i+1, n-i+1$

c) $i, j := i+1, n-i-1$

d) cannot be made to be correct

**Answer:** c)

| |
|---|
| $\{\ Q: 0 < i < n\ \}$ |
| $\{\ \text{wp}(\text{“}i, j := i+1, \mathcal{E}\text{”}, R): \mathcal{E} = n-(i+1) \wedge 0 \leq \mathcal{E} < n\ \}$ |
| $i, j := i+1, \mathcal{E}$ |
| $\{\ R: j = n-i \wedge 0 \leq j < n\ \}$ |

From this we conclude that $\mathcal{E}$ should be chosen to equal $n-(i+1)$.

Check:

$$
\begin{array}{ll}
 & 0 < i < n \Rightarrow n-(i+1) = n-(i+1) \wedge 0 \leq n-(i+1) < n \\
\Leftrightarrow & < \text{algebra; } \wedge\text{-simplification} > \\
 & 0 < i < n \Rightarrow 0 \leq n-(i+1) < n \\
\Leftrightarrow & < \text{algebra} > \\
 & 0 < i < n \Rightarrow 0 \leq n-i-1 < n \\
\Leftrightarrow & < \text{algebra} > \\
 & 0 < i < n \Rightarrow -n \leq -i-1 < 0 \\
\Leftrightarrow & < \text{algebra} > \\
 & 0 < i < n \Rightarrow 0 < i+1 \leq n \\
\Leftrightarrow & < \text{algebra} > \\
 & 0 < i < n \Rightarrow -1 < i \leq n-1 \\
\Leftrightarrow & < \text{algebra; } i \text{ and } n \text{ are integers} > \\
 & 0 < i \wedge i < n \Rightarrow 0 \leq i < n \\
\Leftrightarrow & < \text{algebra} > \\
 & 0 < i < n \Rightarrow (0 < i < n) \vee (0 = i) \\
\Leftrightarrow & < \text{weakening/strengthening} > \\
 & T
\end{array}
$$

☛ BACK TO TEXT

**Homework 3.2.3.2** Find the missing assignment to make the following program segment correct. **(Check your results!)**

| |
|---|
| $\{\ Q: 0 < i < n\ \}$ |
| $i := i+1$ |
| $j :=?$ |
| $\{\ R: j = n-i \wedge 0 \leq j < n\ \}$ |

a) $j := n - i$

b) $j := n - i + 1$

c) $j := n - i - 1$

d) cannot be made to be correct

**Answer:** a)

| |
|---|
| $\{ \; Q : 0 < i < n \; \}$ |
| $\{ \; \text{wp}(\text{“}i := i+1; j := \mathcal{E}(i)\text{”}, R) : \mathcal{E}(i+1) = n - (i+1) \wedge 0 \leq \mathcal{E}(i+1) < n \; \}$ |
| $i := i+1$ |
| $\{ \; \text{wp}(\text{“}j := \mathcal{E}(i)\text{”}, R) : \mathcal{E}(i) = n - i \wedge 0 \leq \mathcal{E}(i) < n \; \}$ |
| $j := E(i)$ |
| $\{ \; R : j = n - i \wedge 0 \leq j < n \; \}$ |

From this we conclude that $\mathcal{E}(i+1)$ should be chosen to equal $n-(i+1)$ and hence $E(i)$ should be chosen to equal $n-i$.

Check:

$$0 < i < n \Rightarrow \text{wp}(\text{“}i := i+1; j := n-i\text{”}, j = n - i \wedge 0 \leq j < n)$$

$$\Leftrightarrow < \text{definition of } := >$$

$$0 < i < n \Rightarrow \text{wp}(\text{“}i := i+1\text{”}, n - i = n - i \wedge 0 \leq n - i < n)$$

$$\Leftrightarrow < \text{algebra; } \wedge\text{-simplification} >$$

$$0 < i < n \Rightarrow \text{wp}(\text{“}i := i+1\text{”}, 0 \leq n - i < n)$$

$$\Leftrightarrow < \text{definition of } := >$$

$$0 < i < n \Rightarrow 0 \leq n - (i+1) < n$$

$$\Leftrightarrow < \text{algebra (for many of the steps, see the answer to the last homework)} >$$

$$0 < i < n \Rightarrow 0 \leq i < n$$

$$\Leftrightarrow < \text{algebra; weakening/strengthening} >$$

$$T$$

☛ BACK TO TEXT

**Homework 3.2.3.3** Find the missing assignment to make the following program segment correct. **(Check your results!)**

| |
|---|
| $\{ \; Q : 0 < i < n \; \}$ |
| $j :=?$ |
| $i := i+1$ |
| $\{ \; R : j = n - i \wedge 0 \leq j < n \; \}$ |

a) $j := n - i$

b) $j := n - i + 1$

c) $j := n - i - 1$

d) cannot be made to be correct

**Answer:** c)

| |
|---|
| $\{\ Q : 0 < i < n\ \}$ |
| $\{\ \text{wp}(\text{“}j := \mathcal{E}; i := i+1\text{”}, R) : \mathcal{E} = n - (i+1) \wedge 0 \le \mathcal{E} < n\ \}$ |
| $j := \mathcal{E}$ |
| $\{\ \text{wp}(\text{“}i := i+1\text{”}, R) : j = n - (i+1) \wedge 0 \le j < n\ \}$ |
| $i := i + 1$ |
| $\{\ R : j = n - i \wedge 0 \le j < n\ \}$ |

From this we conclude that $\mathcal{E}$ should be chosen to equal $n-(i+1)$ or, equivalently, $n-i-1$.

Check:

$$\begin{array}{ll}
& 0 < i < n \Rightarrow \text{wp}(\text{“}j := n-i-1; i := i+1\text{”}, j = n - i \wedge 0 \le j < n) \\
\Leftrightarrow & < \text{definition of} := > \\
& 0 < i < n \Rightarrow \text{wp}(\text{“}j := n-i-1\text{”}, j = n-(i+1) \wedge 0 \le j < n) \\
\Leftrightarrow & < \text{definition of} :=;\ \text{algebra} > \\
& 0 < i < n \Rightarrow n-i-1 = n-i-1 \wedge 0 \le n-i-1 < n) \\
\Leftrightarrow & < \text{algebra}; \wedge\text{-simplification} > \\
& 0 < i < n \Rightarrow 0 \le n-i-1 < n) \\
\Leftrightarrow & < \text{algebra} > \\
& 0 < i < n \Rightarrow 1 \le i < n \\
\Leftrightarrow & < \text{algebra} > \\
& 0 < i < n \Rightarrow 0 < i < n \\
\Leftrightarrow & < \Rightarrow\text{-simplification} > \\
& T
\end{array}$$

☛ BACK TO TEXT

**Homework 3.3.2.1** Identify for each of the operations on the left the corresponding predicate that best expresses it on the right.

| | |
|---|---|
| 1. $z = \text{abs}(x)$ | a. $(x \le 0 \wedge c = \widehat{c} + 1) \vee (x > 0 \wedge c = \widehat{c})$ |
| 2. $z = \text{min}(x, y)$ | b. $(x \le y \wedge z = y) \vee (x \ge y \wedge z = x)$ |
| 3. $z = \text{max}(x, y)$ | c. $(x \le y \wedge z = x) \vee (x \ge y \wedge z = y)$ |
| 4. $z = \text{abs}(x - y)$ | d. $(x \ge 0 \wedge z = x) \vee (x \le 0 \wedge z = -x)$ |
| 5. Increment $c$ by one if $x \le 0$ | e. $(x \ge y \wedge z = x - y) \vee (y \ge x \wedge z = y - x)$ |

**Answer:**

| | |
|---|---|
| 1. $z = \text{abs}(x)$ | d. $(x \geq 0 \wedge z = x) \vee (x \leq 0 \wedge z = -x)$ |
| 2. $z = \min(x, y)$ | c. $(x \leq y \wedge z = x) \vee (x \geq y \wedge z = y)$ |
| 3. $z = \max(x, y)$ | b. $(x \leq y \wedge z = y) \vee (x \geq y \wedge z = x)$ |
| 4. $z = \text{abs}(x - y)$ | e. $(x \geq y \wedge z = x - y) \vee (y \geq x \wedge z = y - x)$ |
| 5. Increment $c$ by one if $x \leq 0$ | a. $(x \leq 0 \wedge c = \widehat{c} + 1) \vee (x > 0 \wedge c = \widehat{c})$ |

☛ BACK TO TEXT

**Homework 3.3.2.2** Use Figure 3.6 to develop a code segment that computes $z = \min(x, y)$:

| |
|---|
| $\{\ Q : T \}$ |
| $S$ |
| $\{\ R : (x \leq y \wedge z = x) \vee (x \geq y \wedge z = y)\ \}$ |

**Answer:**

| |
|---|
| $\{\ Q : T\ \}$ |
| $\{\ Q \Rightarrow G_0 \vee G_1$? YES $\}$ |
| **if** |
| $x \leq y \rightarrow$ |
| $\{G_0 \wedge Q : x \leq y \wedge T\ \}$ |
| $\{G_0 \wedge Q \Rightarrow \text{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0)$? YES! $\}$ |
| $\{\text{wp}(\text{“}S_0\text{”}, G_0 \wedge R_0) : x \leq y \wedge \mathcal{E}_0 = x\ \}$ |
| $S_0 : z := \mathcal{E}_0 = x$ |
| $\{G_0 \wedge R_0 : x \leq y \wedge x \leq y \wedge z = x\ \}$ |
| $\square\, x \geq y \rightarrow$ |
| $\{G_1 \wedge Q) :\ \}$ |
| $\{G_1 \wedge Q \Rightarrow \text{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1)$? YES! $\}$ |
| $\{\text{wp}(\text{“}S_1\text{”}, G_1 \wedge R_1) : x \geq y \wedge \mathcal{E}_1 = y\ \}$ |
| $S_1 : z := \mathcal{E}_1 = y$ |
| $\{G_1 \wedge R_1 : x \geq y \wedge x \geq y \wedge z = y\ \}$ |
| **fi** |
| $\{\ R : (x \leq y \wedge z = x) \vee (x \geq y \wedge z = y)\ \}$ |

