Linear Two Point Boundary Value Problems

- We are going to consider the second order boundary value problem (BVP)

\[-u''(x) = f(x), \quad x_L < x < x_R\]

\[u(x_L) = u_L \quad u(x_R) = u_R\]

- Notice that this problem is inherently different from our IVP \(y'(t) = f(t, y)\) which told us the value of \(y\) initially and its slope and we were to determine \(y\) for future times.

- Here we know the value of \(u\) at each endpoint of our domain \([x_L, x_R]\), its second derivative and we want to determine \(u\) in the interior of the domain, i.e., the interior of the interval.

- There is no time involved so we call this a time independent problem.

- The values of \(u\) specified at \(x = x_L\) and \(x = x_R\) are called boundary conditions.
• We can have other boundary conditions such as
  \[ u'(x_L) = g \]

  or
  \[ \alpha u(x_L) + \beta u'(x_L) = g \]

• Note, however, if we have the problem
  \[ -u''(x) = f(x), \quad a < x < b \]

  \[ u'(x_L) = u_L \quad u'(x_R) = u_R \]

then there is NOT a unique solution to this problem. Because if we can find
a \( u(x) \) which satisfies the ODE and the boundary conditions then so does,
for example, \( u(x) + 5 \).

• Our strategy in solving this BVP is to discretize our domain, that is, break
the interval \([x_L, x_R]\) in \( n + 1 \) subintervals of length \( \Delta x \) using
  \[ x_0 = x_L, \quad x_1 = x_L + \Delta x, \quad x_i = x_L + i\Delta x, \quad x_{n+1} = x_R \]

and to find an approximation \( U_i \) to \( u(x_i) \).
• Once again, $U_i$’s are discrete and only “live” at the nodes $x_i$. This is in contrast to the exact solution $y(x)$ which is defined for all $x$ in $[x_L, x_R]$.

• At which points $x_i$ do we have an unknown?

• Clearly we can set $U_0 = u_L$, $U_{n+1} = u_R$. So we want to obtain $U_i$, $i = 1, 2, \ldots, n$.

• As before, we can replace the derivative with a difference quotient to get an equation involving only function values.

• Recall that the difference quotient we derived from Taylor series for $u''(x)$ was the second centered difference quotient

$$u''(x) = \frac{u(x + \Delta x) - 2u(x) + u(x - \Delta x)}{\Delta x^2} + O(\Delta x^2)$$
• So if we let $U_i \approx u(x_i)$ and set $x = x_i$ then this difference quotient becomes

$$\frac{U_{i+1} - 2U_i + U_{i-1}}{\Delta x^2}$$

where $U_{i\pm 1} \approx u(x_i \pm \Delta x)$.

• This is our approximation to $u''(x_i)$ so at the point $x_i$ the difference equation is just

$$\frac{-U_{i+1} + 2U_i - U_{i-1}}{\Delta x^2} = f(x_i)$$

Recall that the original DE was $-u'' = f(x)$ so that we have multiplied our quotient by a negative one.

• This equation can’t be solved sequentially for $U_i$ because we don’t know $U_{i+1}$.

• Note that this is completely different from the IVP where given $y(0)$ we solved for our approximation at $\Delta t$, then at $2\Delta t$, etc.

• This is the important difference in a BVP. The unknowns are all coupled and so we must solve for them at one time.

• To do this we write this difference equation at each of the nodes where we
have an unknown; for us this is just $x_1, x_2, \ldots, x_n$.

• Doing this we have (where we have multiplied through by $\Delta x^2$)
\[-U_2 + 2U_1 - U_0 = \Delta x^2 f(x_1)\]

\[-U_3 + 2U_2 - U_1 = \Delta x^2 f(x_2)\]

\[-U_4 + 2U_3 - U_2 = \Delta x^2 f(x_3)\]

\[\vdots\]

\[-U_n + 2U_{n-1} - U_{n-2} = \Delta x^2 f(x_{n-1})\]

\[-U_{n+1} + 2U_n - U_{n-1} = \Delta x^2 f(x_n)\]

• Note that in the first equation
\[-U_2 + 2U_1 - U_0 = \Delta x^2 f(x_1)\]

the value $U_0$ is known (since we set it to $u_L$) and so the equation can be written as
\[-U_2 + 2U_1 = \Delta x^2 f(x_1) + U_0\]
• Also the last equation becomes

$$2U_n - U_{n-1} = \Delta x^2 f(x_n) + U_{n+1}$$

• How do we solve this? The first thing to notice is that we have \( n \) unknowns \( U_i, i = 1, \ldots, n \) and \( n \) equations.

