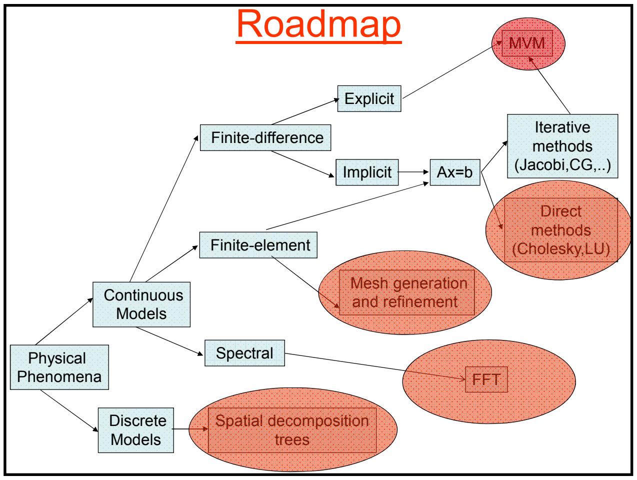
Computational Science Algorithms

Computational science

- Simulations of physical phenomena
  - fluid flow over aircraft (Boeing 777)
  - fatigue fracture in aircraft bodies
  - evolution of galaxies
  - ….
- Two main approaches
  - continuous models: fields and differential equations (eg. Navier-Stokes equations, Maxwell's equations,…)
  - discrete models: particles and forces (eg. gravitational forces)
- Paradox
  - most differential equations cannot be solved exactly
    - must use numerical techniques that convert calculus problem to matrix computations: discretization
  - n-body methods are straight-forward
    - but need to use a lot of bodies to get accuracy
    - must find a way to reduce $O(N^2)$ complexity of obvious algorithm

Roadmap

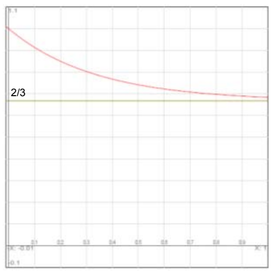


Organization

- Finite-difference methods
  - ordinary and partial differential equations
  - discretization techniques
    - explicit methods: Forward-Euler method
    - implicit methods: Backward-Euler method
- Finite-element methods
  - mesh generation and refinement
  - weighted residuals
- N-body methods
  - Barnes-Hut
- Key algorithms and data structures
  - matrix computations
    - algorithms
      - MVM and MMM
      - solution of systems of linear equations
        - » direct methods
        - » iterative methods
    - data structures
      - dense and sparse matrices
  - graph computations
    - mesh generation and refinement
  - spatial decomposition trees

Ordinary differential equations

- Consider the ode
  $u'(t) = -3u(t)+2$
  $u(0) = 1$
- This is called an initial value problem
  – initial value of u is given
  – compute how function u evolves for t > 0
- Using elementary calculus, we can solve this ode exactly
  $u(t) = 1/3\ (e^{-3t}+2)$

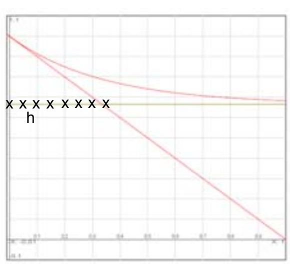


Problem

- For general ode's, we may not be able to express solution in terms of elementary functions
- In most practical situations, we do not need exact solution anyway
  – enough to compute an approximate solution, provided
    - we have some idea of how much error was introduced
    - we can improve the accuracy as needed
- General solution:
  – convert calculus problem into algebra/arithmetic problem
    - discretization: replace continuous variables with discrete variables
    - in finite differences,
      – time will advance in fixed-size steps: t=0,h,2h,3h,…
      – differential equation is replaced by difference equation

Forward-Euler method

- Intuition:
  – we can compute the derivative at t=0 from the differential equation
  $u'(t) = -3u(t)+2$
  – so compute the derivative at t=0 and advance along tangent to t =h to find an approximation to u(h)
- Formally, we replace derivative with forward difference to get a difference equation
  – $u'(t) \rightarrow (u(t+h) - u(t))/h$
- Replacing derivative with difference is essentially the inverse of how derivatives were probably introduced to you in elementary calculus

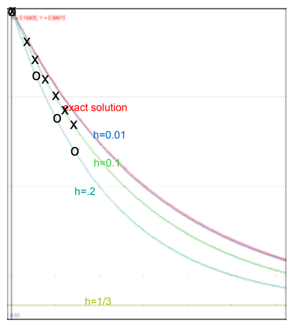


Back to ode

- Original ode
  $u'(t) = -3u(t)+2$
- After discretization using Forward-Euler:
  $(u_f(t+h) - u_f(t))/h = -3u_f(t)+2$
- After rearrangement, we get difference equation
  $u_f(t+h) = (1-3h)u_f(t)+2h$
- We can now compute values of u:
  $u_f(0) = 1$
  $u_f(h) = (1-h)$
  $u_f(2h) = (1-2h+3h^2)$
  …..