☛ BACK TO TEXT

**Homework 3.3.2.3** Use Figure 3.6 to develop a code segment that computes $z = \text{abs}(x - y)$:

| |
|---|
| $\{\ Q : T \}$ |
| $S$ |
| $\{\ R : (x \geq y \wedge z = x - y) \vee (y \geq x \wedge z = y - x) \}$ |

**Answer:**

| |
|---|
| $\{\ Q : T \}$ |
| $\{\ Q \Rightarrow G_0 \vee G_1$? YES $\}$ |
| **if** |
| $x \geq y \rightarrow$ |
| $\{G_0 \wedge Q : x \geq y \wedge T \}$ |
| $\{G_0 \wedge Q \Rightarrow$ wp("$S_0$", $G_0 \wedge R_0$)? YES! $\}$ |
| $\{$wp("$S_0$", $G_0 \wedge R_0$) $: x \geq y \wedge \mathcal{E}_0 = x - y \}$ |
| $S_0 : z := \mathcal{E}_0 = x - y$ |
| $\{G_0 \wedge R_0 : x \geq y \wedge z = x - y \}$ |
| ▯$y \geq x \rightarrow$ |
| $\{G_1 \wedge Q) : \}$ |
| $\{G_1 \wedge Q \Rightarrow$ wp("$S_1$", $G_1 \wedge R_1$)? YES! $\}$ |
| $\{$wp("$S_1$", $G_1 \wedge R_1$) $: y \geq x \wedge \mathcal{E}_1 = y - x \}$ |
| $S_1 : z := \mathcal{E}_1 = y - x$ |
| $\{G_1 \wedge R_1 : y \geq x \wedge z = y - x \}$ |
| **fi** |
| $\{\ R : (x \geq y \wedge z = x - y) \vee (y \geq x \wedge z = y - x) \}$ |

☛ BACK TO TEXT

**Homework 3.4.5.1** Derive the algorithm for adding array $x$ to array $y$ corresponding to Invariant 2 using the worksheet in Figure 3.10.

**Answer:** Watch the videos!

☛ BACK TO TEXT

**Homework 3.5.2.1** At the end of the last video, you were asked to derive the loop guard $G$ from

| |
|---|
| $\{\ P_{\text{inv}} \wedge \neg G : y = (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \wedge 1 \leq k \leq n+1 \wedge \neg G \}$ |
| $\{\ R : C = AB + \widehat{C} \}$ |

Indicate which of the following is a correct loop guard $G$ (there may be more than one...)

a) $1 < k$.

b) $k < n$.

$$\{\; Q : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge 0 \le n \;\}$$

$$\left\{ \begin{array}{l} \mathrm{wp}(\text{“}S_I\text{”}, P_{\mathrm{inv}}) : \; (\forall i \mid 0 \le i < 0 : y(i) = \widehat{y}(i) + x(i)) \;\wedge\; (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\; 0 \le n \le n \end{array} \right\}$$

$k := n$

$$\{\; P_{\mathrm{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \;\}$$

**while** $0 < k$ **do**

$$\{ P_{\mathrm{inv}} \wedge G : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \wedge 0 < k \}$$

$$\left\{ \begin{array}{l} P_{\mathrm{inv}} \wedge G : \; (\forall i \mid 0 \le i < k-1 : y(i) = \widehat{y}(i) + x(i)) \;\wedge\; y(k-1) = \widehat{y}(k-1) \\ \qquad\qquad \wedge\; (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \;\wedge\; 0 < k \le n. \end{array} \right\}$$

$$\left\{ \begin{array}{l} \mathrm{wp}(\text{“}U; k := k-1\text{”}, P_{\mathrm{inv}}) : \\ \qquad\qquad (\forall i \mid 0 \le i < k-1 : y(i) = \widehat{y}(i)) \;\wedge\; y(k-1) + x(k-1) = \widehat{y}(k-1) + x(k-1) \\ \qquad \wedge\; (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \;\wedge\; 0 \le k-1 \le n. \end{array} \right\}$$

$U : y(k-1) := y(k-1) + x(k-1)$

$$\left\{ \begin{array}{l} \mathrm{wp}(\text{“}k := k-1\text{”}, P_{\mathrm{inv}}) : \\ \qquad\qquad (\forall i \mid 0 \le i < k-1 : y(i) = \widehat{y}(i)) \;\wedge\; y(k-1) = \widehat{y}(k-1) + x(k-1) \\ \qquad \wedge\; (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \;\wedge\; 0 \le k-1 \le n. \end{array} \right\}$$

$k := k - 1$

$$\left\{ \begin{array}{l} P_{\mathrm{inv}} : \\ \qquad\qquad (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \;\wedge\; y(k) = \widehat{y}(k) + x(k) \\ \qquad \wedge\; (\forall i \mid k < i < n : y(i) = \widehat{y}(i) + x(i)) \;\wedge\; 0 \le k \le n. \end{array} \right\}$$

$$\{ P_{\mathrm{inv}} : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \}$$

**endwhile**

$$\{\; P_{\mathrm{inv}} \wedge \neg G : (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \wedge \neg(0 < k) \;\}$$

$$\{\; R : (\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i) + x(i)) \;\}$$

Figure 3.4: Completed worksheet for adding vector $x$ to vector $y$ (Variant 2).

c) $k \neq 1$.

d) $k \neq n$.

**Answer:** Both a) and c) are correct:

$$y = (\sum i \mid k \le i \le n : p(i) x^{i-k}) \wedge 1 \le k \le n+1 \wedge \neg(1 < k)$$
$$\Rightarrow C = AB + \widehat{C}$$

and

$$y = (\sum i \mid k \le i \le n : p(i) x^{i-k}) \wedge 1 \le k \le n+1 \wedge \neg(1 \neq k)$$
$$\Rightarrow C = AB + \widehat{C}$$

are both TRUE.

Enter a correct loop guard in the Live Script.

☛ BACK TO TEXT

**Homework 3.5.2.2** At the end of the last video, you were asked to derive the initialization command

$k =$
$y =$

to make

| |
|---|
| $\{ \; Q : 0 < n \; \}$ |
| $k :=$ |
| $y :=$ |
| $\{ \; P_{\text{inv}} : y = (\sum i \mid k \le i \le n : p(i) x^{i-k}) \wedge 1 \le k \le n+1 \; \}$ |

correct. Indicate which of the following is a correct initialization.
(There may be more than one...)

a) $k := n$
$y := p(n)$

b) $k := n+1$
$y := 0$

c) $k := n$
$y := 1$

d) $k := 0$
$y := 0$

**Answer:** Both a) and b) are correct.

$$0 < n \Rightarrow \text{wp}(\text{“}k := n; y := p(n)\text{”}, y = (\sum i \mid k \le i \le n : p(i) x^{i-k}) \wedge 1 \le k \le n+1)$$

and

$$0 < n \Rightarrow \text{wp}(\text{“}k := n+1; y := 0\text{”}, y = (\sum i \mid k \le i \le n : p(i) x^{i-k}) \wedge 1 \le k \le n+1)$$

are both TRUE.

Enter a correct initialization command in the Live Script.

☛ BACK TO TEXT

**Homework 3.5.2.3** At the end of the last video, you were asked to derive the commands in the loop body

$$\begin{array}{l} k = k-1 \\ y = \end{array}$$

to make

| |
|---|
| $\{P_{\rm inv} \wedge G : y = (\sum i \mid k \le i \le n : p(i)x^{i-k}) \wedge 1 \le k \le n+1 \wedge 1 < k \quad \}$ |
| $k := k-1$ |
| $y :=???$ |
| $\{P_{\rm inv} : y = (\sum i \mid k \le i \le n : p(i)x^{i-k}) \wedge 1 \le k \le n+1 \quad \}$ |

correct. Indicate which of the following is the correct choice for updating $y$. Hint: derive it systematically!

a) $k := k-1$

$y := p(k) \times x^{k-1} + y$

b) $k := k-1$

$y := p(k) + y \times x$

c) $k := k-1$

$y := y + p(k-2) \times x^{k-1}$

d) $k := k-1$

$y := y + p(k) \times x$

**Answer:** Only b) is correct. (Hmmm, showing that the other ones are not correct could be quite burdensome... But you won't be able to prove them correct.)

