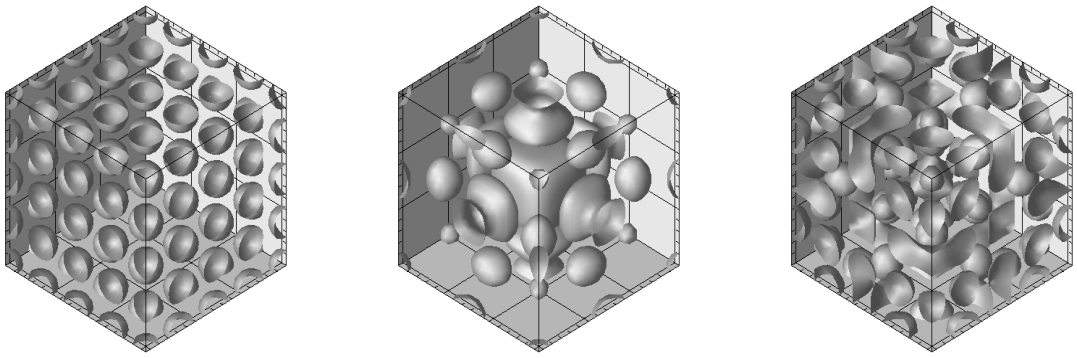
AMATH 301
Beginning Scientific Computing*

J. Nathan Kutz†

January 5, 2005

Abstract

This course is a survey of basic numerical methods and algorithms used for linear algebra, ordinary and partial differential equations, data manipulation and visualization. Emphasis will be on the implementation of numerical schemes to practical problems in the engineering and physical sciences. Full use will be made of MATLAB and its programming functionality.



*These notes are intended as the primary source of information for AMATH 301. The notes are imcomplete and may contain errors. Any other use aside from classroom purposes and personal research please contact me at `kutz@amath.washington.edu`.

†Department of Applied Mathematics, Box 352420, University of Washington, Seattle, WA 98195-2420 (`kutz@amath.washington.edu`).

Contents

1 MATLAB Introduction

The first three sections of these notes deals with the preliminaries necessary for manipulating data, constructing logical statements and plotting data. The remainder of the course relies heavily on proficiency with these basic skills.

1.1 Vectors and Matrices

The manipulation of matrices and vectors is of fundamental importance in MATLAB proficiency. This section deals with the construction and manipulation of basic matrices and vectors. We begin by considering the construction of row and column vectors. Vectors are simply a subclass of matrices which have only a single column or row. A row vector can be created by executing the command struture

```
>>x=[1 3 2]
```

which creates the row vector

$$\vec{x} = (1\ 2\ 3)\,. \tag{1.1.1}$$

Row vectors are oriented in a horizontal fashion. In contrast, column vectors are oriented in a vertical fashion and are created with either of the two following command structures:

```
>>x=[1; 3; 2]
```

where the semicolons indicate advancement to the next row. Otherwise a return can be used in MATLAB to initiate the next row. Thus the following command structure is equivalent

```
>>x=[1
     3
     2]
```

Either one of these creates the column vector

$$\vec{x} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}\,. \tag{1.1.2}$$

All row and column vectors can be created in this way.

Vectors can also be created by use of the colon. Thus the following command line

```
>>x=0:1:10
```

creates a row vector which goes from 0 to 10 in steps of 1 so that

$$\vec{x} = (0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10)\,. \tag{1.1.3}$$

Note that the command line

```
>>x=0:2:10
```

creates a row vector which goes from 0 to 10 in steps of 2 so that

$$\vec{x} = (0\ 2\ 4\ 6\ 8\ 10)\,. \tag{1.1.4}$$

Steps of integer size need not be used. This allows

```
>>x=0:0.2:1
```

to create a row vector which goes from 0 to 1 in steps of 0.2 so that

$$\vec{x} = (0\ 0.2\ 0.4\ 0.6\ 0.8\ 1)\,. \tag{1.1.5}$$

Matrices are just as simple to generate. Now there are a specified number of rows (N) and columns (M). This matrix would be referred to as an $N \times M$ matrix. The 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 3 & 2 \\ 5 & 6 & 7 \\ 8 & 3 & 1 \end{pmatrix} \tag{1.1.6}$$

can be created by use of the semicolons

```
>>A=[1 3 2; 5 6 7; 8 3 1]
```

or by using the return key so that

```
>>A=[1 3 2
    5 6 7
    8 3 1]
```

In this case, the matrix is square with an equal number of rows and columns ($N = M$).

Accessing components of a given matrix of vector is a task that relies on knowing the row and column of a certain piece of data. The coordinate of any data in a matrix is found from its row and column location so that the elements of a matrix $\mathbf{A}$ are given by

$$\mathbf{A}(i, j) \tag{1.1.7}$$

where i denotes the row and j denotes the column. To access the second row and third column of (1.1.6), which takes the value of 7, use the command

```
>>x=A(2,3)
```

This will return the value $x = 7$. To access an entire row, the use of the colon is required

```
>>x=A(2,:)
```

This will access the entire second row and return the row vector

$$\vec{x} = (5\ 6\ 7)\,. \tag{1.1.8}$$

Columns can be similarly be extracted. Thus

```
>>x=A(:,3)
```

will access the entire third column to produce

$$\vec{x} = \begin{pmatrix} 2 \\ 7 \\ 1 \end{pmatrix}\,. \tag{1.1.9}$$

More sophisticated data extraction and manipulation can be performed with the aid of colons. To show examples of these techniques we consider the 4×4 matrix

$$\mathbf{B} = \begin{pmatrix} 1 & 7 & 9 & 2 \\ 2 & 3 & 3 & 4 \\ 5 & 0 & 2 & 6 \\ 6 & 1 & 5 & 5 \end{pmatrix}\,. \tag{1.1.10}$$

The command

```
>>x=B(2:3,2)
```

removes the second through third row of column 2. This produces the column vector

$$\vec{x} = \begin{pmatrix} 3 \\ 0 \end{pmatrix}\,. \tag{1.1.11}$$

The command

```
>>x=B(4,2:end)
```

removes the second through last columns of row 4. This produces the row vector

$$\vec{x} = (1\ 5\ 5)\,. \tag{1.1.12}$$

We can also remove a specified number of rows and columns. The command

```
>>C=B(1:end-1,2:4)
```

removes the first row through the next to last row along with the second through fourth columns. This then produces the matrix

$$\mathbf{C} = \begin{pmatrix} 7 & 9 & 2 \\ 3 & 3 & 4 \\ 0 & 2 & 6 \end{pmatrix}. \tag{1.1.13}$$

As a last example, we make use of the transpose symbol which turns row vectors into column vectors and vice-versa. In this example, the command

```
>>D=[B(1,2:4);  B(1:3,3).']
```

makes the first row of **D** the second through fourth columns of the first row of **B**. The second row of **D**, which is initiated with the semicolon, is made from the transpose (.') of the first three rows of the third column of **B**. This produces the matrix

$$\mathbf{D} = \begin{pmatrix} 7 & 9 & 2 \\ 9 & 3 & 2 \end{pmatrix}. \tag{1.1.14}$$

An important comment about the transpose function is in order. In particular, when transposing a vector with complex numbers, the period must be put in before the ' symbol. Specifically, when considering the transpose of the column vector

$$\vec{x} = \begin{pmatrix} 3+2i \\ 1 \\ 8 \end{pmatrix}. \tag{1.1.15}$$

where i is the complex (imaginary) number $i = \sqrt{-1}$, the command

```
>>y=x.'
```

produces the row vector

$$\vec{y} = (3+2i \ 1 \ 8), \tag{1.1.16}$$

whereas the command

```
>>y=x'
```

produces the row vector

$$\vec{y} = (3-2i \ 1 \ 8). \tag{1.1.17}$$

Thus the use of the ' symbol alone also conjugates vector, i.e. it changes the sign of the imaginary part.

1.2 Logic, Loops and Iterations

The basic building blocks of any MATLAB program are **for** loops and **if** statements. They form the background for carrying out the complicated manipulations required of sophisticated and simple codes alike. This lecture will focus on the use of these two ubiquitous logic structures in programming.

To illustrate the use of the **for** loop structure, we consider some very basic programs which revolve around its implementation. We begin by constructing a loop which will recursively sum a series of numbers. The basic format for such a loop is the following:

```
sum=0
for j=1:5
    sum=sum+j
end
```

This program begins with the variable sum being zero. It then proceeds to go through the **for** loop five times, i.e. the counter j takes the value of one, two, three, four and five in succession. In the loop, the value of sum is updated by adding the current value of j. Thus starting with an initial value of sum equal to zero, we find that sum is equal to 1 ($j = 1$), 3 ($j = 2$), 6 ($j = 3$), 10 ($j = 4$), and 15 ($j = 5$).

The default incremental increase in the counter j is one. However, the increment can be specified as desired. The program

```
sum=0
for j=1:2:5
    sum=sum+j
end
```

is similar to the previous program. But for this case the incremental steps in j are specified to be two. Thus starting with an initial value of sum equal to zero, we find that sum is equal to 1 ($j = 1$), 4 ($j = 3$), and 9 ($j = 5$). And even more generally, the **for** loop counter can be simply given by a row vector. As an example, the program

```
sum=0
for j=[1 5 4]
    sum=sum+j
end
```

will go through the loop three times with the values of j being 1, 5, and 4 successively. Thus starting with an initial value of sum equal to zero, we find that sum is equal to 1 ($j = 1$), 6 ($j = 5$), and 10 ($j = 4$).

The **if** statement is similarly implemented. However, unlike the **for** loop, a series of logic statements are usually considered. The basic format for this logic is as follows:

```
if (logical statement)
      (expressions to execute)
elseif (logical statement)
      (expressions to execute)
elseif (logical statement)
      (expressions to execute)
else
      (expressions to execute)
end
```

In this logic structure, the last set of expressions are executed if the three previous logical statements do not hold.

In practice, the logical **if** may only be required to execute a command if something is true. Thus there would be no need for the *else* logic structure. Such a logic format might be as follows:

```
if (logical statement)
      (expressions to execute)
elseif (logical statement)
      (expressions to execute)
end
```

In such a structure, no commands would be executed if the logical statements do not hold.

To make a practical example of the use of **for** and **if** statements, we consider the bisection method for finding zeros of a function. In particular, we will consider the transcendental function

$$\exp(x) - \tan(x) = 0 \tag{1.2.1}$$

for which the values of x which make this true must be found computationally. We can begin to get an idea of where the relevant values of x are by plotting this function. The following MATLAB script will plot the function over the interval $x \in [-10, 10]$.

```
clear all  % clear all variables
close all  % close all figures

x=-10:0.1:10;     % define plotting range
y=exp(x)-tan(x);  % define function to consider
plot(x,y)         % plot the function
```

It should be noted that $\tan(x)$ takes on the value of $\pm\infty$ at $\pi/2 + 2n\pi$ where $n = \cdots, -2, -1, 0, 1, 2, \cdots$. By zooming in to smaller values of the function, one can find that there are a large number (infinite) of roots to this equation. In

particular, there is a root located between $x \in [-4, 2.8]$. At $x = -4$, the function $\exp(x) - \tan(x) > 0$ while at $x = -2.8$ the function $\exp(x) - \tan(x) < 0$. Thus, in between these points lies a root.

The bisection method simply cuts the given interval in half and determines if the root is on the left or right side of the cut. Once this is established, a new interval is chosen with the mid point now becoming the left or right end of the new domain, depending of course on the location of the root. This method is repeated until the interval has become so small and the function considered has come to within some tolerance of zero. The following algorithm uses an outside **for** loop to continue cutting the interval in half while the imbedded **if** statement determines the new interval of interest. A second **if** statement if used to ensure that once a certain tolerance has been achieved, i.e. the absolute value of the function $\exp(x) - \tan(x)$ is less than 10^{-5}, then the iteration process is stopped.

bisection.m

```
clear all  % clear all variables

xr=-2.8;  % initial right boundary
xl=-4;    % initial left boundary

for j=1:1000  % j cuts the interval

   xc=(xl+xr)/2;        % calculate the midpoint
   fc=exp(xc)-tan(xc);  % calculate function

   if fc>0
       xl=xc;  % move left boundary
   else
       xr=xc;  % move right boundary
   end

   if abs(fc)<10^(-5)
       break   % quit the loop
   end

end
xc   % print value of root
fc   % print value of function
```

Note that the **break** command ejects you from the current loop. In this case, that is the j loop. This effectively stops the iteration procedure for cutting the intervals in half. Further, extensive use has been made of the semicolons at the

end of each line. The semicolon simply represses output to the computer screen, saving valuable time and clutter.

1.3 Plotting and Importing/Exporting Data

The graphical representation of data is a fundamental aspect of any technical scientific communications. MATLAB provides an easy and powerful interface for plotting and representing a large variety of data formats. Like all MATLAB structures, plotting is dependent on the effective manipulation of vectors and matrices.

To begin the consideration of graphical routines, we first construct a set of functions to plot and manipulate. To define a function, the plotting domain must first be specified. This can be done by using a routine which creates a row vector

```
x1=-10:0.1:10;
y1=sin(x);
```

Here the row vector $\vec{x}$ spans the interval $x \in [-10, 10]$ in steps of $\Delta x = 0.1$. The second command creates a row vector $\vec{y}$ which gives the values of the sine function evaluated at the corresponding values of $\vec{x}$. A basic plot of this function can then be generated by the command

```
plot(x1,y1)
```

Note that this graph lacks a title or axis labels. These are important in generating high quality graphics.

The preceding example considers only equally spaced points. However, MATLAB will generate functions values for any specified coordinate values. For instance, the two lines

```
x2=[-5 sqrt(3) pi 4];
y2=sin(x2);
```

will generate the values of the sine function at $x = -5, \sqrt{3}, \pi$ and 4. The **linspace** command is also helpful in generating a domain for defining and evaluating functions. The basic procedure for this function is as follows

```
x3=linspace(-10,10,64);
y3=x3.*sin(x3);
```

This will generate a row vector **x** which goes from -10 to 10 in 64 steps. This can often be a helpful function when considering intervals where the number of discretized steps give a complicated Δx. In this example, we are considering the function $x \sin x$ over the interval $x \in [-10, 10]$. By doing so, the period must be

included before the multiplication sign. This will then perform a *component by component* multiplication. Thus creating a row vector with the values of $x \sin x$.

To plot all of the above data sets in a single plot, we can do either one of the following routines.

```
figure(1)
plot(x1,y1), hold on
plot(x2,y2)
plot(x3,y3)
```

In this case, the **hold on** command is necessary after the first plot so that the second plot command will not overwrite the first data plotted. This will plot all three data sets on the same graph with a default blue line. This graph will be *figure 1*. Alternatively, all the data sets can be plotted on the same graph with

```
figure(2)
plot(x1,y1,x2,y2,x3,y3)
```

For this case, the three pairs of vectors are prescribed within a single plot command. This figure generated will be *figure 2*. And advantage to this method is that the three data sets will be of different colors, which is better than having them all the default color of blue.

This is only the beginning of the plotting and visualization process. Many aspects of the graph must be augmented with specialty commands in order to more accurately relay the information. Of significance is the ability to change the line colors and styles of the plotted data. By using the **help plot** command, a list of options for customizing data representation is given. In the following, a new figure is created which is customized as follows

```
figure(3)
plot(x1,y1,x2,y2,'g*',x3,y3,'m0:')
```

This will create the same plot as in *figure 2*, but now the second data set is represented by green * and the third data set is represented by a magenta dotted line with the actual data points given by a magenta hollow circle. This kind of customization greatly helps distinguish the various data sets and their individual behaviors.

Labeling the axis and placing a title on the figure is also of fundamental importance. This can be easily accomplished with the commands

```
xlabel('x values')
ylabel('y values')
title('Example Graph')
```

The strings given within the ' sign are now printed in a centered location along the x-axis, y-axis and title location respectively.

A legend is then important to distinguish and label the various data sets which are plotted together on the graph. Within the **legend** command, the position of the legend can be easily specified. The command **help legend** gives a summary of the possible legend placement positions, one of which is to the right of the graph and does not interfere with the graph data. To place a legend in the above graph in an *optimal* location, the following command is used.

```
legend('Data set 1','Data set 2','Data set 3',0)
```

Here the strings correspond to the the three plotted data sets in the order they were plotted. The zero at the end of the **legend** command is the option setting for placement of the legend box. In this case, the option zero tries to pick the best possible location which does not interfere with any of the data.

subplots

Another possible method for representing data is with the **subplot** command. This allows for multiple graphs to be arranged in a row and column format. In this example, we have three data sets which are under consideration. To plot each data set in an individual subplot requires three plot frames. Thus we construct a plot containing three rows and a single column. The format for this command structure is as follows:

```
figure(4)
subplot(3,1,1), plot(x1,y1), axis([-10 10 -10 10])
subplot(3,1,2), plot(x2,y2,'*'), axis([-10 10 -10 10])
subplot(3,1,3), plot(x3,y3,'m0:')
```

Note that the **subplot** command is of the structure **subplot(row,column,graph number)** where the graph number refers to the current graph being considered. In this example, the axis command has also been used to make all the graphs have the same x and y values. The command structure for the axis command is **axis([xmin xmax ymin ymax])**. For each subplot, the use of the **legend**, **xlabel**, **ylabel**, and **title** command can be used.

Remark 1: All the graph customization techniques discussed can be performed directly within the MATLAB graphics window. Simply go to the edit button and choose to edit axis properties. This will give full control of all the axis properties discussed so far and more. However, once MATLAB is shut down or the graph is closed, all the customization properties are lost and you must start again from scratch. This gives an advantage to developing a nice set of commands in a .m file to customize your graphics.

Remark 2: To put a grid on the a graph, simply use the **grid on** command. To take it off, use **grid off**. The number of grid lines can be controlled from the editing options in the graphics window. This feature can aid in illustrating more clearly certain phenomena and should be considered when making plots.

Remark 3: To put a variable value in a string, the **num2str** (number to string) command can be used. For instance, the code

```
rho=1.5;
title(['value of rho=' num2str(rho)])
```

creates a title which reads "value of rho=1.5". Thus this command converts variable values into useful string configurations.

load, save and print

Once a graph has been made or data generated, it may be useful to save the data or print the graphs for inclusion in a write up or presentation. The **save** and **print** commands are the appropriate commands for performing such actions.

The **save** command will write any data out to a file for later use in a separate program or for later use in MATLAB. To save workspace variables to a file, the following command structure is used

```
save filename
```

this will save all the current workspace variables to a binary file named **filename.mat**. In the preceding example, this will save all the vectors created along with any graphics settings. This is an extremely powerful and easy command to use. However, you can only access the data again through MATLAB. To recall this data at a future time, simply load it back into MATLAB with the command

```
load filename
```

This will reload into the workspace all the variables saved in **filename.mat**. This command is ideal when closing down operations on MATLAB and resuming at a future time.

Alternatively, it may be advantageous to save data for use in a different software package. In this case, data needs to be saved in an ASCII format in order to be read by other software engines. The **save** command can then be modified for this purpose. The command

```
save x1 x1.dat -ASCII
```

saves the row vector **x1** generated previously to the file **x1.dat** in ASCII format. This can then be loaded back into MATLAB with the command

```
load x1.dat
```

This saving option is advantageous when considering the use of other software packages to manipulate or analyze data generated in MATLAB.

If you desire to print a figure to a file, the the **print** command needs to be utilized. There a large number of graphics formats which can be used. By typing **help print**, a list of possible options will be listed. A common graphics format would involve the command

```
print -djpeg fig.jpg
```

which will print the current figure as a jpeg file named **fig.jpg**. Note that you can also print or save from the figure window by pressing the **file** button and following the links to the print or export option respectively.

2 Linear Systems

The solution of linear systems is one of the most basic aspects of computational science. In many applications, the solution technique often gives rise to a system of linear equations which need to be solved as efficiently as possible. In addition to Gaussian elimination, there are a host of other techniques which can be used to solve a given problem. This section offers an overview for these methods and techniques.

2.1 Direct solution methods for Ax=b

A central concern in almost any computational strategy is a fast and efficient computational method for achieving a solution of a large system of equations $\mathbf{Ax} = \mathbf{b}$. In trying to render a computation tractable, it is crucial to minimize the operations it takes in solving such a system. There are a variety of direct methods for solving $\mathbf{Ax} = \mathbf{b}$: Gaussian elimination, LU decomposition, and inverting the matrix $\mathbf{A}$. In addition to these direct methods, iterative schemes can also provide efficient solution techniques. Some basic iterative schemes will be discussed in what follows.

The standard beginning to discussions of solution techniques for $\mathbf{Ax} = \mathbf{b}$ involves Gaussian elimination. We will consider a very simple example of a 3×3 system in order to understand the operation count and numerical procedure involved in this technique. Thus consider $\mathbf{Ax} = \mathbf{b}$ with

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}. \tag{2.1.1}$$

The Gaussian elimination procedure begins with the construction of the augmented matrix

$$\begin{aligned}
[\mathbf{A}|\mathbf{b}] &= \left[\begin{array}{ccc|c} \underline{\mathbf{1}} & 1 & 1 & 1 \\ 1 & 2 & 4 & -1 \\ 1 & 3 & 9 & 1 \end{array}\right] \\
&= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & -2 \\ 0 & 2 & 8 & 0 \end{array}\right] \\
&= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & \underline{\mathbf{1}} & 3 & -2 \\ 0 & 1 & 4 & 0 \end{array}\right] \\
&= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & -2 \\ 0 & 0 & 1 & 2 \end{array}\right]
\end{aligned} \tag{2.1.2}$$

where we have underlined and bolded the pivot of the augmented matrix. Back substituting then gives the solution

$$x_3 = 2 \quad \rightarrow \; x_3 = 2 \tag{2.1.3a}$$

$$x_2 + 3x_3 = -2 \quad \rightarrow \; x_2 = -8 \tag{2.1.3b}$$

$$x_1 + x_2 + x_3 = 1 \quad \rightarrow \; x_1 = 7\,. \tag{2.1.3c}$$

This procedure can be carried out for any matrix $\mathbf{A}$ which is nonsingular, i.e. $\det \mathbf{A} \neq 0$. In this algorithm, we simply need to avoid these singular matrices and occasionally shift the rows around to avoid a zero pivot. Provided we do this, it will always yield an answer.

The fact that this algorithm works is secondary to the concern of the time required in generating a solution in scientific computing. The operation count for the Gaussian elimination can be easily be estimated from the algorithmic procedure for an $N \times N$ matrix:

1. Movement down the N pivots
2. For each pivot, perform N additions/subtractions across a given row.
3. For each pivot, perform the addition/subtraction down the N rows.

In total, this results in a scheme whose operation count is $O(N^3)$. The back substitution algorithm can similarly be calculated to give an $O(N^2)$ scheme.

LU Decomposition

Each Gaussian elimination operation costs $O(N^3)$ operations. This can be computationally prohibitive for large matrices when repeated solutions of $\mathbf{Ax} = \mathbf{b}$

must be found. When working with the same matrix $\mathbf{A}$ however, the operation count can easily be brought down to $O(N^2)$ using LU factorization which splits the matrix $\mathbf{A}$ into a lower triangular matrix L, and an upper triangular matrix U. For a 3×3 matrix, the LU factorization scheme splits $\mathbf{A}$ as follows:

$$\mathbf{A}=\mathbf{LU} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}. \tag{2.1.4}$$

Thus the $\mathbf{L}$ matrix is lower triangular and the $\mathbf{U}$ matrix is upper triangular. This then gives

$$\mathbf{Ax} = \mathbf{b} \ \rightarrow \ \mathbf{LUx} = \mathbf{b} \tag{2.1.5}$$

where by letting $\mathbf{y} = \mathbf{Ux}$ we find the coupled system

$$\mathbf{Ly} = \mathbf{b} \ \text{ and } \ \mathbf{Ux} = \mathbf{y} \tag{2.1.6}$$

where the system $\mathbf{Ly} = \mathbf{b}$

$$y_1 = b_1 \tag{2.1.7a}$$
$$m_{21}y_1 + y_2 = b_2 \tag{2.1.7b}$$
$$m_{31}y_1 + m_{32}y_2 + y_3 = b_3 \tag{2.1.7c}$$

can be solved by $O(N^2)$ forward substitution and the system $\mathbf{Ux} = \mathbf{y}$

$$u_{11}x_1 + u_{12}x_2 + x_3 = y_1 \tag{2.1.8a}$$
$$u_{22}x_2 + u_{23}x_3 = y_2 \tag{2.1.8b}$$
$$u_{33}x_3 = y_3 \tag{2.1.8c}$$

can be solved by $O(N^2)$ back substitution. Thus once the factorization is accomplished, the LU results in an $O(N^2)$ scheme for arriving at the solution. The factorization itself is $O(N^3)$, but you only have to do this once. Note, you should **always** use LU decomposition if possible. Otherwise, you are doing far more work than necessary in achieving a solution.

As an example of the application of the LU factorization algorithm, we consider the 3×3 matrix

$$\mathbf{A}=\begin{pmatrix} 4 & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{pmatrix}. \tag{2.1.9}$$

The factorization starts from the matrix multiplication of the matrix $\mathbf{A}$ and the identity matrix $\mathbf{I}$

$$\mathbf{A} = \mathbf{IA} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \underline{\mathbf{4}} & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{pmatrix}. \tag{2.1.10}$$

The factorization begins with the pivot element. To use Gaussian elimination, we would multiply the pivot by $-1/2$ to eliminate the first column element in the second row. Similarly, we would multiply the pivot by $1/4$ to eliminate the first column element in the third row. These multiplicative factors are now part of the first matrix above:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 3 & -1 \\ 0 & \underline{\mathbf{-2.5}} & 4.5 \\ 0 & 1.25 & 6.25 \end{pmatrix}. \tag{2.1.11}$$

To eliminate on the third row, we use the next pivot. This requires that we multiply by $-1/2$ in order to eliminate the second column, third row. Thus we find

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & -1/2 & 1 \end{pmatrix} \begin{pmatrix} 4 & 3 & -1 \\ 0 & -2.5 & 4.5 \\ 0 & 0 & 8.5 \end{pmatrix}. \tag{2.1.12}$$

Thus we find that

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & -1/2 & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{U} = \begin{pmatrix} 4 & 3 & -1 \\ 0 & -2.5 & 4.5 \\ 0 & 0 & 8.5 \end{pmatrix}. \tag{2.1.13}$$

It is easy to verify by direct substitution that indeed $\mathbf{A} = \mathbf{LU}$. Just like Gaussian elimination, the cost of factorization is $O(N^3)$. However, once $\mathbf{L}$ and $\mathbf{U}$ are known, finding the solution is an $O(N^2)$ operation.

The Permutation Matrix

As will often happen with Gaussian elimination, following the above algorithm will at times result in a zero pivot. This is easily handled in Gaussian elimination by shifting rows in order to find a non-zero pivot. However, in LU decomposition, we must keep track of this row shift since it will effect the right hand side vector $\mathbf{b}$. We can keep track of row shifts with a row permutation matrix $\mathbf{P}$. Thus if we need to permute two rows, we find

$$\mathbf{Ax} = \mathbf{b} \rightarrow \mathbf{PAx} = \mathbf{Pb} \rightarrow \mathbf{PLUx} = \mathbf{Pb} \tag{2.1.14}$$

thus $\mathbf{PA} = \mathbf{LU}$. To shift rows one and two, for instance, we would have

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & \cdots \\ 1 & 0 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & & & \end{pmatrix}. \tag{2.1.15}$$

If permutation is necessary, MATLAB can supply the permutation matrix associated with the LU decomposition.

MATLAB: A \ b

Given the alternatives for solving the linear system $\mathbf{Ax} = \mathbf{b}$, it is important to know how the MATLAB command structure for $\mathbf{A} \setminus \mathbf{b}$ works. The following is an outline of the algorithm performed.

1. It first checks to see if $\mathbf{A}$ is triangular, or some permutation thereof. If it is, then all that is needed is a simple $O(N^2)$ substitution routine.
2. It then checks if $\mathbf{A}$ is symmetric, i.e. Hermitian or Self-Adjoint. If so, a Cholesky factorization is attempted. If $\mathbf{A}$ is positive definite, the Cholesky algorithm is always succesful and takes half the run time of LU factorization.
3. It then checks if $\mathbf{A}$ is Hessenberg. If so, it can be written as an upper triangular matrix and solved by a substitution routine.
4. If all the above methods fail, then LU factorization is used and the forward and backward substitution routines generate a solution.
5. If $\mathbf{A}$ is not square, a QR (Householder) routine is used to solve the system.
6. If $\mathbf{A}$ is not square and sparse, a least squares solution using QR factorization is performed.