• Consequently we have a linear system of algebraic equations to solve which can be written in the form

$$A\vec{U} = \vec{F}$$

where

$$\vec{U} = \left( U_1, U_2, \ldots, U_{n-1}, U_n \right)^T,$$

$$\vec{F} = \Delta x^2 \left( f(x_1) + u_L, f(x_2), \ldots, f(x_{n-1}), f(x_n) + u_R \right)^T$$
and $A$ is the $n \times n$ matrix

$$
A = \begin{pmatrix}
2 & -1 & 0 & \cdots & 0 \\
-1 & 2 & -1 & 0 & \cdots & 0 \\
0 & -1 & 2 & -1 & 0 & \cdots & 0 \\
0 & 0 & -1 & 2 & -1 & 0 & \cdots & 0 \\
& & & & & & & \\
& & & & & & & \\
0 & -1 & 2 & -1 & 0 & \cdots & 0 \\
0 & -1 & 2 & -1 \\
0 & -1 & 2
\end{pmatrix}
$$

- You have probably seen several methods for solving a linear system $Ax = b$. For example, Gaussian elimination or Cramer’s rule.
- In addition, you might have written $x = A^{-1}b$ where $A^{-1}$ is the inverse of $A$, i.e., $AA^{-1}$ is the $n \times n$ identity matrix.
- We NEVER solve a linear system numerically by finding the inverse and multiplying it by the right hand side; this is the most costly approach.
- There are standard methods for solving a linear system. However, our matrix has a special structure which we want to exploit.
• First of all, our matrix is symmetric since $A_{ij} = A_{ji}$ where $A_{ij}$ represents the entry in our matrix in the $i$th row and the $j$th column.

• The other property of our matrix is that it has a lot of 0’s which are in a definite pattern. Specifically, all entries that are not in the main diagonal, the first superdiagonal and the first subdiagonal are zeros.

• We call such a matrix a symmetric, tridiagonal matrix.

• In the next slides we will briefly look at solving a linear system of equations.
Linear Systems of Equations

• When we discretize BVPs (ODEs and PDEs) we are often lead to linear systems of equations which we will generically write as \( A\vec{x} = \vec{b} \).

• Solving linear systems is computational expensive so we need to exploit any structure that the matrix has.

• There are both direct and iterative methods for solving linear systems.

• The simplest type of linear system is a diagonal system, i.e., when \( A \) is diagonal. In this case we can solve the system immediately. For example, if we have

\[
\begin{pmatrix}
2 & 0 & 0 & 0 \\
0 & \frac{1}{2} & 0 & 0 \\
0 & 0 & 6 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{pmatrix}
= \begin{pmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4
\end{pmatrix}
\]

then clearly we have to perform 4 divisions (in general \( n \) divisions if \( A \) is
\( n \times n \) to get

\[
x_1 = b_1/2 \quad x_2 = 2b_2, \quad x_3 = b_3/6, \quad x_4 = b_4
\]

- We say that to solve an \( n \times n \) diagonal system requires \( \mathcal{O}(n) \) operations.

- Another inexpensive system to solve is a triangular system (either upper or lower triangular).

- In a lower triangular matrix \( A_{ij} = 0 \) for \( j > i \); i.e., it has zero entries above the main diagonal. For example, the following matrix is a \( 4 \times 4 \) lower triangular system

\[
\begin{pmatrix}
2 & 0 & 0 & 0 \\
3 & 1/2 & 0 & 0 \\
1 & 2 & 6 & 0 \\
-1 & 0 & 3 & 1 \\
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
\end{pmatrix}
=
\begin{pmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4 \\
\end{pmatrix}
\]

- Here we first solve for \( x_1 \) from \( 2x_1 = b_1 \), then use this to solve for \( x_2 \) since \( 3x_1 + x_2/2 = b_2 \), etc.

- In an upper triangular matrix \( A_{ij} = 0 \) for \( i > j \); i.e., it has zero entries below the main diagonal. For example, the following matrix is a \( 4 \times 4 \) upper
triangular system

\[
\begin{pmatrix}
2 & 0 & 3 & 1 \\
0 & \frac{1}{2} & 2 & 0 \\
0 & 0 & 6 & 2 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{pmatrix}
=
\begin{pmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4
\end{pmatrix}
\]

• Recall that when you did Gauss elimination you converted the original matrix into an upper triangular matrix. This was because an upper triangular is easy to solve.