Here is the derivation of the loop body:

| |
|---|
| $\{P_{\rm inv} \wedge G : y = (\sum i \mid k \le i \le n : p(i)x^{i-k}) \wedge 1 \le k \le n+1 \wedge 1 < k \quad \}$ |
| $\{\text{wp}(\text{“}k := k-1; y := E(k)\text{”}, P_{\rm inv}) : E(k-1) = (\sum i \mid k-1 \le i \le n : p(i)x^{i-(k-1)}) \wedge 1 \le k-1 \le n+1 \quad \}$ |
| $k := k-1$ |
| $\{\text{wp}(\text{“}y := E(k)\text{”}, P_{\rm inv}) : E(k) = (\sum i \mid k \le i \le n : p(i)x^{i-k}) \wedge 1 \le k \le n+1 \quad \}$ |
| $y := E(k)$ |
| $\{P_{\rm inv} : y = (\sum i \mid k \le i \le n : p(i)x^{i-k}) \wedge 1 \le k \le n+1 \quad \}$ |

Now, let's play with

$$E(k-1) = (\sum i \mid k-1 \le i \le n : p(i)x^{i-(k-1)}) \wedge 1 \le k-1 \le n+1.$$

$$\begin{array}{rcl} E(k-1) & = & (\sum i \mid k-1 \le i \le n : p(i)x^{i-(k-1)}) \wedge 1 \le k-1 \le n+1 \\ & = & p(k-1) + (\sum i \mid k \le i \le n : p(i)x^{i-k+1}) \wedge 1 \le k-1 \le n+1 \\ & = & p(k-1) + (\sum i \mid k \le i \le n : p(i)x^{i-k}) \times x \wedge 0 \le k \le n. \end{array}$$

So that we get

| |
|---|
| $\left\{ P_{\text{inv}} \wedge G : y = (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \wedge 1 \leq k \leq n+1 \wedge 1 < k \right\}$ |
| $\left\{ \text{wp}(``k := k-1; y := E(k)", P_{\text{inv}}) : E(k-1) = p(k-1) + (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \times x \wedge 2 \leq k \leq n+2 \right\}$ |
| $k := k-1$ |
| $\left\{ \text{wp}(``y := E(k)", P_{\text{inv}}) : E(k) = (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \wedge 1 \leq k \leq n+1 \right\}$ |
| $y := E(k)$ |
| $\left\{ P_{\text{inv}} : y = (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \wedge 1 \leq k \leq n+1 \right\}$ |

From what we highlight we notice that

$$E(k-1) = p(k-1) + y \times x$$

or, equivalently, that

$$E(k) = p(k) + y \times x.$$

**Enter a correct update to $y$ in the Live Script and enjoy getting the right answer the first time!**

☛ BACK TO TEXT

**Homework 3.7.1.1** Identify which of the following are valid loop invariants for deriving a loop that computes the dot product:

a) $d = (\sum i \mid 1 \leq i < k : x(i) \times y(i)) \wedge 1 \leq k \leq n+1.$

b) $d = (\sum i \mid k \leq i \leq n : x(i) \times y(i)) \wedge 1 \leq k \leq n+1.$

c) $d = (\sum i \mid 1 \leq i \leq k : x(i) \times y(i)) \wedge 0 \leq k \leq n.$

d) $d = (\sum i \mid k < i \leq n : x(i) \times y(i)) \wedge 0 \leq k \leq n.$

**Answer:** All four are valid loop invariants that lead to algorithms for computing this operation. Of course, you would have to derive algorithms for all four to show this...

☛ BACK TO TEXT

**Homework 3.7.1.2** For the loop invariant

$$d = (\sum i \mid 1 \leq i < k : x(i) \times y(i)) \wedge 1 \leq k \leq n+1$$

derive a correct program for computing $x^T y$. You will want to use the worksheet in Figure 3.11 for this exercise. **Answer:**

☛ BACK TO TEXT

**Homework 3.7.1.3** Implement the program from the last exercise using the Live Script in

```
LAFFPfC −> Assignments −> Week3 −> matlab −> DotVariant1.mlx.
```

For additional instructions, see Homework 3.5.3.3 on the edX platform.

Make sure you get the right answer the first time! **Answer:**

☛ BACK TO TEXT

**Homework 4.1.1.1** Match the predicate on the left with the corresponding predicate on the right:

| | |
|---|---|
| (1) $d = (\sum i \mid 0 \le i < n : x(i) \times y(i))$ | (a) $d = x_B^T y_B$ |
| (2) $d = (\sum i \mid 0 \le i < k : x(i) \times y(i)) + (\sum i \mid k \le i < n : x(i) \times y(i)) \wedge 0 \le k \le n$ | (b) $d = x^T y$ |
| (3) $d = (\sum i \mid 0 \le i < k : x(i) \times y(i)) \wedge 0 \le k \le n$ | (c) $d = x_T^T y_T + x_B^T y_B$ |
| (4) $d = (\sum i \mid k \le i < n : x(i) \times y(i)) \wedge 0 \le k \le n$ | (d) $d = x_T^T y_T$ |

**Answer:**

| | |
|---|---|
| (1) $d = (\sum i \mid 0 \le i < n : x(i) \times y(i))$ | (b) $d = x^T y$ |
| (2) $d = (\sum i \mid 0 \le i < k : x(i) \times y(i)) + (\sum i \mid k \le i < n : x(i) \times y(i)) \wedge 0 \le k \le n$ | (c) $d = x_T^T y_T + x_B^T y_B$ |
| (3) $d = (\sum i \mid 0 \le i < k : x(i) \times y(i)) \wedge 0 \le k \le n$ | (d) $d = x_T^T y_T$ |
| (4) $d = (\sum k \mid k \le i < n : x(i) \times y(i)) \wedge 0 \le k \le n$ | (a) $d = x_B^T y_B$ |

☛ BACK TO TEXT

**Homework 4.1.1.2** In the annotated algorithm for computing $d = x^T y$ in Figure 4.1, place the following expressions where they can replace the quantifiers. Use your intuition!

- $d = x^T y$
- $x \rightarrow \left(\frac{x_T}{x_B}\right)$ and $y \rightarrow \left(\frac{y_T}{y_B}\right)$, where $x_T$ and $y_T$ have no elements
- $d = x_T^T y_T$

☛ BACK TO TEXT

**Homework 4.2.1.1** In the annotated algorithm on Page 183 for copying vector x into vector y insert the expressions where they make sense. Use your intuition!

a) $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$

b) $m(y_T) < m(y)$ (three places).

c) $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$

```
{ $(0 \le n)$ }
$k := 0$        $x \Rightarrow \left(\frac{x_T}{x_B}\right)$ and $y \Rightarrow \left(\frac{y_T}{y_B}\right)$, where $x_T$ and $y_T$ have no elements
$d := 0$
{ $(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n$
  $d = x_T^T y_T$ }
while $k < n$ do
    { $(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \wedge (k < n)$ }        $d = x_T^T y_T$
    $d := d + x(k) \times y(k)$
    $k := k + 1$
    { $(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n$ }  $d = x_T^T y_T$
endwhile
{ $(\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \wedge \neg(k < n)$
  $d = x_T^T y_T$ }
{ $d = (\sum i \mid 0 \le i < n : x(i) \times y(i))$        $d = x^T y$ }
```

Figure 3.5: Answer to Homework 4.1.1.1.

$\{$ $(\forall i \mid 0 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le n)$ $\qquad y = \widehat{y}$ $\}$

$k := 0$ $\qquad x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ and $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$ where $x_T$ and $y_T$ are empty

$$\left\{\begin{array}{l} (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \\ \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \end{array}\right\}$$

**while** $k < n$ **do** $\qquad m(y_T) < m(y)$

$$\left\{\begin{array}{l} P_{\text{inv}} \wedge (k < n) : \\ (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \wedge (k < n) \\ \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \wedge m(y_T) < m(y) \end{array}\right\}$$

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right) \text{ and } \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right)$$

$$\left\{\begin{array}{ll} P_{\text{inv}} \wedge (k < n) \text{ with } k \text{ term split off) :} & \\ (\forall i \mid 0 \le i < k : y(i) = x(i)) \quad \wedge \quad ((y(k) = \widehat{y}(k)) \wedge & \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \widehat{\psi}_1 \\ \hline \widehat{y}_2 \end{array}\right) \\ (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \quad \wedge \quad (0 \le k \le n) \wedge (k < n) & \end{array}\right\}$$

$S :\ \ y(k) := x(k)$ $\qquad \psi_1 := \chi_1$

$$\left\{\begin{array}{ll} \text{wp(“}k := k+1\text{”}, P_{\text{inv}}) \text{ (with } k \text{ term split off) :} & \\ (\forall i \mid 0 \le i < k : y(i) = x(i)) \wedge (y(k) = x(k)) & \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline \widehat{y}_2 \end{array}\right) \\ \wedge (\forall i \mid k+1 \le i < n : y(i) = \widehat{y}(i)) \wedge (0 \le k+1 \le n) & \end{array}\right\}$$

$k := k+1$ $\qquad \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$

$$\left\{\begin{array}{l} (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \\ \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \end{array}\right\}$$

**endwhile**

$$\left\{\begin{array}{l} (\forall i \mid 0 \le i < k : y(i) = \widehat{y}(i)) \wedge (\forall i \mid k \le i < n : y(i) = \widehat{y}(i) + x(i)) \wedge 0 \le k \le n \\ \wedge \neg(k < n) \\ \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \widehat{y}_B \end{array}\right) \wedge \neg(m(y_T) < m(y)) \end{array}\right\}$$

