Some Computational Science Algorithms and Data Structures

Keshav Pingali
University of Texas, Austin

Computational science

- Simulations of physical phenomena
  - fluid flow over aircraft (Boeing 777)
  - fatigue fracture in aircraft bodies
  - evolution of galaxies
  - ….
- Two main approaches
  - continuous methods: fields and differential equations (eg. Navier-Stokes equations, Maxwell's equations,…)
  - discrete methods/n-body methods: particles and forces (eg. gravitational forces)
- We will focus first on continuous methods in this lecture
  - most differential equations cannot be solved exactly
  - must use numerical methods that compute approximate solutions
    - discretization: convert calculus problem to linear algebra problem
  - finite-difference, finite-element and spectral methods
- Then we will study n-body methods

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
  - spatial decomposition trees
- Key algorithms and data structures
  - matrix computations
    - algorithms
      - matrix-vector multiplication (MVM)
      - matrix-matrix multiplication (MMM)
      - solution of systems of linear equations
        - direct methods
        - iterative methods
    - data structures
      - dense matrices
      - sparse matrices
  - graph computations
    - mesh generation and refinement
    - spatial decomposition trees

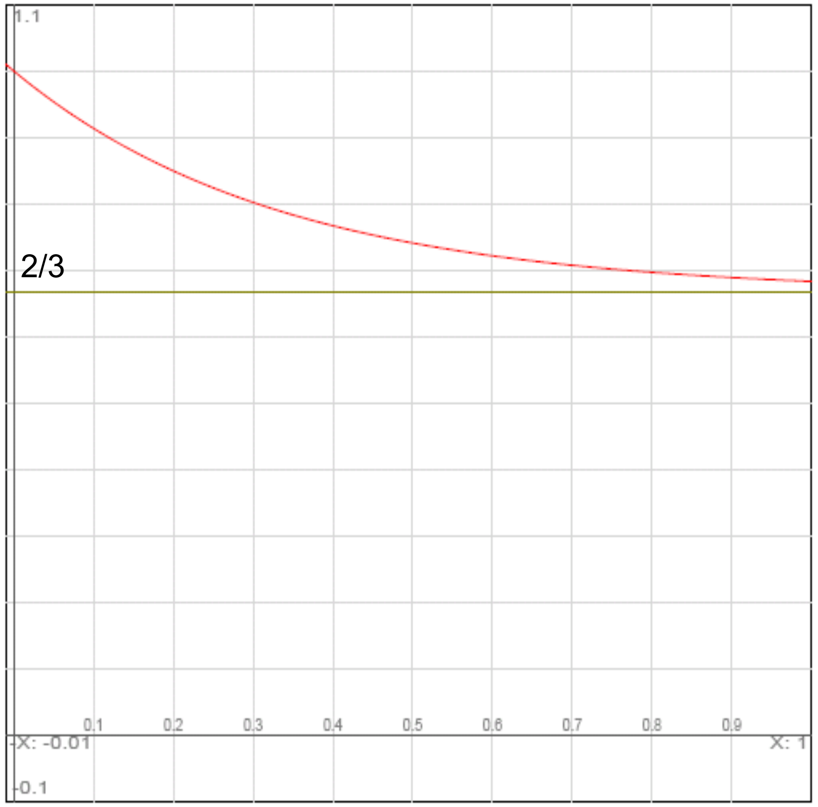
Ordinary differential equations

- 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

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

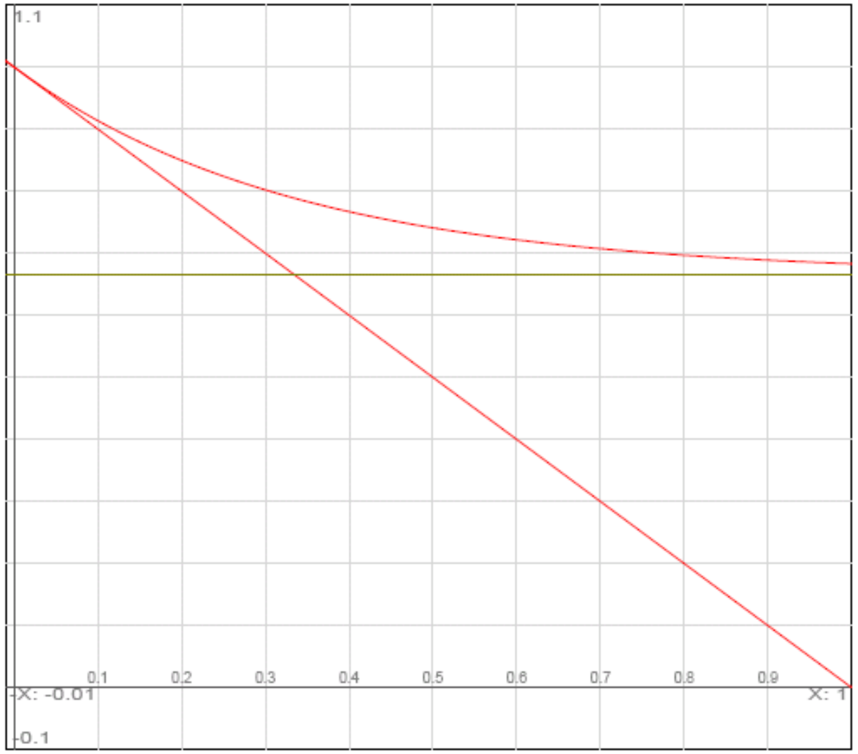


Problem

- 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: $t=0,h,2h,3h,\ldots$
      - 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:

  $(u(t+h) - u(t))/h = -3u(t)+2$

- After rearrangement, we get difference equation

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

- We can now compute values of u:

  $u(0) = 1$

  $u(h) = (1-h)$

  $u(2h) = (1-2h+3h^2)$

  …..

Exact solution of difference equation

- In this particular case, we can actually solve difference equation exactly
- It is not hard to show that if difference equation is

  $u(n+1) = a*u(n)+b$

  $u(0) = 1$

  the solution is

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

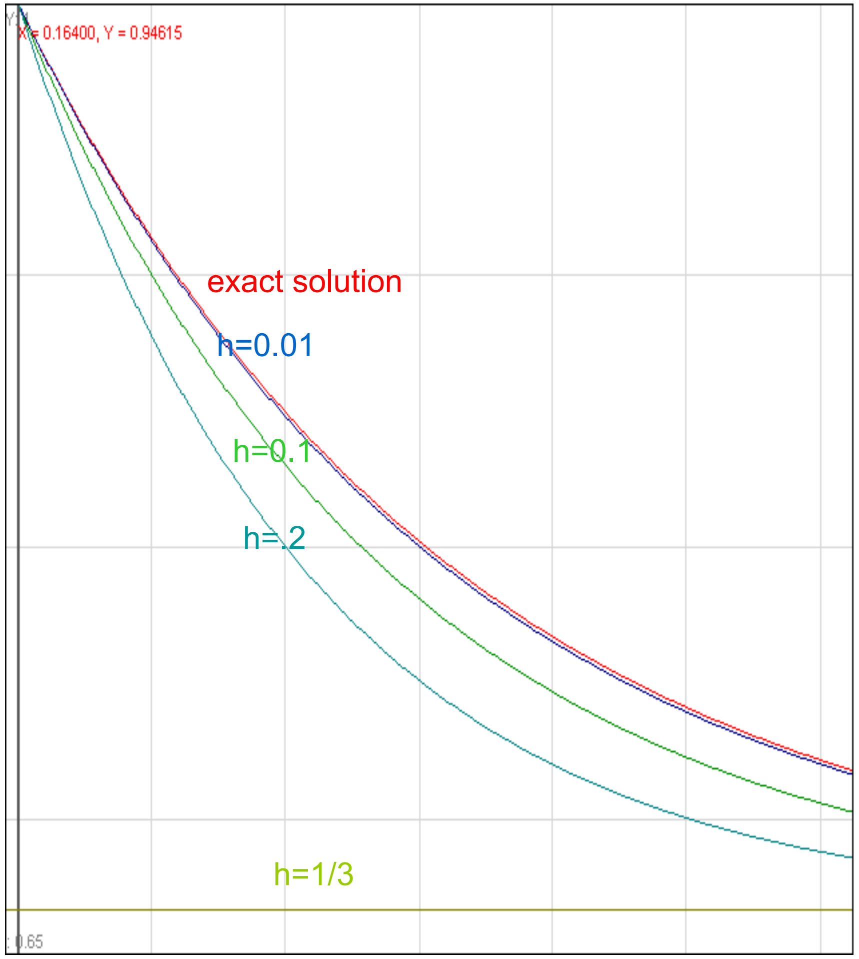
  $u(nh+h) = (1-3h)u(nh)+2h$

  the exact solution is

  $u(nh) = 1/3((1-3h)^n+2)$
- Stability:
  - values computed from difference equation will blow up if
    - $\|(1-3h)\| > 1 \Rightarrow h > 2/3$
  - for this problem, forward-Euler is stable only if step size is less than 2/3
  - in general, forward-Euler is stable only for small enough step sizes

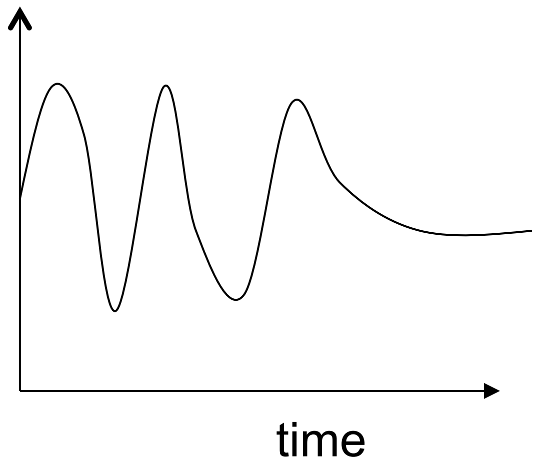
Comparison

- 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 = $O(nh^2)$ (provided $h < 2/3$)
- Conclusion:
  - error per time step (local error) = $O(h^2)$
  - error at time nh = $O(nh^2)$



