Methods of Computational Physics
Notes for PHSX 331

Dana W. Longcope and Carla M. Riedel
Department of Physics
Montana State University
Bozeman, MT

June 27, 2011

Contents

Preface **v**

List of Programs **vii**

1 Algorithms and Programs **1**
1.1 An example 1

2 Algebraic Equations — Root Finding **9**
2.1 The general problem 9
2.2 Method 1: Bisection 11
2.3 Method 2: Newton's method 19
2.4 Comparison — which one should I use? 22

3 Root Finding in More Dimensions **25**
3.1 Vector-valued functions 25
3.2 Root finding with vector-valued functions 28

4 Solving an Ordinary Differential Equation **33**
4.1 The general problem 33
4.2 Time stepping — Euler's method 34
4.3 Truncation errors and order of accuracy 38
4.4 A second algorithm — The Runge-Kutta method 41

5 Sets of ODEs — Multiple Dimensions **47**
5.1 First order ODEs 47
5.2 High order ODEs 52
5.3 Finding specific results — Driver programs 54

6 Eigenvalue Problems **61**
6.1 The general problem 61
6.2 Analytical solution: Case I 62
6.3 Numerical solution — The Shooting Method 63
6.4 Harder problems: case II 68

7 Least-Squares Fitting of Data **71**
7.1 The General Problem 71
7.2 Method of Maximum Likelihood 71
7.3 Least-Squares Fitting to a Straight Line 72

7.4 Goodness of Fit 74
7.5 Least-Squares Linear Fitting Program 77
7.6 Other Methods 78

8 Statistical Measures **79**
8.1 Moments 79
8.2 Probability Distributions 82
8.3 Numerical Modeling 84

A Programming Tips **87**

B Precision & Round-off **91**
B.1 Integer storage 91
B.2 Floating point numbers 92

C Using Arrays in `Matlab` **97**

Preface

This is the second edition of the primary text that accompanies the course *Methods of Computational Physics* (PHSX 331) at Montana State University. PHSX 331 is a one-credit course taken by students who have already successfully completed three semesters of introductory calculus, three semesters of introductory physics, and one semester of ordinary differential equations. The formal pre-requisite for PHSX 331 is successful completion of *Introduction to Theoretical Physics* (PHSX 301), which is a broad survey of mathematical techniques that are useful in physics. Of special relevance to students in PHSX 331 are the topics of Taylor series, complex numbers, matrices, vectors, and eigenvalue problems, which are all reviewed or introduced in PHSX 301. Familiarity with computers is assumed in PHSX 331, but no previous programming experience is necessary.

By reading documentation, anyone can learn how to use the black boxes that exist in most higher-level, scientific computer software for solving mathematical problems. In a physics problem, for example, one might encounter the equation $3x^2 = 4x - 5$, which is straightforward to solve analytically. It's easy enough to read a user's manual to learn how to use a computer (or calculator) to find the values of x that solve the equation. But how does the computer respond to equations, like this one, that have complex solutions? Are complex solutions meaningful to the physics problem? Are there certain combinations of constants for which the computer has difficulty solving the equation? What are they? Are there limits on the degree of the polynomial that the computer can solve? Why do some equations take more time for the computer to solve than others? Are there ways to speed that up? How many digits in the results are significant? Are there ways to gain more significant figures?

Courses that offer instruction on specific software packages are available outside of a physics curriculum. The goal of PHSX 331 is to enable the student to understand and optimally use computer software to solve several types of common physics problems, critically evaluating the "answer" to decide whether it's right and answering the kinds of questions posed above. To that end, the programming language chosen for use in PHSX 331 is `Matlab`, which has the advantages of being fairly intuitive, not so idiosyncratic that one must start from scratch when learning another language, and affordable.

The first edition of this text treated only problems in theoretical physics and comprised the first six chapters, introducing methods of solving algebraic equations, ordinary differential equations, and eigenvalue equations. This second edition includes two additional chapters of interest in experimental physics, introducing the subjects of Least-Squares Fitting and Statistical Measures of data. We hope that the tools developed here are useful to students using other programming languages and in other coursework.

List of Programs

Algorithms and Programs
Program 1: `mars_angle` 4

Algebraic Equations — Root Finding
Program 2: `bisect_eq2` 13
Program 3: `bisect` 16
Program 4: `func_eq2` 17
Program 5: `func_eq3` 18
Program 6: `newton` 20
Program 7: `deriv_eq2` 21

Root Finding in More Dimensions
Program 8: `efld` 27
Program 9: `efld_mag2` 30

Solving an Ordinary Differential Equation
Program 10: `sky_diver` 35
Program 11: `euler_1d` 37
Program 12: `rk2_1d` 42

Sets of ODEs — Multiple Dimensions
Program 13: `rk2` 49
Program 14: `tan_vec` 50
Program 15: `pendulum` 53
Program 16: `baseball` 55
Program 17: `long_ball` 56

Eigenvalue Problems
Program 18: `caseI` 65
Program 19: `BC_caseI` 65
Program 20: `caseII` 68
Program 21: `BC_caseII` 68

Least-Squares Fitting of Data
Program 22: `linfit` 77

Statistical Measures
Program 23: `getstats` 81

Chapter 1

Algorithms and Programs

The basic building block for computational physics is the **algorithm**. An algorithm is a series of simple calculations which produces a final result, *the answer*. This course will introduce algorithms for solving algebraic equations, solving differential equations and for solving eigenvalue problems. In each case the algorithm will be fairly simple — two or three mathematical operations. It is easy to do this with a pencil and paper, or on a pocket calculator. Indeed, the subject of computational physics dates back many centuries, long before the invention of electronic computers.

To solve a given physics problem, however, it is often necessary to repeat one algorithm hundreds or even millions of times. This would prove rather tedious to do by hand, and so we use a computer. In particular, we write **a program** to implement the **algorithm**.

The prospect of implementing algorithms can seem daunting to the uninitiated. Isn't programming done by computer scientists? Which computer language is best? Actually, within the framework of numerical algorithms most computer languages look pretty similar, and they are extremely simple. Algorithms in these notes are implemented in `Matlab`, however, other than a few control statements like `if` or `while` they look like a series of equations. Nor will it ever be necessary to actually "write programs" here, or probably ever in your life (unless you are so inclined.) The best way to solve a computational problem is to find a working program which performs are related task, to understand how it works, and to change it just enough so it suits your purpose[1].

1.1 An example

The problem

As a (relatively) simple example of how this will work let's consider the algorithm and the program designed to answer to following question: *Where in the sky can we find the planet Mars on a given day in the year 2000?* This might appear to be a relatively complicated calculation since it involves the motions of both Mars and the Earth about the Sun. Actually it involves combining a few simple steps. For clarity we define a two-dimensional coordinate system[2] in which an angle of 0° is defined by midnight (in Montana) Dec. 31, 1999, i.e. the beginning of the year 2000.[3] First we determine the location of the Earth $\mathbf{r}_e = (x_e, y_e)$ and the location of Mars $\mathbf{r}_m = (x_m, y_m)$ in these

[1] It has been said that "Good programmers write good code; great programmers **steal** good code.

[2] This assumes that both orbits lie in the same plane; in fact they are misaligned by less than 2°.

[3] Astronomers use a coordinate system whose zero-angle is defined by the sun-ward direction at the Spring Equinox. Spring Equinox in Y2000 occurred on March 20; i.e. day $d = 79$. This corresponds to $\theta = 78^\circ$ in our coordinates. The direction towards the sun, which is zero degrees for astronomers, is $\theta = 180^\circ + 78^\circ = 258^\circ$ in our coordinates.

coordinates. A viewer on Earth observes Mars along the vector

$$\mathbf{r}_d = \mathbf{r}_m - \mathbf{r}_e \quad ,$$

thus the apparent angle of Mars in the sky, θ, is given by the polar angle of $\mathbf{r}_d$

$$\theta \;=\; \tan^{-1}\left(\frac{y_m - y_e}{x_m - x_e}\right) \quad . \tag{1.1}$$

To actually use this information we would need to consult a star atlas and find that point along the ecliptic[4] which is θ from our reference[5] — this step will not concern us here.

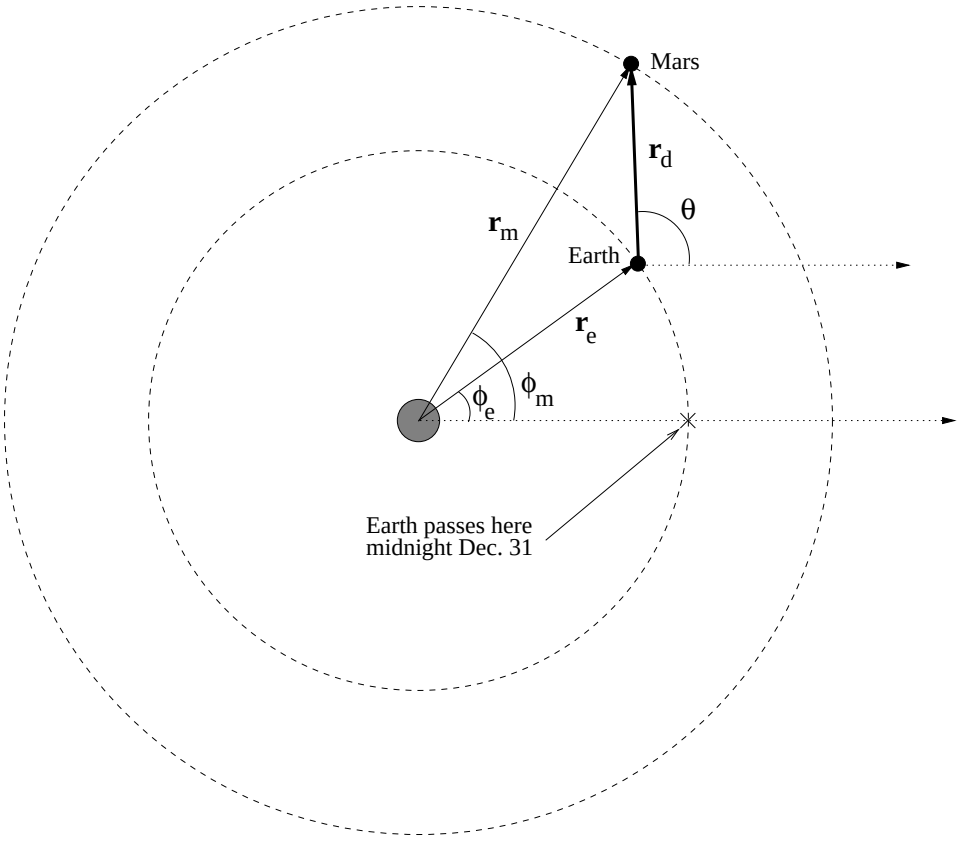

Figure 1.1: Coordinate system for calculating the apparent position of Mars.

Equation (1.1) is the answer to our question, but in order to use it we must find each of the quantities x_e, y_e, x_m and y_m from one given piece of information: the date. We will do this in two steps: first finding the angles ϕ_e and ϕ_m for the given date, and then finding x_e, y_e, x_m and y_m from these angles. To make our life easier we can plan to provide the computer with the day-of-the-year d rather than the month and day. The values of d at the beginning of each month are given in Table 1.1 so that you may figure out d for any given date.

For added simplicity we shall assume that the orbits of the Earth and of Mars are circular, with radii $R_e = 1.496 \times 10^8$ km and $R_m = 2.279 \times 10^8$ km respectively. This means the second step can

[4]The ecliptic is the path through the sky which all planets appear to follow; it crosses through each of the constellations of the zodiac (and through the constellation of Ophiucus).

[5]The point $\phi = 0$ in our coordinates falls inside the constellation Gemini.

date	d	date	d	date	d
Jan 1	0	May 1	121	Sep 1	244
Feb 1	31	Jun 1	152	Oct 1	274
Mar 1	60	Jul 1	182	Nov 1	305
Apr 1	91	Aug 1	213	Dec 1	335

Table 1.1: The day-of-the-year d for the first day of each month in 2000 (a leap year). The value of d corresponds to the beginning of the day, one second after midnight.

be solved with simple trigonometry

$$\begin{aligned} x_e &= R_e \cos(\phi_e) && (1.2)\\ y_e &= R_e \sin(\phi_e) && (1.3)\\ x_m &= R_m \cos(\phi_m) && (1.4)\\ y_m &= R_m \sin(\phi_m) && (1.5) \end{aligned}$$

Furthermore, planets in circular orbits move at constant angular velocity, $d\phi/dt = 2\pi/P$ where P is the orbital period of the planet. The orbital periods (i.e. years) for Earth and Mars are $P_e = 365.256$ days and $P_m = 686.980$ respectively, so the angles are

$$\begin{aligned} \phi_e &= 2\pi\, d/P_e \ , && (1.6)\\ \phi_m &= 2\pi\,(d - d_0)/P_m && (1.7) \end{aligned}$$

where $d_0 = 187$ (i.e. July 6, 2000) is the day on which Mars passed through the angle $\phi_m = 0$.

The algorithm

With this bit of trigonometry under our belt we are ready to define the algorithm. The "algorithm" consists of performing each of the simple calculations above, in reverse order:

$$\begin{aligned} \phi_e &= 2\pi\, d/P_e \\ \phi_m &= 2\pi\,(d - d_0)/P_m \\ x_e &= R_e \cos(\phi_e) \\ y_e &= R_e \sin(\phi_e) \\ x_m &= R_m \cos(\phi_m) \\ y_m &= R_m \sin(\phi_m) \end{aligned}$$

Here we should be alert to a subtle distinction between the equals sign in mathematics and in an algorithm. In mathematics $a = 3$ is a statement of fact and it can be read in either direction — it is also true that $3 = a$. When implementing an algorithm, however, we are describing an operation to be performed. A variable on the left of the equals sign will be set to the value of the expression on the right. So $a = 3$ means "assign the value 3 to the variable a". On the other hand, the statement $3 = a$ is meaningless, since 3 is not a variable, and its value cannot be changed.

Finally, we should note that the arctangent function in `Matlab` returns a value in the range $(-\pi/2,\ \pi/2)$ which is a single period of the function $\tan(x)$. The actual angle, θ should be found in the full range $(-\pi/2,\ 3\pi/2)$, which includes two full periods. This means that the final statement

should be replaced by the conditional one

$$\theta = \begin{cases} \tan^{-1}\left(\dfrac{y_m - y_e}{x_m - x_e}\right) & \text{if } x_m - x_e > 0 \\ \tan^{-1}\left(\dfrac{y_m - y_e}{x_m - x_e}\right) + \pi & \text{if } x_m - x_e \leq 0 \end{cases}$$

Conditional assignments like this are easy to implement in any programming language.

The program

Translating the algorithm above into a program is quite straightforward. Below is a copy[6] of a computer file named `mars_angle.m`:

..... Begin Program 1

file: `mars_angle.m`

```
function theta = mars_angle( d )
%
%   Find the angle (in degrees) at which Mars appears in the sky
%   on day d of the year 2000.

  Pe = 365.256;     %  period of Earth orbit (days)
  Pm = 686.980;     %  period of Mars orbit (days)
  d0 = 187;         %  day of 2000 on which Mars crosses angle 0
  Re = 1.496e8;     %  radius of Earth orbit (km)
  Rm = 2.279e8;     %  radius of Mars orbit (km)

  phi_e = 2*pi*d/Pe;
  phi_m = 2*pi*(d-d0)/Pm;

  xe = Re*cos( phi_e );
  ye = Re*sin( phi_e );
  xm = Rm*cos( phi_m );
  ym = Rm*sin( phi_m );

  if ( xm - xe > 0 )
    theta = atan( ( ym - ye )/( xm - xe ) );
  else
    theta = atan( ( ym - ye )/( xm - xe ) ) + pi;
  end

  theta = 180*theta/pi;   %  convert from radians to degrees
```

..... End Program 1

Anything to the right of a `%` is a comment, to be read by humans and not by the computer. The constants of the problem are first assigned, then each of the calculations is performed in order until the final result `theta` is found, and converted from radians to degrees.

To use the program we must call it from `Matlab`. `Matlab` itself is a program, which maintains a set of variables and permits you, the user, to perform mathematical operations on these. The set of commands and variables being used by the user is called *the main level* of `Matlab`. Typing the following statement at the main level

[6] The name of the file is provided in the upper right for reference; it is not actually typed into the file.

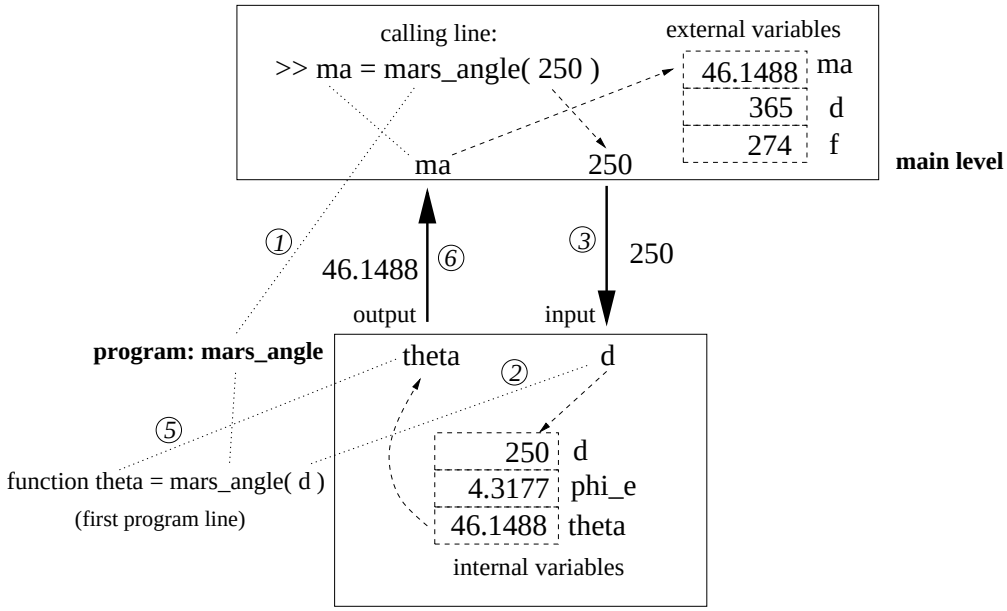

Figure 1.2: A schematic depiction of the steps performed by `Matlab` when it runs the program `mars_angle`. Step numbers are indicated by circled digits.

```
>> ma = mars_angle( 250 )
```

causes the computer (i.e. `Matlab`) to *run* our program. This means the computer will attempt to perform the following 7 simple steps, illustrated in Fig. 1.2. It all begins with the single line above, which is referred to as the *calling line*. This causes `Matlab` to do the following

1. Seek a file named `mars_angle.m` where it hopes to find the definition of this function.

2. "Read" the first line of the file to learn that the function `mars_angle()` takes a single *input argument*. Within the program this argument is called `d`.

3. From the calling line, at the "main level",

   ```
   >> ma = mars_angle( 250 )
   ```

   it learns that the input argument is to be set to 250. It assigns `d = 250` *within the program*.

4. The computer now "enters" the program. That is to say it begins to execute each of the statements in the program file in order. Upon entering the program the computer temporarily "forgets" all variables stored at the *main level*, and begins creating and using *internal* variables. When it is done with the program it will forget all of the internal variables. For example, when it gets to the line

   ```
   phi_e = 2*pi*d/Pe;
   ```

   it creates a new internal variable called `phi_e`, and assigns to it the value 4.3177 (the result of the multiplications and division). We will return shortly to this selective amnesia called *internal* and *external* variables.

5. After executing the last line of the program, the computer is now ready to return to the *main level*. It "rereads" the first line of the program and learns that the function `mars_angle` returns a single *output value* to the main level. The value to be returned is that held by the internal variable `theta`. At the instant the program finishes the internal variable `theta` contains the value 46.1488.

6. Returning to the calling line, at the main level, a new variable `ma` is created and it is set to the value *returned* by the program `mars_angle`: 46.1488.

7. Since the statement on the calling line does not end in a semi-colon, `Matlab` reports on the screen that it has just set `ma` to 46.1488.

   ```
   ma =

      46.1488
   ```

   Each line in the program ends in a semicolon to prevent `Matlab` from reporting back each calculation it performs. Should you ever find that you want to hear reports from within the program simply remove the semicolon from the line in the program file.

The sequence of events above is a *best case scenario*. At each step there are many things that can go wrong, and you will undoubtedly experience every type of failure at some point. Here are some examples of failures at a few of the more problematic steps:

- If `Matlab` fails to find a file with the name `mars_angle.m` it complains:

  ```
  ??? Undefined function or variable 'mars_angle'
  ```

  This happens more often than you'd like, and the explanation is usually quite simple:

  - You mistyped the name of the function on the calling line.
  - You forgot to save the file or you saved it with an incorrect name.
  - You saved the file in a folder where `Matlab` cannot find it. `Matlab` has a list of folders called a *PATH*, where it looks for program files. You must either save the file in a folder already in its PATH, or add your folder to its PATH. See Appendix A for more information on `Matlab` idiosyncracies.

- If the number of input arguments in the calling line doesn't agree with the number in the definition we're in trouble. Had we used two arguments in the calling line

  ```
  >> ma = mars_angle( 250, 34 )
  ```

  `Matlab` would complain with the (uncharacteristically) informative statement

  ```
  ??? Error using ==> mars_angle
  Too many input arguments.
  ```

- If one of the statements in the program doesn't make any sense you will hear about it. This is known as a *bug* in the program, and the process of *debugging* a program will become quite familiar to you. Since `Matlab` performs the steps in order it will crash at the first bug it encounters. You will fix this, save the revised program file, and call it again from the main level (this is debugging). If you are unlucky the program will find another bug at a later point, and crash again *there*. Be patient!

- If the first line names an output variable which is never set within the program there will be confusion. This is a actually an example of a bug in the program file but it produces a puzzling error message. Changing the first line of the program to

  ```
  function th = mars_angle( d )
  ```

  causes the following message at execution time:

  ```
  Warning: One or more output arguments not assigned during call to
  'mars_angle'.
  ```

  Since this is only a `Warning` it is tempting to ignore it. In fact, `Matlab` has dutifully performed all of the calculations and then thrown away the answer. It was expecting the answer to be stored in a variable called `th` at the end of the calculation, Since it found no variable called `th` it simply returned no output value at all, and the external variable `ma` was never created.

The detailed list of 7 steps is very explicit about the strict rules for passing information into and out of a program. All programming languages have similar rules for this process, in order to make it very clear what operation is being performed on which quantities. Programs are frequently compared to *boxes* or *rooms* which maintain strict secrecy by permitting information to enter and leave only through defined channels. Information is passed *into* the program using one or more *input arguments* listed within parentheses after the name of the program. Information is passed *out* of the program as one or more *returned values* or *output arguments*.[7] The calling line which invokes the program also gives the name of the external variable into which the output information will be deposited.

The very first line of any program is critical for defining which will be inputs and which will be the outputs. In our example the all important first statement is

```
function theta = mars_angle( d )
```

The first word, `function` alerts the computer to the fact that you are defining a new function. Next comes the name by which the returned value will be known **internally** — `theta`. When the program is finished the value held by `theta` will be passed back top the main level as the returned value. Since this is the only information which will survive once the external variables are forgotten it is important to get its name right.[8] Next comes the name of the function itself: `mars_angle`. This name is known at the main level, and is the way the computer will know to execute this particular program. Finally, within parentheses is the name of the single argument `d`.

When the program `mars_angle` is called, from the main level, the computer finds the correct program and then begins executing the instructions. As it does so it keeps a list of all *internal* variables. You can think of these as scratch paper which will be thrown away when the program is finished. Even the names of the argument(s) and returned value(s) will be forgotten. Notice that at the main level we assigned the returned value to a variable called `ma`. Typing the command

```
>> who
```

at the main level produces a list of all variables currently defined — at the main level. We see that the only variable defined is `ma`; no trace remains of any internal variable, even `theta` is gone. Of course the final value of `theta`, namely 46.1488, has been copied into `ma` and therefore lives on.

As one further demonstration, we can type the three statements at the main level

[7] It is also possible for the values of arguments to be changed, but this is often considered an inexcusable breach of security.

[8] Our list of potential programming problems included a case in step 5 where this part of the first line was incorrect.

```
>> d = 365
>> f = 274
>> ma2 = mars_angle( f )
```

Now we have created one variable `d` to contain the value 365, another variable `f` to contain the value 274. Then we called `mars_angle` with `f` in the argument list. This caused the value of `f`, 274, to be copied over to an **internal** variable called `d`. Note that we have tried to muddy the waters by defining an **external** variable called `d`, but the program neither knows nor cares about this. Its only concern is the variable on its argument list, which it copies onto a secret internal variable named `d`. Typing

```
>> who
```

shows that we now have 4 variables at the main level. Typing one of their names will show what value they currently contain. The variable `d` still contains 365; it was absolutely unaffected by the variable of the same name used internally by `mars_angle`.

Where we're headed

The example above illustrates the framework in which we will solve all other physics problems. First we will identify the problem to be solved, and then think of an algorithm in abstract mathematical terms. This algorithm will be a series of operations to be performed on input which ultimately return an answer as output. The input/output structure provides the algorithm with its flexibility: we can now calculate apparent positions of Mars for any number of days by changing the value of the input

```
>> ma = mars_angle( 100 )
>> ma = mars_angle( 120 )
>> ma = mars_angle( 140 )
>> ma = mars_angle( 160 )
```

In addition we will see that even very complicated physics problems can be broken down into simpler component problems, each solved by a basic algorithm. By the end of the course we will calculate the trajectory of a spacecraft traveling to Mars. The program above is one component of this larger calculation.

Chapter 2

Algebraic Equations — Root Finding

2.1 The general problem

Many physics problems may be worked until the answer takes to form of an *algebraic equation*[1] such as

$$v^2 - 3v = 7 \ , \tag{2.1}$$

$$e^x = 3x^2 \ , \tag{2.2}$$

$$250\cos\theta\,[\sin\theta + \sqrt{\sin^2\theta + 0.08}\,] = 200 \ . \tag{2.3}$$

Each of the cases above represents a single equation for a single unknown, v, x and θ respectively. As you may know already, it is of utmost importance that there be the same number of unknowns as there are equations. If equations outnumber unknowns there is probably no solution, and if unknowns outnumber equations there will be embarrassingly many solutions. Before proceeding we must assure an even match. (The three cases above pass this test.) The equation may be solved by a simple series of steps. The basic steps are always the same

1. Zero the right hand side
2. Graph the function
3. Guess at an answer
4. Improve the answer

Below we discuss each step in detail.

Step 1 — Zero the right It is conventional to manipulate the equation one final time to yield an equation of the form

$$f(x) = 0 \ . \tag{2.4}$$

where the function $f(x)$ contains all of the algebra. Simply subtract the expression on the right of the equals sign from both sides.

[1] As distinguished from a *differential equation* which would not be a solution at all, but rather a new problem. Computational physicists are known to use the modifier *just* as in "It's *just* an algebraic equation.". As we will see any algebraic problem can be solved numerically.

For the equations above we find three different functions

$$f(v) = v^2 - 3v - 7 \quad , \tag{2.1$'$}$$

$$f(x) = e^x - 3x^2 \quad , \tag{2.2$'$}$$

$$f(\theta) = 250\cos\theta\,[\sin\theta + \sqrt{\sin^2\theta + 0.08}\,] - 200 \quad . \tag{2.3$'$}$$

We will assume here that the unknowns in each equation is purely *real*, with no imaginary part.[2]

You may recognize Eq. (2.1′) as a *quadratic* equation whose solutions (there happen to be two) are given by the quadratic formula:

$$v = \overbrace{\frac{3+\sqrt{37}}{2}}^{\text{exact}} = 4.54 \quad ,$$

$$v = \frac{3-\sqrt{37}}{2} = -1.54 \quad .$$

This is an example of an *analytic* solution to an algebraic equation, in which the answer may be expressed exactly.

Try as you might it is not possible to cast either Eq. (2.2′) or (2.3′) in forms where the exact solution can be found . . . there is no counterpart to the quadratic formula for these equations. In such cases (which are *most* cases in the real world) we must settle for *approximate* numerical answers. There are a number of methods which provide approximate answers to any equations of the form $f(x) = 0$.

Step 2 — graph it The single most enlightening approach to any algebraic equation is to graph the function $f(x)$ and look for places it crosses $f = 0$. Computers and graphing calculators make this task almost trivial, but even without these aids it is always enlightening to simply plug in a few values and see what happens. Consider a few values of the function in Eq. (2.2′).

x	$f(x) = e^x - 3x^2$
-1	-2.63
0	1
0.5	0.90
1	-0.28
4	6.60

From this handful of numbers we can see the general shape of the unknown function: as x increases $f(x)$ goes up becoming positive, then down (becoming negative), and then up once more. A computer generated plot, shown in Fig. 2.1, confirms this behavior. In particular two places where $f(x) = 0$ can be estimated by eye. Crossing A is at $x \simeq -0.45$ while crossing B is at $x \simeq 0.9$.

[2] As a physicist you will usually know whether the number you seek should be real, such as a speed, or may be complex, such as a frequency. If you are seeking a complex number you really have *two* algebraic equations for *two* real unknowns: the real and imaginary parts of the equation and the answer, respectively.