• In our example, we simply solve for \(x_4\) since \(1 \cdot x_4 = b_4\) then use this to solve for \(x_3\) since \(6x_3 + 2x_4 = b_3\) and proceed to solve for the \(x_i\) in reverse order.

• Solving an upper or lower triangular requires \(O(n^2)\) operations.

• This is the same order of work that it takes to multiply an \(n \times n\) matrix times a \(n\)-vector.
Solving a Symmetric Tridiagonal System

- A tridiagonal system can be solved very efficiently.
- Note that unlike the previous systems we have considered, we can’t solve for any $x_i$ explicitly.
- For example, consider the system

$$
\begin{pmatrix}
2 & -1 & 0 & 0 \\
-1 & 2 & -1 & 0 \\
0 & -1 & 2 & -1 \\
0 & 0 & -1 & 2 \\
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
\end{pmatrix}
=
\begin{pmatrix}
f_1 \\
f_2 \\
f_3 \\
f_4 \\
\end{pmatrix}
$$

We can’t start at the top because $x_1$ and $x_2$ are coupled ($2x_1 - x_2 = f_1$) and similarly we can’t start at $x_4$ since it is coupled with $x_3$ ($-x_3 + 2x_4 = f_4$).

- A common practice in solving linear systems is to factor the original matrix $A$ into the product of a unit lower triangular and an upper triangular matrix.
• We write

\[ A = LU \implies LUX = f \]

where \( L \) is lower triangular, \( L_{ii} = 1 \) for all \( i \) and \( U \) is upper triangular.

• Can we always perform this factorization if \( A \) is nonsingular? The answer is no, which the following example illustrates

\[
\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \neq \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{pmatrix}
\]

This can’t be done even though \( A \) is nonsingular since \( 1 \cdot u_{11} = 0 \) implies \( u_{11} = 0 \) but \( l_{21}u_{11} = 1 \) which can’t be satisfied.

• What is true, is that we can interchange the rows of \( A \) (i.e., just interchange equations) so that this is true.

• How does this help us in solving a linear system?

• Recall that solving an upper or lower triangular system is “easy”.

• Keeping this in mind we solve the lower triangular system

\[ L\vec{y} = \vec{f} \quad (\text{where } \vec{y} = U\vec{x}) \]
and then solve

\[ U \bar{x} = \bar{y} \]

which is an upper triangular system.

• Each of these solves takes \( O(n^2) \) operations.

• However, if \( A \) has no special structure, i.e., it is a dense \( n \times n \) matrix, then the factorization \( A = LU \) takes \( O(n^3) \) operations and solving the entire system requires \( O(n^3) + 2O(n^2) = O(n^3) \). That is why we say solving a linear system without any special structure requires \( O(n^3) \) operations.

• Notice that this says that the work in solving a linear system increases as the cube of the number of equations. For example

\[
100^3 = 10^6 \quad (1000)^3 = 10^9 \quad (100,000)^3 = 10^{15}
\]

• Notice that even if our original matrix is symmetric it still requires \( O(n^3) \) operations (although it is half as much as for a nonsymmetric matrix).

• For a symmetric, positive definite matrix (\( \bar{x}^T A \bar{x} > 0 \) for all \( \bar{x} \neq \bar{0} \)), we can write \( A = L^T L \) where \( L \) is no longer unit lower triangular but simply triangular.
• One can show that our matrix is positive definite; clearly it is symmetric.

• We can solve our tridiagonal system very efficiently.

• Our first step is to obtain the equations to factor the matrix into \( A = LL^T \).

• In general, we have the following symmetric tridiagonal matrix which we factor as follows:

\[
\begin{pmatrix}
a_1 & b_1 & 0 & 0 & 0 \\
b_1 & a_2 & b_2 & 0 & 0 \\
0 & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & b_{n-2} & a_{n-1} & b_{n-1} \\
0 & 0 & 0 & b_{n-1} & a_n
\end{pmatrix} = \begin{pmatrix}
\alpha_1 & 0 & 0 & 0 & 0 \\
\beta_1 & \alpha_2 & 0 & 0 & 0 \\
0 & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & \beta_{n-2} & a_{n-1} & 0 \\
0 & 0 & 0 & \beta_{n-1} & \alpha_n
\end{pmatrix} \begin{pmatrix}
\alpha_1 & \beta_1 & 0 & 0 & 0 \\
0 & \alpha_2 & \beta_2 & 0 & 0 \\
0 & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & \alpha_{n-1} & \beta_{n-1} & 0 \\
0 & 0 & 0 & 0 & \alpha_n
\end{pmatrix}
\]

• Note that we have made the assumption that \( L \) is bidiagonal (main diagonal plus one subdiagonal) which one can verify by direct multiplication.