Note that solving by $\mathbf{b} = \mathbf{A}^{-1}\mathbf{x}$ is the slowest of all methods, taking 2.5 times longer or more than $\mathbf{A} \setminus \mathbf{b}$. It is not recommended. However, just like LU factorization, once the inverse is known it need not be calculated again. Care must also be taken when the $\det \mathbf{A} \approx 0$, i.e. the matrix is ill-conditioned.

MATLAB commands

The commands for executing the linear system solve are as follows

- **$\mathbf{A} \setminus \mathbf{b}$:** Solve the system in the order above.
- **$[L, U] = lu(A)$:** Generate the L and U matrices.
- **$[L, U, P] = lu(A)$:** Generate the L and U factorization matrices along with the permutation matrix P.

2.2 Iterative solution methods for Ax=b

In addition to the standard techniques of Gaussian elimination or LU decomposition for solving $\mathbf{Ax} = \mathbf{b}$, a wide range of iterative techniques are available. These iterative techniques can often go under the name of Krylov space methods [6]. The idea is to start with an initial guess for the solution and develop an iterative procedure that will converge to the solution. The simplest example

of this method is known as a Jacobi iteration scheme. The implementation of this scheme is best illustrated with an example. We consider the linear system

$$4x - y + z = 7 \tag{2.2.1a}$$
$$4x - 8y + z = -21 \tag{2.2.1b}$$
$$-2x + y + 5z = 15\,. \tag{2.2.1c}$$

We can rewrite each equation as follows

$$x = \frac{7 + y - z}{4} \tag{2.2.2a}$$
$$y = \frac{21 + 4x + z}{8} \tag{2.2.2b}$$
$$z = \frac{15 + 2x - y}{5}\,. \tag{2.2.2c}$$

To solve the system iteratively, we can define the following Jacobi iteration scheme based on the above

$$x_{k+1} = \frac{7 + y_k - z_k}{4} \tag{2.2.3a}$$
$$y_{k+1} = \frac{21 + 4x_k + z_k}{8} \tag{2.2.3b}$$
$$z_{k+1} = \frac{15 + 2x_k - y_k}{5}\,. \tag{2.2.3c}$$

An algorithm is then easily implemented computationally. In particular, we would follow the structure:

1. Guess initial values: (x_0, y_0, z_0).
2. Iterate the Jacobi scheme: $\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k$.
3. Check for convergence: $\| \mathbf{x}_{k+1} - \mathbf{x}_k \|$<tolerance.

Note that the choice of an initial guess is often critical in determining the convergence to the solution. Thus the more that is known about what the solution is supposed to look like, the higher the chance of successful implementation of the iterative scheme. Table 1 shows the convergence of this scheme for this simple example.

Given the success of this example, it is easy to conjecture that such a scheme will always be effective. However, we can reconsider the original system by interchanging the first and last set of equations. This gives the system

$$-2x + y + 5z = 15 \tag{2.2.4a}$$
$$4x - 8y + z = -21 \tag{2.2.4b}$$
$$4x - y + z = 7\,. \tag{2.2.4c}$$

k	x_k	y_k	z_k
0	1.0	2.0	2.0
1	1.75	3.375	3.0
2	1.84375	3.875	3.025
⋮	⋮	⋮	⋮
15	1.99999993	3.99999985	2.9999993
⋮	⋮	⋮	⋮
19	2.0	4.0	3.0

Table 1: Convergence of Jacobi iteration scheme to the solution value of $(x, y, z) = (2, 4, 3)$ from the initial guess $(x_0, y_0, z_0) = (1, 2, 2)$.

k	x_k	y_k	z_k
0	1.0	2.0	2.0
1	-1.5	3.375	5.0
2	6.6875	2.5	16.375
3	34.6875	8.015625	-17.25
⋮	⋮	⋮	⋮
	$\pm\infty$	$\pm\infty$	$\pm\infty$

Table 2: Divergence of Jacobi iteration scheme from the initial guess $(x_0, y_0, z_0) = (1, 2, 2)$.

To solve the system iteratively, we can define the following Jacobi iteration scheme based on this rearranged set of equations

$$x_{k+1} = \frac{y_k + 5z_k - 15}{2} \tag{2.2.5a}$$

$$y_{k+1} = \frac{21 + 4x_k + z_k}{8} \tag{2.2.5b}$$

$$z_{k+1} = y_k - 4x_k + 7\,. \tag{2.2.5c}$$

Of course, the solution should be exactly as before. However, Table 2 shows that applying the iteration scheme leads to a set of values which grow to infinity. Thus the iteration scheme quickly fails.

Strictly Diagonal Dominant

The difference in the two Jacobi schemes above involves the iteration procedure being strictly diagonal dominant. We begin with the definition of strict diagonal

dominance. A matrix $\mathbf{A}$ is strictly diagonal dominant if for each row, the sum of the absolute values of the off-diagonal terms is less than the absolute value of the diagonal term:

$$|a_{kk}| > \sum_{j=1, j\neq k}^{N} |a_{kj}| \,. \tag{2.2.6}$$

Strict diagonal dominance has the following consequence; given a stricly diagonal dominant matrix $\mathbf{A}$, then $\mathbf{Ax} = \mathbf{b}$ has a unique solution $\mathbf{x} = \mathbf{p}$. Jacobi iteration produces a sequence $\mathbf{p}_k$ that will converge to $\mathbf{p}$ for any $\mathbf{p}_0$. For the two examples considered here, this property is crucial. For the first example (2.2.1), we have

$$\mathbf{A} = \begin{pmatrix} 4 & -1 & 1 \\ 4 & -8 & 1 \\ -2 & 1 & 5 \end{pmatrix} \rightarrow \begin{array}{l} \text{row 1: } |4| > |-1| + |1| = 2 \\ \text{row 2: } |-8| > |4| + |1| = 5 \\ \text{row 3: } |5| > |2| + |1| = 3 \end{array} , \tag{2.2.7}$$

which shows the system to be strictly diagonal dominant and guaranteed to converge. In contrast, the second system (2.2.4) is not stricly diagonal dominant as can be seen from

$$\mathbf{A} = \begin{pmatrix} -2 & 1 & 5 \\ 4 & -8 & 1 \\ 4 & -1 & 1 \end{pmatrix} \rightarrow \begin{array}{l} \text{row 1: } |-2| < |1| + |5| = 6 \\ \text{row 2: } |-8| > |4| + |1| = 5 \\ \text{row 3: } |1| < |4| + |-1| = 5 \end{array} . \tag{2.2.8}$$

Thus this scheme is not guaranteed to converge. Indeed, it diverges to infinity.

Modification and Enhancements: Gauss-Seidel

It is sometimes possible to enhance the convergence of a scheme by applying modifications to the basic Jacobi scheme. For instance, the Jacobi scheme given by (2.2.3) can be enhanced by the following modifications

$$x_{k+1} = \frac{7 + y_k - z_k}{4} \tag{2.2.9a}$$

$$y_{k+1} = \frac{21 + 4x_{k+1} + z_k}{8} \tag{2.2.9b}$$

$$z_{k+1} = \frac{15 + 2x_{k+1} - y_{k+1}}{5} \,. \tag{2.2.9c}$$

Here use is made of the supposedly improved value x_{k+1} in the second equation and x_{k+1} and y_{k+1} in the third equation. This is known as the Gauss-Seidel scheme. Table 3 shows that the Gauss-Seidel procedure converges to the solution in half the number of iterations used by the Jacobi scheme.

Unlike the Jacobi scheme, the Gauss-Seidel method is not guaranteed to converge even in the case of strict diagonal dominance. Further, the Gauss-Seidel modification is only one of a large number of possible changes to the iteration

k	x_k	y_k	z_k
0	1.0	2.0	2.0
1	1.75	3.75	2.95
2	1.95	3.96875	2.98625
⋮	⋮	⋮	⋮
10	2.0	4.0	3.0

Table 3: Convergence of Jacobi iteration scheme to the solution value of $(x, y, z) = (2, 4, 3)$ from the initial guess $(x_0, y_0, z_0) = (1, 2, 2)$.

scheme which can be implemented in an effort to enhance convergence. It is also possible to use several previous iterations to achieve convergence. Krylov space methods [6] are often high end iterative techniques especially developed for rapid convergence. Included in these iteration schemes are conjugant gradient methods and generalized minimum residual methods which we will discuss and implement [6].

2.3 Eigenvalues, Eigenvectors, and Solvability

Another class of linear systems of equations which are of fundamental importance are known as eigenvalue problems. Unlike the system $\mathbf{Ax} = \mathbf{b}$ which has the single unknown vector $\vec{x}$, eigenvalue problems are of the form

$$\mathbf{Ax} = \lambda\mathbf{x} \tag{2.3.1}$$

which have the unknows $\mathbf{x}$ and λ. The values of λ are known as the *eigenvalues* and the corresponding $\mathbf{x}$ are the *eigenvectors*.

Eigenvalue problems often arise from differential equations. Specifically, we consider the example of a linear set of coupled differential equations

$$\frac{d\mathbf{y}}{dt} = \mathbf{Ay}\,. \tag{2.3.2}$$

By attempting a solution of the form

$$\mathbf{y} = \mathbf{x}\exp(\lambda t)\,, \tag{2.3.3}$$

where all the time-dependence is captured in the exponent, the resulting equation for $\mathbf{x}$ is

$$\mathbf{Ax} = \lambda\mathbf{x} \tag{2.3.4}$$

which is just the eigenvalue problem. Once the full set of eigenvalues and eigenvectors of this equation are found, the solution of the differential equation is written as

$$\vec{y} = c_1\mathbf{x}_1\exp(\lambda_1 t) + c_2\mathbf{x}_2\exp(\lambda_2 t) + \cdots + c_N\mathbf{x}_N\exp(\lambda_N t) \tag{2.3.5}$$

where N is the number of linearly independent solutions to the eigenvalue problem for the matrix $\mathbf{A}$ which is of size $N \times N$. Thus solving a linear system of differential equations relies on the solution of an associated eigenvalue problem.

The questions remains: how are the eigenvalues and eigenvectors found? To consider this problem, we rewrite the eigenvalue problem as

$$\mathbf{Ax} = \lambda\mathbf{Ix} \tag{2.3.6}$$

where a multiplication by unity has been performed, i.e. $\mathbf{Ix} = \mathbf{x}$. Moving the right hand side to the left side of the equation gives

$$\mathbf{Ax} - \lambda\mathbf{Ix} = \mathbf{0}. \tag{2.3.7}$$

Factoring out the vector $\mathbf{x}$ then gives the desired result

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0}\,. \tag{2.3.8}$$

Two possibilities now exist.

Option I: The determinant of the matrix $(\mathbf{A}-\lambda\mathbf{I})$ is not zero. If this is true, the matrix is *nonsingular* and its inverse, $(\mathbf{A} - \lambda\mathbf{I})^{-1}$, can be found. The solution to the eigenvalue problem (2.3.8) is then

$$\mathbf{x} = (\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{0} \tag{2.3.9}$$

which implies that

$$\mathbf{x} = \mathbf{0}\,. \tag{2.3.10}$$

This trivial solution could have been guessed from (2.3.8). However, it is not relevant as we require nontrivial solutions for $\mathbf{x}$.

Option II: The determinant of the matrix $(\mathbf{A} - \lambda\mathbf{I})$ is zero. If this is true, the matrix is *singular* and its inverse, $(\mathbf{A} - \lambda\mathbf{I})^{-1}$, cannot be found. Although there is no longer a guarantee that there is a solution, it is the only scenario which allows for the possibility of $\mathbf{x} \neq \mathbf{0}$. It is this condition which allows for the construction of eigenvalues and eigenvectors. Indeed, we choose the eigenvalues λ so that this condition holds and the matrix is singular.

To illustrate how the eigenvalues and eigenvectors are computed, an example is shown. Consider the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 3 \\ -1 & 5 \end{pmatrix} \tag{2.3.11}$$

This gives the eigenvalue problem

$$\mathbf{A} = \begin{pmatrix} 1 & 3 \\ -1 & 5 \end{pmatrix}\mathbf{x} = \lambda\mathbf{x} \tag{2.3.12}$$

which when manipulated to the form $(\mathbf{A}-\lambda\mathbf{I})\mathbf{x}=\mathbf{0}$ gives

$$\left[\begin{pmatrix} 1 & 3 \\ -1 & 5 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\right]\mathbf{x} = \begin{pmatrix} 1-\lambda & 3 \\ -1 & 5-\lambda \end{pmatrix}\mathbf{x} = 0\,. \tag{2.3.13}$$

We now require that the determinant is zero

$$\det\begin{vmatrix} 1-\lambda & 3 \\ -1 & 5-\lambda \end{vmatrix} = (1-\lambda)(5-\lambda)+3 = \lambda^2-6\lambda+8 = (\lambda-2)(\lambda-4) = 0 \tag{2.3.14}$$

which gives the two eigenvalues

$$\lambda = 2, 4\,. \tag{2.3.15}$$

The eigenvectors are then found from (2.3.13) as follows:

$$\lambda = 2: \quad \begin{pmatrix} 1-2 & 3 \\ -1 & 5-2 \end{pmatrix}\mathbf{x} = \begin{pmatrix} -1 & 3 \\ -1 & 3 \end{pmatrix}\mathbf{x} = 0\,. \tag{2.3.16}$$

Given that $\mathbf{x} = (x_1 \;\; x_2)^T$, this leads to the single equation

$$-x_1 + 3x_2 = 0 \tag{2.3.17}$$

This is an underdetermined system of equations. Thus we have freedom in choosing one of the values. Choosing $x_2 = 1$ gives $x_1 = 3$ and

$$\mathbf{x}_1 = \begin{pmatrix} 3 \\ 1 \end{pmatrix}. \tag{2.3.18}$$

The second eigenvector comes from (2.3.13) as follows:

$$\lambda = 4: \quad \begin{pmatrix} 1-4 & 3 \\ -1 & 5-4 \end{pmatrix}\mathbf{x} = \begin{pmatrix} -3 & 3 \\ -1 & 1 \end{pmatrix}\mathbf{x} = 0\,. \tag{2.3.19}$$

Given that $\mathbf{x} = (x_1 \;\; x_2)^T$, this leads to the single equation

$$-x_1 + x_2 = 0 \tag{2.3.20}$$

This is an underdetermined system of equations. Thus we have freedom in choosing one of the values. Choosing $x_2 = 1$ gives $x_1 = 1$ and

$$\mathbf{x}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \tag{2.3.21}$$

These results can be found from MATLAB by using the **eig** command. Specifically, the command structure

```
[V,D]=eig(A)
```

gives the matrix $\mathbf{V}$ containing the eigenvectors as columns and the matrix $\mathbf{D}$ whose diagonal elements are the corresponding eigenvalues.

Matrix Powers

Another important operation which can be performed with eigenvalue and eigenvectors is the evaluation of

$$\mathbf{A}^M \tag{2.3.22}$$

where M is a large integer. For large matrices $\mathbf{A}$, this operation is computationally expensive. However, knowing the eigenvalues and eigenvectors of of $\mathbf{A}$ allows for a significant ease in computational expense. Assuming we have all the eigenvalues and eigenvectors of $\mathbf{A}$, then

$$\begin{aligned}
\mathbf{A}\mathbf{x}_1 &= \lambda_1 \mathbf{x}_1 \\
\mathbf{A}\mathbf{x}_2 &= \lambda_2 \mathbf{x}_2 \\
&\vdots \\
\mathbf{A}\mathbf{x}_n &= \lambda_1 \mathbf{x}_n \,.
\end{aligned}$$

This collection of eigenvalues and eigenvectors gives the matrix system

$$\mathbf{AS} = \mathbf{S}\mathbf{\Lambda} \tag{2.3.23}$$

where the columns of the matrix $\mathbf{S}$ are the eigenvectors of $\mathbf{A}$,

$$\mathbf{S} = (\mathbf{x}_1 \;\; \mathbf{x}_2 \;\; \cdots \;\; \mathbf{x}_n) \,, \tag{2.3.24a}$$

and $\mathbf{\Lambda}$ is a matrix whose diagonals are the corresponding eigenvalues

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & \lambda_n \end{pmatrix} . \tag{2.3.25}$$

By multiplying (2.3.24)on the right by $\mathbf{S}^{-1}$, the matrix $\mathbf{A}$ can then be rewritten as

$$\mathbf{A} = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1} \,. \tag{2.3.26}$$

The final observation comes from

$$\mathbf{A}^2 = (\mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1})(\mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}) = \mathbf{S}\mathbf{\Lambda}^2\mathbf{S}^{-1} \,. \tag{2.3.27}$$

This then generalizes to

$$\mathbf{A}^M = \mathbf{S}\mathbf{\Lambda}^M\mathbf{S}^{-1} \tag{2.3.28}$$

where the matrix $\mathbf{\Lambda}^M$ is easily calculated as

$$\mathbf{\Lambda}^M = \begin{pmatrix} \lambda_1^M & 0 & \cdots & & 0 \\ 0 & \lambda_2^M & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & \lambda_n^M \end{pmatrix} . \tag{2.3.29}$$

Since raising the diagonal terms to the M^{th} power is easily accomplished, the matrix $\mathbf{A}$ can then be easily calculated by multiplying the three matrices in (2.3.28)

Solvability and the Fredholm-Alternative Theorem

It is easy to ask under what conditions the system

$$\mathbf{Ax} = \mathbf{b} \tag{2.3.30}$$

can be solved. Aside from requiring the det$\mathbf{A} \neq 0$, we also have a solvability condition on $\mathbf{b}$. Consider the adjoint problem

$$\mathbf{A}^\dagger \mathbf{y} = 0 \tag{2.3.31}$$

where $\mathbf{A}^\dagger = \mathbf{A}^{*T}$ is the adjoint which is the transpose and complex conjugate of the matrix $\mathbf{A}$.

The definition of the adjoint is such that

$$\mathbf{y} \cdot \mathbf{Ax} = \mathbf{A}^\dagger \mathbf{y} \cdot \mathbf{x}\,. \tag{2.3.32}$$

Since $\mathbf{Ax} = \mathbf{b}$, the left side of the equation reduces to $\mathbf{y} \cdot \mathbf{b}$ while the right side reduces to $\mathbf{0}$ since $\mathbf{A}^\dagger \mathbf{y} = \mathbf{0}$. This then gives the condition

$$\mathbf{y} \cdot \mathbf{b} = \mathbf{0} \tag{2.3.33}$$

which is known as the *Fredholm-Alternative* theorem, or a *solvability* condition. In words, the equation states the in order for the system $\mathbf{Ax} = \mathbf{b}$ to be solvable, the right-hand side forcing $\mathbf{b}$ must be orthogonal to the null-space of the adjoint operator $\mathbf{A}^\dagger$.

3 Curve Fitting

Analyzing data is fundamental to any aspect of science. Often data can be noisy in nature and only the trends in the data are sought. A variety of curve fitting schemes can be generated to provide simplified descriptions of data and its behavior. The least-squares fit method is explored along with fitting methods of polynomial fits and splines.

3.1 Least-Square Fitting Methods

One of the fundamental tools for data analysis and recognizing trends in physical systems is curve fitting. The concept of curve fitting is fairly simple: use a simple function to describe a trend by minimizing the error between the selected function to fit and a set of data. The mathematical aspects of this are laid out in this section.

Suppose we are given a set of n data points

$$(x_1, y_1),\ (x_2, y_2),\ (x_3, y_3),\ \cdots\ , (x_n, y_n)\,. \tag{3.1.1}$$

Further, assume that we would like to fit a best fit line through these points. Then we can approximate the line by the function

$$f(x) = Ax + B \tag{3.1.2}$$

where the constants A and B are chosen to minimize some error. Thus the function gives an approximation to the true value which is off by some error so that

$$f(x_k) = y_k + E_k \tag{3.1.3}$$

where y_k is the true value and E_k is the error from the true value.

Various error measurements can be minimized when approximating with a given function $f(x)$. Three standard possibilities are given as follows

$$\text{I. MaximumError}: \quad E_\infty(f) = \max_{1<k<n} |f(x_k) - y_k| \tag{3.1.4a}$$

$$\text{II. AverageError}: \quad E_1(f) = \frac{1}{n}\sum_{k=1}^{n} |f(x_k) - y_k| \tag{3.1.4b}$$

$$\text{III. Root}-\text{meanSquare}: \quad E_2(f) = \left(\frac{1}{n}\sum_{k=1}^{n} |f(x_k) - y_k|^2\right)^{1/2}. \tag{3.1.4c}$$

In practice, the root-mean square error is most widely used and accepted. Thus when fitting a curve to a set of data, the root-mean square error is chosen to be minimized. This is called a *least-squares fit.*

Least-Squares Line

To apply the least-square fit criteria, consider the data points $\{x_k, y_k\}$. where $k = 1, 2, 3, \cdots, n$. To fit the curve

$$f(x) = Ax + B \tag{3.1.5}$$

to this data, the E_2 is found by minimizing the sum

$$E_2(f) = \sum_{k=1}^{n} |f(x_k) - y_k|^2 = \sum_{k=1}^{n} (Ax_k + B - y_k)^2\,. \tag{3.1.6}$$

Minimizing this sum requires differentiation. Specifically, the constants A and B are chosen so that a minimum occurs. thus we require: $\partial E_2/\partial A = 0$ and

$\partial E_2/\partial B = 0$. This gives:

$$\frac{\partial E_2}{\partial A} = 0: \quad \sum_{k=1}^{n} 2(Ax_k + B - y_k)x_k = 0 \tag{3.1.7a}$$

$$\frac{\partial E_2}{\partial B} = 0: \quad \sum_{k=1}^{n} 2(Ax_k + B - y_k) = 0\,. \tag{3.1.7b}$$

Upon rearranging, the 2×2 system of equations is found for A and B

$$\begin{pmatrix} \sum_{k=1}^{n} x_k^2 & \sum_{k=1}^{n} x_k \\ \sum_{k=1}^{n} x_k & n \end{pmatrix} \begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{n} x_k y_k \\ \sum_{k=1}^{n} y_k \end{pmatrix}. \tag{3.1.8}$$

This equation can be easily solved using the backslash command in MATLAB.

This method can be easily generalized to higher polynomial fits. In particular, a parabolic fit to a set of data requires the fitting function

$$f(x) = Ax^2 + Bx + C \tag{3.1.9}$$

where now the three constants A, B, and C must be found. These can be found from the 3×3 system which results from minimizing the error $E_2(A, B, C)$

$$\frac{\partial E_2}{\partial A} = 0 \tag{3.1.10a}$$

$$\frac{\partial E_2}{\partial B} = 0 \tag{3.1.10b}$$

$$\frac{\partial E_2}{\partial C} = 0 \tag{3.1.10c}$$

Data Linearization

Although a powerful method, the minimization of procedure can result in equations which are nontrivial to solve. Specifically, consider fitting data to the exponential function

$$f(x) = C\exp(Ax)\,. \tag{3.1.11}$$

The error to be minimized is

$$E_2(A, C) = \sum_{k=1}^{n} (C\exp(Ax_k) - y_k)^2\,. \tag{3.1.12}$$

Applying the minimizing conditions leads to

$$\frac{\partial E_2}{\partial A} = 0: \quad \sum_{k=1}^{n} 2(C\exp(Ax_k) - y_k)Cx_k\exp(Ax_k) = 0 \tag{3.1.13a}$$

$$\frac{\partial E_2}{\partial C} = 0: \quad \sum_{k=1}^{n} 2(C\exp(Ax_k) - y_k)\exp(Ax_k) = 0\,. \tag{3.1.13b}$$

This in turn leads to the 2×2 system

$$C\sum_{k=1}^{n} x_k \exp(2Ax_k) - \sum_{k=1}^{n} x_k y_k \exp(Ax_k) = 0 \tag{3.1.14a}$$

$$C\sum_{k=1}^{n} \exp(Ax_k) - \sum_{k=1}^{n} y_k \exp(Ax_k) = 0\,. \tag{3.1.14b}$$

This system of equations is nonlinear and cannot be solved in a straightforward fashion. Indeed, a solution may not even exist. Or many solution may exist.

To avoid the difficulty of solving this nonlinear system, the exponential fit can be *linearized* by the transformation

$$Y = \ln(y) \tag{3.1.15a}$$

$$X = x \tag{3.1.15b}$$

$$B = \ln C\,. \tag{3.1.15c}$$

Then the fit function

$$f(x) = y = C\exp(Ax) \tag{3.1.16}$$

can be linearized by taking the natural log of both sides so that

$$\ln y = \ln(C\exp(Ax)) = \ln C + \ln(\exp(Ax)) = B + Ax \rightarrow Y = AX + B\,. \tag{3.1.17}$$

So by fitting to the natural log of the y-data

$$(x_i, y_i) \rightarrow (x_i, \ln y_i) = (X_i, Y_i) \tag{3.1.18}$$

the curve fit for the exponential function becomes a linear fitting problem which is easily handled.

General Fitting

Given the preceding examples, a theory can be developed for a general fitting procedure. The key idea is to assume a form of the fitting function

$$f(x) = f(x, C_1, C_2, C_3, \cdots, C_M) \tag{3.1.19}$$

where the C_i are constants used to minimize the error and $M < n$. The root-mean square error is then

$$E_2(C_1, C_2, C_3, \cdots, C_m) = \sum_{k=1}^{n} (f(x_k, C_1, C_2, C_3, \cdots, C_M) - y_k)^2 \tag{3.1.20}$$

which can be minimized by considering the $M \times M$ system generated from

$$\frac{\partial E_2}{\partial C_j} = 0 \quad j = 1, 2, \cdots, M\,. \tag{3.1.21}$$

In general, this gives the *nonlinear* set of equations

$$\sum_{k=1}^{n}(f(x_k, C_1, C_2, C_3, \cdots, C_M) - y_k)\frac{\partial f}{\partial C_j} = 0 \quad j = 1, 2, 3, \cdots, M\,. \tag{3.1.22}$$

Solving this set of equations can be quite difficult. Most attempts at solving nonlinear systems are based upon iterative schemes which require good initial guesses to converge to the solution. Regardless, the general fitting procedure is straightforward and allows for the construction of a best fit curve to match the data.

3.2 Polynomial Fits and Splines

One of the primary reasons for generating data fits from polynomials, splines, or least-square methods is to *interpolate* or *extrapolate* data values. In practice, when considering only a finite number of data points

$$\begin{aligned}&(x_0, y_0)\\ &(x_1, y_1)\\ &\vdots\\ &(x_n, y_n)\,.\end{aligned}$$

then the value of the curve at points other than the x_i are unknown. *Interpolation* uses the data points to predict values of $y(x)$ at locations where $x \neq x_i$ and $x \in [x_0, x_n]$. *Extrapolation* is similar, but it predicts values of $y(x)$ for $x > x_n$ or $x < x_0$, i.e. outside the range of the data points.

Interpolation and extrapolation are easy to do given a least-squares fit. Once the fitting curve is found, it can be evaluated for any value of x, thus giving an interpolated or extrapolated value. Polynomial fitting is another method for getting these values. With polynomial fitting, a polynomial is chosen to go through all data points. For the $n+1$ data points given above, an n^{th} degree polynomial is chosen

$$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \tag{3.2.1}$$

where the coefficients a_j are chosen so that the polynomial passes through each data point. Thus we have the resulting system

$$\begin{aligned}(x_0, y_0): &\quad y_0 = a_n x_0^n + a_{n-1}x_0^{n-1} + \cdots + a_1 x_0 + a_0\\ (x_1, y_1): &\quad y_1 = a_n x_1^n + a_{n-1}x_1^{n-1} + \cdots + a_1 x_1 + a_0\\ &\qquad\vdots\\ (x_n, y_n): &\quad y_n = a_n x_n^n + a_{n-1}x_n^{n-1} + \cdots + a_1 x_n + a_0\,.\end{aligned}$$

This system of equations is nothing more than a linear system $\mathbf{Ax} = \mathbf{b}$ which can be solved by using the backslash command in MATLAB.

