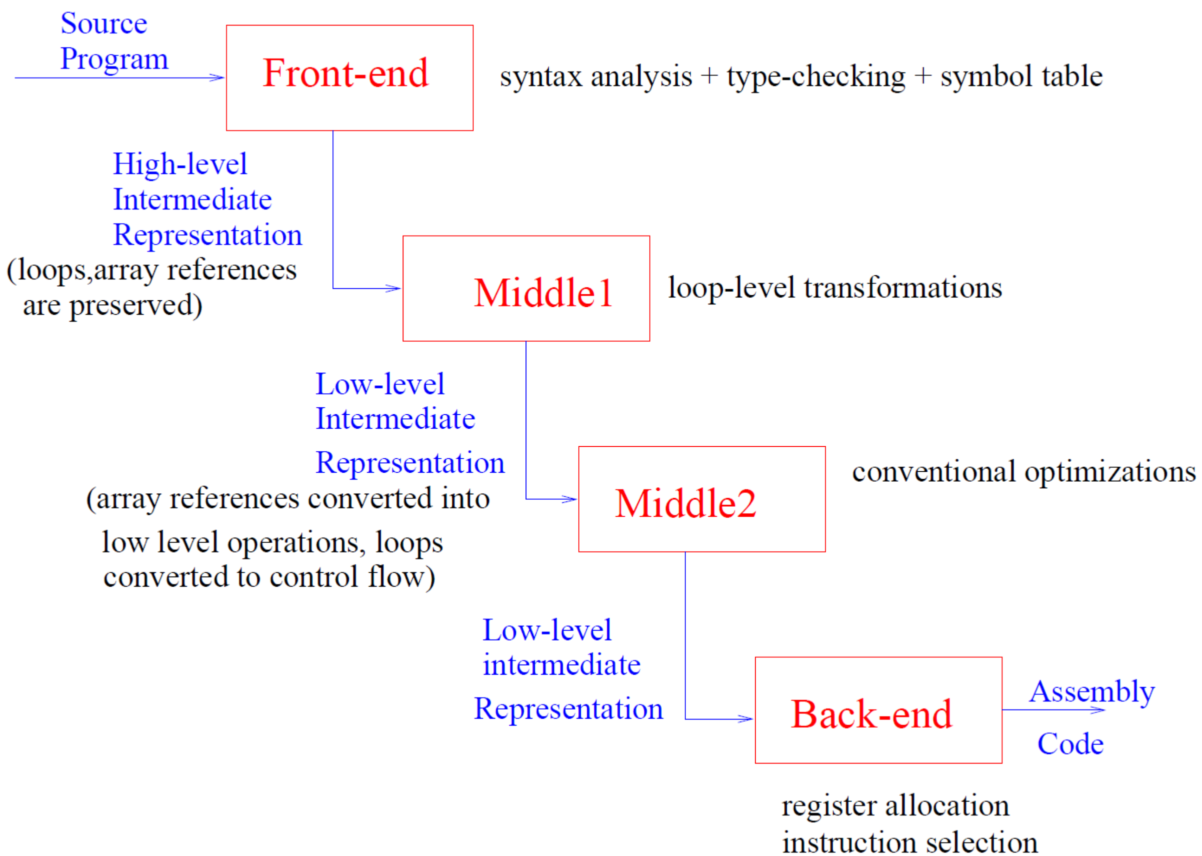
Dependences
and
Loop Transformations

Organization of a Modern Compiler



Key concepts:

Perfectly-nested loop: Loop nest in which all assignment statements occur in body of innermost loop.

```
for J = 1, N
  for I = 1, N
    Y(I) = Y(I) + A(I,J)*X(J)
```

Imperfectly-nested loop: Loop nest in which some assignment statements occur within some but not all loops of loop nest

```
  for k = 1, N
    a(k,k) = sqrt (a(k,k))
    for i = k+1, N
      a(i,k) = a(i,k) / a(k,k)
    for i = k+1, N
      for j = k+1, i
        a(i,j) -= a(i,k) * a(j,k)
```

Our focus: perfectly-nested loops

Overview of lecture

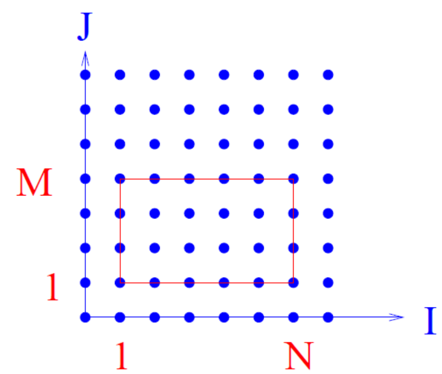
- We have seen two loop transformations for locality enhancement
  - Permutation
  - Tiling
- Many other transformations
  - Skewing, reveral, scaling…
- Code generation: given a loop nest and a transformation,
  - Determine if the transformation is legal (does not violate dependences).
  - If so, generate the transformed loop nest.
- More difficult problem: synthesis of transformation
  - Given a loop nest and a performance objective such as locality enhancement, synthesize a good transformation.

Iteration Space of a Perfectly-nested Loop

Each iteration of a loop nest with n loops can be viewed as an integer point in an n-dimensional space.

Iteration space of loop: all points in n-dimensional space corresponding to loop iterations

```
FOR  I = 1, N
  FOR   J = 1,M
        S
```

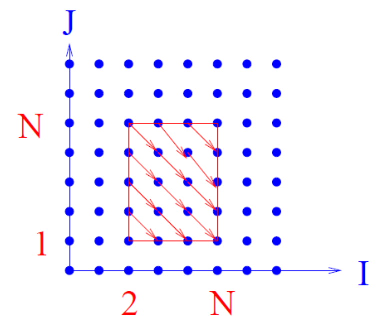


Execution order = lexicographic order on iteration space:

$$(1,1) \preceq (1,2) \preceq ... \preceq (1,M) \preceq (2,1) \preceq (2,2)... \preceq (N,M)$$

Issue 1: (example) : loop permutation may be illegal in some loop nests

```
FOR  I = 2, N
 FOR  J = 1, M
    A[I,J] = A[I-1,J+1] + 1
```



Assume that array has 1's stored everywhere before loop begins.

After loop permutation:

```
FOR   J = 1, M
 FOR   I = 2, N
    A[I,J] = A[I-1,J+1] + 1
```

Transformed loop will produce different values (A[3,1] for example)

=> permutation is illegal for this loop.

Question: How do we determine when loop permutation is legal?

Subtle issue 2: generating code for transformed loop nest may be non-trivial!

Example: triangular loop bounds (triangular solve/Cholesky)

```
FOR I = 1, N
  FOR J = 1, I-1
     S
```

Here, inner loop bounds are functions of outer loop indices!

Just exchanging the two loops will not generate correct bounds.

History

- Dependence and code generation problems used to be solved using heuristics (1965-1990)
  - GCD test, Banerjee test, etc.
  - Pattern matching to generate code
- Today we use powerful integer linear programming (ILP) techniques (1990-)
  - Complemented by heuristics to quickly handle simple problems
  - Use full-blown power of ILP calculator very sparingly

Integer Linear Programming

Two problems:

Given a system of linear inequalities $Ax \leq b$

where A is a m X n matrix of integers,

b is an m vector of integers,

x is an n vector of unknowns,

(i) Are there integer solutions?

(ii) Enumerate all integer solutions.

Most problems regarding correctness of transformations and code generation can be reduced to these problems.