$\{$ $(\forall i \mid 0 \le i < n : y(i) = x(i))$ $\qquad y = x$ $\}$

| Step | Algorithm: $\alpha := x^T y + \alpha$ |
|---|---|
| 1a | $\{\alpha = \widehat{\alpha}$ $\}$ |
| 4 | $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$ where $x_B$ has 0 rows, $y_B$ has 0 rows |
| 2 | $\{\alpha = x_B^T y_B + \widehat{\alpha}$ $\}$ |
| 3 | while $m(x_B) < m(x)$ do |
| 2,3 | $\{$ $\alpha = x_B^T y_B + \widehat{\alpha} \wedge m(x_B) < m(x)$ $\}$ |
| 5a | $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$ where $\chi_1$ has 1 row, $\psi_1$ has 1 row |
| 6 | $\{$ $\alpha = x_2^T y_2 + \widehat{\alpha}$ $\}$ |
| 8 | $\alpha := \chi_1 \times \psi_1 + \alpha$ |
| 7 | $\{$ $\alpha = \chi_1 \times \psi_1 + x_2^T y_2 + \widehat{\alpha}$ $\}$ |
| 5b | $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$ |
| 2 | $\{$ $\alpha = x_B^T y_B + \widehat{\alpha}$ $\}$ |
|  | endwhile |
| 2,3 | $\{\alpha = x_B^T y_B + \widehat{\alpha} \wedge \neg(m(x_B) < m(x))$ $\}$ |
| 1b | $\{\alpha = x^T y + \widehat{\alpha}$ $\}$ |

Figure 4.6: Variant 2 for computing $\alpha := x^T y + \alpha$.

d) $\begin{pmatrix} y_0 \\ \psi_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_0 \\ \widehat{\psi}_1 \\ \widehat{y}_2 \end{pmatrix}$.

e) $\begin{pmatrix} y_0 \\ \psi_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_0 \\ \chi_1 \\ \widehat{y}_2 \end{pmatrix}$

f) $\psi_1 := \chi_1$

**Answer:** See Figure on Page 326

☛ BACK TO TEXT

**Homework 4.2.1.2** Use the ☛ blank worksheet to derive Variant 2 for computing the "sapdot" operation $\alpha := x^T y + \alpha$, the algorithm corresponding to Invariant 2. (In theory, this worksheet is also in `LAFFPfC/Resources/BlankWorksheet.pdf`. In practice, you may want to put a copy there yourself, since the one that is there is not quite the same.)

**Answer:** See Figure **??**

☛ BACK TO TEXT

```
{ $y = \widehat{y}$ }
$x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ and $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$
    where $x_B$ and $y_B$ are empty
{ $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} \widehat{y}_T \\ \hline x_B \end{array}\right)$ }
while $m(y_B) < m(y)$ do
    { $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} \widehat{y}_T \\ \hline x_B \end{array}\right) \wedge (m(y_B) < m(y))$ }
    $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right)$
        where $\chi_1$ and $\psi_1$ are scalars
    { $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} \widehat{y}_T \\ \hline x_B \end{array}\right)$ with split range: $\left(\begin{array}{c} y_0 \\ \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} \widehat{y}_0 \\ \widehat{\psi}_1 \\ x_2 \end{array}\right)$ }
    $\psi_1 := \chi_1$
    { wp(“$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) := \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$; $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) := \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$”, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} \widehat{y}_T \\ \hline x_B \end{array}\right)$) : $\left(\begin{array}{c} y_0 \\ \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} \widehat{y}_0 \\ \chi_1 \\ x_2 \end{array}\right)$ }
    $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$ and $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$
    { $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} \widehat{y}_T \\ \hline x_B \end{array}\right)$ }
endwhile
{ $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} \widehat{y}_T \\ \hline x_B \end{array}\right) \wedge \neg(m(y_B) < m(y))$ }
{ $y = x$ }
```

Figure 4.7: Answer for Homework 4.2.1.2.

**Homework 5.1.1.1** Compute

$$\begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix}\begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 8 \\ -5 \\ -5 \end{pmatrix}$$

using algorithmic Variant 1 given in Figure 5.1.

**Answer:** First iteration:

$$\left(\begin{array}{c} \hline \left(\begin{array}{ccc} \hline 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{array}\right)\begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + 3 \\ 1 \\ 0 \end{array}\right) = \left(\begin{array}{c} \hline 8 \\ 1 \\ 0 \end{array}\right)$$

Second iteration:

$$\left(\begin{array}{c} 8 \\ \hline \left(\begin{array}{ccc} 1 & -1 & 2 \\ \hline -2 & 2 & 0 \\ -1 & 1 & -2 \end{array}\right)\begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + 1 \\ 0 \end{array}\right) = \left(\begin{array}{c} 8 \\ \hline -5 \\ 0 \end{array}\right)$$

Third iteration:

$$\left(\begin{array}{c} 8 \\ -5 \\ \hline \left(\begin{array}{ccc} 1 & -1 & 2 \\ -2 & 2 & 0 \\ \hline -1 & 1 & -2 \end{array}\right)\begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + 0 \end{array}\right) = \left(\begin{array}{c} 8 \\ -5 \\ \hline -5 \end{array}\right)$$

☛ BACK TO TEXT

**Homework 5.1.1.2** Download the Live Script `MatVec1LS.mlx` into `Assignments/Week5/matlab/` and follow the directions in it to execute function `MatVec1`.

**Answer:** Examine `MatVec1LS.mlx`.

☛ BACK TO TEXT

**Homework 5.1.1.3** Knowing that the matrix is symmetric, compute

$$\begin{pmatrix} 1 & \star & \star \\ -2 & 2 & \star \\ -1 & 1 & -2 \end{pmatrix}\begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 6 \\ -4 \\ -5 \end{pmatrix}$$

$$\text{PME: } \left(\frac{C_T}{C_B}\right) = \left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right).$$

| $A_{TL}B_T$ | $A_{BL}^T B_B$ | $A_{BL}B_T$ | $A_{BR}B_B$ | $\left(\frac{C_T}{C_B}\right) =$ | |
|---|---|---|---|---|---|
| Yes | No | No | No | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 1 |
| Yes | Yes | No | No | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 2 |
| Yes | No | Yes | No | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 3 |
| Yes | Yes | Yes | No | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 4 |
| No | Yes | Yes | Yes | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 5 |
| No | Yes | No | Yes | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 6 |
| No | No | Yes | Yes | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 7 |
| No | No | No | Yes | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 8 |

Figure 5.8: Loop-invariants for $C := AB + C$ where $A$ is symmetric and only its lower triangular part is stored.

using algorithmic Variant 1 given in Figure 5.1.

**Answer:** First iteration:

$$\left(\begin{array}{c}\hline \left(\begin{array}{ccc}\hline 1 & -2 & -1\\ -2 & 2 & \star \\ -1 & 1 & -2\end{array}\right)\left(\begin{array}{c}2\\-1\\1\end{array}\right)+3 \\ 1 \\ 0\end{array}\right) = \left(\begin{array}{c}\hline 6\\1\\0\end{array}\right)$$

Second iteration:

$$\left(\begin{array}{c} 6 \\ \hline \left(\begin{array}{ccc} 1 & \star & \star\\ \hline -2 & 2 & 1 \\ -1 & 1 & -2\end{array}\right)\left(\begin{array}{c}2\\-1\\1\end{array}\right)+1 \\ 0\end{array}\right) = \left(\begin{array}{c} 8\\ \hline -4\\0\end{array}\right)$$

Third iteration:

$$\left(\begin{array}{c} 8 \\ -5 \\ \hline \left(\begin{array}{ccc} 1 & \star & \star\\ -2 & 2 & \star \\ \hline -1 & 1 & -2\end{array}\right)\left(\begin{array}{c}2\\-1\\1\end{array}\right)+0 \end{array}\right) = \left(\begin{array}{c} 6\\ -4 \\ \hline -5\end{array}\right)$$

☛ BACK TO TEXT

**Homework 5.1.1.4** Download the Live Script `SymVec1LS.mlx` into `Assignments/Week5/matlab/` and follow the directions in it to change the given function to only compute with the lower triangular part of the matrix.

**Answer:** See Figure 5.9.

☛ BACK TO TEXT

**Homework 5.1.1.5** Knowing that the matrix is symmetric, compute

$$\begin{pmatrix} 1 & \star & \star \\ 2 & -2 & \star \\ -2 & 1 & 3 \end{pmatrix}\begin{pmatrix} 1\\-1\\1\end{pmatrix}+\begin{pmatrix}1\\2\\3\end{pmatrix}=\begin{pmatrix}-2\\7\\3\end{pmatrix}$$

using algorithmic Variant 3 given in Figure 5.4.

☛ BACK TO TEXT

**Homework 5.1.1.6** Which algorithm for computing $y := Ax + y$ casts more computation in terms of the columns of the stored matrix (and is therefore probably higher performing)?

**Answer:** If you analyze it, each of the implementations (`SymMatVec1` and `SymMatVec3`) casts roughly half of the computations in terms of operations with rows and the other half in terms of operations with columns.

☛ BACK TO TEXT

```
function [ y_out ] = SymMatVec1( A, x, y )
% Compute y := A x + y, assuming A is symmetric and stored in lower
% triangular part of array A.

% Extract the row and column size of A
[ m, n ] = size( A );

% (Strictly speaking you should check that m = n, x is a vector size n and y is a
% vector of size n...)

% Copy y into y_out
y_out = y;

% Compute y_out = A * x + y_out
for i = 1:n
    for j=1:i
        y_out( i ) = A( i,j ) * x( j ) + y_out( i );
    end
    for j=i+1:n
        y_out( i ) = A( j,i ) * x( j ) + y_out( i );
    end
end

end
```

LAFFPfC/Assignments/Week5/matlab/SymMatVec1.m

Figure 5.9: Function that computes $y := Ax + y$, returning the result in vector `y_out`. Matrix $A$ is assumed to be symmetric and only stored in the lower triangular part of array `A`.
.