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

  $u'(t+h) \rightarrow (u(t+h) - u(t))/h$

- For our ode, we get

  $u(t+h)-u(t)/h = -3u(t+h)+2$

  which after rearrangement

  $u(t+h)= (2h+u(t))/(1+3h)$

- As before, this equation is simple enough that we can write down the exact solution:

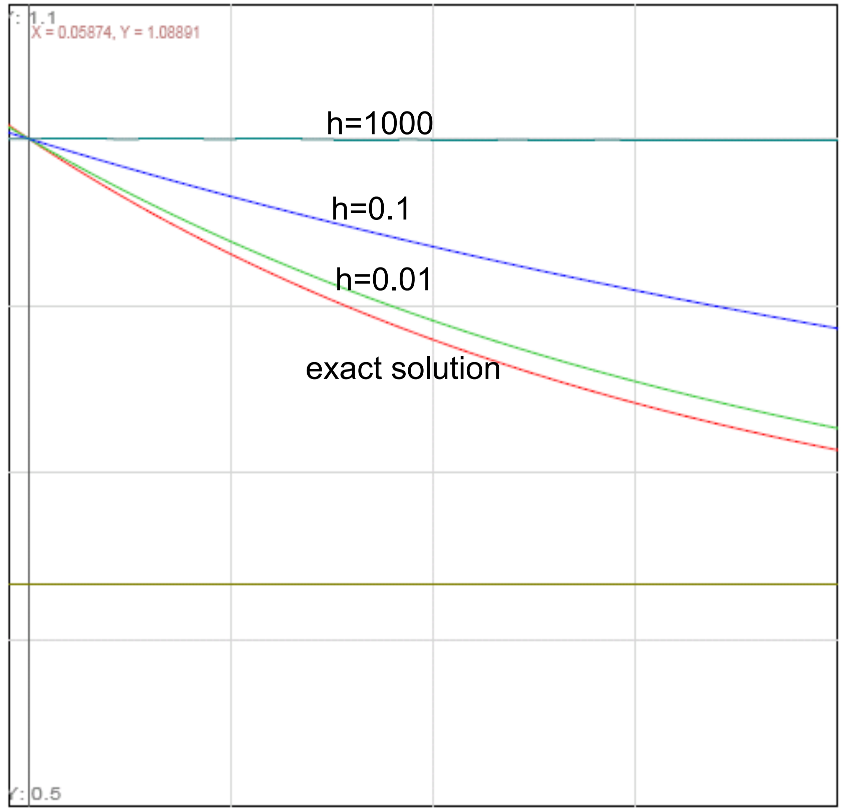
  $u(nh) = ((1/(1+3h))^n + 2)/3$

- Using series expansion, we get

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

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

  So error = $O(nh^2)$ (for any value of h)



Comparison

- Exact solution

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

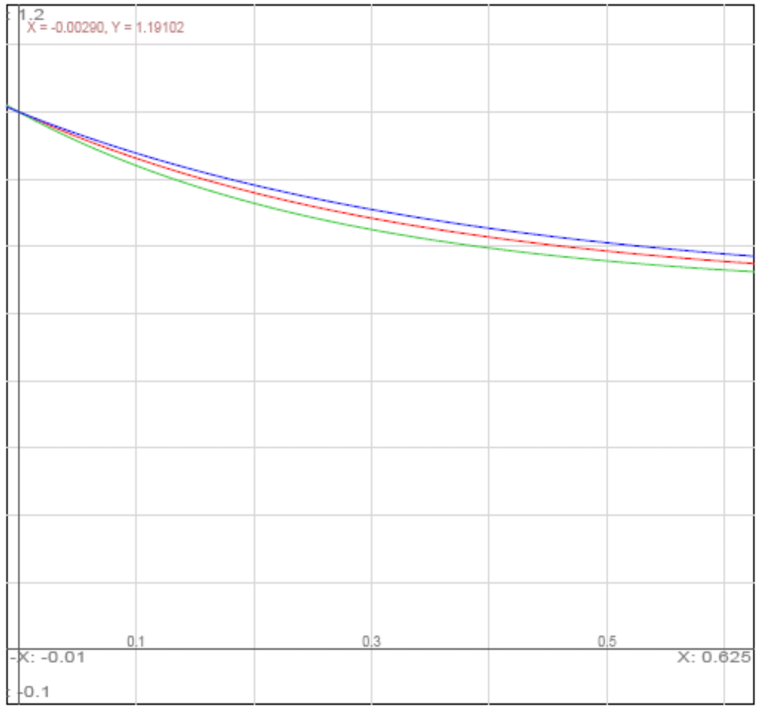
  $u(nh) = 1/3(e^{-3nh}+2)$ (at time-steps)
- Forward-Euler solution

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

  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)

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

  $u(t+h)–u(t) /h = a_{11}*u(t) + a_{12}*v(t) + a_{13}*w(t) + c_1(t)$

  $v(t+h)–v(t) /h = a_{21}*u(t) + a_{22}*v(t) + a_{23}*w(t) + c_2(t)$

  $w(t+h)–w(t) /h= a_{31}*u(t) + a_{32}*v(t) + a_{33}*w(t) + c_3(t)$

Forward-Euler (contd.)

- Rearranging, we get

  $u(t+h) = (1+ha_{11})*u(t) + ha_{12}*v(t) + ha_{13}*w(t) + hc_1(t)$

  $v(t+h) = ha_{21}*u(t) + (1+ha_{22})*v(t) + ha_{23}*w(t) + hc_2(t)$

  $w(t+h) = ha_{31}*u(t) + ha_{32}*v(t) + (1+a_{33})*w(t) + hc_3(t)$

- Introduce vector/matrix notation

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

  $A \quad = \ldots\ldots$

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

Vector notation

- Our systems of equations was

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

- This system can be written compactly as follows

  $\underline{u}(t+h) = (I+hA)\underline{u}(t)+h\underline{c}(t)$

- We can use this form to compute values of $\underline{u}(h)$,$\underline{u}(2h)$,$\underline{u}(3h)$,…
- Forward-Euler is an example of explicit method of discretization
  - key operation: matrix-vector (MVM) multiplication
  - in principle, there is a lot of data-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

  $u(t+h)–u(t) /h = a_{11}*u(t+h) + a_{12}*v(t+h) + a_{13}*w(t+h) + c_1(t+h)$

  $v(t+h)–v(t) /h = a_{21}*u(t+h) + a_{22}*v(t+h) + a_{23}*w(t+h) + c_2(t+h)$

  $w(t+h)–w(t) /h= a_{31}*u(t+h) + a_{32}*v(t+h) + a_{33}*w(t+h) + c_3(t+h)$

- We can write this in matrix notation as follows

  $(I-hA)\underline{u}(t+h) = \underline{u}(t)+h\underline{c}(t+h)$

- Backward-Euler is example of implicit method of discretization
  - key operation: solving a dense linear system $M\underline{x} = \underline{v}$
- How do we solve large systems of linear equations?

Higher-order ode's

- Higher-order ode's can be reduced to systems of first-order ode's
- Example:

  $y'' + y = f(t)$

  Introduce an auxiliary variable $v = y'$

  Then $v' = y''$, so original ode becomes

  $v' = -y + f(t)$

  Therefore, original ode can be reduced to the following system of first order ode's

  $y'(t) = 0*y(t)+ \quad v(t) + 0$

  $v'(t) = -y(t) + 0*v(t) + f(t)$
- We can now use the techniques introduced earlier to discretize this system.
- Interesting point:
  - coefficient matrix A will have lots of zeros (sparse matrix)
  - for large systems, it is important to exploit sparsity to reduce computational effort

Intuition for system

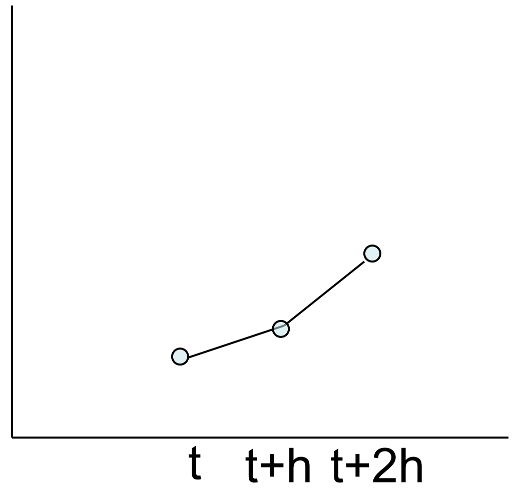
- Discretize system using forward-Euler

  $y(t+h)-y(t)\ /h = v(t)$

  $v(t+h)-v(t)\ /h = -y(t) + f(t)$

- You can eliminate v from this system to get a recurrence relation purely in terms of y

$$\underbrace{\frac{y(t+2h)-2y(t+h)+y(t)}{h^2}}_{\text{Approximation for second derivative}} + y(t) = f(t)$$



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
  $x_{i+1} = x_i - (4x_i+2y_i-8)/4$
  $y_{i+1} = y_i - (3x_i+4y_i-11)/4$