Tabulation

- Numerical solution
  - Choose a value for h
  - Tabulate the values of u_f at t = nh for n = 0,1,2,…, using the recurrence formula
- Question: how do you choose the step size h?
  - Small h is more accurate but also more computationally intensive
  - If we assume we want to estimate the value of u at t = T, we will need O(T/h) evaluations of the recurrence formula
- Important property of forward-Euler:
  - Numerical solution is stable only if h is "small enough"
  - If h is too big, numerical estimate will blow up
  - Recurrence formula is a feedback loop and error introduced at one time step gets amplified by the recurrence formula



Analysis of recurrence formula

- Understanding notions like stability of finite-difference formulas is complex in general
- In this particular case, we can do the analysis easily because we can solve difference equation exactly
- It is not hard to show that if difference equation is

  $u_f(t+h) = a*u_f(t)+b$

  $u_f(0) = 1$

  the solution is

  $u_f(nh) = a^n+b*(1-a^n)/(1-a)$
- For our difference equation,

  $u_f(t+h) = (1-3h)u_f(t)+2h$

  the exact solution is

  $u_f(nh) = 1/3((1-3h)^n+2)$

Comparison

- Exact solution

  $u(t) = 1/3 (e^{-3t}+2)$

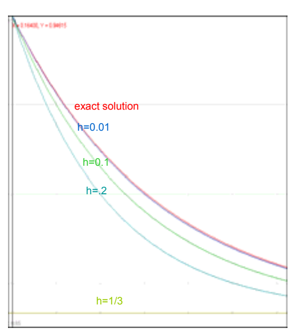
  $u(nh) = 1/3(e^{-3nh}+2)$ (at time-steps)
- Forward-Euler solution

  $u_f(nh) = 1/3((1-3h)^n+2)$
- Use series expansion to compare

  $u(nh) = 1/3(1-3nh+9/2\ n^2h^2 \cdots + 2)$

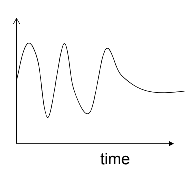
  $u_f(nh) = 1/3(1-3nh+n(n-1)/2\ 9h^2+\ldots+2)$

  So error = $O(nh^2)$
- Conclusion:
  - error per time step (local error) = $O(h^2)$
  - error at time nh = $O(nh^2)$
- In general, Forward-Euler converges only if time step is "small enough"



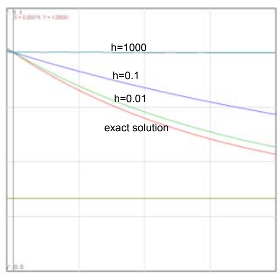
Choosing time step

- Time-step needs to be small enough to capture highest frequency phenomenon of interest
- Nyquist's criterion
  - sampling frequency must be at least twice highest frequency to prevent aliasing
  - for most finite-difference formulas, you need sampling frequencies (much) higher than the Nyquist criterion
- In practice, most functions of interest are not band-limited, so use
  - insight from application or
  - reduce time-step repeatedly till changes are not significant
- Fixed-size time-step can be inefficient if frequency varies widely over time interval
  - other methods like finite-elements permit variable time-steps as we will see later



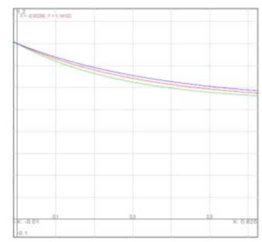
Backward-Euler method

- Replace derivative with backward difference
  $u'(t) \rightarrow (u(t) - u(t-h))/h$
- For our ode, we get
  $u_b(t)-u_b(t-h)/h = -3u_b(t)+2$
  which after rearrangement
  $u_b(t)= (2h+u_b(t-h))/(1+3h)$
- As before, this equation is simple enough that we can write down the exact solution:
  $u_b(nh) = ((1/(1+3h))^n + 2)/3$
- Using series expansion, we get
  $u_b(nh) = (1-3nh + (-n(-n-1)/2)\, 9h^2 + \ldots+2)/3$
  $u_b(nh) = (1 -3nh + 9/2\, n^2h^2 + 9/2\, nh^2 + \ldots+2)/3$
  So error = $O(nh^2)$ (for any value of h)



Comparison

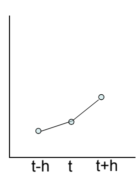
- Exact solution
  $u(t) = 1/3\ (e^{-3t}+2)$
  $u(nh) = 1/3(e^{-3nh}+2)$ (at time-steps)
- Forward-Euler solution
  $u_f(nh) = 1/3(\ (1-3h)^n+2)$
  error = $O(nh^2)$ (provided $h < 2/3$)
- Backward-Euler solution
  $u_b(n*h) = 1/3\ ((1/(1+3h))^n + 2)$
  error = $O(nh^2)$ (h can be any value you want)
- Many other discretization schemes have been studied in the literature
  - Runge-Kutta
  - Crank-Nicolson
  - Upwind differencing
  - …



Red: exact solution
Blue: Backward-Euler solution (h=0.1)
Green: Forward-Euler solution (h=0.1)

Higher-order difference formulas

- First derivatives:
  - Forward-Euler: $y'(t) \rightarrow y_f(t+h)-y_f(t)\ /h$
  - Backward-Euler: $y'(t) \rightarrow y_b(t)-y_b(t-h)\ /h$
  - Centered: $y'(t) \rightarrow y_c(t+h)-y_c(t-h)/2h$
- Second derivatives:
  - Forward: $y''(t) \rightarrow$
    $(y_f(t+2h)-y_f(t+h))- (y_f(t+h)-y_f(t))/h^2$
    $= y_f(t+2h)-2y_f(t+h)+y_f(t)/h^2$
  - Backward: $y''(t) \rightarrow y_b(t)-2y_b(t-h)+y_b(t-2h)/h^2$
  - Centered: $y''(t) \rightarrow y_c(t+h) - 2y_c(t)+y_c(t-h)/h^2$



Finite-differences: partial differential equations

Finite-difference methods for solving partial differential equations

- Basic ideas carry over unchanged
- Example: 2-d heat equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