• It is straightforward to derive the equations for the \( \alpha_i, i = 1, \ldots, n \) and the \( \beta_i, i = 1, \ldots, n - 1 \).

• To do this, we simply multiply the two matrices on the right and set the
value equal to the corresponding entry in the matrix on the left. We have

\[ \alpha_1^2 = a_1 \implies \alpha_1 = \sqrt{a_1} \]

Note that this is only valid if \( a_1 > 0 \) which is the case if \( A \) is positive definite.

- Once we have \( \alpha_1 \) then we can determine \( \beta_1 \) by looking at the (1,2) entry of \( A \). We have

\[ \alpha_1 \beta_1 = b_1 \implies \beta_1 = b_1 / \alpha_1 \]

- We can now determine \( \alpha_2 \) by looking at the (2,2) entry of \( A \). We have

\[ \beta_1^2 + \alpha_2^2 = a_2 \implies \alpha_2 = \sqrt{a_2 - \beta_1^2} \]

- Then \( \beta_2 \) is found by looking at the (2,3) entry of \( A \)

\[ \alpha_2 \beta_2 = b_2 \implies \beta_2 = b_2 / \alpha_2 \]

- Continuing in this fashion we get the following equations for \( \alpha_i, \beta_i \) in terms of the entries of \( A \)

\[ \alpha_1 = \sqrt{a_1} \]
For $i = 2, n$

$$\beta_{i-1} = \frac{b_{i-1}}{\alpha_{i-1}}$$

$$\alpha_i = \sqrt{a_i - \beta_{i-1}^2}$$

Note that we can not determine all the $\alpha_i$ and then all the $\beta_i$.

- So to get the components of $L$ we simply program these equations.
- Once we have the entries of $L$ we perform our forward solve $L\vec{y} = \vec{b}$ and then our back solve $L^T\vec{x} = \vec{y}$.
- To perform the forward solve we solve the lower triangular system

\[
\begin{pmatrix}
\alpha_1 & 0 & 0 & 0 & 0 \\
\beta_1 & \alpha_2 & 0 & 0 & 0 \\
0 & \ldots & \ldots & \ldots & \\
0 & 0 & \beta_{n-2} & \alpha_{n-1} & 0 \\
0 & 0 & 0 & \beta_{n-1} & \alpha_n \\
\end{pmatrix}
\begin{pmatrix}
y_1 \\
y_2 \\
\vdots \\
y_{n-1} \\
y_n \\
\end{pmatrix}
= 
\begin{pmatrix}
f_1 \\
f_2 \\
\vdots \\
f_{n-1} \\
f_n \\
\end{pmatrix}
\]
• Again this is easy to do by multiplying the matrix by the vector \( \vec{y} \) and equating to the appropriate entry on the right. We have

\[
\alpha y_1 = f_1 \implies y_1 = f_1/\alpha_1
\]

\[
\beta_1 y_1 + \alpha_2 y_2 = f_2 \implies y_2 = (f_2 - \beta_1 y_1)/\alpha_2
\]

\[
\beta_2 y_2 + \alpha_3 y_3 = f_3 \implies y_3 = (f_3 - \beta_2 y_2)/\alpha_3
\]

• So our algorithm for the forward solve if \( L \) is a bidiagonal matrix is just

\[
y_1 = \frac{f_1}{\alpha_1}
\]

For \( i = 2, \ldots, n \)

\[
y_i = \frac{f_i - \beta_{i-1} y_{i-1}}{\alpha_i}
\]

• Lastly, we perform the back solve using the upper triangular (bidiagonal) matrix \( L^T \). We have
\[
\begin{pmatrix}
\alpha_1 & \beta_1 & 0 & 0 & 0 \\
0 & \alpha_2 & \beta_2 & 0 & 0 \\
0 & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & \alpha_{n-1} & \beta_{n-1} \\
0 & 0 & 0 & 0 & \alpha_n
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_{n-1} \\
x_n
\end{pmatrix}
= 
\begin{pmatrix}
y_1 \\
y_2 \\
\vdots \\
y_{n-1} \\
y_n
\end{pmatrix}
\]

- Starting at \( x_n \) we have

\[
\alpha_n x_n = y_n \implies x_n = \frac{y_n}{\alpha_n}
\]

\[
\alpha_{n-1} x_{n-1} + \beta_{n-1} x_n = y_{n-1} \implies x_{n-1} = \frac{(y_{n-1} - \beta_{n-1} x_n)}{\alpha_{n-1}}
\]

- Continuing in this way we have the algorithm

\[
x_n = \frac{y_n}{\alpha_n}
\]

For \( i = n-1, n-2, \ldots, 1 \)

\[
x_i = \frac{y_i - \beta_i x_{i+1}}{\alpha_i}
\]
• How much work is it to solve this symmetric tridiagonal system? To answer this we count operations.