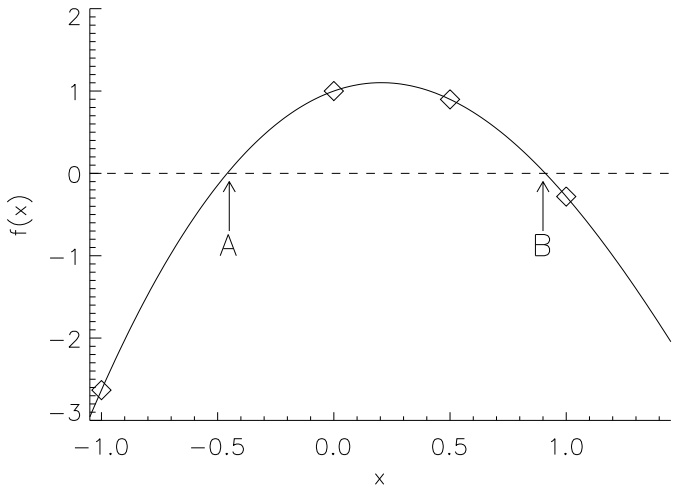

Figure 2.1: Part of the curve $f(x) = e^x - 3x^2$ (solid). It crosses the dashed $f = 0$ line at two points, labeled A and B. The entries for the table are shown as $\diamond$s.

The points A and B represent two solutions to Eq. (2.2). Are these the only solutions? To answer this question try adding a few more rows to the table of function values, or continue the graph further to the right and left. The key point here is that **there is no general rule about how many times a function $f(x)$ can cross zero**. Contrast this to the quadratic formula which always gives two solutions, although it is possible that both are complex. A polynomial of order n,

$$f(x) \;=\; a_n\,x^n \;+\; a_{n-1}\,x^{n-1} \;+\; \cdots \;+\; a_1\,x \;+\; a_0 \quad , \tag{2.5}$$

can have as many as n different real roots. A function like e^x is actually a polynomial of order $n = \infty$:

$$e^x \;=\; 1 + x + \tfrac{1}{2}x^2 + \cdots + \tfrac{1}{n!}\,x^2 + \cdots \tag{2.6}$$

so we can expect as many as ∞ different solutions, or as few as zero! Every problem will be different, which is why graphing is so important.

Step 3 — Guess at the answer There could be any number of solutions to an algebraic equation $f(x) = 0$. All methods of solving such equations require a *guess* at the answer to get started. If there is more than one solution, such as points A and B in Eq. (2.2′), then the solution found will depend on the guess.

Step 4 — Improve the guess iteratively There are many algorithms for iteratively improving a guess until it represents a reasonable approximation to a solution $f(x_0) = 0$. Below we describe two such algorithms, called *Bisection* and *Newton's* method.

2.2 Method 1: Bisection

We found the solution $x \simeq 0.9$ to Eq. (2.2) by "eyeballing" the graph of $f(x)$. This approximate solution is only as accurate as our eyes are good. One natural method of improving the answer is to "zoom" in around the crossing point. This is the basis of the method known as *bisection*.

The initial interval We begin by noting that two value from the table define an interval [0.5, 1.0] which *must* contain the exact solution x_0 where $f(x_0) = 0$. On the left $f(0.5) = +0.90$ is positive and on the right $f(1.0) = -0.28$ is negative. Somewhere between $x = 0.5$ and $x = 1.0$ the function must cross $f = 0$ as it goes from being positive to being negative.

Dividing the interval The middle of the interval is $x = 0.75$. Evaluating the function at this point gives $f(0.75) = 0.43$. This is positive, while $f(1.0) = -0.28$ is (still) negative. Thus we have determined that the crossing must lie within the smaller interval [0.75, 1.0]

Repeating We now have a smaller interval which we can divide in half once more at $x = 0.875$ to find $f(0.875) = 0.10$, leaving the interval $[0.875, 1.0]$. The midpoint of this interval is $x = 0.9375$ where $f(0.9375) = -0.083$ making the interval [0.875, 0.9375]. This series of steps is illustrated in Figure 2.2.

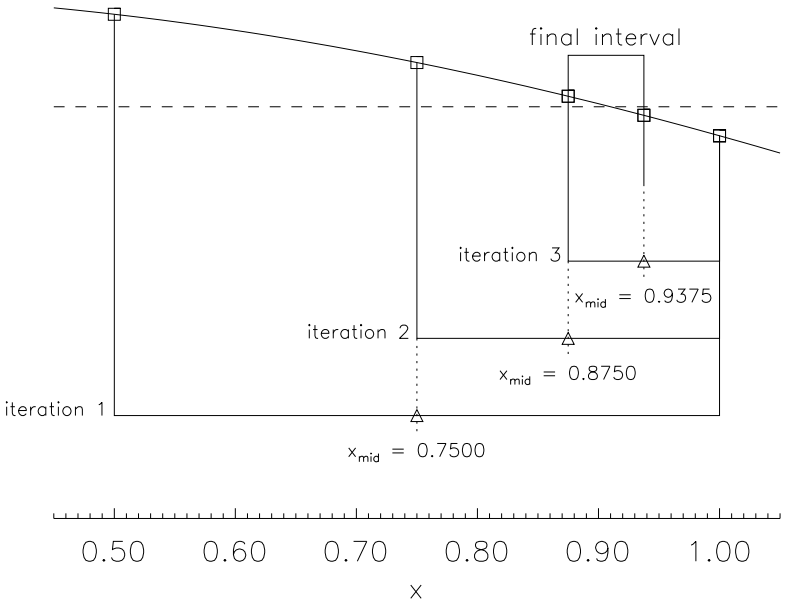

Figure 2.2: The first three iterations in a solution to Eq. (2.2). Each iteration consists of dividing an interval in half, evaluating the function at the midpoint and then defining a new interval containing the root $f(x_0) = 0$.

Settling for an answer We can continue dividing intervals in half and then half again hoping eventually to find x_0 such that $f(x_0) = 0$. Alas, this will probably never happen: in general it is not possible to find an exact answer to an algebraic equation. Instead we must settle for an approximate solution

$$x_* \simeq x_0 \quad .$$

Suppose we take $x_* = 0.90625$, the midpoint of our final interval [0.875, 0.9375]. We know that the true solution x_0 cannot be farther from x_* than half that final interval: $\frac{1}{2}(0.9375 - 0.875) = 0.03125$. This is the largest possible error in our approximate solution.

Perhaps an error of 0.03 is not good enough; for instance if we would like to know x_0 to two significant digits. In other words we would like $|x_0 - x_*| < 0.005$. To accomplish this the final

interval should be 0.01. How many iterations must we perform to achieve this accuracy?

The first interval [0.5, 1.0] is of size $\frac{1}{2} = 2^{-1}$, the next, [0.75, 1.0] is of size $\frac{1}{4} = 2^{-2}$ and so on. The seventh interval (after 6 iterations) is therefore $2^{-7} = \frac{1}{128} < 0.01$, and its midpoint can be found (see Figure 2.3) to be

$$x_* \;=\; 0.9102 \quad ,$$

as shown in the figure.

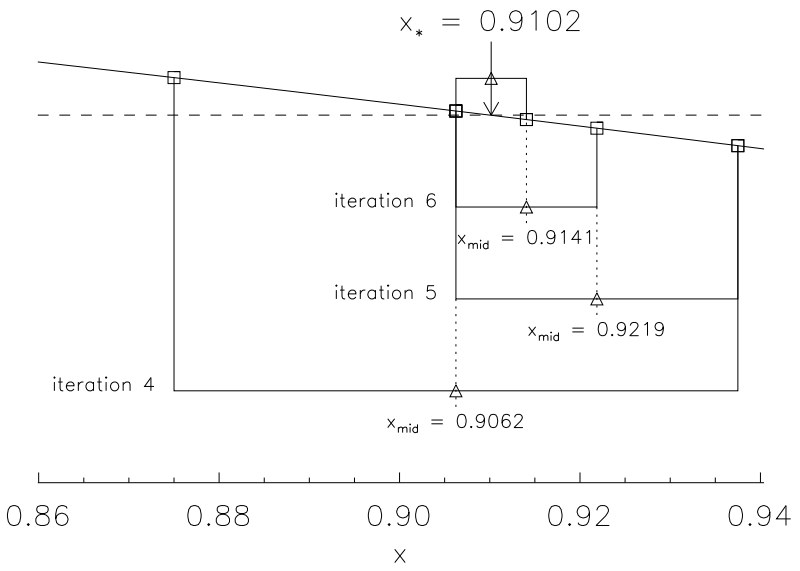

Figure 2.3: A continuation of the bisection from figure 2.2.

The bisection method is an *iterative* procedure. Each iteration (repetition) makes the approximate solution a little more accurate. How many iterations are required to find x_0 to 4 decimal places i.e. $|x_0 - x_*| < 0.5 \times 10^{-4}$?.

Programming the algorithm

While the basic procedure above is quite simple it can be a bit tedious to evaluate the function $f(x)$ over and over again. This is a perfect job for a computer.[3] Below is an example of a program, written in `Matlab`, which will perform the bisection on the function $f(x) = e^x - 3x^2$.

..... Begin Program 2

file: `bisect_eq2.m`

```
function x_star = bisect_eq2( x_lft, x_rgt )
%
%   Find a solution to the equation f(x) = exp( x ) - 3 x^2 = 0
```

[3]Before *electronic* computers, assistants were hired to perform such tedious calculations by hand or with mechanical adding machines. These people were called "computers".

```
%   using the method of bisection.  Begin with the interval
%    [ x_lft, x_rgt ] and return the solution accurate to 0.5e-5

  f_lft = exp( x_lft ) - 3*x_lft^2;   %   f(x_lft)
  f_rgt = exp( x_rgt ) - 3*x_rgt^2;   %   f(x_rgt)

  while ( x_rgt - x_lft ) > 1.0e-5
                                      %      LOOP ---+
    x_mp = 0.5*( x_lft + x_rgt );  % the midpoint     |
    f_mp = exp( x_mp ) - 3*x_mp^2; %  f(x_mp)         |
                                    %                 |
                    %  replace that end with         |
                    %  the same sign as f(x_mp)       |
    if f_mp*f_lft > 0.0             %                 |
              % f(x_mp) is same sign as f(x_lft)...   |
      x_lft = x_mp;  %   replace x_lft                |
      f_lft = f_mp;  %   and f(x_lft)                 |
    else                            %                 |
              % f(x_mp) is same sign as f(x_rgt)...   |
      x_rgt = x_mp;  %  replace x_rgt                 |
      f_rgt = f_mp;  %  and f(x_rgt)                  |
   end        % end of if - else statement            |
                                    %         ----------+
  end   %

  x_star = 0.5*( x_lft + x_rgt );   %     the value to return
```

..... End Program 2

For simple programs such as this, the `Matlab` language[4] does not look all that different from other programming languages such as FORTRAN, C or Pascal. Note that everything to the right of a `%` symbol is ignored by the computer; these are *comments* for human eyes only. A few remarks concerning program 2, `bisect_eq2`:

- The program for the function `bisect_eq2`, should be stored on the computer in a file named `bisect_eq2.m`. When `Matlab` sees a function it does not recognize it searches all files ending in `.m` for one with the same name as the unfamiliar function. If it does not find such a file, it will complain. The list of places it searches is called the *path*. Sometimes it will fail to find the file because its current path does not include the directory (folder) containing the file.

- The first line of program 2

  ```
  function x_star = bisect_eq2( x_lft, x_rgt )
  ```

  provides the name of the function, `bisect_eq2`; shows that the function will take two *input arguments*, `x_lft` and `x_rgt`; and will return a single output value, `x_star`. It can be run from `Matlab` by typing

  ```
  >> xb = bisect_eq2( 0, 1 )
  ```

  Typing this command sets the input arguments `x_lft = 0` and `x_rgt = 1` and then begins executing each of the statements in program 2. The final answer is called `x_star` within the function, however, when execution of `bisect_eq2` is complete this number is assigned to the variable `xb`. Typing `>>who` shows all of the variables currently stored by `Matlab`. You will see

[4] To assure total confusion the language used by the program `Matlab` is also called `Matlab`.

that none of the variables `x_lft`, `x_rgt`, `f_lft`, etc. exist. These names are "internal" to the function `bisect_eq2`; they all cease to exist the instant the program is finished executing. To simply view the result without storing it, you may type

```
>> bisect_eq2( 0, 1 )
```

- The part to be repeated appears inside a `while` statement (marked with the little box and the word `LOOP`). The statement `x_rgt - x_lft > 1.0e-5` is tested. If it is true then the entire sequence of commands will be executed, in order once. The statement is then tested again, if it is still true then the sequence is executed again. This continues until it is no longer true that `x_rgt - x_lft > 1.0e-5` when tested. Put another way the statements are repeated until the interval size is $\leq 10^{-5}$.

  It is a good thing to convince yourself that a condition will become false *eventually*: computers are frustratingly gullible and patient and will continue to execute commands forever if you give it a silly command like `while 2.0 > 1.0` which will always be true.[5]

- A sort of trick has been used to decide which side of the interval the midpoint should replace. The correct choice will depend on the signs of $f(x)$ at the ends of the interval *and* on the sign of f at the midpoint. The rule is "replace that end at which $f(x)$ has the same sign as `f(x_mp)`". To implement this rule we note that if $f(x_{\rm lft})$ and $f(x_{\rm mp})$ have the same sign, either both positive or both negative, then $f(x_{\rm lft}) \times f(x_{\rm mp}) > 0$. If this is the case then the course of action is: replace `x_lft` with `x_mp` and replace `f_lft` with `f_mp`. If this is not the case we have assumed that $f(x_{\rm lft})$ and $f(x_{\rm mp})$ are of opposite signs, and that $f(x_{\rm rgt})$ and $f(x_{\rm mp})$ must have the same sign. We have totally neglected the possibility that one of them is exactly zero (see below).

An improved program

It is now worth pointing out some problems with the simple program `bisect_eq2`.

1. This program finds roots of the function $f(x) = e^x - 3x^2$ only. To find roots of a different function a whole new program must be written. The alternative is a more general program which is demonstrated below.

2. The accuracy of the answer is fixed to be 10^{-5}. This is not such a bad thing with $f(x)$ from Eq. (2.2′), but consider a different function whose values are all very small. Perhaps x represents a time in seconds, and one solution is $x_0 \sim 4.23 \times 10^{-9}$, i.e. several nanoseconds. Using the fixed accuracy of 10^{-5} would make the answer meaningless. Optimally, we would like to specify a relative accuracy:[6] refine the interval until $|x_{\rm rgt} - x_{\rm lft}|$ is smaller than some fraction of *either* $|x_{\rm rgt}|$ or $|x_{\rm lft}|$ (whichever is larger). The mathematical formulation of such a termination conditions is

$$|x_{\rm rgt} - x_{\rm lft}| \;\leq\; 10^{-5} \times \max(\,|x_{\rm lft}|,\, |x_{\rm rgt}|\,) \quad . \tag{2.7}$$

3. What would happen is one were to accidentally enter the interval in reverse order

[5] If you feel like testing the claim, be sure you know how to *abort* `Matlab`. The computer will fall into a stupor, often called "being hung".

[6] Relative accuracy makes the most intuitive sense, i.e. how many significant digits, but it is also the correct choice for computational reasons (see Appendix B on precision).

```
>> bisect_eq2( 1, 0 )
```

This would result in the assignments `x_lft = 1` and `x_rgt = 0`. When the interval size is tested by the command `x_rgt - x_lft > 1.0e-5` it will be false immediately: $-1 < 10^{-5}$, and the program will terminate with the result 0.5. This is one instance where we have assumed the user will make no mistakes; a very dubious assumption. It pays to anticipate user errors.

4. What would happen if one were to enter `x = bisect_eq2( -1, 1 )`? Who knows. This is a more forgivable user error. We have assumed the initial interval will contain one and only one root. If, on the other hand, $f(x_{\text{lft}}) \times f(x_{\text{lft}}) > 0$, then the user has goofed, and the interval contains zero, two or perhaps four roots, rather than one! It is best to check for this kind of goof before proceeding.

5. What if, at some point, the exact answer x_0 is found? At present that possibility is not considered. If such a stroke of good fortune occurs it would be best to simply set x_* to the exact answer and exit.

A revised version of the program is given below as program 3. This is a more **general** program than `bisect_eq2` — it can solve *any* algebraic problem $f(x) = 0$. The user must, however, tell the program which specific problem it is to solve; i.e. he must specify the function $f(x)$. This is accomplished by putting the "name" of the function $f(x)$ in the argument list, where it is stored in an internal variable called `func_name`.[7] We discuss this more below.

..... Begin Program 3

file: `bisect.m`

```
function x_star = bisect( func_name, x_lft, x_rgt )
%
%   Find a solution to the equation f(x) = 0 for the function
%   named func_name.m using the method of bisection.
%   Begin with the interval [ x_lft, x_rgt ] and return the solution
%   with relative error 0.5e-5.

f_lft = feval( func_name, x_lft );    % f_lft = f(x_lft)
f_rgt = feval( func_name, x_rgt );    % f_rgt = f(x_rgt)

if f_lft*f_rgt > 0          % ooops.  see problem #4.
   return                   % bail.
end

while abs( x_rgt - x_lft ) > 1.0e-5*max( abs( [ x_lft x_rgt ] ) )
   x_mp = 0.5*( x_lft + x_rgt );
   f_mp = feval( func_name, x_mp );

   if f_mp == 0           % got lucky!  see problem #5.
      x_star = x_mp;
      return
   end

   if f_mp*f_rgt > 0
      x_rgt = x_mp;
      f_rgt = f_mp;
```

[7] Words are stored in a special type of variable called a *string*. When setting a string variable the word is enclosed in single quotes.

```
   else
      x_lft= x_mp;
      f_lft= f_mp;
   end
end

x_star = 0.5*( x_lft + x_rgt );
```

..... End Program 3

The program `bisect` will find the root of any function which `Matlab` knows by name. The name is put in the argument list, and `Matlab` then searches its path to find the corresponding program file. For instance the command

```
>> my_pi = bisect( 'sin', 3, 4 )
```

will find the root of $\sin(x)$ inside the range $x \in [3, 4]$. Since all computers evaluate trigonometric functions in radians rather than degrees the root found will be $x_0 = \pi$. Thus the command above is one way to calculate π to 4 significant digits. The name of the function is in quotes so it will be passed to `bisect` as a string (of characters). Without the quotes `Matlab` would assume there must be some variable named `sin` whose value should be passed. The string `'sin'` is recorded in the internal variable `func_name`.

The `Matlab` program `feval` will evaluate the function whose name is given by its first argument. To see this type

```
>> feval( 'sin', 1.5 )
```

This statement is exactly the same as the more logical one

```
>> sin( 1.5 )
```

However, if the name of the function is given as a character string, such as `func_name`, the only way to perform the evaluation is using `feval`. This weird trick is essential when writing general implementations of an algorithm.

To find the roots of functions in Eqs. (2.2′) and (2.3′) we must create programs which evaluate these functions the same way the program `sin.m` evaluates the trigonometric function $\sin(x)$.[8] In other words we must write a program which takes a single input argument x, and returns the value $f(x)$. Below are `Matlab` programs, called `func_eq2` (program 4) and `func_eq3` (program 5), which evaluate the functions from Eq. (2.2′) and Eq. (2.3′) respectively.

..... Begin Program 4

file: `func_eq2.m`

```
function f = func_eq2( x )
%
%   Evaluate the function from equation (2.2')

f = exp( x ) - 3*x^2;    % set f to return function value
```

..... End Program 4

[8] Where else do you think values of $\sin(x)$ come from? Some poor programmer has to roll up his sleeves and implement an algorithm for it.

BEGIN PROGRAM 5

file: `func_eq3.m`

```
function f = func_eq3( theta )
%
%   Evaluate the function from equation (2.3') for theta in degrees

th_rad = pi*theta/180.0;  %  convert angle to radians

f = 250*cos(th_rad)*( sin(th_rad) + sqrt( sin(th_rad)^2 + 0.08 ) ) - 200;
```

END PROGRAM 5

It is important to step back and see that we are tackling our problem in a two-stage attack, writing **two** programs

1. A **general** program like `bisect` to solve the general class of problem — e.g. all algebraic problems.
2. A small **specific** program which contains the details of the problem at hand.

Even though the general/specific breakdown entails writing more programs it has many advantages. If we ever encounter a different problem from the same general class we need only write a new version of the specific program — a very simple task. In fact, there are generous people out there who may have already written a version of the general program for you[9]. If you're lucky enough to skip part 1, you can see from Program 4 and Program 5 that the second part might be completely trivial. It is still very important.

A second advantage of the general/specific division is that it allows us to use `Matlab` to graph the functions $f(x)$. Typing

```
>> fplot( 'func_eq2', [ -1, 1 ] )
```

produces a plot similar to Fig. 2.1. Furthermore, it permits us to perform the most important task in computational physics: **Testing the code**. In this case we want to convince ourselves we have not made any mistakes typing in the single line in `func_eq2` (graphing is a good test as well). Typing the commands

```
>> func_eq2( 0 )
>> func_eq2( 1 )
>> func_eq2( -1 )
```

should produce results identical to those in the table. If `func_eq2` passes the test above, we are ready to find its roots.[10]

Finally, we may now use our general purpose program `bisect` to solve Eq. (2.2), by typing the command

```
>> xb = bisect( 'func_eq2', 0, 1 )
```

this will find root B.

[9] In case you missed the point, Program 3 is exactly that.

[10] Finding π as the root of $\sin(x)$ was a good test of the code `bisect`. Always test all your codes.

2.3 Method 2: Newton's method

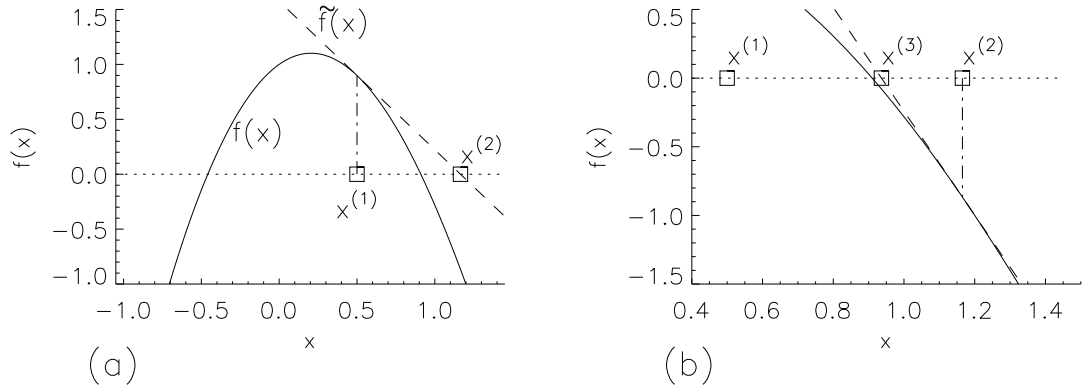

Figure 2.4: Application of Newton's method to solve Eq. (2.2). The initial guess is $x^{(1)} = 0.5$. (a) Both $f(x)$ and $f'(x)$ are evaluated there, and the approximation $\tilde{f}(x)$, dashed line, is found. The intercept of this line, where $\tilde{f}(x) = 0$, is the second approximation $x^{(2)}$. (b) The procedure is repeated beginning with $x^{(2)}$ to give the next approximation $x^{(3)}$. Note that each approximation is successively closer to the actual root.

A second technique for iteratively solving the equation $f(x) = 0$ involves approximating the function as a line $y = mx + b$, and finding its intercept. We will denote the initial guess $x^{(1)}$, the second approximation as $x^{(2)}$ and so on. We begin by approximating the function using only the first two terms in its Taylor series about $x^{(1)}$

$$\tilde{f}(x) \;=\; f(x^{(1)}) \;+\; (x - x^{(1)})\, f'(x^{(1)}) \quad , \tag{2.8}$$

where $f'(x)$ is the derivative of the function $f(x)$. Because $\tilde{f}(x)$ is a linear function of the unknown x it is easy to solve $\tilde{f}(x) = 0$

$$\tilde{f}(x) \;=\; 0 \;=\; f(x^{(1)}) \;+\; (x - x^{(1)})\, f'(x^{(1)}) \quad ,$$

$$(x - x^{(1)})\, f'(x^{(1)}) \;=\; -f(x^{(1)}) \quad ,$$

$$\Longrightarrow \quad x \;=\; x^{(1)} \;-\; \frac{f(x^{(1)})}{f'(x^{(1)})} \quad . \tag{2.9}$$

This process is shown in Fig. 2.4. Since $\tilde{f}(x)$ is only an approximation to the real function, however, this solution is only approximate. It will serve as our second approximation

$$x^{(2)} \;=\; x^{(1)} \;-\; \frac{f(x^{(1)})}{f'(x^{(1)})} \quad . \tag{2.10}$$

To improve upon the second approximation we simply repeat the same step to yield $x^{(3)}$, the third approximation. To get approximation $x^{(n+1)}$ we use the general formula

$$\boxed{x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})}} \quad , \tag{2.11}$$

n	$x^{(n)}$	$f(x^{(n)})$	$f'(x^{(n)})$	$\Delta x^{(n)}$
1	0.5000	0.8987	-1.3513	0.6651
2	1.1651	-0.8661	-3.7843	-0.2289
3	0.9362	-0.0792	-3.0670	-0.0258
4	0.9104	-0.0011	-0.2977	-0.0004
5	0.9100	$< 10^{-6}$	-2.9757	$< 10^{-6}$

Table 2.1: Application of Newton's method to solve Eq. (2.2).

where all the terms on the right involve the previous approximation $x^{(n)}$.

Let us illustrate this by solving Eq. (2.2) using Newton's method. In addition to the function $f(x) = e^x - 3x^2$, we must know its first derivative

$$f'(x) = e^x - 6x \quad . \tag{2.12}$$

Beginning with $x^{(1)} = 0.5$ the next four approximations are given in Table 2.1. The quantity $\Delta x^{(n)} = x^{(n+1)} - x^{(n)}$ is the change from approximation n to approximation $n+1$. The approximations are said to *converge* to the correct solution: they get closer and closer. The primary evidence for this is that the corrections $\Delta x^{(n)}$ get smaller and smaller. Notice that the fifth approximation (after four iterations) is $x^{(5)} = 0.9100$ which is correct at least to the fourth decimal place. The next correction, $\Delta x^{(5)} = -1.6 \times 10^{-7}$ would change only the seventh decimal place! Correction for $n > 5$ will be even smaller, so approximation $x^{(5)}$ is actually correct to six decimal places (although only four are given in the table).

A `Matlab` implementation of Newton's method, designed to give answers correct to 5 significant digits, is provided below.

..... Begin Program 6

file: `newton.m`

```
function x_star = newton( func_name, deriv_name, x_guess )
%
%   Find a solution to the equation f(x) = 0 for the function
%   named func_name using Newton's method.  The derivative f'(x)
%   is provided by the function deriv_name and the initial guess
%   is x_guess.

x_star = x_guess;   % first approximation to answer

for i = 1:20        % repeat procedure no more than 20 times

   f = feval( func_name, x_star );        %  f(x)
   fp = feval( deriv_name, x_star );      %  f'(x)
   delta_x = -f/fp;                       %  the change: -f(x)/f'(x)
   x_star = x_star + delta_x;             %  find new approximation

   if abs( delta_x ) < 0.5e-5*abs( x_star )     % have converged ...
     return                                     % ... exit early
   end

end
```

..... End Program 6

A few remarks concerning this very short program:

- It is necessary to evaluate both the function $f(x)$ and its derivative $f'(x)$. Here we have chosen to use two different programs to do that: `func_name` and `deriv_name` respectively. Before we can use `newton` to solve Eq. (2.2), for example, I must also write a program `deriv_eq2` similar to `func_eq2` (program 4) which evaluates $f'(x)$ (this is done below in program 7). The calling sequence is then

```
>> xb = newton( 'func_eq2', 'deriv_eq2', 1 )
```

..... Begin Program 7

file: `deriv_eq2.m`

```
function f_prime = deriv_eq2( x )
%
%   Evaluate the derivative f'(x) of the function from equation (2')

f_prime = exp( x ) - 6*x;    %    f'(x) = exp(x) - 6 x
```

..... End Program 7

- The main loop is performed by a `for` statement instead of the `while` statement used in the Bisection program. (The specific line is `for i = 1:20` which uses the variable `i` to count from 1 to 20. This variable is not used for anything else.) This means that the procedure will be repeated 20 times at most.[11] The `if` statement permits an early exit if x_* has converged to 5 significant digits, i.e. $|\Delta x^{(n)}| < -0.5 \times 10^{-5} |x^{(n+1)}|$. The 20 iteration limit is necessary because Newton's method is not guaranteed to converge to an answer. If we waited for convergence we could possibly wait forever. This is a significant drawback of Newton's method, discussed further below.

- The program above is prone to another kind of failure. If it ever happens that $f'(x^{(n)}) = 0$ then using Newton's method will be difficult since $f/f' = \infty$. Why does Newton's method have this flaw? If the computer is asked to divide by zero it will complain bitterly. On the other hand, if asked to divide by a number which is small, but not quite zero, the computer might not complain, but the user certainly may. Try the commands

```
>> x1 = newton( 'func_eq2', 'deriv_eq2', 0.205 )
>> x2 = newton( 'func_eq2', 'deriv_eq2', 0.206 )
```

Exercise 2.1 Newton's method is one of the most effective way to calculate square roots. It is what your calculator probably does when you press the $\sqrt{x}$ button. In this exercise you will use the method to find a decimal expression for $\sqrt{2}$ *without* ever touching the $\sqrt{x}$ button. $\sqrt{2}$ is one solution to the equation

$$f(x) = x^2 \; - \; 2 \; = \; 0 \quad .$$

Beginning with $x^{(1)} = 1$ use a calculator to write down the next three approximations, $x^{(2)}$, $x^{(3)}$ and $x^{(4)}$ from Newton's method. To how many digits does $x^{(3)}$ approximate $\sqrt{2}$? What is $\Delta x^{(3)}$?

[11] The choice of 20 as the limit is based entirely on experience. It is sometimes helpful to experiment with such choices.