**Homework 5.1.1.7** **(Challenge)** Download the Live Script `SymMatVecByColumnsLS.mlx` into `Assignments/Week5/matlab/` and follow the directions in it to change the given function to only compute with the lower triangular part of the matrix **and** only access the matrix by columns. (Not sort-of-kind-of as in `SymMatVec3.mlx`.)

**Answer:** See Figure 5.10.

☛ BACK TO TEXT

**Homework 5.1.1.8** **(Challenge)** Find someone who knows a little (or a lot) about linear algebra and convince this person that the answer to the last exercise is correct. Alternatively, if you did not manage to come up with an answer for the last exercise, look at the answer to that exercise and convince yourself it is correct.

**Answer:** A less systematic way of providing a convincing argument goes as follows: The key is to view the symmetric matrix as consisting of three parts: the strictly lower triangular part, the diagonal part, and the strictly upper triangular part:

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{1,0} & \cdots & \alpha_{n-1,0} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{n-1,1} \\ \vdots & \vdots & & \vdots \\ \alpha_{n-1,0} & \alpha_{n-1,0} & \cdots & \alpha_{n-1,n-1} \end{pmatrix} =$$

```
function [ y_out ] = SymMatVec2( A, x, y )
% Compute y := A x + y, assuming A is symmetric and stored in lower
% triangular part of array A.

% Extract the row and column size of A
[ m, n ] = size( A );

% (Strictly speaking you should check that m = n, x is a vector size n and y is a
% vector of size n...)

% Copy y into y_out
y_out = y;

% Compute y_out = A * x + y_out
for j = 1:n
    y_out( j ) = A( j,j ) * x( j ) + y_out( j );
    for i=j+1:n
        y_out( j ) = A( i,j ) * x( i ) + y_out( j );
        y_out( i ) = A( i,j ) * x( j ) + y_out( i );
    end
end

end
```

LAFFPfC/Assignments/Week5/matlab/SymMatVec3.m

Figure 5.10: Function that computes $y; = Ax + y$, returning the result in vector `y_out`. Matrix $A$ is assumed to be symmetric and only stored in the lower triangular part of array `A`. This implementation accesses the matrix by columns, which has the potential for better performance.

.

$$\underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 \\ \alpha_{1,0} & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ \alpha_{n-1,0} & \alpha_{n-1,0} & \cdots & 0 \end{pmatrix}}_{L} + \underbrace{\begin{pmatrix} \alpha_{0,0} & 0 & \cdots & 0 \\ 0 & \alpha_{1,1} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \alpha_{n-1,n-1} \end{pmatrix}}_{D} + \underbrace{\begin{pmatrix} 0 & \alpha_{1,0} & \cdots & \alpha_{n-1,0} \\ 0 & 0 & \cdots & \alpha_{n-1,0} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}}_{U}.$$

Now, $Ax = (L + D + U)x = Lx + Dx + Ux$. Think about how the matrix-vector multiplication algorithms can be modified for each of these special cases and which algorithm you would want to pick to access the lower triangular part of $A$ by columns. This gives you three sets loops, one loop per term. You can then "merge" these sets of loops into one outer loop, and you arrive at the answer.

But would this convince someone? Better yet: if you aren't that comfortable with matrix operations, would have thought of the algorithms? Are there other algorithms? Might there be an algorithm that accesses the matrix by columns that would even be better by some criteria? How would you *prove* that the algorithm is correct?

More importantly, this is a "one of" argument. Such arguments are great if you want the person making the argument to look really smart. It leaves the person on the receiving end of the argument thinking "I wish I had thought of that, because then people would think I am really smart!" It is specifically tailored towards this one situation. How can it be made more systematic? How can you come up with the solution to the last exercise hand-in-hand with the proof that it is correct? It is what we do as computer scientist: make knowledge systematic.

☛ BACK TO TEXT

**Homework 5.2.2.1** Below on the left you find four loop invariants for computing $y := Ax + y$ where $A$ has no special structure. On the right you find four loop invariants for computing $y := Ax + y$ when $A$ is symmetric and stored in the lower triangular part of $A$. Match the loop invariants on the right to the loop invariants on the left that you would expect maintain the same values in $y$ before and after each iteration of the loop. (In the video, we mentioned asking you to find two invariants. We think you can handle finding these four!)

(1) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} \widehat{y}_T \\ \hline A_B x + \widehat{y}_B \end{array}\right)$ (a) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BR} x_B + \widehat{y}_B \end{array}\right)$

(2) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} A_T x + \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right)$ (b) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} \widehat{y}_T \\ \hline A_{BL} x_T + A_{BR} x_B + \widehat{y}_B \end{array}\right)$

(3) $y = A_L x_T + \widehat{y}$ (c) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{lr} A_{TL} x_T + & \widehat{y}_T \\ \hline A_{BL} x_T + & \widehat{y}_B \end{array}\right)$

(4) $y = A_R x_B + \widehat{y}$ (d) $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{r} A_{TL} x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline \widehat{y}_B \end{array}\right)$

☛ BACK TO TEXT

**Homework 5.2.3.1** You may want to derive the algorithm corresponding to Invariant 1 yourself, consulting the video if you get stuck. Some resources:

- The ☛ blank worksheet.
- Download ☛ `symv_unb_var1_ws.tex` and place it in `LAFFPfC/Assignments/Week5/LaTeX/`. You will need ☛ `color_flatex.tex` as well in that directory.
- The ☛ Spark webpage.

Alternatively, you may want to download the completed worksheet (with intermediate steps later in the PDF) ☛ `symv_unb_var1_ws_answer.pdf` and/or its source ☛ `symv_unb_var1_ws_answer.tex`.

☛ BACK TO TEXT

**Homework 5.2.4.1** Derive algorithms for Variants 2-8, corresponding to the loop invariants in Figure 5.8. (If you don't have time to do all, then we suggest you do at least Variants 2-4 and Variant 8). Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `symv_unb_var2_ws.tex`, ☛ `symv_unb_var3_ws.tex`, ☛ `symv_unb_var4_ws.tex`, ☛ `symv_unb_var5_ws.tex`, ☛ `symv_unb_var6_ws.tex`, ☛ `symv_unb_var7_ws.tex`, ☛ `symv_unb_var8_ws.tex`.

**Answer:**

Completed worksheets:

☛ `symv_unb_var2_ws_answer.tex`, ☛ `symv_unb_var2_ws_answer.pdf`
☛ `symv_unb_var3_ws_answer.tex`, ☛ `symv_unb_var3_ws_answer.pdf`

☛ symv_unb_var4_ws_answer.tex, ☛ symv_unb_var4_ws_answer.pdf
☛ symv_unb_var5_ws_answer.tex, ☛ symv_unb_var5_ws_answer.pdf
☛ symv_unb_var6_ws_answer.tex, ☛ symv_unb_var6_ws_answer.pdf
☛ symv_unb_var7_ws_answer.tex, ☛ symv_unb_var7_ws_answer.pdf
☛ symv_unb_var8_ws_answer.tex, ☛ symv_unb_var8_ws_answer.pdf

☛ BACK TO TEXT

**Homework 5.2.4.2** Match the loop invariant (on the left) to the "update" in the loop body (on the right):

Invariant 1: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(a) $y_0 := \chi_1 a_{01} + y_0$
$\psi_1 := \alpha_{11}\chi_1 + \psi_1$
$y_2 := \chi_1 a_{21} + y_2$

Invariant 2: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(b) $\psi_1 := \alpha_{11}\chi_1 + a_{21}^T x_2 + \quad \psi_1$
$y_2 := \chi_1 a_{21} + \qquad\quad y_2$

Invariant 3: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(c) $y_0 := \chi_1 (a_{10}^T)^T + y_0$
$\psi_1 := \alpha_{11}\chi_1 + \psi_1$
$y_2 := \chi_1 a_{21} + y_2$

Invariant 4: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(d) $\psi_1 := a_{10}^T x_0 + \alpha_{11}\chi_1 + a_{21}^T x_2 + \psi_1$

Invariant 8: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$

(e) $y_0 := \qquad \chi_1 (a_{10}^T)^T + y_0$
$\psi_1 := a_{10}^T x_0 + \alpha_{11}\chi_1 + \psi_1$

**Answer:**

Invariant 1: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ (e) $\begin{array}{l} y_0 := \quad \chi_1 (a_{10}^T)^T + y_0 \\ \psi_1 := a_{10}^T x_0 + \alpha_{11}\chi_1 + \psi_1 \end{array}$

Invariant 2: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ (d) $\psi_1 := a_{10}^T x_0 + \alpha_{11}\chi_1 + a_{21}^T x_2 + \psi_1$

Invariant 3: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ (c) $\begin{array}{l} y_0 := \chi_1 (a_{10}^T)^T + y_0 \\ \psi_1 := \alpha_{11}\chi_1 + \psi_1 \\ y_2 := \chi_1 a_{21} + y_2 \end{array}$

Invariant 4: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ (b) $\begin{array}{l} \psi_1 := \alpha_{11}\chi_1 + a_{21}^T x_2 + \psi_1 \\ y_2 := \chi_1 a_{21} + y_2 \end{array}$

Invariant 8: $\left(\begin{array}{c} A_{TL}x_T + A_{BL}^T x_B + \widehat{y}_T \\ \hline A_{BL}x_T + A_{BR}x_B + \widehat{y}_B \end{array}\right)$ (b) $\begin{array}{l} y_0 := \chi_1 (a_{10}^T)^T + y_0 \\ \psi_1 := \alpha_{11}\chi_1 + \psi_1 \\ y_2 := \chi_1 a_{21} + y_2 \end{array}$

☛ BACK TO TEXT

**Homework 5.2.4.3** Derive algorithms for Variants 2-8, corresponding to the loop invariants in Figure 5.8. (If you don't have time to do all, then we suggest you do at least Variants 2-4 and Variant 8). Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `symv_unb_var2_ws.tex`, ☛ `symv_unb_var3_ws.tex`, ☛ `symv_unb_var4_ws.tex`, ☛ `symv_unb_var5_ws.tex`, ☛ `symv_unb_var6_ws.tex`, ☛ `symv_unb_var7_ws.tex`, ☛ `symv_unb_var8_ws.tex`.