• **Back solve**
  additions/subtractions: \( n - 1 \)
  multiplications/divisions: \( n + n - 1 = 2n - 1 \)

• **Forward solve**
  additions/subtractions: \( n - 1 \)
  multiplications/divisions: \( n + n - 1 = 2n - 1 \)

• **Factorization**
  additions/subtractions: \( n - 1 \)
  multiplications/divisions: \( 2(n - 1) = 2n - 2 \)
  square root: \( n \)

• So we have a total of \( O(n) \) operations to perform which is a lot less than using a full solver requiring \( O(n^3) \) operations.
Implementing a Tridiagonal Solver

- One advantage to factoring a matrix $A = LU$ (or in the case of $A$ symmetric, positive definite $A = LL^T$) is that if we have multiple right hand sides to solve, then we do the factorization once (which recall is $O(n^3)$ for a general matrix) and then for each right hand side we perform a forward and backward solve.

- If we want to solve $q$ systems with the same coefficient matrix, then this allows us to solve them in $O(n^3) + 2qO(n^2)$ as opposed to $qO(n^3)$ operations.

- It is because of this fact that usually we have separate routines to factor a matrix and to perform forward or backward solves.

- The next question to consider is storage.

- First how much storage do we need for the original matrix $A$? All we really need is its main diagonal and either the upper or lower bidiagonal. Thus we can either store $A$ as a $2 \times n$ matrix or in two 1-d arrays of length $n$. 
• Clearly we need a 1-d array for the right hand side.

• Do we need to have separate storage for the factors of $A$ or can we overwrite?

• Typically the storage for a matrix takes your largest chunk of memory so we usually overwrite. This is possible if you recall our equations

\[ \alpha_1 = \sqrt{a_1} \]

For $i = 2, n$

\[ \beta_{i-1} = \frac{b_{i-1}}{\alpha_{i-1}} \]

\[ \alpha_i = \sqrt{a_i - \beta_{i-1}^2} \]

Notice that $a_i$, $b_i$ are not needed after the formation of $\alpha_i$, $\beta_i$ so we can overwrite. For example we have

\[ b_{i-1} = \frac{b_{i-1}}{a_{i-1}}, \quad a_i = \sqrt{a_i - b_{i-1}^2} \]

• Similarly when we do the forward solve we overwrite the right hand side with $\vec{y}$ and when we do the back solve we overwrite the right hand side (now $\vec{y}$ in
our terminology) with the solution. So on input to the forward solve we have the right hand side and on return we have $\vec{y}$. Similarly on input to the back solve we have what is called the right hand side but actually now containing $\vec{y}$ and on output it contains the solution.

- This is what is done typically in all library routines for (direct) linear solvers.
- We can now look at our code for the factorization. Note that I have put in a check to make sure the matrix is positive definite; otherwise return an error.

```fortran
subroutine factor_sym_tridiagonal ( n, a, b )

integer, intent(in) :: n
real(prec), dimension(n), intent(inout) :: a ! main diagonal
real(prec), dimension(n), intent(inout) :: b ! upper diagonal
integer :: i
real(prec) :: term

if ( a(1) <= zero ) then
```

print *, "matrix not positive definite"
else
  a(1) = sqrt ( a(1) )
end if
do i = 2, n
  b(i-1) = b(i-1) / a(i-1)
term = a(i) - b(i-1)*b(i-1)
if ( term <= zero ) then
  print *, "matrix not positive definite"
  stop
else
  a(i) = sqrt ( term )
end if
end do
end subroutine factor_sym_tridiagonal