2.4 Comparison — which one should I use?

> *A man with one watch always knows the time; a man with two is never sure.*
>
> *- Proverb*

We set out to answer a very simple question: for which value of x does $f(x) = 0$? Why are there two ways to answer this question? In fact there many more than just two ways. Most texts on the subject provide several additional methods, which are basically variations of the techniques above. The effect of so many options is often to leave the prospective user dazed. Practitioners of the art will wax eloquent about the various advantages and disadvantages of each of possible method. Consider using the expedient principle that *a method which works, which I understand, and which can be coded quickly is The Best Method.*

The comparison between Bisection and Newton's method is a very typical case. On the one hand, Newton's method converges to a solution *extremely* quickly. Many practitioners find speed to have overpowering allure. They spend years developing variants which converge to a solution in ever fewer steps. Of course, there are also race car mechanics who spend lifetimes making cars go a tiny bit faster. In the end, however, we will not be driving a race car on North 19th street.

In numerical analysis, as with automobiles, there is a trade-off between speed and practical usefulness. The Bisection method has one major advantage, which can often justify its slowness: *it never fails to converge*. Newton's method, on the other hand, works if the initial guess is sufficiently close to the answer. Otherwise it may wander aimlessly through space without coming to rest.

Two other elements may be relevant to a choice of method. Bisection requires the solution to be bracketed before is can be invoked. This requires more advance-work than simply guessing at an answer. Newton's method requires a single guess to get it started.

The flip side is that Newton's method requires both $f(x)$ and $f'(x)$ to be supplied by the user. This means twice as much work for the user, and in cases we will encounter later taking the derivative of the function $f(x)$ may be extremely difficult.

Bisection	Newton's method
+ Never fails	− No guarantee of convergence
+ Requires only $f(x)$	− Requires $f(x)$ and $f'(x)$
− Solution must be bracketed	+ Starts with a single guess
− Converges relatively slowly	+ Converges very quickly

Common properties of root finders

Newton's method and Bisection have certain things in common, in spite of their differences. These traits are common to most root finders and are summarized below:

- User must supply the specific program to evaluate function $f(x)$.
- User must provide an initial guess at solution. If multiple solutions exist, this guess will determine which solution is found.
- Algorithm improves solution iteratively. More iterations give more accurate solution.
- A *convergence criterion* is used to decide when the approximate solution is "good enough", and iteration may cease.

Using black boxes

In this section we have written two general programs to solve algebraic equations. This has given some insight into how algebraic equations are solved numerically. Instead of writing your own program, it is sometimes possible to use one which is already written. `Matlab` has a program `fzero`[12] already in its library which will perform the same task as `bisect` or `newton`. Calling this program

```
>> xb = fzero( 'func_eq2', 1.0 )
```

gives the solution `xb = 0.9100` we now know to be correct.

Because we do not know exactly how `fzero` works, it is falls under the term *black box*. Using black boxes (otherwise known as *canned routines*) is a matter of great ethical debate among computational physicists. The topic excites the kind of heated debates otherwise associated with arguments over the designated hitter in baseball, or plastic Christmas trees. The principle argument in favor of using a black box is that it saves time. And this is an advantage never to be dismissed lightly. The argument against their use is that one is never sure how it works, and is therefore not sure how much trust to place in the result. How accurate will the solution be in general? Under what conditions might it fail to converge?

The main goal in this course is to give you enough experience that you can become informed users of black boxes. Based on our experience with Bisection and Newton's method we can make an educated guess at how `fzero` works.

1. It does not require a program to evaluate $f'(x)$, so it cannot be using Newton's method.
2. On the other hand, `fzero` cannot be using Bisection, since it needs only a single initial guess, rather than a pair of guesses bracketing the solution.[13]

We might guess that it uses a variant on Newton's method where $f'(x)$ is approximated by evaluating $f(x)$ twice.[14] This allows it to combine some of the advantages of Newton's method with those of Bisection. Without bracketing solution, however, it is unlikely that `fzero` is "unbreakable". This is to say that, like Newton's method, there may be circumstances under which `fzero` fails to find a solution.

[12] In older versions of `Matlab`, this program was called `fsolve`.

[13] In fact `fzero` is so flexible it can also accept an interval as its second argument, in the form of a two-element vector.

[14] This method of approximating derivatives will be covered in Chapter 4.

Chapter 3

Root Finding in More Dimensions

So far we have used only *scalar* variables — variables like x which contain a single value. On the computer the variable `x` refers to a location in memory containing a floating-point number. It is also possible to use vector variables which represent an ordered list of values. The mathematical convention is to represent the entire vector with bold-face type $\mathbf{a}$, and each of its components with a subscript

$$\mathbf{a} = (a_1, a_2, a_3, a_4) \quad .$$

In this case $\mathbf{a}$ is a four-dimensional vector, and a_3 is its third component.

Computer languages, such as `Matlab`, are able to manipulate vector variables[1] but they type their names in the same font as scalars — plain old computer font. `Matlab` learns that it must store a vector when the variable is first assigned. For instance, typing

```
>> a = [ 2.4, 3, -6.9, 1145 ]
```

at `Matlab`'s main level will result in the allocation of 4 floating-point words for the variable `a`.[2] A particular component can be referenced using parentheses. Typing

```
>> a(3) = -8.12
```

will cause `Matlab` to change the value of a_3 to -8.12 from its previous value of -6.9; no other components of $\mathbf{a}$ will be affected.

When the variable is first assigned `Matlab` determines whether the variable is a scalar or a vector. Once it has decided it will not change it's mind. The assignment

```
>> x = 6.4
```

makes `x` a scalar variable, so a subsequent reference to a component, such as `x(3)` will be an error. Be sure you are clear in you mind about which variables are scalars and which are vectors.

3.1 Vector-valued functions

In Chapter 2 we solved a single equation for a single unknown. This problem was summarized by the function $f(x)$, and its various computer implementations such as `func_eq2` and `func_eq3`.

[1] Vectors are part of a general category of variable known as *arrays*. For further discussion of arrays, and `Matlab` array operations see Appendix C.

[2] Vectors in `Matlab` are defined using square brackets since parentheses are already used for too many other things. This kind of syntax changes from language to language.

Each of these functions takes a single scalar input argument x and returns a scalar value f. It is also possible, both mathematically and computationally, to define a function which accepts a vector input argument $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ and returns a vector output value $\mathbf{f} = (f_1, f_2, \ldots, f_n)$. This is called a *vector-valued function of a vector argument.*

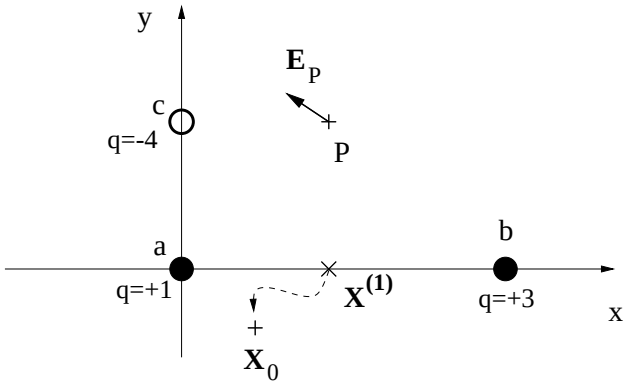

Figure 3.1: Three charges, labeled a, b and c, are placed in the x–y plane. The electric field at point P, $\mathbf{E}_P$, is the sum of the electric field from each charge. The electric field vanishes ($\mathbf{E} = 0$) at point $\mathbf{x}_0$. This point is found numerically beginning with an initial guess $\mathbf{x}^{(1)}$ and iterating until both equations $E_x(x, y) = 0$ and $E_y(x, y) = 0$ are solved.

A familiar example of a vector-valued function is the electric field from a collection of point charges. Figure 3.1 shows an example where three charges (two positive and one negative) are placed in the $x - y$ plane. The electric field at a point $\mathbf{x} = (x, y)$ is a vector

$$\mathbf{E}(\mathbf{x}) \;=\; (\, E_x(x,y),\, E_y(x,y)\,) \quad , \tag{3.1}$$

where the E_z component can be ignored since it will always be zero within the x–y plane. The electric field is the sum of the Coulomb's-law electric field from each charge:

$$\mathbf{E}(\mathbf{x}) \;=\; \frac{1}{4\pi\epsilon_0}\left[\, q_a \frac{\mathbf{x}-\mathbf{x}_a}{|\mathbf{x}-\mathbf{x}_a|^3} + q_b \frac{\mathbf{x}-\mathbf{x}_b}{|\mathbf{x}-\mathbf{x}_b|^3} + q_c \frac{\mathbf{x}-\mathbf{x}_c}{|\mathbf{x}-\mathbf{x}_c|^3}\right] \tag{3.2}$$

where charges and locations are

$$\begin{aligned} \mathbf{x}_a &= (0,0) \quad , \quad q_a &&= +1\,\mathrm{C} \quad , \\ \mathbf{x}_b &= (1,0) \quad , \quad q_b &&= +3\,\mathrm{C} \quad , \\ \mathbf{x}_c &= (0,0.5) \quad , \quad q_c &&= -4\,\mathrm{C} \quad . \end{aligned} \tag{3.3}$$

The structure of Eq. (3.2) may look unfamiliar, since it is written entirely in Cartesian coordinates. It is more common for textbooks to write Coulomb's law in polar coordinates with the charge at the origin

$$\mathbf{E}(\mathbf{r}) \;=\; \frac{1}{4\pi\epsilon_0}\,\frac{\hat{\mathbf{r}}}{r^2} \quad . \tag{3.4}$$

While polar coordinates are very useful in analytic calculations, they can be quite confusing to do real work with. For instance the unit vector $\hat{\mathbf{r}}$ points in a different direction at different points. They are most useful when the problem has some underlying symmetry; this same symmetry is usually the only reason the problem can be solved analytically at all. Numerical solutions are sought when there is no symmetry; this is also when polar coordinates become a liability rather than an asset. Thus when working with a computer you are almost always advised to **work in**

Cartesian coordinates. The following definitions allow polar expressions to be quickly translated to Cartesian coordinates:

$$\mathbf{r} = x\hat{\mathbf{x}} + y\hat{\mathbf{y}} \tag{3.5}$$

$$r = |\mathbf{r}| = \sqrt{x^2 + y^2} \tag{3.6}$$

$$\hat{\mathbf{r}} = \frac{\mathbf{r}}{|\mathbf{r}|} \tag{3.7}$$

This is exactly what has been done to write out the electric field in Eq. (3.2).

Implementing the mathematical expressions Eq. (3.2) and (3.3) as a `Matlab` program is straightforward. As in the program `mars_angle`, we first define the constants in the problem ϵ_0, q_a, $\mathbf{x}_a$ etc., and then calculate the electric field (in Volts per meter)

..... Begin Program 8

file: `efld.m`

```
function e = efld( x )
%  return the electric field from three charges located on the x-y plane
%

eps0 = 8.8542e-12;       %  permittivity of free space (Farads/meter)

% charge a
xa = [ 0.0, 0.0 ];       %  location
qa = 1.0;                %  magnitude

% charge b
xb = [ 1.0, 0.0 ];       %  location
qb = 3.0;                %  magnitude

% charge c
xc = [ 0.0, 0.5 ];       %  location
qc = -4.0;               %  magnitude

% calculate relative displacements
ra = x-xa;
ramag = sqrt( ra(1)^2 + ra(2)^2 );   % ra = | x - xa |
rb = x-xb;
rbmag = sqrt( rb(1)^2 + rb(2)^2 );
rc = x-xc;
rcmag = sqrt( rc(1)^2 + rc(2)^2 );

%  calculate electric field
e = ( qa*ra/ramag^3 + qb*rb/rbmag^3 + qc*rc/rcmag^3 )/(4*pi*eps0);
```

..... End Program 8

Note that with `Matlab`, which is particularly apt at manipulating vectors,[3] it is possible to perform operations on every component with a single statement. The statement

```
ra = x-xa;
```

creates a new vector variable `ra` and then sets each of its components. In languages like FORTRAN or C such an assignment could only be accomplished one component at a time like

[3] Also at manipulate matrices, which is where its name comes from.

```
ra(1) = x(1) - xa(1);
ra(2) = x(2) - xa(2);
```

By calling this program from the main level we can determine the exact electric field at any point in space:

```
>> ep = efld( [ 0.5, 0.5 ] )
```

passes the input argument `x=[0.5,0.5]`. The last line of Program 8 yields the vector value

```
ep =

  1.0e+011 *

  -1.6922     0.5084
```

which means that electric field at point P is

$$\mathbf{E}_P = (-1.7 \times 10^{11},\, 0.51 \times 10^{11}\,) \ \text{V/m} \ .$$

This two-dimensional vector is returned to the main level where it is assigned to the new variable `ep`.

3.2 Root finding with vector-valued functions

Vector-valued functions of vector arguments allow us to generalize our root-finding recipe to solve several equations for several unknowns. First write the m unknowns as an m-dimensional vector

$$\mathbf{x} = (\, x_1,\, x_2,\, \ldots,\, x_m\,) \ .$$

In a well-posed problem we should have m equations for these m unknowns. Setting to zero the right hand sides of each equation defines m functions f_i of m-unknowns

$$\begin{aligned} f_1(\, x_1,\, x_2,\, \ldots,\, x_m\,) &= 0 \ , \\ f_2(\, x_1,\, x_2,\, \ldots,\, x_m\,) &= 0 \ , \\ &\vdots \\ f_m(\, x_1,\, x_2,\, \ldots,\, x_m\,) &= 0 \ . \end{aligned}$$

Using vector notation this system looks like the one-dimensional version, except with bold-face characters

$$\mathbf{f}(\mathbf{x}) \ = \ 0 \ . \tag{3.8}$$

Our task is to find the m-values $x_1, \ldots, x_m$ which make all m functions vanish at the same moment. Of course, if $m = 1$ we have the problem we solved in Chapter 2.

Difficulties in more dimensions

Our first hint of a problem comes generalizing step 2 of the procedure: "Graph the function". We have years of experience graphing scalar-valued functions of scalar arguments, i.e. making "x–y plots", but much less experience with higher dimensions. For two dimensions ($m = 2$) it is *possible* but not simple to graph a function. One method is to draw little arrows at selected points in the

plane; this is sometimes called "a pincushion plot", for its resemblance to a jumbled cushion full of pins. Making and interpreting pincushion plots is more complicated than a simple x–y plot we are familiar with. First we must choose at which points to draw the arrows — perhaps on a grid? but how fine a grid? Choosing incorrectly could prevent you from spotting a solution on the graph — assuming you even knew what the graph would look like where $\mathbf{f}(\mathbf{x}) = 0$. Even if you became an expert at making and reading pincushion plots, they only work in two-dimensions. For $m \geq 3$ there is simply no way of "graphing" functions. The human eyes and brain are simply not designed to comprehend things in so many dimensions. We find ourselves much more at the mercy of the computer in these high-dimensional spaces.

The discussion above hints at a further difficulty in $m \geq 2$ dimensions: a solution cannot be truly "bracketed". In one dimension two points will define a bracket inside which we are guaranteed to find at least one solution. In two dimensions it is possible (if you know much more mathematics) to show that a closed curve can enclose a solution. But this fact is of no practical use since a closed curve has an infinite number of points, rather than just two! For this reason there is no analog of the bisection method for $m \geq 2$.

A multi-dimensional analog of Newton's method does exist and it is the preferred method for professional computational physicists. As in the one-dimensional case, however, Newton's method needs to evaluate function, in this case an m-dimensional vector, *and* its derivative. The derivative of a vector-valued function is an $m \times m$ matrix called the Jacobian. If this sounds complicated it is.

Making life simpler

Fortunately there is a way of restating the root-finding problem which makes it much easier to solve. Consider the scalar-valued function of vector argument

$$\phi(\mathbf{x}) \equiv |\mathbf{f}(\mathbf{x})|^2 = \sum_{i=1}^{m} f_i^2(\mathbf{x}) \quad . \tag{3.9}$$

From basic properties of the vector norm $|\cdot|$ we know that $\phi \geq 0$ and that $\phi = 0$ only if every component of $\mathbf{f}$ equals zero at the same time. Therefore, points where $\phi(\mathbf{x})$ is an absolute minimum are points which satisfy all m equations (3.8). This recasts our original problem of solving m equations for m unknowns as a different problem: **minimization**.

Minimizing a function of m unknowns is much easier than solving m equation; it is literally as easy as rolling down hill. Consider the following algorithm. Starting from an m-dimensional initial guess $\mathbf{x}^{(1)}$ first evaluate the scalar function: $\phi(\mathbf{x}^{(1)})$. If this is not zero then "search" the neighborhood of $\mathbf{x}^{(1)}$ to learn in which direction $\phi(\mathbf{x})$ gets smaller. Move to a new point $\mathbf{x}^{(2)}$ which is in the downhill direction from $\mathbf{x}^{(1)}$. Evaluate $\phi(\mathbf{x}^{(2)})$ to assure that it is smaller than $\phi(\mathbf{x}^{(1)})$; if it is not, you must take a new step from $\mathbf{x}^{(1)}$ which is in the same direction but smaller. This procedure is then repeated so that $\phi(\mathbf{x}^{(n)})$ gets smaller with each iteration. Terminate the repetition at $\mathbf{x}_* = \mathbf{x}^{(n)}$ when no direction from $\mathbf{x}_*$ is downhill; you have reached a local minimum in the function $\phi(\mathbf{x})$.

The algorithm described above is quite clever, and (almost) guaranteed to find a local minimum of the function $\phi(\mathbf{x})$. It is slightly messy to program, but fortunately `Matlab` contains a black box program called `fminsearch`[4] which implements the algorithm. `fminsearch` is a general program to minimize a scalar function of m variables. It is our responsibility to write a specific program which

[4]In older versions of `Matlab`, this program was called `fmins`.

evaluates the function ϕ given a vector input argument $\mathbf{x}$. We then call the program `fminsearch` with the name of our evaluation programs (as a string variable) and our initial guess $\mathbf{x}^{(1)}$.

Example

As an example let's find the point or points where the electric field from `efld` equals zero. In other words we wish to solve the two equations

$$E_x(x,y) = 0 \quad , \tag{3.10}$$
$$E_y(x,y) = 0 \quad , \tag{3.11}$$

for the two unknowns x and y. To play our minimization trick we define the scalar-valued function

$$\phi(\mathbf{x}) = E_x^2(\mathbf{x}) + E_y^2(\mathbf{x}) \quad , \tag{3.12}$$

whose minima should solve equations (3.10) and (3.11) simultaneously. A `Matlab` implementation of this is very simple if we use `efld` to find each component

..... Begin Program 9

file: `efld_mag2.m`

```
function phi = efld_mag2( x )
%
%  return the magnitude squared of the electric field at point x

e = efld( x );
phi = e(1)^2 + e(2)^2;  %   phi = |E|^2 = E_x^2 + E_y^2
```

..... End Program 9

To find the minimum of this function we simply pass its name and an initial guess to the `Matlab` program.

```
>> x0 = fminsearch( 'efld_mag2', [ 0.5, 0.0 ] )
```

Upon returning `x0` is set to $(\,0.2639,\,-0.1911\,)$.

The point returned by `fminsearch` should be the location where $\mathbf{E} = 0$. To verify this we use the answer, `x0`, as the input argument to `efld`:

```
>> e0 = efld( x0 )
```

This returns an electric field of

$$\mathbf{E}(\mathbf{x}_0) = (-0.0095, -0.0040) \quad . \tag{3.13}$$

Clearly this is not exactly zero; it is, however, smaller than the electric field at point P by fourteen orders of magnitude. Furthermore, the electric field at points very near $\mathbf{x}_0$

```
>> e1 = efld( x0 + [ 0.01, 0.0 ] )
>> e2 = efld( x0 + [ 0.0, 0.01 ] )
```

are all much bigger than Eq. (3.13). Thus it seems reasonable to assume that `fminsearch` has given an accurate approximation to the solution of equations (3.10) and (3.11).

This exercise has much in common with our experiences in one dimension. The iterative procedure solves equations (3.8) only approximately. Iterations are ended when some tolerance is satisfied. This tolerance is probably specified by the relative rather than absolute error. Thus while $|E_x| = 0.0095$ seems rather large, it is very small compared to the typical size of $|E_x|$.

It turns out that the point $\mathbf{x}_0$ is not the only location where $\mathbf{E}(\mathbf{x}) \simeq 0$. The electric field vanishes at great distances from the charge, $|\mathbf{x}| \gg 1$.[5] This is a less interesting solution to (3.10) and (3.11), but for many initial guesses $\mathbf{x}^{(1)}$ it is where the minimizing algorithm rolls to:

```
>> xx = fminsearch( 'efld_mag2', [ 0.5, 0.5 ] )

xx =

   1.0e+015 *

    4.1191    0.7667
```

While this is not $|\mathbf{x}| = \infty$ it is clearly far from all of the charges. Evaluating $\mathbf{E}(\mathbf{x})$ at this point `xx` confirms that the electric field is very small there. Thus, exactly as in one dimension, a careful choice of the initial guess $\mathbf{x}^{(1)}$ is the only way to find the solution which you are after.

[5] Since $q_a + q_b + q_c = 0$ the leading multipole moment is the dipole, and $|\mathbf{E}| \sim r^{-3}$ as $r \to \infty$.

Chapter 4

Solving an Ordinary Differential Equation

4.1 The general problem

Almost every area of physics, from Newton's first law to Einstein's equations for general relativity, is expressed in the language of differential equations. The term *ordinary differential equation* or its abbreviation ODE, refers to one or more equations where all derivatives are taken with respect to a single variable, called the *independent variable*. In Newton's laws the independent variable is usually time. For example, a sky diver jumps from an airplane and falls at velocity $\mathbf{v}$. This velocity changes according to Newton's law

$$\frac{d\mathbf{v}}{dt} \;=\; \frac{\mathbf{F}}{M} \quad . \tag{4.1}$$

The total force $\mathbf{F}$ acting on the sky diver consists of both the downward force of gravity $F_g = Mg$, and the force of aerodynamic drag (air resistance) which acts in the direction opposite to the velocity (in this case up). The general expression for aerodynamic drag force on a body moving with velocity $\mathbf{v}$ relative to the air is

$$\mathbf{F}_D = -\tfrac{1}{2} C_D \rho A |\mathbf{v}| \mathbf{v} \quad , \tag{4.2}$$

where $\rho \simeq 1.2\,\mathrm{kg/m^3}$ is the mass density of air and A is the cross-sectional area of the body.[1] An adult, falling in belly-flop position will have a cross section about $A \simeq 1\,\mathrm{m}^2$. C_D is a dimensionless number called the *drag coefficient*, which depends on details of the object, such as its shape, and the roughness of its surface. Values of C_D for different shapes are found experimentally using wind tunnels. Actually, C_D falls in the range $0.1 \le C_D \le 10$ for most things, unless they are specially designed to be stream-lined. In what follows we will take the representative value $C_D = 1$.

Combining the gravitational force and the drag force gives an ODE for the downward component of the velocity of the falling sky diver

$$\frac{dv}{dt} \;=\; g \;-\; \alpha\, v^2 \quad , \tag{4.3}$$

where $\alpha = \frac{1}{2} C_D \rho A / M = 0.006\,\mathrm{m^{-1}}$ after taking the mass of the man (and parachute!) to be $M = 100$ kg. This is an example of an ODE where the independent variable is time t. Its solution, the function $v(t)$, depends on the initial condition $v(0) = v_0$.

[1] In other words the area presented to the oncoming air.

4.2 Time stepping — Euler's method

A first order ODE, like that for the sky-diver, can be written in its most general form as

$$\frac{dy}{dt} \;=\; f(y,t) \quad , \tag{4.4}$$

where the function $f(y,t)$ contains all of the algebraic terms. For the case of the falling sky-diver the function

$$f(v,t) \;=\; g - \alpha\, v^2 \;=\; 9.8 \;-\; 0.006\, v^2 \quad , \tag{4.5}$$

does not actually depend on time. The ODE (4.4) actually gives a simple recipe for its own approximate solution. Replacing the derivatives by the ratio of differences

$$\frac{\Delta y}{\Delta t} \;\simeq\; \frac{dy}{dt} \;=\; f(y,t) \quad , \tag{4.6}$$

reminds us that the function $f(y,t)$ describes the change in the solution for a given change in t. Although this is only strictly correct in the limit $\Delta t \to 0$ we will apply it for a non-vanishing intervals Δt.

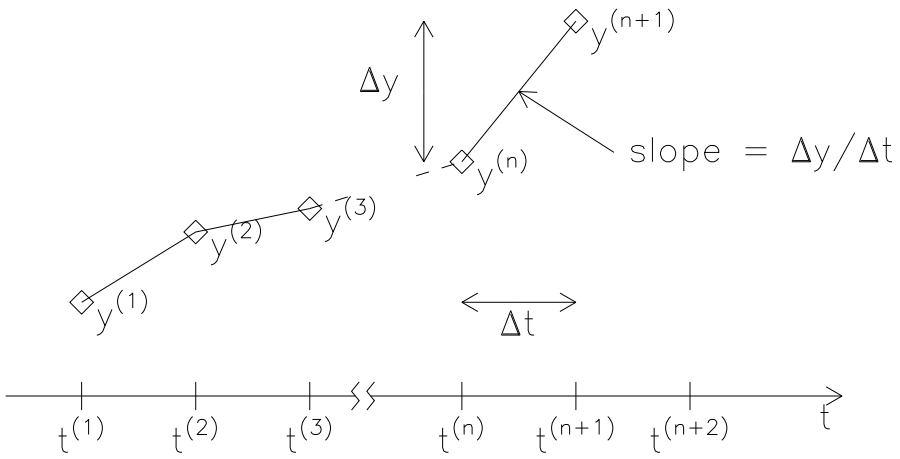

Figure 4.1: The time axis discretized at time-levels $t^{(1)}$, $t^{(2)}$, ..., $t^{(n)}$, $t^{(n+1)}$, ..., separated by time-step Δt. Euler's method solution values $y^{(1)}$ etc. are shown by diamonds. Step n advances from $y^{(n)}$ to $y^{(n+1)}$ along the solid line with slope $\Delta y/\Delta t$.

Consider a set of discrete times $t^{(n)}$ separated by the fixed interval Δt, called the *time-step*. Taking the first time $t^{(1)} = 0$ the subsequent time-levels will be

$$t^{(n)} \;=\; (n-1)\,\Delta t \quad . \tag{4.7}$$

We wish to find the value of the solution $y(t)$ at each of the discrete time-levels (see Fig. 4.1), denoting the value at time level n

$$y(t^{(n)}) \;\equiv\; y^{(n)} \quad . \tag{4.8}$$

Interpreting the differential equation (4.6) in terms of the discrete values $t^{(n)}$ and $y^{(n)}$ gives the prescription

$$\Delta y \;=\; y^{(n+1)} \;-\; y^{(n)} \;=\; \Delta t\, f(y,t) \quad . \tag{4.9}$$

Choosing both arguments in $f(y,t)$ from time-level n gives the explicit form of *Euler's Method*

$$\boxed{y^{(n+1)} \;=\; y^{(n)} \;+\; \Delta t\, f(y^{(n)},\, t^{(n)})} \quad . \tag{4.10}$$

Euler's method is a recipe for advancing the solution from $y^{(n)}$ at time $t^{(n)}$ to its next value $y^{(n+1)}$ at time $t^{(n+1)}$. Note that expression (4.10) gives the later value, $y^{(n+1)}$, explicitly in terms of previous values $y^{(n)}$ and $t^{(n)}$.

Let us demonstrate this using the sky diver problem given by $f(v,t)$ in Eq. (4.5). We will employ Euler's method using the time step $\Delta t = 1$ second, yielding

$$v^{(n+1)} \;=\; v^{(n)} + \Delta t\, f(v^{(n)}, t^{(n)}) \;=\; v^{(n)} + 9.8 - 0.006\,[v^{(n)}]^2 \quad . \tag{4.11}$$

This recipe is applied repeatedly beginning with the initial condition $v^{(1)} \;=\; v(0)$. If the sky-diver begins from rest then $v^{(1)} = 0$. The next value $v^{(2)}$ is given by expression (4.11) taking $n = 1$:

$$v^{(2)} \;=\; v^{(1)} + 9.8 - 0.006\left[v^{(1)}\right]^2 \;=\; 0 + 9.8 - 0 \;=\; 9.8 \quad .$$

The next value, $v^{(3)}$ comes from repeating this with $n = 2$:

$$v^{(3)} \;=\; v^{(2)} + 9.8 - 0.006\left[v^{(2)}\right]^2 \;=\; 9.8 + 9.8 - 0.006\,(9.8)^2 \;=\; 19.024 \quad .$$

This procedure can be iterated several more times with a hand calculator; the results are given in Table 4.1 and graphed in Fig. 4.2. For comparison the solution neglecting air resistance, $v(t) = gt$,

n	1	2	3	4	5	6	7	8	9
$t^{(n)}$	0	1	2	3	4	5	6	7	8
$v^{(n)}$	0.000	9.800	19.024	26.652	32.190	35.773	37.895	39.079	39.716
$g\,t^{(n)}$	0.0	9.8	19.6	29.4	39.2	49.0	58.8	68.6	78.4

Table 4.1: The result of solving the sky-diver problem using Euler's method with time-step $\Delta t = 1$ s. For comparison, the result without air resistance, $v = gt$, are given in the bottom row.

is shown as a dashed line. Note that with air resistance the diver approaches a terminal velocity of magnitude ~ 40 m/s (~ 90 miles per hour).

The computer program

Like the root solving algorithms, Euler's method is simple in principle, but involves a great deal of tedious calculation in practice. This is a perfect opportunity to do some more `Matlab` programming. Our goal is to write a general program which uses the recipe in Eq. (4.10) to solve a first-order ODE. The ODE will be specified by the function $f(y,t)$ which must also be written as a `Matlab` program — this is the specific program. For the sky-diver problem the specific program is given below.