Linear inequalities

$$\begin{bmatrix} 2 & 3 & 4 \\ 1 & -1 & 3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leq \begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

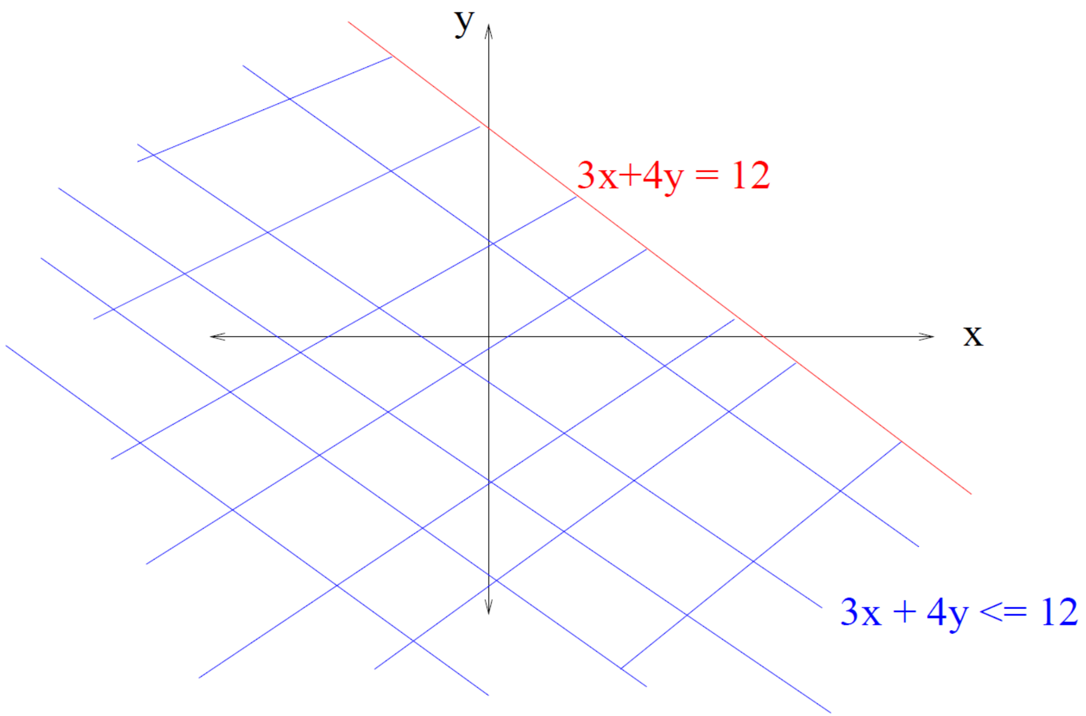
is equivalent to

$$2x + 3y + 4z \leq 4$$
$$x - y + 3z \leq 1$$

Intuition about systems of linear inequalities:

Equality: line (2D), plane (3D), hyperplane (> 3D)

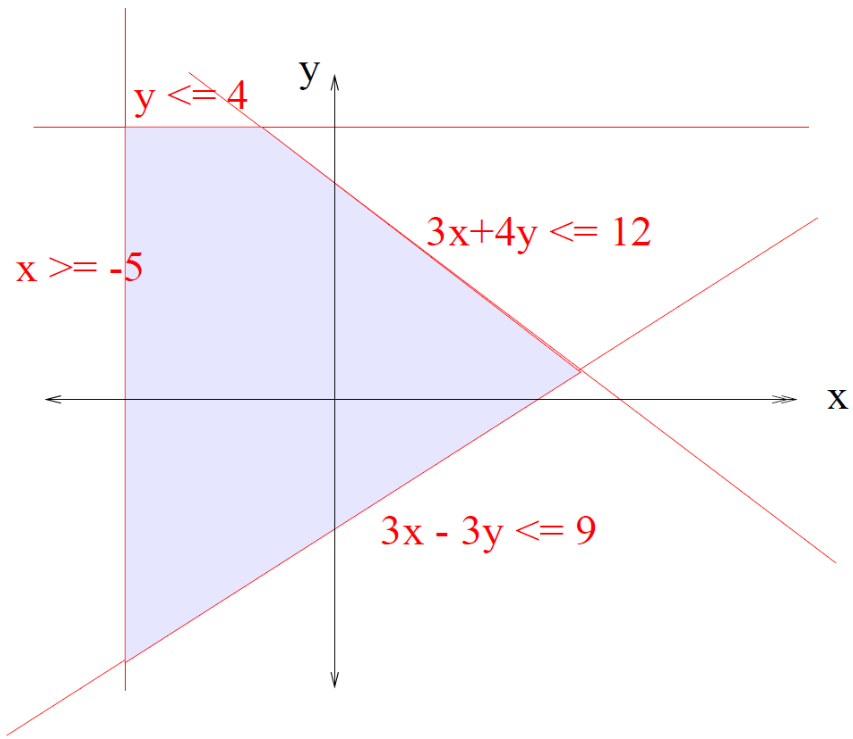
Inequality: half-plane (2D), half-space(>2D)



Region described by inequality is convex
(if two points are in region, all points in between them are in region)

Intuition about systems of linear inequalities:

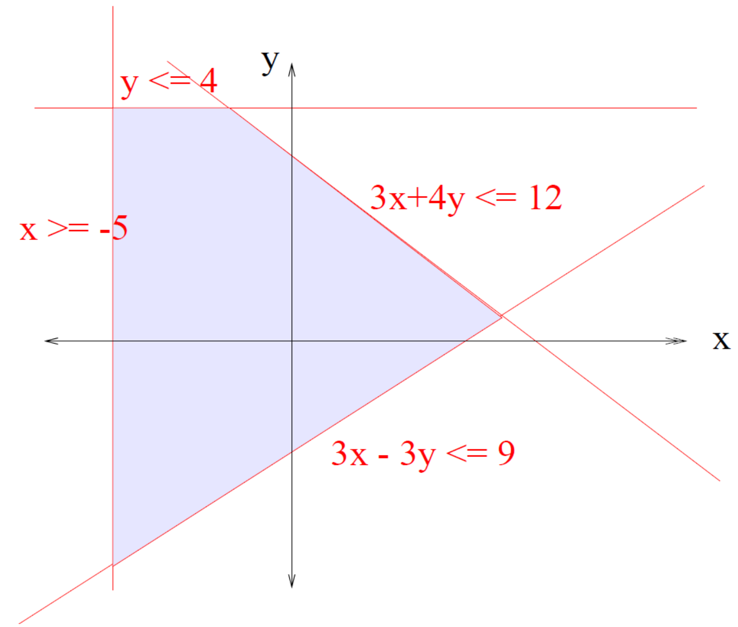
Conjunction of inequalties = intersection of half-spaces
=> some convex region



Region described by inequalities is a convex polyhedron
(if two points are in region, all points in between them are in region)

Geometric intuition for ILP problems

- Given a convex polyhedron, solve two problems.
- Enumerate all the integer points in polyhedron
  - Fourier-Motzkin elimination
  - Used to generate code for transformed loop nest
- Decision problem: is there an integer point within the polyhedron?
  - Cutting plane method (Gomory 1958)
  - Used to determine if transformation is legal
- In compilers, we deal with underdetermined systems
  - No solution or many solutions



Fourier-Motzkin Elimination

Running example:

$$3x + 4y \geq 16 \tag{1}$$

$$4x + 7y \leq 56 \tag{2}$$

$$4x - 7y \leq 20 \tag{3}$$

$$2x - 3y \geq -9 \tag{4}$$

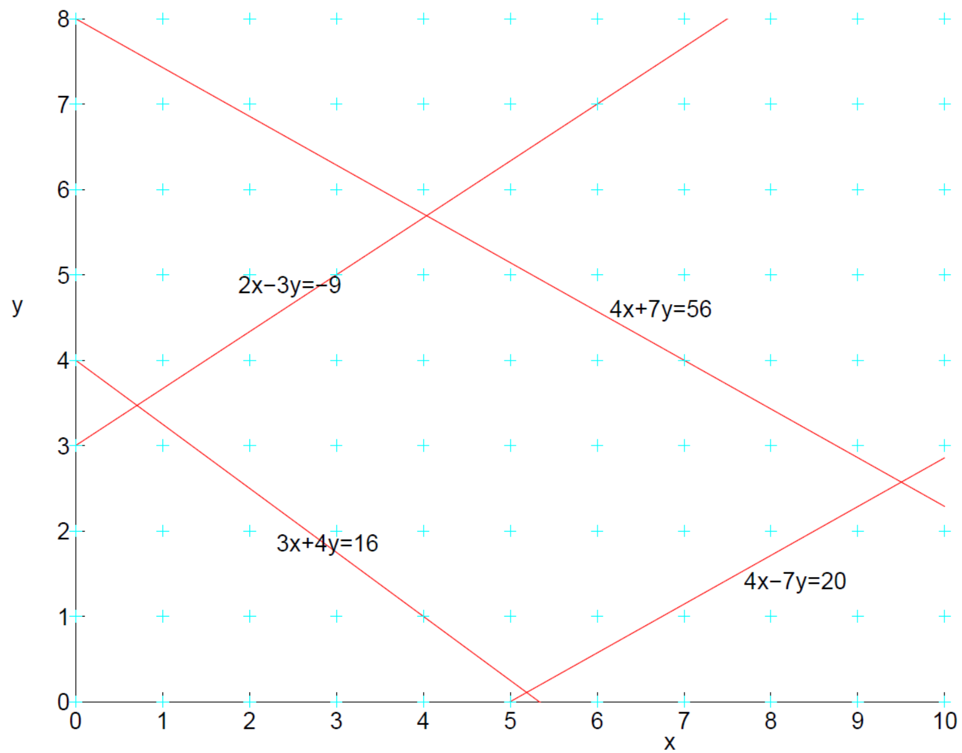
Upper bounds for x: (2) and (3)

Lower bounds for x: (1) and (4)

Upper bounds for y: (2) and (4)

Lower bounds for y: (1) and (3)

MATLAB graphs:



Code for enumerating integer points in polyhedron: (see graph)

Outer loop: Y, Inner loop: X

DO Y= $\lceil 4/37 \rceil$, $\lfloor 74/13 \rfloor$

  DO X= $\lceil max(16/3 - 4y/3, -9/2 + 3y/2) \rceil$, $\lfloor min(5 + 7y/4, 14 - 7y/4) \rfloor$

   .....

Outer loop: X, Inner loop: Y

DO X=1, 9

 DO Y= $\lceil max(4 - 3y/4, (4x - 20)/7) \rceil$, $\lfloor (min(8 - 4x/5, (2x + 9)/3) \rfloor$

   .....

How do we can determine loop bounds?

Fourier-Motzkin elimination: variable elimination technique for inequalities

$$3x + 4y \geq 16 \tag{5}$$
$$4x + 7y \leq 56 \tag{6}$$
$$4x - 7y \leq 20 \tag{7}$$
$$2x - 3y \geq -9 \tag{8}$$

Let us project out x.
First, express all inequalities as upper or lower bounds on x.

$$x \geq 16/3 - 4y/3 \tag{9}$$
$$x \leq 14 - 7y/4 \tag{10}$$
$$x \leq 5 + 7y/4 \tag{11}$$
$$x \geq -9/2 + 3y/2 \tag{12}$$

For any y, if there is an x that satisfies all inequalities, then every lower bound on x must be less than or equal to every upper bound on x.

Generate a new system of inequalities from each pair (upper,lower) bounds.

$$\begin{aligned}
5 + 7y/4 &\geq 16/3 - 4y/3 (\text{Inequalities} 3, 1) \\
5 + 7y/4 &\geq -9/2 + 3y/2 (\text{Inequalities} 3, 4) \\
14 - 7y/4 &\geq 16/3 - 4y/3 (\text{Inequalities} 2, 1) \\
14 - 7y/4 &\geq -9/2 + 3y/2 (\text{Inequalities} 2, 4)
\end{aligned}$$

Simplify:

$$\begin{aligned} y &\geq 4/37 \\ y &\geq -38 \\ y &\leq 104/5 \\ y &\leq 74/13 \end{aligned}$$

=>

$$max(4/37, -38) \quad \leq \quad y \leq min(104/5, 74/13)$$

$$=>$$

$$4/37 \quad \leq y \quad \leq 74/13$$

This means there are rational solutions to original system of inequalities.

We can now express solutions in closed form as follows:

$$\begin{aligned} 4/37 &\leq y \leq 74/13 \\ max(16/3 - 4y/3, -9/2 + 3y/2) &\leq x \leq min(5 + 7y/4, 14 - 7y/4) \end{aligned}$$

Fourier-Motzkin elimination: iterative algorithm

Iterative step:

- obtain reduced system by projecting out a variable
- if reduced system has a rational solution, so does the original

Termination: no variables left

Projection along variable x: Divide inequalities into three categories

$$a_1 * y + a_2 * z + ... \leq c_1 (no\ x)$$
$$b_1 * x \leq c_2 + b_2 * y + b_3 * z + ...(upper\ bound)$$
$$d_1 * x \geq c_3 + d_2 * y + d_3 * z + ...(lower\ bound)$$

New system of inequalities:

- All inequalities that do not involve x
- Each pair (lower,upper) bounds gives rise to one inequality:

$$b_1[c_3 + d_2 * y + d_3 * z + ...] \leq d_1[c_2 + b_2 * y + b_3 * z + ...]$$

Enumeration: Given a system $Ax \leq b$, we can use Fourier-Motzkin elimination to generate a loop nest to enumerate all integer points that satisfy system as follows:

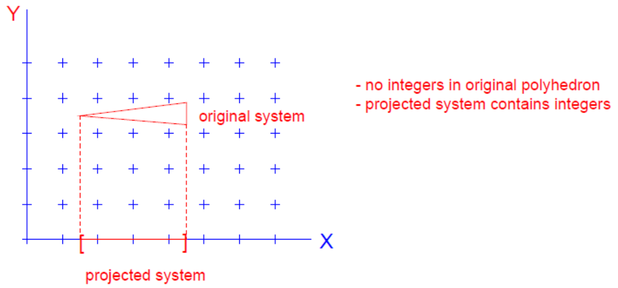
- pick an order to eliminate variables (this will be the order of variables from innermost loop to outermost loop)
- eliminate variables in that order to generate upper and lower bounds for loops as shown in theorem in previous slide

Remark: if polyhedron has no integer points, then the lower bound of some loop in the loop nest will be bigger than the upper bound of that loop

What can we conclude about integer solutions?

Corollary: If reduced system has no integer solutions, neither does the original system.

Not true: Reduced system has integer solutions => original system does too.

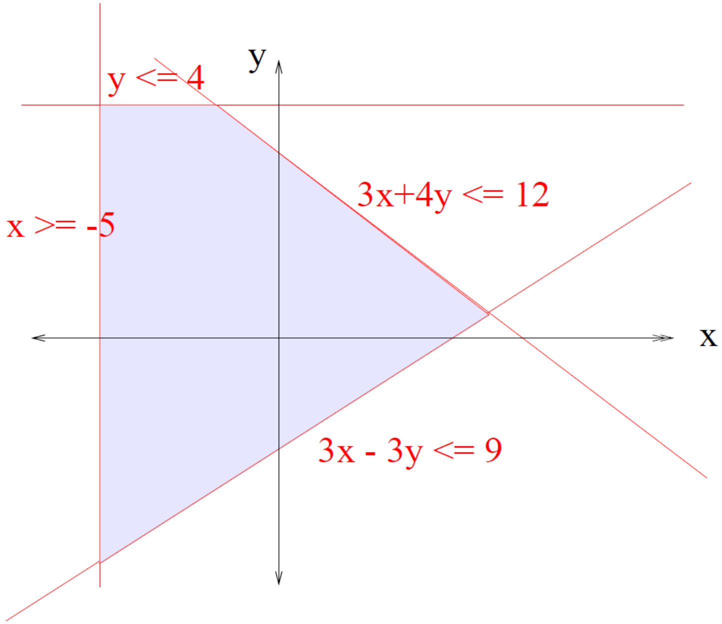


Key problem: Multiplying one inequality by b_1 and other by d_1 is not guaranteed to preserve "integrality" (cf. equalities)

Exact projection: If all upper bound coefficients b_i or all lower bound coefficients d_i happen to be 1, then integer solution to reduced system implies integer solution to original system.