  - 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: general picture

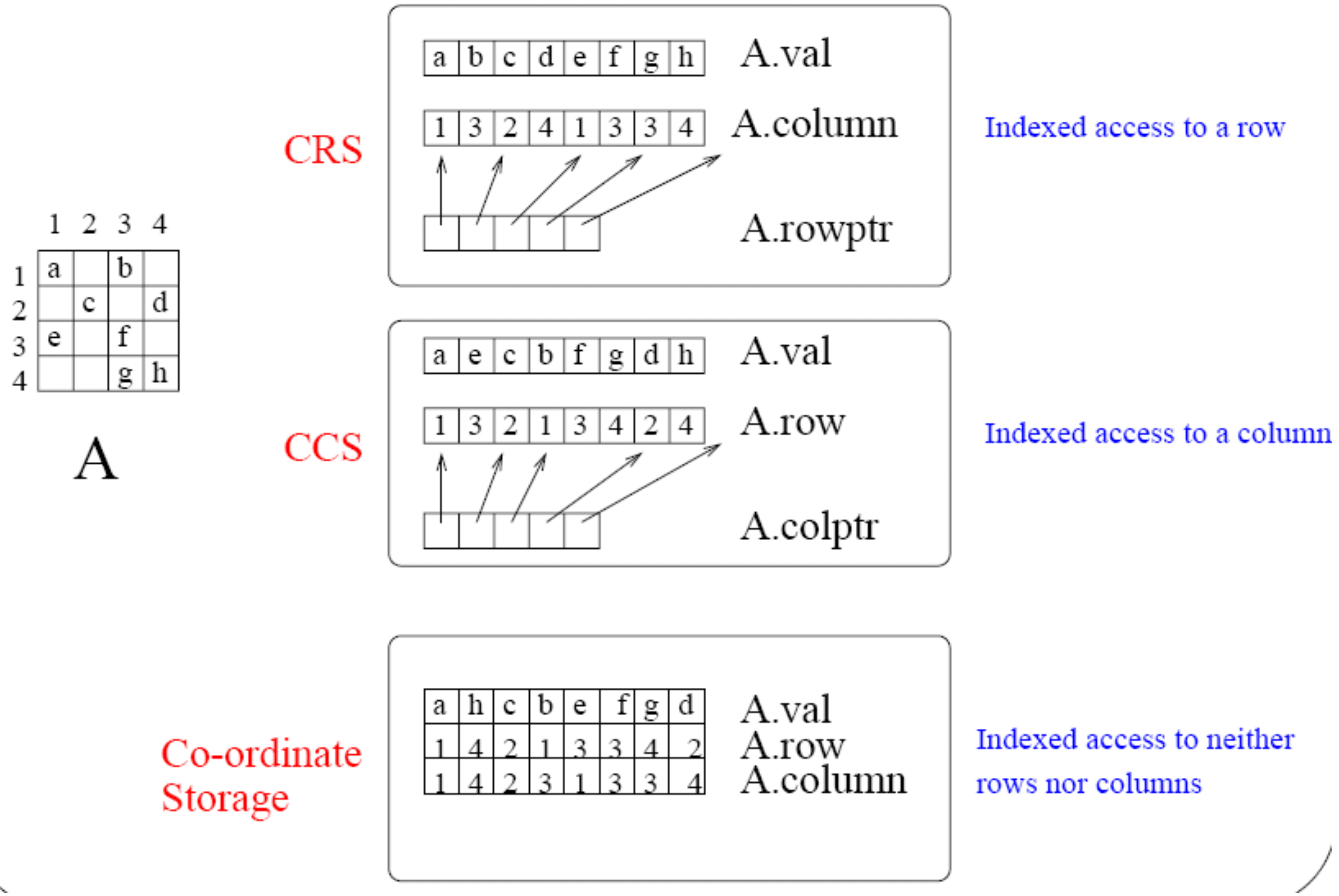
- Linear system $Ax = b$
- Jacobi iteration

  $M^*x_{i+1} = (M-A)x_i + b$ (where M is the diagonal of A)

  This can be written as

  $x_{i+1} = x_i - M^{-1}(Ax_i - b)$
- Key operation:
  - matrix-vector multiplication
- 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

Sparse matrix representations



Indexed access to a row

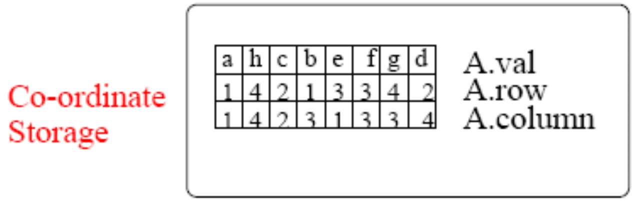
Indexed access to a column

Indexed access to neither rows nor columns

MVM with sparse matrices

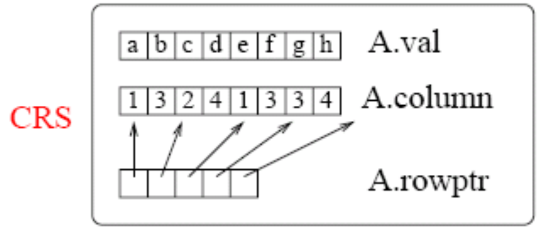
- Coordinate storage

```
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)))
```



Finite-difference methods for solving partial differential equations

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

  $\pm^2 u/\pm x^2 + \pm^2 u/\pm y^2 = f(x,y)$

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

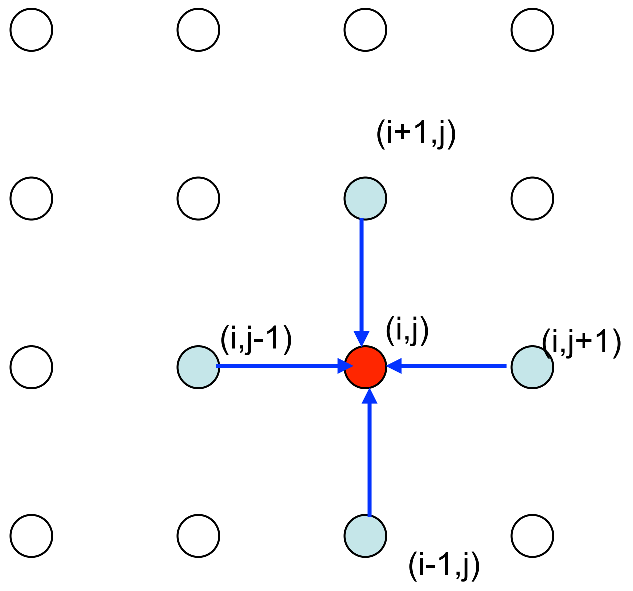
$\pm^2 u/\pm x^2 = ((u(i,j+1)-u(i,j))/h - (u(i,j)-u(i,j-1))/h)/h$

$\pm^2 u/\pm y^2 = ((u(i+1,j)-u(i,j))/h - (u(i,j)-u(i-1,j))/h)/h$

- 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

8 (i,j) such that (i,j) is an interior point

$u(i,j+1)+u(i,j-1)+u(i+1,j)+u(i-1,j) – 4u(i,j) = h^2 f(ih,jh)$



<u>5-point stencil</u>

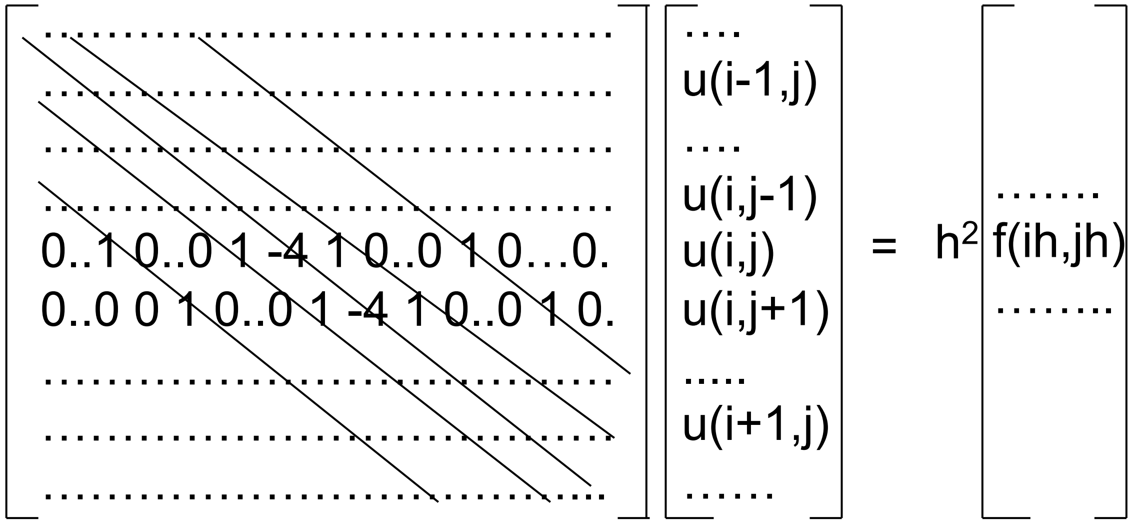
Finite-difference methods for solving partial differential equations contd.)

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

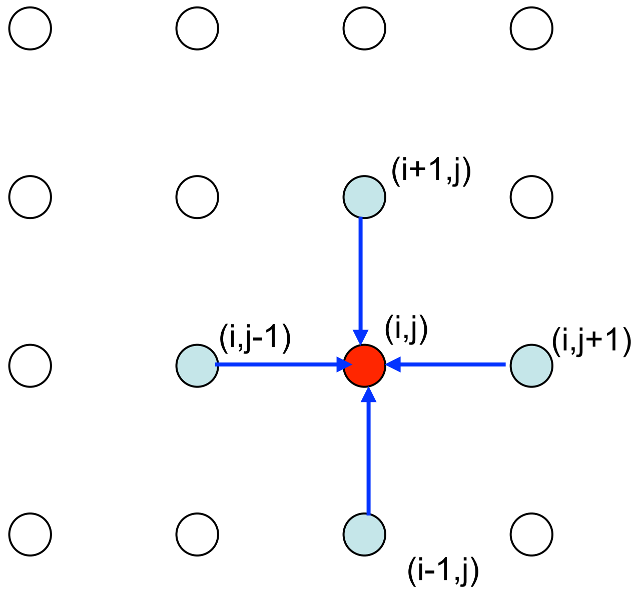
8 (i,j) such that (i,j) is an interior point

$$u(i,j+1)+u(i,j-1)+u(i+1,j)+u(i-1,j) – 4u(i,j) = h^2 f(ih,jh)$$

- Matrix notation: use row-major (natural) order for u's



Pentadiagonal sparse matrix



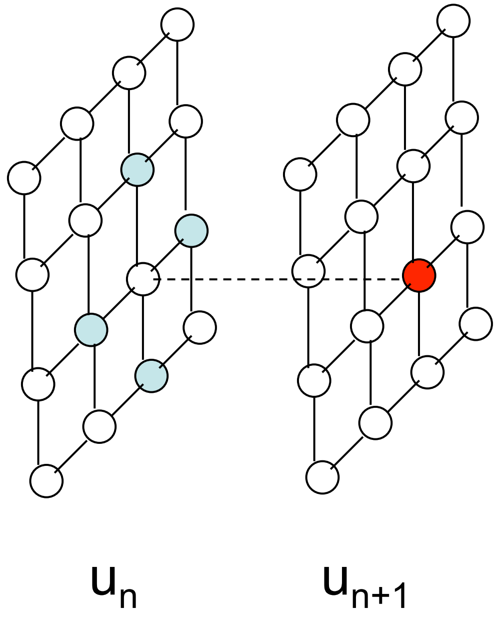
5-point stencil

Since matrix is sparse, we should use an iterative method like Jacobi.