Although this polynomial fit method will generate a curve which passes through all the data, it is an expensive computation since we have to first set up the system and then perform an $O(n^3)$ operation to generate the coefficients a_j. A more direct method for generating the relevant polynomial is the *Lagrange Polynomials* method. Consider first the idea of constructing a line between the two points (x_0, y_0) and (x_1, y_1). The general form for a line is $y = mx + b$ which gives

$$y = y_0 + (y_1 - y_0)\frac{x - x_0}{x_1 - x_0}\,. \tag{3.2.2}$$

Although valid, it is hard to continue to generalize this technique to fitting higher order polynomials through a larger number of points. Lagrange developed a method which expresses the line through the two points as

$$p_1(x) = y_0\frac{x - x_1}{x_0 - x_1} + y_1\frac{x - x_0}{x_1 - x_0} \tag{3.2.3}$$

which can be easily verified to work. In a more compact and general way, this first degree polynomial can be expressed as

$$p_1(x) = \sum_{k=0}^{1} y_k L_{1,k}(x) = y_0 L_{1,0}(x) + y_1 L_{1,1}(x) \tag{3.2.4}$$

where the *Lagrange coefficients* are given by

$$L_{1,0}(x) = \frac{x - x_1}{x_0 - x_1} \tag{3.2.5a}$$

$$L_{1,1}(x) = \frac{x - x_0}{x_1 - x_0}\,. \tag{3.2.5b}$$

The power of this method is that it can be easily generalized to consider the $n+1$ points of our original data set. In particular, we fit an n^{th} degree polynomial through the given data set of the form

$$p_n(x) = \sum_{k=0}^{n} y_k L_{n,k}(x) \tag{3.2.6}$$

where the Lagrange coefficient is

$$L_{n,k}(x) = \frac{(x - x_0)(x - x_1)\cdots(x - x_{k-1})(x - x_{k+1})\cdots(x - x_n)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)}\,. \tag{3.2.7}$$

Thus there is no need to solve a linear system to generate the desired polynomial. This is the preferred method for generating a polynomial fit to a given set of data and is the core algorithm employed by most commercial packages such as MATLAB for polynomial fitting.

Splines

Although MATLAB makes it a trivial matter to fit polynomials or least-square fits through data, a fundamental problem can arise as a result of a polynomial fit: *polynomial wiggle*. Polynomial wiggle is generated by the fact that an n^{th} degree polynomial has, in general, $n-1$ turning points from up to down or vice-versa. One way to overcome this is to use a *piecewise polynomial interpolation* scheme. Essentially, this simply draws a line between neighboring data points and uses this line to give interpolated values. This technique is rather simple minded, but it does alleviate the problem generated by polynomial wiggle. However, the interpolating function is now only a piecewise function. Therefore, when considering interpolating values between the points (x_0, y_0) and (x_n, y_n), there will be n linear functions each valid only between two neighboring points.

The data generated by a piecewise linear fit can be rather crude and it tends to be choppy in appearance. *Splines* provide a better way to represent data by constructing cubic functions between points so that the first and second derivative are continuous at the data points. This gives a smooth looking function without polynomial wiggle problems. The basic assumption of the spline method is to construct a cubic function between data points:

$$S_k(x) = S_{k,0} + S_{k,1}(x - x_k) + S_{k,2}(x - x_k)^2 + S_{k,3}(x - x_k)^3 \tag{3.2.8}$$

where $x \in [x_k, x_{k+1}]$ and the coefficients $S_{k,j}$ are to be determined from various constraint conditions. Four constraint conditions are imposed

$$S_k(x_k) = y_k \tag{3.2.9a}$$
$$S_k(x_{k+1}) = S_{k+1}(x_{k+1}) \tag{3.2.9b}$$
$$S'_k(x_{k+1}) = S'_{k+1}(x_{k+1}) \tag{3.2.9c}$$
$$S''_k(x_{k+1}) = S''_{k+1}(x_{k+1})\,. \tag{3.2.9d}$$

This allows for a smooth fit to the data since the four constraints correspond to fitting the data, continuity of the function, continuity of the first derivative, and continuity of the second derivative respectively.

To solve for these quantities, a large system of equations $\mathbf{Ax} = \mathbf{b}$ is constructed. The number of equations and unknowns must be first calculated.

- $S_k(x_k) = y_k \rightarrow$ Solution fit: $n+1$ equations
- $S_k = S_{k+1} \rightarrow$ Continuity: $n-1$ equations
- $S'_k = S'_{k+1} \rightarrow$ Smoothness: $n-1$ equations
- $S''_k = S''_{k+1} \rightarrow$ Smoothness: $n-1$ equations

This gives a total of $4n-2$ equations. For each of the n intervals, there are 4 parameters which gives a total of $4n$ unknowns. Thus two extra constraint

conditions must be placed on the system to achieve a solution. There are a large variety of options for assumptions which can be made at the edges of the spline. It is usually a good idea to use the default unless the application involved requires a specific form. The spline problem is then reduced to a simple solution of an $\mathbf{Ax} = \mathbf{b}$ problem which can be solved with the backslash command.

As a final remark on splines, splines are heavily used in computer graphics and animation. The primary reason is for their relative ease in calculating, and for their smoothness properties. There is an entire spline toolbox available for MATLAB which attests to the importance of this technique for this application. Further, splines are also commonly used for smoothing data before differentiating. This will be considered further in upcoming lectures.

3.3 Data Fitting with MATLAB

This section will discuss the practical implementation of the curve fitting schemes presented in the preceding two sections. The schemes to be explored are least-square fits, polynomial fits, line interpolation, and spline interpolation. Additionally, a non-polynomial least-square fit will be considered which results in a nonlinear system of equations. This nonlinear system requires additional insight into the problem and sophistication in its solution technique.

To begin the data fit process, we first import a relevant data set into the MATLAB environment. To do so, the **load** command is used. The file *linefit.dat* is a collection of x and y data values put into a two column format separated by spaces. The file is the following:

linefit.dat

```
0.0  1.1
0.5  1.6
1.1  2.4
1.7  3.8
2.1  4.3
2.5  4.7
2.9  4.8
3.3  5.5
3.7  6.1
4.2  6.3
4.9  7.1
5.3  7.1
6.0  8.2
6.7  6.9
7.0  5.3
```

This data is just an example of you may want to import into MATLAB. The command structure to read this data is as follows

```
load linefit.dat
x=linefit(:,1);
y=linefit(:,2);
figure(1), plot(x,y,'O:')
```

After reading in the data, the two vectors **x** and **y** are created from the first and second column of the data respectively. It is this set of data which will be explored with line fitting techniques. The code will also generate a plot of the data in *figure 1* of MATLAB.

Least-Squares Fitting

The least-squares fit technique is considered first. The **polyfit** and **polyval** commands are essential to this method. Specifically, the **polyfit** command is used to generate the $n+1$ coefficients a_j of the n^{th}-degree polynomial

$$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \tag{3.3.1}$$

used for fitting the data. The basic structure requires that the vectors **x** and **y** be submitted to **polyval** along with the desired degree of polynomial fit n. To fit a line ($n = 1$) through the data, the structure, use the command

```
pcoeff=polyfit(x,y,1);
```

The output of this function call is a vector **pcoeff** which includes the coefficients a_1 and a_0 of the line fit $p_1(x) = a_1 x + a_0$. To evaluate and plot this line, values of x must be chosen. For this example, the line will be plotted for $x \in [0, 7]$ in steps of $\Delta x = 0.1$.

```
xp=0:0.1:7;
yp=polyval(pcoeff,xp);
figure(2), plot(x,y,'O',xp,yp,'m')
```

The **polyval** command uses the coefficients generated from **polyfit** to generate the $y-$values of the polynomial fit at the desired values of x given by **xp**. *Figure 2* in MATLAB depicts both the data and the best line fit in the least-square sense.

To fit a parabolic profile through the data, a second degree polynomial is used. This is generated with

```
pcoeff2=polyfit(x,y,2);
yp2=polyval(pcoeff2,xp);
figure(3), plot(x,y,'O',xp,yp2,'m')
```

Here the vector **yp2** contains the parabolic fit to the data evaluated at the x-values **xp**. These results are plotted in MATLAB *figure 3*. To find the least-square error, the sum of the squares of the differences between the parabolic fit and the actualy data must be evaluated. Specifically, the quantity

$$E_2(f) = \left(\frac{1}{n} \sum_{k=1}^{n} |f(x_k) - y_k|^2 \right)^{1/2} \tag{3.3.2}$$

is calculated. For the parabolic fit considered in the last example, the polynomial fit must be evaluated at the $x-$values for the given data **linefit.dat**. The error is then calculated

```
yp3=polyval(pcoeff2,x);
E2=sqrt( sum( ( abs(yp3-y) ).^2  )/n )
```

This is a quick and easy calculation which allows for an evaluation of the fitting procedure. In general, the error will continue to drop as the degree of polynomial is increased. this is because every extra degree of freedom allows for a better least-squares fit to the data.

Interpolation

In addition to least-square fitting, interpolation techniques can be developed which go through all the given data points. The error in this case is zero, but each interpolation scheme must be evaluated for its accurate representation of the data. The first interpolation scheme is a polynomial fit to the data. Given $n+1$ points, an n^{th} degree polynomial is chosen. The **polyfit** command is again used for the calculation

```
n=length(x)-1;
pcoeffn=polyfit(x,y,n);
ypn=polyval(pcoeffn,xp);
figure(4), plot(x,y,'O',xp,ypn,'m')
```

The MATLAB script will produce an n^{th} degree polynomial through the data points. But as always is the danger with polynomial interpolation, *polynomial wiggle* can dominate the behavior. This is indeed the case as illustrated in *figure 4*. The strong oscillatory phenomena at the edges is a common feature of this type of interpolation.

In contrast to a polynomial fit, a piecewise linear fit gives a simple minded connect-the-dot interpolation to the data. The **interp1** command gives the piecewise linear fit algorithm

```
yint=interp1(x,y,xp);
figure(5), plot(x,y,'O',xp,yint,'m')
```

The linear interpolation is illustrated in *figure 5* of MATLAB. The are a few options available with the **interp1** command, including the *nearest* option and the *spline* option. The *nearest* option gives the nearest value of the data to the interpolated value while the *spline* option gives a cubic spline fit interpolation to the data. The two options can be compared with the default *linear* option.

```
yint=interp1(x,y,xp);
yint2=interp1(x,y,xp,'nearest')
yint3=interp1(x,y,xp,'spline')
figure(5), plot(x,y,'O',xp,yint,'m',xp,int2,'k',xp,int3,'r')
```

Note that the spline option is equivalent to using the **spline** algorithm supplied by MATLAB. Thus a smooth fit can be achieved with either **spline** or **interp1**.

The **spline** command is used by giving the x and y data along with a vector **xp** for which we desire to generate corresponding $y-$values.

```
yspline=spline(x,y,xp);
figure(6), plot(x,y,'O',xp,yspline,'k')
```

The generated spline is depicted in *figure 6*. This is the same as that using **interp1** with the *spline* option. Note that the data is smooth as expected from the enforcement of continuous smooth derivatives.

Nonlinear Least-Square Fitting

To consider more sophisticated least-square fitting routines, consider the following data set which looks like it good be nicely fitted with a Gaussian profile.

gaussfit.dat

```
-3.0  -0.2
-2.2   0.1
-1.7   0.05
-1.5   0.2
-1.3   0.4
-1.0   1.0
-0.7   1.2
-0.4   1.4
-0.25  1.8
-0.05  2.2
 0.07  2.1
 0.15  1.6
 0.3   1.5
 0.65  1.1
 1.1   0.8
```

```
1.25  0.3
1.8   -0.1
2.5    0.2
```

To fit the data to a Gaussian, we indeed assume a Gaussian function of the form

$$f(x) = A\exp(-Bx^2)\,. \tag{3.3.3}$$

Following the procedure to minimize the least-square error leads to a set of nonlinear equations for the coefficients A and B. In general, solving a nonlinear set of equations can be a difficult task. A solution is not guaranteed to exist. And in fact, there may be many solutions to the problem. Thus the nonlinear problem should be handled with care.

To generate a solution, the least-square error must be minimized. In particular, the sum

$$E_2 = \sum_{k=0}^{n} |f(x_k) - y_k|^2 \tag{3.3.4}$$

must be minimized. The command **fmins** in MATLAB minimizes a function of several variables. In this case, we minimize (3.3.4) with respect to the variables A and B. Thus the minimum of

$$E_2 = \sum_{k=0}^{n} |A\exp(-Bx_k^2) - y_k|^2 \tag{3.3.5}$$

must be found with respect to A and B. The **fmins** algorithm requires a function call to the quantity to minimized along with an initial guess for the parameters which are being used for minimization, i.e. A and B.

```
coeff=fmins('gafit',[1 1]);
```

This command structure uses as initial guesses $A = 1$ and $B = 1$ when calling the file **gafit.m**. After minimizing, it returns the vector **coeff** which contains the appropriate values of A and B which minimize the least-square error. The function called is then constructed as follows:

gafit.m

```
function E=gafit(x0)

load gaussfit.dat
x=gaussfit(:,1);
y=gaussfit(:,2);
E=sum(  ( x0(1)*exp(-x0(2)*x.^2)-y ).^2  )
```

The vector $\mathbf{x_0}$ accepts the initial guesses for A and B and is updated as A and B are modified through an iteration procedure to converge to the least-square values. Note that the sum is simply a statement of the quantity to minimized, namely (3.3.5). The results of this calculation can be illustrated with MATLAB *figure 7*

```
xga=-3:0.1:3;
a=coeff(1); b=coeff(2)
yga=a*exp(-b*xga.^2);
figure(7), plot(x2,y2,'O',xga,yga,'m')
```

Note that for this case, the initial guess is extremely important. For any given problem where this technique is used, an educated guess for the values of parameters like A and B can determine if the technique will work at all. The results should also be checked carefully since there is no guarantee that a minimization near the desired fit can be found.

4 Numerical Differentiation and Integration

Differentiation and integration form the backbone of the mathematical techniques required to describe and analyze physical systems. These two mathematical concepts describe how certain quantities of interest change with respect to either space and time or both. Understanding how to evaluate these quantities numerically is essential to understanding systems beyond the scope of analytic methods.

4.1 Numerical Differentiation

Given a set of data or a function, it may be useful to differentiate the quantity considered in order to determine a physically relevant property. For instance, given a set of data which represents the position of a particle as a function of time, then the derivative and second derivative give the velocity and acceleration respectively. From calculus, the definition of the derivative is given by

$$\frac{df(t)}{dt} = \lim_{\Delta t \to 0} \frac{f(t+\Delta t) - f(t)}{\Delta t} \tag{4.1.1}$$

Since the derivative is the slope, the formula on the right is nothing more than a rise-over-run formula for the slope. The general idea of calculus is that as $\Delta t \to 0$, then the rise-over-run gives the instantaneous slope. Numerically, this means that if we take Δt sufficiently small, than the approximation should be fairly accurate. To quantify and control the error associated with approximating the derivative, we make use of *Taylor series* expansions.

To see how the Taylor expansions are useful, consider the following two Taylor series:

$$f(t+\Delta t) = f(t) + \Delta t\frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} + \frac{\Delta t^3}{3!}\frac{d^3f(c_1)}{dt^3} \quad (4.1.2a)$$

$$f(t-\Delta t) = f(t) - \Delta t\frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} - \frac{\Delta t^3}{3!}\frac{d^3f(c_2)}{dt^3} \quad (4.1.2b)$$

where $c_i \in [a,b]$. Subtracting these two expressions gives

$$f(t+\Delta t) - f(t-\Delta t) = 2\Delta t\frac{df(t)}{dt} + \frac{\Delta t^3}{3!}\left(\frac{d^3f(c_1)}{dt^3} + \frac{d^3f(c_2)}{dt^3}\right). \quad (4.1.3)$$

By using the mean-value theorem of calculus, we find $f'''(c) = (f'''(c_1) + f'''(c_2))/2$. Upon dividing the above expression by $2\Delta t$ and rearranging, we find the following expression for the first derivative:

$$\frac{df(t)}{dt} = \frac{f(t+\Delta t) - f(t-\Delta t)}{2\Delta t} - \frac{\Delta t^2}{6}\frac{d^3f(c)}{dt^3} \quad (4.1.4)$$

where the last term is the truncation error associated with the approximation of the first derivative using this particular Taylor series generated expression. Note that the truncation error in this case is $O(\Delta t^2)$.

We could improve on this by continuing our Taylor expansion and truncating it at higher orders in Δt. This would lead to higher accuracy schemes. Specifically, by truncating at $O(\Delta t^5)$, we would have

$$f(t+\Delta t) = f(t) + \Delta t\frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} + \frac{\Delta t^3}{3!}\frac{d^3f(t)}{dt^3} + \frac{\Delta t^4}{4!}\frac{d^4f(t)}{dt^4} + \frac{\Delta t^5}{5!}\frac{d^5f(c_1)}{dt^5} \quad (4.1.5a)$$

$$f(t-\Delta t) = f(t) - \Delta t\frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} - \frac{\Delta t^3}{3!}\frac{d^3f(t)}{dt^3} + \frac{\Delta t^4}{4!}\frac{d^4f(t)}{dt^4} - \frac{\Delta t^5}{5!}\frac{d^5f(c_2)}{dt^5} \quad (4.1.5b)$$

where $c_i \in [a,b]$. Again subtracting these two expressions gives

$$f(t+\Delta t) - f(t-\Delta t) = 2\Delta t\frac{df(t)}{dt} + \frac{2\Delta t^3}{3!}\frac{d^3f(t)}{dt^3} + \frac{\Delta t^5}{5!}\left(\frac{d^5f(c1)}{dt^5} + \frac{d^5f(c2)}{dt^5}\right). \quad (4.1.6)$$

In this approximation, there is third derivative term left over which needs to be removed. By using two additional points to approximate the derivative, this term can be removed. Thus we use the two additional points $f(t+2\Delta t)$ and $f(t-2\Delta t)$. Upon replacing Δt by $2\Delta t$ in (4.1.6), we find

$$f(t+2\Delta t) - f(t-2\Delta t) = 4\Delta t\frac{df(t)}{dt} + \frac{16\Delta t^3}{3!}\frac{d^3f(t)}{dt^3} + \frac{32\Delta t^5}{5!}\left(\frac{d^5f(c3)}{dt^5} + \frac{d^5f(c4)}{dt^5}\right). \quad (4.1.7)$$

$O(\Delta t^2)$ center-difference schemes
$f'(t) = [f(t+\Delta t) - f(t-\Delta t)]/2\Delta t$
$f''(t) = [f(t+\Delta t) - 2f(t) + f(t-\Delta t)]/\Delta t^2$
$f'''(t) = [f(t+2\Delta t) - 2f(t+\Delta t) + 2f(t-\Delta t) - f(t-2\Delta t)]/2\Delta t^3$
$f''''(t) = [f(t+2\Delta t) - 4f(t+\Delta t) + 6f(t) - 4f(t-\Delta t) + f(t-2\Delta t)]/\Delta t^4$

Table 4: Second-order accurate center-difference formulas.

By multiplying (4.1.6) by eight and subtracting (4.1.7) and using the mean-value theorem on the truncation terms twice, we find the expression:

$$\frac{df(t)}{dt} = \frac{-f(t+2\Delta t) + 8f(t+\Delta t) - 8f(t-\Delta t) + f(t-2\Delta t)}{12\Delta t} + \frac{\Delta t^4}{30} f^{(5)}(c) \tag{4.1.8}$$

where $f^{(5)}$ is the fifth derivative and the truncation is of $O(\Delta t^4)$.

Approximating higher derivatives works in a similar fashion. By starting with the pair of equations (4.1.2) and adding, this gives the result

$$f(t+\Delta t)+f(t-\Delta t) = 2f(t)+\Delta t^2 \frac{d^2 f(t)}{dt^2} + \frac{\Delta t^4}{4!}\left(\frac{d^4 f(c_1)}{dt^4} + \frac{d^4 f(c_2)}{dt^4}\right). \tag{4.1.9}$$

By rearranging and solving for the second derivative, the $O(\Delta t^2)$ accurate expression is derived

$$\frac{d^2 f(t)}{dt^2} = \frac{f(t+\Delta t) - 2f(t) + f(t-\Delta t)}{\Delta t^2} + O(\Delta t^2) \tag{4.1.10}$$

where the truncation error is of $O(\Delta t^2)$ and is found again by the mean-value theorem to be $-(\Delta t^2/12) f''''(c)$. This process can be continued to find any arbitrary derivative. Thus, we could also approximate the third, fourth, and higher derivatives using this technique. It is also possible to generate backward and forward difference schemes by using points only behind or in front of the current point respectively. Tables 4-6 summarize the second-order and fourth-order central difference schemes along with the forward- and backward-difference formulas which are accurate to second-order.

A final remark is in order concerning these differentiation schemes. The central difference schemes are an excellent method for generating the values of the derivative in the interior points of a data set. However, at the end points, forward and backward difference methods must be used since they do not have

$O(\Delta t^4)$ center-difference schemes
$f'(t) = [-f(t+2\Delta t) + 8f(t+\Delta t) - 8f(t-\Delta t) + f(t-2\Delta t)]/12\Delta t$
$f''(t) = [-f(t+2\Delta t) + 16f(t+\Delta t) - 30f(t) + 16f(t-\Delta t) - f(t-2\Delta t)]/12\Delta t^2$
$f'''(t) = [-f(t+3\Delta t) + 8f(t+2\Delta t) - 13f(t+\Delta t) + 13f(t-\Delta t) - 8f(t-2\Delta t) + f(t-3\Delta t)]/8\Delta t^3$
$f''''(t) = [-f(t+3\Delta t) + 12f(t+2\Delta t) - 39f(t+\Delta t) + 56f(t) - 39f(t-\Delta t) + 12f(t-2\Delta t) - f(t-3\Delta t)]/6\Delta t^4$

Table 5: Fourth-order accurate center-difference formulas.

$O(\Delta t^2)$ forward- and backward-difference schemes
$f'(t) = [-3f(t) + 4f(t+\Delta t) - f(t+2\Delta t)]/2\Delta t$
$f'(t) = [3f(t) - 4f(t-\Delta t) + f(t-2\Delta t)]/2\Delta t$
$f''(t) = [2f(t) - 5f(t+\Delta t) + 4f(t+2\Delta t) - f(t+3\Delta t)]/\Delta t^3$
$f''(t) = [2f(t) - 5f(t-\Delta t) + 4f(t-2\Delta t) - f(t-3\Delta t)]/\Delta t^3$

Table 6: Second-order accurate forward- and backward-difference formulas.

neighboring points to the left and right respectively. Thus special care must be taken at the end points of any computational domain.

It may be tempting to deduce from the difference formulas that as $\Delta t \to 0$, the accuracy only improves in these computational methods. However, this line of reasoning completely neglects the second source of error in evaluating derivatives: numerical round-off.

Round-off and optimal step-size

An unavoidable consequence of working with numerical computations is round-off error. When working with most computations, *double precision* numbers are used. This allows for 16-digit accuracy in the representation of a given number. This round-off has significant impact upon numerical computations

and the issue of time-stepping.

As an example of the impact of round-off, we consider the approximation to the derivative

$$\frac{dy}{dt} \approx \frac{y(t+\Delta t) - y(t)}{\Delta t} + \epsilon(y(t), \Delta t) \tag{4.1.11}$$

where $\epsilon(y(t), \Delta t)$ measures the truncation error. Upon evaluating this expression in the computer, round-off error occurs so that

$$y(t) = Y(t) + e(t)\,, \tag{4.1.12}$$

where $Y(t)$ is the approximated value given by the computer and $e(t)$ measures the error from the true value $y(t)$. Thus the combined error between the round-off and truncation gives the following expression for the derivative:

$$\frac{dy}{dt} = \frac{y(t+\Delta t) - y(t)}{\Delta t} + E(y(t), \Delta t) \tag{4.1.13}$$

where the total error, E, is the combination of round-off and truncation such that

$$E = E_{\text{round}} + E_{\text{trunc}} = \frac{e(t+\Delta t) - e(t)}{\Delta t} - \frac{\Delta t^2}{2}\frac{d^2y(c)}{dt^2}\,. \tag{4.1.14}$$

We now determine the maximum size of the error. In particular, we can bound the maximum value of round-off and the derivate to be

$$|e(t+\Delta t)| \le e_r \tag{4.1.15a}$$

$$|-e(t)| \le e_r \tag{4.1.15b}$$

$$M = \max_{c\in[t_n, t_{n+1}]} \left\{ \left| \frac{d^2y(c)}{dt^2} \right| \right\}\,. \tag{4.1.15c}$$

This then gives the maximum error to be

$$|E| \le \frac{e_r + e_r}{\Delta t} + \frac{\Delta t^2}{2}M = \frac{2e_r}{\Delta t} + \frac{\Delta t^2 M}{2}\,. \tag{4.1.16}$$

Note that as Δt gets large, the error grows quadratically due to the truncation error. However, as Δt decreases to zero, the error is dominated by round-off which grows like $1/\Delta t$.

To minimize the error, we require that $\partial|E|/\partial(\Delta t) = 0$. Calculating this derivative gives

$$\frac{\partial|E|}{\partial(\Delta t)} = -\frac{2e_r}{\Delta t^2} + M\Delta t = 0\,, \tag{4.1.17}$$

so that

$$\Delta t = \left(\frac{2e_r}{M}\right)^{1/3}\,. \tag{4.1.18}$$

This gives the step size resulting in a minimum error. Thus the smallest step-size is not necessarily the most accurate. Rather, a balance between round-off error and truncation error is achieved to obtain the optimal step-size. For $e_r \approx 10^{-16}$, the optimal $\Delta t \approx 10^{-5}$. Below this value of Δt, numerical round-off begins to dominate the error

A similar procedure can be carried out for evaluating the optimal step size associated with the $O(\Delta t^4)$ accurate scheme for the first derivative. In this case

$$\frac{dy}{dt} = \frac{-f(t+2\Delta t) + 8f(t+\Delta t) - 8f(t-\Delta t) + f(t-2\Delta t)}{12\Delta t} + E(y(t), \Delta t) \tag{4.1.19}$$

where the total error, E, is the combination of round-off and truncation such that

$$E = \frac{-e(t+2\Delta t) + 8e(t+\Delta t) - 8e(t-\Delta t) + e(t-2\Delta t)}{12\Delta t} + \frac{\Delta t^4}{30}\frac{d^5 y(c)}{dt^5}\,. \tag{4.1.20}$$

We now determine the maximum size of the error. In particular, we can bound the maximum value of round-off to e as before and set $M = \max\{|y''''(c)|\}$. This then gives the maximum error to be

$$|E| = \frac{3e_r}{2\Delta t} + \frac{\Delta t^4 M}{30}\,. \tag{4.1.21}$$

Note that as Δt gets large, the error grows like a quartic due to the truncation error. However, as Δt decreases to zero, the error is again dominated by round-off which grows like $1/\Delta t$.

To minimize the error, we require that $\partial|E|/\partial(\Delta t) = 0$. Calculating this derivative gives

$$\Delta t = \left(\frac{45e_r}{4M}\right)^{1/5}\,. \tag{4.1.22}$$

Thus in this case, the optimal step $\Delta t \approx 10^{-3}$. This shows that the error can be quickly dominated by numerical round-off if one is not careful to take this significant effect into account.

4.2 Numerical Integration

Numerical integration simply calculates the area under a given curve. The basic ideas for performing such an operation come from the definition of integration

$$\int_a^b f(x)dx = \lim_{h\to 0}\sum_{j=0}^{N} f(x_j)h \tag{4.2.1}$$

where $b-a = Nh$. Thus the area under the curve, from the calculus standpoint, is thought of as a limiting process of summing up an ever-increasing number

of rectangles. This process is known as numerical quadrature. Specifically, any sum can be represented as follows

$$Q[f] = \sum_{j=0}^{N} w_k f(x_k) = w_0 f(x_0) + w_1 f(x_1) + \cdots + w_N f(x_N) \tag{4.2.2}$$

where $a = x_0 < x_1 < x_2 < \cdots < x_N = b$. Thus the integral is evaluated as

$$\int_a^b f(x)dx = Q[f] + E[f] \tag{4.2.3}$$

where the term $E[f]$ is the error in approximating the integral by the quadrature sum (4.2.2). Typically, the error $E[f]$ is due to truncation error. To integrate, use will be made of polynomial fits to the y-values $f(x_j)$. Thus we assume the function $f(x)$ can be approximated by a polynomial

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \tag{4.2.4}$$

where the truncation error in this case is proportional to the $(n+1)^{th}$ derivative $E[f] = Af^{(n+1)}(c)$ and A is a constant. This process of polynomial fitting the data gives the *Newton-Cotes Formulas*.

