Partial Differential Equations in HPC, I

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2013

Types of PDEs

Recapitulate - types of partial differential equations (2nd order, 2D):

$$A\phi_{xx} + 2B\phi_{xy} + C\phi_{yy} + D\phi_x + E\phi_y + F = 0$$

Three major classes:

- Hyperbolic (often time dependent, e.g., wave equation)
- Elliptic (steady state, e.g., Poisson's Equation)
- Parabolic (often time dependent, e.g. diffusion equation)
- (nonlinear when coefficients depend on ϕ)

Hyperbolic Problems

Hyperbolic problems:

$$\det(Z) = \begin{vmatrix} A & B \\ B & C \end{vmatrix} < 0$$

- Examples: Wave equation
- Explicit time steps
- Solution at point depends on neighboring points at prior time step

Elliptic Problems

Elliptic Problems:

$$Z = \begin{pmatrix} A & B \\ B & C \end{pmatrix} = \text{positive definite}$$

- Examples: Laplace equation, Poisson Equation
- Steady-state problems
- Solution at points depends on all other points
- Locality is tougher than hyperbolic problems

Parabolic Problems

Parabolic Problems:

$$\det(Z) = \begin{vmatrix} A & B \\ B & C \end{vmatrix} = 0$$

- Examples: Heat equation, Diffusion equation
- Needs an elliptic solve at each time step

Elements, Differences, Discretizations ...

There are many forms of discretization that can be used on these systems:

- Finite Elements
- Finite Differences
- Adaptive combinations of the above

for our purposes, though, we can illustrate many of the basic ideas using a simple one-dimensional example on a regular mesh.

Simple Elliptic Example in 1D

For starters we consider the following boundary value problem:

$$-\frac{d^2\phi}{dx^2} = f(x), \qquad x \in [0,1],$$

with initial conditions:

$$d\phi/dx(0) = 0, \qquad \phi(1) = 0.$$

The plus side of this example is that we have an integral general solution:

$$\phi(x) = \int_x^1 dt \int_0^t f(s)ds$$

Mesh Approach

Now we apply a *regular* mesh, i.e. a uniform one defined by N points uniformly distributed over the domain:

$$x_i = (i-1)\Delta x, \qquad \Delta x = h = 1/N.$$

Using a *mid-point* formula for the derivative:

$$2h d\phi/dx(x) = \phi(x + h/2) - \phi(x - h/2),$$

Applying this *central difference* formula twice yields

$$-\phi(x-h) + 2\phi(x) - \phi(x+h) = h^2 d^2\phi/dx^2(x),$$

or in another form

$$-\phi_{n-1} + 2\phi_n - \phi_{n+1} = h^2\phi_n'',$$

So now our original boundary value problem can be expressed in *finite differences* as

$$-\phi_{n-1} + 2\phi_n - \phi_{n+1} = h^2 f_n$$

and this is now easily reduced to a matrix problem, of course. But before we go there, we can generalize the above to use an arbitrary mesh, rather than uniform ...

Finite Differences on an Arbitrary Mesh

Now consider a (possibly) non-uniform mesh, discretized according to $0 = x_0 < x_1 < ... < x_{N+1} = 1$. The discretized boundary value problem then becomes

$$-\frac{2}{x_{n+1} - x_{n-1}} \left(\frac{\phi_{n+1} - \phi_n}{x_{n+1} - x_n} - \frac{\phi_n - \phi_{n-1}}{x_n - x_{n-1}} \right) = f(x_n)$$

Let $h_n = x_n - x_{n-1}$, and we can scale by $(h_{n+1} - h_n)/2$:

$$\begin{aligned} \frac{h_{n+1} - h_n}{2} f(x_n) &= -\frac{\phi_{n+1} - \phi_n}{h_{n+1}} + \frac{\phi_n - \phi_{n-1}}{h_n}, \\ &= \left(\frac{1}{h_{n+1}} + \frac{1}{h_n} \right) \phi_n - \frac{1}{h_{n+1}} \phi_{n+1} - \frac{1}{h_n} \phi_{n-1}. \end{aligned} \tag{1}$$

Eq. 1 is a symmetric *tridiagonal* matrix, which can be shown to be positive definite. The i-th row ($1 < i < N$) has entries:

$$a_{i,i-1} = -\frac{1}{h_i} \qquad a_{i,i} = \frac{1}{h_i} + \frac{1}{h_{i+1}} \qquad a_{i,i+1} = -\frac{1}{h_{i+1}}$$

and the resulting matrix equation looks like:

$$\mathbf{A}\Phi = \mathbf{F}$$

and we can subject this matrix equation to our full arsenal of matrix solution methods.

Finite Elements for the Boundary Value Problem

Using a mesh-based discretization is not the only way, of course. The **finite element** method uses piecewise-linear "hat" functions, u_i which are unity at the i-th node, x_i, and zero elsewhere. The mathematical basis is the *variational* formulation:

$$\int_0^1 \phi'(x) u_i'(x) dx = \int_0^1 f(x) u_i(x) dx,$$

and an approximation scheme for the general integral, e.g. the *trapezoidal* rule:

$$\int_{x_i}^{x_{i+1}} f(x) dx \simeq \frac{\Delta x}{2} \left(f(x_i) + f(x_{i+1}) \right).$$

Matrix Factorization

This formulation reduces to the same form as our finite difference one, and can also be recast for an arbitrary mesh.
Regardless of whether we used finite elements or finite differences, our simple 1D example reduced to the simple tridiagonal form:

$$\mathbf{A}\Phi = \mathbf{F},$$

where ...

the tridiagonal matrix **A** is given by:

$$\mathbf{A} = \begin{pmatrix} 1 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & -1 & 2 & -1 & 0 \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & \dots & 0 & 0 & -1 & 2 \end{pmatrix}$$

and $F_1 = h^2 f(x_1)/2$, $F_i = h^2 f(x_i)$.