Solving decision problem

- ILP calculators: variations of cutting plane method (Gomory 1958)
- In practice, we use simple tests like GCD test to handle easy cases
  - (e.g.) equation 2x + 4y = 5 does not have integer solutions because lhs must be an even number for any integer values of x and y but rhs is an odd number
  - Generalization: if GCD of lhs coefficients does not divide rhs, equation has no solutions
  - Given a system of equalities and inequalities, use GCD test to quickly determine if some equality has no solution; otherwise use ILP calculator
- Dependence tests (1965-1990)
  - Cottage industry in inventing more general tests than GCD



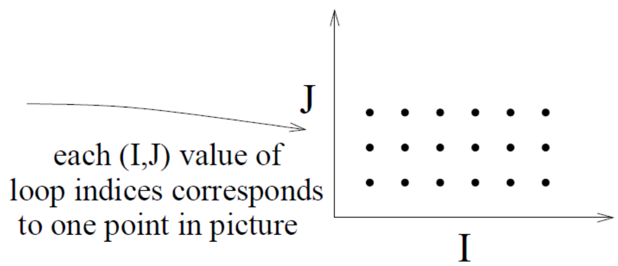
Treatment of equalities

- In principle, we need to consider only inequalities
  - Convert equality (A = B) to conjunction of two inequalities ($A \leq B$) and ($B \leq A$)
- Better approach for decision problem
  - Solve equalities first using integer Gaussian elimination to find parametric solution (remember: underdetermined systems)
  - Substitute parametric solution into inequalities and then solve system of inequalities using cutting plane method
- Integer Gaussian elimination
  - Rich theory for solving Diophantine equations going back more than 2000 years (Greeks, Hindus)

Using ILP for Dependence Analysis

Loop level Analysis: granularity is a loop iteration

```
FOR  I = 1, 100
  FOR  J = 1, 100
    S
```



Dynamic instance of a statement:

Execution of a statement for given loop index values

Dependence between iterations:

Iteration (I1,J1) is said to be dependent on iteration (I2,J2) if a dynamic instance (I1,J1) of a statement in loop body is dependent on a dynamic instance (I2,J2) of a statement in the loop body.

How do we compute dependences between iterations of a loop nest?

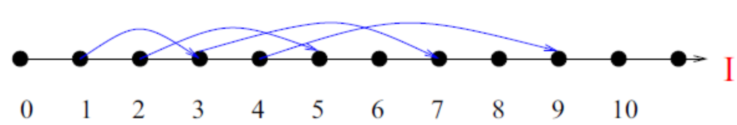
Dependence Example

Consider single loop case first:

```
FOR I = 1, 100
  X(2I+1) = ....X(I)...
```

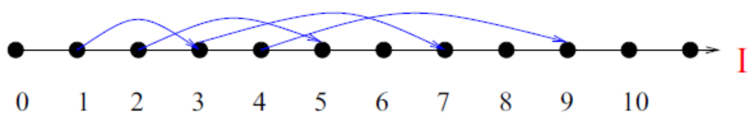
```
Flow dependences between iterations:
   Iteration 1 writes to X(3) which is read by iteration 3.
   Iteration 2 writes to X(5) which is read by iteration 5.
   ....
   Iteration 49 writes to X(99) which is read by iteration 99.
```

If we ignore the array locations and just think about dependence between iterations, we can draw this geometrically as follows:



Dependence arrows always go forward in iteration space. (eg. there cannot be a dependence from iteration 5 to iteration 2)

Intuitively, dependence arrows tell us constraints on transformations.



Suppose a transformed program does iteration 2 before iteration 1. OK!

Transformed program does iteration 3 before iteration 1. Illegal!

Dependences in loops

```
FOR 10 I = 1, N
  X(f(I)) = ...
10        = ...X(g(I))..
```

- Conditions for flow dependence from iteration I_w to I_r:
  - $1 \le I_w \le I_r \le N$ *(write before read)*
  - $f(I_w) = g(I_r)$ *(same array location)*
- Conditions for anti-dependence from iteration I_g to I_o:
  - $1 \le I_g < I_o \le N$ *(read before write)*
  - $f(I_o) = g(I_g)$ *(same array location)*
- Conditions for output dependence from iteration I_{w1} to I_{w2}:
  - $1 \le I_{w1} < I_{w2} \le N$ *(write in program order)*
  - $f(I_{w1}) = f(I_{w2})$ *(same array location)*

Dependences in nested loops

```
FOR 10 I = 1, 100
 FOR 10 J = 1, 200
  X(f(I,J),g(I,J)) = ...
10          = ...X(h(I,J),k(I,J))..
```

Conditions for flow dependence from iteration (I_w, J_w) to (I_r, J_r):
Recall: $\preceq$ is the lexicographic order on iterations of nested loops.

$$\begin{aligned}
1 &\leq I_w \leq 100 \\
1 &\leq J_w \leq 200 \\
1 &\leq I_r \leq 100 \\
1 &\leq J_r \leq 200 \\
(I_w, J_w) &\preceq (I_r, J_r) \\
f(I_w, J_w) &= h(I_r, J_r) \\
g(I_w, J_w) &= k(I_r, J_r)
\end{aligned}$$

Connection to ILP

- Dependence problem becomes an ILP problem if
  - Array subscripts are affine functions of loop variables
  - Loop bounds are affine functions of outer loop variables
- Examples: "regular programs"
  - Most dense linear algebra algorithms like BLAS routines, Cholesky factorization, LU without pivoting
  - Stencil codes: finite differences
- Counter-examples: "irregular programs"
  - Dense linear algebra with pivoting
  - Spare matrix codes, graph algorithms

ILP Formulation

```
FOR I = 1, 100
  X(2I) = .... X(2I+1)...
```

Is there a flow dependence between different iterations?

$$\begin{aligned} 1 &\leq Iw < Ir \leq 100 \\ 2Iw &= 2Ir + 1 \end{aligned}$$

which can be written as

$$\begin{aligned} 1 &\leq Iw \\ Iw &\leq Ir - 1 \\ Ir &\leq 100 \\ 2Iw &\leq 2Ir + 1 \\ 2Ir + 1 &\leq 2Iw \end{aligned}$$

The system

$$
\begin{aligned}
1 &\leq Iw \\
Iw &\leq Ir - 1 \\
Ir &\leq 100 \\
2Iw &\leq 2Ir + 1 \\
2Ir + 1 &\leq 2Iw
\end{aligned}
$$

can be expressed in the form $Ax \leq b$ as follows

$$
\begin{pmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \\ 2 & -2 \\ -2 & 2 \end{pmatrix} \begin{bmatrix} Iw \\ Ir \end{bmatrix} \leq \begin{bmatrix} -1 \\ -1 \\ 100 \\ 1 \\ -1 \end{bmatrix}
$$

ILP Formulation for Nested Loops

```
FOR I = 1, 100
 FOR J = 1, 100
   X(I,J) = ..X(I-1,J+1)...
```

Is there a flow dependence between different iterations?

$$
\begin{aligned}
1 &\leq Iw \leq 100 \\
1 &\leq Ir \leq 100 \\
1 &\leq Jw \leq 100 \\
1 &\leq Jr \leq 100 \\
(Iw, Jw) &\prec (Ir, Jr)(lexicographic\ order) \\
Ir - 1 &= Iw \\
Jr + 1 &= Jw
\end{aligned}
$$

Convert lexicographic order $\prec$ into integer equalities/inequalities.

$(Iw, Jw) \prec (Ir, Jr)$ is equivalent to

$Iw < Ir$ OR $((Iw = Ir)\ AND\ (Jw < Jr))$

We end up with two systems of inequalities:

$$\begin{array}{ccc}
\begin{array}{l}
1 \leq Iw \leq 100 \\
1 \leq Ir \leq 100 \\
1 \leq Jw \leq 100 \\
1 \leq Jr \leq 100 \\
Iw < Ir \\
Ir - 1 = Iw \\
Jr + 1 = Jw
\end{array}
& \quad OR \quad &
\begin{array}{l}
1 \leq Iw \leq 100 \\
1 \leq Ir \leq 100 \\
1 \leq Jw \leq 100 \\
1 \leq Jr \leq 100 \\
Iw = Ir \\
Jw < Jr \\
Ir - 1 = Iw \\
Jr + 1 = Jw
\end{array}
\end{array}$$

Dependence exists if either system has a solution.