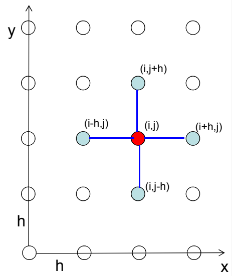
- Assume temperature at boundary is fixed
- Discretize domain using a regular NxN grid of pitch h
- Approximate derivatives as centered differences

$$\frac{\partial^2 u}{\partial y^2} \rightarrow \frac{\frac{u(i,j+h)-u(i,j)}{h} - \frac{u(i,j)-u(i,j-h)}{h}}{h}$$

$$\frac{\partial^2 u}{\partial x^2} \rightarrow \frac{\frac{u(i+h,j)-u(i,j)}{h} - \frac{u(i,j)-u(i-h,j)}{h}}{h}$$

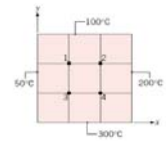
- So we get a system of (N-1)x(N-1) difference equations in terms of the unknowns at the (N-1)x(N-1) interior points

for all interior point (i,j)
u(i,j+h)+u(i,j-h)+u(i+h,j)+u(i-h,j) – 4u(i,j) = h^2 f(i,j)



5-point stencil

Example



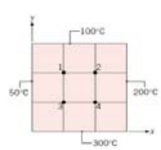
$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

Assume *f(x,y)* = 0

- **Unknown temperatures are T1, T2, T3, T4**
- **Discretized equation at point 1:**

$$\frac{\frac{T2-T1}{h} - \frac{T1-50}{h}}{h} + \frac{\frac{100-T1}{h} - \frac{T1-T3}{h}}{h} = 0$$

Example (contd)



$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

Assume f(x,y) = 0

-4T1 + T2 + T3 = -150
T1 – 4T2 + T4 = -300
T1 - 4T3 + T4 = -350
T2 + T3 - 4T4 = -500

$$\begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix} \begin{bmatrix} T1 \\ T2 \\ T3 \\ T4 \end{bmatrix} = \begin{bmatrix} -150 \\ -300 \\ -350 \\ -500 \end{bmatrix}$$

Solution: T1 = 119, T2 = 156, T3 = 169, T4 = 206

How do we solve large systems of linear equations?

General picture for heat equation

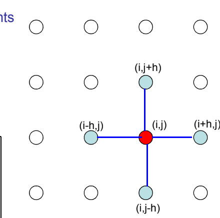
- System of (N-1)x(N-1) difference equations in terms of the unknowns at the (N-1)x(N-1) interior points

for all interior point (i,j)
u(i,j+h)+u(i,j-h)+u(i+h,j)+u(i-h,j) – 4u(i,j) = h^2 f(i,j)

- Matrix notation: use natural order for u's

$$\begin{bmatrix} \ldots\ldots \\ \ldots\ldots \\ \ldots\ldots \\ \ldots\ldots \\ 0..1\,0..0\,1\,-4\,1\,0..0\,1\,0\ldots0. \\ 0..0\,0\,1\,0..0\,1\,-4\,1\,0..0\,1\,0. \\ \ldots\ldots \\ \ldots\ldots \\ \ldots\ldots \end{bmatrix} \begin{bmatrix} \ldots \\ u(i-h,j) \\ \ldots \\ u(i,j-h) \\ u(i,j) \\ u(i,j+h) \\ \ldots \\ u(i+h,j) \\ \ldots \end{bmatrix} = h^2 \begin{bmatrix} \ldots \\ f(i,j) \\ \ldots \end{bmatrix}$$

Pentadiagonal sparse matrix
Matrix is known at compile-time and has simple structure.



5-point stencil

Solving linear systems

- Linear system: $A\underline{x} = \underline{b}$
- Two approaches
  - direct methods: Cholesky, LU with pivoting
    - factorize A into product of lower and upper triangular matrices A = LU
    - solve two triangular systems
      $L\underline{y} = \underline{b}$
      $U\underline{x} = \underline{y}$
    - problems:
      - even if A is sparse, L and U can be quite dense ("fill")
      - no useful information is produced until the end of the procedure
  - iterative methods: Jacobi, Gauss-Seidel, CG, GMRES
    - guess an initial approximation $\underline{x}_0$ to solution
    - error is $A\underline{x}_0 - \underline{b}$ (called residual)
    - repeatedly compute better approximation $\underline{x}_{i+1}$ from residual $(A\underline{x}_i - \underline{b})$
    - terminate when approximation is "good enough"

Iterative method: Jacobi iteration

- Linear system
  4x+2y=8
  3x+4y=11
- Exact solution is (x=1,y=2)
- Jacobi iteration for finding approximations to solution
  - guess an initial approximation
  - iterate
    - use first component of residual to refine value of x
    - use second component of residual to refine value of y
- For our example
  $x_{i+1} = (8 - 2y_i)/4$
  $y_{i+1} = (11 - 3x_i)/4$
  - for initial guess $(x_0=0, y_0=0)$

i	0	1	2	3	4	5	6	7
x	0	2	0.625	1.375	0.8594	1.1406	0.9473	1.0527
y	0	2.75	1.250	2.281	1.7188	2.1055	1.8945	2.0396

Jacobi iteration: matrix notation

- Linear system
  4x+2y=8
  3x+4y=11
- Jacobi iteration
  $x_{i+1} = (8 - 2y_i)/4$
  $y_{i+1} = (11 - 3x_i)/4$
- Useful to write Jacobi iteration in terms of residual (error):
  $x_{i+1} = x_i - \frac{1}{4}(4x_i+2y_i-8)$
  $y_{i+1} = y_i - \frac{1}{4}(3x_i+4y_i-11)$
- In matrix terms, this is