Direct Solution

The tridiagonal matrix **A** can be factored very simply into a product of lower (**L**) and upper ($\mathbf{L}^{\mathrm{T}}$) bidiagonal matrices:

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ -1 & 1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & -1 & 1 & 0 & 0 \\ 0 & \dots & 0 & -1 & 1 & 0 \\ 0 & \dots & 0 & 0 & -1 & 1 \end{pmatrix}$$

$$\mathbf{L}^{\mathrm{T}} = \begin{pmatrix} 1 & -1 & 0 & 0 & \dots & 0 \\ 0 & 1 & -1 & 0 & \dots & 0 \\ 0 & 0 & 1 & -1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & 1 & -1 & 0 \\ 0 & \dots & 0 & 0 & 1 & -1 \\ 0 & \dots & 0 & 0 & 0 & 1 \end{pmatrix}$$

note that $\mathbf{L}^{\mathrm{T}}$ is just the transpose of $\mathbf{L}$.

So this starts to look quite familiar to our old friend, LU decomposition. We introduce the auxiliary vector **Y** which solves by forward substitution:

$$\mathbf{LY} = \mathbf{F},$$

and then the solution Φ is given by backward solution

$$\mathbf{L}^{\mathrm{T}}\Phi = \mathbf{Y},$$

The pseudo-code for this forward and backward solution is given by the following:

```
y(1) = f(1)
do i=2,N
   y(i) = y(i-1) + f(i)
end do
phi(N) = y(N)
do i=N-1,1,-1
   phi(i) = phi(i-1) + y(i)
end do
```

Parallel Prefix for Tridiagonal Systems

Let us consider the solution of a general tridiagonal system given by:

$$\mathbf{A} = \begin{pmatrix} a_1 & b_1 & 0 & 0 & \dots & 0 \\ c_1 & a_2 & b_2 & 0 & \dots & 0 \\ 0 & c_2 & a_3 & b_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix}$$

which we factor:

$$\mathbf{A} = \mathbf{LU}$$

and the factors **L** and **U** are lower and upper bidiagonal matrices, respectively,

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ l_1 & 1 & 0 & 0 & \dots & 0 \\ 0 & l_2 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix}$$

$$\mathbf{U} = \begin{pmatrix} d_1 & e_1 & 0 & 0 & \dots & 0 \\ 0 & d_2 & e_2 & 0 & \dots & 0 \\ 0 & 0 & d_3 & e_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix}$$

Matching up terms in $A_{ij} = (LU)_{ij}$ yields the following three equations:

$$c_{i-1} = l_{i-1} d_{i-1} \qquad (i, i-1) \qquad (2)$$

$$a_i = l_{i-1} e_{i-1} + d_i \qquad (i, i) \qquad (3)$$

$$b_i = e_i \qquad (i-1, i) \qquad (4)$$

From Eq. 2 we obtain $l_{i-1} = c_{i-1}/d_{i-1}$ and substituting into Eq. 3 yields a recurrence relation for d_i:

$$d_i = a_i - \frac{c_{i-1} e_{i-1}}{d_{i-1}} = a_i + \frac{f_i}{d_{i-1}},$$

where $f_i = -c_{i-1} e_{i-1}$.

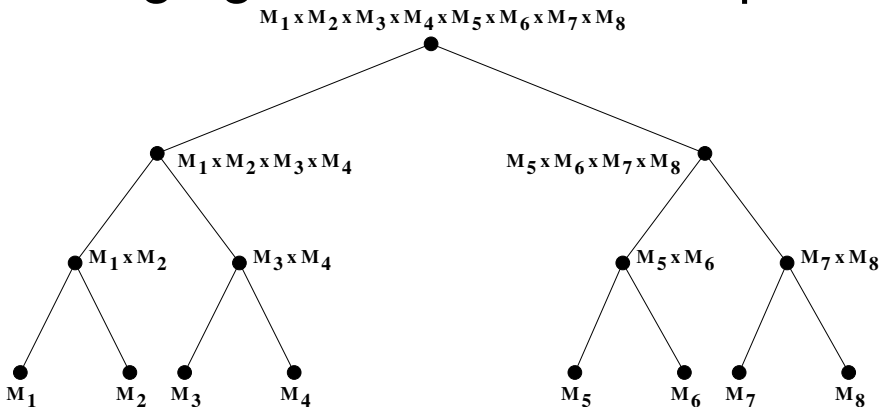
The **parallel prefix** method is then used for solving the recurrence relation for the d_i by defining $d_i = p_i/q_i$, and expressing in matrix form:

$$\begin{aligned}\begin{pmatrix} p_i \\ q_i \end{pmatrix} &= a_i + f_i q_{i-1}/p_{i-1} = \begin{pmatrix} a_i & f_i \\ 1 & 0 \end{pmatrix} \begin{pmatrix} p_{i-1} \\ q_{i-1} \end{pmatrix} \\ &= M_i \begin{pmatrix} p_{i-1} \\ q_{i-1} \end{pmatrix} = M_i M_{i-1} \ldots M_1 \begin{pmatrix} p_0 \\ q_0 \end{pmatrix} \end{aligned} \tag{5}$$

where $f_1 = 0$, $p_0 = 1$, and $q_0 = 0$. In other words, we can write the recurrence relation as the direct product of a sequence of $2x2$ matrix multiplications.

Parallel Prefix

The parallel prefix method is then used (in parallel) to quickly evaluate the product in Eq. 5 by using a binary tree structure to quickly evaluate the individual products. The following figure illustrates this process (albeit for only eight terms):



Note that this procedure is quite generally valid for any associative operation (i.e. $(A \otimes B) \otimes C = A \otimes (B \otimes C)$), and can be done in only $\mathcal{O}(log_2 N)$ steps.

Parallel Efficiency

The time taken for this parallel method is then given by:

$$\tau(P) \simeq \frac{2c_1 n}{P} + 2c_2 \log_2 P,$$

where c_1 is the time taken for 1 Flop, and c_2 that for one step of parallel prefix using 2x2 matrices. The parallel efficiency is then just

$$\frac{1}{E(P)} \leq 2 + \frac{2\gamma P \log_2 P}{n},$$

where $\gamma = c_2/c_1$.

$$E(P) \geq \frac{1}{2 + 2\gamma\mu},$$

with $\mu n \geq P \log_2 P$.

Recapitulate

Recall that in solving the matrix equation $\mathbf{Ax} = \mathbf{b}$ Jacobi's iterative method consisted of repeatedly updating the values of $\mathbf{x}$ based on their value in the previous iteration:

$$x_i^{(k)} = \frac{1}{A_{i,i}} \left(b_i - \sum_{j \neq i} A_{i,j} x_j^{(k-1)} \right),$$

we will come back to convergence properties of this relation a bit later. Now, however, we have a matrix $\mathbf{A}$ that has been determined by our choice of a simple stencil for the derivatives in our PDE, which is very *sparse*, making for a large reduction in the workload.