Forward solve

subroutine forward_solve_bidiagonal ( n, a, b, rhs )
integer, intent(in) :: n
real(prec), dimension(n), intent(in) :: a ! main diagonal
real(prec), dimension(n), intent(in) :: b ! upper diagonal
real(prec), dimension(n), intent(inout) :: rhs ! rhs
integer :: i
real(prec) :: term
rhs(1) = rhs(1) / a(1)

do i = 2, n
    rhs(i) = ( rhs(i) - b(i-1)*rhs(i-1) ) / a(i)
end do

end subroutine forward_solve_bidiagonal

Back solve

subroutine back_solve_bidiagonal ( n, a, b, rhs )
    integer, intent(in) ::  n
    real(prec), dimension(n), intent(in ) ::  a ! main diagonal
    real(prec), dimension(n), intent(in ) ::  b ! upper diagonal
    real(prec), dimension(n), intent(inout) ::  rhs ! rhs
integer ::  i

real(prec) ::  term

rhs ( n ) = rhs(n) / a(n)

do i = n-1, 1, -1
    rhs(i) = ( rhs(i) - b(i )*rhs(i+1) ) / a(i)
end do

end subroutine back_solve_bidiagonal
Solving our Two-Point BVP

- Now that we have routines to solve our linear system resulting from discretizing our ODE, we can turn to the solution.
- So basically all we need to do is to form the right hand side vector and the coefficient matrix.
- In our case the matrix is trivial since we just set
  \[ a(1:n) = \text{two}; \quad b(1:n-1) = -\text{one} \]
- The right hand side vector is simply our given \( f \) evaluated at the appropriate point \( x_i \) times \( \Delta x^2 \). Remember, however, that you need to adjust the first and last entries of the right hand side due to the boundary values.
- So the first thing you could do is set up the grid, i.e., the \( x_i \). As before you want to input the points defining the interval and say the number of subintervals of length \( \Delta x \) you divide it into. Then you can simply determine the \( x_i \) as we have done before.
- Structure of code is to set up grid, set coefficient matrix, form the right hand side, factor the matrix, perform a forward solve, perform a back solve.

- The only thing you need to be careful with is the indexing of $x_i$ and the unknown numbers. The notation we have used in the notes is $n + 2$ points, $n + 1$ intervals and $n$ unknowns using

$$x(i), i = 0, n + 1, \quad \text{rhs}(i), i = 1, n$$
So far we have considered the second order boundary value problem (BVP)

\[-u''(x) = f(x), \quad x_L < x < x_R \]

\[u(x_L) = u_L, \quad u(x_R) = u_R\]

However, this is not the most general form of a second order linear BVP.

The most general form of the second order linear ODE is

\[-u''(x) + p(x)u'(x) + q(x)u(x) = f(x)\]

How does this change our discretization?

Recall that we wrote a difference equation at each point \(x_i, i = 1, \ldots, n\); in particular we substituted the second centered difference approximation for

\[-u''(x_i)\]

\[
\frac{-U_{i+1} + 2U_i - U_{i-1}}{\Delta x^2}
\]
• Now we must also replace $u'(x)$ with a difference quotient.
• We have seen three difference quotients for $u'(x)$: forward Euler, backward Euler, and centered difference.
• Which one should we use and why?
• If we use a centered difference we replace $u'(x_i)$ by
  \[ \frac{U_{i+1} - U_{i-1}}{2\Delta x} \]
• What do we do with $u(x)$? We simply evaluate at $x_i$ since there is no derivative; so this term becomes $U_i$.
• Then our difference equation at $x_i$ is
  \[ \frac{-U_{i+1} + 2U_i - U_{i-1}}{\Delta x^2} + p(x_i) \frac{U_{i+1} - U_{i-1}}{2\Delta x} + q(x_i)U_i = f(x_i) \]
  for $i = 1, n$.
• Multiplying through by $\Delta x^2$ we get
  \[ \left(-U_{i+1} + 2U_i - U_{i-1}\right) + p(x_i)\frac{\Delta x}{2}\left(U_{i+1} - U_{i-1}\right) + q(x_i)\Delta x^2U_i = \Delta x^2 f(x_i) \].
Now we combine terms to get

\[
\left(-1+p(x_i)\frac{\Delta x}{2}\right)U_{i+1} + \left(2+q(x_i)\Delta x^2\right)U_i + \left(-1-p(x_i)\frac{\Delta x}{2}\right)U_{i-1} = \Delta x^2 f(x_i)
\]