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \end{pmatrix} - \begin{pmatrix} 1/4 & 0 \\ 0 & 1/4 \end{pmatrix} \begin{pmatrix} 4xi+2yi-8 \\ 3xi+4yi-11 \end{pmatrix}$$

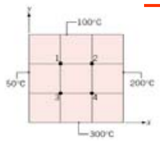
Jacobi iteration: general picture

- Linear system $A\underline{x} = \underline{b}$
- Jacobi iteration

  $\underline{x}_{i+1} = \underline{x}_i - M^{-1}(A\underline{x}_i - \underline{b})$ (where M is the diagonal of A)

- Key operation:
  - matrix-vector multiplication
  - important to exploit sparsity structure of A to reduce storage and computation
- Caveat:
  - Jacobi iteration does not always converge
  - even when it converges, it usually converges slowly
  - there are faster iterative methods available: CG,GMRES,..
  - what is important from our perspective is that key operation in all these iterative methods is matrix-vector multiplication

Example (contd)



$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

Assume $f(x,y) = 0$

$-4T1 + T2 + T3 = -150$
$T1 - 4T2 + T4 = -300$
$T1 - 4T3 + T4 = -350$
$T2 + T3 - 4T4 = -500$

Jacobi →

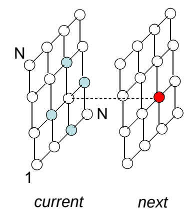
$-4T1^{n+1} + T2^n + T3^n + 0 = -150$
$T1^n - 4T2^{n+1} + 0 + T4^n = -300$
$T1^n + 0 - 4T3^n{+}1 + T4^n = -350$
$0 + T2^n + T3^n - 4T4^{n+1} = -500$

$T1^{n+1} = \frac{1}{4}(T2^n + T3^n + 100 + 50)$
$T2^{n+1} = \frac{1}{4}(T1^n + T4^n + 100 + 200)$
$T3^{n+1} = \frac{1}{4}(T1^n + T4^n + 300 + 50)$
$T4^{n+1} = \frac{1}{4}(T2^n + T3^n + 300 + 200)$

- Initialize the interior temperatures to some values
- Iterate until some measure of convergence is met
- Key operation is a sparse MVM
- However, code is implemented without explicit sparse matrices

Implementation

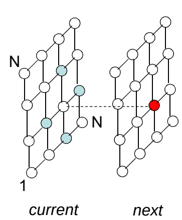
- Data structures
  - temperature values at a given iteration can be stored in a NxN matrix
  - use two matrices, *current* and *next*
- Algorithm
  - compute values in *next* using values in *current*
  - operator: five-point stencil
    next[i,j]=(*current*[i-1,j]+*current*[i,j-1]
    +*current*[i+1,j]+*current*[i,j+1])/4
  - switch between *current* and *next* in successive iterations
- Questions:
  - Where is the parallelism in this algorithm?
    - All grid points in *next* can be computed in parallel
  - Can we exploit locality?
    - Postpone discussion



5-point stencil

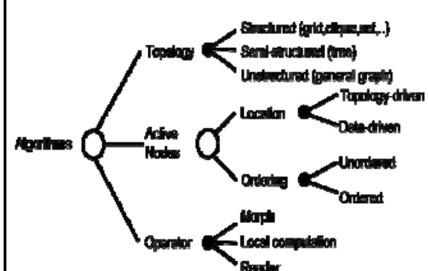
Operator formulation of algorithms

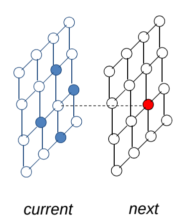
- Data-centric abstraction of algorithms
- Active element
  - Node /edge where computation is needed
  - Jacobi: all nodes of *next* grid
- Operator
  - Computation at active element
  - Jacobi: five-point stencil
- Activity: application of operator to active element
- Neighborhood
  - Set of nodes/edges read/written by activity
  - Jacobi: active node in *next* grid and neighbors in *current* grid
- Ordering : scheduling constraints on execution order of activities
  - Unordered algorithms: no semantic constraints but performance may depend on schedule
  - Ordered algorithms: problem-dependent order
  - Jacobi: unordered algorithm



5-point stencil

TAO analysis: algorithm abstraction



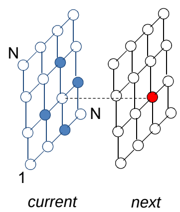


5-point stencil

Jacobi
- Topology: structured (grid)
- Active nodes: topology-driven (all nodes of next grid), unordered
- Operator: local computation for *next*, reader for *current*

Parallelism in unordered algorithms

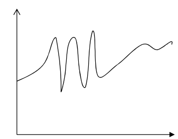
- Work on multiple active nodes simultaneously
- Constraint:
  - final state must be identical to state produced by processing active nodes serially in some order
  - amorphous data-parallelism
- One implementation:
  - activities can be executed in parallel if and only if their neighborhoods are disjoint (otherwise, activities conflict)
  - correct but conservative: nearby active nodes in grid cannot be processed in parallel
- Another implementation:
  - if neighborhoods of concurrent activities overlap, graph elements in intersection of neighborhoods are read-only (more refined notion of conflict)
  - satisfactory for Jacobi
  - most general picture: commutativity of activities (we won't worry about this)
- Data parallelism:
  - topology-driven algorithm
  - no conflicts between activities