..... Begin Program 10

file: `sky_diver.m`

```
function f = sky_diver( v, t )
%
%  return the downward acceleration (m/s^2) of a sky-diver
%  at time t (s) with a downward velocity v (m/s).
```

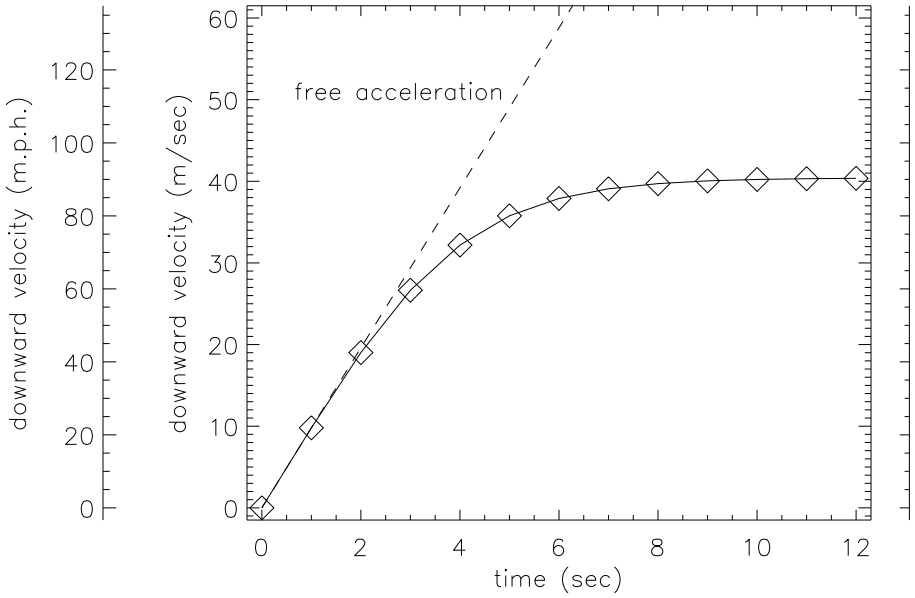

Figure 4.2: Solutions of the sky-diver problem by Euler's method with $\Delta t = 1$ s, are shown with diamonds. The analytic solution for the case of zero air resistance is shown as a dashed line.

```
%

g = 9.8;   %  acceleration of gravity (m/s^2)
rho = 1.2; %  mass density of air (kg/m^3)
a = 1.0;   %  cross-sectional area of sky-diver (m^2)
C_d = 1.0; %  coefficient of drag
M = 100.0; %  mass of diver (kg)

alpha = 0.5*rho*a*C_d/M;

f = g - alpha*abs(v)*v;
```

..... End Program 10

- `sky_diver` takes two input arguments `v` and `t`, but the second one, `t`, is not used at all. Why is it there? It is there to fool the general Euler's method program (below) which assumes the function $f(y,t)$ takes two input arguments and will therefore pass them both. If `sky_diver` was defined to only accept one argument, there would be trouble as a second argument was "force-fed" to it.

- This program has many lines to set constants which ultimately get multiplied together. The same multiplication will happen over and over every time `sky_diver` is called. A far more efficient program would contain the single line

  ```
  f = 9.8 - 0.006*abs(v)*v;
  ```

  We have chosen the *computationally* inefficient route so that we (the programmers) might remember how the numbers in that formula were derived. While this choice makes the

computer's life unpleasant it might very well make our lives much easier if we decide in the future to, for example, use a lighter sky diver, $M = 80$ kg. Given a choice between the convenience of the programmer or that of the computer, the programmer should take a very one-sided view.

- Instead of v^2, which is correct only for downward velocities, we use $|v|v$, which is correct for both upward and downward. We don't anticipate an upward velocity, but why not do it correctly?

The general program to implement Euler's method is comparatively simple

..... BEGIN PROGRAM 11

file: `euler_1d.m`

```
function y = euler_1d( y0, t0, dt, n_steps, deriv_func )
%
%  take n_steps time steps, of size dt, in the differential equation
%    dy/dt = f( y, t ) = deriv_func( y, t )
%  begining with y(1) = y0 and time t = t0.  return the entire vector
%  y = [ y(1), y(2), ..., y(n_steps+1) ]
%

y = 0:n_steps;  %  create a vector to put answers into

y(1) = y0;  % initial condition

for n = 1:n_steps
   t = t0 + dt*(n-1);                    % current time
   f = feval( deriv_func, y(n), t );     % evaluate function f(y,t)
   y(n+1) = y(n) + dt*f;                 % Euler's method
end
```

..... END PROGRAM 11

The program `euler_1d` returns its answer as a vector of dimension $N_{\text{steps}} + 1$. The value at time step n will be stored in vector component n

$$\texttt{y(n)} = y^{(n)} \quad .$$

This vector variable must first be created and then it can be filled with the correct answers one element at a time. An easy way to create a $N_{\text{steps}} + 1$ vector is with `Matlab`'s `:` operator.[2] The command

```
y = 0:n_steps;
```

creates `y` as a vector with dimension `n_steps+1`, and then fills it with the integers 0, 1, ..., N_{steps}. These values are unimportant since they will be changed during the calculation — it's just an easy way to create a vector.

`euler_1d` can be called to quickly re-produce the values in the table. Be careful to include every argument in its rather extensive argument list... *in the correct order.*

```
>> y = euler_1d( 0, 0, 1, 8, 'sky_diver' )
```

[2]This and many other array tricks are explained in Appendix C.

The values in vector `y` may be plotted, but first we need a vector with values of the time $t^{(n)}$. This can be created with a `Matlab` command similar to the one which created `y`

```
>> t = 0:8;
>> plot( t, y, 'o' )
```

Here we use the colon to create `t` with values 0, 1, 2, ..., 8. Note that the third argument to `plot`, is a string which indicates the kind of symbol to use: `'o'`. Try a few others like `'+'`, `'*'` or `'-'`. If you omit the third argument, typing simply `plot( t, y )`, it will use a solid line; this is the *default*.

4.3 Truncation errors and order of accuracy

Euler's method provides values $y^{(n)}$ which are *approximations* to the exact solution of the differential equation

$$\frac{dy}{dt} \;=\; f(y,t) \quad , \tag{4.12}$$

evaluated at times $t^{(n)}$. Let us denote the (unknown) exact solution $\bar{y}(t)$. Figure 4.3 shows the exact solution $\bar{y}(t)$ as a dashed line, and the Euler step $y^{(n+1)}$ as the solid line connecting diamonds at $y^{(n)}$ and $y^{(n+1)}$. We assume that $y^{(n)}$ matches the exact solution, however, the Euler step to $y^{(n+1)}$ diverges from the exact solution $\bar{y}(t^{(n+1)})$. The separation of these two values is called the *truncation error* $\epsilon_{\rm tr}$. It is the amount of error in a single iteration.

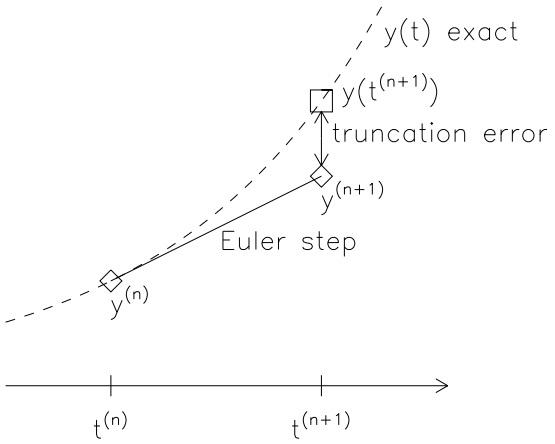

Figure 4.3: A single Euler step, from $y^{(n)}$ to $y^{(n+1)}$ (diamonds). The exact solution $\bar{y}(t)$ is shown by a dashed line. Its value at $t^{(n+1)}$, namely $\bar{y}(t^{(n+1)})$, is shown by a square.

Using a Taylor expansion of $\bar{y}(t)$ we can write the exact answer

$$\begin{aligned} \bar{y}(t^{(n+1)}) &= \bar{y}(t^{(n)} + \Delta t) \\ &= \bar{y}(t^{(n)}) + \Delta t\, \bar{y}'(t^{(n)}) + \tfrac{1}{2}(\Delta t)^2\, \bar{y}''(t^{(n)}) + \tfrac{1}{6}(\Delta t)^3\, \bar{y}'''(t^{(n)}) + \cdots \end{aligned} \tag{4.13}$$

Euler's method settles for the approximate answer

$$y^{(n+1)} = \bar{y}(t^{(n)}) + \Delta t\, f(\, \bar{y}(t^{(n)}),\, t^{(n)}\,) = \bar{y}(t^{(n)}) + \Delta t\, \bar{y}'(t^{(n)}) \quad . \tag{4.14}$$

since $\bar{y}' = f(\bar{y}, t)$ by equation (4.12). The error in this approximation

$$\epsilon_{\rm tr} \;\equiv\; \bar{y}(t^{(n+1)}) - y^{(n+1)} \;=\; \tfrac{1}{2}(\Delta t)^2\, \bar{y}''(t^{(n)}) \;+\; \cdots \;=\; \mathcal{O}(\Delta t^2) \tag{4.15}$$

is called the *truncation error* since it comes about from truncating the Taylor expansion after a finite number of terms (namely 2). The notation $\mathcal{O}(\Delta t^2)$ refers to a sum of terms whose *lowest* power of Δt is the second.

Euler's method makes a truncation error which scales as $(\Delta t)^2$. We don't know the exact value of the error since we don't know, for example $\bar{y}''$. Nevertheless, we do know that if we take time steps half as big, $\Delta t = 0.5$ s, the truncation errors at each step should be roughly one-quarter the size. Unfortunately, taking steps half as big we need to take twice as many to integrate Eq. (4.12) to the same time. This means we make twice as many errors, which are one-quarter as large. The total error should therefore be only one half as large. In other words the total error scales as Δt.

By extension, taking time steps one-tenth as big, $\Delta t = 0.1$ s, will result in ten times as many steps, each with an error 1% as large. The end result is an error only one tenth as large as the original. Figure 4.4 shows the results of integrating the sky-diver problem with $\Delta t = 0.5$ s, $\Delta t = 0.1$ s, as well as $\Delta t = 2$ s, for comparison.

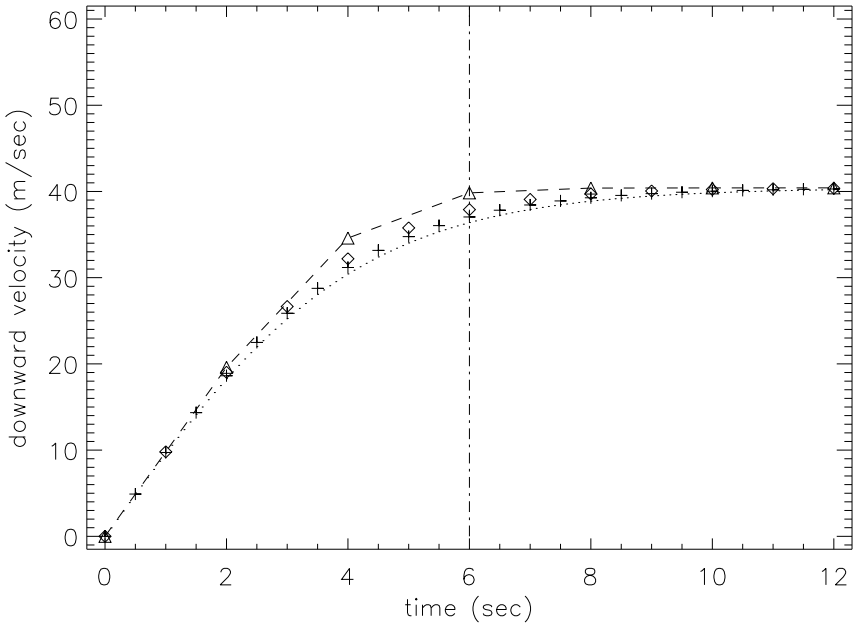

Figure 4.4: A comparison of Euler's method with $\Delta t = 2$ s (triangles), $\Delta t = 1$ s (diamonds), $\Delta t = 0.5$ s (pluses), and $\Delta t = 0.1$ s as points. The $\Delta t = 0.1$ s points form a dotted line which can be taken as the exact solution. The vertical dot-dashed line shows the time $t = 6.0$ where the different time-steps are later compared.

To see the accumulated truncation errors consider a single time, say $t = 6.0$ s. Running `euler_1d` with different values of Δt gives different solutions $y^{(n)}$ at $t^{(n)} = 6.0$. The table below shows several of these, along with the total error $|\bar{y}(6.0) - y^{(n)}|$ in the solution (the differential equation happens to have an analytic solution). Scanning across the table, the convergence of the approximate solution $y^{(n)}$ is easy to see, even without comparison to the exact solution. The total error is decreasing, and at a rate comparable to Δt (for instance, compare the errors at $\Delta t = 1$ s,

Δt	2	1	0.5	0.25	0.1	0.01	exact
steps $(n-1)$	3	6	12	24	60	600	∞
$y^{(n)}$	39.8324	37.8947	37.0420	36.6346	36.3955	36.2538	36.2381
error	3.5942	1.6565	0.80384	0.39643	0.15732	0.015649	—

0.1 s and 0.01 s.) Plotting these values (see Figure 4.5) shows that the error (actually the error divided by the exact result — the relative error) does scale almost exactly as $\propto \Delta t$. Also note how the result of the truncation error $\epsilon_{\rm tr}$ is far greater than the roundoff error $\epsilon_{\rm M} \simeq 10^{-16}$ — this is a clear case where round-off is not a significant factor.

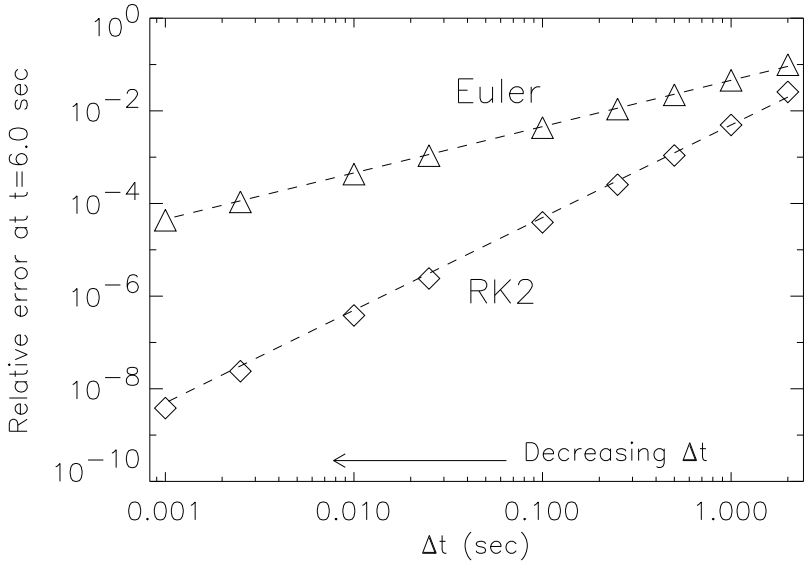

Figure 4.5: The relative error, Error/$\bar{y}$, from the solution of the sky-diver problem to $t = 6.0$ s. The relative error is plotted against time step for both Euler's method (triangles) and RK2 (diamonds). Dashed lines show the theoretical fit Δt and $(\Delta t)^2$ respectively.

The bottom line is that Euler's method works by advancing the solution by discrete time-steps Δt. The smaller the time step the more accurate is the final solution. The trade-off is that taking very small time-steps means it will take very many steps to advance to a particular time. In the end one must choose a time step which is a good compromise between accuracy and speed.

Choosing the time step

For a given problem it is not always obvious what the best choice of Δt might be. The best way to start is to try to guess how fast you expect the actual solution $\bar{y}(t)$ to change; what is the *natural time-scale* of the problem. From approximate solutions to the sky-diver problem we see that the falling person is near terminal velocity after about 10 seconds; this is the natural time scale. Even before attempting a solution we might have guessed this from our experience — at least we might have guessed it took longer than 1 second (Many of us have jumped from tree-branches and never reached terminal velocity.).[3] For these purposes you need to *conservatively* estimate the natural

[3]It is also possible to estimate the natural time scale by dimensional analysis, but this would take us on a bit of a tangent — albeit a very interesting one.

time scale to a factor of ten. Then take a time step which is a great deal smaller than the natural time scale. Here some judgment is called for. We might say that 0.1 s is far smaller than 10, but you may be tempted to go farther, to say 0.01 s. A second element of judgment may impinge at this point — choosing 10^{-6} s will mean taking ten million time steps; you will be waiting for considerable time to see you answer.

After solving your equation with the small time step, **repeat the solution with a still smaller time step**, say half as big. Comparing the two results you should see some evidence that the solutions are "converging" to the exact answer. Figure 4.4 shows an example of such convergence. This will give you some confidence in your result.

A common line of reasoning is to work backwards from the total number of time-steps you are willing to wait for. You can usually take $\sim 10^3$ to 10^4 steps without really waiting for the computer. Runs of $\sim 10^5$ steps can take several minutes, i.e. bring something to read while you wait. Taking 10^6 or more might require an hour or more. Runs like this should only be attempted as a last resort.

It is worth noting that the penalty for misjudging the natural time-scale can be far worse than simple inaccuracy. Taking very large time steps can often result in *bizarre, pathological behavior*. Say we tried to do the sky-diver problem with $\Delta t = 8$ s. The first few time steps show some truly unexpected behavior: At time step 3 the downward velocity component is negative — the sky-diver

n	1	2	3	4	5	6	7
$t^{(n)}$	0	8	16	24	32	40	48
$v^{(n)}$	0.00	78.4	-138.2	857.4	$-34,350$	6×10^7	-10^{14}

Table 4.2: The first few time steps in a solution of the sky-diver problem using Euler's method with $\Delta t = 8$ s.

is *falling up!* This happens again, with increasing upward velocities, at steps 5 and 7. In fact, by time step 7 the sky-diver is falling upward much faster than the speed of light! This is clearly a crazy result. Had we been foolish enough to continue to time step $n = 11$ (only one and a half minutes of falling) the speed would have exceeded $2^{1023} \simeq 10^{308}$, the largest number possible in double precision — in other words the calculation would have produced an *overflow error*.

Were this your first attempt to solve the problem you would be justified in wondering "Did I type something wrong in `sky_diver.m`?". By this example we seek to introduce a second line of doubt: "Perhaps my time-steps are too big." Cutting back drastically on the time steps, say to $\Delta t = 1$ s from $\Delta t = 8$ s, completely clears up this strange behavior.

4.4 A second algorithm — The Runge-Kutta method

As with root solving, there is more than one algorithm to solve an ODE. Euler's method is the simplest algorithm, and there are literally scores of other algorithms which improve upon it in some way. Because Euler's method matches the Taylor series of the exact solution up to and including the $\mathcal{O}(\Delta t)$ term, it is called *first-order accurate*. In other words its truncation error is second order in Δt: $\epsilon_{\text{tr}} = \mathcal{O}(\Delta t^2)$. Many of the more sophisticated algorithms match more terms in the Taylor series, and are called *higher-order* methods. One example in the so-called *Second-order Runge-Kutta* algorithm, or RK2 for short. As the name implies this algorithm has a truncation error $\epsilon_{\text{tr}} = \mathcal{O}(\Delta t^3)$.

The RK2 algorithm begins with $y^{(n)}$ at time $t^{(n)}$ and advances one time-step Δt to produce $y^{(n+1)}$. It does this by making a first guess at the solution, called y^* at time $t^{(n)} + \frac{1}{2}\Delta t$. It then

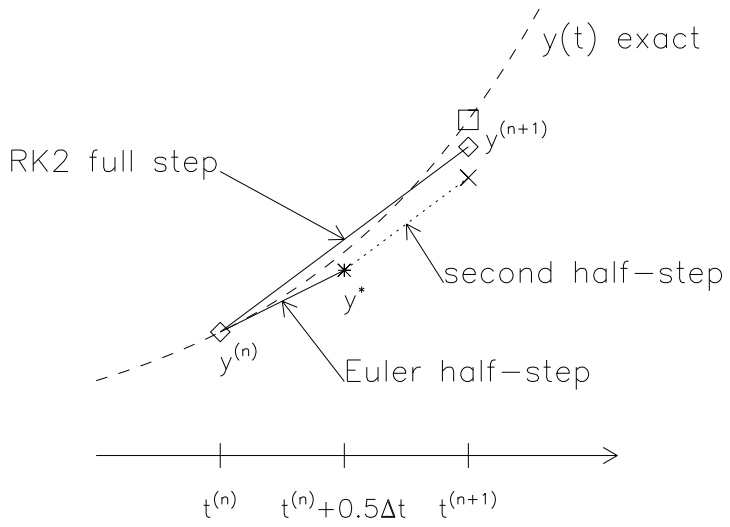

Figure 4.6: A single step of RK2 from $y^{(n)}$ to $y^{(n+1)}$, shown by diamonds. The initial half Euler step to y^* (marked with an asterisk) is show by a solid line. A second Euler half step, Eq. (4.17') is shown by a dotted line ending at an X. Instead of this, RK2 takes a full step with exactly the same slope, but beginning at $y^{(n)}$.

uses this to back up and take one (better) full time step. This procedure is a kind of two-step:

$$\boxed{y^* = y^{(n)} + \tfrac{1}{2}\Delta t\, f(\, y^{(n)},\, t^{(n)}\,)} \tag{4.16}$$

$$\boxed{y^{(n+1)} = y^{(n)} + \Delta t\, f(\, y^*,\, t^{(n)} + \tfrac{1}{2}\Delta t\,)} \tag{4.17}$$

After $y^{(n+1)}$ is found, the intermediate quantity y^* is discarded, it is not part of the final solution.

The preliminary step, (4.16) is just an Euler time-step of size $\frac{1}{2}\Delta t$. If this were simply followed by a second Euler time step of the same size

$$y^{(n+1)} = y^* + \tfrac{1}{2}\Delta t\, f(\, y^*,\, t^{(n)} + \tfrac{1}{2}\Delta t\,) \quad , \tag{4.17'}$$

the result would have a truncation error

$$\epsilon_{\rm tr} \;=\; \tfrac{1}{4}(\Delta t)^2 \left[\, \bar{y}''(t^{(n)}) \;+\; \bar{y}''(t^{(n)} + \tfrac{1}{2}\Delta t)\,\right] \;+\; \mathcal{O}(\Delta t^3) \tag{4.18}$$

which is still $\mathcal{O}(\Delta t^2)$, since all we would be doing is taking twice as many Euler steps of half the size. Equation (4.17) is subtly different: it takes a full step Δt, but evaluating the function $f(y,t)$ at the point found by the preliminary step (see Figure 4.6). This subtle difference gives RK2 a truncation error $\mathcal{O}(\Delta t^3)$. This is harder to verify (it is done below, with lots of Taylor expansion) than it is to use. Equations (4.16) and (4.17) can be implemented by making a small modification to program 11

..... Begin Program 12

file: `rk2_1d.m`

```
function y = rk2_1d( y0, t0, dt, n_steps, deriv_func )
%
%  take n_steps time steps, of size dt, in the differential equation
%    dy/dt = deriv_func( y, t )
%  begining with y(0) = y0 and time t = t0.  return the entire vector
%  y = [ y(0), y(1), ..., y(n_steps) ]
%

y = 0:n_steps;  %  create a vector to put answers into

y(1) = y0;  % initial condition

for n = 1:n_steps

% the first half-step
   t = t0 + dt*(n-1);
   f = feval( deriv_func, y(n), t );
   y_star = y(n) + 0.5*dt*f;

% the full step
   t_star = t + 0.5*dt;
   f_star = feval( deriv_func, y_star, t_star );
   y(n+1) = y(n) + dt*f_star;
end
```

..... End Program 12

The truncation error in RK2

Calculating truncation errors is an exercise in Taylor expansion. For higher-order methods this means keeping many terms, and doing lots of algebra. This is done below for RK2 with a bit of rather knotty algebra. Always keep sight of the objective: to calculate the difference between the exact answer $\bar{y}(t^{(n+1)})$ and the approximate answer $y^{(n+1)}$.

The Taylor expansion of the exact answer, (4.13), is

$$\begin{aligned} \bar{y}(t^{(n+1)}) &= \bar{y}(t^{(n)} + \Delta t) \\ &= \bar{y}(t^{(n)}) + \Delta t\, \bar{y}'(t^{(n)}) + \tfrac{1}{2}(\Delta t)^2\, \bar{y}''(t^{(n)}) + \tfrac{1}{6}(\Delta t)^3\, \bar{y}'''(t^{(n)}) + \dots \end{aligned} \tag{4.19}$$

Note that the differential equation provides a way of writing the derivative $\bar{y}'$ in terms of the magic function $f(y,t)$

$$\bar{y}'(t^{(n)}) \;=\; \frac{d\bar{y}}{dt} \;=\; f(\bar{y}, t^{(n)}) \quad . \tag{4.20}$$

Pursuing this logic further, the second derivative $\bar{y}''$ can also be written in terms of f

$$\begin{aligned} \bar{y}''(t^{(n)}) &= \frac{d\bar{y}'}{dt} \;=\; \frac{d}{dt} f[\,\bar{y}(t),\, t\,] \\ &= \frac{d\bar{y}}{dt}\frac{\partial f}{\partial y} \;+\; \frac{\partial f}{\partial t} \;=\; f\, f_y \;+\; f_t \end{aligned}$$

where the total time-derivative of f involves partial derivatives with respect to each argument that depends on time t (and application of the dreaded *chain-rule*). The final expression is written using the subscript notation where $f_t = \partial f/\partial t$, and all functions are evaluated at $\bar{y}(t^{(n)})$ and $t^{(n)}$, which are not written. The exact answer is therefore written

$$\bar{y}(t^{(n+1)}) \;=\; \bar{y}(t^{(n)}) + \Delta t\, f + \underbrace{\tfrac{1}{2}(\Delta t)^2\,[\,f f_y + f_t]}_{A} + \tfrac{1}{6}(\Delta t)^3\, \bar{y}'''(t^{(n)}) + \dots \quad , \tag{4.21}$$

where the $\mathcal{O}(\Delta t^2)$ term has been labeled A.

Next we write the approximate answer $y^{(n+1)}$ in a form similar to Eq. (4.21). Substituting Eq. (4.16) directly into (4.17) gives the expression for the RK2 approximation

$$y^{(n+1)} = \bar{y} + \Delta t\, f\Big[\, \bar{y} + \tfrac{1}{2}\Delta t\, f(\,\bar{y},\, t^{(n)}\,),\, t^{(n)} + \tfrac{1}{2}\Delta t\,\Big] \tag{4.22}$$

where all occurrences of $\bar{y}$ refer to $y^{(n)} = \bar{y}(t^{(n)})$. This rather ugly expression can be improved by performing Taylor expansions on both arguments of $f(y,t)$ at once

$$f(y+a,\, t+b\,) \;=\; f(y,t) \;+\; a\, f_y(y,t) \;+\; b\, f_t(y,t) \;+\; \cdots \quad .$$

Using this formula we can re-write

$$f\Big[\, \bar{y} + \underbrace{\tfrac{1}{2}\Delta t\, f}_{a},\, t^{(n)} + \underbrace{\tfrac{1}{2}\Delta t}_{b}\Big] \;=\; f \;+\; \underbrace{\tfrac{1}{2}\Delta t\, f}_{a} f_y \;+\; \underbrace{\tfrac{1}{2}\Delta t}_{b} f_t \;+\; \mathcal{O}(\Delta t^2) \quad ,$$

where the first few omitted terms involve f_{yy}, f_{tt} and f_{yt} all multiplied by Δt^2. Replacing this in Eq. (4.22) gives

$$y^{(n+1)} = \bar{y} + \Delta t\, f \;+\; \underbrace{\tfrac{1}{2}(\Delta t)^2\,[\, f f_y + f_t]}_{B} \;+\; \mathcal{O}(\Delta t^3) \quad . \tag{4.23}$$

The truncation error $\epsilon_{\rm tr}$ is found by subtracting (4.23) from (4.21). It is clear that the $\mathcal{O}(\Delta t^2)$ terms in (4.23), labeled B, will cancel those in (4.21), labeled A. This is the magic moment for RK2 since only terms with $(\Delta t)^3$ and higher remain, making the truncation error

$$\epsilon_{\rm tr} \;=\; \bar{y}(t^{(n+1)}) - y^{(n+1)} \;=\; \mathcal{O}(\Delta t^3) \quad , \tag{4.24}$$

for RK2. The recipe for RK2, namely Eqs. (4.16) and (4.17), was specially devised so as to make terms A and B cancel.[4]

The $\mathcal{O}(\Delta t^3)$ truncation error is a significant accomplishment. It means that if you decrease the time-step by 0.1, the truncation error decreases by 0.001. Of course you must take ten times as many steps, thereby making ten times as many errors. The net result is a total error one percent of the original. Contrast this with Euler's method where decreasing Δt by 0.1 decreased the final error by one-tenth (i.e. ten percent). This behavior, a total error scaling as $(\Delta t)^2$, is nicely exhibited in Figure 4.5. Clearly RK2 better rewards the extra effort involved in taking smaller time steps. The cost is a slightly longer program (program 12) which makes twice as many calls to `deriv_func`.

[4]In fact there are other two-step recipes that cause this same cancellation, *all* of which go by the name "Second-order Runge-Kutta" method.