Newton-Cotes Formulas

The following integration approximations result from using a polynomial fit through the data to be differentiated. It is assumed that

$$x_k = x_0 + hk \qquad\qquad f_k = f(x_k)\,. \tag{4.2.5}$$

This gives the following integration algorithms:

$$\text{Trapezoid Rule } \int_{x_0}^{x_1} f(x)dx = \frac{h}{2}(f_0 + f_1) - \frac{h^3}{12}f''(c) \tag{4.2.6a}$$

$$\text{Simpson's Rule } \int_{x_0}^{x_2} f(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2) - \frac{h^5}{90}f''''(c) \tag{4.2.6b}$$

$$\text{Simpson's 3/8 Rule } \int_{x_0}^{x_3} f(x)dx = \frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3) - \frac{3h^5}{80}f''''(c) \tag{4.2.6c}$$

$$\text{Boole's Rule } \int_{x_0}^{x_4} f(x)dx = \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4) - \frac{8h^7}{945}f^{(6)}(c) \tag{4.2.6d}$$

These algorithms have varying degrees of accuracy. Specifically, they are $O(h^2)$, $O(h^4)$, $O(h^4)$ and $O(h^6)$ accurate schemes respectively. The accuracy condition is determined from the truncation terms of the polynomial fit. Note that the *Trapezoid rule* uses a sum of simple trapezoids to approximate the integral.

Simpson's rule fits a quadratic curve through three points and calculates the area under the quadratic curve. *Simpson's 3/8 rule* uses four points and a cubic polynomial to evaluate the area, while *Boole's rule* uses five points and a quintic polynomial fit to generate an evaluation of the integral.

The derivation of these integration rules follows from simple polynomial fits through a specified number of data points. To derive the Simpson's rule, consider a second degree polynomial through the three points (x_0, f_0), (x_1, f_2), and (x_2, f_2):

$$p_2(x) = f_0\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f_1\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f_2\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}\,. \tag{4.2.7}$$

This quadratic fit is derived by using Lagrange coefficients. The truncation error could also be included, but we neglect it for the present purposes. By plugging in (4.2.7) into the integral

$$\int_{x_0}^{x_2} f(x)dx \approx \int_{x_0}^{x_2} p_2(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2)\,. \tag{4.2.8}$$

The integral calculation is easily performed since it only involves integrating powers of x^2 or less. Evaluating at the limits then causes many terms to cancel and drop out. Thus the Simpson's rule is recovered. The trapezoid rule, Simpson's 3/8 rule, and Boole's rule are all derived in a similar fashion. To make connection with the quadrature rule (4.2.2), $Q = w_0f_0 + w_1f_1 + w_2f_2$, Simpson's rule gives $w_0 = h/3$, $w_1 = 4h/3$, and $w_2 = h/3$ as weighting factors.

Composite Rules

The integration methods (4.2.6) give values for the integrals over only a small part of the integration domain. Trapezoid rule, for instance, only gives a value for $x \in [x_0, x_1]$. However, our fundamental aim is to evaluate the integral over the entire domain $x \in [a, b]$. Assuming once again that our interval is divided as $a = x_0 < x_1 < x_2 < \cdots < x_N = b$, then the trapezoid rule applied over the interval gives the total integral

$$\int_a^b f(x)dx \approx Q[f] = \sum_{j=1}^{N-1} \frac{h}{2}\left(f_j + f_{j+1}\right)\,. \tag{4.2.9}$$

Writing out this sum gives

$$\begin{aligned}\sum_{j=1}^{N-1} \frac{h}{2}\left(f_j + f_{j+1}\right) &= \frac{h}{2}(f_0+f_1) + \frac{h}{2}(f_1+f_2) + \cdots + \frac{h}{2}(f_{N-1}+f_N) \\ &= \frac{h}{2}(f_0 + 2f_1 + 2f_2 + \cdots + 2f_{N-1} + f_N) \end{aligned} \tag{4.2.10}$$

$$= \frac{h}{2}\left(f_0 + f_N + 2\sum_{j=1}^{N-1} f_j\right)$$

The final expression no longer double counts the values of the points between f_0 and f_N. Instead, the final sum only counts the intermediate values once, thus making the algorithm about twice as fast as the previous sum expression. These are computational savings which should always be exploited if possible.

Recursive Improvement of Accuracy

Given an integration procedure and a value of h, a function or data set can be integrated to a prescribed accuracy. However, it may be desirable to improve the accuracy without having to disregard previous approximations to the integral. To see how this might work, consider the trapezoidal rule for a step size of $2h$. Thus, the even data points are the only ones of interest and we have the basic one-step integral

$$\int_{x_0}^{x_2} f(x)dx = \frac{2h}{2}(f_0 + f_2) = h(f_0 + f_2)\,. \tag{4.2.11}$$

The composite rule associated with this is then

$$\int_a^b f(x)dx \approx Q[f] = \sum_{j=0}^{N/2-1} h\left(f_{2j} + f_{2j+2}\right)\,. \tag{4.2.12}$$

Writing out this sum gives

$$\begin{aligned}\sum_{j=0}^{N/2-1} h\left(f_{2j} + f_{2j+2}\right) &= h(f_0 + f_2) + h(f_2 + f_4) + \cdots + h(f_{N-2} + f_N) \\ &= h(f_0 + 2f_2 + 2f_4 + \cdots + 2f_{N-2} + f_N) \qquad (4.2.13)\\ &= h\left(f_0 + f_N + 2\sum_{j=1}^{N/2-1} f_{2j}\right)\,.\end{aligned}$$

Comparing the middle expressions in (4.2.10) and (4.2.13) gives a great deal of insight into how recursive schemes for improving accuracy work. Specifically, we note that the more accurate scheme with step size h contains all the terms in the integral approximation using step size $2h$. Quantifying this gives

$$Q_h = \frac{1}{2}Q_{2h} + h(f_1 + f_3 + \cdots + f_{N-1})\,, \tag{4.2.14}$$

where Q_h and Q_{2h} are the quadrature approximations to the integral with step size h and $2h$ respectively. This then allows us to half the value of h and

improve accuracy without jettisoning the work required to approximate the solution with the accuracy given by a step size of $2h$. This recursive procedure can be continued so as to give higher accuracy results. Further, this type of procedure holds for Simpson's rule as well as any of the integration schemes developed here. This recursive routine is used in MATLAB in order to generate results to a prescribed accuracy.

4.3 Implementation of Differentiation and Integration

This lecture focuses on the implementation of differentiation and integration methods. Since they form the backbone of calculus, accurate methods to approximate these calculations are essential. To begin, we consider a specific function to be differentiated, namely a hyperbolic secant. Part of the reason for considering this function is that the exact value of its derivative is known. This allows us to make a comparison of our approximate methods with the actual solution. Thus, we consider

$$u = \text{sech}(x) \tag{4.3.1}$$

whose derivative is

$$\frac{du}{dx} = -\text{sech}(x)\tanh(x)\,. \tag{4.3.2}$$

Differentiation

To begin the calculations, we define a spatial interval. For this example we take the interval $x \in [-10, 10]$. In MATLAB, the spatial discritization, Δx, must also be defined. This gives

```
dx=0.1;        % spatial discritization
x=-10:dx:10;   % spatial domain
```

Once the spatial domain has been defined, the function to be differentiated must be evaluated

```
u=sech(x);
ux_exact=-sech(x)*tanh(x);
figure(1), plot(x,u,x,ux_exact)
```

MATLAB *figure 1* produces the function and its derivative.

To calculate the derivative numerically, we use the center-, forward-, and backward-difference formulas derived for differentiation. Specifically, we will make use of the following four first-derivative approximations:

$$\text{center} - \text{difference } O(h^2) : \frac{y_{n+1} - y_{n-1}}{2h} \tag{4.3.3a}$$

$$\text{center} - \text{difference } O(h^4) : \frac{-y_{n+2} + 8y_{n+1} - 8y_{n-1} + y_{n-2}}{12h} \tag{4.3.3b}$$

$$\text{forward} - \text{difference } O(h^2) : \frac{-3y_n + 4y_{n+1} - y_{n+2}}{2h} \tag{4.3.3c}$$

$$\text{backward} - \text{difference } O(h^2) : \frac{3y_n - 4y_{n-1} + y_{n-2}}{2h} . \tag{4.3.3d}$$

Here we have used $h = \Delta x$ and $y_n = y(x_n)$.

Second-order accurate derivative

To calculate the second order accurate derivative, we use the first, third and fourth equations of (4.3.3). In the interior of the domain, we use the center-difference scheme. However, at the left and right boundaries, there are no left and right neighboring points respectively to calculate the derivative with. Thus we require the use of forward- and backward-difference schemes respectively. This gives the basic algorithm

```
n=length(x)
% 2nd order accurate
ux(1)=(-3*u(1)+4*u(2)-u(3))/(2*dx);
for j=2:n-1
    ux(j)=(u(j+1)-u(j-1))/(2*dx);
end
ux(n)=(3*u(n)-4*u(n-1)+u(n-2))/(2*dx);
```

The values of $ux(1)$ and $ux(n)$ are evaluated with the forward- and backward-difference schemes.

Fourth-order accurate derivative

A higher degree of accuracy can be achieved by using a fourth-order scheme. A fourth-order center-difference scheme such as the second equation of (4.3.3) relied on two neighboring points. Thus the first two and last two points of the computational domain must be handled separately. In what follows, we use a second-order scheme at the boundary points, and a fourth-order scheme for the interior. This gives the algorithm

```
% 4th order accurate
ux2(1)=(-3*u(1)+4*u(2)-u(3))/(2*dx);
ux2(2)=(-3*u(2)+4*u(3)-u(4))/(2*dx);
for j=3:n-2
  ux2(j)=(-u(j+2)+8*u(j+1)-8*u(j-1)+u(j-2))/(12*dx);
end
ux2(n-1)=(3*u(n-1)-4*u(n-2)+u(n-3))/(2*dx);
ux2(n)=(3*u(n)-4*u(n-1)+u(n-2))/(2*dx);
```

For the $dx = 0.1$ considered here, the second-order and fourth-order schemes result in errors of 10^{-2} and 10^{-4} respectively. To view the accuracy, the results are plotted together with the analytic solution for the derivative

```
figure(2), plot(x,ux_exact,'o',x,ux,'c',x,ux2,'m')
```

To the naked eye, these results all look to be exactly on the exact solution. However, by zooming in on a particular point, it quickly becomes clear that the errors for the two schemes are indeed 10^{-2} and 10^{-4} respectively. The failure of accuracy can also be observed by taking large values of dx. For instance, $dx = 0.5$ and $dx = 1$ illustrate how the derivatives fail to be properly determined with large dx values.

As a final note, if you are given a set of data to differentiate. It is always recommended that you first run a spline through the data with an appropriately small dx and then differentiate the spline. This will help to give smooth differentiated data. Otherwise, the data will tend to be highly inaccurate and choppy.

Integration

There are a large number of integration routines built into MATLAB. So unlike the differentiation routines presented here, we will simply make use of the built in MATLAB functions. The most straight-forward integration routine is the trapezoidal rule. Given a set of data, the **trapz** command can be used to implement this integration rule. For the x and y data defined previously for the hyperbolic secant, the command structure gives

```
int_sech=trapz(x,y.^2);
```

where we have integrated $\text{sech}^2(x)$. The value of this integral is exactly two. The **trapz** command gives a value that is within 10^{-7} of the true value. To generate the cumulative values, the **cumtrapz** command is used.

```
int_sech2=cumtrapz(x,y.^2);
figure(3), plot(x,int_sech2)
```

MATLAB *figure 3* gives the value of the integral as a function of x.

Alternatively, a function can be specified for integration over a given domain. The **quad** command is implemented by specifying a function and the range of integration. This will give a value of the integral using a recursive Simpson's rule that is accurate to within 10^{-6}. The command structure for this evaluation is as follows

```
int_quad=quad(inline('sech(x).^2'),-10,10)
```

Here the **inline** command allows us to circumvent a traditional function call. The **quad** command can, however, be used with a function call. This command executes the integration of $\text{sech}^2(x)$ over $x \in [-10, 10]$.

Double and triple integrals over two-dimensional rectangles and three-dimensional boxes can be performed with the **dbequad** and **triplequad** commands. Note that no version of the **quad** command exists that produces cumulative integration values. However, this can be easily handled in a **for** loop.

5 Differential Equations

Our ultimate goal is to solve very general nonlinear partial differential equations of elliptic, hyperbolic, parabolic, or mixed type. However, a variety of basic techniques are required from the solutions of ordinary differential equations. By understanding the basic ideas for computationally solving initial and boundary value problems for differential equations, we can solve more complicated partial differential equations. The development of numerical solution techniques for initial and boundary value problems originates from the simple concept of the Taylor expansion. Thus the building blocks for scientific computing are rooted in concepts from freshman calculus. Implementation, however, often requires ingenuity, insight, and clever application of the basic principles. In some sense, our numerical solution techniques reverse our understanding of calculus. Whereas calculus teaches us to take a limit in order to define a derivative or integral, in numerical computations we take the derivative or integral of the governing equation and go backwards to define it as the difference.

5.1 Initial value problems: Euler, Runge-Kutta and Adams methods

The solutions of general partial differential equations rely heavily on the techniques developed for ordinary differential equations. Thus we begin by considering systems of differential equations of the form

$$\frac{d\mathbf{y}}{dt} = f(\mathbf{y}, t) \tag{5.1.1}$$

where $\mathbf{y}$ represents a vector and the initial conditions are given by

$$\mathbf{y}(0) = \mathbf{y}_0 \tag{5.1.2}$$

with $t \in [0, T]$. Although very simple in appearance, this equation cannot be solved analytically in general. Of course, there are certain cases for which the problem can be solved analytically, but it will generally be important to rely on numerical solutions for insight. For an overview of analytic techniques, see Boyce and DiPrima [1].

The simplest algorithm for solving this system of differential equations is known as the *Euler method*. The Euler method is derived by making use of the definition of the derivative:

$$\frac{d\mathbf{y}}{dt} = \lim_{\Delta t \to 0} \frac{\Delta \mathbf{y}}{\Delta t}\,. \tag{5.1.3}$$

Thus over a time span $\Delta t = t_{n+1} - t_n$ we can approximate the original differential equation by

$$\frac{d\mathbf{y}}{dt} = f(\mathbf{y}, t) \quad \Rightarrow \quad \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} \approx f(\mathbf{y}_n, t_n)\,. \tag{5.1.4}$$

The approximation can easily be rearranged to give

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(\mathbf{y}_n, t_n)\,. \tag{5.1.5}$$

Thus the Euler method gives an iterative scheme by which the future values of the solution can be determined. Generally, the algorithm structure is of the form

$$\mathbf{y}(t_{n+1}) = F(\mathbf{y}(t_n)) \tag{5.1.6}$$

where $F(\mathbf{y}(t_n)) = \mathbf{y}(t_n) + \Delta t \cdot f(\mathbf{y}(t_n), t_n)$ The graphical representation of this iterative process is illustrated in Fig. 1 where the slope (derivative) of the function is responsible for generating each subsequent approximation to the solution $\mathbf{y}(t)$. Note that the Euler method is exact as the step size descreases to zero: $\Delta t \to 0$.

The Euler method can be generalized to the following iterative scheme:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot \phi\,. \tag{5.1.7}$$

where the function ϕ is chosen to reduce the error over a single time step Δt and $\mathbf{y}_n = \mathbf{y}(t_n)$. The function ϕ is no longer constrained, as in the Euler scheme, to make use of the derivative at the left end point of the computational step. Rather, the derivative at the mid-point of the time-step and at the right end of the time-step may also be used to possibly improve accuracy. In particular, by generalizing to include the slope at the left and right ends of the time-step Δt, we can generate an iteration scheme of the following form:

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \Delta t\,[Af(t, \mathbf{y}(t)) + Bf(t + P \cdot \Delta t, \mathbf{y}(t) + Q\Delta t \cdot f(t, \mathbf{y}(t)))] \tag{5.1.8}$$

where A, B, P and Q are arbitrary constants. Upon Taylor expanding the last term, we find

$$\begin{aligned} &f(t + P \cdot \Delta t, \mathbf{y}(t) + Q\Delta t \cdot f(t, \mathbf{y}(t))) = \\ &\quad f(t, \mathbf{y}(t)) + P\Delta t \cdot f_t(t, \mathbf{y}(t)) + Q\Delta t \cdot f_{\mathbf{y}}(t, \mathbf{y}(t)) \cdot f(t, \mathbf{y}(t)) + O(\Delta t^2) \end{aligned} \tag{5.1.9}$$

where f_t and $f_{\mathbf{y}}$ denote differentiation with respect to t and $\mathbf{y}$ respectively, use has been made of (5.1.1), and $O(\Delta t^2)$ denotes all terms that are of size Δt^2 and

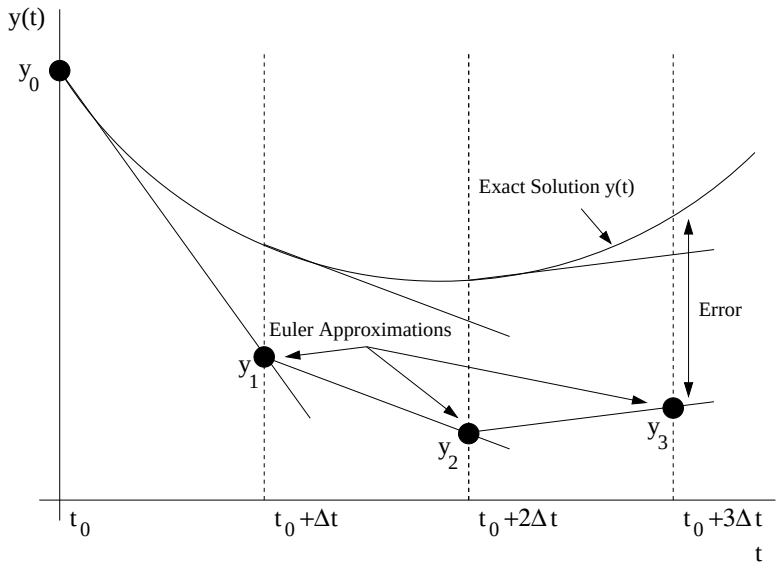


Figure 1: Graphical description of the iteration process used in the Euler method. Note that each subsequent approximation is generated from the slope of the previous point. This graphical illustration suggests that smaller steps Δt should be more accurate.

smaller. Plugging in this last result into the original iteration scheme (5.1.8) results in the following:

$$\begin{aligned}\mathbf{y}(t+\Delta t) &= \mathbf{y}(t) + \Delta t(A+B)f(t,\mathbf{y}(t)) \\ &+PB\Delta t^2 \cdot f_t(t,\mathbf{y}(t)) + BQ\Delta t^2 \cdot f_{\mathbf{y}}(t,\mathbf{y}(t)) \cdot f(t,\mathbf{y}(t)) + O(\Delta t^3) \quad (5.1.10)\end{aligned}$$

which is valid up to $O(\Delta t^2)$.

To proceed further, we simply note that the Taylor expansion for $\mathbf{y}(t+\Delta t)$ gives:

$$\begin{aligned}\mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \Delta t \cdot f(t,\mathbf{y}(t)) + \frac{1}{2}\Delta t^2 \cdot f_t(t,\mathbf{y}(t)) \\ +\frac{1}{2}\Delta t^2 \cdot f_{\mathbf{y}}(t,\mathbf{y}(t))f(t,\mathbf{y}(t)) + O(\Delta t^3)\,. \quad (5.1.11)\end{aligned}$$

Comparing this Taylor expansion with (5.1.10) gives the following relations:

$$A + B = 1 \quad (5.1.12a)$$

$$PB = \frac{1}{2} \quad (5.1.12b)$$

$$BQ = \frac{1}{2} \quad (5.1.12c)$$

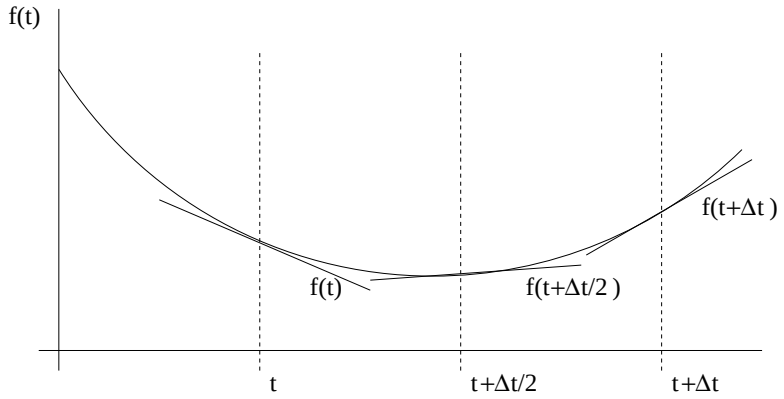


Figure 2: Graphical description of the initial, intermediate, and final slopes used in the 4th order Runke-Kutta iteration scheme over a time Δt.

which yields three equations for the four unknows A, B, P and Q. Thus one degree of freedom is granted, and a wide variety of schemes can be implemented. Two of the more commonly used schemes are known as *Heun's method* and *Modified Euler-Cauchy* (second order Runge-Kutta). These schemes assume $A = 1/2$ and $A = 0$ respectively, and are given by:

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \frac{\Delta t}{2}\left[f(t,\mathbf{y}(t)) + f(t+\Delta t, \mathbf{y}(t) + \Delta t \cdot f(t,\mathbf{y}(t)))\right] \quad (5.1.13a)$$

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \Delta t \cdot f\left(t + \frac{\Delta t}{2}, \mathbf{y}(t) + \frac{\Delta t}{2}\cdot f(t,\mathbf{y}(t))\right). \quad (5.1.13b)$$

Generally speaking, these methods for iterating forward in time given a single initial point are known as *Runge-Kutta methods*. By generalizing the assumption (5.1.8), we can construct stepping schemes which have arbitrary accuracy. Of course, the level of algebraic difficulty in deriving these higher accuracy schemes also increases significantly from Heun's method and Modified Euler-Cauchy.

4th-order Runge-Kutta

Perhaps the most popular general stepping scheme used in practice is known as the *4th order Runge-Kutta method*. The term "4th order" refers to the fact that the Taylor series local truncation error is pushed to $O(\Delta t^5)$. The total cumulative (global) error is then $O(\Delta t^4)$ and is responsible for the scheme name of "4th order". The scheme is as follows:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{6}\left[f_1 + 2f_2 + 2f_3 + f_4\right] \quad (5.1.14)$$

where

$$f_1 = f(t_n, \mathbf{y}_n) \quad (5.1.15a)$$

$$f_2 = f\left(t_n + \frac{\Delta t}{2}, \mathbf{y}_n + \frac{\Delta t}{2} f_1\right) \tag{5.1.15b}$$

$$f_3 = f\left(t_n + \frac{\Delta t}{2}, \mathbf{y}_n + \frac{\Delta t}{2} f_2\right) \tag{5.1.15c}$$

$$f_4 = f\left(t_n + \Delta t, \mathbf{y}_n + \Delta t \cdot f_3\right) . \tag{5.1.15d}$$

This scheme gives a local truncation error which is $O(\Delta t^5)$. The cumulative (global) error in this case case is fourth order so that for $t \sim O(1)$ then the error is $O(\Delta t^4)$. The key to this method, as well as any of the other Runge-Kutta schemes, is the use of intermediate time-steps to improve accuracy. For the 4th order scheme presented here, a graphical representation of this derivative sampling at intermediate time-steps is shown in Fig. 2.

Adams method: multi-stepping techniques

The development of the Runge-Kutta schemes rely on the definition of the derivative and Taylor expansions. Another approach to solving (5.1.1) is to start with the fundamental theorem of calculus [2]. Thus the differential equation can be integrated over a time-step Δt to give

$$\frac{d\mathbf{y}}{dt} = f(\mathbf{y}, t) \quad \Rightarrow \quad \mathbf{y}(t + \Delta t) - \mathbf{y}(t) = \int_t^{t+\Delta t} f(t, y) dt \, . \tag{5.1.16}$$

And once again using our iteration notation we find

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} f(t, y) dt \, . \tag{5.1.17}$$

This iteration relation is simply a restatement of (5.1.7) with $\Delta t \cdot \phi = \int_{t_n}^{t_{n+1}} f(t, y) dt$. However, at this point, no approximations have been made and (5.1.17) is exact. The numerical solution will be found by approximating $f(t, y) \approx p(t, y)$ where $p(t, y)$ is a polynomial. Thus the iteration scheme in this instance will be given by

$$\mathbf{y}_{n+1} \approx \mathbf{y}_n + \int_{t_n}^{t_{n+1}} p(t, y) dt \, . \tag{5.1.18}$$

It only remains to determine the form of the polynomial to be used in the approximation.

The *Adams-Bashforth* suite of computational methods uses the current point and a determined number of past points to evaluate the future solution. As with the Runge-Kutta schemes, the order of accuracy is determined by the choice of ϕ. In the Adams-Bashforth case, this relates directly to the choice of the polynomial approximation $p(t, y)$. A first-order scheme can easily be constructed by allowing

$$p_1(t) = \text{constant} = f(t_n, \mathbf{y}_n) \, , \tag{5.1.19}$$

where the present point and no past points are used to determine the value of the polynomial. Inserting this first-order approximation into (5.1.18) results in the previously found Euler scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_n, \mathbf{y}_n)\,. \tag{5.1.20}$$

Alternatively, we could assume that the polynomial used both the current point and the previous point so that a second-order scheme resulted. The linear polynomial which passes through these two points is given by

$$p_2(t) = f_{n-1} + \frac{f_n - f_{n-1}}{\Delta t}(t - t_n)\,. \tag{5.1.21}$$

When inserted into (5.1.18), this linear polynomial yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \left(f_{n-1} + \frac{f_n - f_{n-1}}{\Delta t}(t - t_n) \right) dt\,. \tag{5.1.22}$$

Upon integration and evaluation at the upper and lower limits, we find the following 2nd order Adams-Bashforth scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}\left[3f(t_n, \mathbf{y}_n) - f(t_{n-1}, \mathbf{y}_{n-1})\right]. \tag{5.1.23}$$

In contrast to the Runge-Kutta method, this is a *two-step algorithm* which requires two initial conditions. This technique can be easily generalized to include more past points and thus higher accuracy. However, as accuracy is increased, so are the number of initial conditions required to step forward one time-step Δt. Aside from the first-order accurate scheme, any implementation of Adams-Bashforth will require a *boot strap* to generate a second "initial condition" for the solution iteration process.

The Adams-Bashforth scheme uses current and past points to approximate the polymial $p(t, y)$ in (5.1.18). If instead a future point, the present, and the past is used, then the scheme is known as an *Adams-Moulton method*. As before, a first-order scheme can easily be constructed by allowing

$$p_1(t) = \text{constant} = f(t_{n+1}, \mathbf{y}_{n+1})\,, \tag{5.1.24}$$

where the future point and no past and present points are used to determine the value of the polynomial. Inserting this first-order approximation into (5.1.18) results in the *backward Euler scheme*

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_{n+1}, \mathbf{y}_{n+1})\,. \tag{5.1.25}$$

Alternatively, we could assume that the polynomial used both the future point and the current point so that a second-order scheme resulted. The linear polynomial which passes through these two points is given by

$$p_2(t) = f_n + \frac{f_{n+1} - f_n}{\Delta t}(t - t_n)\,. \tag{5.1.26}$$

Inserted into (5.1.18), this linear polynomial yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \left(f_n + \frac{f_{n+1} - f_n}{\Delta t}(t - t_n) \right) dt \,. \tag{5.1.27}$$

Upon integration and evaluation at the upper and lower limits, we find the following 2nd order Adams-Moulton scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2} \left[f(t_{n+1}, \mathbf{y}_{n+1}) + f(t_n, \mathbf{y}_n) \right] . \tag{5.1.28}$$

Once again this is a two-step algorithm. However, it is categorically different than the Adams-Bashforth methods since it results in an *implicit scheme*, i.e. the unknown value $\mathbf{y}_{n+1}$ is specified through a nonlinear equation (5.1.28). The solution of this nonlinear system can be very difficult, thus making *explicit schemes* such as Runge-Kutta and Adams-Bashforth, which are simple iterations, more easily handled. However, implicit schemes can have advantages when considering stability issues related to time-stepping. This is explored further in the notes.