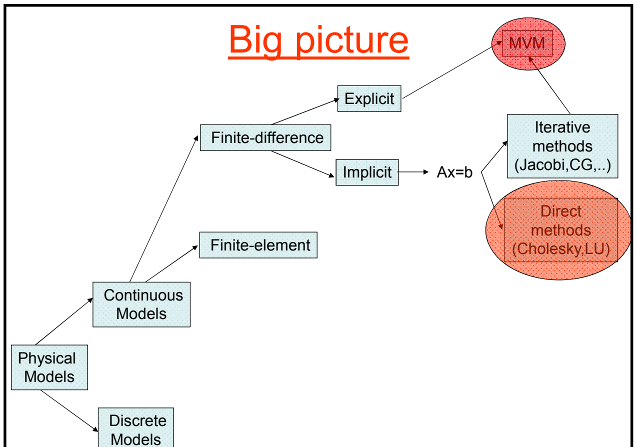
5-point stencil

Summary

- Finite-difference methods
  - can be used to find approximate solutions to ode's and pde's
- Many large-scale computational science simulations use these methods
- Time step or grid step needs to be constant and is determined by highest-frequency phenomenon
  - can be inefficient for when frequency varies widely in domain of interest
  - one solution: structured AMR methods



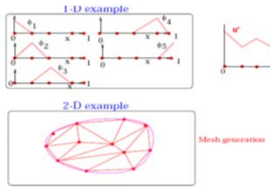
Big picture



Finite-element methods

- Express approximate solution to pde as a linear combination of certain basis functions
- Similar in spirit to Fourier analysis
  - express periodic functions as linear combinations of sines and cosines
- Questions:
  - what should be the basis functions?
    - mesh generation: discretization step for finite-elements
    - mesh defines basis functions ˘$_3$, ˘$_4$, ˘$_5$,…which are low-degree piecewise polynomial functions
  - given the basis functions, how do we find the best linear combination of these for approximating solution to pde?
    - u = Σ_i c_i ˘$_i$
    - weighted residual method: similar in spirit to what we do in Fourier analysis, but more complex because basis functions are not necessarily orthogonal

Mesh generation and refinement



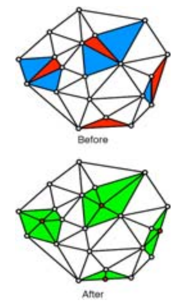
- 1-D example:
  - mesh is a set of points, not necessarily equally spaced
  - basis functions are "hats" which
    - have a value of 1 at a mesh point,
    - decay down to 0 at neighboring mesh points
    - 0 everywhere else
  - linear combinations of these produce piecewise linear functions in domain, which may change slope only at mesh points
- In 2-D, mesh is a triangularization of domain, while in 3-D, it might be a tetrahedralization
- Mesh refinement: called h-refinement
  - add more points to mesh in regions where discretization error is large
  - irregular nature of mesh makes this easy to do this locally
  - finite-differences require global refinement which can be computationally expensive

Delaunay Mesh Refinement

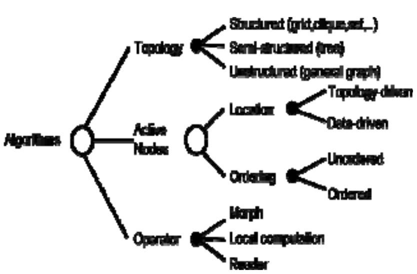
- Iterative refinement to remove *bad* triangles with lots of discretization error:

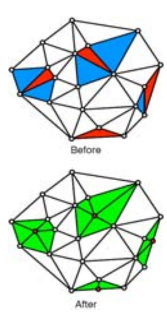
```
while there are bad triangles do {
        Pick a bad triangle;
        Find its cavity;
        Retriangulate cavity;
          // may create new bad triangles
        }
```

- Don't-care non-determinism:
  - final mesh depends on order in which bad triangles are processed
  - applications do not care which mesh is produced
- Data structure:
  - graph in which nodes represent triangles and edges represent triangle adjacencies
- Parallelism:
  - bad triangles with cavities that do not overlap can be processed in parallel
  - parallelism is dependent on runtime values
    - compilers cannot find this parallelism



TAO analysis





DMR
- Topology: unstructured graph
- Active nodes: data-driven, unordered
- Operator: morph

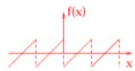
Finding coefficients

- Weighted residual technique
  - similar in spirit to what we do in Fourier analysis, but basis functions are not necessarily orthogonal
- Key idea:
  - problem is reduced to solving a system of equations A$\underline{x}$ = $\underline{b}$
  - solution gives the coefficients in the weighted sum
  - because basis functions are zero almost everywhere in the domain, matrix A is usually very sparse
    - number of rows/columns of A ~ O(number of points in mesh)
    - number of non-zeros per row ~ O(connectivity of mesh point)
  - typical numbers:
    - A is 10^9x10^9
    - only about ~100 non-zeros per row

Finding the best choices of the coefficients:

Analogy with Fourier series:

$$f(x) = a_0 + \sum_i a_i \cos(ix) + \sum_i b_i \sin(ix)$$



How do you find 'best' choices for a's and b's?