- Writing this equation at each \(x_i, i = 1, n\) we have

\[
\left(-1+p(x_1)\frac{\Delta x}{2}\right)U_2 + \left(2+q(x_1)\Delta x^2\right)U_1 + \left(-1-p(x_1)\frac{\Delta x}{2}\right)U_0 = \Delta x^2 f(x_1)
\]

\[
\left(-1+p(x_2)\frac{\Delta x}{2}\right)U_3 + \left(2+q(x_2)\Delta x^2\right)U_2 + \left(-1-p(x_2)\frac{\Delta x}{2}\right)U_1 = \Delta x^2 f(x_2)
\]

\[\vdots\]

\[
\left(-1+p(x_n)\frac{\Delta x}{2}\right)U_{n+1} + \left(2+q(x_n)\Delta x^2\right)U_n + \left(-1-p(x_n)\frac{\Delta x}{2}\right)U_{n-1} = \Delta x^2 f(x_n)
\]

- So our linear system becomes \(A \vec{U} = \vec{F}\) where \(\vec{U}, \vec{F}\) are as before and \(A\) is
given by the tridiagonal matrix

\[ A = \begin{pmatrix}
2 + q_1 \Delta x^2 & -1 - p_1 \frac{\Delta x}{2} \\
-1 + p_2 \frac{\Delta x}{2} & 2 + q_2 \Delta x^2 & -1 - p_2 \frac{\Delta x}{2} \\
& \ddots & \ddots & \ddots \\
& & -1 + p_n \frac{\Delta x}{2} & 2 + q_n \Delta x^2
\end{pmatrix} \]

where we have used the notation \( q_i = q(x_i), \ p_i = p(x_i) \).

• Note that if \( p(x) \neq 0 \) then our matrix is nonsymmetric. We would have to incorporate a different linear solver since ours only works for symmetric positive definite matrices.
Solving a Nonlinear Two Point BVP

• Suppose we have a BVP like

\[-u''(x) = -\left(\frac{u'(x)}{u(x)}\right)^2, \quad u(0) = 1, \quad u(1) = 2\]

• This DE is nonlinear because the derivative of the unknown appears nonlinearly, i.e., as a quadratic and then it is divided by the unknown \(u(x)\).

• When we discretize the \(u'(x)\) term we use the centered difference as before, but now the term \(\frac{(u'(x))^2}{u(x)}\) appears as

\[
\frac{1}{U_i} \left( \frac{U_{i+1} - U_{i-1}}{2\Delta x} \right)^2 = \frac{U_{i+1}^2 - 2U_{i+1}U_{i-1} + U_{i-1}^2}{2\Delta x U_i}
\]

• This means that our unknowns \(U_i\) appear nonlinearly in our equation.

• When we write the difference equation at each point \(x_i\) we get a nonlinear equation.
• This leads to a **nonlinear system of algebraic equations** to be solved.

• When we looked at nonlinear equations we only considered solving a single nonlinear equation but clearly instead of \( f(x) = 0 \) we could have a nonlinear system \( \vec{F}(\vec{x}) = 0 \) where \( \vec{F} = (f_1, f_2, \ldots, f_n)^T \), \( \vec{x} = (x_1, x, \ldots, x_n)^T \) and

\[
\begin{align*}
f_1(x_1, x_2, \ldots, x_n) &= 0 \\
f_2(x_1, x_2, \ldots, x_n) &= 0 \\
&\vdots \\
f_n(x_1, x_2, \ldots, x_n) &= 0
\end{align*}
\]

• This nonlinear system can be solved by Newton’s method.

• Recall that Newton’s method for a single equation \( f(x) = 0 \) is just

\[
x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}
\]
• This can be generalized for a system of equations to be

$$\bar{x}^{k+1} = \bar{x}^k - J^{-1}(\bar{x}^k) \vec{F}(\bar{x}^k)$$

where $J(\bar{x})$ is the Jacobian matrix of $\vec{F}$ where $J_{ij} = \frac{\partial f_i}{\partial x_j}$ is our matrix of partial derivatives.

• We can rewrite this as a linear system $Ax = b$ as

$$J(\bar{x}^k) \Delta x^{k+1} = -\vec{F}(\bar{x}^k)$$

where $\Delta x^{k+1} = \bar{x}^{k+1} - \bar{x}^k$.

• So at each step of the Newton iteration we solve this linear system for $\Delta x^{k+1}$ and set $\bar{x}^{k+1} = \bar{x}^k + \Delta x^{k+1}$

• This means that instead of solving a single linear system to get the solution of our BVP we have to solve a linear system at each step of Newton’s method which typically could be 5-6 iterations depending on your tolerance.