Iterative Techniques

Now we consider, for our simple 1D boundary value problem used in the last section, *iterative methods* for solution. The simplest scheme is **Jacobi iteration**. For the general mesh in 1D,

$$\phi_n^{k+1} = \frac{h_n}{h_{n+1}+h_n}\phi_{n+1}^k + \frac{h_{n+1}}{h_{n+1}+h_n}\phi_{n-1}^k + \frac{h_n h_{n+1}}{2} f(x_n),$$

whereas for our even simpler uniform mesh:

$$\begin{aligned}
\phi_1^{k+1} &= \phi_2^k + (h^2/2) f(x_1), \\
\phi_n^{k+1} &= \left(\phi_{n+1}^k + \phi_{n-1}^k + h^2 f(x_n)\right)/2. \qquad n \in [2, N]
\end{aligned}$$

Recall that $x_{N+1} = 1$, and $\phi(1) = 0$.

Parallelization Strategies for Jacobi

First, in the classical Jacobi, each iteration involves a matrix multiplication, plus a vector addition. As long as the solution vectors ϕ^k and ϕ^{k+1} are kept distinct, there is a high degree of data parallelism (i.e. well suited to OpenMP or similar shared memory API).

Each iteration takes $\mathcal{O}(N)$ work, but typically $\mathcal{O}(N^2 \log N)$ iterations are required to obtain the same accuracy as the original discrete equations (in fact we can solve such a tridiagonal system in $\mathcal{O}(N)$ work in the serial case).

Gauss-Seidel

The **Gauss-Seidel** method is a variation on Jacobi that makes use of the updated solution values as they become available, i.e.

$$\phi_1^{k+1} = \phi_2^k + (h^2/2) f(x_1),$$
$$\phi_n^{k+1} = \left(\phi_{n+1}^k + \phi_{n-1}^{k+1} + h^2 f(x_n) \right) /2. \qquad n \in [2, N]$$

(where n is increasing as we proceed). Gauss-Seidel has advantages - it converges faster than Jacobi, but we have introduced some rather severe data dependencies at the same time.

Parallel Gauss-Seidel

Gauss-Seidel is made more difficult to parallelize by the data dependencies introduced by the use of the updated solution vector data on prior grid points. We can break the dependencies by introducing a "tile," $N = rP$,

$$\phi_1^{k+1} = \phi_2^k + (h^2/2) f(x_1),$$
$$\phi_{ri+1}^{k+1} = \frac{1}{2}\left(\phi_{ri+2}^k + \phi_{ri}^k + h^2 f(x_{ri+1})\right) \qquad i \in [1, P-1]$$
$$\phi_{ri+n}^{k+1} = \frac{1}{2}\left(\phi_{ri+n+1}^k + \phi_{ri+n-1}^{k+1} + h^2 f(x_{ri+n})\right) \qquad n \in [2, r] \quad i \in [0, P-1]$$

In this way we get P parallel tasks worth of work. Pseudo-code for MPI-based version of this approach follows ...

```
do iter=1,N_iterations
   if (myID == 0) phi(0) = phi(1)
   if (myiD == N_procs-1) phi(1+N/N_procs) = 0 ! upper BC
   do i=1,N/P
      u(i) = 0.5*( u(i+1)+u(i-1) ) +f(i) ! note h=1
   end do
   SENDRECV( u(0) to/from myID-1 )
   SENDRECV( u(1+N/P) to/from myID+1 )
end do
```

in practice this method is never used, though (well, ok, as a 'smoother' in multigrid, sometimes).

Reordering

So what is done, in practice, for Gauss-Seidel, is to **reorder** the equations. For example using odd/even combinations:

$$\phi_1^{k+1} = \phi_2^k + (h^2/2) f(x_1),$$

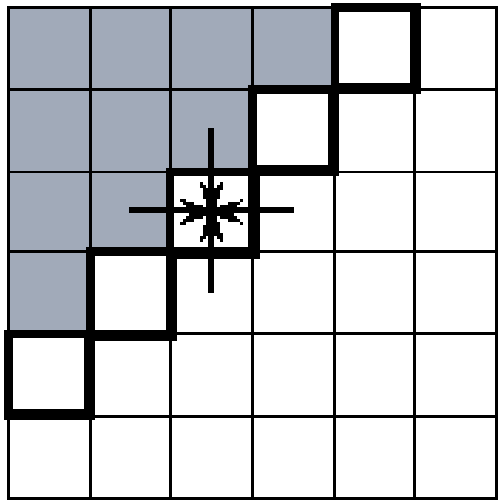
$$\phi_{2n-1}^{k+1} = \frac{1}{2}\left(\phi_{2n-2}^k + \phi_{2n}^k + h^2 f(x_{2n-1})\right) \qquad n \in [2, N/2]$$

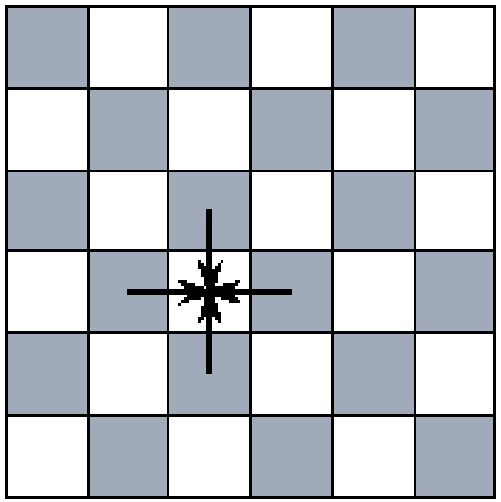
$$\phi_{2n}^{k+1} = \frac{1}{2}\left(\phi_{2n-1}^{k+1} + \phi_{2n+1}^{k+1} + h^2 f(x_{2n-1})\right) \qquad n \in [1, N/2]$$

and this looks very much (it is) like a pair of Jacobi updates. This is referred to as **coloring** the index sets, and the above is an example of the **red-black** coloring scheme (looks like a chess board). Generally you can use more than two colors, albeit with increasing complexity.