**Answer:**

Completed worksheets:
☛ `symv_unb_var2_ws_answer.tex`, ☛ `symv_unb_var2_ws_answer.pdf`
☛ `symv_unb_var3_ws_answer.tex`, ☛ `symv_unb_var3_ws_answer.pdf`
☛ `symv_unb_var4_ws_answer.tex`, ☛ `symv_unb_var4_ws_answer.pdf`
☛ `symv_unb_var5_ws_answer.tex`, ☛ `symv_unb_var5_ws_answer.pdf`
☛ `symv_unb_var6_ws_answer.tex`, ☛ `symv_unb_var6_ws_answer.pdf`
☛ `symv_unb_var7_ws_answer.tex`, ☛ `symv_unb_var7_ws_answer.pdf`
☛ `symv_unb_var8_ws_answer.tex`, ☛ `symv_unb_var8_ws_answer.pdf`

☛ BACK TO TEXT

**Homework 5.2.5.1** We now return to the launch for this week and the question of how to find an algorithm for computing $y := Ax + y$, where $A$ is symmetric and stored only in the lower triangular part of $A$. Consult Figure 5.10 to answer the question of which invariant(s) yield an algorithm that accesses the matrix by columns.

**Answer:**

The algorithms corresponding to Invariants 4 and 8 access the matrix by columns: In the update

$$\begin{aligned} \psi_1 &:= \alpha_{11}\chi_1 + a_{21}^T x_2 + \psi_1 \\ y_2 &:= \chi_1 a_{21} + \qquad\quad y_2 \end{aligned}$$

Most computation involves $a_{21}$, which is part of a column of $A$. If the size of $a_{21}$ is $k$, then the update performs $4k$ floating point operations with the $k$ elements of $A$ in $a_{21}$ that must be read from main memory.

If you examine `SymMatVecByColumns.mlx` you will find that it organizes the loops to implement the algorithm that corresponds to Invariant 4. Since you derived it to be correct, you now have the answer of how to convince someone of its correctness. Of course, you still would have to convince that person that it is a correct translation of the given updates!

☛ BACK TO TEXT

**Homework 5.3.2.1** Identify two loop invariants from PME 1. **Answer:**

- Invariant 1: $\left( C_L \,\middle|\, C_R \right) = \left( AB_L + \widehat{C}_L \,\middle|\, AB_R \right)$.
- Invariant 2: $\left( C_L \,\middle|\, C_R \right) = \left( AB_L \,\middle|\, AB_R + \widehat{C}_R \right)$.

☛ BACK TO TEXT

**Homework 5.3.2.2** Derive Variant 1, the algorithm corresponding to Invariant 1, in the answer to the last homework. Assume the algorithm "marches" through the matrix one row or column at a time (meaning you are to derive an unblocked algorithm).

Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `gemm_unb_var1_ws.tex`
- ☛ `GemmUnbVar1LS.mlx`

**Answer:**

Answers:

- ☛ `gemm_unb_var1_ws_answer.tex`,
- ☛ `gemm_unb_var1_ws_answer.pdf`
- ☛ `GemmUnbVar1LSAnswer.mlx`

☛ BACK TO TEXT

**Homework 5.3.2.3** If you feel energetic, repeat the last homework for Invariant 2. **Answer:**

☛ BACK TO TEXT

**Homework 5.3.3.1** Identify a second PME (PME 2) that corresponds to the case where $A$ is partitioned by rows. **Answer:** Partition

$$C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right).$$

Then $C = AB + \widehat{C}$ becomes

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) B + \left(\begin{array}{c} \widehat{C}_T \\ \hline \widehat{C}_B \end{array}\right).$$

Manipulating this yields

- PME 2: $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_T B + \widehat{C}_T \\ \hline A_B B + \widehat{C}_B \end{array}\right).$

☛ BACK TO TEXT

**Homework 5.3.3.2** Identify two loop invariants from this second PME (PME 2). Label these Invariant 3 and Invariant 4. **Answer:**

- Invariant 3: $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_T B + \widehat{C}_T \\ \hline A_B B \end{array}\right).$

- Invariant 4: $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_T B \\ \hline A_B B + \widehat{C}_B \end{array}\right).$

☛ BACK TO TEXT

**Homework 5.3.3.3** Derive Variant 3, the algorithm corresponding to Invariant 3, in the answer to the last homework. Assume the algorithm "marches" through the matrix one row or column at a time (meaning you are to derive an unblocked algorithm).

Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `gemm_unb_var3_ws.tex`
- ☛ `GemmUnbVar3LS.mlx`

**Answer:**

Answers:

- ☛ gemm_unb_var3_ws_answer.tex,
- ☛ gemm_unb_var3_ws_answer.pdf
- ☛ GemmUnbVar3LSAnswer.mlx

☛ BACK TO TEXT

**Homework 5.3.3.4** If you feel energetic, repeat the last homework for Invariant 4, **Answer:**

☛ BACK TO TEXT

**Homework 5.3.4.1** Identify a third PME that corresponds to the case where $A$ is partitioned by columns. **Answer:**
Partition

$$B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right).$$

Then $C = AB + \widehat{C}$ becomes

$$C = \left(\begin{array}{c|c} A_L & A_R \end{array}\right) \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) + \widehat{C}.$$

Manipulating this yields

- PME 3: $C = A_L B_T + A_R B_B + \widehat{C}$.

☛ BACK TO TEXT

**Homework 5.3.4.2** Identify two loop invariants from PME 3. Label these Invariant 5 and Invariant 6. **Answer:**

- Invariant 5: $C = A_L B_T + \widehat{C}$.
- Invariant 6: $C = A_R B_T B + \widehat{C}$.

☛ BACK TO TEXT

**Homework 5.3.4.3** Derive Variant 5, the algorithm corresponding to Invariant 5, in the answer to the last homework. Assume the algorithm "marches" through the matrix one row or column at a time (meaning you are to derive an unblocked algorithm).

Some resources:

- The ☛ blank worksheet.
- ☛ color_flatex.tex.
- Spark webpage.
- ☛ gemm_unb_var5_ws.tex
- ☛ GemmUnbVar5LS.mlx

**Answer:**

Answers:

- ☛ gemm_unb_var5_ws_answer.tex,
- ☛ gemm_unb_var5_ws_answer.pdf
- ☛ GemmUnbVar5LSAnswer.mlx

☛ BACK TO TEXT

**Homework 5.3.4.4** If you feel energetic, repeat the last homework for Invariant 6. **Answer:**

☛ BACK TO TEXT

**Homework 5.3.5.1** Derive Variants 1, 3, and 5, the algorithms corresponding to Invariant 1, 3, and 5.
Some resources:

- The ☛ blank worksheet.
- ☛ color_flatex.tex.
- Spark webpage.
- ☛ gemm_blk_var1_ws.tex, ☛ gemm_blk_var3_ws.tex ☛ gemm_blk_var5_ws.tex
- ☛ GemmBlkVar1LS.mlx , ☛ GemmBlkVar3LS.mlx , ☛ GemmBlkVar5LS.mlx

**Answer:**
Answers:

- ☛ gemm_blk_var1_ws_answer.tex, ☛ gemm_blk_var3_ws_answer.tex ☛ gemm_blk_var5_ws_answer.tex
- ☛ gemm_blk_var1_ws_answer.tex ☛ gemm_blk_var3_ws_answer.tex ☛ gemm_blk_var5_ws_answer.tex
- ☛ GemmBlkVar1LSAnswer.mlx ☛ GemmBlkVar3LSAnswer.mlx ☛ GemmBlkVar5LSAnswer.mlx

☛ BACK TO TEXT

**Homework 5.3.5.2** If you feel energetic, also derive Blocked Variants 2, 4, and 6. **Answer:**

☛ BACK TO TEXT

**Homework 5.4.2.1** Create a table of all loop invariants for PME 1, disgarding those for which there is no viable loop guard or initialization command. You may want to start with Figure 5.11. The gray text there will help you decide what to include in the loop invariant.

**Answer:** See Figure 5.11.

☛ BACK TO TEXT

**Homework 5.4.3.1** Derive as many unblocked algorithmic variants as you find useful.
Some resources:

- The ☛ blank worksheet.
- ☛ color_flatex.tex.