One way to circumvent the difficulties of the implicit stepping method while still making use of its power is to use a *Predictor-Corrector method*. This scheme draws on the power of both the Adams-Bashforth and Adams-Moulton schemes. In particular, the second order implicit scheme given by (5.1.28) requires the value of $f(t_{n+1}, \mathbf{y}_{n+1})$ in the right hand side. If we can predict (approximate) this value, then we can use this predicted value to solve (5.1.28) explicitly. Thus we begin with a predictor step to estimate $\mathbf{y}_{n+1}$ so that $f(t_{n+1}, \mathbf{y}_{n+1})$ can be evaluated. We then insert this value into the right hand side of (5.1.28) and explicitly find the corrected value of $\mathbf{y}_{n+1}$. The second-order predictor-corrector steps are then as follows:

$$\text{predictor (Adams-Bashforth): } \mathbf{y}^P_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}[3f_n - f_{n-1}] \tag{5.1.29a}$$

$$\text{corrector (Adams-Moulton): } \mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}[f(t_{n+1}, \mathbf{y}^P_{n+1}) + f(t_n, \mathbf{y}_n)]. \tag{5.1.29b}$$

Thus the scheme utilizes both explicit and implicit time-stepping schemes without having to solve a system of nonlinear equations.

Higher order differential equations

Thus far, we have considered systems of first order equations. Higher order differential equations can be put into this form and the methods outlined here can be applied. For example, consider the third-order, nonhomogeneous, differential equation

$$\frac{d^3u}{dt^3} + u^2\frac{du}{dt} + \cos t \cdot u = g(t) \,. \tag{5.1.30}$$

By defining

$$y_1 = u \tag{5.1.31a}$$
$$y_2 = \frac{du}{dt} \tag{5.1.31b}$$
$$y_3 = \frac{d^2u}{dt^2}\,, \tag{5.1.31c}$$

we find that $dy_3/dt = d^3u/dt^3$. Using the original equation along with the definitions of y_i we find that

$$\frac{dy_1}{dt} = y_2 \tag{5.1.32a}$$
$$\frac{dy_2}{dt} = y_3 \tag{5.1.32b}$$
$$\frac{dy_3}{dt} = \frac{d^3u}{dt^3} = -u^2\frac{du}{dt} - \cos t \cdot u + g(t) = -y_1^2 y_2 - \cos t \cdot y_1 + g(t) \tag{5.1.32c}$$

which results in the original differential equation (5.1.1) considered previously

$$\frac{d\mathbf{y}}{dt} = \frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ -y_1^2 y_2 - \cos t \cdot y_1 + g(t) \end{pmatrix} = f(\mathbf{y}, t)\,. \tag{5.1.33}$$

At this point, all the time-stepping techniques developed thus far can be applied to the problem. It is imperative to write any differential equation as a first-order system before solving it numerically with the time-stepping schemes developed here.

MATLAB commands

The time-stepping schemes considered here are all available in the MATLAB suite of differential equation solvers. The following are a few of the most common solvers:

- **ode23:** second-order Runge-Kutta routine
- **ode45:** fourth-order Runge-Kutta routine
- **ode113:** variable order predictor-corrector routine
- **ode15s:** variable order Gear method for stiff problems [3, 4]

5.2 Error analysis for time-stepping routines

Accuracy and *stability* are fundamental to numerical analysis and are the key factors in evaluating any numerical integration technique. Therefore, it is essential

to evaluate the accuracy and stability of the time-stepping schemes developed. Rarely does it occur that both accuracy and stability work in concert. In fact, they often are offsetting and work directly against each other. Thus a highly accurate scheme may compromise stability, whereas a low accuracy scheme may have excellent stability properties.

We begin by exploring accuracy. In the context of time-stepping schemes, the natural place to begin is with Taylor expansions. Thus we consider the expansion

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \Delta t \cdot \frac{d\mathbf{y}(t)}{dt} + \frac{\Delta t^2}{2} \cdot \frac{d^2\mathbf{y}(c)}{dt^2} \tag{5.2.1}$$

where $c \in [t, t+\Delta t]$. Since we are considering $dy/dt = f(t,y)$, the above formula reduces to the Euler iteration scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_n, \mathbf{y}_n) + O(\Delta t^2)\,. \tag{5.2.2}$$

It is clear from this that the truncation error is $O(\Delta t^2)$. Specifically, the truncation error is given by $\Delta t^2/2 \cdot d^2\mathbf{y}(c)/dt^2$.

Of importance is how this truncation error contributes to the overall error in the numerical solution. Two types of error are important to identify: *local* and *global error*. Each is significant in its own right. However, in practice we are only concerned with the global (cumulative) error. The *global discretization error* is given by

$$E_k = \mathbf{y}(t_k) - \mathbf{y}_k \tag{5.2.3}$$

where $\mathbf{y}(t_k)$ is the exact solution and $\mathbf{y}_k$ is the numerical solution. The *local discretization error* is given by

$$\epsilon_{k+1} = \mathbf{y}(t_{k+1}) - (\mathbf{y}(t_k) + \Delta t \cdot \phi) \tag{5.2.4}$$

where $\mathbf{y}(t_{k+1})$ is the exact solution and $\mathbf{y}(t_k) + \Delta t \cdot \phi$ is a one-step approximation over the time interval $t \in [t_n, t_{n+1}]$.

For the Euler method, we can calculate both the local and global error. Given a time-step Δt and a specified time interval $t \in [a,b]$, we have after K steps that $\Delta t \cdot K = b - a$. Thus we find

$$\text{local:} \quad \epsilon_k = \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(c_k)}{dt^2} \sim O(\Delta t^2) \tag{5.2.5a}$$

$$\begin{aligned}\text{global:} \quad E_k &= \sum_{j=1}^{K} \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(c_j)}{dt^2} \approx \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \cdot K \\ &= \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \cdot \frac{b-a}{\Delta t} = \frac{b-a}{2}\Delta t \cdot \frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \sim O(\Delta t)\end{aligned} \tag{5.2.5b}$$

which gives a local error for the Euler scheme which is $O(\Delta t^2)$ and a global error which is $O(\Delta t)$. Thus the cumulative error is poor for the Euler scheme, i.e. it is not very accurate.

scheme	local error ϵ_k	global error E_k
Euler	$O(\Delta t^2)$	$O(\Delta t)$
2nd order Runge-Kutta	$O(\Delta t^3)$	$O(\Delta t^2)$
4th order Runge-Kutta	$O(\Delta t^5)$	$O(\Delta t^4)$
2nd order Adams-Bashforth	$O(\Delta t^3)$	$O(\Delta t^2)$

Table 7: Local and global discretization errors associated with various time-stepping schemes.

A similar procedure can be carried out for all the schemes discussed thus far, including the multi-step Adams schemes. Table 7 illustrates various schemes and their associated local and global errors. The error analysis suggests that the error will always decrease in some power of Δt. Thus it is tempting to conclude that higher accuracy is easily achieved by taking smaller time steps Δt. This would be true if not for round-off error in the computer.

Round-off and step-size

An unavoidable consequence of working with numerical computations is round-off error. When working with most computations, *double precision* numbers are used. This allows for 16-digit accuracy in the representation of a given number. This round-off has significant impact upon numerical computations and the issue of time-stepping.

As an example of the impact of round-off, we consider the Euler approximation to the derivative

$$\frac{d\mathbf{y}}{dt} \approx \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} + \epsilon(\mathbf{y}_n, \Delta t) \tag{5.2.6}$$

where $\epsilon(\mathbf{y}_n, \Delta t)$ measures the truncation error. Upon evaluating this expression in the computer, round-off error occurs so that

$$\mathbf{y}_{n+1} = \mathbf{Y}_{n+1} + \mathbf{e}_{n+1}\,. \tag{5.2.7}$$

Thus the combined error between the round-off and truncation gives the following expression for the derivative:

$$\frac{d\mathbf{y}}{dt} = \frac{\mathbf{Y}_{n+1} - \mathbf{Y}_n}{\Delta t} + E_n(\mathbf{y}_n, \Delta t) \tag{5.2.8}$$

where the total error, E_n, is the combination of round-off and truncation such that

$$E_n = E_{\text{round}} + E_{\text{trunc}} = \frac{\mathbf{e}_{n+1} - \mathbf{e}_n}{\Delta t} - \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(c)}{dt^2}\,. \tag{5.2.9}$$

We now determine the maximum size of the error. In particular, we can bound the maximum value of round-off and the derivate to be

$$|\mathbf{e}_{n+1}| \leq e_r \tag{5.2.10a}$$

$$|-\mathbf{e}_n| \leq e_r \tag{5.2.10b}$$

$$M = \max_{c\in[t_n,t_{n+1}]} \left\{ \left| \frac{d^2\mathbf{y}(c)}{dt^2} \right| \right\} . \tag{5.2.10c}$$

This then gives the maximum error to be

$$|E_n| \leq \frac{e_r + e_r}{\Delta t} + \frac{\Delta t^2}{2} M = \frac{2e_r}{\Delta t} + \frac{\Delta t^2 M}{2} . \tag{5.2.11}$$

To minimize the error, we require that $\partial|E_n|/\partial(\Delta t) = 0$. Calculating this derivative gives

$$\frac{\partial|E_n|}{\partial(\Delta t)} = -\frac{2e_r}{\Delta t^2} + M\Delta t = 0 , \tag{5.2.12}$$

so that

$$\Delta t = \left(\frac{2e_r}{M} \right)^{1/3} . \tag{5.2.13}$$

This gives the step size resulting in a minimum error. Thus the smallest step-size is not necessarily the most accurate. Rather, a balance between round-off error and truncation error is achieved to obtain the optimal step-size.

Stability

The accuracy of any scheme is certainly important. However, it is meaningless if the scheme is not stable numerically. The essense of a stable scheme: the numerical solutions do not blow up to infinity. As an example, consider the simple differential equation

$$\frac{dy}{dt} = \lambda y \tag{5.2.14}$$

with

$$y(0) = y_0 . \tag{5.2.15}$$

The analytic solution is easily calculated to be $y(t) = y_0 \exp(\lambda t)$. However, if we solve this problem numerically with a forward Euler method we find

$$y_{n+1} = y_n + \Delta t \cdot \lambda y_n = (1 + \lambda\Delta t) y_n . \tag{5.2.16}$$

After N steps, we find this iteration scheme yields

$$y_N = (1 + \lambda\Delta t)^N y_0 . \tag{5.2.17}$$

Given that we have a certain amount of round off error, the numerical solution would then be given by

$$y_N = (1 + \lambda\Delta t)^N (y_0 + e)\,. \tag{5.2.18}$$

The error then associated with this scheme is given by

$$E = (1 + \lambda\Delta t)^N e\,. \tag{5.2.19}$$

At this point, the following observations can be made. For $\lambda > 0$, the solution $y_N \to \infty$ in Eq. (5.2.18) as $N \to \infty$. So although the error also grows, it may not be significant in comparison to the size of the numerical solution.

In contrast, Eq. (5.2.18) for $\lambda < 0$ is markedly different. For this case, $y_N \to 0$ in Eq. (5.2.18) as $N \to \infty$. The error, however, can dominate in this case. In particular, we have the two following cases for the error given by (5.2.19):

$$\text{I:} \quad |1 + \lambda\Delta t| < 1 \;\text{ then }\; E \to 0 \tag{5.2.20a}$$

$$\text{II:} \quad |1 + \lambda\Delta t| > 1 \;\text{ then }\; E \to \infty\,. \tag{5.2.20b}$$

In case I, the scheme would be considered stable. However, case II holds and is unstable provided $\Delta t > -2/\lambda$.

A general theory of stability can be developed for any one-step time-stepping scheme. Consider the one-step recursion relation for an $M \times M$ system

$$\mathbf{y}_{n+1} = \mathbf{A}\mathbf{y}_n\,. \tag{5.2.21}$$

After N steps, the algorithm yields the solution

$$\mathbf{y}_N = \mathbf{A}^N \mathbf{y}_0\,, \tag{5.2.22}$$

where $\mathbf{y}_0$ is the initial vector. A well known result from linear algebra is that

$$\mathbf{A}^N = \mathbf{S}^{-1}\mathbf{\Lambda}^N\mathbf{S} \tag{5.2.23}$$

where $\mathbf{S}$ is the matrix whose columns are the eigenvectors of $\mathbf{A}$, and

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda_M \end{pmatrix} \;\rightarrow\; \mathbf{\Lambda}^N = \begin{pmatrix} \lambda_1^N & 0 & \cdots & 0 \\ 0 & \lambda_2^N & 0 & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda_M^N \end{pmatrix} \tag{5.2.24}$$

is a diagnonal matrix whose entries are the eigenvalues of $\mathbf{A}$. Thus upon calculating $\mathbf{\Lambda}^N$, we are only concerned with the eigenvalues. In particular, instability occurs if $\Re\{\lambda_i\} > 1$ for $i = 1, 2, ..., M$. This method can be easily generalized to two-step schemes (Adams methods) by considering $\mathbf{y}_{n+1} = \mathbf{A}\mathbf{y}_n + \mathbf{B}\mathbf{y}_{n-1}$.

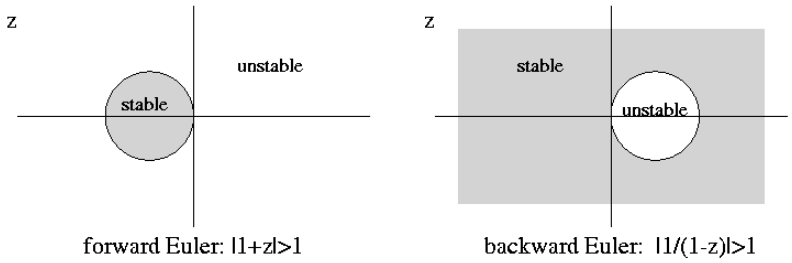


Figure 3: Regions for stable stepping for the forward Euler and backward Euler schemes.

Lending further significance to this stability analysis is its connection with practical implementation. We contrast the difference in stability between the forward and backward Euler schemes. The forward Euler scheme has already been considered in (5.2.16)-(5.2.19). The backward Euler displays significant differences in stability. If we again consider (5.2.14) with (5.2.15), the backward Euler method gives the iteration scheme

$$y_{n+1} = y_n + \Delta t \cdot \lambda y_{n+1}\,, \tag{5.2.25}$$

which after N steps leads to

$$y_N = \left(\frac{1}{1-\lambda\Delta t}\right)^N y_0\,. \tag{5.2.26}$$

The round-off error associated with this scheme is given by

$$E = \left(\frac{1}{1-\lambda\Delta t}\right)^N e\,. \tag{5.2.27}$$

By letting $z = \lambda\Delta t$ be a complex number, we find the following criteria to yield unstable behavior based upon (5.2.19) and (5.2.27)

$$\text{forward Euler:} \quad |1+z| > 1 \tag{5.2.28a}$$

$$\text{backward Euler:} \quad \left|\frac{1}{1-z}\right| > 1\,. \tag{5.2.28b}$$

Figure 3 shows the regions of stable and unstable behavior as a function of z. It is observed that the forward Euler scheme has a very small range of stability whereas the backward Euler scheme has a large range of stability. This large stability region is part of what makes implicit methods so attractive. Thus stability regions can be calculated. However, control of the accuracy is also essential.

5.3 Boundary value problems: the shooting method

To this point, we have only considered the solutions of differential equations for which the initial conditions are known. However, many physical applications do not have specified initial conditions, but rather some given boundary (constraint) conditions. A simple example of such a problem is the second-order boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{5.3.1}$$

on $t \in [a, b]$ with the general boundary conditions

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1 \tag{5.3.2a}$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2 \,. \tag{5.3.2b}$$

Thus the solution is defined over a specific interval and must satisfy the relations (5.3.2) at the end points of the interval. Figure 4 gives a graphical representation of a generic boundary value problem solution. We discuss the algorithm necessary to make use of the time-stepping schemes in order to solve such a problem.

The Shooting Method

The boundary value problems constructed here require information at the present time ($t = a$) and a future time ($t = b$). However, the time-stepping schemes developed previously only require information about the starting time $t = a$. Some effort is then needed to reconcile the time-stepping schemes with the boundary value problems presented here.

We begin by reconsidering the generic boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{5.3.3}$$

on $t \in [a, b]$ with the boundary conditions

$$y(a) = \alpha \tag{5.3.4a}$$

$$y(b) = \beta \,. \tag{5.3.4b}$$

The stepping schemes considered thus far for second order differential equations involve a choice of the initial conditions $y(a)$ and $y'(a)$. We can still approach the boundary value problem from this framework by choosing the "initial" conditions

$$y(a) = \alpha \tag{5.3.5a}$$

$$\frac{dy(a)}{dt} = A \,, \tag{5.3.5b}$$

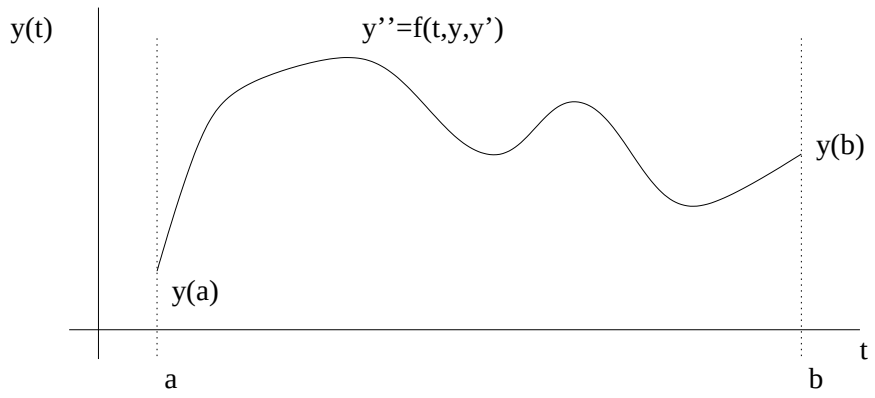


Figure 4: Graphical depiction of the structure of a typical solution to a boundary value problem with constraints at $t = a$ and $t = b$.

where the constant A is chosen so that as we advance the solution to $t = b$ we find $y(b) = \beta$. The shooting method gives an iterative procedure with which we can determine this constant A. Figure 5 illustrates the solution of the boundary value problem given two distinct values of A. In this case, the value of $A = A_1$ gives a value for the initial slope which is too low to satisfy the boundary conditions (5.3.4), whereas the value of $A = A_2$ is too large to satisfy (5.3.4).

Computational Algorithm

The above example demonstrates that adjusting the value of A in (5.3.5b) can lead to a solution which satisfies (5.3.4b). We can solve this using a self-consistent algorithm to search for the appropriate value of A which satisfies the original problem. The basic algorithm is as follows:

1. Solve the differential equation using a time-stepping scheme with the initial conditions $y(a) = \alpha$ and $y'(a) = A$.

2. Evaluate the solution $y(b)$ at $t = b$ and compare this value with the target value of $y(b) = \beta$.

3. Adjust the value of A (either bigger or smaller) until a desired level of tolerance and accuracy is achieved. A bisection method for determining values of A, for instance, may be appropriate.

4. Once the specified accuracy has been achieved, the numerical solution is complete and is accurate to the level of the tolerance chosen and the discretization scheme used in the time-stepping.

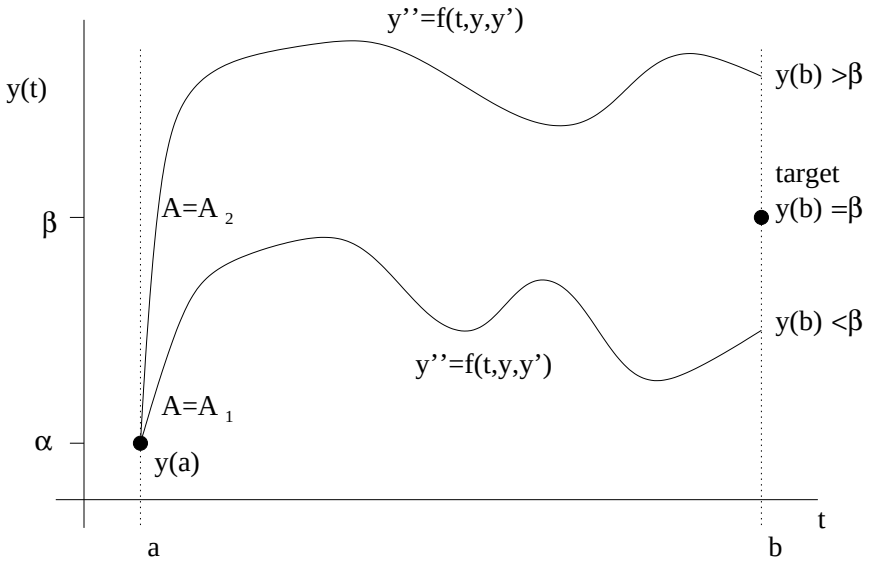


Figure 5: Solutions to the boundary value problem with $y(a) = \alpha$ and $y'(a) = A$. Here, two values of A are used to illustrate the solution behavior and its lack of matching the correct boundary value $y(b) = \beta$. However, the two solutions suggest that a bisection scheme could be used to find the correct solution and value of A.

We illustrate graphically a bisection process in Fig. 6 and show the convergence of the method to the numerical solution which satisfies the original boundary conditions $y(a) = \alpha$ and $y(b) = \beta$. This process can occur quickly so that convergence is achieved in a relatively low amount of iterations provided the differential equation is well behaved.

Implementing MATLAB time-stepping schemes

Up to this point in our discussions of differential equations, our solution techniques have relied on solving initial value problems with some appropriate iteration procedure. To implement one of these solution schemes in MATLAB requires information about the equation, the time or spatial range to solve for, and the initial conditions.

To build on a specific example, consider the third-order, nonhomogeneous, differential equation

$$\frac{d^3u}{dt^3} + u^2\frac{du}{dt} + \cos t \cdot u = A\sin^2 t\,. \tag{5.3.6}$$

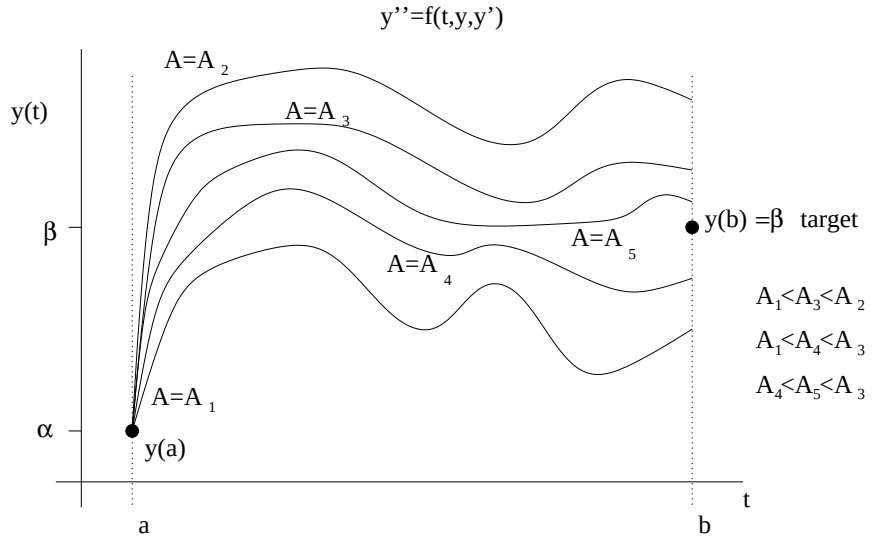


Figure 6: Graphical illustration of the shooting process which uses a bisection scheme to converge to the appropriate value of A for which $y(b) = \beta$.

By defining

$$y_1 = u \tag{5.3.7a}$$

$$y_2 = \frac{du}{dt} \tag{5.3.7b}$$

$$y_3 = \frac{d^2u}{dt^2}\,, \tag{5.3.7c}$$

we find that $dy_3/dt = d^3u/dt^3$. Using the original equation along with the definitions of y_i we find that

$$\frac{dy_1}{dt} = y_2 \tag{5.3.8a}$$

$$\frac{dy_2}{dt} = y_3 \tag{5.3.8b}$$

$$\frac{dy_3}{dt} = \frac{d^3u}{dt^3} = -u^2\frac{du}{dt} - \cos t\cdot u + A\sin^2 t = -y_1^2 y_2 - \cos t\cdot y_1 + A\sin^2 t \tag{5.3.8c}$$

which results in the first-order system (5.1.1) considered previously

$$\frac{d\mathbf{y}}{dt} = \frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ -y_1^2 y_2 - \cos t\cdot y_1 + A\sin^2 t \end{pmatrix} = f(\mathbf{y}, t)\,. \tag{5.3.9}$$

At this point, all the time-stepping techniques developed thus far can be applied to the problem. It is imperative to write any differential equation as a first-order system before solving it numerically with the time-stepping schemes developed here.

Before solving the equation, the initial conditions and time span must be defined. The initial conditions in this case specify $y_1(t_0)$, $y_2(t_0)$ and $y_3(t_0)$, or equivalently $u(t_0)$, $u_t(t_0)$ and $u_{tt}(t_0)$. The time span specifies the beginning and end times for solving the equations. Thus $t \in [t_0, t_f]$, where t_0 is the initial time and t_f the final time. For this example, let's consider solving the equations with initial conditions $y_1(0) = 1$, $y_2(0) = 0$ and $y_3(0) = 0$ over the interval $t \in [t_0, t_f] = [0, 30]$ with $A = 2$. In MATLAB this would be accomplished with

```
y0=[1 0 0];       % initial conditions
tspan=[0 30];     % time interval
A=2;              % parameter value
```

These can now be used in calling the differential equation solver.

The basic command structure and function call to solve this equation is given as follows:

```
[t,y]=ode45('rhs',tspan,y0,[],A)
```

This piece of code calls a function **rhs.m** which contains information about the differential equations. It sends to this function the time span for the solution, the initial conditions, and the parameter value of A. The brackets are placed in a position for specifying accuracy and tolerance options. By placing the brackets with nothing inside, the default accuracy will be used by **ode45**. To access other methods, simply **ode45** with **ode23**, **ode113**, or **ode15s** as needed.

The function **rhs.m** contains all the information about the differential equation. Specifically, the right hand side of (5.3.9) is of primary importance. This right hand side function looks as follows:

```
function rhs=rhs(t,y,dummy,A)  % function call setup
rhs=[y(1);
     y(2);
    -(y(1)^2)*y(2)-cos(t)*y(1)+A*sin(t)];  % rhs vector
```

This function imports the variable t which takes on values between those given by *tspan*. Thus t starts at $t = 0$ and ends at $t = 30$ in this case. In between these points, t gives the current value of time. The variable y stores the solution. Initially, it takes on the value of $y0$. The *dummy* slot is for sending in information about the tolerances and the last slot with A allows you to send in any parameters necessary in the right hand side.

Note that the output of **ode45** are the vectors t and y. The vector t gives all the points between *tspan* for which the solution was calculated. The solution

at these time points is given as the columns of matrix y. The first column corresponds to the solution y_1, the second column contains y_2 as a function of time, and the third column gives y_3. To plot the solution $u = y_1$ as a function of time, we would plot the vector t versus the first column of y:

```
figure(1), plot(t,y(:,1))   %  plot solution u
figure(2), plot(t,y(:,2))   %  plot derivative du/dt
```

A couple of important points and options. The **inline** command could also be used in place of the function call. Also, it may desirable to have the solution evaluated at specified time points. For instance, if we wanted the solution evaluated from $t = 0$ to $t = 30$ in steps of 0.5, then choose

```
tspan=0:0.5:30  % evaluate the solution every dt=0.5
```

Note that MATLAB still determines the step sizes necessary to maintain a given tolerance. It simply interpolates to generate the values at the desired locations.