Red-Black Ordering in Gauss-Seidel

The red-black scheme is illustrated in the following examples (this is in 2D, where it looks a bit more pretty). Gray squares have been updated. Left figure is simple Gauss-Seidel, while right is a classic red-black.





Note the 5-point stencil illustrating the data dependence at a particular grid point.

Parallel Example

Updating our example, a two-color scheme is used, but in a slightly unorthodox way, in which the values `phi(k)` are one color, and remaining values `phi(1) ...phi(k-1)` the other color. This is done to facilitate the use of message-passing parallelism, and to better overlap computation with communication.

```
k = N/P
tag_left = 1
if (myID < Nprocs-1) SEND( phi(k) to myID+1, tag_left )

do iter=1,Niters
   if (myID > 0) RECV( phi(0) from myID-1, tag_left )
   if (myID == 0) phi(0)=phi(1)
   phi(1) = 0.5*( phi(0) + phi(2) ) + f(1)
   tag_right = 2*iter
   if (myID > 0) SEND( phi(1) to myID-1, tag_right )
   do i=2,k-1
      phi(i) = 0.5*( phi(i-1) + phi(i+1) ) + f(i)
   end do
   if (myID < Nprocs-1) RECV( phi(k+1) from myID+1, tag_right )
   if (myID == Nprocs-1) phi(k+1) = 0
   phi(k) = 0.5*( phi(k+1) + phi(k-1) ) + f(k)
   tag_left = 2*iter+1
   if (myID < Nprocs-1) SEND( phi(k) to myID+1, tag_left )
end do
```

Multigrid Basics

Multigrid is easy to conceptualize, but considerably more tedious in its details. In a nutshell:

- Jacobi or Gauss-Seidel forms the basic building block
- Iterations are halted while problem is transferred to a coarser mesh
- Recursively applied

Multigrid, in a Nutshell

For simplicity, consider once again, our linear elliptic problem,

$$\mathcal{L}\phi = f$$

and if we discretize using a uniform length, h,

$$\mathcal{L}_h\phi_h = f_h$$

To this point we have addressed several methods of solving for the "exact" solution to the discretized equations (directly solving the matrix equations as well as "relaxation" methods like Jacobi/Gauss-Seidel).

Now, let $\tilde{\phi}_h$ be an approximate solution to the discretized equation, and $v_h = \phi_h - \tilde{\phi}_h$. Then we can define a "defect" or "residual" as

$$d_h = \mathcal{L}_h \tilde{\phi}_h - f_h,$$

using which our error can be written as

$$\mathcal{L}_h v_h = \mathcal{L}_h \phi_h - \mathcal{L}_h \tilde{\phi}_h = -d_h$$

We have been finding v_h by approximating $\mathcal{L}_h$, $\hat{\mathcal{L}}_h$:

- Jacobi: $\hat{\mathcal{L}}_h$ = diagonal part of $\mathcal{L}_h$
- Gauss-Seidel: $\hat{\mathcal{L}}_h$ = lower triangular part of $\mathcal{L}_h$:

$$\phi_i = -\left(\sum_{j \neq i = 1}^{N} L_{ij}\phi_j - f_i \right) / L_{ii}$$

In multigrid, what we are instead going to do is *coarsen*,

$$\hat{\mathcal{L}}_h = \mathcal{L}_H$$

(and typically $H = 2h$).

Using this “two-grid” approximation,

$$\mathcal{L}_H v_H = -d_H$$

and we need to handle both the fine->coarse (sometimes called *restriction* or *injection*) transition:

$$d_H = R d_h,$$

and the coarse->fine (*prolongation* or *interpolation*)

$$\tilde{v}_h = P \tilde{v}_H$$

and our new solution is just $\tilde{\phi}_h^{new} = \tilde{\phi}_h + \tilde{v}_h$.

In summary, our **two-grid cycle** looks like:

- Compute defect on fine grid: $d_h = \mathcal{L}_h \tilde{\phi}_h - f_h$
- Restrict defect: $d_H = R d_h$
- Solve on coarse grid for correction: $\mathcal{L}_H v_H = -d_H$
- Interpolate correction to fine grid: $\tilde{v}_h = P \tilde{v}_H$
- Compute next approximation, $\tilde{\phi}_h^{new} = \tilde{\phi}_h + \tilde{v}_h$

The next level of refinement is that we **smooth** out higher-frequency errors, for which relaxation schemes (Jacobi/Gauss-Seidel) are well suited. In the context of our “two-grid” scheme, we pre-smooth on the initial fine grid, and post-smooth the corrected solution.

Putting the pieces together, the next idea is to develop a full multigrid by applying the scheme recursively down to some trivially simple (either by direct or relaxation methods) coarse grid. This is known as a **cycle**, but let us first develop a bit more mathematical formalism.

Multigrid Notation

Some definitions are in order:

- Let $K > 1$ define the **multigrid levels**
- **Level-zero Grid** is our starting coarse grid:

$$x_i = (i-1)\Delta x, \quad \Delta x = 1/N, \quad 1 \leq i \leq N+1$$

- Next level, the level-one grid, comes from subdividing level-zero:

$$x_{N+2i} = x_i \qquad i \in [1, N+1]$$

$$x_{N+2i+1} = \frac{1}{2}(x_i + x_{i+1}) \qquad i \in [1, N]$$

Multigrid Notation (cont'd)

- Level-k grid (ν_k is the number of points in level k, $\nu_0 = N + 1$, and N_k is the number of points in all previous levels $N_0 \equiv 0$, $N_1 = N + 1$):