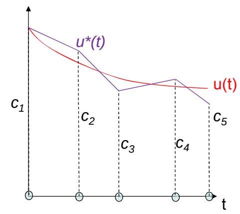
$$\int_{-\pi}^{+\pi} f(x)\cos(kx)\,dx = \int_{-\pi}^{+\pi} \left(a_0 + \sum_i a_i \cos(ix) + \sum_i b_i \sin(ix)\right)\cos(kx)\,dx$$

$$= \int_{-\pi}^{+\pi} a_k \cos(kx)\cos(kx)\,dx$$

$$= a_k \pi$$

Key idea:
- residual $f(x) - a_0 + \sum_i a_i \cos(ix) + \sum_i b_i \sin(ix)$
- weight residual by known function and integrate to find corresponding coefficient

Weighted residuals: intuitive idea



ODE: $\frac{du}{dt} = -3u + 2$

Approximate solution: $u^*(t) = \sum_{i=1}^{N} c_i \phi_i(t)$

$Residual(t) = \frac{d(u^*(t))}{dt} + 3u^*(t) - 2$

Write this as

$Residual(t) = L(u^*(t)) - f(t)$

- Residual depends on choice of c_i
- Choose c_i so that integral of residual, weighted by each ϕ_k is zero.
- This gives N equations in N unknowns, which can be solved to find values for c_i

Weighted Residual Technique:

Residual: $(L u^* - f) = (L(\sum_i^N c_i \phi_i) - f)$

Weighted Residual $= (L(\sum_i^N c_i \phi_i) - f)\ \phi_k$

Equation for k^{th} unknown: $\int_\Omega \phi_k * (L(\sum_i^N c_i \phi_i) - f)\, dV = 0 \Rightarrow$

If the differential equation is linear:

$$c_1 \int_\Omega \phi_k * L\phi_1\, dV + \dots + c_N \int_\Omega \phi_k * L\phi_N\, dV = \int_\Omega \phi_k f\, dV$$

$k = 1,2,\dots N$

This system can be written as

$K c = b$ where

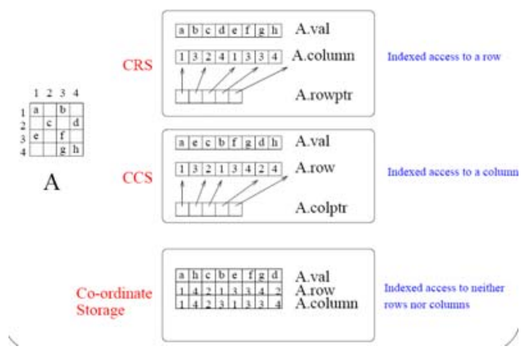
$$K(i,j) = \int_\Omega \phi_i * L\phi_j\, dV \qquad b(i) = \int_\Omega \phi_i f\, dV$$

Key insight: Calculus problem of solving pde is converted to linear algebra problem of solving K c = b where K is sparse

Sparse matrices in finite-element method

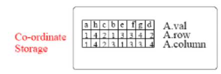
- Sparsity pattern is complex and irregular
  - Pattern and values of non-zeros depends on the mesh and basis functions, and is not known at compile-time
  - Cannot be inlined into code like we did for heat equation
- Solution:
  - represent sparse matrix explicitly
  - Use sparse MVM code specialized to that representation

Sparse matrix representations



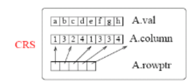
MVM with sparse matrices

- Coordinate storage

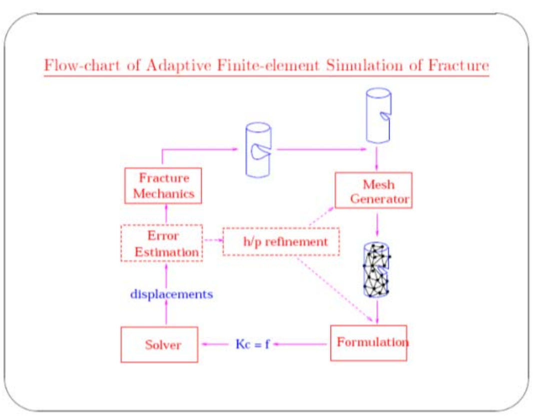



```
for P = 1 to NZ do
  Y(A.row(P))=Y(A.row(P)) + A.val(P)*X(A.column(P))
```

- CRS storage




```
for I = 1 to N do
  for JJ = A.rowptr(I) to A.rowPtr(I+1)-1 do
    Y(I)=Y(I)+A.val(JJ)*X(A.column(J)))
```



Barnes Hut N-body Simulation

Introduction

- Physical system simulation (time evolution)
  - System consists of bodies
  - "n" is the number of bodies
  - Bodies interact via pair-wise forces
- Many systems can be modeled in these terms
  - Galaxy clusters (gravitational force)
  - Particles (electric force, magnetic force)

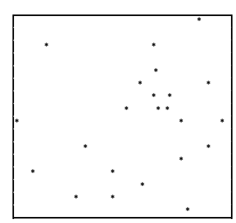
Barnes Hut Idea

- Precise force calculation
  - Requires $O(n^2)$ operations ($O(n^2)$ body pairs)
- Barnes and Hut (1986)
  - Algorithm to approximately compute forces
    - Bodies' initial position & velocity are also approximate
  - Requires only $O(n \log n)$ operations
  - Idea is to "combine" far away bodies
  - Error should be small because *force* ~ $1/r^2$

Barnes Hut Algorithm

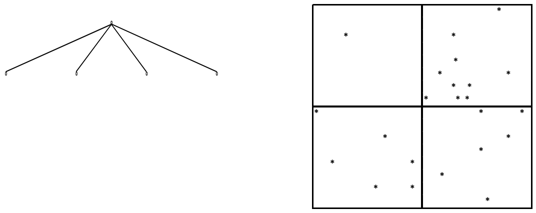
- Set bodies' initial position and velocity
- Iterate over time steps
  1. Subdivide space until at most one body per cell
     - Record this spatial hierarchy in an octree
  2. Compute mass and center of mass of each cell
  3. Compute force on bodies by traversing octree
     - Stop traversal path when encountering a leaf (body) or an internal node (cell) that is far enough away
  4. Update each body's position and velocity

Build Tree (Level 1)