5.4 Implementation of the shooting method

The implementation of the shooting scheme relies on the effective use of a time-stepping algorithm along with a root finding method for choosing the appropriate initial conditions which solve the boundary value problem. Boundary value problems often arise as eigenvalue systems for which the eigenvalue and eigenfunction must both be determined. As an example of such a problem, we consider the Schrödinger equation from quantum mechanics which is a second order differential equation

$$\frac{d^2\psi_n}{dx^2} + [n(x) - \beta_n]\,\psi_n = 0 \tag{5.4.1}$$

with the boundary conditions $\psi(\pm 1) = 0$.

Rewriting the differential equation into a coupled first order system gives

$$\mathbf{x}' = \begin{pmatrix} 0 & 1 \\ \beta_n - n(x) & 0 \end{pmatrix} \mathbf{x} \tag{5.4.2}$$

where $\mathbf{x} = (x_1 \;\; x_2)^T = (\psi_n \;\; d\psi_n/dx)^T$. The boundary conditions are

$$x = 1: \qquad \psi_n(1) = x_1(1) = 0 \tag{5.4.3a}$$

$$x = -1: \qquad \psi_n(-1) = x_1(-1) = 0\,. \tag{5.4.3b}$$

At this stage, we will also assume that $n(x) = n_0$ for simplicity.

With the problem thus defined, we turn our attention to the key aspects in the computational implementation of the boundary value problem solver. These are

- **FOR** loops
- **IF** statements
- time-stepping algorithms: **ode23, ode45, ode113, ode15s**
- step-size control
- code development and flow

Every code will be controlled by a set of FOR loops and IF statements. It is imperative to have proper placement of these control statements in order for the code to operate successfully.

Flow Control

In order to begin coding, it is always prudent to construct the basic structure of the algorithm. In particular, it is good to determine the number of FOR loops and IF statements which may be required for the computations. What is especially important is determining the hierarchy structure for the loops. To solve the boundary value problem proposed here, we require two FOR loops and one IF statement block. The outermost FOR loop of the code should determine the number of eigenvalues and eigenmodes to be searched for. Within this FOR loop exists a second FOR loop which iterates the shooting method so that the solution converges to the correct boundary value solution. This second FOR loop has a logical IF statement which needs to check whether the solution has indeed converged to the boundary value solution, or whether adjustment of the value of β_n is necessary and the iteration procedure needs to be continued. Figure 7 illustrates the backbone of the numerical code for solving the boundary value problem. It includes the two FOR loops and logical IF statement block as the core of its algorithmic structure. For a nonlinear problem, a third FOR loop would be required for A in order to achieve the normalization of the eigenfunctions to unity.

The various pieces of the code are constructed here using the MATLAB programming language. We begin with the initialization of the parameters.

Initialization

```
clear all;  % clear all previously defined variables
close all;  % clear all previously defined figures

tol=10^(-4);  % define a tolerance level to be achieved
            % by the shooting algorithm
col=['r','b','g','c','m','k'];  % eigenfunction colors

n0=100;    % define the parameter n0
```

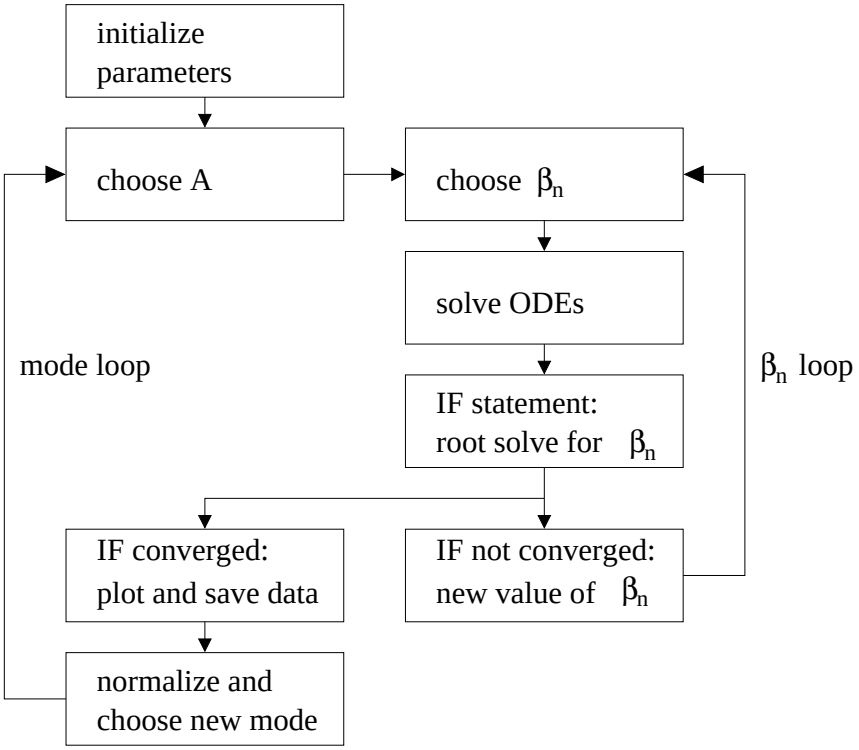


Figure 7: Basic algorithm structure for solving the boundary value problem. Two FOR loops are required to step through the values of β_n and A along with a single IF statement block to check for convergence of the solution

```
A=1;       % define the initial slope at x=-1
x0=[0 A];  % initial conditions: x1(-1)=0, x1'(-1)=A
xp=[-1 1]; % define the span of the computational domain
```

Upon completion of the initialization process for the parameters which are not involved in the main loop of the code, we move into the main FOR loop which searches out a specified number of eigenmodes. Embedded in this FOR loop is a second FOR loop which attemps different values of β_n until the correct eigenvalue is found. An IF statement is used to check the convergence of values of β_n to the appropriate value.

Main Program

```
beta_start=n0;  % beginning value of beta
for modes=1:5   %  begin mode loop
```

```
    beta=beta_start;   % initial value of eigenvalue beta
    dbeta=n0/100;      % default step size in beta
    for j=1:1000  % begin convergence loop for beta
      [t,y]=ode45('shoot2',xp,x0,[],n0,beta); % solve ODEs
      if abs(y(end,1)-0) < tol  % check for convergence
         beta                   % write out eigenvalue
         break                  % get out of convergence loop
      end
      if (-1)^(modes+1)*y(end,1)>0  %  this IF statement block
         beta=beta-dbeta;           %  checks to see if beta
      else                          %  needs to be higher or lower
         beta=beta+dbeta/2;         %  and uses bisection to
         dbeta=dbeta/2;             %  converge to the solution
      end                           %
    end   % end convergence loop
    beta_start=beta-0.1;  % after finding eigenvalue, pick
                          % new starting value for next mode
    norm=trapz(t,y(:,1).*y(:,1))  % calculate the normalization
    plot(t,y(:,1)/sqrt(norm),col(modes)); hold on  % plot modes
  end   % end mode loop
```

The code uses *ode45*, which is a fourth-order Runge-Kutta method, to solve the differential equation and advance the solution. The function *shoot2.m* is called in this routine. For the differential equation considered here, the function *shoot2.m* would be the following:

shoot2.m

```
function rhs=shoot2(xspan,x,dummy,n0,beta)
rhs=[ x(2)
     (beta-n0)*x(1) ];
```

This code will find the first five eigenvalues and plot their corresponding normalized eigenfunctions. The bisection method implemented to adjust the values of β_n to find the boundary value solution is based upon observations of the structure of the even and odd eigenmodes. In general, it is always a good idea to first explore the behavior of the solutions of the boundary value problem before writing the shooting routine. This will give important insights into the behavior of the solutions and will allow for a proper construction of an accurate and efficient bisection method. Figure 8 illustrates several characteristic features of this boundary value problem. In Fig. 8(a) and 8(b), the behavior of the solution near the first even and first odd solution is exhibited. From Fig. 8(a) it is seen that for the even modes increasing values of β bring the solution from $\psi_n(1) > 0$ to $\psi_n(1) < 0$. In contrast, odd modes go from $\psi_n(1) < 0$ to $\psi_n(1) > 0$ as β is increased. This observation forms the basis for the bisection

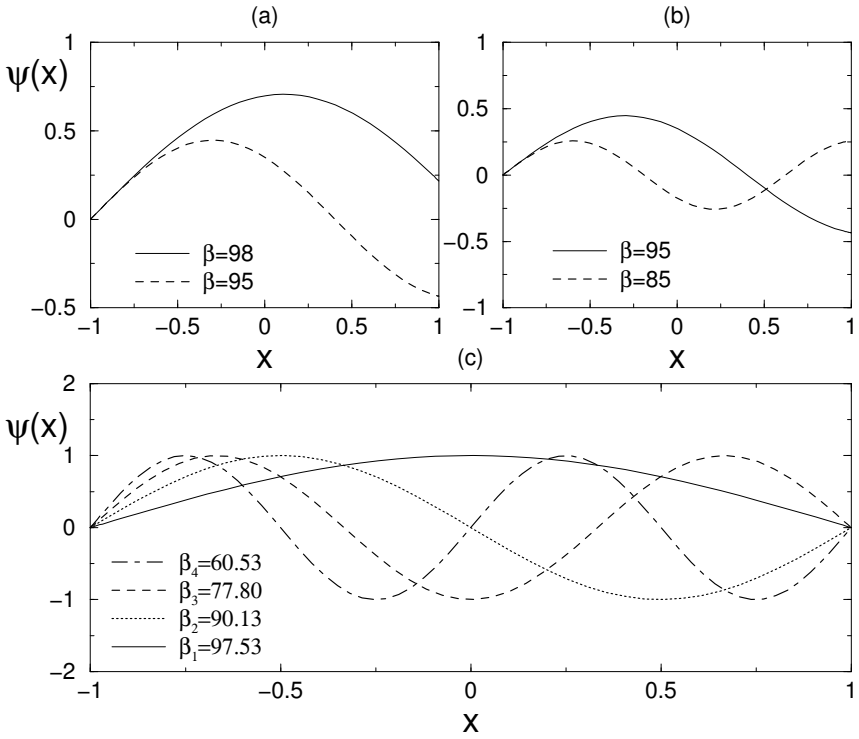


Figure 8: In (a) and (b) the behavior of the solution near the first even and first odd solution is depicted. Note that for the even modes increasing values of β bring the solution from $\psi_n(1) > 0$ to $\psi_n(1) < 0$. In contrast, odd modes go from $\psi_n(1) < 0$ to $\psi_n(1) > 0$ as β is increased. In (c) the first four normalized eigenmodes along with their corresponding eigenvalues are illustrated for $n_0 = 100$.

method developed in the code. Figure 8(c) illustrates the first four normalized eigenmodes along with their corresponding eigenvalues.

5.5 Boundary value problems: direct solve and relaxation

The shooting method is not the only method for solving boundary value problems. The direct method of solution relies on Taylor expanding the differential equation itself. For linear problems, this results in a matrix problem of the form $\mathbf{Ax} = \mathbf{b}$. For nonlinear problems, a nonlinear system of equations must be solved using a relaxation scheme, i.e. a Newton or Secant method. The proto-

typical example of such a problem is the second-order boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{5.5.1}$$

on $t \in [a, b]$ with the general boundary conditions

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1 \tag{5.5.2a}$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2 \,. \tag{5.5.2b}$$

Thus the solution is defined over a specific interval and must satisfy the relations (5.5.2) at the end points of the interval.

Before considering the general case, we simplify the method by considering the linear boundary value problem

$$\frac{d^2y}{dt^2} = p(t)\frac{dy}{dt} + q(t)y + r(t) \tag{5.5.3}$$

on $t \in [a, b]$ with the simplified boundary conditions

$$y(a) = \alpha \tag{5.5.4a}$$

$$y(b) = \beta \,. \tag{5.5.4b}$$

Taylor expanding the differential equation and boundary conditions will generate the linear system of equations which solve the boundary value problem. Tables 4-6 (Sec. 4.1) summarize the second-order and fourth-order central difference schemes along with the forward- and backward-difference formulas which are accurate to second-order.

To solve the simplified linear boundary value problem above which is accurate to second order, we use table 4 for the second and first derivatives. The boundary value problem then becomes

$$\frac{y(t+\Delta t)-2y(t)+y(t-\Delta t)}{\Delta t^2} = p(t)\frac{y(t+\Delta t)-y(t-\Delta t)}{2\Delta t} + q(t)y(t) + r(t) \tag{5.5.5}$$

with the boundary conditions $y(a) = \alpha$ and $y(b) = \beta$. We can rearrange this expression to read

$$\left[1 - \frac{\Delta t}{2}p(t)\right] y(t + \Delta t) - \left[2 + \Delta t^2 q(t)\right] y(t) + \left[1 + \frac{\Delta t}{2}\right] y(t - \Delta t) = \Delta t^2 r(t) \,. \tag{5.5.6}$$

We discretize the computational domain and denote $t_0 = a$ to be the left boundary point and $t_N = b$ to be the right boundary point. This gives the boundary conditions

$$y(t_0) = y(a) = \alpha \tag{5.5.7a}$$

$$y(t_N) = y(b) = \beta \,. \tag{5.5.7b}$$

The remaining $N-1$ points can be recast as a matrix problem $\mathbf{Ax}=\mathbf{b}$ where

$$\mathbf{A}=\begin{bmatrix} 2+\Delta t^2 q(t_1) & -1+\frac{\Delta t}{2}p(t_1) & 0 & \cdots & & 0 \\ -1-\frac{\Delta t}{2}p(t_2) & 2+\Delta t^2 q(t_2) & -1+\frac{\Delta t}{2}p(t_2) & 0 & \cdots & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & & & \ddots & \ddots & -1+\frac{\Delta t}{2}p(t_{N-2}) \\ 0 & \cdots & & 0 & -1-\frac{\Delta t}{2}p(t_{N-1}) & 2+\Delta t^2 q(t_{N-1}) \end{bmatrix} \tag{5.5.8}$$

and

$$\mathbf{x}=\begin{bmatrix} y(t_1) \\ y(t_2) \\ \vdots \\ y(t_{N-2}) \\ y(t_{N-1}) \end{bmatrix} \qquad \mathbf{b}=\begin{bmatrix} -\Delta t^2 r(t_1)+(1+\Delta t p(t_1)/2)y(t_0) \\ -\Delta t^2 r(t_2) \\ \vdots \\ -\Delta t^2 r(t_{N-2}) \\ -\Delta t^2 r(t_{N-1})+(1-\Delta t p(t_{N-1})/2)y(t_N) \end{bmatrix}. \tag{5.5.9}$$

Thus the solution can be found by a direct solve of the linear system of equations.

Nonlinear Systems

A similar solution procedure can be carried out for nonlinear systems. However, difficulties arise from solving the resulting set of nonlinear algebraic equations. We can once again consider the general differential equation and expand with second-order accurate schemes:

$$y''=f(t,y,y') \;\rightarrow\; \frac{y(t+\Delta t)-2y(t)+y(t-\Delta t)}{\Delta t^2}=f\left(t,y(t),\frac{y(t+\Delta t)-y(t-\Delta t)}{2\Delta t}\right). \tag{5.5.10}$$

We discretize the computational domain and denote $t_0=a$ to be the left boundary point and $t_N=b$ to be the right boundary point. Considering again the simplified boundary conditions $y(t_0)=y(a)=\alpha$ and $y(t_N)=y(b)=\beta$ gives the following nonlinear system for the remaining $N-1$ points.

$$\begin{aligned} &2y_1-y_2-\alpha+\Delta t^2 f(t_1,y_1,(y_2-\alpha)/2\Delta t)=0 \\ &-y_1+2y_2-y_3+\Delta t^2 f(t_2,y_2,(y_3-y_1)/2\Delta t)=0 \\ &\qquad\vdots \\ &-y_{N-3}+2y_{N-2}-y_{N-1}+\Delta t^2 f(t_{N-2},y_{N-2},(y_{N-1}-y_{N-3})/2\Delta t)=0 \\ &-y_{N-2}+2y_{N-1}-\beta+\Delta t^2 f(t_{N-1},y_{N-1},(\beta-y_{N-2})/2\Delta t)=0. \end{aligned}$$

This $(N-1)\times(N-1)$ nonlinear system of equations can be very difficult to solve and imposes a severe constraint on the usefulness of the scheme. However,

there may be no other way of solving the problem and a solution to these system of equations must be computed. Further complicating the issue is the fact that for nonlinear systems such as these, there are no guarantees about the existence or uniqueness of solutions. The best approach is to use a *relaxation* scheme which is based upon Newton or Secant method iterations.

Solving Nonlinear Systems: Newton-Raphson Iteration

The only built-in MATLAB command which solves nonlinear system of equations is FSOLVE. However, this command is now packaged within the optimization toolbox. Most users of MATLAB do not have access to this toolbox and alternatives must be sought. We therefore develop the basic ideas of the Newton-Raphson Iteration method, commonly known as a Newton's method. We begin by considering a single nonlinear equation

$$f(x_r) = 0 \tag{5.5.12}$$

where x_r is the root to the equation and the value being sought. We would like to develop a scheme which systematically determines the value of x_r. The Newton-Raphson method is an iterative scheme which relies on an initial guess, x_0, for the value of the root. From this guess, subsequent guesses are determined until the scheme either converges to the root x_r or the scheme diverges and another initial guess is used. The sequence of guesses $(x_0, x_1, x_2, ...)$ is generated from the slope of the function $f(x)$. The graphical procedure is illustrated in Fig. 9. In essence, everything relies on the slope formula as illustrated in Fig. 9(a):

$$\text{slope} = \frac{df(x_n)}{dx} = \frac{\text{rise}}{\text{run}} = \frac{0 - f(x_n)}{x_{n+1} - x_n}\,. \tag{5.5.13}$$

Rearranging this gives the Newton-Raphson iterative relation

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\,. \tag{5.5.14}$$

A graphical example of how the iteration procedure works is given in Fig. 9(b) where a sequence of iterations is demonstrated. Note that the scheme fails if $f'(x_n) = 0$ since then the slope line never intersects $y = 0$. Further, for certain guesses the iterations may diverge. Provided the initial guess is sufficiently close, the scheme usually will converge. Conditions for convergence can be found in Burden and Faires [5].

The Newton method can be generalized for system of nonlinear equations. The details will not be discussed here, but the Newton iteration scheme is similar to that developed for the single function case. Given a system:

$$\mathbf{F}(\mathbf{x_n}) = \begin{bmatrix} f_1(x_1, x_2, x_3, ..., x_N) \\ f_2(x_1, x_2, x_3, ..., x_N) \\ \vdots \\ f_N(x_1, x_2, x_3, ..., x_N) \end{bmatrix} = \mathbf{0}\,, \tag{5.5.15}$$

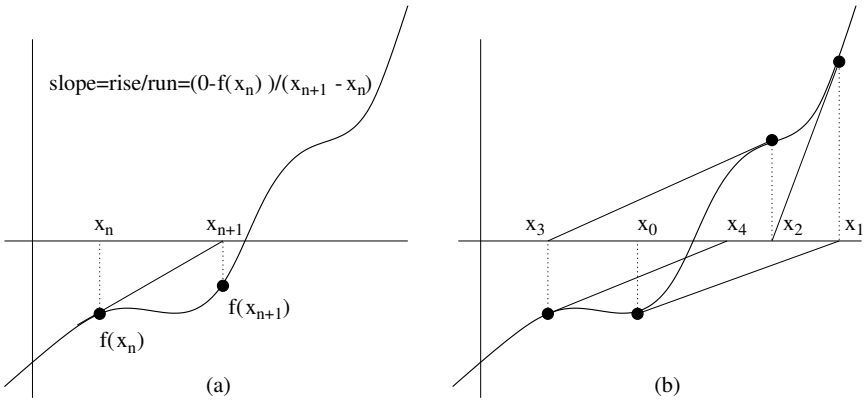


Figure 9: Construction and implementation of the Newton-Raphson iteration formula. In (a), the slope is the determining factor in deriving the Newton-Raphson formula. In (b), a graphical representation of the iteration scheme is given.

the iteration scheme is

$$\mathbf{x_{n+1}} = \mathbf{x_n} + \mathbf{\Delta x_n} \tag{5.5.16}$$

where

$$\mathbf{J}(\mathbf{x_n})\mathbf{\Delta x_n} = -\mathbf{F}(\mathbf{x_n}) \tag{5.5.17}$$

and $\mathbf{J}(\mathbf{x_n})$ is the Jacobian matrix

$$\mathbf{J}(\mathbf{x_n}) = \begin{bmatrix} f_{1_{x_1}} & f_{1_{x_2}} & \cdots & f_{1_{x_N}} \\ f_{2_{x_1}} & f_{2_{x_2}} & \cdots & f_{2_{x_N}} \\ \vdots & \vdots & & \vdots \\ f_{N_{x_1}} & f_{N_{x_2}} & \cdots & f_{N_{x_N}} \end{bmatrix} \tag{5.5.18}$$

This algorithm relies on initially guessing values for $x_1, x_2, ..., x_N$. As before, there is no guarantee that the algorithm will converge. Thus a good initial guess is critical to its success. Further, the determinant of the Jacobian cannot equal zero, $\det \mathbf{J}(\mathbf{x_n}) \neq \mathbf{0}$, in order for the algorithm to work.

5.6 Implementing the direct solve method

To see how the finite difference discretization technique is applied, we consider the same example problem considered with the shooting method. Specifically, the boundary value problem gives rise to an eigenvalue system for which the eigenvalue and eigenfunction must both be determined. The example considered is the Schrödinger equation from quantum mechanics which is a second order

differential equation

$$\frac{d^2\psi}{dx^2} + [n(x) - \beta]\,\psi = 0 \tag{5.6.1}$$

with the boundary conditions $\psi(\pm 1) = 0$. As before, we will consider $n(x) = n_0 = 100$. Unlike the shooting method, there is no need to rewrite the equation as a coupled system of first order equations.

To discretize, we first divide the spatial domain $x \in [-1, 1]$ into a specified number of equally spaced points. At each point, let

$$\psi(x_n) = \psi_n\,. \tag{5.6.2}$$

Upon discretizing, the governing equation becomes

$$\frac{\psi_{n+1} - 2\psi_n + \psi_{n-1}}{\Delta x^2} + (n_0 - \beta)\psi_n = 0 \qquad n = 1, 2, 3, ..N-1\,. \tag{5.6.3}$$

Multiplying through by Δx^2 and collecting terms gives

$$\psi_{n+1} + \left(-2 + \Delta x^2 n_0\right)\psi_n + \psi_{n-1} = \Delta x^2 \beta \psi_n \qquad n = 1, 2, 3, ..N-1\,. \tag{5.6.4}$$

Note that the term on the right hand side is placed there because the values of β are unknown. Thus part of the problem is to determine what these values are as well as all the values of ψ_n.

Defining $\lambda = \Delta x^2 \beta$ gives the eigenvalue problem

$$\mathbf{A}\vec{\psi} = \lambda\vec{\psi} \tag{5.6.5}$$

where

$$\mathbf{A} = \begin{bmatrix} -2+\Delta x^2 n_0 & 1 & 0 & \cdots & & 0 \\ 1 & -2+\Delta x^2 n_0 & 1 & 0 & \cdots & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & & & \ddots & \ddots & 1 \\ 0 & \cdots & & 0 & 1 & -2+\Delta x^2 n_0 \end{bmatrix} \tag{5.6.6}$$

and

$$\vec{\psi} = \begin{bmatrix} \psi(x_1) \\ \psi(x_2) \\ \vdots \\ \psi(x_{N-2}) \\ \psi(x_{N-1}) \end{bmatrix}. \tag{5.6.7}$$

Thus the solution can be found by an eigenvalue solve of the linear system of equations. Note that we already know the boundary conditions $\psi(-1) = \psi_0 = 0$ and $\psi(1) = \psi_N = 0$. Thus we are left with and $N-1 \times N-1$ eigenvalue problem for which we can use the **eig** command.

numerical implementation

To implement this solution, we begin with the preliminaries of the problem

```
clear all; close all;
n=200;                % number of points
x=linespace(-1,1,n);  % linear space of points
dx=x(2)-x(1);         % delta x values
```

With these defined, we turn to building the matrix **A**. It is a sparse matrix which only has three diagonals of consequence.

```
n0=100;  % value of n_0
m=n-2;   % size of A matrix (mXm)
% DIAGONALS
for j=1:m
  A(j,j)=-2+dx^2*n0;
end
% OFF DIAGONALS
for j=1:m-1
  A(j,j+1)=1;
  A(j+1,j)=1;
end
```

Note that the size of the matrix is $m = n-2$ since we already know the boundary conditions $\psi(-1) = \psi_0 = 0$ and $\psi(1) = \psi_N = 0$..

To solve this problem, we simply use the **eig** command

```
[V,D]=eig(A)  % get the eigenvectors (V), eigenvalues (D)
```

The matrix **V** is an $m \times m$ matrix which contains as columns the eigenvectors of the matrix **A**. The matrix **D** is an $m \times m$ matrix whose diagonal elements are the eigenvalues corresponding to the eigenvectors **V**. The **eig** command automatically sorts the eigenvalues puts them in the order of smallest to biggest as you move across the columns of **V**. In this particular case, we are actually looking for the largest five values of β, thus we need to look at the last five columns of **V** for our eigenvectors and last five diagonals of **D** for the eigenvalues.

```
col=['r','b','g','c','m','k'];  % eigenfunction colors
for j=1:5
   eig(1)=0;  % left boundary condition
   eig(2:m+1)=V(:,end+1-j);  % interior points
   eig(n)=0   % right boundary condition
   norm=trapz(x,eig.^2);  % normalization
   plot(x,eig/sqrt(norm),col(j)); hold on  % plot
   beta=D(end+1-j,end+1-j)/dx/dx  % write out eigenvalues
end
```

This plots out the normalized eigenvector solutions and their eigenvalues.

6 Fourier Transforms

Fourier transforms are one of the most powerful and efficient techniques for solving a wide variety of problems. The key idea of the Fourier transform is to represent functions and their derivatives as sums of cosines and sines. This operation can be done with the Fast-Fourier transform (FFT) which is an $O(N \log N)$ operation. Thus the FFT is faster than most linear solvers of $O(N^2)$. The basic properties and implementation of the FFT will be considered here.

6.1 Basics of the Fourier Transform

Other techniques exist for solving many computational problems which are not based upon the standard Taylor series discretization. An alternative to standard discretization is to use the Fast-Fourier Transform (FFT). The FFT is an integral transform defined over the entire line $x \in [-\infty, \infty]$. Given computational practicalities, however, we transform over a finite domain $x \in [-L, L]$ and assume periodic boundary conditions due to the oscillatory behavior of the kernel of the Fourier transform. The Fourier transform and its inverse are defined as

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} f(x) dx \tag{6.1.1a}$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} F(k) dk \,. \tag{6.1.1b}$$

There are other equivalent definitions. However, this definition will serve to illustrate the power and functionality of the Fourier transform method. We again note that formally, the transform is over the entire real line $x \in [-\infty, \infty]$ whereas our computational domain is only over a finite domain $x \in [-L, L]$. Further, the Kernel of the transform, $\exp(\pm ikx)$, describes oscillatory behavior. Thus the Fourier transform is essentially an eigenfunction expansion over all continuous wavenumbers k. And once we are on a finite domain $x \in [-L, L]$, the continuous eigenfunction expansion becomes a discrete sum of eigenfunctions and associated wavenumbers (eigenvalues).

Derivative Relations

The critical property in the usage of Fourier transforms concerns derivative relations. To see how these properties are generated, we begin by considering the Fourier transform of $f'(x)$. We denote the Fourier transform of $f(x)$ as $\widehat{f(x)}$. Thus we find

$$\widehat{f'(x)} = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} f'(x) dx = f(x) e^{ikx}|_{-\infty}^{\infty} - \frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} f(x) dx \,. \tag{6.1.2}$$

Assuming that $f(x) \to 0$ as $x \to \pm\infty$ results in

$$\widehat{f'(x)} = -\frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} f(x) dx = -ik\widehat{f(x)} \,. \tag{6.1.3}$$

Thus the basic relation $\widehat{f}' = -ik\widehat{f}$ is established. It is easy to generalize this argument to an arbitrary number of derivatives. The final result is the following relation between Fourier transforms of the derivative and the Fourier transform itself

$$\widehat{f^{(n)}} = (-ik)^n \widehat{f} \,. \tag{6.1.4}$$

This property is what makes Fourier transforms so useful and practical.