$$x_{N_k+2i-1} = x_{N_{k-1}+i} \qquad i \in [1, \nu_{k-1}]$$

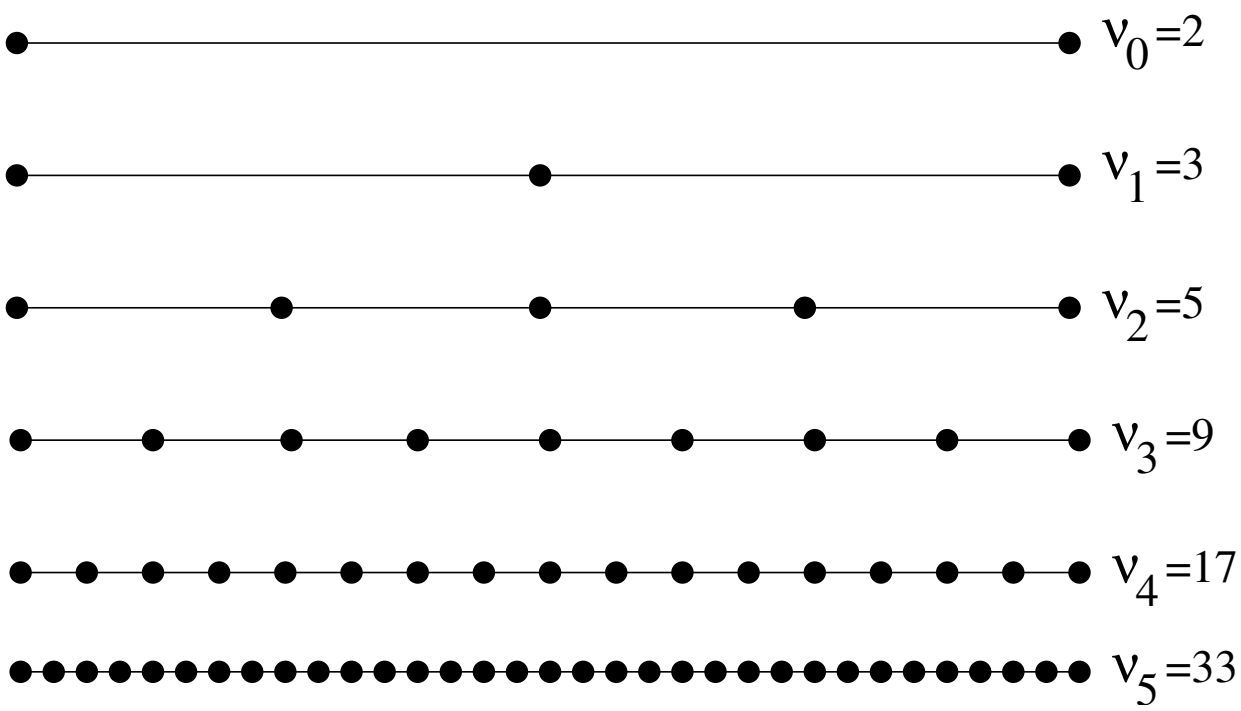
$$x_{N_k+2i} = \frac{1}{2}(x_{N_{k-1}+i} + x_{N_{k-1}+i+1}) \qquad i \in [1, \nu_{k-1} - 1]$$

and induction leads to

$$\nu_k = 2\nu_{k-1} - 1$$

$$N_{k+1} = N_k + \nu_k$$

As an example, consider the following figure:



which shows 5 levels of a one-dimensional multigrid.

Multigrid Solution

Let $\Phi_{N_{k-1}+1}, ..., \Phi_{N_k}$ denote the solution values at level $k-1$. These values define a piecewise linear function on the mesh for the $k-1$-th level,

$$\phi^{k-1}(x) = \sum_{i=1}^{\nu_k} \Phi_{N_{k-1}+i} u_i^{k-1}(x),$$

where the $u_i^{k-1}(x)$ "basis" functions are defined as in the finite element case - unity only for the i-th node at level $k-1$.

Since the grids are nested, $\phi^{k-1}(x)$ is also piecewise linear at level k, for which we can interpolate:

$$\Phi_{N_k+2i-1} = \Phi_{N_{k-1}+i} \qquad i \in [1, \nu_{k-1}]$$

$$\Phi_{N_k+2i} = \frac{1}{2}\left(\Phi_{N_{k-1}+i} + \Phi_{N_{k-1}+i+1}\right) \qquad i \in [1, \nu_{k-1} - 1]$$

If $U_{N_k} = 0$, then $U_{N_{k+1}} = 0$.

It is also necessary to **coarsen**, or transfer a solution to a coarser grid. Inverting the coarse-fine refinement is a reasonable choice:

$$\Phi_{N_{k-1}+i} = \frac{1}{2}\Phi_{N_k+2i-1} + \frac{1}{4}\left(\Phi_{n_k+2i-2} + \Phi_{N_k+2i}\right) \qquad i \in [1, \nu_{k-1}]$$

Applying Jacobi/Gauss-Seidel

The Jacobi or Gauss-Seidel method can be applied on any level of multigrid. Previously we had (for Gauss-Seidel):

$$\phi_1 \leftarrow \phi_2 + \frac{1}{2} h^2 f(x_1)$$
$$\phi_n \leftarrow \frac{1}{2}\left(\phi_{n+1} + \phi_{n-1} + h^2 f_n\right) \qquad n \in [2, N]$$

and now for multigrid:

$$\Phi_{N_k+1} \leftarrow \Phi_{N_k+2} + F_{N_k+1}$$
$$\Phi_{N_k+n} \leftarrow \frac{1}{2}\left(\Phi_{N_k+n+1} + \Phi_{N_k+n-1} + F_{N_k+n}\right) \qquad n \in [1, \nu_k]$$

where $F_{N_k+1} = h_k^2 f(x_{N_k+1})/2$, $F_{N_k+i} = h_k^2 f(x_{N_k+i})$.

Applying Jacobi/Gauss-Seidel (cont'd)

The Gauss-Seidel/Jacobi iterations are not used for the sake of convergence, but rather for **smoothing** the solution, in the sense of smoothing out error between the approximate and full solution.

For the k-th level iteration we can write the solution using the shorthand notation (note this is the converged solution):

$$A^k \Phi^k = F^k,$$

after doing, say, m_k smoothing steps we write the residual error as

$$R^k = F^k - A^k \Phi^k,$$

where

$$R_{N_k+1} \leftarrow F_{N_k+1} - \Phi_{N_k+1} + \Phi_{N_k+2}$$
$$R_{N_k+n} \leftarrow F_{N_k+n} - 2\Phi_{N_k+n} + \Phi_{N_k+n+1} + \Phi_{N_k+n-1} \qquad n \in [2, \nu_k]$$