Implementation of Jacobi iteration for this problem

- Data structures:
  - coefficient matrix is known at compile-time, so inline it into the code
  - values of u at a given time-step are stored in 2-D array
  - we use two arrays to hold values of u at successive time-steps and copy between them

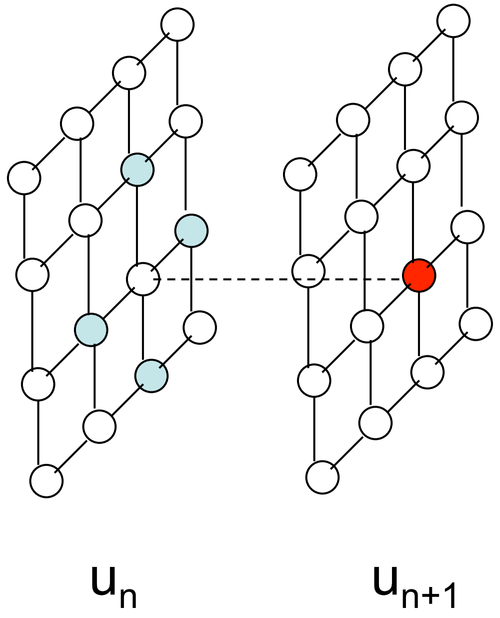
```
//Jacobi iteration
//initialize array u
for time = 1, nsteps
    for i = 2,n-1
      for j = 2,n-1
         temp(i,j)=0.25*(u(i-1,j)+u(i+1,j)+u(i,j-)+u(i,j+1))
    for i = 2,n-1
      for j = 2,n-1
         u(i,j) = temp(i,j)
```



Jacobi iteration with 5-point stencil

Finite-difference methods for solving partial differential equations (contd.)

- Known as stencil codes
- Example shown is Jacobi iteration with five-point stencil
  - many other stencils are used in practice
- Parallelism
  - all interior points can be computed in parallel
  - parallelism is independent of runtime values



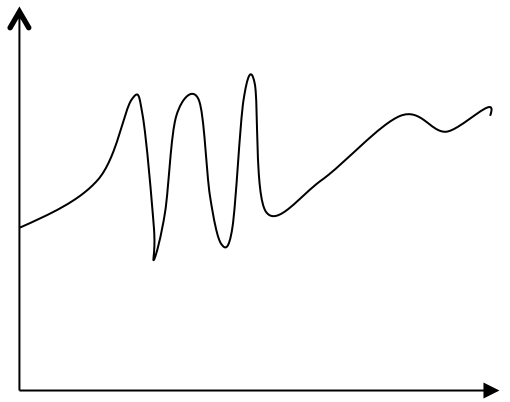
Jacobi iteration with 5-point stencil

Comment on Sparse MVM

- At an abstract level
  - algorithm: matrix-vector multiplication
  - data structures: four sparse representations
    - coordinate storage
    - compressed-row storage
    - compressed-column storage
    - “inlined” into code (stencil)
- Programs:
  - algorithm and data structure are intertwined, making them hard to understand for humans as well as transformation systems

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

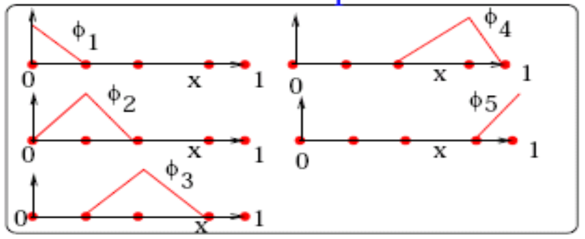


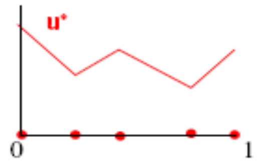
Finite-element methods

- 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 Á$_0$, Á$_1$, Á$_2$,…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 Á$_i$
    - weighted residual method: similar in spirit to what we do in Fourier analysis, but more complex because basis functions are not necessarily orthogonal

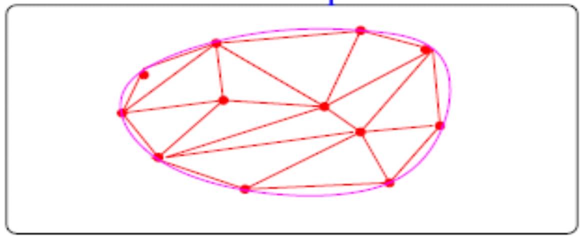
Mesh generation and refinement

1-D example





2-D example



Mesh generation

- 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

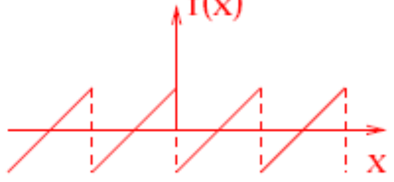
Finding coefficients

- 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 $10^6 \times 10^6$
    - 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 Residual Technique:

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

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

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

If the differential equation is linear:

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

$k = 1,2,\ldots N$

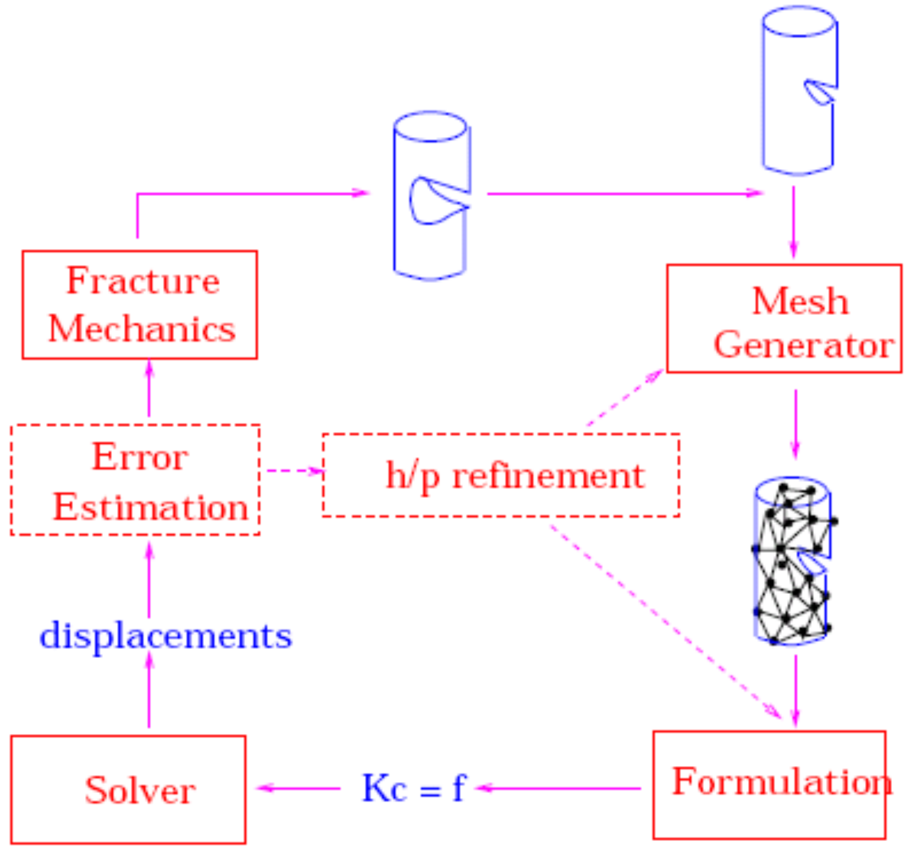
This system can be written as

$K c = b$ 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

Flow-chart of Adaptive Finite-element Simulation of Fracture

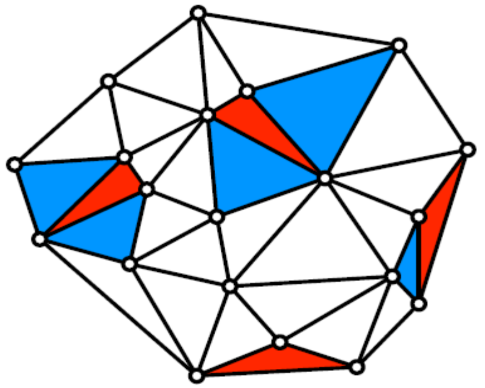


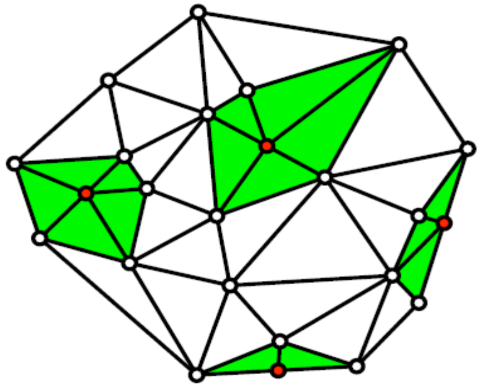
Delaunay Mesh Refinement

- Iterative refinement to remove badly shaped triangles:

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

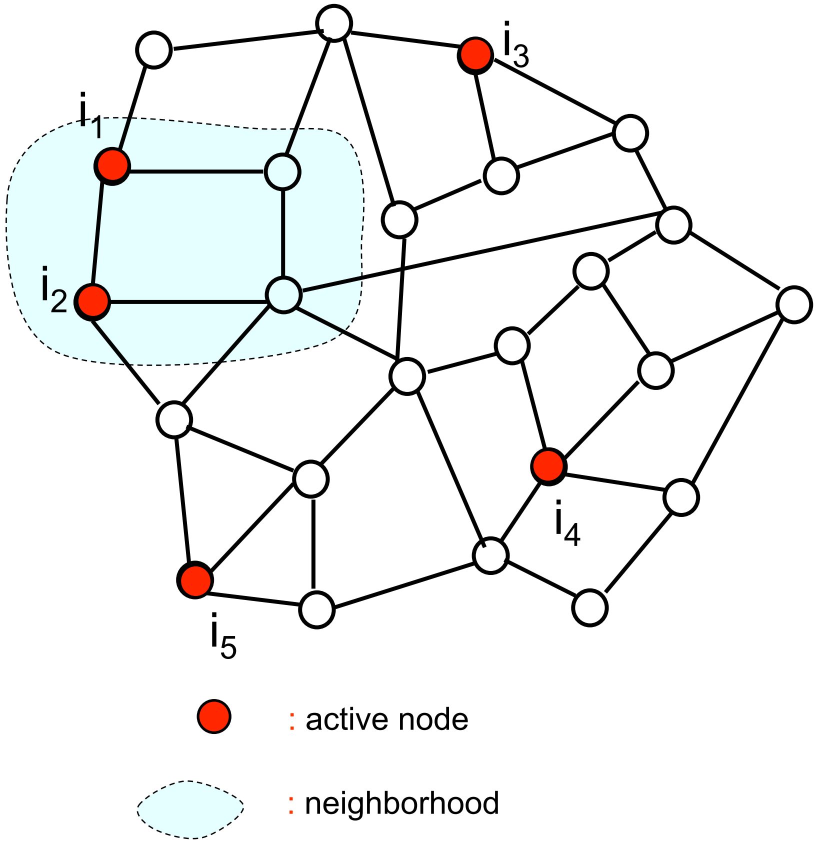
- Don't-care non-determinism:
  - final mesh depends on order in which bad triangles are processed
  - applications do not care which mesh is produced
- Data structure:
  - graph in which nodes represent triangles and edges represent triangle adjacencies
- Parallelism:
  - bad triangles with cavities that do not overlap can be processed in parallel
  - parallelism is dependent on runtime values
    - compilers cannot find this parallelism
  - (Miller et al) at runtime, repeatedly build interference graph and find maximal independent sets for parallel execution


Before


After

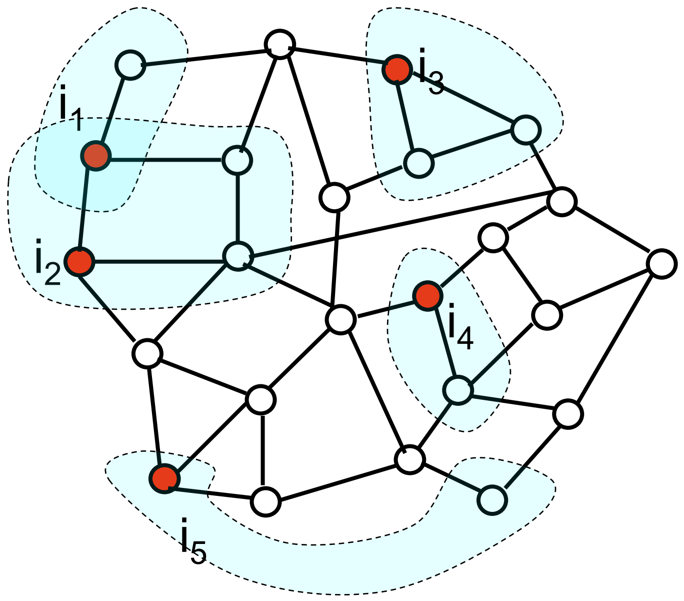
Operator formulation of algorithms

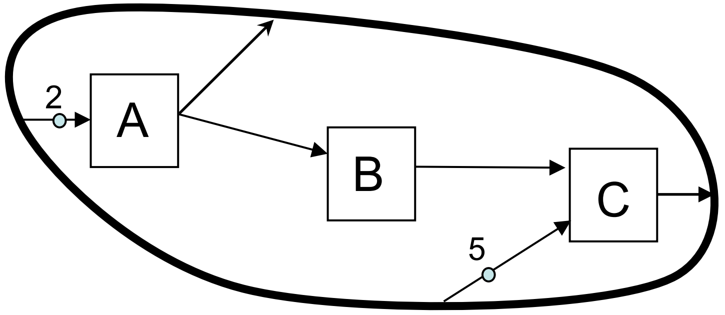
- Algorithm =
  repeated application of operator to graph
  - active element:
    - node or edge where computation is needed
      - DMR: nodes representing bad triangles
      - Event-driven simulation: station with incoming message
      - Jacobi: interior nodes of mesh
  - neighborhood:
    - set of nodes and edges read/written to perform computation
      - DMR: cavity of bad triangle
      - Event-driven simulation: station
      - Jacobi: nodes in stencil
    - distinct usually from neighbors in graph
  - ordering:
    - order in which active elements must be executed in a sequential implementation
      - any order (Jacobi,DMR, graph reduction)
      - some problem-dependent order (event-driven simulation)



Parallelism

- Amorphous data-parallelism
  - active nodes can be processed in parallel, subject to
    - neighborhood constraints
    - ordering constraints
- Computations at two active elements are *independent* if
  - Neighborhoods do not overlap
  - More generally, neither of them writes to an element in the intersection of the neighborhoods
- Unordered active elements
  - Independent active elements can be processed in parallel
  - How do we find independent active elements?
- Ordered active elements
  - Independence is not enough
  - How do we determine what is safe to execute w/o violating ordering?





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)

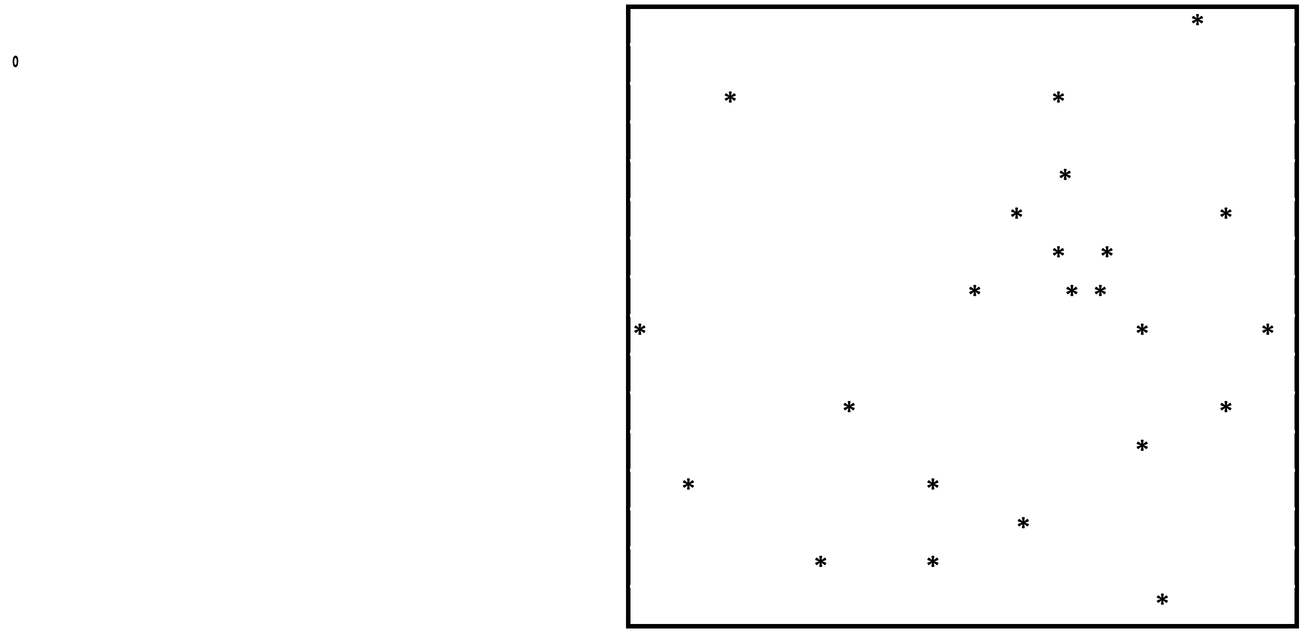
Barnes Hut Idea

- 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

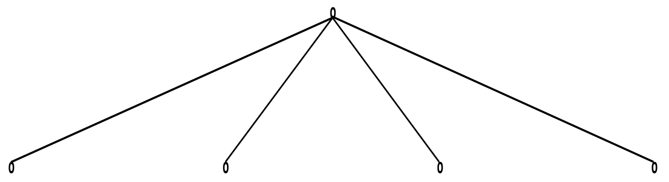
- Set bodies' initial position and velocity
- Iterate over time steps
  1. Subdivide space until at most one body per cell
     - Record this spatial hierarchy in an octree
  2. Compute mass and center of mass of each cell
  3. Compute force on bodies by traversing octree
     - Stop traversal path when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

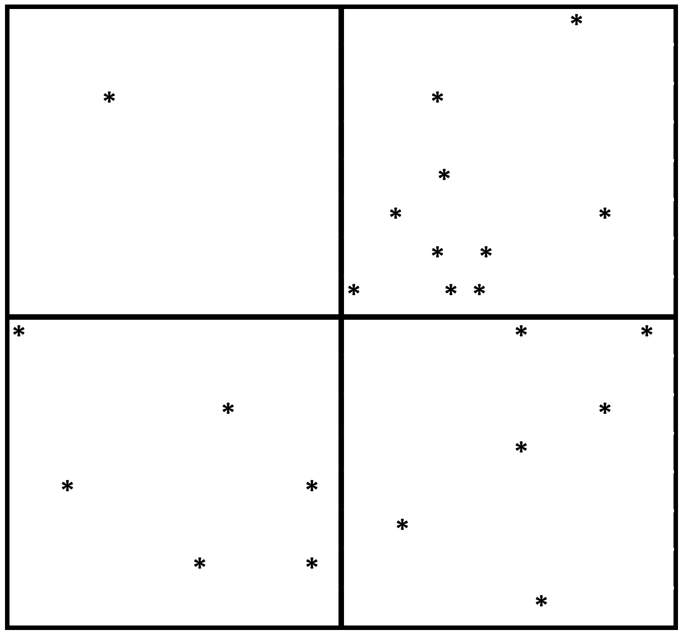
Build Tree (Level 1)