Exercise 4.1 Consider the simple ODE

$$\frac{dy}{dt} = \lambda y \ ,$$

where λ is some constant. Find the analytic solution $y(t)$ with the initial condition $y(0) = 1$. Now find the value $y(t = 1)$ by

1. A single step ($\Delta t = 1$) of Euler's method.
2. Two steps ($\Delta t = 0.5$) of Euler's method.
3. A single step ($\Delta t = 1$) of RK2.
4. Two steps ($\Delta t = 0.5$) of RK2.
5. A series expansion of the analytic solution, carried far enough to permit comparison with the answers above.

Chapter 5

Sets of ODEs — Multiple Dimensions

5.1 First order ODEs

Chapter 4 treats the solution of a single ODE for a single unknown function $y(t)$. It is easy to generalize this treatment to a set of m ODEs for m unknown functions all with the same independent variable t. Let us denote the m unknown functions $y_1(t)$, $y_2(t)$,..., $y_m(t)$. We will refer to the entire collection of unknown functions by the vector $\mathbf{y}(t)$ whose m components $y_j(t)$ are each an unknown. The m unknowns satisfy m different ODEs which can be written in terms of m functions f_1, f_2,..., f_m

$$\begin{aligned} \frac{dy_1}{dt} &= f_1(y_1, y_2, \ldots, y_m, t) \\ \frac{dy_2}{dt} &= f_2(y_1, y_2, \ldots, y_m, t) \\ &\vdots \\ \frac{dy_m}{dt} &= f_m(y_1, y_2, \ldots, y_m, t) \end{aligned}$$

or more concisely

$$\frac{dy_j}{dt} = f_j(\mathbf{y}, t) \quad , \quad j = 1, 2, \ldots, m \quad . \tag{5.1}$$

All m equations involve only *first* derivatives; we discuss below how to work with derivatives of higher order such as d^2y_1/dt^2.

Note that a given unknown, say $y_1(t)$ satisfies an equation involving $f_1(\mathbf{y}, t)$ which can depend, in general, on all the other solutions $y_2(t)$, $y_3(t)$ etc. As an example, consider a projectile traveling in two dimensions x, y, and subject to the aerodynamic drag force of Eq. (4.2)

$$\mathbf{F}_D = -\tfrac{1}{2} C_D \rho A |\mathbf{v}| \mathbf{v} \quad , \tag{5.2}$$

and downward gravity $-Mg\,\hat{\mathbf{y}}$. Newton's law leads to the two ODEs for v_x and v_y

$$\frac{dv_x}{dt} = -\alpha \sqrt{v_x^2 + v_y^2}\, v_x = f_1(v_x, v_y) \quad , \tag{5.3}$$

$$\frac{dv_y}{dt} = -g - \alpha \sqrt{v_x^2 + v_y^2}\, v_y = f_2(v_x, v_y) \quad , \tag{5.4}$$

where once again $\alpha = C_D \rho A / 2M$.

Systems of ODEs such as (5.3) and (5.4) often seem much harder to solve, since they can involve incomprehensible webs of interdependencies. In fact, the same numerical algorithms may be applied to them as we applied to the one-dimensional case. We will step forward in time by time step Δt, denoting the time of the n^{th} step $t^{(n)} = t^{(1)} + (n-1)\Delta t$. The m solutions at this time are simply

$$y_j(t^{(n)}) \;=\; y_j^{(n)} \quad . \tag{5.5}$$

The most difficult aspect of this work is to avoid confusing the subscripts (for vector component) with the superscripts (for time step). Euler's method for multiple dimensions,

$$y_j^{(n+1)} \;=\; y_j^{(n)} \;+\; \Delta t\, f_j(y_1^{(n)}, y_2^{(n)}, \ldots, y_m^{(n)}, t^{(n)}) \quad , \quad j = 1, 2, \ldots, m \tag{5.6}$$

is the same as in one dimension, except that y and f now have subscripts. Equation (5.6) actually requires m different calculations to update the m different unknowns y_j. Each unknown is updated using the *old* values of the solution; the time-step is complete only after all m unknowns have been updated.

Given the structure of the algorithm it is easy to see that initial conditions must be provided for each unknown. In other words we must specify all m quantities $y_j^{(1)}$ in order to begin the process.

The truncation error in m-dimensions, the difference between the exact answer and its approximation

$$\epsilon_j \;=\; \bar{y}_j(t^{(n+1)}) - y_j^{(n+1)} \quad ,$$

is an m-dimensional vector. It is easy to show that each component scales with the same power of Δt. This determines the order of the method, and it must be the same as that in one-dimension. Thus the Euler method is first order accurate. Analogously, the m-dimensional version of RK2,

$$\boxed{y_j^{*} = y_j^{(n)} + \tfrac{1}{2}\Delta t\, f_j(\,\mathbf{y}^{(n)},\, t^{(n)}\,)} \tag{5.7}$$

$$\boxed{y_j^{(n+1)} = y_j^{(n)} + \Delta t\, f_j(\,\mathbf{y}^{*},\, t^{(n)} + \tfrac{1}{2}\Delta t\,)} \tag{5.8}$$

is second order accurate.

Coding in general dimensions

We have chosen to make this general program `rk2` as general as possible — it solves ODEs with *any number of dimensions*. This means that m is assigned when the program is called and not before. But where, you might well ask, does the information about m come from? One option, often used in e.g. FORTRAN programs, is to ask the user directly — i.e. make m an input argument. The program `rk2` is more subtle than this; it figures out the value of m without asking directly. The `Matlab` function `length` is used to find the number of elements in the initial-condition vector `y0`. Unless the user has made a mistake, the length of `y0` must be equal to `m`.

The output from the function named by `deriv_func` must be an m-dimensional vector containing f_1, f_2,..., f_m. If `deriv_func` returns a vector longer or shorter than this there will surely be trouble. The output from `rk2`, after N_{steps} steps, is an $(N_{\text{steps}}+1) \times m$ array called `y`. When finished the array element `y(n,j)` contains $y_j^{(n)}$. The array is created by the convenient `Matlab` functions `zeros` which creates an array and fills it with zeroes. As in the single-dimensional case these initial values are unimportant since they will be changed as the solution is found.

..... Begin Program 13

file: `rk2.m`

```
function y = rk2( y0, t0, dt, n_steps, deriv_func )
%
%  take n_steps time steps, of size dt, in the differential equation
%    dy/dt = deriv_func( y, t )
%  begining with y(1) = y0 and time t = t0.
%  all quantities yand f are of dimension m.

m = length( y0 );           %  determine dimension from y0

y = zeros( n_steps+1, m );  %  create an empty array for results

y( 1, : ) = y0;  % initial condition

for n = 1:n_steps

% the first half-step
   t = t0 + dt*(n-1);
   f = feval( deriv_func, y(n,:), t );
   y_star = y(n,:) + 0.5*dt*f;

% the full step
   t_star = t + 0.5*dt;
   f_star = feval( deriv_func, y_star, t_star );
   y(n+1,:) = y(n,:) + dt*f_star;
end
```

..... End Program 13

In many (primitive) languages such as C or FORTRAN, each implementation of Eq. (5.7) or (5.8) would require a loop where j is incremented from 1 to m. Advanced languages like `Matlab` allow this to be done in a single statement such as

```
y_star = y(n,:) + 0.5*dt*f;
```

where whole vectors are added together. (This does not mean the computer performs the calculation any faster, just that you, the programmer, can type the statement more quickly.) Note the use of the `:` to pick out a single row of the array `y`.

An example — electric field lines

An electric field $\mathbf{E}(\mathbf{x})$ is often visualized by showing electric field lines. A field line is a curve in space described by the vector-valued function $\mathbf{r}(s)$ where s is the length along the curve. A "picture" of the electric field can then be produced by plotting $r_y(s)$ versus $r_x(s)$ over the full range of the parameter s. The space-curve $\mathbf{r}(s)$ is defined by the property that the electric field vector $\mathbf{E}$ is *tangent* to it at every point. The tangent to a space-curve is found from its first derivative, so a field line is actually defined by an ordinary differential equation

$$\text{tangent to } \mathbf{r}(s) \;=\; \frac{d\mathbf{r}}{ds} \;=\; \frac{\mathbf{E}(\mathbf{r})}{|\mathbf{E}(\mathbf{r})|} \tag{5.9}$$

The right hand expression is simply a unit vector pointing in the direction of the electric field at the point $\mathbf{r}(s)$.

Equation (5.9) has exactly the same form as our ODE prototype Eq. (5.1), with the independent variable being s, rather than t, the dependent variables being r_j, rather than y_j, and the vector-valued function being

$$f_j(\mathbf{r}, s) \;=\; \frac{E_j(\mathbf{r})}{\sqrt{E_x^2(\mathbf{r}) + E_y^2(\mathbf{r}) + E_z^2(\mathbf{r})}} \quad . \tag{5.10}$$

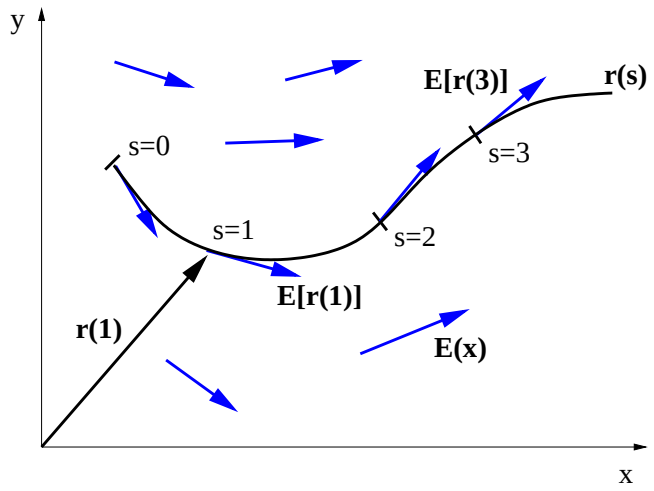

Figure 5.1: An electric field line $\mathbf{r}(s)$. The curve is tangent to the electric field vector $\mathbf{E}(\mathbf{x})$ at each point.

In order to solve for a field line we need only write a specific program to evaluate this vector-valued function for vector argument $\mathbf{r}$ and independent variable s.

Section 3.2 introduced a program `efld` (Program 8) to calculate the electric field vector from an arrangement of charges in the x–y plane. To find the electric field lines from this charge distribution we implement Eq. (5.10) using `efld`:

..... Begin Program 14

file: `tan_vec.m`

```
function f = tan_vec( r, s )
%
%  compute the tangent vector to the electric field at point r

e = efld( r );          % evaluate components of electric field
emag = sqrt( e(1)^2 + e(2)^2 );    % |E|

f = e/emag;             % the tangent vector
```

..... End Program 14

The field line equation (5.9) can now be integrated using the general program `rk2`. First it is necessary to supply an initial condition. There are an infinite number of field lines for any electric field; there is exactly one field line crossing every point in space. Clearly we cannot draw *every* field line, that would take forever and produce a picture which was completely black. Which field lines we see will depend on which points we use for initial conditions. To begin with let us choose a field line near charge a; say $\mathbf{r}(0) = (0.01, 0.0)$. (It would be a very bad idea to start the field line exactly *at* of of the charges, say $\mathbf{r}(0) = \mathbf{x}_a$, since the electric field there is infinite.)

The natural length scale for our equation is the distance between any two charges: ~ 0.5 meters. We will opt for a step size of $\Delta s = 0.01$, and take $N_{\text{steps}} = 100$ steps to trace a total distance of one meter along the field line. All of this is accomplished by the statement

```
>> r = rk2( [ 0.01, 0.0 ], 0, 0.01, 100, 'tan_vec' );
```

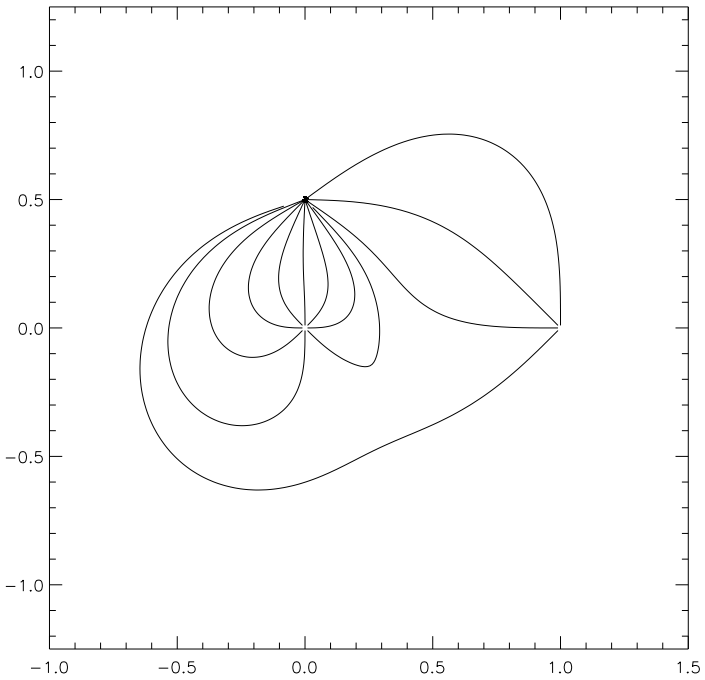

Figure 5.2: Electric field lines from the charge distribution shown in Figure 3.1. Field line equations are solved from initial points in the vicinity of charges a and b.

The result of the integration is an array `r(n,j)` containing both components of the curve $\mathbf{r}^{(n)}$. The x and y components of the curve are given by the first and second rows of the array

$$r_x = \texttt{r(:,1)} \quad , \quad r_y = \texttt{r(:,2)} \quad ,$$

where the colon is used to select the entire row. To show the field line we call `plot` with each of these vectors in its arguments:

```
>> plot( r(:,1), r(:,2) )
```

We can find and plot additional field lines by choosing different initial conditions. First we will instruct the plotter to keep all the old plots on the screen — i.e. to `hold` them:

```
>> hold
>> r = rk2( [ 0.01, 0.01 ], 0, 0.01, 100, 'tan_vec' );
>> plot( r(:,1), r(:,2) )
>> r = rk2( [ 0.01, 0.01 ], 0, 0.01, 100, 'tan_vec' );
>> plot( r(:,1), r(:,2) )
>> r = rk2( [ 0.0, 0.01 ], 0, 0.01, 100, 'tan_vec' );
>> plot( r(:,1), r(:,2) )
>> r = rk2( [ 1.0, 0.01 ], 0, 0.01, 100, 'tan_vec' );
>> plot( r(:,1), r(:,2) )
>> r = rk2( [ 0.99, 0.0 ], 0, 0.01, 100, 'tan_vec' );
>> plot( r(:,1), r(:,2) )
```

The result is a picture, Figure 5.2, of the electric field arising from the three charges in Figure 3.1. The starting points are clustered about charges a and b, and all lines lead ultimately to point c.

5.2 High order ODEs

ODEs which involve only first derivatives dy/dt are called *first order* ODEs.[1] So far we have considered only first order ODEs, the most general being the m-dimensional case (5.1). It turns out that ODEs of higher order can always be written as a larger set of first order ODEs using a clever trick. For this reason it is only necessary to write computer codes to solve first order ODEs. Happily `rk2` is the only ODE solver we'll ever need. Let us demonstrate this trick with an example: the pendulum.

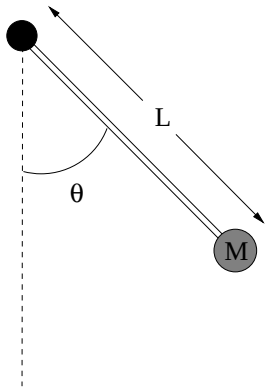

Figure 5.3: The pendulum with massless arm of length L.

Consider a pendulum whose (massless) arm, of length L, supports a mass, and is subject to a friction ν. The angle θ of the arm from vertical, satisfies the second-order ODE[2]

$$\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} \;=\; -\frac{g}{L}\sin(\theta) \quad . \tag{5.11}$$

This might appear more familiar if we made the small angle approximation, $\sin(\theta) \simeq \theta$, and dropped the friction, $\nu = 0$. There is no need to make such approximations in numerical Physics.

The single second-order ODE (5.11) can be written as a set of *two* first-order ODEs. The two unknowns, called $y_1(t)$ and $y_2(t)$, will be defined

$$y_1(t) \;=\; \theta \quad , \tag{5.12}$$

$$y_2(t) \;=\; \frac{d\theta}{dt} \quad . \tag{5.13}$$

All we did in (5.12) was to give a new name to our original unknown $\theta(t)$. The second definition, (5.13), can seem puzzling at first: how can $d\theta/dt$ be considered to be independent of θ? This is the element of trickery.

Using these new definitions it is easy to re-cast (5.11) as a first order ODE since the second derivative of θ is just a first derivative of the new variable y_2

$$\frac{dy_2}{dt} \;=\; -\nu\, y_2 - \frac{g}{L}\sin(y_1) \;=\; f_2(y_1, y_2) \quad . \tag{5.14}$$

[1] This use of "order" is too easily confused with its use as "order of accuracy" describing the truncation error. Because of this confusion we can say "RK2 is a second-order algorithm for first-order ODEs" — alas, there appears to be no alternative.

[2] For a derivation without damping see, for example, Halliday, Resnick, and Walker (9^{th} ed.) section 15-6.

Before we can solve our system, however, we need an equation for dy_1/dt. If we are careful with notation we can use both of our definitions, (5.12) and (5.13), in this role:

$$\frac{dy_1}{dt} \;=\; y_2 \;=\; f_1(y_1, y_2) \quad . \tag{5.15}$$

All we did was to write $d\theta/dt$ two different ways using our two definitions; the resulting function f_1 is not terribly impressive, but it will do the job nicely. Indeed, this is where we confess to our trick, by admitting that y_2 is not independent of y_1.

After successfully playing our trick it is easy to write a `Matlab` program to solve the damped pendulum equation. The program to cast this as a set of two first order ODEs is given below as Program 15, for specific choices of L and ν.

..... Begin Program 15

file: `pendulum.m`

```
function f = pendulum( y, t )
%
%  return the RHS of the equation for a damped pendulum
%    theta'' = - nu theta' - (g/L)*sin(theta)
%

g = 9.8;              %  acceleration of gravity (m/s^2)
L = 9.8;              %  length of pendulum (m)
nu = 0.1;             %  damping coefficient (s^{-1})

f1 = y(2);            %  d y_1 / dt = y_2
f2 = -nu*y(2) - (g/L)*sin( y(1) );

f = [ f1, f2 ];       % cast result as vector
```

..... End Program 15

This function can now be passed to the general program `rk2` to solve the pendulum problem. We must specify the initial conditions for the unknowns y_1 and y_2. Consider drawing the pendulum up to an initial angle $\theta_0 = 1.6$ radians which is about horizontal, and releasing it from rest. Since it is initially at rest

$$\dot\theta(0) \;=\; y_2(0) \;=\; 0 \quad ,$$

while the pendulum's initial position

$$\theta(0) \;=\; y_1(0) \;=\; 1.6 \quad ,$$

completes the initial vector. The natural time scale is $\sqrt{L/g} = 1$ s; we will take time steps of $\Delta t = 0.01$ s, until a time of 30 seconds.

```
>> y = rk2( [ 1.6, 0.0 ], 0, 0.01, 3000, 'pendulum' );
```

Be sure you know what each number in the argument list is for.

The ODE solver RK2 returns all of the points $y_1^{(n)}$ and $y_2^{(n)}$ in the array `y`. These are the position of the pendulum $\theta(t)$ as well as its angular velocity $\dot\theta(t)$. In order to graph the solution we first create a time array, `t` and then pass to `plot` a single column of the array `y`. Passing the first column, `y(:,1)`$= y_1^{(n)}$, shows the pendulum's angle as a function of time:

```
>> t = 0.01*(0:3000);
>> plot( t, y(:,1) )
```

The resulting plot shows the pendulum swinging down from its initial angle $\theta = 1.6$. It continues to swing back and forth, with its amplitude decaying; it never again reaches the horizontal $\theta = \pm 1.6$. Note that the time between peaks is roughly 6 seconds. Had we made the small angle approximation and neglected friction the peak-to-peak interval would be exactly $2\pi\sqrt{L/g} = 6.28$ seconds.

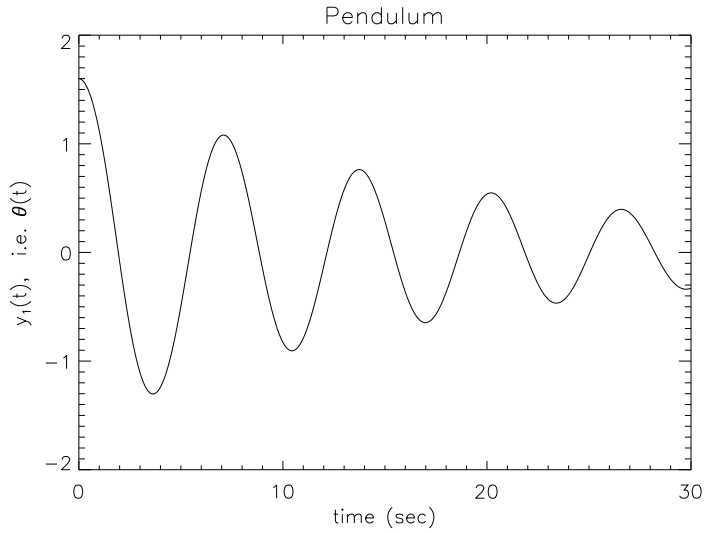

Figure 5.4: A graph of $y_1(t)$ vs. t from the call to rk2.

The solution to the pendulum equation can be graphed in many other interesting ways. Passing to `plot` the second column `y(:,2)`$= y_2^{(n)}$,

```
>> plot( t, y(:,2) )
```

will graph the angular velocity $\dot{\theta}$ versus time. This begins at zero, since the pendulum is at rest, then becomes negative as the pendulum swings downward toward vertical. Note that $\dot{\theta}(t)$ is out of phase with $\theta(t)$ by $\sim 90°$: when $\theta(t)$ is at a peak $\dot{\theta}(t)$ crosses zero. Due to damping the converse is not true: when $\dot{\theta}$ is at a peak $\theta(t)$ is *not* exactly crossing zero — although this is hard to see in the figure.

Finally, it is popular in classical mechanics to plot the "phase portrait" of the trajectory: $\dot{\theta}$ vs. θ. This is much easier to do

```
>> plot( y(:,1), y(:,2) )
```

than it is to interpret. Figure 5.5b shows the phase-portrait. The damping causes the curve to spiral in towards the origin $(\theta, \dot{\theta}) = (0, 0)$. Without damping the curve should return to its starting point and continue to retrace itself. Does this happen with the results of `rk2`?

5.3 Finding specific results — Driver programs

Until now we have written general-purpose programs, such as `rk2`, which returned to the user the entire solution curve for the ODE. In some circumstances we might be interested in a specific aspect

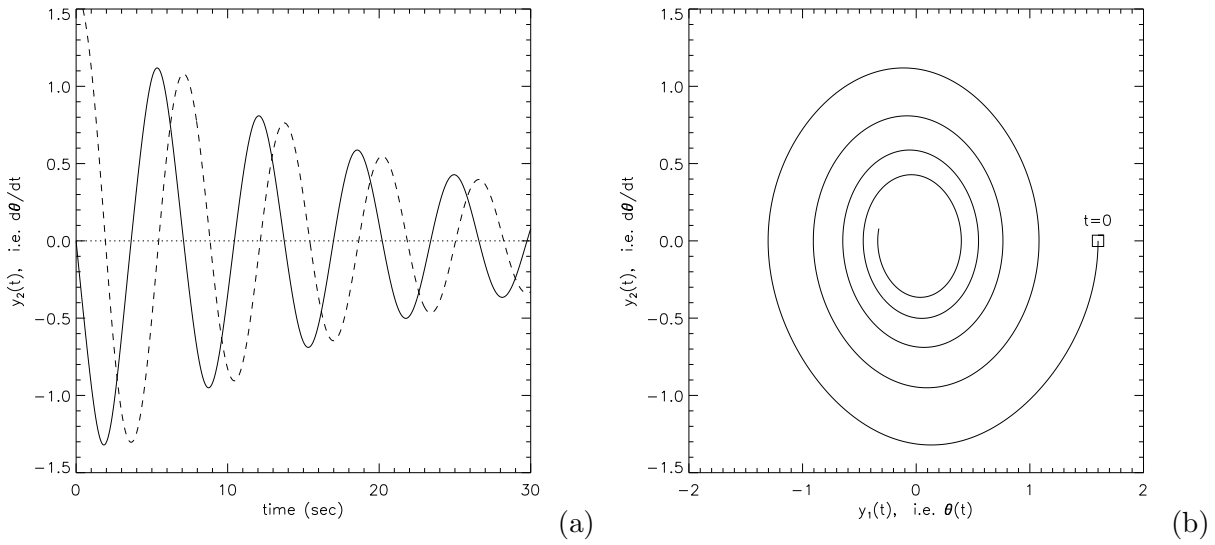

Figure 5.5: (a) A graph of $y_2(t) = \dot{\theta}(t)$ vs. t (solid). The previous figure, $y_1(t)$ vs. t, is reproduced as a dashed line for comparison. (b) The phase-plot $\dot{\theta}$ vs. θ.

of that solution, and not care about the entire curve. Let us use our work on aerodynamic drag to answer the following question: *How far can a baseball be hit?*

A baseball is subject to gravity and aerodynamic drag, so it obeys equations (5.3) and (5.4). According to section 1.09 of the *Official Baseball Rules*, a baseball must weigh between 5 and 5.25 ounces ($M = 0.142$ kg) and have a circumference between 9 and 9.25 inches ($r = 0.0364$ m). Extensive wind-tunnel tests show a baseball to have a drag coefficient[3] $C_D \simeq 0.5$. The bottom line is that the baseball is characterized by $\alpha = 0.0088\,\mathrm{m}^{-1}$, for air at sea-level ($\rho = 1.2\,\mathrm{kg/m}^3$).

Solving equation (5.3) and (5.4) for the velocity components v_x and v_y will not help us answer a question about distance. We need to solve for $x(t)$ and $y(t)$ as well. This requires a set of 4 ODEs for the 4-dimensional vector, called $\mathbf{w}$, which includes both components of both position and velocity:

$$\mathbf{w} = [\, x,\, y,\, v_x,\, v_y \,] \quad .$$

The ODE is the same one we always solve

$$\frac{dw_j}{dt} \;=\; f_j(\mathbf{w}, t) \qquad , \qquad j = 1,\, 2,\, 3,\, 4 \quad ,$$

where the function f_j is a 4-dimensional vector

$$\mathbf{f}(\mathbf{w}) \;=\; [\, w_3,\, w_4,\, -\alpha\sqrt{w_3^2 + w_4^2}\, w_3,\, -g - \alpha\sqrt{w_3^2 + w_4^2}\, w_4, \,] \quad . \tag{5.16}$$

The `Matlab` program to return this function is

..... Begin Program 16

file: `baseball.m`

[3]Unless, that is, the pitcher illegally roughens the ball's surface.

```
function f = baseball( w, t )
%
%  return the acceleration on a baseball
%  w(1) = x,  w(2) = y,  w(3) = v_x,  w(4) = v_y
%

g = 9.8;      %  acceleration of gravity (m/s^2)
rho = 1.2;    %  mass density of air (km/m^3)
a = 4.16e-3;  %  crosssectional area of baseball (m^2)
              %  baseball circumference = 9" = 0.2286 m
              %     ==> r = 0.0364 m
C_d = 0.5;    %  coefficient of drag
M = 0.142;    %  mass of baseball (kg) ( 5 oz. )

alpha = 0.5*rho*a*C_d/M;
v = sqrt( w(3)^2 + w(4)^2 );  %  |v|

f = [ w(3), w(4), -alpha*v*w(3), - g - alpha*v*w(4) ];
```

..... End Program 16

We will consider a baseball with initial speed $v_0 = 50$ m/s (~ 112 miles-per-hour — a reasonable speed for a ball coming off a major-league bat) at an angle $\theta = 45°$ from horizontal. Both components on the initial velocity are then $v_x(0) = v_y(0) = v_0/\sqrt{2} = 35.3$ m/s. We will begin at the origin $x = y = 0$, and take time steps $\Delta t = 0.01$ s.

```
>> w = rk2( [ 0, 0, 35.3, 35.3 ], 0.0, 0.01, 600, 'baseball' );
```

This solves for the evolution of the baseball for $600 \times \Delta t = 6$ seconds. To see the trajectory we graph y vs. x

```
>> plot( w(:,1), w(:,2) )
```

The ball arcs up to a maximum height of $y = 36$ meters (119 feet) before falling to the ground $y = 0\ldots$ and beyond! Neither our force equations (5.16), nor their `Matlab` implementation (Program 16) made any reference to the ground, therefore the solution continues into regions $y < 0$ without difficulty. To answer our original question about the distance we can try to read across the ground, $y = 0$ to where the curve $y(x)$ crosses: about $x = 100$ meters (328 feet).