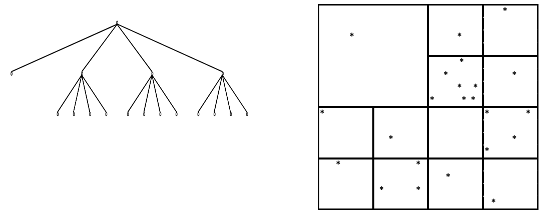
Subdivide space until at most one body per cell

Build Tree (Level 2)



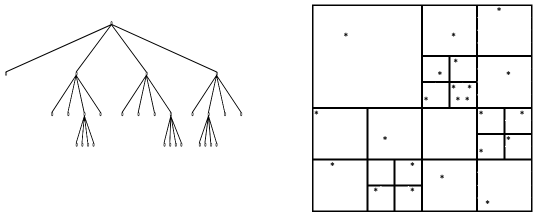
Subdivide space until at most one body per cell

Build Tree (Level 3)



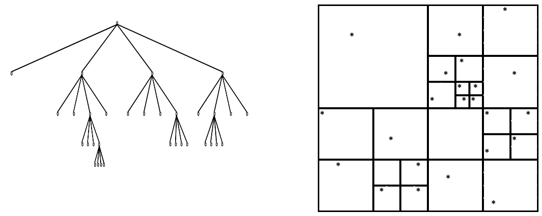
Subdivide space until at most one body per cell

Build Tree (Level 4)



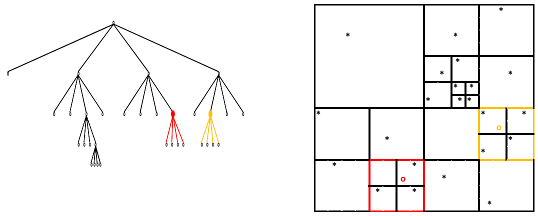
Subdivide space until at most one body per cell

Build Tree (Level 5)



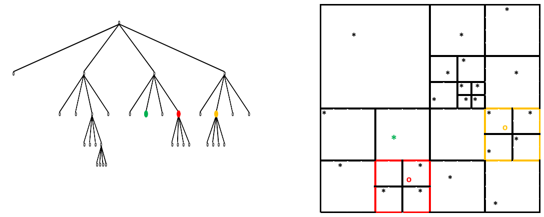
Subdivide space until at most one body per cell

Compute Cells' Center of Mass



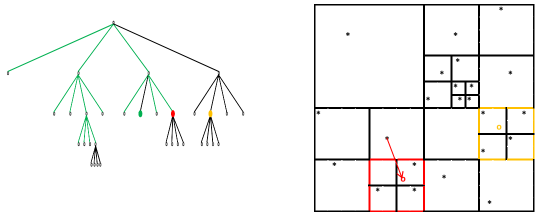
For each internal cell, compute sum of mass and weighted average of position of all bodies in subtree; example shows two cells only

Compute Forces



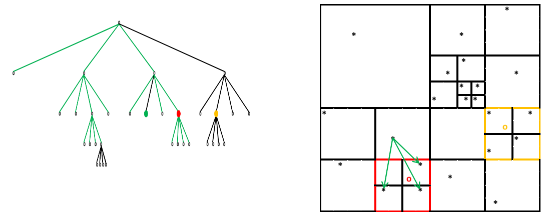
Compute force, for example, acting upon green body

Compute Force (short distance)



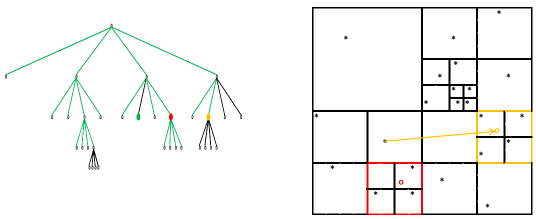
Scan tree depth first from left to right; green portion already completed

Compute Force (down one level)



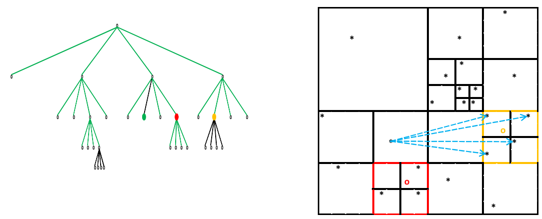
Red center of mass is too close, need to go down one level

Compute Force (long distance)



Yellow center of mass is far enough away

Compute Force (skip subtree)



Therefore, entire subtree rooted in the yellow cell can be skipped

Pseudocode

```
Set bodySet = ...
foreach timestep do {
  Octree octree = new Octree();
  foreach Body b in bodySet {
    octree.Insert(b);
  }
  OrderedList cellList = octree.CellsByLevel();
  foreach Cell c in cellList {
    c.Summarize();
  }
  foreach Body b in bodySet {
    b.ComputeForce(octree);
  }
  foreach Body b in bodySet {
    b.Advance();
  }
}
```

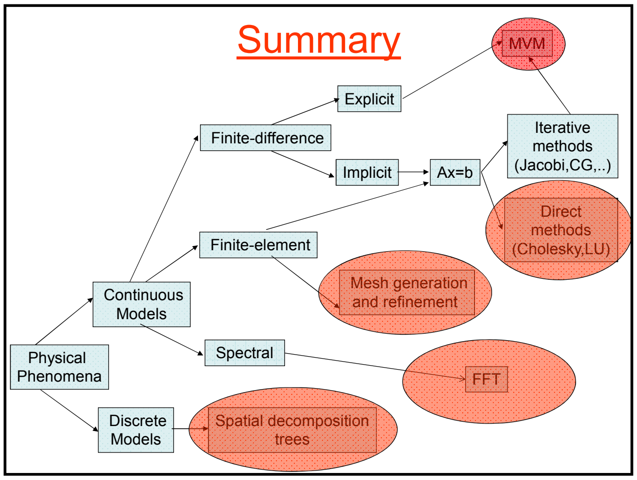

Complexity

```
Set bodySet = ...
foreach timestep do {           // O(n log n)
  Octree octree = new Octree();
  foreach Body b in bodySet {   // O(n log n)
    octree.Insert(b);
  }
  OrderedList cellList = octree.CellsByLevel();
  foreach Cell c in cellList {  // O(n)
    c.Summarize();
  }
  foreach Body b in bodySet {   // O(n log n)
    b.ComputeForce(octree);
  }
  foreach Body b in bodySet {   // O(n)
    b.Advance();
  }
}
```