Subdivide space until at most one body per cell

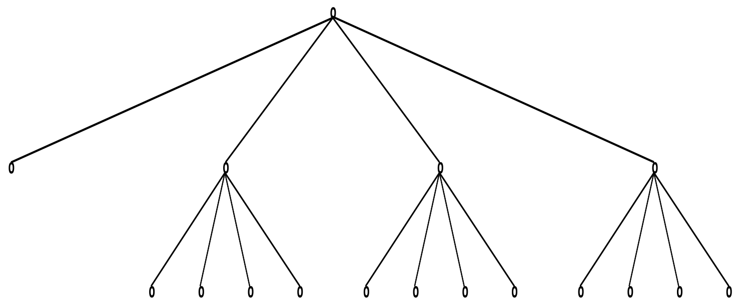
Build Tree (Level 2)

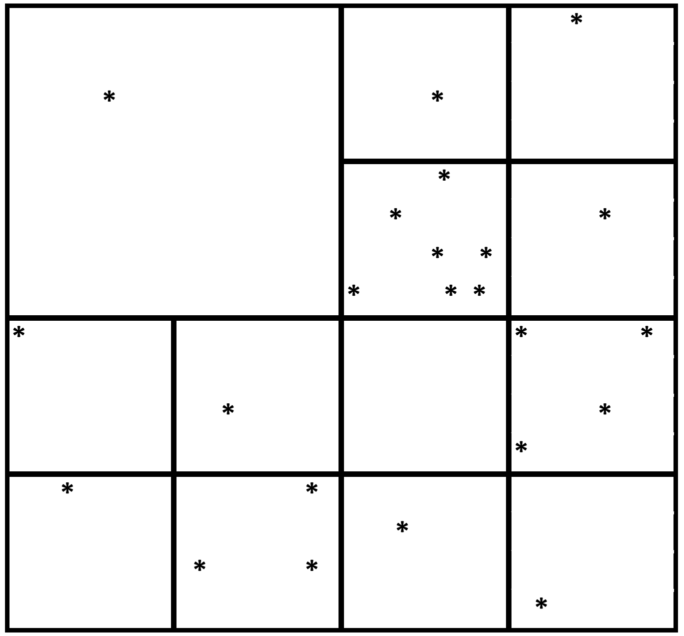




Subdivide space until at most one body per cell

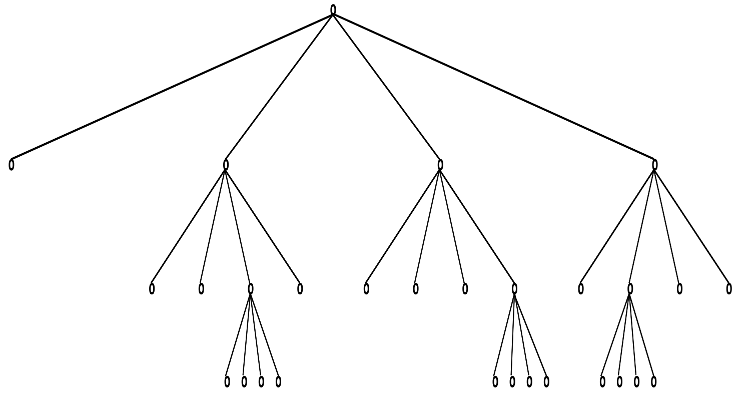
Build Tree (Level 3)

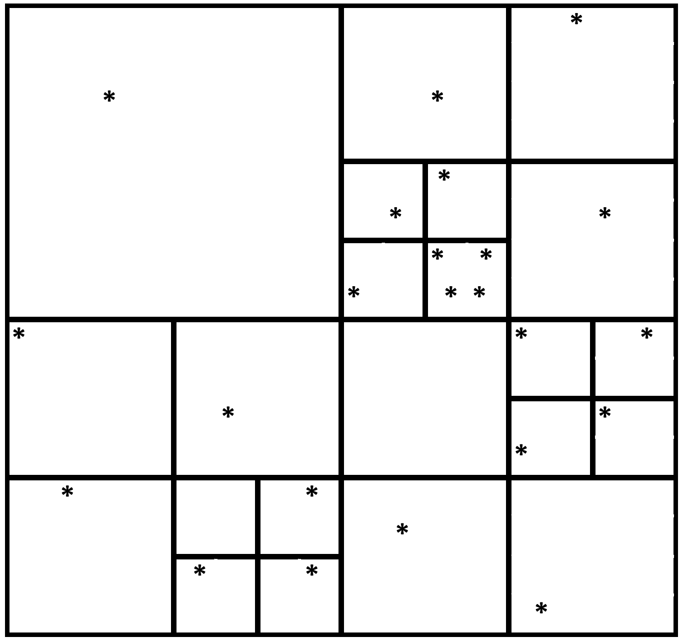




Subdivide space until at most one body per cell

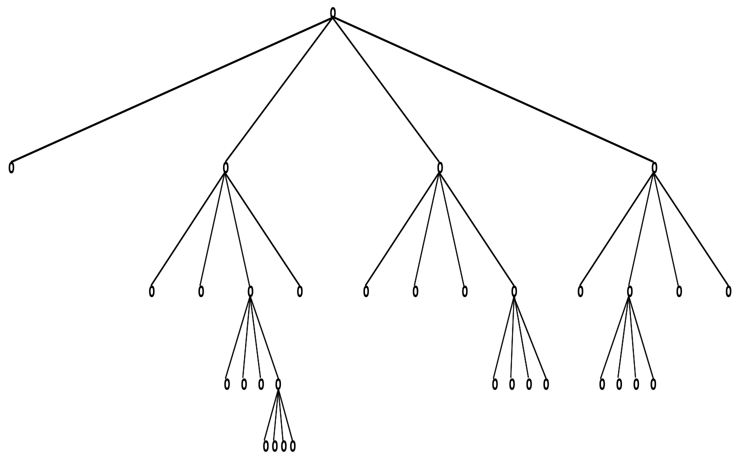
Build Tree (Level 4)

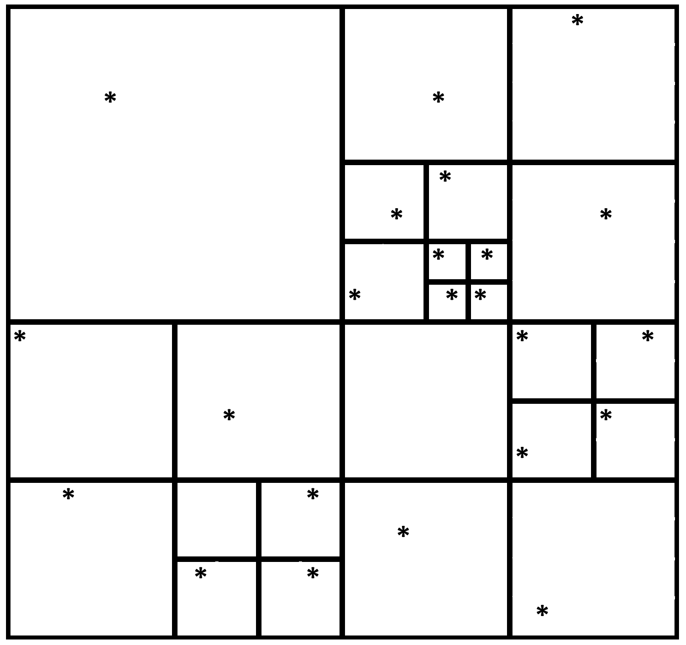




Subdivide space until at most one body per cell

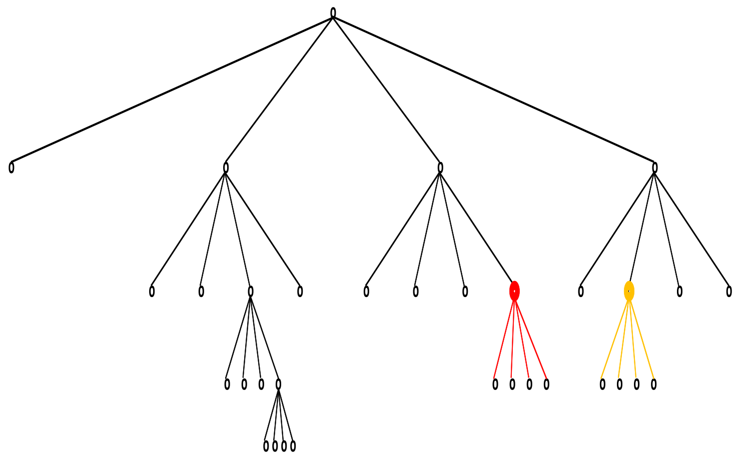
Build Tree (Level 5)