The final distance will depend upon the initial condition $w_j(0)$; in particular upon v_0 and θ. In fact, it would be nice to write a program which returns a single number, the distance d, for given values of the speed and angle v_0 and θ. This kind of program is sometimes called a *driver*; it calls the program `rk2` in order to answer a specific question. The particular driver, called `long_ball`, is given below as Program 17

..... Begin Program 17

file: `long_ball.m`

```
function d = long_ball( v0, theta0 )
%
% return the distance travelled by a baseball with initial
% speed v0 and angle theta0 from horizontal (in degrees).

th_rad = theta0/57.2958;  % translate angle to radians
dt = 0.01;                % time step for integration

% the intial conditions
```

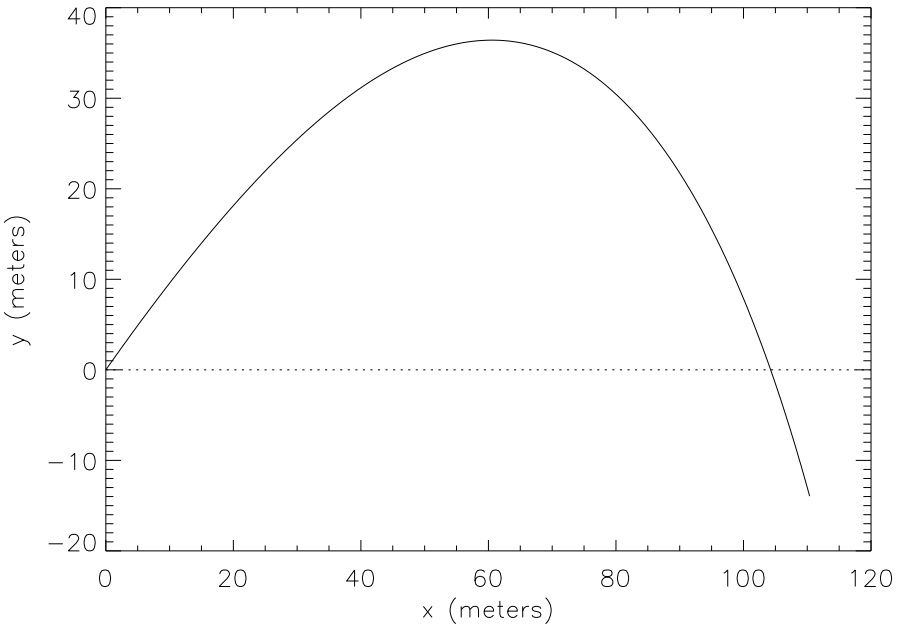

Figure 5.6: Trajectory of a baseball hit at $v_0 = 50$ m/s at an initial angle $\theta = 45°$.

```
w0 = [ 0, 0, v0*cos(th_rad), v0*sin(th_rad) ];
t = 0.0;

% define a "flag" to indicate when program is done
done = 0;   % program is not done... yet

% repeat until calculation is done
while done == 0
   w = rk2( w0, t, dt, 1, 'baseball' );   % take 1 step
   w0 = w(2,:);               %  replace old data with new data
   t = t + dt;                %  update time

   % check to see if calculation is done
   if w(2,2) < 0.0
      done = 1;  % ball has passed ground... end
   end

   if t > 30.0
      done = -1;     % safety precaution... end after 30 seconds
   end
end

if done == 1
% ball passed the ground in the last step
   x0 = w(1,1);  % point before ground...
   y0 = w(1,2);  % ... should be positive

   x1 = w(2,1);  % point after ground...
   y1 = w(2,2);  % ... should be negative

  % linear interpolation to the point of contact...
   frac = y0/(y0-y1);          %  the fraction of the last time step
   d = x0*frac + x1*(1-frac);  %  distance up to impact
else
```

```
   d = -1;       % indication that problem occurred
end
```

..... End Program 17

- The heart of the program is the loop beginning

  ```
  while done == 0
  ```

  which repeatedly calls the integration routine `rk2` until a completion criteria is met. `rk2` is called to take a single step Δt at a time.

- A "flag" called `done` is used to allow multiple completion criteria. `done` is initially set to 0 so the loop will always be executed *at least once*. Integration continues until `done` is changed; this can happen in one of two ways:

  1. The height of the ball, `w(2,2)`, is less than zero — the ball has crossed ground-level.
  2. The total integration has gone longer than 30 seconds. This would never happen under ordinary circumstances — baseballs just don't stay aloft that long. This is here as a precaution. If some kind of mistake is made (perhaps by the programmer), we don't want the computer to get "hung" waiting forever for $y < 0$. Later statements will cause the meaningless distance $d = -1$ to be returned if this criteria is met. This alerts us to a problem in the code. Always practice safe computing!

  `done` is set to a different value depending on which criteria is met, but the integration ends once `done`$\neq 0$.

- We use the final two points to determine the distance the ball is hit. Both points appear in `w`, they are: `w(1,:)`$= (x0, y0)$ the last point *above the ground* and `w(2,:)`$= (x1, y1)$, the first point *below the ground*. Linear interpolation between these two points is used to locate the exact point $(d, 0)$ where the ball hits the ground.

The driver program is simply called by setting its arguments to v_0 in m/s, and θ in degrees.

```
>> long_ball( 50, 45 )
```

returns `104.44` meters, close to our guess from Fig. 5.6.

A major advantage of the driver program is that it can be easily used to explore parameter space. Typing a few lines shows how the distance depends on angle.

```
>> long_ball( 50, 25 )
>> long_ball( 50, 30 )
>> long_ball( 50, 35 )
>> long_ball( 50, 40 )
>> long_ball( 50, 38 )
```

The resulting table clearly shows that the maximum distance occurs for an initial angle around $\theta = 38°$. This is somewhat flatter than the value of $\theta = 45°$ which always produces the maximum distance when air resistance is neglected. The maximum without air resistance $d = v_0^2/g = 255$ meters (837 feet) is considerably longer than any baseball has ever been hit. The distances in the table, up to 349 feet, are more typical for a good hit in the major league.

θ (degrees)	d (meters)	d (feet)
25	97.5	320
30	103.1	338
35	105.9	347
38	106.4	349
40	106.2	348
45	104.4	342

Exercise 5.1 Modify the driver program to answer the following questions.

1. How much farther will the $v_0 = 50$ m/s ball travel if it is hit at Coors Field in Denver? At the altitude of Denver, $z = 1610$ m (5,280 ft) the air is slightly thinner: $\rho = 1.0\,\mathrm{kg/m^3}$.
2. How much farther will the $v_0 = 50$ m/s ball travel if there is a $u = 5$ m/s (11 mph) wind blowing toward center field.

Chapter 6

Eigenvalue Problems

6.1 The general problem

Eigenvalue problems occur in many areas of Physics; they are especially prevalent in Quantum mechanics. Perhaps the most familiar example is a classical problem: the vibration of a stretched string. Consider a string stretched horizontally between $x = 0$ and $x = L$ under tension τ. The mass per unit length of the string $\mu(x)$ might vary over its length. A small vertical displacement can be described by the function $y(x)$, however, the displacement will naturally change over time: $y(x,t)$. The evolution of the displacement is described by the one-dimensional wave equation

$$\mu(x)\,\frac{\partial^2 y}{\partial t^2} \;=\; \tau\,\frac{\partial^2 y}{\partial x^2} \quad . \tag{6.1}$$

The string is stretched between fixed ends (i.e. the string cannot be displaced there), so the solution $y(x,t)$ is subject to the boundary conditions

$$y(x{=}0) = y(x{=}L) = 0 \quad , \tag{6.2}$$

at all times t.

For concreteness lets consider two different cases both involving $L = 1$ meter strings under $\tau = 1$ Newton of tension. In case I the string is uniform with mass density $\mu = 0.01$ kg/m. In case II the string is tapered (like a fishing line)

$$\text{case II:} \qquad \mu(x) \;=\; 0.001 \;+\; 0.018\,x \text{ kg/m} \quad , \tag{6.3}$$

so it is lighter on the left and heavier on the right, than the string in case I. Both strings have the same total mass.

We will seek solutions of equation (6.1) which oscillate in time with some frequency ω. In other words we make the assumption that the solution $y(x,t)$ can be "separated" by writing it in the form

$$y(x,t) \;=\; \phi(x)\,\sin(\omega t) \quad , \tag{6.4}$$

where we must solve for the unknown function $\phi(x)$ *and* the unknown frequency ω; this is an eigenvalue problem. Substituting (6.4) into (6.1) we get a second order ODE for the unknown function $\phi(x)$

$$c^2(x)\,\frac{d^2\phi}{dx^2} \;=\; -\,\omega^2\phi \quad , \tag{6.5}$$

where $c(x) = \sqrt{\tau/\mu(x)}$ is the wave speed at position x. For the two cases we have wave speed functions

$$\begin{aligned} \text{case I:} \qquad c(x) &= 10\,\text{m/sec} \quad , \\ \text{case II:} \qquad c(x) &= \frac{1}{\sqrt{0.001 + 0.018x}}\,\text{m/sec} \quad . \end{aligned}$$

The boundary conditions (6.2) imply that

$$\phi(0) = \phi(L) = 0 \quad . \tag{6.6}$$

The previous section described how to solve any type of ODE, so we might think it will work with the second-order ODE equation (6.5). The first step is to rewrite it as two first order ODEs for the two unknowns $w_1(x)$ and $w_2(x)$

$$w_1(x) = \phi(x) \quad , \tag{6.7}$$
$$w_2(x) = \phi'(x) \quad . \tag{6.8}$$

In terms of these unknowns, equation (6.5) can be written

$$\frac{dw_1}{dx} = w_2(x) = f_1(w_1, w_2, x) \quad , \tag{6.9}$$
$$\frac{dw_2}{dx} = -\frac{\omega^2}{c^2(x)}\, w_1 = f_2(w_1, w_2, x) \quad . \tag{6.10}$$

There are two obstacles to using an ODE solver, such as RK2, on equations (6.9) and (6.10).

1. Initial conditions: In order to use RK2 we need initial conditions $w_1(0)$ and $w_2(0)$. Equation (6.6) gives one initial condition $w_1(0) = 0$, but the second, $w_2(0)$, is not specified. Even worse, the second boundary condition in (6.6), $w_1(L) = 0$, is unlike anything we've encountered so far. It is hard to imagine how we could satisfy such a condition: once RK2 gets stared from $x = 0$ it won't ask for anything more — it will *tell us* what $w_1(L)$ is, and it's a good bet that won't be zero!

2. ω: The second problem with Eq. (6.10) is that the function $f_2(w_1, w_2, x)$, which we want to give RK2, contains a number we don't know: ω. In fact, ω is one of the things we must solve for. How can we integrate the ODE when it contains a number we don't know? How can we find ω until we can solve the ODE?

6.2 Analytical solution: Case I

If you have seen the eigenvalue equation (6.5) before it was almost certainly for case I, where c does not depend on x,

$$\frac{d^2\phi}{dx^2} = -\frac{\omega^2}{c^2}\,\phi \quad .$$

In this case the standard approach is to write down the most general analytic solution which satisfies the initial condition $\phi(0) = 0$

$$\phi(x) = A\,\sin(\omega x/c) \quad . \tag{6.11}$$

In this analytic solution the values of A and ω are as yet unknown. To satisfy the second boundary condition

$$\phi(L) = A\sin(\omega L/c) = 0 \quad , \tag{6.12}$$

we must either choose $A = 0$ *or* choose $A \neq 0$ and ω so that $\sin(\omega L/c) = 0$. In the first option, $A = 0$, the solution (6.11) is really just $\phi(x) = 0$. While this certainly does satisfy the equation (6.5) it is a pretty disappointing solution, often called *the trivial solution*. The alternative, the *non-trivial solution* ($A \neq 0$), leads to the condition

$$\omega \;=\; \frac{\pi n c}{L} \quad , \quad n \;=\; 1,\, 2,\, \ldots \tag{6.13}$$

since $\sin(\pi n) = 0$ for any integer n.

Note that in the non-trivial solution A is free to be anything but zero. This means that the solution to the eigenvalue equation (6.5) is *not unique*. Given one solution $\phi = \phi_1(x)$ it is always possible to construct a second solution $\phi_2(x) = 2\phi_1(x)$. Since both sides of (6.5) will simply be doubled if $\phi(x)$ is doubled, the equation is still satisfied. This is true of all eigenvalue problems.

Since A is completely arbitrary the second initial condition

$$w_2(0) \;=\; \phi'(0) \;=\; A\omega/c \;\;, \tag{6.14}$$

is also arbitrary. This can seem quite surprising at first: $w_2(0)$ *can be given any value except zero.*

Finally, ω can be any one of a set of distinct values — these are the eigenvalues or eigenfreqencies. For the string in case I we find eigenfrequencies

$$\omega_1 \;=\; 31.4159\,\mathrm{Hz} \quad , \quad \omega_2 \;=\; 62.8319\,\mathrm{Hz} \quad , \quad \omega_3 \;=\; 94.2478\,\mathrm{Hz} \quad , \quad \ldots$$

These are the specific frequencies at which the stretched string can vibrate. For each of these frequencies there is a different function, $\phi(x)$, called the eigenmode.

We would like to develop a numerical method to solve the eigenvalue equation (6.5), for both case I and case II. In particular, will the tapered string (case II) vibrate at different frequencies than the uniform string?

6.3 Numerical solution — The Shooting Method

The analytic solution above has given some insight into overcoming the obstacles we previously faced. First we see that our second initial condition, $w_2(0)$, can be set to any number but zero — say one. With this choice the complete set of initial conditions for the ODE are

$$w_1(0) \;=\; 0 \quad , \quad w_2(0) \;=\; 1 \;\;. \tag{6.15}$$

With initial conditions we would seem almost ready to use RK2 to solve Eqs. (6.9) and (6.10). The remaining obstacle is a single unknown ω. Luckily there is also a boundary condition, $w_1(L) = 0$, which we have not satisfied.

Our approach, often called *The Shooting Method*, will be to treat the problem as a single equation for a single unknown. Let us denote this equation

$$g(\omega) \;\equiv\; w_1(L) \;=\; 0 \;\;, \tag{6.16}$$

For a specific value $\omega = \omega_0$, the ODEs, (6.9) and (6.10), will be solved to yield $w_1(x)$. We can think of this as taking a "shot" at the target $x = L$, which will almost certainly miss the mark: $w_1(L) \neq 0$ (see Fig. 6.1). A second "shot" is then taken using a second value $\omega = \omega_1$, giving a solution $w_1(x)$ also (probably) missing the mark. If these two shots bracket the target, then `bisect` can be used to zoom in on the correct frequency $\omega = \omega_*$ where $g(\omega_*) = 0$.

In order to solve Eq. (6.16) using `bisect` we must write a program to evaluate $g(\omega)$. The first lines of this program

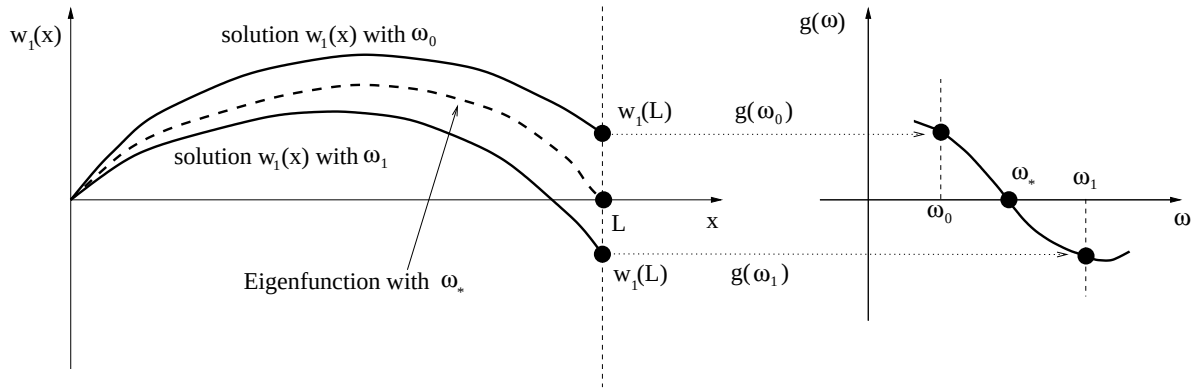

Figure 6.1: The shooting method in action. First and second guesses ω_0 and ω_1 are used to solve the ODE for $w_1(x)$. These give values of the function $g(\omega)$ bracketing the desired solution $g(\omega_*) = 0$.

```
function g = BC_caseI( omega )
%
%  return the value of w_1(x) at x = L = 1 meter
%
```

indicate that it takes the unknown `omega` and returns $g(\omega) = w_1(L)$. In the past we have used `bisect` to solve equations we could write down explicitly, such as $f(x) = e^x - 3x^2$. Here we face a slightly trickier situation: our function $g(\omega)$ depends on the solution to an ODE. Fortunately, this is exactly the problem `rk2` was developed to solve.

To call `rk2` we need to write a program, let us call it `caseI`, to evaluate the functions f_1 and f_2 once supplied with their input arguments w_1, w_2, x. The overall plan is summarized schematically in Figure 6.2. Each program is represented by a box with its inputs and outputs on its left. The

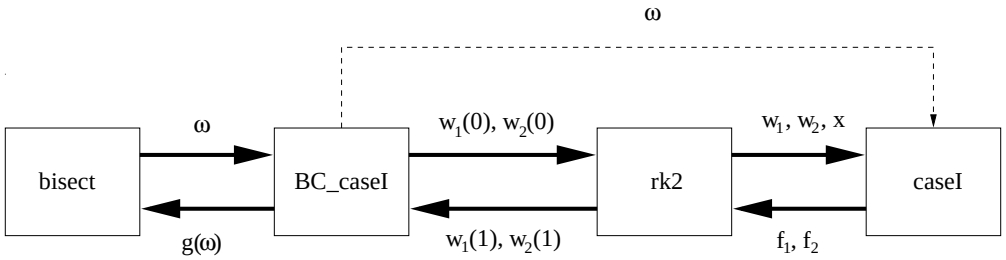

Figure 6.2: Schematic of the shooting method algorithm to solve eigenvalue problem. Inputs enter and outputs leave from the left of a box. The problem of communicating ω is shown with a dashed line.

schematic figure points to one final small obstacle. The function f_2 take arguments w_1, w_2 and x, but also needs to know the value of ω. Unfortunately `rk2` will not accept ω as an argument, so it cannot pass it to `caseI`. We need `BC_caseI` to transmit ω to `caseI` without the help of `rk2` — to somehow "tunnel" it across. This kind of problem occurs frequently and most languages contain some programming mechanism, called *external variables* or *common blocks*, to accomplish it.[1]

Rather than introduce a new programming idea at this point, however, we will use a simple trick to overcome this obstacle. We will fool `rk2` into thinking that ω is an input argument to f_2.

[1]These mechanisms are a way around the very strict input/output information control discussed in Chapter 1. They are often dismissed by purists who consider them a needless breech of security, and an open door for disasters. On the other hand, if used prudently they can eliminate some major headaches.

We will introduce a *third* unknown function $w_3(x)$ which will always be equal to ω and will not change. Since w_3 will not change its ODE is

$$\frac{dw_3}{dx} \;=\; 0 \;=\; f_3(w_1, w_2, w_3, x) \quad . \tag{6.17}$$

`BC_caseI` will set $w_3(0) = \omega$ initially, and $w_3(x) = \omega$ thereafter. The last program in the chain, `caseI`, can then use the input argument w_3 in place of ω. The other functions are modified version of (6.9) and (6.10)

$$f_1(w_1, w_2, w_3, x) \;=\; w_2 \quad , \tag{6.18}$$

$$f_2(w_1, w_2, w_3, x) \;=\; -\frac{w_3^2}{c^2(x)}\, w_1 \quad . \tag{6.19}$$

This plan is summarized in the modified schematic

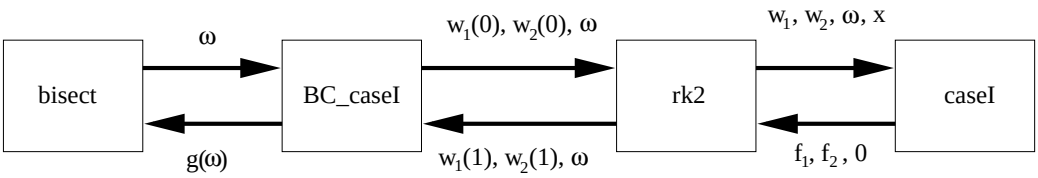

Figure 6.3: Schematic of the *new* plan to solve eigenvalue problem.

All that remains is to write the programs `caseI` and `BC_caseI` along the lines just described. Program 18 below implements equations (6.18), (6.19) and (6.17) for use by `rk2`

..... Begin Program 18

file: `caseI.m`

```
function f = caseI( w, x )
%
% evaluate the functions describing the eignenmode of case I,
% a uniform string.  argument w(3) is actually the eigenfrequency
%

c_sq = 100.0; %  the square of the wave speed, c^2
f1 = w(2);
f2 = - w(1)*w(3)^2/c_sq;
f3 = 0;       %  never change the eigenvalue --- trick!

f = [ f1, f2, f3 ];
```

..... End Program 18

..... Begin Program 19

file: `BC_caseI.m`

```
function g = BC_caseI( omega )
%
%  return the value of w_1(x) at x = L = 1 meter.
%
```

```
L = 1;   % length of domain

n_steps = 1000;  % number of steps to be used
dx = L/n_steps;

% set initial conditions:  w_1(0) = phi(0) = 0
%                          w_2(0) = phi'(0) = 1
%                          w_3(0) = omega
w0 = [ 0.0, 1.0, omega ];

w = rk2( w0, 0.0, dx, n_steps, 'caseI' );  % solve ODE

g = w(n_steps+1,1);  %  w_1(L) = g
```

..... End Program 19

We are now in the familiar position of having to solve an equation $g(\omega) = 0$ for ω. Although the function is defined in a complicated way, we have written a program `BC_caseI` to evaluate it. Calling the function with some frequency ω

```
>> g = BC_caseI( 20.0 )
```

returns the value `g = 0.4546`. This is not zero, so $\omega = 20$ is not an eigenfrequency. Still, this gives us some confidence that our program is working.

Referring to our standard protocol our next step is to graph $g(\omega)$. First we create an array of frequencies `omega` ranging from 0 to 100 for instance. Next we create an array of zeros to hold the results $g(\omega)$, and fill it by typing a loop. Finally we plot the result. All of this is accomplished by typing the commands

```
>> omega = 5*(0:20);
>> g = zeros( 21, 1 );
>> for i = 1:21
   g(i) = BC_caseI( omega(i) );
   end
>> plot( omega, g )
```

The first thing we notice is how much longer it takes to fill the array $g(\omega)$ than it ever did to fill $f(x)$ arrays when we last plotted functions. Obviously the difference here is that each call to `BC_caseI` runs an ODE solver.

The plot (see Fig. 6.4) shows that $g(\omega)$ oscillates about zero, crossing $g = 0$ three times within our range $0 \leq \omega \leq 100$. Thus we have an equation with multiple solutions. This should come as no surprise; each solution is an eigenvalue, and the nature of $g(\omega)$ shows that there are several different eigenvalues. In fact we know from the analytic solution that there are an infinite number of eigenvalues, so $g(\omega)$ must continue to oscillate indefinitely, crossing zero over and over again. Our graph shows three zero crossings, labeled A, B and C. We next find the exact value of each one by calling `bisect`

```
>> omA = bisect( 'BC_caseI', 30, 35 )
>> omB = bisect( 'BC_caseI', 60, 65 )
>> omC = bisect( 'BC_caseI', 90, 95 )
```

The results, $\omega_A = 31.4159$, $\omega_B = 62.8317$ and $\omega_C = 94.2465$ agree closely with the analytic solutions, $\omega = 10\pi n$.

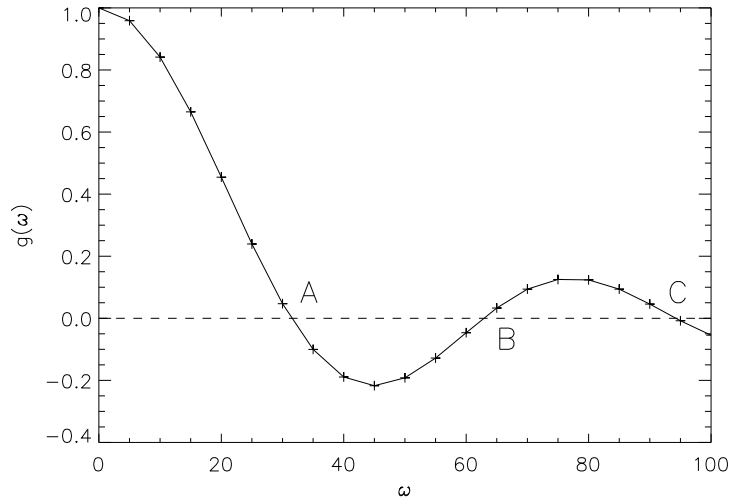

Figure 6.4: A graphs of the function $g(\omega)$ for case I (the uniform string). The 21 values of the array g are shown with pluses.

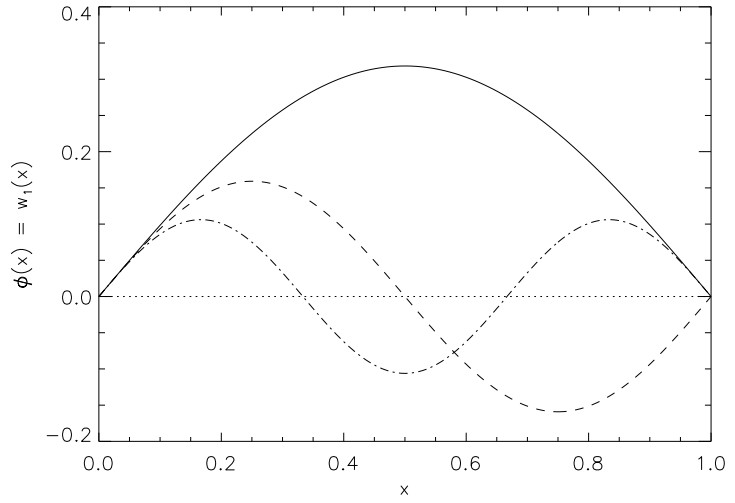

Figure 6.5: The first three eigenfunctions for a uniform string (case I). These correspond to the eigenvalues $\omega_A = 31.4159$ (solid) $\omega_B = 62.8317$ (dashed) and $\omega_C = 94.2465$ (dash-dot).

The eigenfunction $\phi(x) = w_1(x)$ corresponding to an eigenvalue ω is found by calling `rk2`. For instance

```
>> w = rk2( [ 0, 1, omA ], 0, 0.001, 1000, 'caseI' );
>> x = 0.001*(0:1000);
>> plot( x, w( :, 1 ) )
```

shows the eigenfunction $\phi(x)$ for ω_A. It is fairly evident that these eigenfunctions are sine curves, as anticipated by the analytic solution. Note that all three are tangent at $x = 0$ due to our choice of initial conditions.

6.4 Harder problems: case II

The last section developed a numerical method for eigenvalue problems, only to confirm our analytical solution. The method does provide some insight into the nature of eigenvalue problems. More importantly it can be applied to problems for which no analytic solutions exist. Consider the case of a tapered string, i.e. case II. At what frequencies does it vibrate? Are the frequencies evenly spaced (harmonics)? Are they higher or lower than those of the uniform string?

To answer these questions we need to write new versions of our programs, called `caseII` and `BC_caseII`. The first is `caseI` with a single line changed (marked with `****`)

..... Begin Program 20

file: `caseII.m`

```
function f = caseII( w, x )
%
% evaluate the functions describing the eignenmode of case I,
% a uniform string.  argument w(3) is actually the eigenfrequency
%

c_sq = 1/( 0.001 + 0.018*x ); %  **** the only line changed ****
f1 = w(2);
f2 = - w(1)*w(3)^2/c_sq;
f3 = 0;       %  never change the eigenvalue --- trick!

f = [ f1, f2, f3 ];
```

..... End Program 20

The second involves inserting a single character

..... Begin Program 21

file: `BC_caseII.m`

```
function g = BC_caseII( omega )
%
%  return the value of w_1(x) at x = L = 1 meter.
%

L = 1;   % lenght of domain

n_steps = 1000;  % number of steps to be used
dx = L/n_steps;
```

```
% set initial condtions:  w_1(0) = phi(0) = 0
%                         w_2(0) = phi'(0) = 1
%                         w_3(0) = omega
w0 = [ 0.0, 1.0, omega ];

w = rk2( w0, 0.0, dx, n_steps, 'caseII' );  % ***** the only change *****

g = w(n_steps+1,1);  %  return final value w_1(L)
```

..... End Program 21

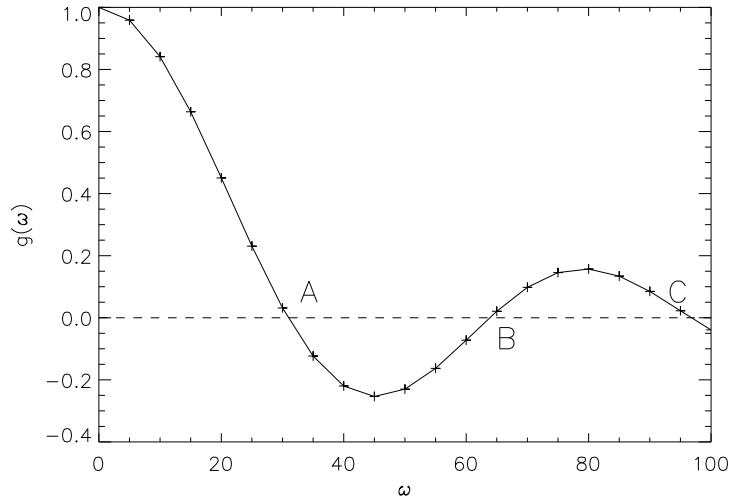

Figure 6.6: A graphs of the function $g(\omega)$ for case II (the tapered string). The 21 values of the array g are shown with pluses.

Repeating the same steps to graph $g(\omega)$ for the new case we see that it is only slightly different in case II than in case I. Using `bisect` we find the new eigenfrequencies

$$\omega_A \;=\; 30.8989 \quad , \quad \omega_B \;=\; 63.8321 \quad , \quad \omega_C \;=\; 96.7587 \quad .$$

The lowest eigenfrequency, ω_A is lower than its counterpart on the uniform string; the other two are higher. Clearly they are no longer uniformly spaced.