PME: $\left(\frac{C_T}{C_B}\right) = \left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$.

| $A_{TL}B_T$ | $A_{BL}^T B_B$ | $A_{BL}B_T$ | $A_{BR}B_B$ | $\left(\frac{C_T}{C_B}\right) =$ | |
|---|---|---|---|---|---|
| Yes | No | No | No | $\left(\frac{A_{TL}B_T + \textcolor{lightgray}{A_{BL}^T B_B +} \widehat{C}_T}{\textcolor{lightgray}{A_{BL}B_T + A_{BR}B_B +} \widehat{C}_B}\right)$ | 1 |
| Yes | Yes | No | No | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{\textcolor{lightgray}{A_{BL}B_T + A_{BR}B_B +} \widehat{C}_B}\right)$ | 2 |
| Yes | No | Yes | No | $\left(\frac{A_{TL}B_T + \textcolor{lightgray}{A_{BL}^T B_B +} \widehat{C}_T}{A_{BL}B_T + \textcolor{lightgray}{A_{BR}B_B +} \widehat{C}_B}\right)$ | 3 |
| Yes | Yes | Yes | No | $\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + \textcolor{lightgray}{A_{BR}B_B +} \widehat{C}_B}\right)$ | 4 |
| No | Yes | Yes | Yes | $\left(\frac{\textcolor{lightgray}{A_{TL}B_T +} A_{BL}^T B_B + \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 5 |
| No | Yes | No | Yes | $\left(\frac{\textcolor{lightgray}{A_{TL}B_T +} A_{BL}^T B_B + \widehat{C}_T}{\textcolor{lightgray}{A_{BL}B_T +} A_{BR}B_B + \widehat{C}_B}\right)$ | 6 |
| No | No | Yes | Yes | $\left(\frac{\textcolor{lightgray}{A_{TL}B_T + A_{BL}^T B_B +} \widehat{C}_T}{A_{BL}B_T + A_{BR}B_B + \widehat{C}_B}\right)$ | 7 |
| No | No | No | Yes | $\left(\frac{\textcolor{lightgray}{A_{TL}B_T + A_{BL}^T B_B +} \widehat{C}_T}{\textcolor{lightgray}{A_{BL}B_T +} A_{BR}B_B + \widehat{C}_B}\right)$ | 8 |

Figure 5.11: Loop-invariants for $C := AB + C$ where $A$ is symmetric and only its lower triangular part is stored.

- Spark webpage.
- ☛ symm_l_unb_var1_ws.tex, ☛ symm_l_unb_var2_ws.tex,
  ☛ symm_l_unb_var3_ws.tex, ☛ symm_l_unb_var4_ws.tex,
  ☛ symm_l_unb_var5_ws.tex, ☛ symm_l_unb_var6_ws.tex,
  ☛ symm_l_unb_var7_ws.tex, ☛ symm_l_unb_var8_ws.tex.
- ☛ SymmLUnbVar1LS.mlx, ☛ SymmLUnbVar2LS.mlx,
  ☛ SymmLUnbVar3LS.mlx, ☛ SymmLUnbVar4LS.mlx,
  ☛ SymmLUnbVar5LS.mlx, ☛ SymmLUnbVar6LS.mlx,
  ☛ SymmLUnbVar7LS.mlx, ☛ SymmLUnbVar8LS.mlx.

**Answer:**
Answers:

- ☛ symm_l_unb_var1_ws_answer.tex, ☛ symm_l_unb_var2_ws_answer.tex,
  ☛ symm_l_unb_var3_ws_answer.tex, ☛ symm_l_unb_var4_ws_answer.tex,
  ☛ symm_l_unb_var5_ws_answer.tex, ☛ symm_l_unb_var6_ws_answer.tex,
  ☛ symm_l_unb_var7_ws_answer.tex, ☛ symm_l_unb_var8_ws_answer.tex.
- ☛ SymmLUnbVar1LSAnswer.mlx, ☛ SymmLUnbVar2LSAnswer.mlx,
  ☛ SymmLUnbVar3LSAnswer.mlx, ☛ SymmLUnbVar4LSAnswer.mlx,
  ☛ SymmLUnbVar5LSAnswer.mlx, ☛ SymmLUnbVar6LSAnswer.mlx,
  ☛ SymmLUnbVar7LSAnswer.mlx, ☛ SymmLUnbVar8LSAnswer.mlx.

☛ BACK TO TEXT

**Homework 5.4.5.1** Derive as many blocked algorithmic variants as you find useful.
Some resources:

- The ☛ blank worksheet.
- ☛ color_flatex.tex.
- Spark webpage.
- ☛ symm_l_blk_var1_ws.tex, ☛ symm_l_blk_var2_ws.tex,
  ☛ symm_l_blk_var3_ws.tex, ☛ symm_l_blk_var4_ws.tex,
  ☛ symm_l_blk_var5_ws.tex, ☛ symm_l_blk_var6_ws.tex,
  ☛ symm_l_blk_var7_ws.tex, ☛ symm_l_blk_var8_ws.tex.
- ☛ SymmLBlkVar1LS.mlx, ☛ SymmLBlkVar2LS.mlx,
  ☛ SymmLBlkVar3LS.mlx, ☛ SymmLBlkVar4LS.mlx,
  (The rest of these are not yet available.)
  ☛ SymmLBlkVar5LS.mlx, ☛ SymmLBlkVar6LS.mlx,
  ☛ SymmLBlkVar7LS.mlx, ☛ SymmLBlkVar8LS.mlx.

**Answer:**
Answers:

- ☛ symm_l_blk_var1_ws_answer.tex, ☛ symm_l_blk_var2_ws_answer.tex,
  ☛ symm_l_blk_var3_ws_answer.tex, ☛ symm_l_blk_var4_ws_answer.tex,
  ☛ symm_l_blk_var5_ws_answer.tex, ☛ symm_l_blk_var6_ws_answer.tex,
  ☛ symm_l_blk_var7_ws_answer.tex, ☛ symm_l_blk_var8_ws_answer.tex.

- ☛ `SymmLBlkVar1LSAnswer.mlx`, ☛ `SymmLBlkVar2LSAnswer.mlx`, ☛ `SymmLBlkVar3LSAnswer.mlx`, ☛ `SymmLBlkVar4LSAnswer.mlx`, ☛ `SymmLBlkVar5LSAnswer.mlx`, ☛ `SymmLBlkVar6LSAnswer.mlx`, ☛ `SymmLBlkVar7LSAnswer.mlx`, ☛ `SymmLBlkVar8LSAnswer.mlx`.

☛ BACK TO TEXT

**Homework 6.2.2.1** Derive the PME from the postcondition and how matrices $L$ and $U$ inherently need to be partitioned.

**Answer:**

Inserting the partitioned $L$ and $U$ into the postcondition $A = L\backslash U \wedge LU = \widehat{A}$ guides us to how $A$ should be partitioned:

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right).$$

This then yields

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & L\backslash U_{BR} \end{array}\right)$$
$$\wedge \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \widehat{A}_{TL} & \widehat{A}_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right).$$

Multiplying the partitioned $L$ and $U$ then gives us the PME:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & L\backslash U_{BR} \end{array}\right) \wedge \begin{array}{c|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & L_{BL}U_{TR} + L_{BR}U_{BR} = \widehat{A}_{BR} \end{array}.$$

☛ BACK TO TEXT

**Homework 6.2.3.1** Derive and implement the unblocked algorithm that corresponds to

$$\text{Invariant 1} : \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge L_{TL}U_{TL} = \widehat{A}_{TL}.$$

If you get stuck, there are hints in the below videos.

Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `LU_unb_var1_ws.tex`.
  This worksheet is already partially filled out (through Step 5).
- If you end up using the first video that follows this homework as a hint, then you may want to continue after watching that with the following worksheet: ☛ `LU_unb_var1_ws_step6.tex`.
  This worksheet is filled out through Step 6.
- ☛ `LUUnbVar1LS.mlx`.
  **Note:** for the implementation, you don't need to include L and U as parameters. They were "temporaries" in the derivation, but don't show up in the actual implementation. This same comment holds for all implementations of LU factorization.

**Answer:**

View the below video and/or the following answer:

- ☛ `LU_unb_var1_ws.tex`
- ☛ `LU_unb_var1_ws_answer.pdf`
- ☛ `LUUnbVar1LSAnswer.mlx`.

☛ BACK TO TEXT

**Homework 6.2.4.1** Derive and implement the unblocked algorithm that corresponds to

$$\text{Invariant 2}\ :\ \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge \begin{array}{c} L_{TL}U_{TL} = \widehat{A}_{TL} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL}. \end{array}$$

Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `LU_unb_var2_ws.tex`.
  This worksheet is already partially filled out (through Step 5).
- ☛ `LUUnbVar2LS.mlx`.

**Answer:**

- ☛ `LU_unb_var2_ws_answer.tex`
- ☛ `LU_unb_var2_ws_answer.pdf`
- ☛ `LUUnbVar2LSAnswer.mlx`.

☛ BACK TO TEXT

**Homework 6.2.4.2** Identify three additional loop invariants (Invariants 3-5) for computing the LU factorization.

**Answer:** (See also the video right after this homework.)

Following the example of how Invariant 2 was derived, two additional loop invariant are relatively easy to identify:

$$\text{Invariant 3}\ :\ \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & U_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge \begin{array}{c|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR}, \end{array}$$

where

$$\begin{array}{c|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \end{array}$$

captures the constraint

$$L_{TL}U_{TL} = \widehat{A}_{TL} \wedge L_{TL}U_{TR} = \widehat{A}_{TR},$$

and

$$\text{Invariant 4}\ :\ \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge \begin{array}{c|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TR} = \widehat{A}_{BL}, & \end{array}$$

| | |
|---|---|
| Invariant 1 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge L_{TL}U_{TL} = \widehat{A}_{TL}$ |
| Invariant 2 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & \widehat{A}_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge \begin{array}{c} L_{TL}U_{TL} = \widehat{A}_{TL} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL}. \end{array}$ |
| Invariant 3 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & U_{TR} \\ \hline \widehat{A}_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge \begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \end{array}$ |
| Invariant 4 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge \begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}$ |
| Invariant 5 : | $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right) \wedge \begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}$ |

Figure 6.12: The five loop invariants for computing the LU factorization.

where

$$\begin{array}{c|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TR} = \widehat{A}_{BL} & \end{array}$$

captures the constraint

$$L_{TL}U_{TL} = \widehat{A}_{TL} \wedge L_{TL}U_{TR} = \widehat{A}_{TR} \wedge L_{BL}U_{TR} = \widehat{A}_{BL}.$$

The fifth loop invariant is a little harder to justify. We notice that the PME includes the constraint $L_{BL}U_{TR} + L_{BR}U_{BR} = \widehat{A}_{BR}$ which can be rewritten as $L_{BL}U_{TR} = \widehat{A}_{BR} - L_{BR}U_{BR}$. Now, **if** $L_{BR}$ and $U_{TR}$ have already been computed, then we can also required $A_{BR}$ to have been updated by subtracting $L_{BL}U_{TR}$ yielding

$$\text{Invariant 5 :} \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right) \wedge \begin{array}{c|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TR} = \widehat{A}_{BL}. & \end{array}$$

Figure 6.12 gives a table with the five loop invariants.