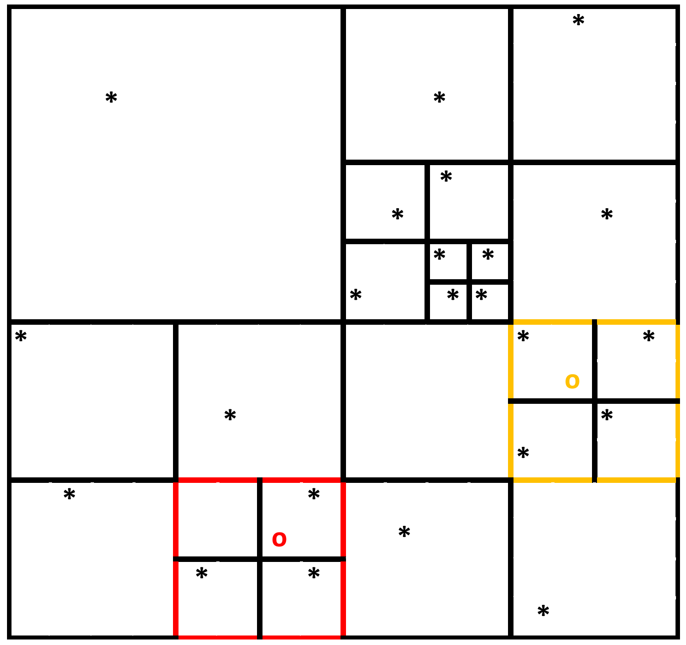




Subdivide space until at most one body per cell

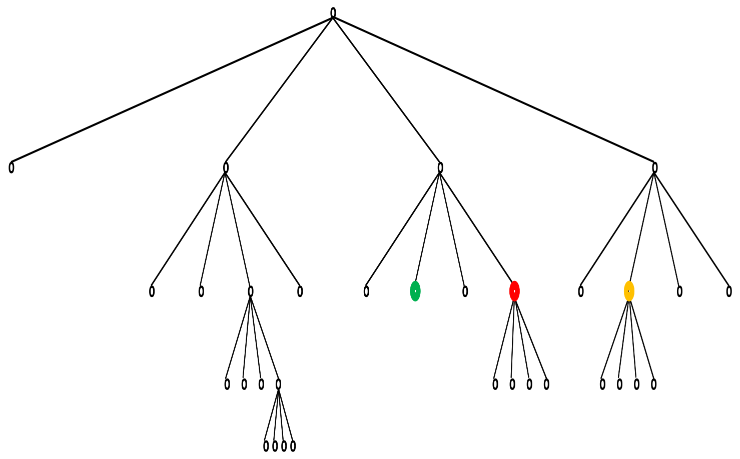
Compute Cells’ Center of Mass

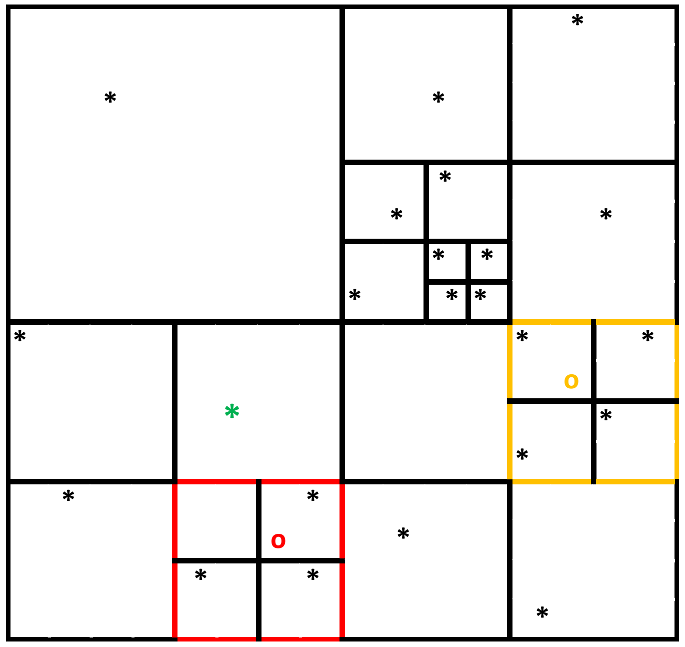




For each internal cell, compute sum of mass and weighted average of position of all bodies in subtree; example shows two cells only

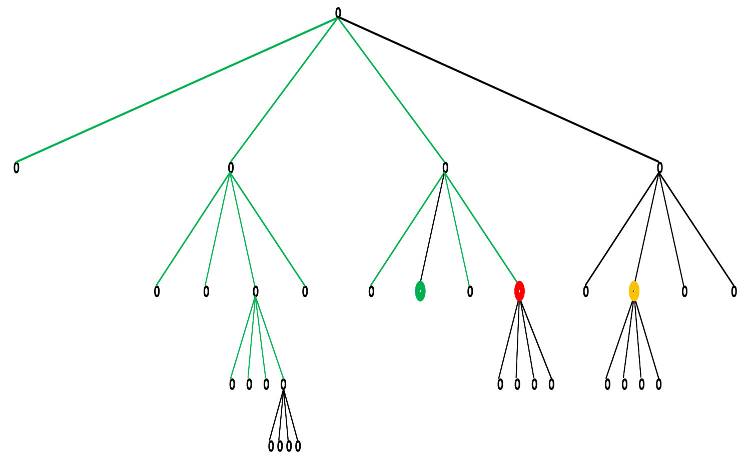
Compute Forces

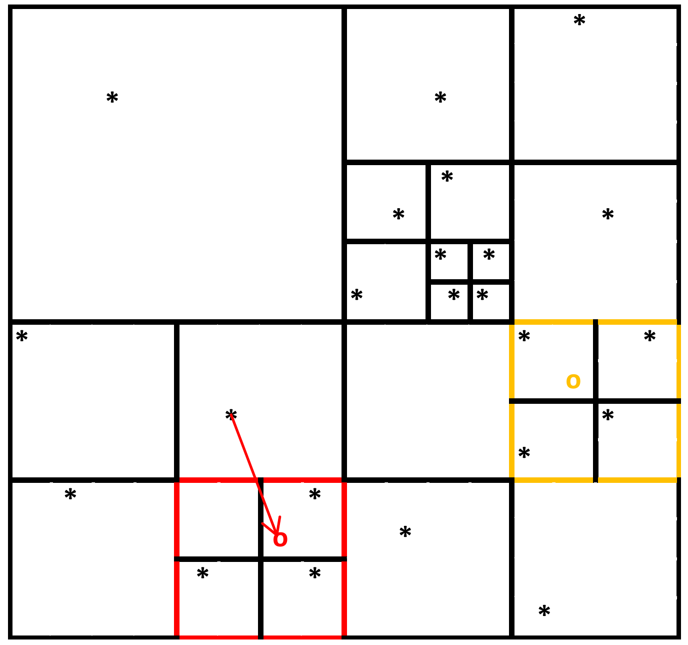




Compute force, for example, acting upon green body

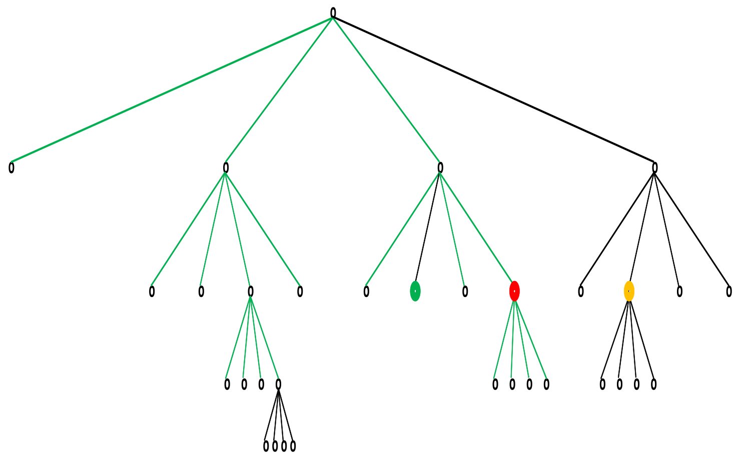
Compute Force (short distance)

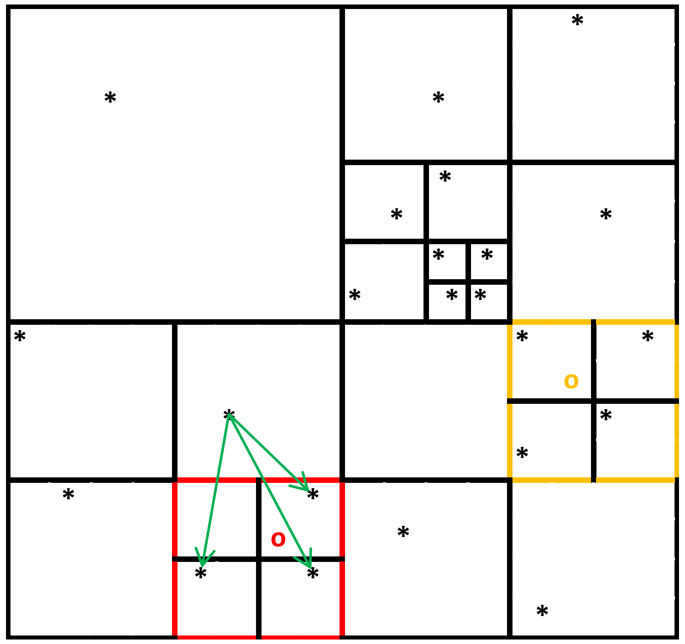




Scan tree depth first from left to right; green portion already completed

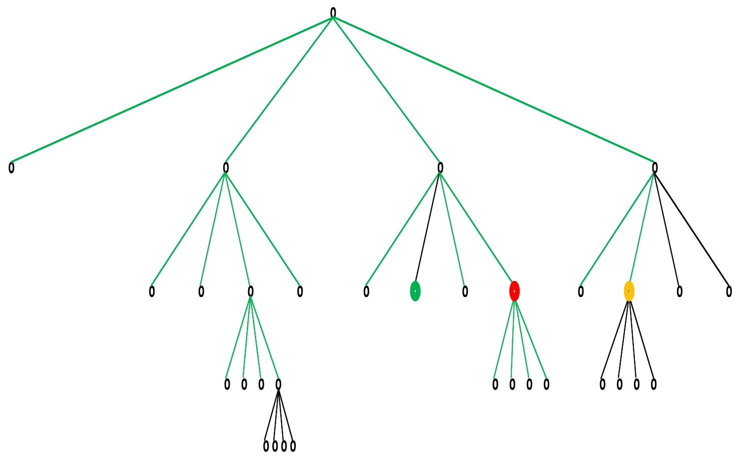
Compute Force (down one level)