The eigenfunctions for case II are not unlike those from case I. The tapering, however, has led to a noticeable lack of symmetry: the functions seem "scrunched" to the right (where the wave speed is slowest).

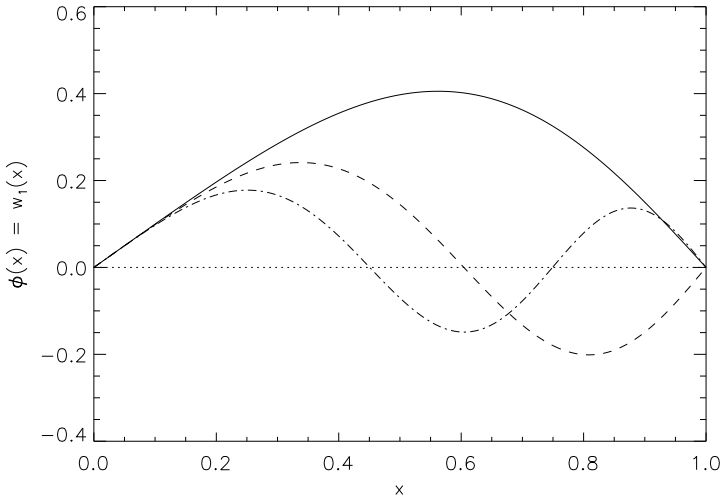

Figure 6.7: The first three eigenfunctions for a tapered string (case II). These correspond to the eigenvalues $\omega_A = 30.8989$ (solid) $\omega_B = 63.8321$ (dashed) and $\omega_C = 96.7587$ (dash-dot).

Chapter 7

Least-Squares Fitting of Data

Usually in physics, the goal of fitting data is to determine physical properties of the system we're measuring. It is important, then, that we develop a theoretical model that accurately reflects the physical system. Half the game, then, is arriving at that model. This chapter describes the other half: a technique for extracting the constants that appear in the model that best represent the data.

7.1 The General Problem

Let's say we have a set of data that generally follows a straight line with some scatter. What are the slope and intercept of the straight line? If we were in an introductory physics class, we might plot the data on graph paper, and then lay down a ruler and do a best fit to all the data by visual inspection. We're doing computational physics, however, so we want an algorithm for determining the slope and intercept that allows us to quantify those values – subject to certain assumptions – and to estimate the uncertainties in those values.

Let's say we're trying to fit N data points, organized according to a list of (x_i, y_i, σ_i), where x and y are the independent and dependent variables, respectively, σ is the uncertainty in the dependent variable, and i is an index on the data points that runs from $i = 1$ to $i = N$. We assume that there is no uncertainty in the independent variable, which is a common situation. It's sometimes the case that all of the σ_i are equal, but we'll develop a general method that allows them to be different.

Even if x and y don't have a linear relationship, we can often transform x and y into new variables that do have a linear relationship. For example, if x is the measured length of a pendulum and y is the measured period, then y and $\sqrt{x}$ should be linearly related. If we can't do such a transformation, however, then we will need a more general fit function. First, we'll look at a standard method for fitting data in the form of (x_i, y_i, σ_i) to any function, and then we'll specialize it to a straight-line function.

7.2 Method of Maximum Likelihood

Assume that we have in mind a function $f(x)$ that we'll use to fit the data. Usually, the choice of $f(x)$ is determined by the physics of the system, or by a visual inspection of the data. This function has some constants in it that we can choose so that we get the best fit. If $f(x)$ is a straight line, then the constants are the slope and intercept. If $f(x)$ is a quadratic, then there are three constants: $a + bx + cx^2$, for example. If $f(x)$ is a sine function, then there might be three constants,

too: $A\sin(2\pi x/\lambda + \phi)$ with the amplitude A, the wavelength λ, and the phase ϕ. As the functions become more complicated, the number of constants can increase, too. There are several different methods for systematically choosing these constants so that we find the best choice for all of them.

The *method of maximum likelihood* begins by making the assumption that the observed set of measurements is more likely to have come from a distribution that follows $f(x)$ with a specific set of constants than from $f(x)$ with any other set of constants[1]. To find the best values of the constants in this method, we begin by defining the dimensionless parameter χ^2 (chi-squared, with the Greek letter pronounced "kye"):

$$\chi^2 \equiv \sum_{i=1}^{N} \left[\frac{y_i - f(x_i)}{\sigma_i}\right]^2 \ ,$$

which is proportional to the weighted average of the squared differences between $f(x)$ and y. The weighting $1/\sigma_i^2$ is chosen so that data points with small uncertainties have more weight in determining that average, than do data points with large uncertainties. If we have a poor fit to the data, the differences $y_i - f(x_i)$ will be large compared to the uncertainty σ_i, so the sum will be larger than N. If we have a good fit to the data, the differences will be approximately the same size as the uncertainty on average, so the sum will be approximately equal to N. Squaring the differences before summing them avoids the situation in which positive and negative differences would cancel out, which would give an artificially small measure of the average deviation between the data and the fitting function.

Because χ^2 characterizes the average discrepancy between the data and our fit function, we want it to be as small as possible, so we minimize it[2]. Because we minimize the sum of the squared differences, the Method of Maximum Likelihood is also commonly referred to as *Least-Squares Fitting*. If $f(x)$ had only one constant in it, we would find the minimum value of χ^2 by differentiating χ^2 with respect to that constant, setting the result equal to zero, and solving the resulting algebraic expression[3]. Usually, however, $f(x)$ has more than one constant in it. If $f(x)$ has k such constants, we need to differentiate χ^2 with respect to each constant, set each of those resulting expressions equal to zero, and solve the resulting k equations for the k unknown constants.

Now, let's use the method with a specific fit function.

7.3 Least-Squares Fitting to a Straight Line

Let the straight line be expressed as $f(x) = a + bx$, where the slope b and the intercept a are the constants we're trying to find using least-squares fitting to the data. For this choice of $f(x)$,

$$\chi^2 = \sum_{i=1}^{N} \frac{1}{\sigma_i^2}\left[y_i - f(x_i)\right]^2 = \sum_{i=1}^{N} \frac{1}{\sigma_i^2}\left[y_i - a - bx_i\right]^2$$

[1]The likelihood is maximized in the same way that we can find the most likely exam score when we are given the scores of everybody in the class: we find the peak of the distribution. If the class is large enough, and the exam is a good one, then the scores will follow a bell curve or Gaussian distribution, with the peak at the exam average. We'll explore the Gaussian distribution more in Chapter 8.

[2]χ^2 is what appears in an exponential in the Gaussian distribution $\propto e^{-\chi^2/2}$. For the minimum possible value $\chi^2 = 0$, the exponential is its maximum, and hence we have achieved *maximum likelihood*.

[3]If the constants are simple coefficients in a series expansion, like in a straight line $f(x) = a + bx$, then this procedure will always find a minimum. If the constants appear in a more complicated way in the function, like the wavelength or the phase in the sine function example, then we'd need to determine whether we found a minimum or a maximum.

We want to find the minimum value of χ^2. This occurs when the derivatives of χ^2 with respect to a and with respect to b are both zero.

$$\begin{aligned} 0 &= \frac{\partial\chi^2}{\partial a} = -2\sum_{i=1}^{N}\frac{1}{\sigma_i^2}\left[y_i - a - bx_i\right] = -2\sum_{i=1}^{N}\frac{y_i}{\sigma_i^2} + 2a\sum_{i=1}^{N}\frac{1}{\sigma_i^2} + 2b\sum_{i=1}^{N}\frac{x_i}{\sigma_i^2} \\ 0 &= \frac{\partial\chi^2}{\partial b} = -2\sum_{i=1}^{N}\frac{x_i}{\sigma_i^2}\left[y_i - a - bx_i\right] = -2\sum_{i=1}^{N}\frac{x_iy_i}{\sigma_i^2} + 2a\sum_{i=1}^{N}\frac{x_i}{\sigma_i^2} + 2b\sum_{i=1}^{N}\frac{x_i^2}{\sigma_i^2} \end{aligned}$$

Now, canceling the overall factors of 2:

$$0 = -\sum_{i=1}^{N}\frac{y_i}{\sigma_i^2} + a\sum_{i=1}^{N}\frac{1}{\sigma_i^2} + b\sum_{i=1}^{N}\frac{x_i}{\sigma_i^2} = -\Sigma_y + a\,\Sigma_w + b\,\Sigma_x \tag{7.1}$$

$$0 = -\sum_{i=1}^{N}\frac{x_iy_i}{\sigma_i^2} + a\sum_{i=1}^{N}\frac{x_i}{\sigma_i^2} + b\sum_{i=1}^{N}\frac{x_i^2}{\sigma_i^2} = -\Sigma_{xy} + a\,\Sigma_x + b\,\Sigma_{x^2} \tag{7.2}$$

In these equations, we defined several variables to minimize clutter: $\Sigma_y \equiv \sum_{i=1}^{N} y_i/\sigma_i^2$, $\Sigma_w \equiv \sum_{i=1}^{N} 1/\sigma_i^2$, etc. These variables are simply numbers that we accumulate from the data set. Then we have two linear equations and the two unknowns: a and b.

Systems of linear equations are conveniently solved using matrix inversion. We can rewrite Eq. (7.1) and (7.2) in terms of one matrix equation:

$$\begin{pmatrix} \Sigma_w & \Sigma_x \\ \Sigma_x & \Sigma_{x^2} \end{pmatrix}\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \Sigma_y \\ \Sigma_{xy} \end{pmatrix} \tag{7.3}$$

Then we find a and b by inverting the 2×2 matrix, which has a determinant of $\Sigma_w\Sigma_{x^2} - \Sigma_x^2$, and multiplying the inverse by the column matrix on the right of the equals sign.

$$\begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{\Sigma_w\Sigma_{x^2} - \Sigma_x^2}\begin{pmatrix} \Sigma_{x^2} & -\Sigma_x \\ -\Sigma_x & \Sigma_w \end{pmatrix}\begin{pmatrix} \Sigma_y \\ \Sigma_{xy} \end{pmatrix}$$

Completing the matrix multiplication, we have

$$a = \frac{\Sigma_y\Sigma_{x^2} - \Sigma_{xy}\Sigma_x}{\Sigma_w\Sigma_{x^2} - \Sigma_x^2} \tag{7.4}$$

$$b = \frac{\Sigma_{xy}\Sigma_w - \Sigma_y\Sigma_x}{\Sigma_w\Sigma_{x^2} - \Sigma_x^2} \tag{7.5}$$

Again, the Σs are taken straight from the data, so we can easily calculate the values of a and b that minimize χ^2, and thereby give a good straight-line fit to the data.

Uncertainties

As in all data analysis, the values of a and b are only meaningful if we can estimate their uncertainties. Because a and b depend on all of the x_i, y_i and σ_i, we need to incorporate all of the uncertainties σ_i into an estimate of the uncertainties in a and b.

One of the most widely used methods to do this is combining the errors *in quadrature*: the square of the uncertainty in a calculated quantity is estimated as the weighted average of the squares of the uncertainty in the measured quantities, with the weighting determined by the chain

rule for differentiation. If we have a function $g(u, v)$ that depends on two independent variables u and v, we find the square of the uncertainty in g

$$\sigma_g^2 = \left(\frac{\partial g}{\partial u}\right)^2 \sigma_u^2 + \left(\frac{\partial g}{\partial v}\right)^2 \sigma_v^2.$$

Now, a and b each depend on $3N$ quantities — each of the x_i, y_i, and σ_i — but only the N values of y_i have uncertainties.

$$\sigma_a^2 = \left(\frac{\partial a}{\partial y_1}\right)^2 \sigma_1^2 + \left(\frac{\partial a}{\partial y_2}\right)^2 \sigma_2^2 + \left(\frac{\partial a}{\partial y_3}\right)^2 \sigma_3^2 + \ldots + \left(\frac{\partial a}{\partial y_N}\right)^2 \sigma_N^2 = \sum_{j=1}^{N} \left(\frac{\partial a}{\partial y_j}\right)^2 \sigma_j^2$$

The summation index j is helpful here to keep this sum separate from the sums that appears in calculating a, for which we had used index i. Taking the N derivatives of Eq. 7.4, recalling that a derivative with respect to y_j effects only y_i variables with $j = i$, and doing a bit of algebra:

$$\begin{aligned}
\sigma_a^2 &= \sum_{j=1}^{N} \left(\frac{\frac{1}{\sigma_j^2}\Sigma_{x^2} - \frac{x_j}{\sigma_j^2}\Sigma_x}{\Sigma_w \Sigma_{x^2} - (\Sigma_x)^2} \right)^2 \sigma_j^2 \\
&= \frac{1}{\left[\Sigma_w \Sigma_{x^2} - (\Sigma_x)^2\right]^2} \sum_{j=1}^{N} \left[\frac{1}{\sigma_j^2} (\Sigma_{x^2})^2 - 2\frac{x_j}{\sigma_j^2}\Sigma_x \Sigma_{x^2} + \frac{x_j^2}{\sigma_j^2} (\Sigma_x)^2 \right] \\
&= \frac{1}{\left[\Sigma_w \Sigma_{x^2} - (\Sigma_x)^2\right]^2} \left[(\Sigma_{x^2})^2 \sum_{j=1}^{N} \frac{1}{\sigma_j^2} - 2\Sigma_x \Sigma_{x^2} \sum_{j=1}^{N} \frac{x_j}{\sigma_j^2} + (\Sigma_x)^2 \sum_{j=1}^{N} \frac{x_j^2}{\sigma_j^2} \right] \\
&= \frac{(\Sigma_{x^2})^2 \Sigma_w - 2(\Sigma_x)^2 \Sigma_{x^2} + (\Sigma_x)^2 \Sigma_{x^2}}{\left[\Sigma_w \Sigma_{x^2} - (\Sigma_x)^2\right]^2} = \frac{(\Sigma_{x^2})^2 \Sigma_w - (\Sigma_x)^2 \Sigma_{x^2}}{\left[\Sigma_w \Sigma_{x^2} - (\Sigma_x)^2\right]^2} \\
&= \frac{\Sigma_{x^2}}{\Sigma_w \Sigma_{x^2} - (\Sigma_x)^2}
\end{aligned}$$

So the uncertainty in a is the (positive) square root of σ_a^2:

$$\sigma_a = \sqrt{\frac{\Sigma_{x^2}}{\Sigma_w \Sigma_{x^2} - (\Sigma_x)^2}} \qquad \text{and} \qquad \sigma_b = \sqrt{\frac{\Sigma_w}{\Sigma_w \Sigma_{x^2} - (\Sigma_x)^2}} \tag{7.6}$$

following a similar derivation for σ_b^2.

7.4 Goodness of Fit

In the method of least likelihood, we assume that the best description of a set of data is one that minimizes the weighted sum of the squares of the deviations between the data and the fitting function. Hence, the best parameters for a given fitting function are the ones that give the smallest χ^2. So, if we have several different fitting functions to choose from, is the one that gives the smallest χ^2 the best one to use? Not always.

Seven measurements are listed in Table 7.1 and are plotted in Fig. 7.1. On a visual inspection, the y values generally increase as the x values increase, so it's no surprise that a straight line with

i	x_i	y_i	σ_i
1	-5.48	1.3	0.8
2	-3.24	22	4
3	-0.15	37	2
4	1.30	55	5
5	3.37	56	5
6	6.82	87	5
7	10.94	114	8

Table 7.1: Seven measurements, *i.e.*, $N = 7$, of arbitrary physical quantities with uncertainties.

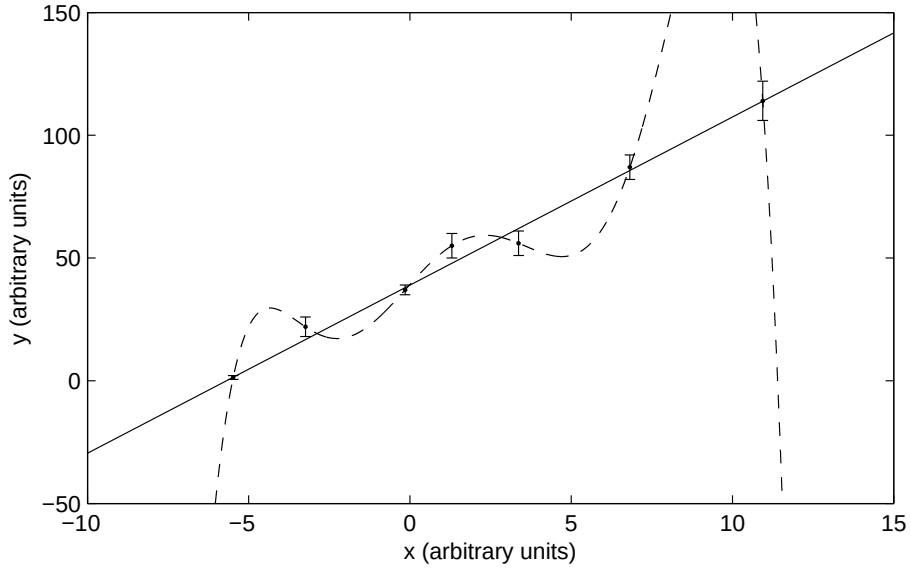

Figure 7.1: The seven data points from Table 7.1. The least-squares straight-line fit is shown as a solid line, and the least-squares fit to a sixth-order polynomial is shown as a dashed line.

$k = 2$ (solid line in Fig. 7.1) does a decent job of fitting these data. Using Eq. 7.4, 7.5, and 7.6, the fitted value of the slope and intercept are $b = 6.8 \pm 0.2$ and $a = 39.0 \pm 1.2$, respectively, giving a value of $\chi^2 = 5.52$.

But maybe other functions would fit the data even better. As an example of another fit function, the dashed line in Fig. 7.1 is the fit to a sixth-order polynomial: $a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6$, with seven constants $k = 7$. The minimum χ^2 is zero, because the function goes through every single data point. Is this a better fit than the straight line?

While a curve that goes through every single data point might seem optimal[4], it is often a poor representation of the physics governing the data. If we look at our 7 data points and attempt to extrapolate the behavior to, say $x = 12$, do we expect a measurement of y to be about $+120$ as the straight-line fit would indicate, or -50 as the sixth-order polynomial would indicate? Based on the overall trend in the data, the value of $+120$ appears to be more reasonable. If we knew that these data were, for example, temperatures measured along the length of a metal bar that was held at two different temperatures at each end, then we would be strongly biased toward the straight-line fit.

χ^2 is not the best metric to choose between two different fitting functions, however, in part because it allows this kind of quantitative ambiguity. Another reason χ^2 is not ideal is because it is in general larger with a larger number of data points. If the data really do come from a straight-line distribution, however, then more data should determine the line even better.

To address both of these issues, we divide the χ^2 by the *number of degrees of freedom* available in the fitting, defined as the excess of data points to fit parameters: $N - k$. The *reduced* χ^2

$$\chi^2_{\text{red}} \equiv \frac{\chi^2}{N-k} = \frac{1}{N-k}\sum_{i=1}^{N}\left[\frac{y_i - f(x_i)}{\sigma_i}\right]^2. \tag{7.7}$$

If $N \gg k$, then the $(N-k)^{-1}$ factor causes the value of χ^2_{red} to approximate the average, weighted, squared deviation, which means that increasing the number of data points doesn't in general increase χ^2_{red} even though it increases χ^2.

To determine the best fit function, we look for the one that gives the smallest value of χ^2_{red}. This favors fitting functions with fewer constants in them, so that we look more at trends rather than wiggles that happen go through all the data. The χ^2_{red} is not defined at all if $N \leq k$ as it is for our sixth-order polynomial fit to the data in Table 7.1, for which $k = 7$ and $N = 7$.

For those data, the straight-line fit has $\chi^2_{\text{red}} = 1.10$. Now, we might ask whether that is a "good" value of χ^2_{red}, or whether we should suspect that there might be a better fitting function using a higher-order polynomial. From a visual inspection of Fig. 7.1, the straight line certainly looks like a good fit. In fact, a χ^2_{red} of 1 is ideal, so 1.10 is extremely good. In the limit that $N \gg k$, a χ^2_{red} of 1 means that each term in the sum in Eq. 7.7 is about 1, that is, the deviations between the data and the fit function are, in general, about the same size as the uncertainty. If those deviations are, in general, much larger than the uncertainties, then we have a poor fit, and χ^2_{red} will be much larger than 1. If those deviations are, in general, smaller than the uncertainties, then either the experimentalist has over-estimated the uncertainties or our fitting function has too much freedom in it so that we're fitting noise, and χ^2_{red} will be smaller than 1. Another rule of thumb that some people use is that a good fit will pass within the error bars of about two-thirds of the data points. In our straight-line fit in Fig. 7.1, the line passes through four out of the seven error bars, $i = 1, 3, 6$,

[4] In fact, it is always possible to *perfectly* fit N data points to an $(N-1)^{\text{th}}$-order polynomial, but any fit to an N^{th}- or higher-order polynomial is not possible, because a matrix of the type of depicted in Eq. 7.3 is singular. Consider trying to fit a straight line (a first-order polynomial with $k = 2$) to a single data point ($N = 1$).

and 7. Using a quadratic fit function, for example, results in a slightly smaller total χ^2, but a larger $\chi^2_{\rm red} = 1.38$. A linear fit function is, therefore, preferred to a quadratic fit function.

In judging goodness of fit, we need both $\chi^2_{\rm red}$ as a quantitative measure, and observation of trends and physical insight as qualitative guides.

7.5 Least-Squares Linear Fitting Program

The `Matlab` function presented here takes data arrays as input, and then performs some basic error-checking on the input. If the three arrays are not all the same size, or if there are fewer than two data points, then `linfit` stops execution and exits. The output consists of the two fit parameters, their uncertainties, and $\chi^2_{\rm red}$.

..... Begin Program 22

file: `linfit.m`

```
function fitpar = linfit( x, y, sigmay )

% Purpose: This function will make a least-squares fit to data with
%          a straight line f(x) = a + bx.
% Usage:   x = independent variable array
%          y = dependent variable array
%          sigmay = array of uncertainties in dependent variable
%          Units on variables are determined by caller.
%          The output array fitpar contains the fit parameters:
%          fitpar(1) = intercept (a, with same units as y)
%          fitpar(2) = uncertainty in a (da)
%          fitpar(3) = slope (b, with units of y/x)
%          fitpar(4) = uncertainty in b (db)
%          fitpar(5) = reduced chi-squared (chi2red, unitless)

% Validate input

npar = 2;  % number of fitting parameters
npts = length( x );  % number of data points
if ( ( npts ~= length( y ) ) || ( npts ~= length( sigmay ) ) )
  display('Arrays must be of the same length.')
  return
end
if ( npts < npar )
  display('Not enough data points to fit a first-order polynomial.')
  return
end

% Accumulate weighted sums

w = sigmay.^(-2); % array containing weights
sumw  = dot(w,ones(npts,1)); % sum over w(i) times 1
sumx  = dot(w,x);            % sum over w(i) times x(i)
sumy  = dot(w,y);            % sum over w(i) times y(i)
sumx2 = dot(w,x.^2);         % sum over w(i) times x(i) squared
sumxy = dot(w,x.*y);         % sum over w(i) times x(i) times y(i)

% Calculate fit parameters

Delta = sumw*sumx2 - sumx*sumx;
a = ( sumx2*sumy - sumx*sumxy ) / Delta;
b = ( sumxy*sumw - sumx*sumy ) / Delta;
da = sqrt( sumx2 / Delta);
db = sqrt( sumw / Delta);
```

```
chi2 = dot( (y-a-b*x).^2, w );
if (npts > npar)
  chi2red = chi2 / (npts - npar);
else % npts = npar
  chi2red = 0;
end

fitpar = [ a, da, b, db, chi2red ];
```

..... End Program 22

7.6 Other Methods

Least-squares fitting is a type of *regression analysis*, but it is not the only method to fit data. One could, for example, use a spline, which does a type of piecewise fitting of subgroups of data. Nor is $\chi^2_{\rm red}$ the only method to characterize the goodness of the fit. For straight-line fits, one could instead use a linear-correlation coefficient. There are many, many more complicated methods for extracting information from data sets, but the least-squares method with a $\chi^2_{\rm red}$ test is the work-horse of data analysis when the uncertainties are small compared to the measurements. For the cases in which the measurements have large uncertainties, then more sophisticated methods are needed because the assumptions implicit in least-squares fitting are violated.

Of course, the most simple-minded fitting strategy is to use a black box. `Matlab` provides a polynomial fitting program called `polyfit`. If we use `polyfit` to determine the slope and intercept for our seven data points in Table 7.1, we find a slope of 6.69 and an intercept of 40.2. These values are slightly different from the ones we found using `linfit` because `polyfit` uses least-squares fitting, but it doesn't incorporate the uncertainties in the data. Quantitative comparison of the two results is not meaningful because the two programs use different data sets: one with uncertainties, and one without[5]. If we want to use the uncertainties to weight the data, or if we want to fit anything other than a polynomial, we can't use `polyfit`.

The advice on using black-box programs is always "Let the buyer beware!"

[5]Using `linfit` but with all the uncertainties set to be equal, gives the same slope and intercept as those given by `polyfit`.

Chapter 8

Statistical Measures

What can we learn from a set repeated measurements? How much do we trust our conclusions? And most importantly, what statistical tools exist so that we can be quantitative about it?

Consider the data set `x = [ x1, x2, ..., xN ]`, representing a sequence of measurements of a particular quantity, say the charge-to-mass ratio of the electron. If we make one measurement, then we don't expect it to be the correct value. If we make multiple measurements, however, we expect them to fall in a distribution about the correct value. The larger the number of measurements, the better we can describe that distribution. We can postulate the existence of a probability distribution function that determines the probability of getting any of the possible values if we do a single measurement. This theoretical function is called the *parent distribution*, and is what we would expect to see if we made an infinite number of measurements. Assuming that the distribution is symmetric about the mean and that the peak of the distribution occurs at the mean value, we take the mean of the parent distribution to be the correct value of the measurement, The more measurements we make, therefore, the more precisely the average of our measurements tells us the correct value.

First, we'll look at the statistical measures of our data, and then we'll connect that to some commonly used theoretical distribution functions.

8.1 Moments

Our set of N measurements comprise a sample from the parent distribution. Some characteristics of these data that might interest us are how much data we have, where the data are concentrated, and how spread out the data are. These three quantities are related to the statistical measures termed the zeroth, first, and second *moments* of the distribution. Higher moments are constructed to characterize the symmetry of the distribution (skewness), the peakedness (kurtosis), etc. We'll limit our discussion to the first three. We'll also assume that all of our measurements have the same experimental uncertainty, and that the uncertainty in our mean value is determined by statistics rather than the experimental uncertainty. In more advanced treatments, both types of uncertainties may be important.

Sample Mean: The first moment is the average of our measurements.

$$\bar{x} \equiv \frac{1}{N}\sum_{i=1}^{N} x_i \tag{8.1}$$

In the limit that $N \to \infty$, $\bar{x}$ approaches μ, the mean in the parent distribution. The *first central moment*, defined as the average of $x - \bar{x}$, is always zero.

Sample Variance: The second central moment is the average of the squares of the deviations from the mean[1]. Note well: s^2 is called the *variance*, not s.

$$s^2 \equiv \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2 \tag{8.2}$$

In the limit that $N \to \infty$, s approaches the *standard deviation* σ, the root mean square deviation from the mean of the parent distribution. A large value of s^2 means the data are widely distributed about the mean. Because s^2 has the same units as x_i^2, we need to specify what "large" means, *i.e.*, large compared to what?

Standard Error of the Mean: Here, we estimate the uncertainty in the mean, and we'll make an additional assumption. We already assumed that the uncertainties in each of the individual measurements are equal. Now, we'll assume that they are each equal to σ, which is approximately s if we have enough data[2].

To calculate the uncertainty in the mean determined from Eq. 8.1, recall how we determined uncertainties in calculated quantities in Chapter 7 by adding the uncertainties in quadrature. We begin by reminding ourselves that each x_i in principle has an associated σ_i, which we'll ultimately set equal to s by our assumption in the previous paragraph.

$$\sigma_{\bar{x}}^2 = \sum_{i=1}^{N}\left(\frac{\partial \bar{x}}{\partial x_i}\right)^2 \sigma_i^2 = \sum_{i=1}^{N}\left[\frac{\partial}{\partial x_i}\left(\frac{1}{N}\sum_{j=1}^{N} x_j\right)\right]^2 s^2 = \sum_{i=1}^{N}\left[\frac{1}{N}\right]^2 s^2 = \frac{1}{N^2} N s^2 = \frac{s^2}{N}$$

Only the term with $i = j$ survives the differentiation. Taking the positive square root gives

$$\sigma_{\bar{x}} \equiv \frac{s}{\sqrt{N}}\ . \tag{8.3}$$

Now, we're in a position to quantify a large or small variance: we do it by comparing $\sigma_{\bar{x}}$ to $\bar{x}$. If $\sigma_{\bar{x}}$ is small compared to $|\bar{x}|$, then there is a small uncertainty in the mean, so we have a reliable mean value. If $\sigma_{\bar{x}}$ approaches $|\bar{x}|$, then there is a large uncertainty in the mean. If $\sigma_{\bar{x}}$ is even larger than $|\bar{x}|$, then the value of $\bar{x}$ is not a convincing result. Experimentally, we can reduce the uncertainty in $\bar{x}$ by increasing the number of measurements, so long as $\sqrt{N}$ increases faster than s does as we add data to the set.

Binned Data: How do things change in Eq. 8.1 and 8.2 for data of the form (x_j, N_j) where N_j is the number of times x_j was found, within some interval, and there are N_b different values of x reported?