The error $E = \tilde{\Phi} - \Phi$, where $\tilde{\Phi}$ is the exact solution of $A\tilde{\Phi} = F$, satisfies the **residual equation**,

$$A^k E^k = R^k.$$

The important result is

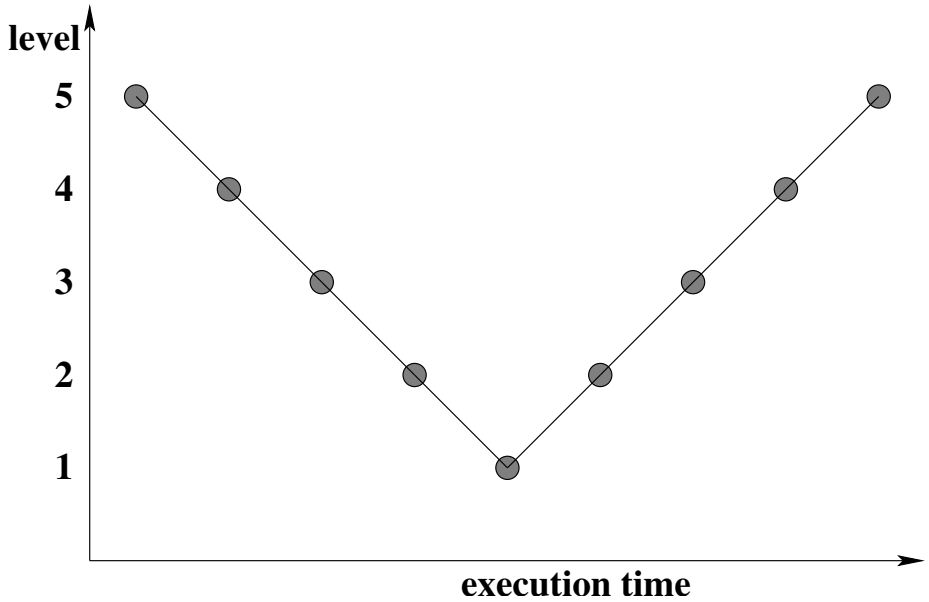
$$\tilde{\Phi}^k = \Phi^k + E^k.$$

The recursive part comes in solving the residual equation, which is done on a coarser mesh,

$$R_{N_{k-1}+i} = \frac{1}{2} R_{N_k+2i-1} + \frac{1}{4}\left(R_{N_k+2i-2} + R_{N_k+2i}\right) \quad i \in [1, \nu_{k-1}]$$

and so on back to level-zero.

Now, see why it is called the **V-cycle**? You go back up the grids (fine-to-coarse) then back again (the residuals are added back to Φ using fine-to-coarse steps), resulting in a classic "**V**" if you plot the levels.



Full Multigrid

Repeating the V-cycle on a given level will converge to $\mathcal{O}(\Delta x^2)$ in $\mathcal{O}(|\log \Delta x|)$ iterations

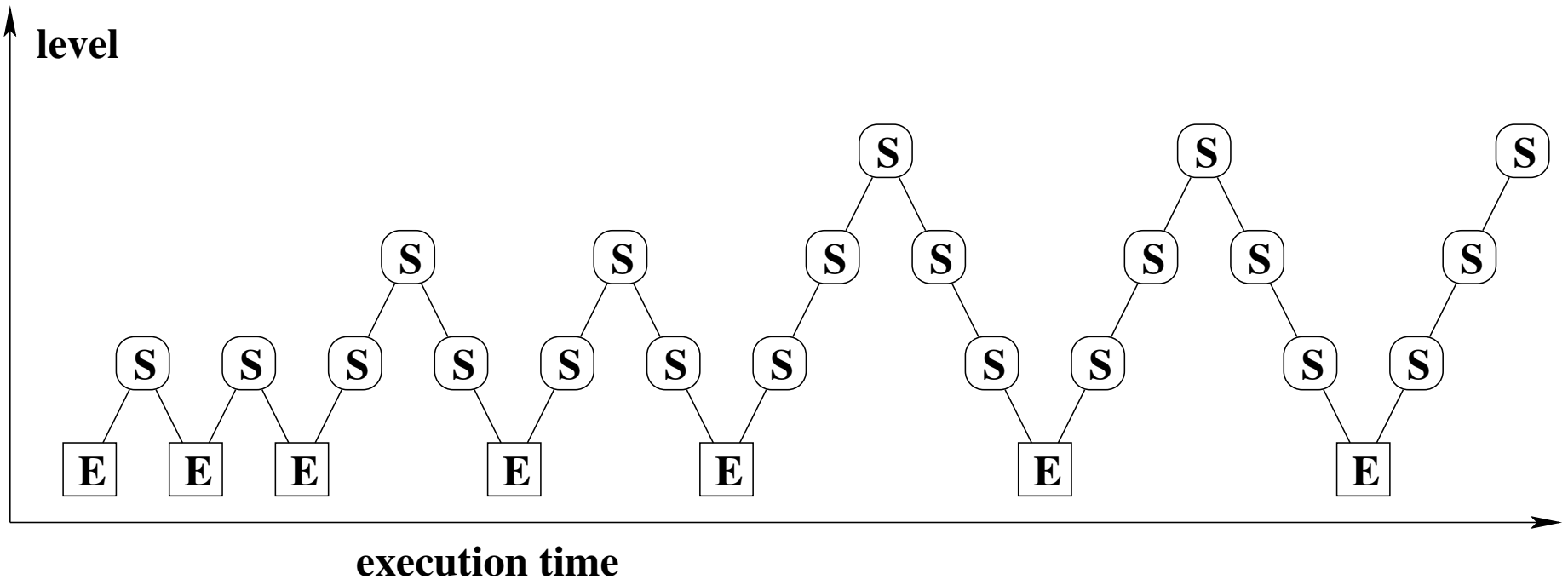
Now that we have all of the pieces, we can outline the **full multigrid V-cycle**

- Solve $A^k \Phi^k = F^k$ at level 0, $\Phi^0 = (\Phi_1, ..., \Phi_{N+1})$
- Φ^1 initially approximated by coarse-to-fine transfer of Φ^0
- Level-one proceeds (smoothing, solve level-zero residual equation, coarse-to-fine interpolation again, more smoothing) r times
- Level-two proceeds as in level-one.

By doing r repetitions at each level, we get the **Full Multigrid V-cycle**.

Full Multigrid (cont'd)

A picture is worth a few hundred words, at least:



where S denotes smoothing, and E the exact solution on the coarsest grid (illustrated is 4-level, 2-cycle, i.e. K=4, r=2).

Full Multigrid (cont'd)

Without proof, it can be shown that full multigrid converges to $\mathcal{O}(\Delta x^2)$ in $\mathcal{O}(1)$ iterations. For "large enough" r, each k-th level iteration will solve to an accuracy $\mathcal{O}(\Delta x_k^2)$. In practice, $r = 10$ is typically sufficient.