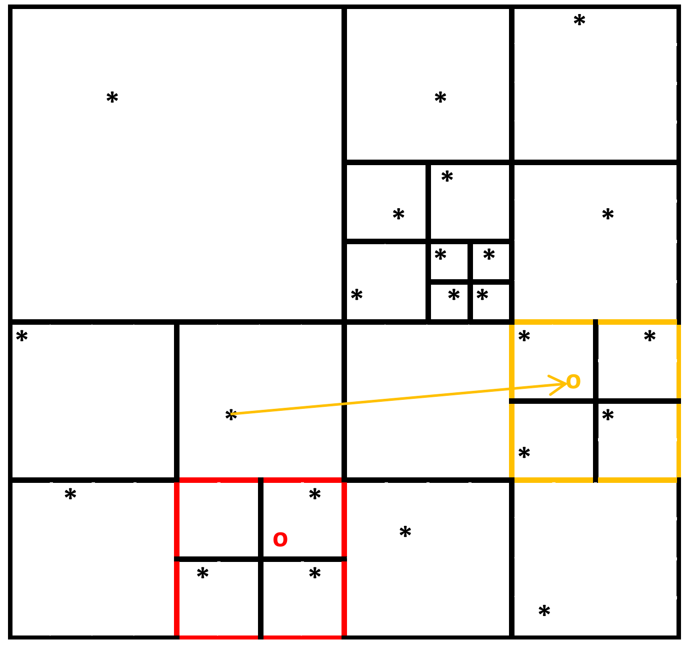




Red center of mass is too close, need to go down one level

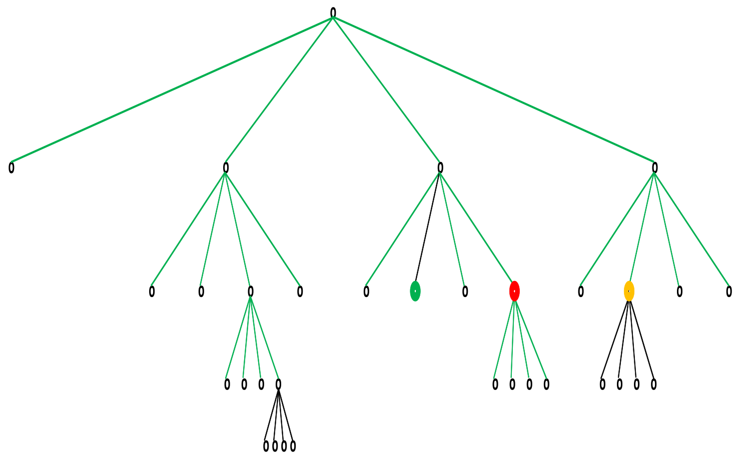
Compute Force (long distance)

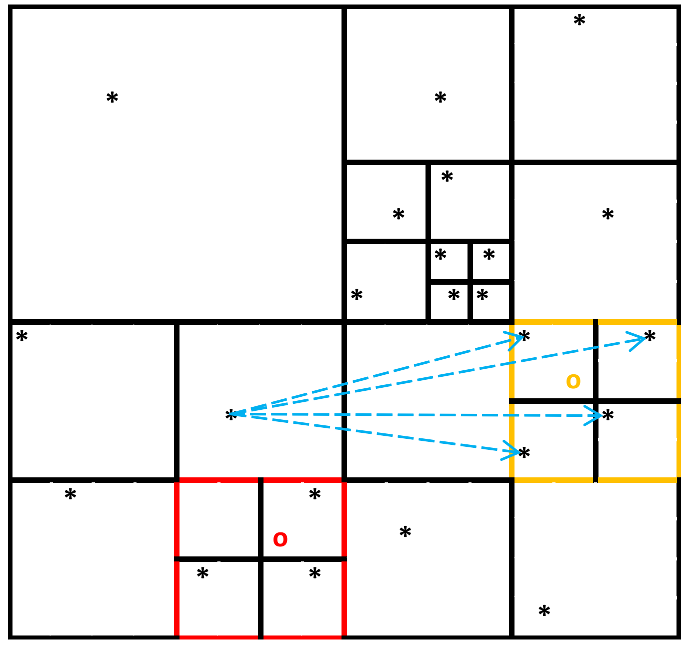




Yellow center of mass is far enough away

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();
  }
}
```

Amorphous Data-Parallelism (1)

- Top-down tree building
  - Topology: tree
  - Operator: morph (refinement)
  - Ordering: unordered
  - Active nodes: new nodes
  - Neighborhoods: active nodes and their parents (the path leading to the parent is only read)
  - Parallelism: increasing from none to a lot

Amorphous Data-Parallelism (2)

- Bottom-up tree summarization
  - Topology: tree
  - Operator: local computation (structure driven)
  - Ordering: ordered (children first, priority is determined by tree level)
  - Active nodes: internal nodes
  - Neighborhoods: active nodes and their children
  - Parallelism: decreasing from a lot to none

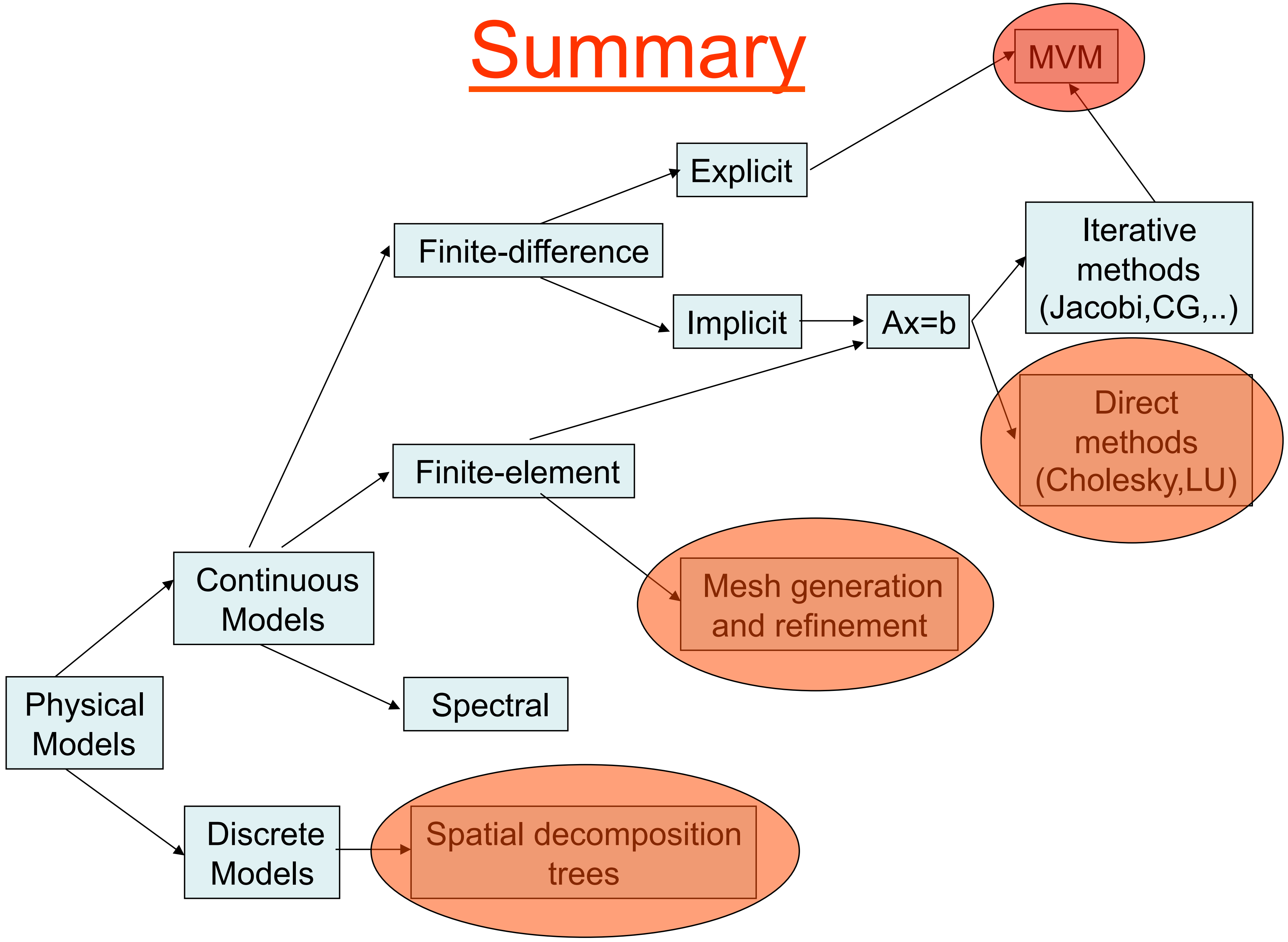
Amorphous Data-Parallelism (3)

- Force computation
  - Topology: tree + set
  - Operator: reader + local computation (structure driven)
  - Ordering: unordered/unordered
  - Active nodes: nodes in set
  - Neighborhoods: active nodes (the tree is only read)
  - Parallelism: full

Amorphous Data-Parallelism (4)

- Advancing bodies
  - Topology: set
  - Operator: local computation (structure driven)
  - Ordering: unordered
  - Active nodes: nodes
  - Neighborhoods: active nodes
  - Parallelism: full

Summary


MVM
Explicit
Iterative methods (Jacobi,CG,..)
Finite-difference
Implicit
Ax=b
Direct methods (Cholesky,LU)
Finite-element
Mesh generation and refinement
Continuous Models
Spectral
Physical Models
Discrete Models
Spatial decomposition trees

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
    - data structures: both dense and sparse matrices
  - stencil computations:
    - source: finite-difference methods for pde's
    - data structures: dense matrices
  - A=LU:
    - terminology: direct methods for solving linear systems, factorization
    - source: boundary-element methods
    - data structures: usually only dense matrices
    - comment: high-performance factorization codes use MMM as a kernel
  - mesh generation and refinement
    - source: finite-element methods
    - data structures: graphs
  - tree construction and traversal
    - source: n-nody methods
    - data structures: spatial decomposition tree

Summary (contd.)

- Terminology
  - regular algorithms:
    - dense matrix computations like MVM, A=LU, stencil computations
    - parallelism in algorithms is independent of runtime values, so all parallelization decisions can be made at compile-time
  - semi-regular algorithms:
    - sparse matrix computations like MVM, A=LU
    - parallelization decisions can be made at runtime once matrix is available, but before computation is actually performed
    - inspector-executor approach (see later)
  - irregular algorithms:
    - graph computations like mesh generation and refinement
    - parallelism in algorithms is dependent on runtime values
    - most parallelization decisions have to be made at runtime during the execution of the algorithm