What about affine loop bounds?

```
FOR I = 1, 100
  FOR J = 1, I
     X(I,J) = ..X(I-1,J+1)...
```

$$
\begin{aligned}
1 &\leq Iw \leq 100 \\
1 &\leq Ir \leq 100 \\
1 &\leq Jw \leq Iw \\
1 &\leq Jr \leq Ir \\
(Iw, Jw) &\prec (Ir, Jr)(lexicographicorder) \\
Ir - 1 &= Iw \\
Jr + 1 &= Jw
\end{aligned}
$$

Summary

- Problem of determining whether a dependence exists between two iterations of a loop nest can be framed as an ILP problem
  - Assumptions: affine loop bounds and array subscripts
- Solve decision problem using ILP calculator augmented with simpler tests like GCD to filter out easy cases

Dependence Relation
and
Dependence Abstractions

Overview

- Dependence relation
  - Closed form expression that gives all dependences for a given loop nest, not just a yes/no answer for existence of dependence
  - Can be computed using ILP calculator
  - Too expensive to compute for most programs
- Dependence abstractions
  - Distance vectors
  - Direction vectors
  - Dependence matrix

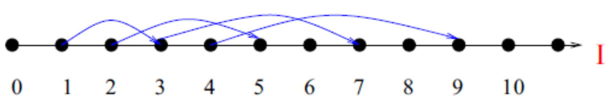
Formal view of dependence

Formal view of a dependence: relation between points in the iteration space.

```
FOR I = 1, 100
  X(2I+1) = ....X(I)...
```

Flow dependence $= \{(Iw, 2Iw+1) | 1 \leq Iw \leq 49\}$

(Note: this is a convex set)



In the spirit of dependence, we will often write this as follows:

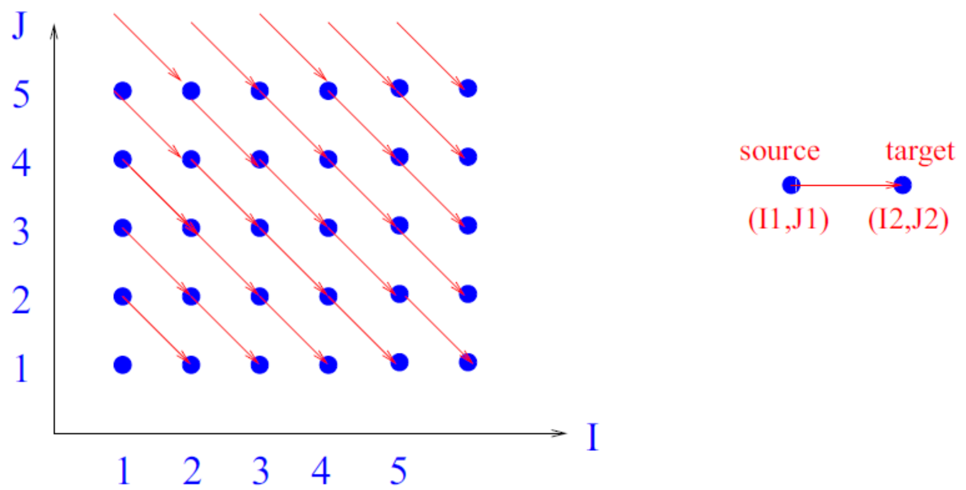
Flow dependence $= \{(Iw \rightarrow 2Iw+1) | 1 \leq Iw \leq 49\}$

2D loop nest

```
FOR  I = 1,100
  FOR  J = 1,100
    X(I,J) = X(I-1,J+1) + 1
```

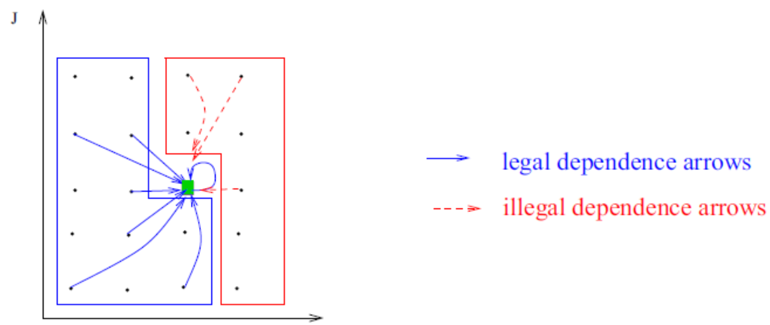
Dependence: relation of the form $(I_1, J_1) \to (I_2, J_2)$.

Picture in iteration space:



Dependence arrows are lexicographically positive

Legal and illegal dependence arrows:



If $(A \rightarrow B)$ is a dependence arrow, then A must be lexicographically less than or equal to B.

Dependence relation can be computed using ILP calculator

```
FOR  I = 1,100
  FOR  J = 1,100
     X(I,J) = X(I-1,J+1) + 1
```

Flow dependence constraints: $(I_w, J_w) \rightarrow (I_r, J_r)$

- $1 \leq Iw, Ir, Jw, Jr \leq 100$
- $(I_w, J_w) \prec (I_r, J_r)$
- $I_w = I_r - 1$
- $J_w = J_r + 1$

Use ILP calculator to determine the following relation:

$$D = \{(Iw, Jw) \rightarrow (Iw + 1, Jw - 1) | (1 \leq Iw \leq 99) \wedge (2 \leq Jw \leq 100)\}$$

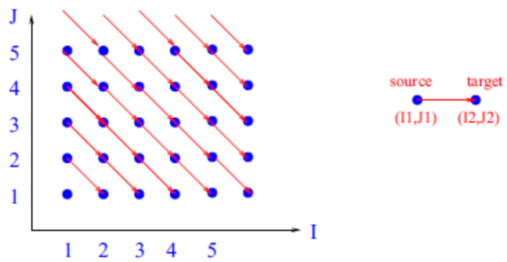
Dependence abstractions

- In practice, working with the full dependence relation for a loop nest is expensive and difficult
- Usually, we use an abstraction of dependence relation
  - Summary information about dependence
  - Summary is an over-approximation of actual dependence relation
- Two abstractions are popular
  - Distance vectors
  - Direction vectors
  - Dependence matrix: collection of distance/direction vectors

Distance/direction: Summarize dependence relation

Look at dependence relation from earlier slides:

$$\{(1,2) \rightarrow (2,1), (1,3) \rightarrow (2,2), ..(2,2) \rightarrow (3,1)...\}$$



Difference between dependent iterations $= (1, -1)$. That is,

$$(I_w, J_w) \rightarrow (I_r, J_r) \in \text{dependence relation, implies}$$

$$I_r - I_w = 1$$

$$J_r - J_w = -1$$

We will say that the *distance vector* is $(1, -1)$.

Note: From distance vector, we can easily recover the full relation.

In this case, distance vector is an *exact* summary of relation.