Multigrid in Parallel

The good news is that multigrid easily parallelizes in basically the same way as Jacobi/Gauss-Seidel. We have basically two parts:

1. Smoothing, which is indeed Jacobi/Gauss-Seidel, and can be parallelized in exactly the same fashion, with little modification
2. Inter-grid transfers (refinement & un-refinement is one way of looking at these interpolations) - these are relatively trivial to deal with, as only elements at the ends of the arrays need to be transferred

Multigrid Scaling

Now let's examine the scaling behavior of multigrid.

- Basic V-cycle, we estimate the time as

$$\tau_V \sim \sum_{k=0}^{K-1} \frac{c_1 N 2^{-k}}{P},$$

  where c_1 is the time spent doing basic floating-point operations during smoothing.
- Communication is involved just at exchanging data at the ends of each interval at each step in the V-cycle,

$$\tau_{comm} \sim \tau_{lat} K,$$

Multigrid Scaling (cont'd)

- Solving on the coarse grid at level K, for which there are $N2^{-K}$ unknowns. We need to collect the right-hand sides at each processor, proportional to $N2^{-K}$. Calling the constant of proportionality c_2,

$$\tau_K \sim c_2 N2^{-K}$$

Multigrid Scaling, cont'd

Putting the pieces together, the overall time is

$$\tau(P) \sim \frac{c_1 N 2}{P} + \tau_{lat} K + c_2 N 2^{-K}$$

If we equate the first and last terms, $P = 2^{K+1}/\gamma$, where $\gamma = c_2/c_1$. In that case,

$$\tau(P) \sim \frac{4 c_1 N}{P} + \tau_{lat} \log_2 (\gamma P/2),$$

and the efficiency

$$\mathcal{E}^{-1}(P) \sim 4 + \frac{\gamma P \log_2 (P\gamma/2)}{c_1 N}$$

$$\mathcal{E}(P) \sim \frac{1}{4} - \frac{\gamma P \log_2 (P\gamma/2)}{c_1 N},$$

which implies (finally!) that the full multigrid is scalable if $P \log_2 P \leq \mu N$, and

$$\mathcal{E}(P \to \infty) \sim \frac{1}{4 + \lambda\mu/c_1}$$

MG Illustrated

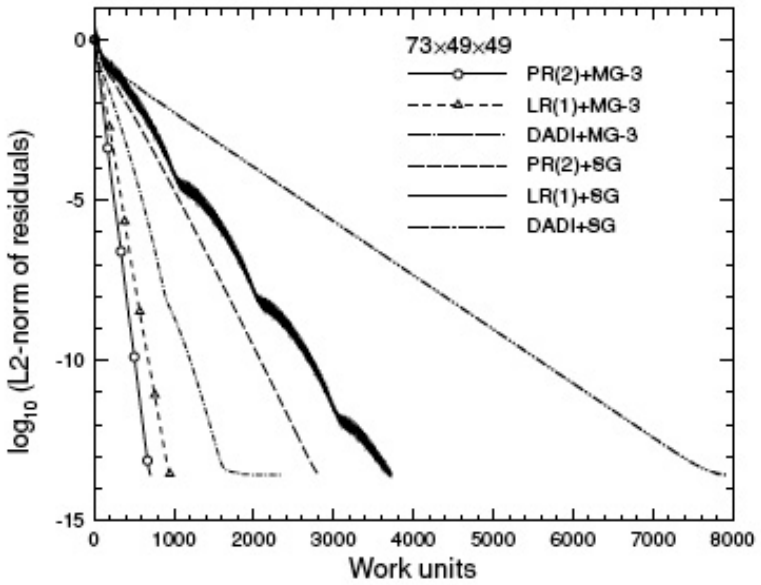


FIG. 3. The convergence histories of the three-level multigrid (MG-3) and single grid (SG) computations in conjunction with the three implicit schemes. The symbols are marked on two plotting lines every 50 iterations.

Flow through square duct with 90° bend, LR = Line Relaxation (Gauss-Seidel), PR = Point Relaxation (Gauss), DADI = (diagonalized) Alternating Direction Implicit, (S)MG = (Single)Multi-Grid. Work units normalzied to DADI-SG iterations.

L. Yuan, "Comparison of Implicit Multigrid Schemes for Three-Dimensional Incompressible Flows," J. Comp. Phys. **177**, 134-155 (2002).

Moving to Multiple Dimensions

Yes, in fact there are problems requiring more than one dimension ...

However, what we have done thus far in treating a one dimensional elliptic problem with **mesh methods** is easily transferable to 2D and 3D.

Let's look at our elliptic problem in 2D ...

2D Example

We start with

$$-\frac{\partial^2 \phi}{\partial x^2} - \frac{\partial^2 \phi}{\partial y^2} = f(x, y), \quad x, y \in [0, 1]$$

and for simplicity we take strictly Dirichlet boundary conditions, $\phi = 0$ on the boundary. Again define a regular mesh, $x_i = (i - 1)\Delta x$, $y_j = (j - 1)\Delta y$. Again for simplicity we will take N mesh points in each direction. Using finite differences or finite elements (and we could easily extend to an arbitrary mesh) results in a simple system of linear equations ...

For a regular tensor product (tensor product of one-dimensional bases) mesh,

$$\mathbf{A} = \frac{1}{h^2} \begin{pmatrix} B_N & -I_N & 0 & 0 & \dots & 0 \\ -I_N & B_N & -I_N & 0 & \dots & 0 \\ 0 & -I_N & B_N & -I_N & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & -I_N & B_N & -I_N & 0 \\ 0 & \dots & 0 & -I_N & B_N & -I_N \\ 0 & \dots & 0 & 0 & -I_N & B_N \end{pmatrix}$$

where I_N is just the $N \times N$ identity matrix, and ...

$$\mathbf{B}_N = \begin{pmatrix} 4 & -1 & 0 & 0 & \dots & 0 \\ -1 & 4 & -1 & 0 & \dots & 0 \\ 0 & -1 & 4 & -1 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & -1 & 4 & -1 & 0 \\ 0 & \dots & 0 & -1 & 4 & -1 \\ 0 & \dots & 0 & 0 & -1 & 4 \end{pmatrix}$$

and note that the numerical entries in $\mathbf{B}_N$ are for 2D, using two-point central differences.