Parallelism

```
Set bodySet = ...
foreach timestep do {            // sequential
  Octree octree = new Octree();
  foreach Body b in bodySet {    // tree building
    octree.Insert(b);
  }
  OrderedList cellList = octree.CellsByLevel();
  foreach Cell c in cellList {   // tree traversal
    c.Summarize();
  }
  foreach Body b in bodySet {    // fully parallel
    b.ComputeForce(octree);
  }
  foreach Body b in bodySet {    // fully parallel
    b.Advance();
  }
}
```


Summary



Summary (contd.)

- Some key computational science algorithms and data structures
  - MVM:
    - Source: explicit finite-difference methods for ode's, iterative linear solvers, finite-element methods
    - Both dense and sparse matrices
  - Stencil computations:
    - Source: explicit finite-difference methods for pde's
    - Dense matrices
  - A=LU:
    - Source: implicit finite-difference methods
    - Direct methods for solving linear systems: factorization
    - Usually only dense matrices
    - High-performance factorization codes use MMM as a kernel
  - Mesh generation and refinement
    - Finite-element methods
    - Graph computations

Extra material

Systems of ode's

- Consider a system of coupled ode's of the form

$u'(t) = a_{11}*u(t) + a_{12}*v(t) + a_{13}*w(t) + c_1(t)$
$v'(t) = a_{21}*u(t) + a_{22}*v(t) + a_{23}*w(t) + c_2(t)$
$w'(t) = a_{31}*u(t) + a_{32}*v(t) + a_{33}*w(t) + c_3(t)$

- If we use Forward-Euler method to discretize this system, we get the following system of simultaneous equations

$u_f(t+h)–u_f(t) /h = a_{11}*u_f(t) + a_{12}*v_f(t) + a_{13}*w_f(t) + c_1(t)$
$v_f(t+h)–v_f(t) /h = a_{21}*u_f(t) + a_{22}*v_f(t) + a_{23}*w_f(t) + c_2(t)$
$w_f(t+h)–w_f(t) /h= a_{31}*u_f(t) + a_{32}*v_f(t) + a_{33}*w_f(t) + c_3(t)$

Forward-Euler (contd.)

- Rearranging, we get

$u_f(t+h) = (1+ha_{11})*u_f(t) + ha_{12}*v_f(t) + ha_{13}*w_f(t) + hc_1(t)$
$v_f(t+h) = ha_{21}*u_f(t) + (1+ha_{22})*v_f(t) + ha_{23}*w_f(t) + hc_2(t)$
$w_f(t+h) = ha_{31}*u_f(t) + ha_{32}*v_f(t) + (1+a_{33})*w_f(t) + hc_3(t)$

- Introduce vector/matrix notation

$\underline{x}(t) = [u(t)\ v(t)\ w(t)]^T$

A = …..

$\underline{c}(t) = [c_1(t)\ c_2(t)\ c_3(t)]^T$

Vector notation

- Our systems of equations was

$u_f(t+h) = (1+ha_{11})*u_f(t) + ha_{12}*v_f(t) + ha_{13}*w_f(t) + hc_1(t)$
$v_f(t+h) = ha_{21}*u_f(t) + (1+ha_{22})*v_f(t) + ha_{23}*w_f(t) + hc_2(t)$
$w_f(t+h) = ha_{31}*u_f(t) + ha_{32}*v_f(t) + (1+a_{33})*w_f(t) + hc_3(t)$

- This system can be written compactly as follows
  $\underline{x}(t+h) = (I+hA)\underline{x}(t)+h\underline{c}(t)$
- We can use this form to compute values of $\underline{x}(h),\underline{x}(2h),\underline{x}(3h),\ldots$
- Forward-Euler is an example of explicit method of discretization
  - key operation: matrix-vector (MVM) multiplication
  - in principle, there is a lot of parallelism
    - $O(n^2)$ multiplications
    - O(n) reductions
  - parallelism is independent of runtime values

Backward-Euler

- We can also use Backward-Euler method to discretize system of ode's

$u_b(t)–u_b(t-h) /h = a_{11}*u_b(t) + a_{12}*v_b(t) + a_{13}*w_b(t) + c_1(t)$
$v_b(t)–v_b(t-h) /h = a_{21}*u_b(t) + a_{22}*v_b(t) + a_{23}*w_b(t) + c_2(t)$
$w_b(t)–w_b(t-h) /h= a_{31}*u_b(t) + a_{32}*v_b(t) + a_{33}*w_b(t) + c_3(t)$

- We can write this in matrix notation as follows
  $(I-hA)\underline{x}(t) = \underline{x}(t-h)+h\underline{c}(t)$
- Backward-Euler is example of implicit method of discretization
  - key operation: solving a linear system $A\underline{x} = \underline{b}$
- How do we solve large systems of linear equations?
- Matrix (I-hA) is often very sparse
  - Important to exploit sparsity in solving linear systems