• Note also that the coefficient matrix is different for each system so we can not save in this way; this is because the coefficient matrix is the Jacobian evaluated at the previous Newton iteration.
• Of course now the user has to provide the Jacobian matrix instead of a fixed coefficient matrix.

• There are other methods for solving nonlinear equations which do not require the solution of a linear system at each step; however they typically lose the (usual) quadratic rate of convergence of Newton’s method.
We can take what we have learned from solving a BVP of one independent variable to solving a BVP of two independent variables.

For example, suppose we want to find $u(x, y)$ which satisfies

$$-(u_{xx} + u_{yy}) = f(x, y) \quad 0 < x < 1, \quad 0 < y < 1$$

where $u(x, y)$ is specified on the boundary of the domain which is a unit square in this case.

In this case we have to discretize our domain in both $x$ and $y$. We have

$$0 = x_0, \quad x_i = x_{i-1} + \Delta x, \quad x_{n+1} = 1$$

$$0 = y_0, \quad y_i = y_{i-1} + \Delta y, \quad y_{m+1} = 1$$

Then we let $U_{ij}$ denote our discrete solution at $(x_i, y_j)$.

Then we can simply replace each second derivative with a difference quotient.
Writing our equation at \((x_i, y_j)\)

\[
\frac{-U_{i+1,j} + 2U_{ij} - U_{i-1,j}}{\Delta x^2} + \frac{-U_{i,j+1} + 2U_{ij} - U_{i,j-1}}{\Delta y^2} = f(x_i, y_j)
\]

- Once again we have a linear system to solve for our \(U_{ij}, i = 1, n, j = 1, m\), i.e., we now have \(nm\) unknowns so if the number of unknowns in the \(x\) and \(y\) directions is the same, we have \(n^2\) unknowns compared with \(n\) unknowns in 1-d. Clearly in 3-d, we would have \(n^3\) unknowns.

- This adversely affects the amount of work to solve the system because recall that the work was quantified in terms of the size of the matrix to some power.

- Also note that in this case we do not get a tridiagonal system. Actually what we get is a banded matrix, i.e., \(A_{ij} = 0\) for \(|i - j| > q\) which can also be solve efficiently.
We can combine the work we did for IVPs and BVPs to solve the problem of finding \( u(x, t) \) satisfying

\[
  u_t - u_{xx} = f(x, t) \quad x_L < x < x_R, \quad t > 0
\]

\[
  u(x, 0) = u_0
\]

\[
  u(x_L, t) = \alpha(t) \quad u(x_R, t) = \beta(t)
\]

Note that we specify \( u \) on the boundaries and \( u \) initially.

This problem is called an initial boundary value problem (IBVP).

Now we let \( U^n_i \) be our approximation to the solution at time \( t^n \) and spatial point \( x_i \).

For example, if we write the difference equation at \( (x_i, t^n) \) and use a Forward
Euler in time and our second centered difference in space we obtain

\[
\frac{U_{i}^{n+1} - U_{i}^{n}}{\Delta t} - \frac{U_{i+1}^{n} - 2U_{i}^{n} + U_{i-1}^{n}}{\Delta x^2} = f(x_i, t^n)
\]

- Note that in this case we have an explicit method and we can solve directly for \(U_{i}^{n+1}\) for all \(i = 1, \ldots, n\).
- However, these explicit methods have severe time step restrictions and so one usually has to take a lot of small time steps which can lead to roundoff error accumulating.
- If, instead, we use a Backward Euler scheme sitting at point \((x_i, t_{n+1}^{\ast})\) we have the difference equation

\[
\frac{U_{i}^{n+1} - U_{i}^{n}}{\Delta t} - \frac{U_{i+1}^{n+1} - 2U_{i}^{n+1} + U_{i-1}^{n+1}}{\Delta x^2} = f(x_i, t^n)
\]

- In this case, at time \(t_{n+1}^{\ast}\) all the \(U_i\) are coupled and so we have to solve a linear system at each time step.
- If our original PDE is nonlinear, then we need to solve a nonlinear system at each time step, i.e., use Newton’s iteration which requires solving several
linear systems at each time step.

- However, we have a good initial guess for Newton’s method at each time step. What is it?

- In summary, I hope that you have seen that you can take what we have learned in solving IVPs and BVPs to obtain finite difference approximations to PDEs.

- Of course, there are a myriad of other methods for approximating solutions to PDEs such as finite element methods, finite volume methods, spectral methods, etc. All are interesting and have certain types of problems where they work best.

CONGRATULATIONS!

You’ve almost made it through the semester!