LECTURE 35: GUIDELINES FOR IMPLEMENTING TIME-STEPPING

In this lecture, we address some practical issues that arise when implementing time-stepping algorithms for the approximate solution of differential equations. We shall focus of the forward Euler method

$$y_{k+1} = y_k + \Delta t A y_k \qquad \text{(FE)}$$

and the backward Euler method

$$y_{k+1} = y_k + \Delta t A y_{k+1}$$

$$\Longrightarrow y_{k+1} = (I - \Delta t A)^{-1} y_k \qquad \text{(BE)}$$

for the differential equation $y'(t) = Ay(t)$, but the principles apply to other time-stepping methods and inhomogeneous linear problems.

① DO NOT COMPUTE POWERS OF THE ITERATION MATRIX!

For purposes of analysis, we often write

$$y_k = (I + \Delta t A)^k y_0 \qquad \text{(FE)}$$

$$y_k = ((I - \Delta t A)^{-1})^k y_0 \qquad \text{(BE)}$$

This is not a viable way to implement these algorithms! MATRIX powers are expensive to compute, and these powers will get ever more expensive as k increases!

It helps to use the language of computational complexity. If A is an $n \times n$ matrix with no special structure, the conventional matrix-matrix multiplication algorithm requires $O(n^3)$ operations - meaning that the number of arithmetic operations scales like (no worse than) n^3 as n increases. Computing $x_k = (I + \Delta t A)^k x_0$ would take $O(kn^3)$ steps at iteration k.

Contrast this with the Forward Euler implementation

```
For $K = 0, 1, 2, \dots$
    $x_{k+1} = x_k + \Delta t (A x_k)$
```

$O(n^2)$ operations for Ax_k computation

$O(n)$ operations for $\Delta t\,(Ax_k)$ computation

$O(n)$ operations for $x_k + (\Delta t (A x_k))$ computation.

This implementation only requires $O(n^2)$ operations per step.

If A has special structure, e.g., it is tridiagonal, like our M and K, then Ax_k can be computed in $O(n)$ operations, so each step of FE is very fast.

(2) DO NOT COMPUTE $(I - \Delta t A)^{-1}$ EXPLICITLY IN B.E.

One might be tempted to compute the matrix $(I - \Delta t A)^{-1}$ directly, before beginning the Backward Euler iteration. For example:

```
B = inv(I - Δt A)                O(n^3) operations

For k = 0, 1, 2, 3, ....
  X_{k+1} = B X_k                O(n^2) operations per step.
```

Alternatively, one might try to implement the inverse using MATLAB's \ ("backslash") command:

```
For k = 0, 1, 2, 3, ...
  X_{k+1} = (I - Δt A) \ X_k     O(n^3) operations per step
```

Both of the approaches have significant disadvantages.

* Even if A has special structure (e.g., tridiagonal), inv$(I-\Delta t A)$ will generally have all of its entries nonzero, so A might only have $O(n)$ nonzeros, while inv$(I-\Delta t A)$ has $O(n^2)$ nonzeros. Storing these when n is very large (e.g., $n = 10^4$ or bigger) becomes a major bottleneck.
* On the other hand, the backslash approach requires $O(n^3)$ operations <u>per step</u>, which is quite expensive.

There is a better approach. GAUSSIAN Elimination factors a matrix as:

$$P(I - \Delta t A) = LU$$

where P is a permutation of the identity ($P^TP = I$),

L is lower triangular, and U is upper triangular. Thus we can implement Backward Euler as:

```
[P, L, U] = lu(I - Δt A)            O(n^3) operations

for k = 0, 1, 2, ...
    x_{k+1} = U \ (L \ (P x_k))     O(n^2) operations
```

$$P(I-\Delta t A) = LU \Rightarrow I - \Delta t A = P^T L U$$
$$\Rightarrow (I-\Delta t A)^{-1} = U^{-1} L^{-1} (P^T)^{-1}$$
$$= U^{-1} L^{-1} P$$

This approach has a distinct advantage: If A has special structure (e.g. tridiagonal) then the P, L, and U matrices can be stored very efficiently: e.g., only $O(n)$ storage — this is much better than storing $(I-\Delta t A)^{-1}$.

For large-scale problems (e.g. $n = 10^6$ or bigger), it might be too expensive to compute this LU factorization. In this case, one can solve

$$(I - \Delta t A)\, x_{k+1} = x_k$$

for x_k using <u>iterative methods</u> that only access A by computing matrix-vector products $(I - \Delta t A)\, y$.

Depending on the situation, one might use the conjugate gradient algorithm (pcg in MATLAB), MINRES, or GMRES (gmres in MATLAB), the latter being the most general (and most expensive).

These iterative methods remain a very active field of research.

Solving $Ma'(t) = -Ka(t)$ with FE and BE

We close with a brief discussion of how to solve the equation $Ma'(t) = -Ka(t)$ that arose in the solution of the heat equation. We could transform this into $a'(t) = -M^{-1}Ka(t)$ and use $A = -M^{-1}K$ in the discussion above. Consider this alternative. Forward Euler gives

$$\begin{aligned} a_{k+1} &= (I - \Delta t M^{-1}K)a_k \\ &= (M^{-1}M - \Delta t M^{-1}K)a_k \\ &= M^{-1}(M - \Delta t K)a_k \end{aligned}$$

requiring application of M^{-1} at each step.

In contrast, Backward Euler gives

$$a_{k+1} = (I + \Delta t M^{-1} K)^{-1} a_k$$
$$= (M^{-1} M + \Delta t M^{-1} K)^{-1} a_k$$
$$= (M^{-1}(M + \Delta t K))^{-1} a_k$$
$$= (M + \Delta t K)^{-1} (M^{-1})^{-1} a_k$$
$$= (M + \Delta t K)^{-1} M a_k$$

Which requires application of $(M+\Delta t K)^{-1}$ at each step. Since M and $M+\Delta t K$ are both symmetric and positive definite, we can replace the LU factorization described earlier with a "Cholesky factorization". For forward Euler, this gives

$$M = LL^T \qquad (L \text{ lower triangular})$$

So FE becomes

```
$L = chol(M)$
For $k = 0, 1, 2, \dots$
    $a_{k+1} = a_k - \Delta t (L' \backslash (L \backslash (M a_k)))$
```

$(a_{k+1} = a_k - \Delta t M^{-1} K a_k)$

For Backward Euler we take

$$M + \Delta t K = LL^T$$

So BE becomes

```
$L = chol(M + \Delta t K)$
For $k = 0, 1, 2, \dots$
    $a_{k+1} = L \backslash (L \backslash (M a_k));$
```

Very similar amount of work, but B.E. has no timestep restriction/ CFL condition for stability!