Computing distance vectors for a dependence

```
DO I = 1, 100
  X(2I+1) = ....X(I)...
```

Flow dependence:

$$\begin{aligned} 1 &\leq Iw < Ir \leq 100 \\ 2Iw + 1 &= Ir \end{aligned}$$

Flow dependence $= \{(Iw, 2Iw + 1) | 1 \leq Iw \leq 49\}$

Computing distance vectors without computing dependence set:
Introduce a new variable $\Delta = Ir - Iw$ and project onto Δ

$$\begin{aligned} 1 &\leq Iw < Ir \leq 100 \\ 2Iw + 1 &= Ir \\ \Delta &= Ir - Iw \end{aligned}$$

Solution: $\Delta = \{d | 2 \leq d \leq 50\}$

Example:2D loop nest

```
DO 10 I = 1,100
  DO 10 J = 1,100
  10 X(I,J) = X(I-1,J+1) + 1
```

Flow dependence constraints: $(I_w, J_w) \rightarrow (I_r, J_r)$
Distance vector: $(\Delta_1, \Delta_2) = (I_r - I_w, J_r - J_w)$

- $1 \leq Iw, Ir, Jw, Jr \leq 100$
- $(I_w, J_w) \prec (I_r, J_r)$
- $I_w = I_r - 1$
- $J_w = J_r + 1$
- $(\Delta_1, \Delta_2) = (I_r - I_w, J_r - J_w)$

Solution: $(\Delta_1, \Delta_2) = (1, -1)$

Direction vectors **Example:**

```
DO 10 I = 1,100
 10 X(2I+1) = X(I) + 1
```

Flow dependence equation: $2I_w + 1 = I_r$.

Dependence relation: $\{(1 \rightarrow 3), (2 \rightarrow 5), (3 \rightarrow 7), ...\}$ (1).

No fixed distance between dependent iterations!

But all distances are +ve, so use *direction vector* instead.

Here, direction = (+).

Intuition: (+) direction = some distances in range $[1, \infty)$

In general, direction = (+) or (0) or (-).

Also written by some authors as (<), (=), or (>).

Direction vectors are not exact.

(eg):if we try to recover dependence relation from direction (+), we get bigger relation than (1):

$\{(1 \rightarrow 2), (1 \rightarrow 3), ..., (1 \rightarrow 100), (2 \rightarrow 3), (2 \rightarrow 4), ...\}$

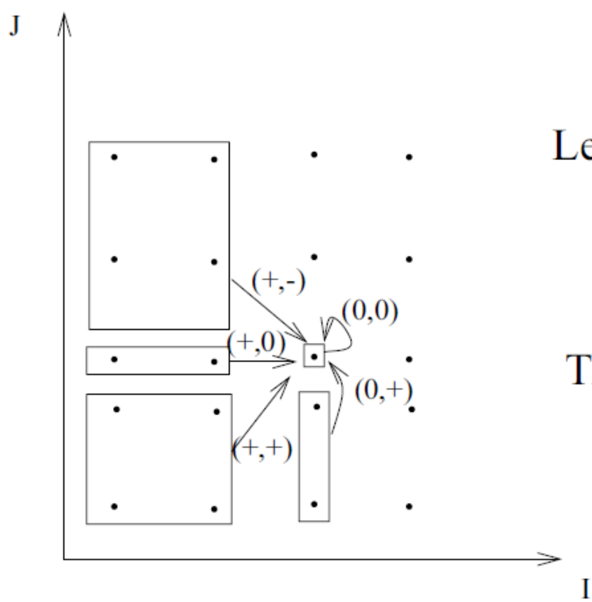
Directions for Nested Loops

Assume loop nest is (I,J).

If $(I_1, J_1) \rightarrow (I_2, J_2) \in$ dependence relation, then

$$\text{Distance} = (I_2 - I_1, J_2 - J_1)$$

$$\text{Direction} = (sign(I_2 - I_1), sign(J_2 - J_1))$$



Legal direction vectors:

(+,+) (0,+)
(+,-) (0,0)
(+,0)

The following direction vectors cannot exist:

(0,-) (-,+)
(-,0)
(-,-)

Valid dependence vectors are lexicographically positive.

How to compute Directions: Use IP engine

```
 DO 10 I = 1, 100
   X(f(I)) = ...
10         = ...X(g(I))..
```

Focus on flow dependences:

$$f(I_w) = g(I_r)$$

$$1 \leq I_w \leq 100$$

$$1 \leq I_r \leq 100$$

First, use inequalities shown above to test if dependence exists in any direction (called (*) direction).

If IP engine says there are no solutions, no dependence.

Otherwise, determine the direction(s) of dependence.

Test for direction (+): add inequality $I_w < I_r$

Test for direction (0): add inequality $I_w = I_r$

In a single loop, direction (−) cannot occur.

Computing Directions: Nested Loops

Same idea as single loop: *hierarchical testing*

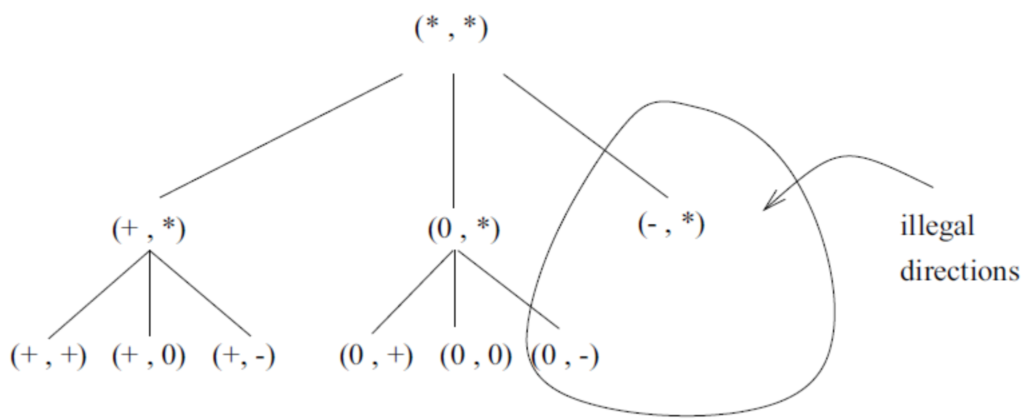


Figure 1: Hierarchical Testing for Nested Loop

Key ideas:

(1) Refine direction vectors top down.

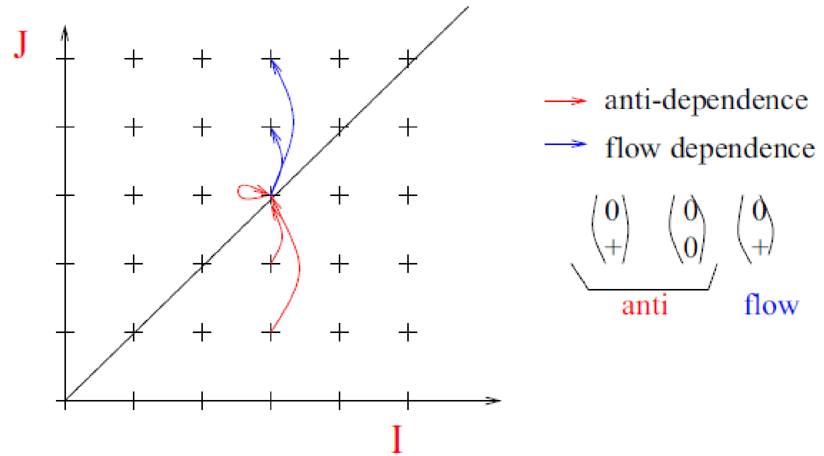
(eg),no dependence in $(*, *)$ direction

$\Rightarrow$ no need to do more tests.

(2) Do not test for impossible directions like $(-, *)$.

Example

```
DO I = 1,N
  DO J = 1,N
    X(I,J) = ...X(I,I)...
```



Linear system for anti-dependence:

$$I_w = I_r$$

$$J_w = I_r$$

$$1 \leq I_w, I_r, J_w, J_r \leq N$$

$$(I_r, J_r) \preceq (I_w, J_w)$$

$$\Delta 1 = (I_w - I_r)$$

$$\Delta 2 = (J_w - J_r)$$

Projecting onto $\Delta 1$ and $\Delta 2$, we get

$$\Delta 1 = 0$$

$$0 \leq \Delta 2 \leq (N - 1)$$

So directions for anti-dependence are

```
0   and  0
0        +
```

Dependence matrix

Dependence matrix for a loop nest

Matrix containing all dependence distance/direction vectors for all dependences of loop nest.