☛ BACK TO TEXT

**Homework 6.2.4.3** Derive and implement the unblocked algorithms that correspond to Invariants 3-5 from the last homework. If you have limited time, then derive at least the algorithm corresponding to Invariant 5.

Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `LU_unb_var3_ws.tex`, ☛ `LU_unb_var4_ws.tex`, ☛ `LU_unb_var5_ws.tex`.
  These worksheets are already partially filled out (through Step 5).
- ☛ `LUUnbVar3LS.mlx`, ☛ `LUUnbVar4LS.mlx`, ☛ `LUUnbVar5LS.mlx`.

**Answer:**

- ☛ LU_unb_var3_ws_answer.tex, ☛ LU_unb_var4_ws_answer.tex, ☛ LU_unb_var5_ws_answer.tex
- ☛ LU_unb_var3_ws_answer.pdf, ☛ LU_unb_var4_ws_answer.pdf. ☛ LU_unb_var5_ws_answer.pdf
- ☛ LUUnbVar3LSAnswer.mlx, ☛ LUUnbVar4LSAnswer.mlx, ☛ LUUnbVar5LSAnswer.mlx

☛ BACK TO TEXT

**Homework 6.2.5.1** Derive and implement the blocked algorithm that corresponds to

Invariant 5 :

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{TL} & \widehat{U}_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right) \wedge \begin{array}{c|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL}. & \end{array}$$

If you need hints along the way, you may want to watch the videos that follow this homework.

Some resources:

- The ☛ blank worksheet.
- ☛ color_flatex.tex.
- Spark webpage.
- ☛ LU_blk_var5_ws.tex.
- ☛ LUBlkVar5LS.mlx.

**Answer:**

- ☛ LU_blk_var5_ws_answer.tex.
- ☛ LU_blk_var5_ws_answer.pdf.
- ☛ LUBlkVar5LSAnswer.mlx

☛ BACK TO TEXT

**Homework 6.2.5.2** Derive and implement the blocked algorithms that correspond to Invariants 1-4. If you have limited time, then you may want to focus on Invariant 2. A table of all loop invariants was given in the solution for Homework 6.2.4.2.

Some resources:

- The ☛ blank worksheet.
- ☛ color_flatex.tex.
- Spark webpage.
- ☛ LU_blk_var1_ws.tex, ☛ LU_blk_var2_ws.tex, ☛ LU_blk_var3_ws.tex, ☛ LU_blk_var4_ws.tex.
- ☛ LUBlkVar1LS.mlx, ☛ LUBlkVar2LS.mlx, ☛ LUBlkVar3LS.mlx, ☛ LUBlkVar4LS.mlx.

**Answer:**

- ☛ LU_blk_var1_ws_answer.tex, ☛ LU_blk_var2_ws_answer.tex, ☛ LU_blk_var3_ws_answer.tex, ☛ LU_blk_var4_ws_answe

- ☛ LU_blk_var1_ws_answer.pdf, ☛ LU_blk_var2_ws_answer.pdf, ☛ LU_blk_var3_ws_answer.pdf. ☛ LU_blk_var4_ws_answ
- ☛ LUBlkVar1LSAnswer.mlx, ☛ LUBlkVar2LSAnswer.mlx, ☛ LUBlkVar3LSAnswer.mlx, ☛ LUBlkVar4LSAnswer.mlx

☛ BACK TO TEXT

**Homework 6.3.1.1** Derive the PME for solving $Lx = y$, overwriting $y$ with the solution, where $L$ is **unit lower triangular** and $y$ is stored as a column vector. Next, derive and implement unblocked algorithms. You will want to find an algorithm that accesses $L$ by columns. Can you already tell from the loop invariant which algorithm will have that property, so that you only have to derive one?

We use `trsv_lnu` and `TrsvLNU` to indicate that the **tr**iangular **s**ol**v**e involves the **l**ower triangular part of the matrix, that the matrix stored there is **n**ot to be transposed, and that matrix is a **u**nit triangular matrix.

Some resources:

- The ☛ blank worksheet.
- ☛ color_flatex.tex.
- Spark webpage.
- ☛ trsv_lnu_unb_var1_ws.tex, ☛ trsv_lnu_unb_var2_ws.tex
- ☛ TrsvLNUUnbVar1LS.mlx, ☛ TrsvLNUUnbVar2LS.mlx
  **Note:** for the implementation, you don't need to include $x$ as a parameter. It was a "temporary" in the derivation, but doesn't show up in the actual implementation. This same comment holds for all implementations of triangular solve.

**Answer:**

The postcondition can be given as

$$y = x \wedge Lx = \hat{y}.$$

From this, one can derive the PME

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \wedge \begin{array}{r} L_{TL}x_T = \hat{y}_T \\ \hline L_{BL}x_T + L_{BR}x_B = \hat{y}_B. \end{array}$$

This, in turn, yields two loop invariants:

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{y}_B \end{array}\right) \wedge L_{TL}x_T = \hat{y}_T$$

and

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{y}_B - L_{BL}x_T \end{array}\right) \wedge L_{TL}x_T = \hat{y}_T$$

Here you need to recognize that inherently $x_T$ must be computed before $x_B$.

These observations now allow you to derive the various algorithms:

- ☛ trsv_lnu_unb_var1_ws_answer.tex, ☛ trsv_lnu_unb_var2_ws_answer.tex
- ☛ trsv_lnu_unb_var1_ws_answer.pdf, ☛ trsv_lnu_unb_var2_ws_answer.pdf
- ☛ TrsvLNUUnbVar1LSAnswer.mlx, ☛ TrsvLNUUnbVar2LSAnswer.mlx

☛ BACK TO TEXT

**Homework 6.3.1.2** Derive the PME for solving $Ux = y$, overwriting $y$ with the solution, where $U$ is upper triangular and $y$ is stored as a column vector. In what direction should you march through the matrix and vectors? (The PME should tell you.) Derive and implement unblocked algorithms. You will want to find an algorithm that accesses $U$ by columns. Can you already tell from the loop invariant which algorithm will have that property, so that you only have to derive one?

We use `trsv_unn` and `TrsvUNN` to indicate that the **tr**iangular **s**ol**v**e involves the **u**pper triangular part of the matrix, that the matrix stored there is **n**ot to be transposed, and that matrix is **n**ot a unit triangular matrix.

Some resources:

- The ☛ blank worksheet.
- ☛ `color_flatex.tex`.
- Spark webpage.
- ☛ `trsv_unn_unb_var1_ws.tex`, ☛ `trsv_unn_unb_var2_ws.tex`
- ☛ `TrsvUNNUnbVar1LS.mlx`, ☛ `TrsvUNNUnbVar2LS.mlx`
  **Note:** for the implementation, you don't need to include $x$ as a parameter. It was a "temporary" in the derivation, but doesn't show up in the actual implementation. This same comment holds for all implementations of triangular solve.

**Answer:**

The postcondition can be given as

$$y = x \wedge Ux = \widehat{y}.$$

From this, one can derive the PME

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \wedge \begin{array}{r} U_{TL}x_T + U_{TR}x_B = \widehat{y}_T \\ \hline U_{BR}x_B = \widehat{y}_B. \end{array}$$

This, in turn, yields two loop invariants:

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} \widehat{y}_T \\ \hline x_B \end{array}\right) \wedge\ U_{BR}x_B = \widehat{y}_B$$

and

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} \widehat{y}_T - U_{TR}x_B \\ \hline x_B \end{array}\right) \wedge\ U_{BR}x_B = \widehat{y}_B$$

Here you need to recognize that inherently $x_B$ must be computed before $x_T$.

- ☛ `trsv_unn_unb_var1_ws_answer.tex`, ☛ `trsv_unn_unb_var2_ws_answer.tex`
- ☛ `trsv_unn_unb_var1_ws_answer.pdf`, ☛ `trsv_unn_unb_var2_ws_answer.pdf`
- ☛ `TrsvUNNUnbVar1LSAnswer.mlx`, ☛ `TrsvUNNUnbVar2LSAnswer.mlx`

☛ BACK TO TEXT

**Homework 6.3.2.1** Derive an alternative PME that corresponds to partitioning $L$ into quadrants. You are on your own: derive the invariants, the algorithms, the implementations. If you still have energy left after that, do the same for solving $UX = B$ or $L^TX = B$ ore $U^TX = B$ (without explicitly transposing $L$ or $U$). You are now an expert!

☛ BACK TO TEXT

# Index