As an example of the Fourier transform, consider the following differential equation

$$y'' - \omega^2 y = -f(x) \quad x \in [-\infty, \infty] \,. \tag{6.1.5}$$

We can solve this by applying the Fourier transform to both sides. This gives the following reduction

$$\begin{aligned} &\widehat{y''} - \omega^2 \widehat{y} = -\widehat{f} \\ &-k^2 \widehat{y} - \omega^2 \widehat{y} = -\widehat{f} \\ &(k^2 + \omega^2)\widehat{y} = \widehat{f} \\ &\widehat{y} = \frac{\widehat{f}}{k^2 + \omega^2} \,. \end{aligned} \tag{6.1.6}$$

To find the solution $y(x)$, we invert the last expression above to yield

$$y(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} \frac{\widehat{f}}{k^2 + \omega^2} dk \,. \tag{6.1.7}$$

This gives the solution in terms of an integral which can be evaluated analytically or numerically.

The Fast Fourier Transform

The Fast Fourier transform routine was developed specifically to perform the forward and backward Fourier transforms. In the mid 1960s, Cooley and Tukey developed what is now commonly known as the FFT algorithm [7]. Their algorithm was named one of the top ten algorithms of the 20th century for one reason: the operation count for solving a system dropped to $O(N \log N)$. For N large, this operation count grows almost linearly like N. Thus it represents a great leap forward from Gaussian elimination and LU decomposition. The key features of the FFT routine are as follows:

1. It has a low operation count: $O(N \log N)$.

2. It finds the transform on an interval $x \in [-L, L]$. Since the integration Kernel $\exp(ikx)$ is oscillatory, it implies that the solutions on this finite interval have periodic boundary conditions.

3. The key to lowering the operation count to $O(N \log N)$ is in discretizing the range $x \in [-L, L]$ into 2^n points, i.e. the number of points should be $2, 4, 8, 16, 32, 64, 128, 256, \cdots$.

4. The FFT has excellent accuracy properties, typically well beyond that of standard discretization schemes.

We will consider the underlying FFT algorithm in detail at a later time. For more information at the present, see [7] for a broader overview.

The practical implementation of the mathematical tools available in MATLAB is crucial. This lecture will focus on the use of some of the more sophisticated routines in MATLAB which are cornerstones to scientific computing. Included in this section will be a discussion of the Fast Fourier Transform routines (*fft, ifft, fftshift, ifftshift, fft2, ifft2*), sparse matrix construction (*spdiag, spy*), and high end iterative techniques for solving $\mathbf{Ax} = \mathbf{b}$ (*bicgstab, gmres*). These routines should be studied carefully since they are the building blocks of any serious scientific computing code.

Fast Fourier Transform: FFT, IFFT, FFTSHIFT, IFFTSHIFT2

The Fast Fourier Transform will be the first subject discussed. Its implementation is straightforward. Given a function which has been discretized with 2^n points and represented by a vector $\mathbf{x}$, the FFT is found with the command *fft(x)*. Aside from transforming the function, the algorithm associated with the FFT does three major things: it shifts the data so that $x \in [0, L] \to [-L, 0]$ and $x \in [-L, 0] \to [0, L]$, additionally it multiplies every other mode by -1, and it assumes you are working on a 2π periodic domain. These properties are a consequence of the FFT algorithm discussed in detail at a later time.

To see the practical implications of the FFT, we consider the transform of a Gaussian function. The transform can be calculated analytically so that we have the exact relations:

$$f(x) = \exp(-\alpha x^2) \quad \to \quad \widehat{f}(k) = \frac{1}{\sqrt{2\alpha}} \exp\left(-\frac{k^2}{4\alpha}\right). \tag{6.1.8}$$

A simple MATLAB code to verify this with $\alpha = 1$ is as follows

```
clear all; close all;  % clear all variables and figures

L=20;   % define the computational domain [-L/2,L/2]
n=128;  % define the number of Fourier modes 2^n
```

```
x2=linspace(-L/2,L/2,n+1);  % define the domain discretization
x=x2(1:n);   % consider only the first n points:  periodicity

u=exp(-x.*x);         % function to take a derivative of
ut=fft(u);            % FFT the function
utshift=fftshift(ut); % shift FFT

figure(1), plot(x,u)     % plot initial gaussian
figure(2), plot(abs(ut))      % plot unshifted transform
figure(3), plot(abs(utshift)) % plot shifted transform
```

The second figure generated by this script shows how the pulse is shifted. By using the command *fftshift*, we can shift the transformed function back to its mathematically correct positions as shown in the third figure generated. However, before inverting the transformation, it is crucial that the transform is shifted back to the form of the second figure. The command *ifftshift* does this. In general, unless you need to plot the spectrum, it is better not to deal with the *fftshift* and *ifftshift* commands. A graphical representation of the *fft* procedure and its shifting properties is illustrated in Fig. 10 where a Gaussian is transformed and shifted by the *fft* routine.

To take a derivative, we need to calculate the k values associated with the transformation. The following example does this. Recall that the FFT assumes a 2π periodic domain which gets shifted. Thus the calculation of the k values needs to shift and rescale to the 2π domain. The following example differentiates the function $f(x) = \text{sech}(x)$ three times. The first derivative is compared with the analytic value of $f'(x) = -\text{sech}(x)\tanh(x)$.

```
clear all; close all;  % clear all variables and figures

L=20;   % define the computational domain [-L/2,L/2]
n=128;  % define the number of Fourier modes 2^n

x2=linspace(-L/2,L/2,n+1);  % define the domain discretization
x=x2(1:n);   % consider only the first n points:  periodicity
u=sech(x);     % function to take a derivative of
ut=fft(u);     % FFT the function
k=(2*pi/L)*[0:(n/2-1) (-n/2):-1]; % k rescaled to 2pi domain

ut1=i*k.*ut;         % first derivative
ut2=-k.*k.*ut;       % second derivative
ut3=-i*k.*k.*k.*ut;  % third derivative

u1=ifft(ut1); u2=ifft(ut2); u3=ifft(ut3); % inverse transform
u1exact=-sech(x).*tanh(x);  % analytic first derivative
```

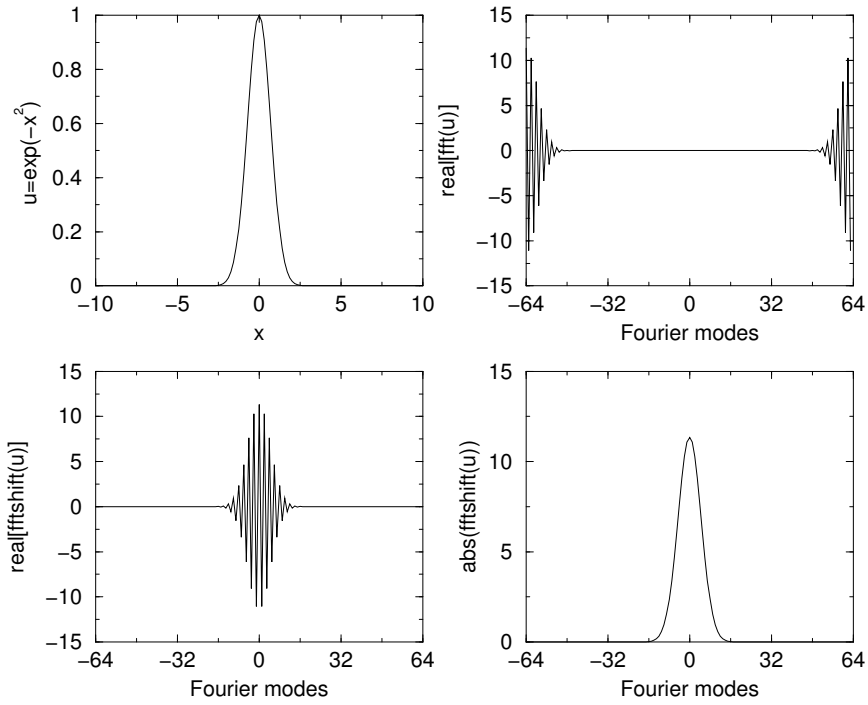


Figure 10: Fast Fourier Transform of Gaussian data illustrating the shifting properties of the FFT routine. Note that the *fftshift* command restores the transform to its mathematically correct, unshifted state.

```
figure(1)
plot(x,u,'r',x,u1,'g',x,u1exact,'go',x,u2,'b',x,u3,'c') % plot
```

The routine accounts for the periodic boundaries, the correct k values, and differentiation. Note that no shifting was necessary since we constructed the k values in the shifted space.

For transforming in higher dimensions, a couple of choices in MATLAB are possible. For 2D transformations, it is recommended to use the commands *fft2* and *ifft2*. These will transform a matrix $\mathbf{A}$, which represents data in the x and y direction respectively, along the rows and then columns. For higher dimensions, the *fft* command can be modified to *fft(x,[],N)* where N is the number of dimensions.

6.2 Spectral Analysis

Although our primary use of the FFT will be related to differentiation and solving ordinary and partial differential equations, it should be understood that the impact of the FFT is far greater than this. In particular, FFTs and other related frequency transforms have revolutionized the field of digital signal processing and imaging. The key concept in any of these applications is to use the FFT to analyze and manipulate data in the frequency domain.

This lecture will discuss a very basic concept and manipulation procedure to be performed in the frequency domain: noise attenuation via frequency (band-pass) filtering. This filtering process is fairly common in electronics and signal detection. To begin we consider a ideal signal generated in the time-domain.

```
clear all; close all;

T=20;    % Total time slot to transform
n=128;   % number of Fourier modes 2^7
t2=linspace(-T/2,T/2,n+1); t=t2(1:n);  % time values
k=(2*pi/L)*[0:(n/2-1) -n/2:-1];  % frequency components of FFT

u=sech(t);       % ideal signal in the time domain
figure(1), plot(t,u,'k'), hold on
```

This will generate an ideal hyperbolic secant shape in the time domain.

In most applications, the signals as generated above are not ideal. Rather, they have a large amount of noise on top of them. Usually this noise is what is called *white noise*, i.e. a noise which effects all frequencies the same. We can add white noise to this signal by considering the pulse in the frequency domain.

```
noise=10;
ut=fft(u);
utn=ut+noise*(rand(1,n)+i*rand(1,n));
```

These three lines of code generate the Fourier transform of the function along with the vector **utn** which is the spectrum with a complex and Gaussian distributed (mean zero, unit variance) noise source term added in.

6.3 Applications of the FFT

Spectral methods are one of the most powerful solution techniques for ordinary and partial differential equations. The best known example of a spectral method is the Fourier transform. We have already made use of the Fourier transform using FFT routines. Other spectral techniques exist which render a variety of problems easily tractable and often at significant computational savings.

Discrete Cosine and Sine Transforms

When considering a computational domain, the solution can only be found on a finite length domain. Thus the definition of the Fourier transform needs to be modified in order to account for the finite sized computational domain. Instead of expanding in terms of a continuous integral for values of wavenumber k and cosines and sines ($\exp(ikx)$), we expand in a Fourier series

$$F(k) = \sum_{n=1}^{N} f(n) \exp\left[-i\frac{2\pi(k-1)}{N}(n-1)\right] \quad 1 \le k \le N \tag{6.3.1a}$$

$$f(n) = \frac{1}{N}\sum_{k=1}^{N} F(k) \exp\left[i\frac{2\pi(k-1)}{N}(n-1)\right] \quad 1 \le n \le N\,. \tag{6.3.1b}$$

Thus the Fourier transform is nothing but an expansion in a basis of cosine and sine functions. If we define the fundamental oscillatory piece as

$$w^{nk} = \exp\left(\frac{2i\pi(k\text{–}1)(n\text{–}1)}{N}\right) = \cos\left(\frac{2\pi(k\text{–}1)(n\text{–}1)}{N}\right) + i\sin\left(\frac{2\pi(k\text{–}1)(n\text{–}1)}{N}\right), \tag{6.3.2}$$

then the Fourier transform results in the expression

$$F_n = \sum_{k=0}^{N-1} w^{nk} f_k \qquad 0 \le n \le N-1\,. \tag{6.3.3}$$

Thus the calculation of the Fourier transform involves a double sum and an $O(N^2)$ operation. Thus at first it would appear that the Fourier transform method is the same operation count as LU decomposition. The basis functions used for the Fourier transform, sine transform and cosine transform are depicted in Fig. 11. The process of solving a differential or partial differential equation involves evaluating the coefficient of each of the modes. Note that this expansion, unlike the finite difference method, is a *global* expansion in that every basis function is evaluated on the entire domain.

The Fourier, sine, and cosine transforms behave very differently at the boundaries. Specifically, the Fourier transform assumes periodic boundary conditions whereas the sine and cosine transforms assume pinned and no-flux boundaries respectively. The cosine and sine transform are often chosen for their boundary properties. Thus for a given problem, an evaluation must be made of the type of transform to be used based upon the boundary conditions needing to be satisfied. Table 8 illustrates the three different expansions and their associated boundary conditions. The appropriate MATLAB command is also given.

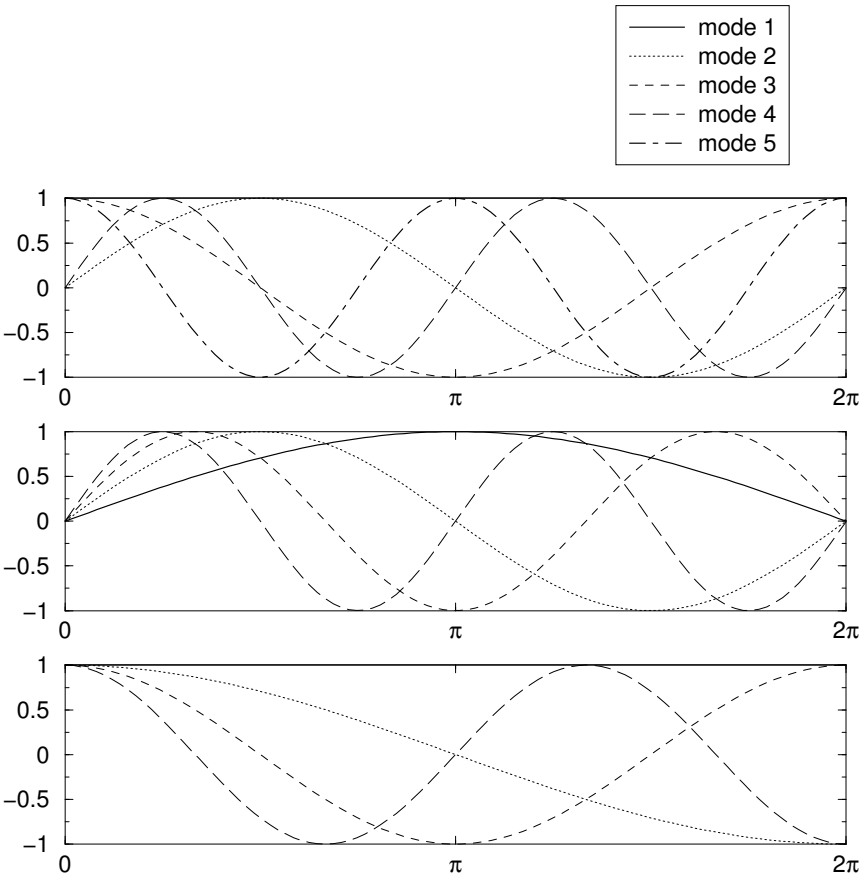


Figure 11: Basis functions used for a Fourier mode expansion (top), a sine expansion (middle), and a cosine expansion (bottom).

Fast-Poisson Solvers

The FFT algorithm provides a fast and efficient method for solving the Poisson equation

$$\nabla^2 \psi = \omega \tag{6.3.4}$$

given the function ω. In two dimensions, this equation is equivalent to

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = \omega \,. \tag{6.3.5}$$

By discretizing in both the x and y directions, this forces us to solve an associated linear problem $\mathbf{Ax} = \mathbf{b}$. At best, we can use a factorization scheme to solve this problem in $O(N^2)$ operations. Although iteration schemes have

command	expansion	boundary conditions
fft	$F_k = \sum_{j=0}^{2N-1} f_j \exp(i\pi jk/N)$	periodic: $f(0) = f(L)$
dst	$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N)$	pinned: $f(0) = f(L) = 0$
dct	$F_k = \sum_{j=0}^{N-2} f_j \cos(\pi jk/2N)$	no-flux: $f'(0) = f'(L) = 0$

Table 8: MATLAB functions for Fourier, sine, and cosine transforms and their associated boundary conditions. To invert the expansions, the MATLAB commands are *ifft, idst,* and *idct* repectively.

the possibility of outperforming this, it is not guaranteed. Fourier transform methods allow us to solve the problem in $O(N \log N)$.

Denoting the Fourier transform in x as $\widehat{f(x)}$ and the Fourier transform in y as $\widetilde{g(y)}$, we transform the equation. We begin by transforming in x:

$$\widehat{\frac{\partial^2 \psi}{\partial x^2}} + \widehat{\frac{\partial^2 \psi}{\partial y^2}} = \widehat{\omega} \quad \rightarrow \quad -k_x^2 \widehat{\psi} + \frac{\partial^2 \widehat{\psi}}{\partial y^2} = \widehat{\omega}\,, \tag{6.3.6}$$

where k_x are the wavenumbers in the x direction. Transforming now in the y direction gives

$$-k_x^2 \widetilde{\widehat{\psi}} + \widetilde{\frac{\partial^2 \widehat{\psi}}{\partial y^2}} = \widetilde{\widehat{\omega}} \quad \rightarrow \quad -k_x^2 \widetilde{\widehat{\psi}} + -k_y^2 \widetilde{\widehat{\psi}} = \widetilde{\widehat{\omega}}\,. \tag{6.3.7}$$

This can be rearranged to obtain the final result

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2}\,. \tag{6.3.8}$$

The remaining step is to inverse transform in x and y to get back to the solution $\psi(x, y)$.

The algorithm to solve this is surprisingly simple. Before generating the solution, the spatial domain, Fourier wavenumber vectors, and two-dimensional function $\omega(x, y)$ are defined.

```
Lx=20; Ly=20, nx=128; ny=128;  % domains and Fourier modes
x2=linspace(-Lx/2,Lx/2,nx+1); x=x2(1:nx);  % x-values
y2=linspace(-Ly/2,Ly/2,ny+1); y=y2(1:nx);  % y-values
[X,Y]=meshgrid(x,y);    % generate 2-D grid
omega=exp(-X.^2-Y.^2);  % generate 2-D Gaussian
```

After defining the preliminaries, the wavenumbers are generated and the equation is solved

```
kx=(2*pi/Lx)*[0:nx/2-1 -nx/2:-1];   % x-wavenumbers
ky=(2*pi/Ly)*[0:ny/2-1 -ny/2:-1];   % y-wavenumbers
kx(1)=10^(-6); ky(1)=10^(-6);       % fix divide by zero
[KX,KY]=meshgrid(kx,ky);  % generate 2-D wavenumbers
psi=real( ifft2( -fft2(omega)./(KX.^2+KY.^2) ) );  % solution
```

Note that the real part of the inverse 2-D FFT is taken since numerical round-off generates imaginary parts to the solution $\psi(x, y)$.

An observation concerning (6.3.8) is that there will be a divide by zero when $k_x = k_y = 0$ at the zero mode. Two options are commonly used to overcome this problem. The first is to modify (6.3.8) so that

$$\widetilde{\widetilde{\psi}} = -\frac{\widetilde{\widetilde{\omega}}}{k_x^2 + k_y^2 + eps} \tag{6.3.9}$$

where *eps* is the command for generating a machine precision number which is on the order of $O(10^{-15})$. It essentially adds a round-off to the denominator which removes the divide by zero problem. A second option, which is more highly recommended, is to redefine the $\mathbf{k}_x$ and $\mathbf{k}_y$ vectors associated with the wavenumbers in the x and y directions. Specifically, after defining the $\mathbf{k}_x$ and $\mathbf{k}_y$, we could simply add the command line

```
kx(1)=10^(-6);
ky(1)=10^(-6);
```

The values of $kx(1) = ky(1) = 0$ by default. This would make the values small but finite so that the divide by zero problem is effectively removed with only a small amount of error added to the problem.

There is a second mathematical difficulty which must be addressed. The function $\psi(x, y)$ with periodic boundary conditions does not have a unique solution. Thus if $\psi_0(x, y)$ is a solution, so is $\psi_0(x, y) + c$ where c is an arbitrary constant. When solving this problem with FFTs, the FFT will arbitrarily add a constant to the solution. Fundamentally, we are only interested in derivatives of the streamfunction. Therefore, this constant is inconsequential. When solving with direct methods for $\mathbf{Ax} = \mathbf{b}$, the non-uniqueness gives a singular matrix $\mathbf{A}$. Thus solving with Gaussian elimination, LU decomposition or iterative methods is problematic. Often in applications the arbitrary constant does not matter. Specifically, most applications only are concerned with the derivative of the function $\psi(x, y)$. Thus we can simply pin the value of $\psi(x, y)$ to some prescribed value on our computational domain. This will fix the constant c and give a unique solution to $\mathbf{Ax} = \mathbf{b}$. For instance, we could impose the following constraint condition $\psi(-L, -L, t) = 0$. Such a condition pins the value of $\psi(x, y, t)$ at the left hand corner of the computational domain and fixes c.

7 Partial Differential Equations

With the Fourier transform and discretization in hand, we can turn towards the solution of partial differential equations whose solutions need to be advanced forward in time. Thus, in addition to spatial discretization, we will need to discretize in time in a self-consistent way so as to advance the solution to a desired future time. Issues of stability and accuracy are, of course, at the heart of a discussion on time and space stepping schemes.

7.1 Basic time- and space-stepping schemes

Basic time-stepping schemes involve discretization in both space and time. Issues of numerical stability as the solution is propagated in time and accuracy from the space-time discretization are of greatest concern for any implementation. To begin this study, we will consider very simple and well known examples from partial differential equations. The first equation we consider is the heat (diffusion) equation

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \tag{7.1.1}$$

with the periodic boundary conditions $u(-L) = u(L)$. We discretize the spatial derivative with a second-order scheme (see Table 4) so that

$$\frac{\partial u}{\partial t} = \frac{\kappa}{\Delta x^2}\left[u(x+\Delta x) - 2u(x) + u(x-\Delta x)\right] . \tag{7.1.2}$$

This approximation reduces the partial differential equation to a system of ordinary differential equations. We have already considered a variety of time-stepping schemes for differential equations and we can apply them directly to this resulting system.

The ODE System

To define the system of ODEs, we discretize and define the values of the vector $\mathbf{u}$ in the following way.

$$\begin{aligned} &u(-L) = u_1 \\ &u(-L+\Delta x) = u_2 \\ &\vdots \\ &u(L-2\Delta x) = u_{n-1} \\ &u(L-\Delta x) = u_n \\ &u(L) = u_{n+1} . \end{aligned}$$

Recall from periodicity that $u_1 = u_{n+1}$. Thus our system of differential equations solves for the vector

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}. \tag{7.1.3}$$

The governing equation (7.1.2) is then reformulated as the differential equations system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\Delta x^2}\mathbf{A}\mathbf{u}, \tag{7.1.4}$$

where $\mathbf{A}$ is given by the sparse matrix

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}, \tag{7.1.5}$$

and the values of one on the upper right and lower left of the matrix result from the periodic boundary conditions.

MATLAB implementation

The system of differential equations can now be easily solved with a standard time-stepping algorithm such as *ode23* or *ode45*. The basic algorithm would be as follows

1. Build the sparse matrix $\mathbf{A}$.

```
e1=ones(n,1);  % build a vector of ones
A=spdiags([e1 -2*e1 e1],[-1 0 1],n,n);  % diagonals
A(1,n)=1;  A(n,1)=1;  % periodic boundaries
```

2. Generate the desired initial condition vector $\mathbf{u} = \mathbf{u}_0$.

3. Call an ODE solver from the MATLAB suite. The matrix $\mathbf{A}$, the diffusion constant κ and spatial step Δx need to be passed into this routine.

```
[t,y]=ode45('rhs',tspan,u0,[],k,dx,A);
```

The function *rhs.m* should be of the following form

```
function rhs=rhs(tspan,u,dummy,k,dx,A)
rhs=(k/dx^2)*A*u;
```

4. Plot the results as a function of time and space.

The algorithm is thus fairly routine and requires very little effort in programming since we can make use of the standard time-stepping algorithms already available in MATLAB.

2D MATLAB implementation

In the case of two dimensions, the calculation becomes slightly more difficult since the 2D data is represented by a matrix and the ODE solvers require a vector input for the initial data. For this case, the governing equation is

$$\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{7.1.6}$$

where we discretize in x and y. Provided $\Delta x = \Delta y = \delta$ are the same, the system can be reduced to the linear system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\delta^2} \mathbf{B}\mathbf{u}\,, \tag{7.1.7}$$

where we have arranged the vector $\mathbf{u}$ so that

$$\mathbf{u} = \begin{pmatrix} u_{11} \\ u_{12} \\ \vdots \\ u_{1n} \\ u_{21} \\ u_{22} \\ \vdots \\ u_{n(n-1)} \\ u_{nn} \end{pmatrix}, \tag{7.1.8}$$

where we have defined $u_{jk} = u(x_j, y_k)$. The matrix $\mathbf{B}$ is a nine diagonal matrix. The **kron** command can be used to generate it from its 1D version.

Again, the system of differential equations can now be easily solved with a standard time-stepping algorithm such as *ode23* or *ode45*. The basic algorithm follows the same course as the 1D case, but extra care is taken in arranging the 2D data into a vector.

1. Build the sparse matrix **B**. Here the matrix **B** can be built by using the fact that we know the differentiation matrix in one dimension, i.e. the matrix **A** of the previous example.

```
n=100;    % number of discretization points in x and y
I=eye(n); % idendity matrix of size nXn
B=kron(I,A)+kron(A,I); % 2-D differentiation matrix
```

2. Generate the desired initial condition matrix $\mathbf{U} = \mathbf{U}_0$ and reshape it to a vector $\mathbf{u} = \mathbf{u}_0$. This example considers the case of a simple Gaussian as the initial condition. The *reshape* and *meshgrid* commands are important for computational implementation.

```
Lx=20;   Ly=20;  % spatial domain of x and y
nx=100;  ny=100; % discretization points in x and y
N=nx*ny; % elements in reshaped initial condition

x2=linspace(-Lx/2,Lx/2,nx+1); % account for periodicity
x=x2(1:nx);      % x-vector
y2=linspace(-Ly/2,Ly/2,ny+1); % account for periodicity
y=y2(1:ny);      % y-vector

[X,Y]=meshgrid(x,y);  % set up for 2D initial conditions
U=exp(-X.^2-Y.^2);    % generate a Gaussian matrix
u=reshape(U,N,1);     % reshape into a vector
```

3. Call an ODE solver from the MATLAB suite. The matrix **A**, the diffusion constant κ and spatial step $\Delta x = \Delta y = dx$ need to be passed into this routine.

```
[t,y]=ode45('rhs',tspan,u0,[],k,dx,A);
```

   The function *rhs.m* should be of the following form

```
function rhs=rhs(tspan,u,dummy,k,dx,A)
rhs=(k/dx^2)*A*u;
```

4. Plot the results as a function of time and space.

The algorithm is again fairly routine and requires very little effort in programming since we can make use of the standard time-stepping algorithms.

