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 on the forward Euler method

\[ y_{k+1} = y_k + \Delta t \cdot A \cdot y_k \quad \text{(FE)} \]

and the backward Euler method

\[ y_{k+1} = y_k + \Delta t \cdot A \cdot y_{k+1} \]

\[ \Rightarrow y_{k+1} = \left( I - \Delta t A \right)^{-1} y_k \quad \text{(BE)} \]

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

\( \text{1. Do not compute powers of the iteration matrix!} \)

For purposes of analysis, we often write

\[ y_k = (I + \Delta t A)^k y_0 \quad \text{(FE)} \]

\[ y_k = \left( I - \Delta t A \right)^{-1} y_0 \quad \text{(BE)} \]

This is not a viable way to implement these algorithms! Matrix powers are expensive to compute, and those 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 \( n \) increased. Computing \( X_k = (I + \Delta tA)^k X_0 \) would take \( O(kn^3) \) steps at iteration \( k \).

Contrast this with the Forward Euler implementation:

\[
\text{For } k = 0, 1, 2, \ldots \quad \begin{align*}
X_{k+1} &= X_k + \Delta t(Ax_k) \\
\end{align*}
\]

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

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

\( \textbf{2) Do not compute } (I - \Delta tA)^{-1} \text{ explicitly in FE.} \)

One might be tempted to compute the matrix \( (I - \Delta tA)^{-1} \) directly, before beginning the Backward Euler iteration. For example:
\[ B = \text{inv} (I - \Delta t A) \quad O(n^3) \text{ operations} \]

For \( k = 0, 1, 2, 3, \ldots \)
\[ X_{k+1} = B \times X_k \quad O(n^2) \text{ operations per step.} \]

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

For \( k = 0, 1, 2, 3, \ldots \)
\[ X_{k+1} = (I - \Delta t A) \backslash X_k \quad O(n^3) \text{ operations per step.} \]

Both of the approaches have significant disadvantages.

* Even if \( A \) has special structure (e.g., tridiagonal), \( \text{inv}(I - \Delta t A) \) will generally have all of its entries nonzero, so \( A \) might only have \( O(n) \) nonzeros, while \( \text{inv}(I - \Delta t A) \) has \( O(n^2) \) nonzeros. Storing these when \( n \) is very large (e.g., \( n = 10^7 \) or bigger) becomes a major bottleneck.

* On the other hand, the backslash approach requires \( O(n^3) \) operations per step, 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:

$$[P, L, U] = LU(I - \Delta A)$$  
$$O(n^3)$$ operations

For \( k = 0, 1, 2, \ldots \)

$$L X_{k+1} = U \backslash (L \backslash (P X_k))$$  
$$O(n^2)$$ operation

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

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 \((I - \Delta A)^{-1}\).

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

$$(I - \Delta A) X_{k+1} = X_k$$

for \( X_k \) using iterative methods that only
access \( A \) by computing matrix-vector products
\((I - \Delta A) y\).
Depending on the situation, one might use the conjugate gradient algorithm (pcg in MATLAB), MINRES, or GMRES (guaranteed in MATLAB), the latter being the most general (and most expensive).

Iterative methods remain a very active field of research.

Solving $M \dot{a}(t) = -K a(t)$ with FE and BE

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

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

requiring application of $M^T$ at each step.

In contrast, Backward Euler gives
\[ a_{kn} = (I + \Delta t M^{-1} K)^{-1} a_k \]
\[ = (M^{-1} M + \Delta t M^{-1} K)^{-1} a_k \]
\[ = (M^{-1} (M + \Delta t K))^{-1} a_k \]
\[ = (M + \Delta t K)^{-1} (M^{-1})^{-1} a_k \]
\[ = (M + \Delta t K)^{-1} M a_k \]

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

\[ M = LL^T \quad (L\text{ lower triangular}) \]

So FE becomes:

\[ L = \text{chol}(M) \]

For \(k = 0, 1, 2, \ldots\)

\[ L a_{k+1} = a_k - \Delta t (L \backslash (L \backslash (M a_k))) \]

For Backward Euler we take

\[ M + \Delta t K = LL^T \]

So BE becomes:

\[ L = \text{chol}(M + \Delta t K) \]

For \(k = 0, 1, 2, \ldots\)

\[ L a_{k+1} = L \backslash (L \backslash (M a_k)) \]

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