In our example, the dependence matrix is

$$\begin{pmatrix} 0 & 0 \\ 0 & + \end{pmatrix}$$

Conclusions

Traditional position: exact dependence testing (using IP engine) is too expensive

Recent experience:

(i) exact dependence testing is OK provided we first check for easy cases (ZIV,strong SIV, weak SIV)

(ii) IP engine is called for 3-4% of tests for direction vectors

(iii) Cost of exact dependence testing: 3-5% of compile time

Unimodular transformations
and
Transformation synthesis

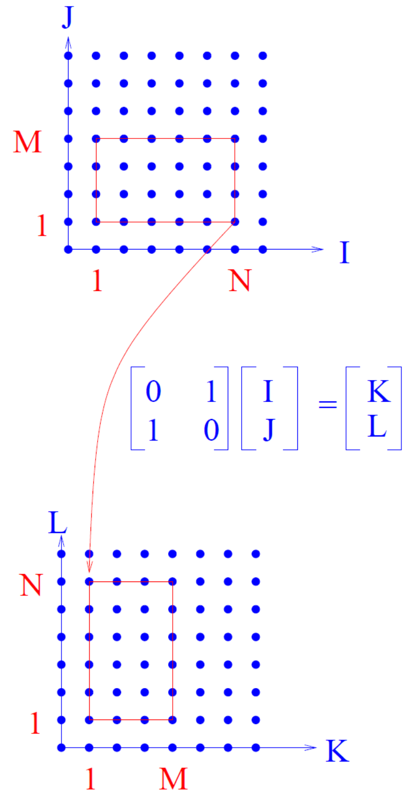
Overview

- Unimodular transformations
  - Permutation, skewing, reversal
  - These are linear transformations on iteration spaces and can be represented using integer matrices
  - Special property: unimodular matrix
    - Integer matrix with determinant of 1 or -1
    - Integer equivalent of orthogonal matrix in numerical linear algebra
  - Compositions of these transformations can be represented as unimodular matrices as well
- Synthesizing unimodular transformations for locality enhancement
  - Making a loop nest fully permutable to enable tiling

Permutation is linear transformation

Loop permutation = linear transformation on iteration space

```
DO  I = 1, N
  DO  J  = 1,M
    S(I,J)
```



$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} K \\ L \end{bmatrix}$$

```
DO  K  = 1, M
  DO  L = 1, N
    S'(K,L)
```

Using dependence matrices to establish correctness of permutation

Correctness of general permutation

Transformation matrix: T

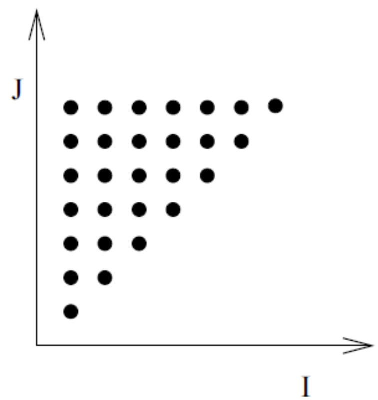
Dependence matrix: D

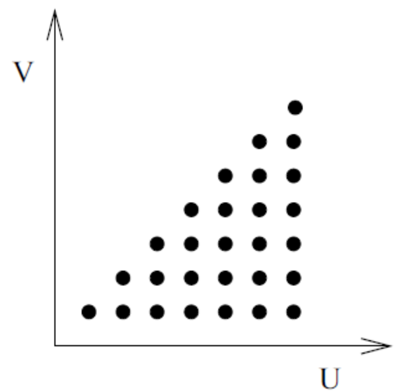
Matrix in which each column is a distance/direction vector

Legality: $T.D \succ 0$

Dependence matrix of transformed program: $T.D$

Loop permutation






```
DO I =  1, N
   DO J = I,N
   ..........
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

```
DO U = 1, N
   DO V = 1,U
   .........
```

$$\begin{bmatrix} I1 \\ J1 \end{bmatrix} \rightarrow \begin{bmatrix} I2 \\ J2 \end{bmatrix} \qquad T \begin{bmatrix} I1 \\ J1 \end{bmatrix} \qquad T \begin{bmatrix} I2 \\ J2 \end{bmatrix}$$

Dependence distance $= \begin{bmatrix} I2 - I1 \\ J2 - J1 \end{bmatrix}$

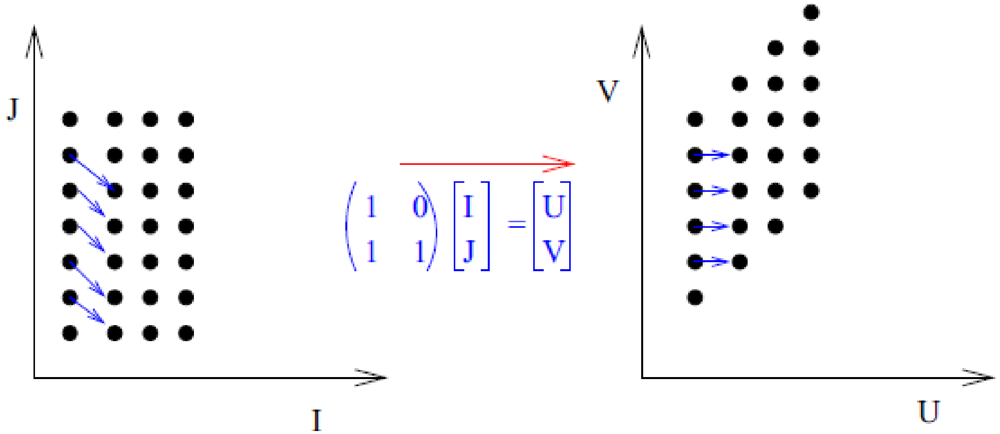
Distance between iterations =

$$T \begin{bmatrix} I2 \\ J2 \end{bmatrix} - T \begin{bmatrix} I1 \\ J1 \end{bmatrix} = T \begin{bmatrix} I2 - I1 \\ J2 - J1 \end{bmatrix} = \begin{bmatrix} J2 - J1 \\ I2 - I1 \end{bmatrix}$$

Check for legality: interchange positions in distance/direction vector & check for lex +ve

If transformation P is legal and original dependence matrix is D, new dependence matrix is T*D.

Loop Skewing: a linear loop transformation



$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

Skewing of inner loop by outer loop: $\begin{pmatrix} 1 & 0 \\ k & 1 \end{pmatrix}$ (k is some fixed integer)

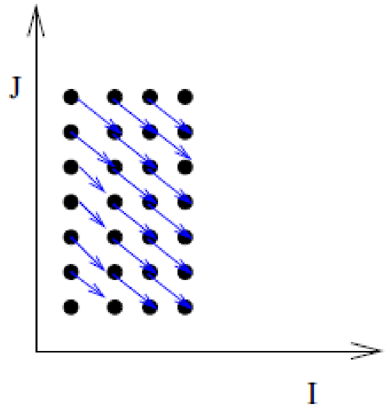
Skewing of inner loop by an outer loop: always legal

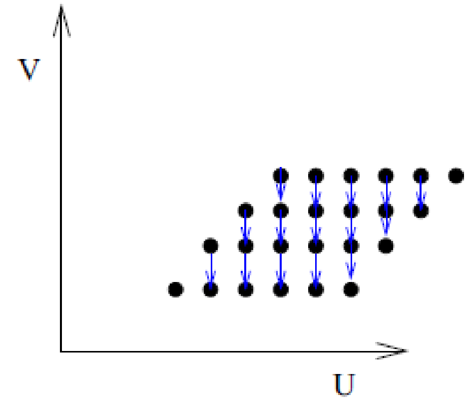
New dependence vectors: compute T*D

In this example, $D = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ $\quad T*D = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

This skewing has changed dependence vector but it has not brought dependent iterations closer together....

Skewing outer loop by inner loop





$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

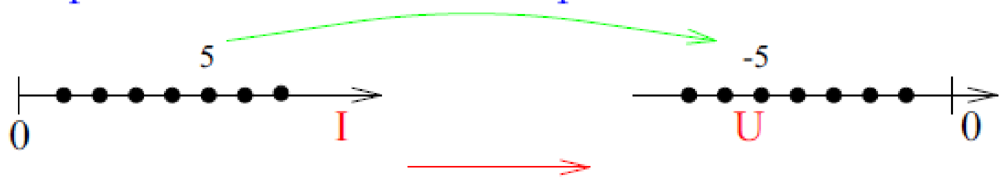
Outer loop skewing: $\begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix}$

Skewing of outer loop by inner loop: not necessarily legal

In this example, $D = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ $T*D = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$ incorrect

Dependent iterations are closer together (good) but program is illegal (bad).
How do we fix this??

Loop Reversal:a linear loop transformation



U = [-1][I]