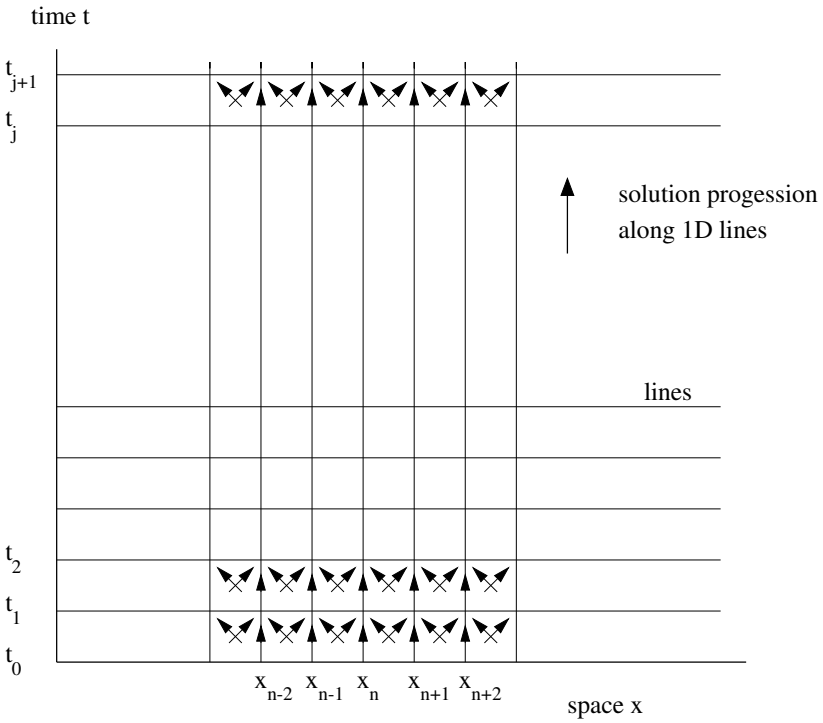


Figure 12: Graphical representation of the progression of the numerical solution using the method of lines. Here a second order discretization scheme is considered which couples each spatial point with its nearest neighbor.

Method of Lines

Fundamentally, these methods use the data at a single slice of time to generate a solution Δt in the future. This is then used to generate a solution $2\Delta t$ into the future. The process continues until the desired future time is achieved. This process of solving a partial differential equation is known as the *method of lines*. Each *line* is the value of the solution at a given time slice. The lines are used to update the solution to a new timeline and progressively generate future solutions. Figure 12 depicts the process involved in the method of lines for the 1D case.

7.2 Time-stepping schemes: explicit and implicit methods

Now that several technical and computational details have been addressed, we continue to develop methods for time-stepping the solution into the future. Some of the more common schemes will be considered along with a graphical

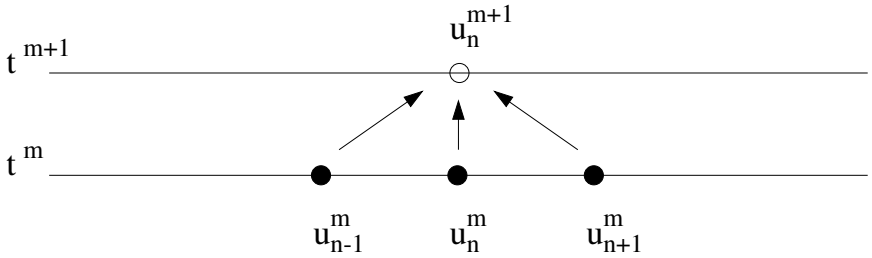


Figure 13: Four-point stencil for second-order spatial discretization and Euler time-stepping of the one-wave wave equation.

representation of the scheme. Every scheme eventually leads to an iteration procedure which the computer can use to advance the solution in time.

We will begin by considering the most simple partial differential equations. Often, it is difficult to do much analysis with complicated equations. Therefore, considering simple equations is not merely an excersize, but rather they are typically the only equations we can make analytical progress with.

As an example, we consider the one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}\,. \tag{7.2.1}$$

The simplest discretization of this equation is to first central difference in the x direction. This yields

$$\frac{\partial u}{\partial t} = \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right)\,, \tag{7.2.2}$$

where $u_n = u(x_n, t)$. We can then step forward with an Euler time-stepping method. Denoting $u_n^{(m)} = u(x_n, t_m)$, and applying the method of lines iteration scheme gives

$$u_n^{(m+1)} = u_n^{(m)} + \frac{c\Delta t}{2\Delta x}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right)\,. \tag{7.2.3}$$

This simple scheme has the four-point stencil shown in Fig. 13. To illustrate more clearly the iteration procedure, we rewrite the discretized equation in the form

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{7.2.4}$$

where

$$\lambda = \frac{c\Delta t}{\Delta x} \tag{7.2.5}$$

is known as the CFL (Courant, Friedrichs, and Levy) condition. The iteration procedure assumes that the solution does not change significantly from one time-step to the next, i.e. $u_n^{(m)} \approx u_n^{(m+1)}$. The accuracy and stability of this scheme

is controlled almost exclusively by the CFL number λ. This parameter relates the spatial and time discretization schemes in (7.2.5). Note that decreasing Δx without decreasing Δt leads to an increase in λ which can result in instabilities. Smaller values of Δt suggest smaller values of λ and improved numerical stability properties. In practice, you want to take Δx and Δt as large as possible for computational speed and efficiency without generating instabilities.

There are practical considerations to keep in mind relating to the CFL number. First, given a spatial discretization step-size Δx, you should choose the time discretization so that the CFL number is kept in check. Often a given scheme will only work for CFL conditions below a certain value, thus the importance of choosing a small enough time-step. Second, if indeed you choose to work with very small Δt, then although stability properties are improved with a lower CFL number, the code will also slow down accordingly. Thus achieving good stability results is often counter-productive to fast numerical solutions.

Central Differencing in Time

We can discretize the time-step in a similar fashion to the spatial discretization. Instead of the Euler time-stepping scheme used above, we could central difference in time using Table 4. Thus after spatial discretization we have

$$\frac{\partial u}{\partial t} = \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right), \tag{7.2.6}$$

as before. And using a central difference scheme in time now yields

$$\frac{u_n^{(m+1)} - u_n^{(m-1)}}{2\Delta t} = \frac{c}{2\Delta x}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right). \tag{7.2.7}$$

This last expression can be rearranged to give

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right). \tag{7.2.8}$$

This iterative scheme is called *leap-frog (2,2)* since it is $O(\Delta t^2)$ accurate in time and $O(\Delta x^2)$ accurate in space. It uses a four point stencil as shown in Fig. 14. Note that the solution utilizes two time slices to leap-frog to the next time slice. Thus the scheme is not self-starting since only one time slice (initial condition) is given.

Improved Accuracy

We can improve the accuracy of any of the above schemes by using higher order central differencing methods. The fourth-order accurate scheme from Table 5 gives

$$\frac{\partial u}{\partial x} = \frac{-u(x+2\Delta x) + 8u(x+\Delta x) - 8u(x-\Delta x) + u(x-2\Delta x)}{12\Delta x}. \tag{7.2.9}$$

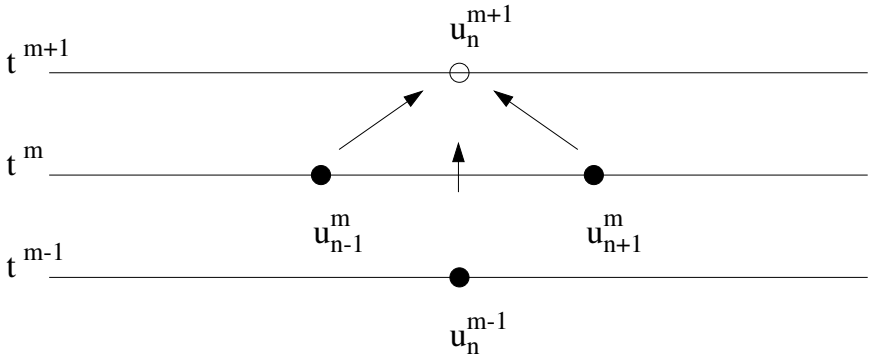


Figure 14: Four-point stencil for second-order spatial discretization and central-difference time-stepping of the one-wave wave equation.

Combining this fourth-order spatial scheme with second-order central differencing in time gives the iterative scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda \left[\frac{4}{3} \left(u_{n+1}^{(m)} - u_{n-1}^{(m)} \right) - \frac{1}{6} \left(u_{n+2}^{(m)} - u_{n-2}^{(m)} \right) \right] . \qquad (7.2.10)$$

This scheme, which is based upon a six point stencil, is called *leap frog (2,4)*. It is typical that for *(2,4)* schemes, the maximum CFL number for stable computations is reduced from the basic *(2,2)* scheme.

Lax-Wendroff

Another alternative to discretizing in time and space involves a clever use of the Taylor expansion

$$u(x, t + \Delta t) = u(x, t) + \Delta t \frac{\partial u(x,t)}{\partial t} + \frac{\Delta t^2}{2!} \frac{\partial^2 u(x,t)}{\partial t^2} + O(\Delta t^3) . \qquad (7.2.11)$$

But we note from the governing one-way wave equation

$$\frac{\partial u}{\partial t} = c \frac{\partial u}{\partial x} \approx \frac{c}{2\Delta x} \left(u_{n+1} - u_{n-1} \right) . \qquad (7.2.12a)$$

Taking the derivative of the equation results in the relation

$$\frac{\partial^2 u}{\partial t^2} = c \frac{\partial^2 u}{\partial x^2} \approx \frac{c}{\Delta x^2} \left(u_{n+1} - 2u_n + u_{n-1} \right) . \qquad (7.2.13a)$$

These two expressions for $\partial u/\partial t$ and $\partial^2 u/\partial t^2$ can be substituted into the Taylor series expression to yield the iterative scheme

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2} \left(u_{n+1}^{(m)} - u_{n-1}^{(m)} \right) + \frac{\lambda^2}{2} \left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)} \right) . \qquad (7.2.14)$$

Scheme	Stability
Forward Euler	unstable for all λ
Backward Euler	stable for all λ
Leap Frog (2,2)	stable for $\lambda \leq 1$
Leap Frog (2,4)	stable for $\lambda \leq 0.707$

Table 9: Stability of time-stepping schemes as a function of the CFL number.

This iterative scheme is similar to the Euler method. However, it introduces an important *stabilizing* diffusion term which is proportional to λ^2. This is known as the *Lax-Wendroff* scheme. Although useful for this example, it is difficult to implement in practice for variable coefficient problems. It illustrates, however, the variety and creativity in developing iteration schemes for advancing the solution in time and space.

Backward Euler: Implicit Scheme

The backward Euler method uses the future time for discretizing the spatial domain. Thus upon discretizing in space and time we arrive at the iteration scheme

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m+1)} - u_{n-1}^{(m+1)}\right). \tag{7.2.15}$$

This gives the tridiagonal system

$$u_n^{(m)} = -\frac{\lambda}{2}u_{n+1}^{(m+1)} + u_n^{(m+1)} + \frac{\lambda}{2}u_{n-1}^{(m+1)}, \tag{7.2.16}$$

which can be written in matrix form as

$$\mathbf{A}\mathbf{u}^{(m+1)} = \mathbf{u}^{(m)} \tag{7.2.17}$$

where

$$\mathbf{A} = \frac{1}{2}\begin{pmatrix} 2 & -\lambda & \cdots & 0 \\ \lambda & 2 & -\lambda & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \lambda & 2 \end{pmatrix}. \tag{7.2.18}$$

Thus before stepping forward in time, we must solve a matrix problem. This can severely affect the computational time of a given scheme. The only thing which may make this method viable is if the CFL condition is such that much larger time-steps are allowed, thus overcoming the limitations imposed by the matrix solve.

MacCormack Scheme

In the MacCormack Scheme, the variable coefficient problem of the Lax-Wendroff scheme and the matrix solve associated with the backward Euler are circumvented by using a predictor-corrector method. The computation thus occurs in two pieces:

$$u_n^{(P)} = u_n^{(m)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{7.2.19a}$$

$$u_n^{(m+1)} = \frac{1}{2}\left[u_n^{(m)} + u_n^{(P)} + \lambda\left(u_{n+1}^{(P)} - u_{n-1}^{(P)}\right)\right]. \tag{7.2.19b}$$

This method essentially combines forward and backward Euler schemes so that we avoid the matrix solve and the variable coefficient problem.

The CFL condition will be discussed in detail in the next section. For now, the basic stability properties of many of the schemes considered here are given in Table 9. The stability of the various schemes hold only for the one-way wave equation considered here as a prototypical example. Each partial differential equation needs to be considered and classified individually with regards to stability.

7.3 Implementing Time- and Space-Stepping Schemes

Computational performance is always crucial in choosing a numerical scheme. Speed, accuracy and stability all play key roles in determining the appropriate choice of method for solving. We will consider three prototypical equations:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} \qquad \text{one-way wave equation} \tag{7.3.1a}$$

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \qquad \text{diffusion equation} \tag{7.3.1b}$$

$$i\frac{\partial u}{\partial t} = \frac{1}{2}\frac{\partial^2 u}{\partial x^2} + |u|^2 u \qquad \text{nonlinear Schrödinger equation}. \tag{7.3.1c}$$

Periodic boundary conditions will be assumed in each case. The purpose of this section is to build a numerical routine which will solve these problems using the iteration procedures outlined in the previous two sections. Of specific interest will be the setting of the CFL number and the consequences of violating the stability criteria associated with it.

The equations are considered in one dimension such that the first and second

derivative are given by

$$\frac{\partial u}{\partial x} \rightarrow \frac{1}{2\Delta x}\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 & -1 \\ -1 & 0 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & -1 & 0 & 1 \\ 1 & 0 & \cdots & 0 & -1 & 0 \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} \tag{7.3.2}$$

and

$$\frac{\partial^2 u}{\partial x^2} \rightarrow \frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}. \tag{7.3.3}$$

From the previous lectures, we have the following discretization schemes for the one-way wave equation (7.3.1):

$$\text{Euler (unstable):} \qquad u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{7.3.4a}$$

$$\text{leap-frog (2,2) (stable for } \lambda \leq 1\text{):} \quad u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{7.3.4b}$$

where the CFL number is given by $\lambda = \Delta t/\Delta x$. Similarly for the diffusion equation (7.3.1)

$$\text{Euler (stable for } \lambda \leq 1/2\text{):} \quad u_n^{(m+1)} = u_n^{(m)} + \lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) \tag{7.3.5a}$$

$$\text{leap-frog (2,2) (unstable):} \quad u_n^{(m+1)} = u_n^{(m-1)} + 2\lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) \tag{7.3.5b}$$

where now the CFL number is given by $\lambda = \Delta t/\Delta x^2$. The nonlinear Schrödinger equation discretizes to the following form:

$$\frac{\partial u_n^{(m)}}{\partial t} = -\frac{i}{2\Delta x^2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) - i|u_n^{(m)}|^2 u_n^{(m)}. \tag{7.3.6}$$

The Euler and leap-frog (2,2) time-stepping schemes will be explored with this equation.

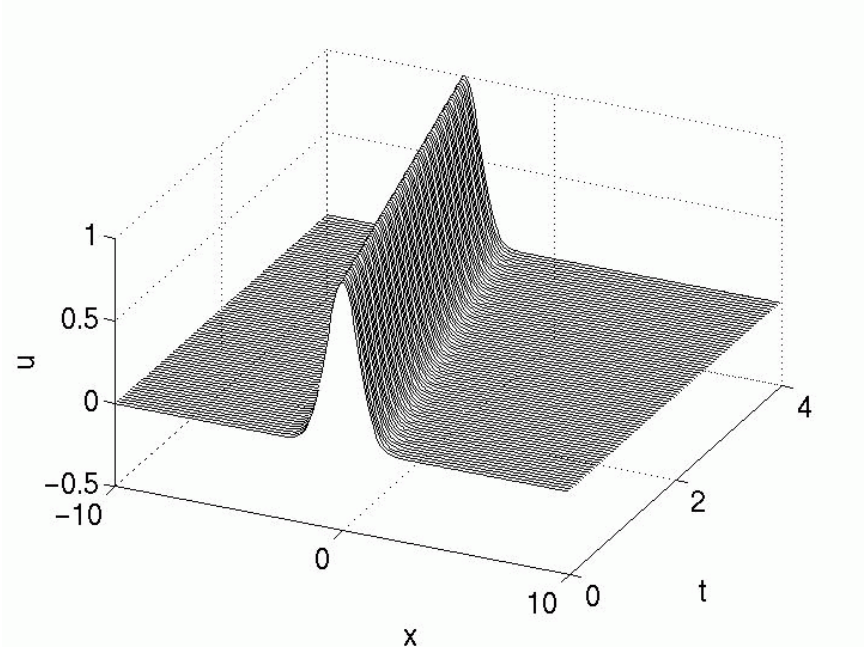


Figure 15: Evolution of the one-way wave equation with the leap-frog (2,2) scheme and with CFL=0.5. The stable traveling wave solution is propagated in this case to the left.

One-Way Wave Equation

We first consider the leap-frog (2,2) scheme applied to the one-way wave equation. Figure 15 depicts the evolution of an initial Gaussian pulse. For this case, the CFL=0.5 so that stable evolution is analytically predicted. The solution propagates to the left as expected from the exact solution. The leap-frog (2,2) scheme becomes unstable for $\lambda \geq 1$ and the system is always unstable for the Euler time-stepping scheme. Figure 16 depicts the unstable evolution of the leap-frog (2,2) scheme with CFL=2 and the Euler time-stepping scheme. The initial conditions used are identical to that in Fig. 15. Since we have predicted that the leap-frog numerical scheme is only stable provided $\lambda < 1$, it is not surprising that the figure on the left goes unstable. Likewise, the figure on the right shows the numerical instability generated in the Euler scheme. Note that both of these unstable evolutions develop high frequency oscillations which eventually blow up. The MATLAB code used to generate the leap-frog and Euler iterative solutions is given by

```
clear all; close all;  % clear previous figures and values
```

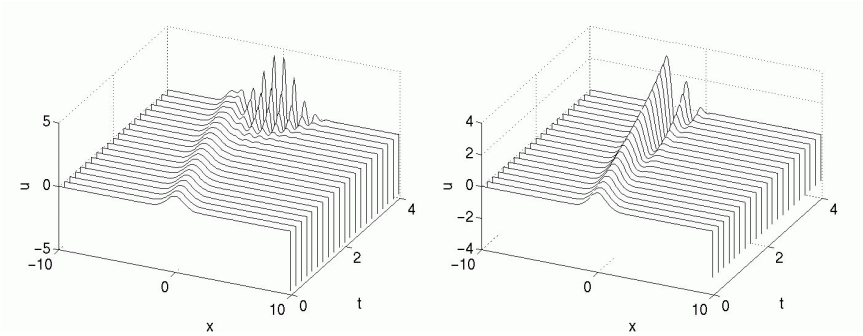


Figure 16: Evolution of the one-way wave equation using the leap-frog (2,2) scheme with CFL=2 (left) along with the Euler time-stepping scheme (right). The analysis predicts stable evolution of leap-frog provided the CFL≤ 1. Thus the onset of numerical instability near $t \approx 3$ for the CFL=2 case is not surprising. Likewise, the Euler scheme is expected to be unstable for all CFL.

```
% initialize grid size, time, and CFL number
Time=4;
L=20;
n=200;
x2=linspace(-L/2,L/2,n+1);
x=x2(1:n);
dx=x(2)-x(1);
dt=0.2;
CFL=dt/dx
time_steps=Time/dt;
t=0:dt:Time;

% initial conditions
u0=exp(-x.^2)';
u1=exp(-(x+dt).^2)';
usol(:,1)=u0;
usol(:,2)=u1;

% sparse matrix for derivative term
e1=ones(n,1);
A=spdiags([-e1  e1],[-1 1],n,n);
A(1,n)=-1;  A(n,1)=1;

% leap frog (2,2) or euler iteration scheme
for j=1:time_steps-1
```

```
%  u2 = u0 + CFL*A*u1;  % leap frog (2,2)
%  u0 = u1; u1 = u2;    % leap frog (2,2)
  u2 = u1 + 0.5*CFL*A*u1;  % euler
  u1 = u2;                 % euler
  usol(:,j+2)=u2;
end

% plot the data
waterfall(x,t,usol');
map=[0 0 0];
colormap(map);

% set x and y limits and fontsize
set(gca,'Xlim',[-L/2 L/2],'Xtick',[-L/2 0 L/2],'FontSize',[20]);
set(gca,'Ylim',[0 Time],'ytick',[0 Time/2 Time],'FontSize',[20]);
view(25,40)

% set axis labels and fonts
xl=xlabel('x'); yl=ylabel('t'); zl=zlabel('u');
set(xl,'FontSize',[20]);set(yl,'FontSize',[20]);set(zl,'FontSize',[20]);

print -djpeg -r0 fig.jpg % print jpeg at screen resolution
```

Heat Equation

In a similar fashion, we investigate the evolution of the diffusion equation when the space-time discretization is given by the leap-frog (2,2) scheme or Euler stepping. Figure 17 shows the expected diffusion behavior for the stable Euler scheme ($\lambda \leq 0.5$). In contrast, Fig. 18 shows the numerical instabilities which are generated from violating the CFL constraint for the Euler scheme or using the always unstable leap-frog (2,2) scheme for the diffusion equation. The numerical code used to generate these solutions follows that given previously for the one-way wave equation. However, the sparse matrix is now given by

```
% sparse matrix for second derivative term
e1=ones(n,1);
A=spdiags([e1 -2*e1 e1],[-1 0 1],n,n);
A(1,n)=1;  A(n,1)=1;
```

Further, the iterative process is now

```
% leap frog (2,2) or euler iteration scheme
for j=1:time_steps-1
  u2 = u0 + 2*CFL*A*u1;  % leap frog (2,2)
```

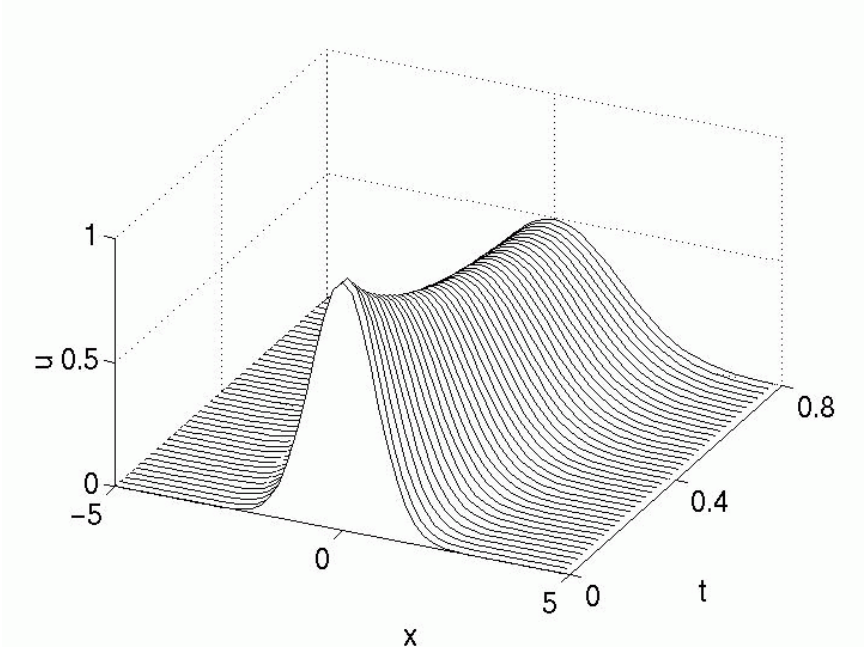


Figure 17: Stable evolution of the heat equation with the Euler scheme with CFL=0.5. The initial Gaussian is diffused in this case.

```
  u0 = u1; u1 = u2;      % leap frog (2,2)
%  u2 = u1 + CFL*A*u1;  % euler
%  u1 = u2;             % euler
  usol(:,j+2)=u2;
end
```

where we recall that the CFL condition is now given by $\lambda = \Delta t/\Delta x^2$, i.e.

```
CFL=dt/dx/dx
```

This solves the one-dimensional heat equation with periodic boundary conditions.

Nonlinear Schrödinger Equation

The nonlinear Schrödinger equation can easily be discretized by the above techniques. However, as with most nonlinear equations, it is a bit more difficult to perform a von Neumann analysis. Therefore, we explore the behavior for this system for two different discretization schemes: Euler and leap-frog (2,2). The CFL number will be the same with both schemes ($\lambda = 0.05$) and the stability

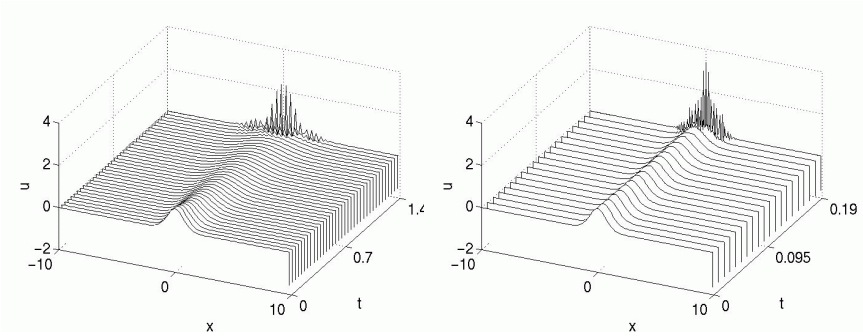


Figure 18: Evolution of the heat equation with the Euler time-stepping scheme (left) and leap-frog (2,2) scheme (right) with CFL=1. The analysis predicts that both these schemes are unstable. Thus the onset of numerical instability is observed.

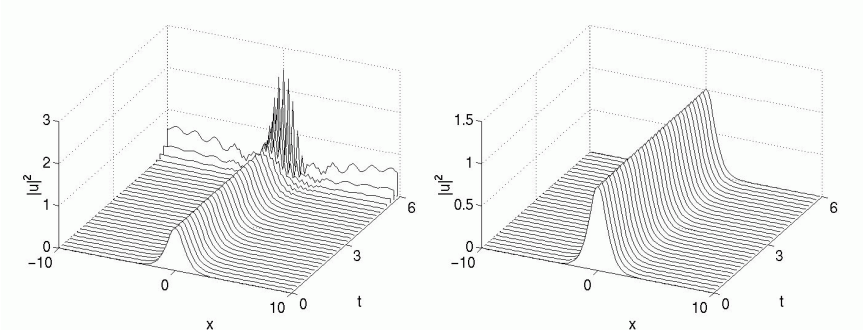


Figure 19: Evolution of the nonlinear Schrödinger equation with the Euler time-stepping scheme (left) and leap-frog (2,2) scheme (right) with CFL=0.05.

will be investigated through numerical computations. Figure 19 shows the evolution of the exact one-soliton solution of the nonlinear Schrödinger equation ($u(x,0) = \mathrm{sech}(x)$) over six units of time. The Euler scheme is observed to lead to numerical instability whereas the leap-frog (2,2) scheme is stable. In general, the leap-frog schemes work well for wave propagation problems while Euler methods are better for problems of a diffusive nature.

The MATLAB code modifications necessary to solve the nonlinear Schrödinger equation are trivial. Specifically, the iteration scheme requires change. For the stable leap-frog scheme, the following command structure is required

```
u2 = u0 + -i*CFL*A*u1- i*2*dt*(conj(u1).*u1).*u1;
u0 = u1; u1 = u2;
```

Note that i is automatically defined in MATLAB as $i = \sqrt{-1}$. Thus it is imperative that you do not use the variable i as a counter in your *FOR* loops. You will solve a very different equation if not careful with this definition.

References

[1] W. E. Boyce and R. C. DiPrima, *Elementary Differential Equations and Boundary Value Problems*, 7th Ed. (Wiley, 2001).

[2] See, for instance, R. Finney, F. Giordano, G. Thomas, and M. Weir, *Calculus and Analytic Geometry*, 10th Ed. (Prentice Hall, 2000).

[3] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, (Prentice Hall, 1971).

[4] J. D. Lambert, *Computational Methods in Ordinary Differential Equations*, (Wiley, 1973)

[5] R. L. Burden and J. D. Faires, *Numerical Analysis*, (Brooks/Cole, 1997).

[6] A. Greenbaum, *Iterative Methods for Solving Linear Systems*, (SIAM, 1997).

[7] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes*, 2nd Ed. (Cambridge, 1992).

[8] R. Courant, K. O. Friedrichs, and H. Lewy, "Über die partiellen differenzengleichungen der mathematischen physik," *Mathematische Annalen* **100**, 32-74 (1928).

[9] J. C. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*, (Chapman & Hall, 1989).

[10] N. L. Trefethen, *Spectral methods in MATLAB*, (SIAM, 2000).

[11] G. Strang, *Introduction to Applied Mathematics*, (Wellesley-Cambridge Press, 1986).

[12] I. M. Gelfand and S. V. Fomin, *Calculus of Variations*, (Prentice-Hall, 1963).