A is an $N^2 \times N^2$ matrix (2D). The "bandwidth" is N, meaning that we will need more sophisticated sparse matrix algorithms that we have used thus far (which have largely been limited to tridiagonal systems).

Iterative techniques are easily adapted to the multidimensional case, regardless of the index ordering. Multigrid in particular can be applied directly.

Forward to the Future

Initial value problems can be a bit more difficult to handle with scalable parallel algorithms. In a similar vein to our simple example to illustrate boundary value problems, we can illustrate initial value problems using a simple physical problem - the massless pendulum, which obeys:

$$\frac{d^2u}{dt^2} = f(u),$$

with initial conditions $u(0) = a_0$, $u^{'}(0) = a_1$. In this context u is the angle that the pendulum makes with respect to the force of gravity, and $f(u) = mg\sin(u)$.

Finite Differences Redux

Applying our usual central difference scheme using a two-point formula yields:

$$u_{n+1} = 2u_n - u_{n-1} - \tau f(u_n),$$

where $\tau = \Delta t^2$. we can make this a **linear** problem by taking the small angle approximation

$$\sin(u) \simeq u + \ldots \qquad u << 1$$

Linear Form

In the linear problem (small oscillations in the case of our pendulum):

$$\frac{d^2u}{dt^2} = mgu$$

and the difference equations become a simple recursion relation:

$$u_{n+1} = (2 - \tau mg)u_n - u_{n-1}$$

with starting values

$$u_0 = a,$$
$$u_{-1} = a_0 - a_1 \Delta t,$$

which will allow for the solution for all $n \geq 0$.

Matrix/Linear Form

We can write the linear form using a very simple matrix-vector equation:

$$\mathbf{L}U = b$$

or

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \dots \\ T & 1 & 0 & 0 & 0 & \dots \\ 1 & T & 1 & 0 & 0 & \dots \\ 0 & 1 & T & 1 & 0 & \dots \\ 0 & 0 & 1 & T & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ \vdots \end{pmatrix} = \begin{pmatrix} (1-\tau mg)a_0 + a_1\Delta t \\ -a_0 \\ 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix}$$

where $T = (\tau mg - 2)$. For this set of linear equations we can bring our usual bag of tricks in terms of scalable parallel algorithms.

Nonlinearity

If we do not make the linear approximation in our simple example, we do not have such an easy (or at least well traveled) path to follow. Using a similar matrix-vector notation,

$$F(U) = \mathbf{L}^0 U + \tau mg\Theta(U) - b = 0 \tag{6}$$

where $\mathbf{L}^0$ is of similar form as before, but with the τ terms removed (i.e. diagonal elements are -2). The $\Theta(U)$ term can also be written as a matrix,

$$\Theta(U)_{ij} = \delta_{i-1,j} \sin(u_{i-1}),$$

where δ_{ij} is the Kronecker delta-function, and this term contains all of our nonlinearities.

Newton-Raphson Method

The **Newton-Raphson** method is one of the most powerful in terms of solving nonlinear problems. Basically we want to solve Eq. 6,

$$F(U) = 0$$

using a root-finding method, i.e. when $F(U)$ crosses the U axis. So for a given guess U^n we drop a tangent line to F and find where that line crosses the axis:

$$\frac{0 - F(U^n)}{U^{n+1} - U^n} = F'(U^n),$$

or

$$U^{n+1} = U^n - \frac{F(U^n)}{F'(U^n)}.$$

Convergence of Newton-Raphson

Recall that Newton-Raphson has some very nice convergence properties, provided that you start "close" to a solution:

$$|U^{n+1} - U^n| \simeq C|U^n - U^{n-1}|^2$$

namely the number of correct figures doubles with each iteration.

Multidimensional Newton-Raphson

Extending the previous relations to multiple dimensions is straightforward: we replace $F^{'}(U^n)$ by the Jacobian matrix of all partial derivatives of F with respect to vector U:

$$J_F(U)_{ij} = \frac{\partial F_i(U)}{\partial u_j},$$

and then solve:

$$J_F(U^n)V = -F(U^n),$$
$$U^{n+1} = U^n + V.$$

and repeat until convergence. In our simple pendulum case, we have

$$J_\Theta(U)_{ij} = \delta_{i-1,j}\cos(u_{i-1}).$$

Starting Newton-Raphson

So if Newton-Raphson converges quickly given a good starting approximation, how do we ensure that we have a good starting point? Generally there is not a good answer to that question, but one approach is to use a simplified approximation, $\tilde{f}$ to the nonlinear function f, and then solve

$$\frac{d^2u}{dt^2} = \tilde{f}(u),$$

along with the original initial conditions. In our pendulum example, we could have used $\tilde{f}(u) = u$ instead of $f(u) = \sin(u)$. This starting point gives us a *residual*,

$$R(u) = \frac{d^2u}{dt^2} - f(u) = \tilde{f}(u) - f(u).$$

and if we view v as a correction to u, we can write:

$$\frac{d^2v}{dt^2} = vf^{'}(u) - R(u)$$

using a simple Taylor expansion of $f(u+v) \simeq f(u) + vf^{'}(u) + \mathcal{O}(v^2)$. Or in other terms:

$$\frac{d^2v}{dt^2} = vf^{'}(u) - \left(\tilde{f}(u) - f(u)\right).$$

The size of v is zero at the start (thanks to the initial conditions), so its size can be tracked as the calculation of u proceeds, and provides a way of monitoring the calculation error.

Other Methods

It is worth mentioning other techniques for initial value problems as well:

- Multigrid in the time variable - start coarse, and then improve. See, for example:
  L. Baffico et. al., "Parallel-in-time Molecular-Dynamics Simulations," Phys. Rev. E **66**, 057701 (2002)
- Wave-form relaxation for a system of ODEs, using domain decomposition rather than parallelize over the time domain. See, for example:
  J. A. McCammon et. al., "Ordinary Differential Equations of Molecular Dynamics," Comput. Math. Appl. **28**, 319 (1994).

And Then?

Of course, this has been a mere sampling of parallel methods for partial differential equations. Hopefully it has given you a feel for at least the general principles involved.

In the next discussion of PDEs, we will focus more on available HPC solution libraries and specialized methods.