```
DO  I =  1, N
  X(I) = I+2
```

```
DO U =  -N,-1
  X(-U) = -U +2
```

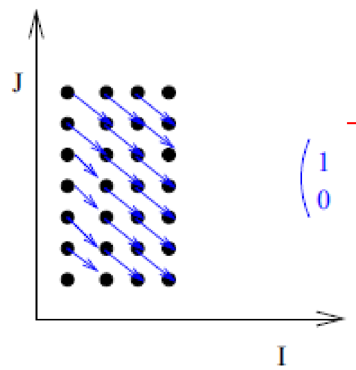
Transformation matrix = [-1]

Another example: 2-D loop, reverse inner loop $$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix}$$

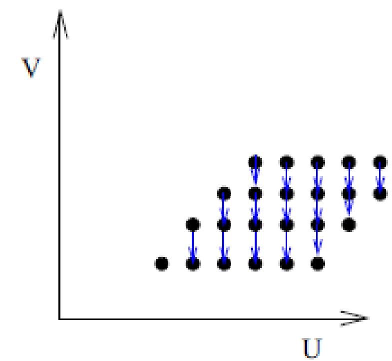
Legality of loop reversal: Apply transformation matrix to all dependences & verify lex +ve

Code generation: easy

Need for composite transformations



$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$



$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} G \\ H \end{bmatrix}$$

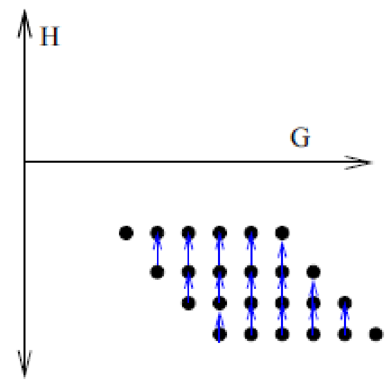
Transformation: skewing followed by reversal

In final program, dependent iterations are close together!

Composition of linear transformations = another linear transformation!

Composite transformation matrix is

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}$$



How do we synthesize this composite transformation??

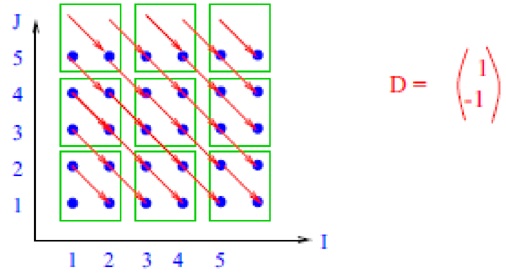
Some facts about permutation/reversal/skewing

- Transformation matrices for permutation/reversal/skewing are unimodular.
- Any composition of these transformations can be represented by a unimodular matrix.
- Any unimodular matrix can be decomposed into product of permutation/reversal/skewing matrices.
- Legality of composite transformation T: check that $T.D \succ 0$. (Proof: $T_3 * (T_2 * (T_1 * D)) = (T_3 * T_2 * T_1) * D$.)
- Code generation algorithm:
  - Original bounds: $A * \underline{I} \leq b$
  - Transformation: $\underline{U} = T * \underline{I}$
  - New bounds: compute from $A * T^{-1}\underline{U} \leq b$

Synthesizing composite transformations using matrix-based approaches

- Rather than reason about sequences of transformations, we can reason about the single matrix that represents the composite transformation.
- Enabling abstraction: dependence matrix

In general, tiling is not legal.



Tiling is illegal!

Tiling is legal if loops are fully permutable (all permutations of loops are legal).

Tiling is legal if all entries in dependence matrix are non-negative.

- Can we always convert a perfectly nested loop into a fully permutable loop nest?
- When we can, how do we do it?

Theorem: If all dependence vectors are distance vectors, we can convert entire loop nest into a fully permutable loop nest.

Example: wavefront

Dependence matrix is $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

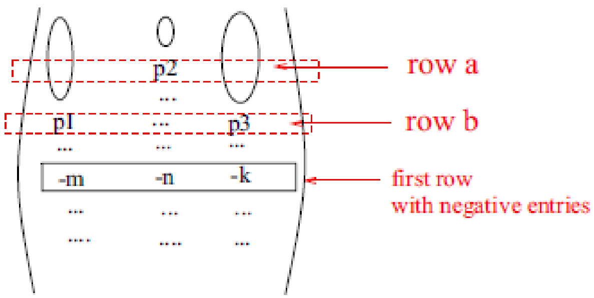
Dependence matrix of transformed program must have all positive entries.

So first row of transformation can be (1 0).

Second row of transformation (m 1) (for any m > 0).

General idea: skew inner loops by outer loops sufficiently to make all negative entries non-negative.

Transformation to make first row with negative entries into row with non-negative entries



(a) for each negative entry in the first row with negative entries,
find the first positive number in the corresponding column
assume the rows for these positive entries are a,b etc as shown above

(b) skew the row with negative entries by appropriate multiples of
rows a,b....
For our example, multiple of row a = ceiling(n/p2)
multiple of row b = ceiling(max(m/p1,k/p3))

Transformation:
$$\left(\begin{array}{l} \quad I \\ 0\ 0\ ..0\ \text{ceiling}(n/p2)\ 0\ 0\ \text{ceiling}(\max(m/p1,k/p3))0...0 \\ \quad I \end{array} \right)$$

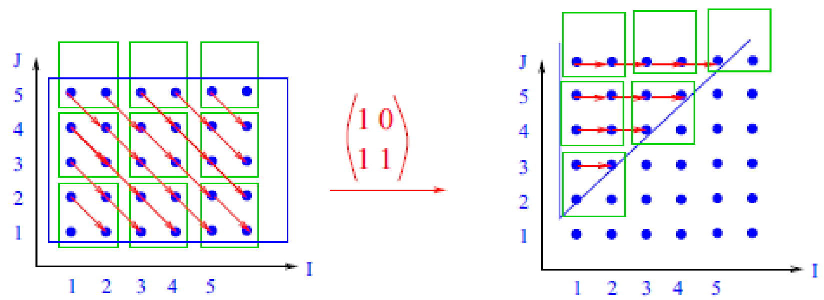
General algorithm for making loop nest fully permutable:

If all entries in dependence matrix are non-negative, done.

Otherwise,

1. Apply algorithm on previous slide to first row with non-negative entries.
2. Generate new dependence matrix.
3. If no negative entries, done.
4. Otherwise, go step (1).

Result of tiling transformed wavefront



Original loop   Tiled fully permutable loop

Tiling generates a 4-deep loop nest.

Not as nice as height reduction solution, but it will work fine for locality enhancement except at tile boundaries (but boundary points small compared to number of interior points).

What happens with direction vectors?

In general, we cannot make loop nest fully permutable.

Example: $D = \begin{pmatrix} + \\ - \\ + \end{pmatrix}$

Best we can do is to make some of the loops fully permutable.

We try to make outermost loops fully permutable, so we would interchange the second and third loops, and then tile the first two loops only.

Summary

- Dependence relation
  - Binary relation between points in iteration space
  - Can be computed using ILP calculator
- Dependence abstractions
  - Summary of dependence relation
    - Not as accurate but easier to compute and use
  - Distance/direction vectors
    - Put them together in dependence matrix
- Unimodular transformations
  - Can be represented using unimodular matrix
    - permutation, skewing, reversal, compositions of these
  - Synthesize unimodular transformations using dependence matrix as driver
    - Making a loop nest fully permutable