Typically, the x values in binned data are equally spaced. Defining this spacing as Δx, then $x_j = x_{j-1} + \Delta x$, and the number N_j might correspond[3] to all measurements between $x_j - \frac{1}{2}\Delta x$

[1] This is our $\chi^2_{\rm red}$ from Chapter 7 with a constant fitting function (one constant to fit: $k = 1$ with $f(x) = \bar{x}$) and unit uncertainty for all x_i.

[2] This hand-waving is not too crazy considering that uncertainties are estimates anyway, and that we usually assume that deviation between the correct value and any given measured value shouldn't be too much larger or smaller than the uncertainty.

[3] The number N_j might correspond, instead, to all measurements between x_j and $x_j + \Delta x$, or between $x_j - \Delta x$ and x_j. Such information should be given with the data set, because a modification of the calculation of $\bar{x}$ and s^2 might be needed.

and $x_j + \frac{1}{2}\Delta x$. In that case,

$$N = \sum_{j=1}^{N_b} N_j \ , \qquad \bar{x} = \frac{1}{N}\sum_{j=1}^{N_b} N_j x_j \ , \qquad \text{and} \qquad s^2 = \frac{1}{N-1}\sum_{j=1}^{N_b} N_j \left(x_j - \bar{x}\right)^2 \ . \tag{8.4}$$

Here, $\bar{x}$ can be interpreted as a weighted average, with weights N_j/N, which is the fraction of times x_j is observed. These expressions revert to our earlier ones if all of the N_j are either zero or one.

The `Matlab` function presented here takes binned data as input, and then performs some basic error-checking on the input. If the arrays are not the same size, then `getstats` stops execution and exits. The output consists of the mean, uncertainty in the mean, and the variance.

..... Begin Program 23

file: `getstats.m`

```
function stats = getstats( x, y )

% Purpose: This function will determine the statistical measures for
%          an array data sets (x,y) containing a distribution of values.
% Usage:   Both x and y are arrays of arbitrary length and arbitrary units
%          x = independent variables
%          y = number of occurrences of x
%          The output array stats contains the statistical measures.
%          stats(1) = mean value of x
%          stats(2) = uncertainty in mean
%          stats(3) = sample variance

npts = length( x );  % number of data points
if ( npts ~= length( y ) )
  display('Fatal Error: Arrays x and y must be of the same length.')
  return;
end

% Accumulate sums

sumy   = dot(y,ones(npts,1));  % sums over y(i) times 1
sumxy  = dot(x,y);             % sums over x(i) times y(i)
sumx2y = dot(x.^2,y);          % sums over (x(i))^2 times y(i)

% Determine statistical measures from sums

mean = sumxy / sumy;

if ( npts > 1 )
  variance = ( sumx2y - 2 * mean * sumxy + mean^2 * sumy ) / ( sumy - 1 );
else
  variance = 0;
end

dmean = sqrt( variance / sumy );

% Cast statistical measures into an array

stats = [ mean, dmean, variance ];
```

..... End Program 23

8.2 Probability Distributions

If we plot the binned data, we have a histogram. Dividing the number in each bin N_j by the total number of measurements N, gives us a vertical scale that tells us the proportion of times that a single measurement fell within one of the bins. If we were to *fit* the newly scaled histogram with a parent distribution function $P(x)$ and integrate that function[4] over all possible values of x, we'd get exactly 1. This is because the parent distribution is a probability distribution.

$$1 = \int_{-\infty}^{\infty} P(x)\, dx$$

The value 1 means that any single value that we measure will be within the range of possible values with 100% probability. The area under any subregion of the curve is the probability than a single measured value will be within that subrange.

We convert the expressions in Eq. 8.4 for the sample mean and variance for the binned data into the analogous expressions for the parent distribution by replacing the sum over the discrete x_j with an integral over the continuous x, and all the N_j/N with the probability distribution. We give the quantities from the parent distribution new names, μ and σ^2 as we've already referred to them, to distinguish them from those from the measurement sample, $\bar{x}$ and s^2.

$$\mu = \int_{-\infty}^{\infty} P(x)\, x\, dx \qquad\qquad \sigma^2 = \int_{-\infty}^{\infty} P(x)\, (x-\mu)^2\, dx \tag{8.5}$$

In transforming to the parent distributions, we have also assumed that N is very large so that $N - 1 \simeq N$.

There are many, many different probability distributions, but here is a brief list of some that are commonly seen in physical systems.

Uniform: The uniform probability distribution is used in cases of equal likelihood to measure any value of the independent variable within some range. It is zero outside the allowed range, flat within that range, and symmetric about the mean. See the solid line in Fig. 8.1 for an example of a uniform distribution.

$$P_U(x, x_{\min}, x_{\max}) = \begin{cases} 0 & \text{for } x < x_{\min} \text{ or } x > x_{\max} \\ \frac{1}{x_{\max} - x_{\min}} & \text{for } x_{\min} < x < x_{\max} \end{cases}$$

Using Eq. 8.5, the mean $\mu = \frac{1}{2}(x_{\max} + x_{\min})$ is just the midpoint in the range, and the standard deviation $\sigma = (x_{\max} - x_{\min})/\sqrt{12}$.

A uniform probability distribution is what we might see if we were to measure the position at an arbitrary time of an object on a horizontal, frictionless surface, bouncing in one dimension between two rigid barriers — one at $x_{\min}$ and the other at $x_{\max}$ — with which the object has perfectly elastic collisions. Without knowing the time or the object's initial location and velocity, we have no ability to follow the object's trajectory. We conclude, based on the limited information available to us, that the object is equally likely to be anywhere between $x_{\min}$ and $x_{\max}$ at that instant[5].

[4] This integral is the *zeroth central moment*, or the weighted average of $(x-\mu)^0$.

[5] If we incorporate extra information about the system, like the time of the measurement and the object's initial location and velocity, then, of course, the object's position at a later time can be determined using kinematics. This extra information constitutes a qualitatively different measurement from the one described.

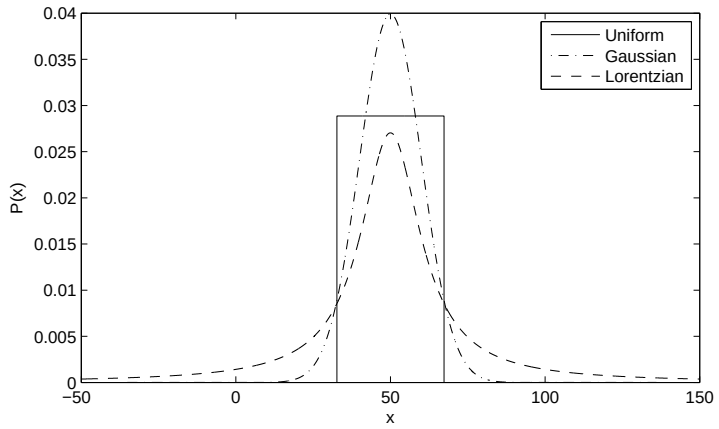

Figure 8.1: Three different probability distributions, each with mean of 50.0. The uniform distribution shown has the same standard deviation as the Gaussian distribution, namely 10.0, and the Gaussian has the same FWHM as the Lorentzian, namely 23.5. The area under each curve is 1.

Normal: The normal probability distribution is described by a *Gaussian function*. It is called "normal" because it regularly occurs in statistical samples, and is also referred to as a "bell curve". See the dot-dashed line in Fig. 8.1 for an example of a Gaussian distribution.

$$P_G(x,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

Data for which the parent distribution is normal will have a mean of μ, a standard deviation of σ (not a coincidence), and be symmetric about the mean. The largest probability occurs if the measurement $x = \mu$, and at the value of $x = \mu + \sigma$, the probability has fallen to $e^{-1/2} \simeq 0.606$ of its peak value. If we integrate the distribution from $\mu - \sigma$ to $\mu + \sigma$, we find approximately 0.68. This means that roughly 68% of the data drawn from a normal distribution lie within one standard deviation of the mean. Roughly 95% of the data lie within two standard deviations of the mean, *i.e.* between $\mu - 2\sigma$ and $\mu + 2\sigma$. It is unlikely, with only a 5% probability, that a single measurement will fall outside of two standard deviations from the mean.

There are many occurrences of the Gaussian distribution in physics and statistics[6], but one important example is that of measurements with observational errors, which leads to the adding of errors in quadrature, and the derivation of the least-squares algorithm. The Gaussian function is also straightforward to integrate and differentiate, so it is often used just for convenience.

Lorentzian: The Lorentzian (or Cauchy) probability distribution is similar to a Gaussian, but with longer tails. See the dashed line in Fig. 8.1 for an example of a Lorentzian distribution.

$$P_L(x,\mu,\Gamma) = \frac{1}{\pi}\,\frac{\Gamma/2}{(x-\mu)^2 + (\Gamma/2)^2},$$

where Γ is defined as the full width of the distribution measured at half the level of the maximum probability, the so-called full-width at half-max (FWHM). Due to the long tails,

[6]The *Central Limit Theorem* states that different samples from any given parent distribution with have means that vary with respect to the parent mean according to a normal distribution, independent of the type of parent distribution.

the integral appearing in the calculation of the standard deviation does not converge, which is why the FWHM is instead used to characterize the width of the distribution.

The Lorentzian arises in the formal solution to the differential equation describing driven resonances. Consequently, it is often seen in peak shapes in spectra from resonant phenomena in particle decays and collisions, LRC circuits, and detector response functions. It also appears in collisional broadening of peak shapes in general spectroscopy.

Delta Function: The delta-function probability distribution is used when there is 100% probability to find just one value. There is no closed-form, analytic expression for a delta function[7]. We write $\delta(x-\mu)$ to indicate a delta-function probability distribution with a mean of μ and zero standard deviation. We can think of it as the limit of a uniform distribution, or of a Gaussian distribution, with $\sigma \to 0$. There are also integral representations.

The delta function is truly pathological, but it has wide-ranging theoretical applications, especially in quantum mechanics and in signal processing.

Poisson: The Poisson probability distribution has a mean is equal to the square of the standard deviation, but this distribution is only applicable for measurements in which x is a non-negative integer.

$$P_P(x,\mu) = \frac{\mu^x}{x!}\, e^{-\mu}$$

The Poisson distribution is decidedly not symmetric about the mean for small values of μ, but approximates a Gaussian (restricted to $\mu = \sigma^2$, for which it is necessary that x is unitless) for large values of μ.

Poisson statistics are used, for example, if we observe x events per unit time. This occurs in radioactive decay processes for which the probability of a decay occurring in that time interval is small.

8.3 Numerical Modeling

Often in the physical sciences, we are not able to analytically calculate an outcome, either because the physical system is too complicated, or the mathematics is too difficult or even impossible. In such cases, we often resort to numerical modeling, sometimes called simulations.

In the *Monte Carlo method*[8] of numerical modeling, random numbers are drawn from a preselected probability distribution, and those numbers are then used to construct physical observables, the distributions of which are compared to the distribution of data. The discrepancy between the Monte-Carlo distribution and that of the real data is a measure of the quality of the physical model used to construct the Monte-Carlo distribution from the random numbers. The model is then tuned to optimally fit all available data. After the best fit to existing data is determined, the model can then be used to predict the results of new measurements.

As a first ingredient to a Monte-Carlo model, we need random numbers. `Matlab` provides two random number generators: `rand` returns random values uniformly distributed between 0 and 1 (*i.e.*, with a mean of 1/2 and a standard deviation of $1/\sqrt{12}$), and `randn` returns random values normally distributed with a mean of zero and a standard deviation of 1.

[7] Actually, mathematicians get very angry with us for calling it a *function*. It is properly termed a distribution, because it only really has meaning as part of an integrand.

[8] Gambling is based on probabilities and random outcomes. *Monte Carlo* is the name of a famous casino in Monaco, and was the code name for this technique, developed in the 1940s at then top-secret Los Alamos National Laboratory.

We can modify these distributions to have the moments we want. For example, consider the following two lines of code, for which `xmin` and `xmax` are previously assigned values:

```
x0 = rand(1000,1);
x = x0*(xmax-xmin) + xmin;
```

The first line fills a 1000-element array (a 1000-by-1 matrix) named `x0` with random numbers *uniformly* distributed from 0 to 1. The second line performs a linear transformation[9] on `x0` so that it is still uniformly distributed, but now the range is from `xmin` to `xmax`. If `xmin` is 4.5 and `xmax` is 6.9, for example, then the mean of these 1000 items of simulated data will then be approximately 5.7 and the standard deviation 0.69. Alternatively, we could achieve the same result with the transformation

```
x = x0*sqrt(12)*sigma + mu - sqrt(3)*sigma
```

in which both `mu` and `sigma` have been previously assigned to 5.7 and 0.69, respectively.

The more simulated data we have, the closer its distribution will be to the parent distribution (with its intended mean and standard deviation), just like with a sample of real data. From Eq. 8.3, we know that the uncertainty in the mean decreases as the square root of the number of data points increases. That square root tells us that we need to make a factor of 25 more data, for example, to reduce the uncertainty in the mean by a factor of 5. Making more simulated data, and then running them through some model, however, requires more computer time or more human effort to re-write programs to speed them up. In modeling complex systems, just as in laboratory work, we are often confronted with the issue that smaller uncertainties come with a price.

If we want, instead, to create a *normally* distributed set of random numbers with an approximate mean of `mu` and standard deviation of `sigma`, where both variables are previously assigned values, then we could do the following:

```
y0 = randn(1000,1);
y = y0*sigma + mu;
```

Again, we've done a linear transformation, so the resulting distribution will still be normally distributed. The linear transformation here is different from the one for uniformly distributed data because both the means and the standard deviations of the parent distributions produced by `rand` and `randn` are different.

Typically, the simulated data are treated as an initial distribution of some physical quantity, which is then sent through a deterministic calculation to produce distributions of predicted final quantities. Then the moments of those final distributions are compared to in-hand data, or are used as a prediction of future measurements. Alternatively, we can the think of the simulated data as the fitting function in constructing a χ^2 (as in Chapter 7), and iteratively modify the Monte Carlo until we get the smallest χ^2.

On Random Number Generators

Most programming languages have built-in random number generators, but none of them is truly random, because computers are designed *not* to do random things. There are many different algorithms for constructing pseudorandom numbers, but most of them function by manipulating

[9] Linear transformations do not change the type of probability distribution. Examples of transformations that do alter the type of distribution are logarithmic and square-root transformations, which both changes the relative spacing between items in the distribution.

digits in a large integer, or manipulating bits in a section of memory. Each algorithm needs an initial "seed" as input to the function: either a system variable, *e.g.* the clock time in seconds, or a user-selected integer.

One of the most common failure modes for a random number generator is that it generates a sequence of random numbers which has some internal patterns in it. As black boxes go, the functions `rand` and `randn` are fairly robust against internal patterns. The claimed period (number of random numbers generated before a repeating pattern emerges) is on the order of 10^9. If we are constructing a simulation using more than a billion data points, however, we may find trends in the output distributions simply because there are trends in the inputs. To minimize this problem, one can play tricks like generating sets of smaller sequences, each with different seeds or different choices of algorithms for drawing the random numbers within `rand`, and then concatenating those sequences.

Appendix A

Programming Tips

Computer programming is a very powerful skill, but the process can be frustrating as one learns the logic and idiosyncracies of any computer language. Some important philosophers have general advice for us.

> *Simplify, simplify, simplify.* – Thoreau
>
> *One 'simplify' would have sufficed.* – Emerson
>
> *Programming is an art form that fights back.* – Unattributed on Wikiquotes

But here is some advice that is specific to `Matlab`.

File Locations

Any function that we want to run must be in a file located in either the Current Directory, or in the Search Path.

The Search Path is a predefined set of directories (sometimes called folders) that `Matlab` uses to organize its internal files and functions. Type `path` in the `Matlab` Command Window to see which directories are in the Search Path. The Search Path can be modified, but there is no reason to do so at the level of programming required in PHSX 331.

The Current Directory is identified near the top of the `Matlab` display environment. Assuming that we have not modified the Search Path and we download a function from the class web site to the Desktop, `Matlab` can only find it if the Current Directory is set to be the Desktop. When we save a file from within `Matlab`, either a function or a plot, it is placed by default into the last directory we saved files to, which may not be the Current Directory.

Naming Conventions

A text file called an M-file that contains a function statement line like the following:

```
function [out1, out2, ...]  = funname(in1, in2, ...)
```

defines the function named `funname` that accepts inputs `in1`, `in2`, etc. and returns outputs `out1`, `out2`, etc. Note the precise locations of brackets, `[` and `]`, parentheses, `(` and `)`, commas, and the equals sign. The ellipses, ..., are only there to indicate that these lists of input and output variables can be longer. Ellipses are not allowed in an actual function statement, but form one of a set of special characters (see below).

- The name of a function should be the same as the name of the file without the `.m` filename extension. In the example above, the M-file should be named `funname.m`.
- All instructions within the file must occur after the function statement line.
- Functions need not have input variables, in which case even the parentheses can be omitted. Functions need not have output variables, in which case the brackets and equals sign can be omitted, and they are then referred to as scripts.
- The variables within the body of the function are all local variables. Local variables exist only during the time that the function is actually executing. Once the function is finished executing, those values are gone.

Choose variable names and function names that mean something so that they're easy to keep track of. For example, when specifying an angle, we might call the variable `theta` instead of `t`. The variable name `launchangleoftheprojectilewithrespecttohorizontal` is nicely descriptive, but hard to read and subject to typos that will be hard to find. Avoid the upper-case letter O and the lower-case letter l as single-letter variable names, because they are very difficult to distinguish visually from the numbers zero 0 and one 1, respectively.

`Matlab` is a case-specific language. This means that the variables `pi`, `Pi`, `pI`, and `PI` can all be used to mean different things in one function. It is unwise to do so, however, because confusing upper- and lower-case letters is easy to do when typing, and can be difficult to find when looking for errors.

Operators and Special Characters

Special characters are those that have definite meaning to `Matlab` and may not be used as a part of the names of variables or functions. For example, `f*9_[` is not an acceptable variable name because both the asterisk `*` and the bracket `[` are special characters. The underscore `_` is not a special character. Variables names can contain numbers like 9, but they cannot begin with a number. The variable name `distance2` is valid, but `2themoon` is not.

Variable names and function names cannot contain spaces or blank lines. Use blank lines and spaces within a line to make programs easier to read. (Easier to read also means easier to debug.)

- The equals symbol `=` is used in two ways: to separate input variables from the function name in a function statement, and to set variable to the left of the equal sign to the value of the calculation to the right of the equal sign.
- The mathematical operators `+`, `-`, `*`, and `/` represent addition, subtraction, multiplication, and division. The symbol `^` means "raised to the power of" (e.g., `4^2` is 16). The exclamation point `!` is not used to make factorials (e.g., `4!` provokes an error message). Instead, we use `factorial(4)`.
- The semicolon `;` at the end of a line suppresses output of that line's calculation to the command window.
- The percent symbol `%` preceeds a single-line comment, prose for the user which `Matlab` ignores.
- Parentheses `(` and `)` are used in two ways: to delimit input to a function, and to modify algebraic order of operations.
- Brackets `[` and `]` are used to construct arrays.

- The ellipsis `...` is used to continue a statement on the next line. This can make a function easier to read, rather than having one big long line.
- The colon `:` is used to specify looping iterations and array dimensions.
- The comma `,` is used to separate items in a list (e.g., several input variables).
- The relational operators are used in `if` statements. Greater than and less than are the usual `>` and `<`, respectively. `<=` means less than or equal to. `>=` means greater than or equal to. `==` means equal to (exclusively in an `if` statement). `~=` means not equal to.
- The logical operators `&&` and || mean "and" and "or", respectively.

There are other such characters, and combinations of characters, but we will not encounter them in PHSX 331. A full listing of *special characters* is available under the Help utility within `Matlab`.

Appendix B

Precision & Round-off

As far as we are concerned a computer is a machine for storing and manipulating numbers. When we type

```
>> x = 19
```

on the `Matlab` command line the computer (a.k.a `Matlab`) performs the following tasks. It sets aside a small piece of its memory, a *memory location*, and lets us refer to this bit of memory by the name `x`.[1] Next it records the number `19` in this memory location. Next time we refer to the variable `x` it should still contain the number `19`, which we may choose to change. For instance, after we type the following statement

```
>> x = 2*x + 7
```

the memory location known as `x` will contain `45`. Typing

```
>> who
```

will list the names of all variables currently being used.

We as Physicists would like to remain ignorant of such gruesome details whenever possible. It will help us, however, to know a few basic things about the way computers store numbers in memory. The first fact of note is the *numbers can be stored in two different ways: 1. as Integers or 2. as Floating point numbers*. There is more to this distinction than whether anything comes after then decimal place. The data types commonly called INT and FLOAT are stored in the computer quite differently, thus a number, like 19, will appear differently in the computer memory if it is stored as an INT than if it is stored as a FLOAT. If this sounds confusing, it is. And because it's confusing the program `Matlab` uses only *one* representation to store all numbers: Floating point. It will help to consider the alternative briefly.

B.1 Integer storage

An integer is a positive or negative number like 3, 37 or -2,472. As you may know, computers store numbers in their binary representation. The binary expression for 19 is

$$19 \;=\; \underbrace{10011}_{\text{binary}} = 1\times 2^4 \;+\; 0\times 2^3 \;+\; 0\times 2^2 \;+\; 1\times 2^1 \;+\; 1\times 2^0 \quad .$$

[1] The computer itself knows the location by a less memorable name like `0x035A3BF`. It is the principle job of a program like `Matlab` to translate between simple human names like `x` and complicated computer addresses. With any luck you will never need to understand how computer addressing really works.

Each 1 or 0 is a single *bit* in the binary representation. For convenience a computer's bits are grouped into sets of eight called *bytes*. Computers typically allocate either two or four bytes to store integers. Using two bytes the number 19 would appear with eleven leading zeros

$$19 \;=\; \overbrace{\underbrace{0000\,0000}_{\text{byte 1}}\;\underbrace{0001\,0011}_{\text{byte 2}}}^{\text{2 byte integer}} \quad .$$

Negative numbers are distinguished from positive numbers using the left-most bit.[2] If this bit is 0 the number is positive, if it 1 the number is negative. Thus the largest positive integer which can be stored in two bytes is

$$\underbrace{0111\,1111}_{\text{byte 1}}\;\underbrace{1111\,1111}_{\text{byte 2}} \;=\; 2^{15}-1 \;=\; 32,767$$

Changing the leading zero to a one the most negative number is $-32,767$. Four byte integers are used when it is necessary to store numbers outside the range [−32767, 32767]. What is the largest number you can store as a four byte integer?

B.2 Floating point numbers

To store a number which is not an integer, such as 19.25, the computer uses floating point representation. This works exactly like scientific notation where 19.25 would be written

$$19.25 \;=\; 1.925\times 10^{1} \;=\; \underbrace{+}_{\text{sign}} \quad \underbrace{1.925}_{\text{mantissa } M} \quad \times \quad \underbrace{10}_{\text{base}}{}^{1} \;\leftarrow \text{exponent } E$$

The rules for scientific notation are

1. The base in our decimal notation is always 10.
2. The exponent E is an integer.
3. The mantissa M is in the range $1 \le M < 10$. For any value M' outside this range it is always possible to change the exponent and shift the decimal place until the mantissa is in the allowed range. e.g. $0.04\times 10^4 = 4.0\times 10^2$.
4. The sign, either + or −, applies to the mantissa.

Since the computer uses binary rather than decimal notation, its version of scientific notation appears strange at first

$$19.25 \;=\; 1.001101\times 2^{4} \quad .$$

The mantissa is a simply a generalization of the standard binary notation, to include negative powers of 2

$$\begin{aligned} 1.001101 &= 1\times 2^{0}+0\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3}+1\times 2^{-4}+0\times 2^{-4}+1\times 2^{-5} \\ &= 1+\tfrac{1}{8}+\tfrac{1}{16}+\tfrac{1}{64} = 1.203125 = 19.25/16 \end{aligned}$$

The rules for binary scientific notation are slightly changed[3] for the decimal rules

[2] The remainder of the number may be represented differently, using a technique known as "ones compliment", but that is irrelevant for the present discussion.

[3] Actually only rules 1 and 3 change.

1. The base in binary notation is always 2.
2. The exponent E is an integer.
3. The mantissa M which is in the range $1 \leq M < 2$.
4. The sign, either + or −, applies to the mantissa.

A floating point number is stored in computer memory by storing its exponent, mantissa and sign (not necessarily in that order). This is typically done using four or eight bytes. Four byte floats are often called *single precision* while eight byte floats are *double precision*.

For illustration let us consider a hypothetical single precision (4 bytes = 32 bits) storage scheme. This is not the scheme used by `Matlab` but is not unlike many single precisions schemes in use. The exponent is stored as an eight bit integer, the mantissa using 23 bits and the sign using the remaining bit. Thus 19.25 would appear as

$$19.25 \;=\; \underbrace{0}_{+}\;\overbrace{1.\underbrace{001\,1010\,0000\,0000\,0000\,0000}_{\text{mantissa}}}^{\text{23 bits actually stored}}\;\overbrace{\underbrace{0000\,0100}_{\text{exponent}}}^{\text{8 bits}}$$

where the decimal point is not actually stored by the computer. A skeptical reader may have noticed that the mantissa shown above actually includes 24 0s and 1s. Since the mantissa is always in the range $1 \leq M < 2$, however, it will *always* have a 1 before the decimal place. The computer does not need to store this 1 since it can be assumed to be there.

What actually appears in the computer is the cryptic sequence of 0s and 1s

$$\overbrace{\underbrace{0001\,1010}_{\text{byte 1}}\;\underbrace{0000\,0000}_{\text{byte 2}}\;\underbrace{0000\,0000}_{\text{byte 3}}\;\underbrace{0000\,0100}_{\text{byte 4}}}^{\text{4 byte float}}$$

In order to interpret this it is necessary to know the memory is storing a FLOAT and not a four byte INT. If we (erroneously) read this as an INT it is 27262980. Fortunately for us, there is no chance that `Matlab` might make such a mistake.

The reason all of these details are important is that they determine how accurately the computer can represent a given number. First of all the largest positive number which can be represented is determined by the exponent E. The largest positive exponent representable by eight bits is

$$\max(E) \;=\; 0111\,1111 \;=\; 2^7 - 1 \;=\; 127$$

The largest mantissa

$$\max(M) \;=\; 1.111\,1111\,1111\,1111\,1111\,1111 \;\simeq\; 2$$

is extremely close to 2 independent of how many bits the mantissa has (recall that 2 is the upper bound for M). Thus the largest positive number possible is

$$2 \times 2^{127} \;=\; 2^{128} \;\simeq 3 \times 10^{38} \quad .$$

(Actually, the number 2^{128} is *just a bit* larger than the largest possible number. Remember the mantissa $M < 2$.) If you attempt a calculation which would give rise to a number larger than this, say `exp(1000)`, the computer will complain with a message about "floating point overflow". This

simply means the number is too big to represent. 10^{38} is a truly huge number, and it is unlikely you will ever need to work with numbers this large. In almost all cases a "floating point overflow" message is symptomatic of an error in your calculation. Note that very negative numbers, like `-exp(1000)` also constitute "overflows".

The smallest positive number has the smallest mantissa $M = 1$ and the most negative exponent $E = -127$,

$$1 \times 2^{-127} \simeq 6 \times 10^{-39} \quad .$$

Attempts to create numbers smaller than this, say `exp(-1000)`, will result in "floating point underflow" errors. These are less serious than overflows, and the computer will usually set the result to exactly zero, i.e. `exp(-1000) = 0`, and continue on its merry way. After all, an underflow is a number which is too close to zero.

Even though it is possible to represent numbers as small as 10^{-38} the computer cannot actually store 38 digits after the decimal place. The number of significant digits is determined by the number of bits in the mantissa. Consider the floating point representation of the number $1 = 2^0$

$$1 \;=\; \underbrace{0}_{+}\;\underbrace{1.000\,0000\,0000\,0000\,0000\,0000}_{\text{mantissa } M=1}\;\;\underbrace{0000\,0000}_{\text{exponent } E=0}$$

The next largest number possible on the computer is the one where the last bit (the least significant bit) of the mantissa is changed from 0 to 1

$$1+\epsilon_{\rm M} \;=\; \underbrace{0}_{+}\;\underbrace{1.000\,0000\,0000\,0000\,0000\,0001}_{\text{mantissa}}\;\underbrace{0000\,0000}_{\text{exponent}}$$

This least significant bit, 23 places after the decimal, corresponds to

$$\epsilon_{\rm M} \;=\; 2^{-23} \;\simeq\; 1.2 \times 10^{-7} \quad .$$

Thus the computer is capable of representing 1.0000001 (decimal), a number containing eight significant digits. Here, we have referred to digits, which are base 10, because that is the number system we normally use. It is easy to see that this convenient idea is only approximate since the true limit is binary: 2^{-23} which is only approximately 10^{-7}.

It is possible to represent numbers smaller than 10^{-7}, for example

$$2^{-30} \;\simeq\; 10^{-9} \quad .$$

If we add this number to 1, however, we get the decimal expression

$$1 + 10^{-9} \;=\; \underbrace{1.0000000}_{\text{stored}}\,01$$

which is the computer regards as 1, since it can only store 7 digits as a single precision float. As far as the computer knows

$$1 + 10^{-9} = 1$$

This is called a *round-off* error, since the computer has "rounded off" the digits after the seventh. For any computer there is a machine precision $\epsilon_{\rm M}$ which describes the level of the roundoff error. It is the smallest number which can be added to 1, and give a result different from 1. When the computer stores or calculates a number x, it should be thought of as having a small amount of noise associated with the round-off error,

$$x \text{ stored} \;=\; x\,(1 \pm \epsilon) \quad ,$$

where the $\pm\epsilon$ indicates the round-off error.

`Matlab` stores all of its data in *double precision*, using 52 bits for the mantissa. Thus for `Matlab` the machine precision is

$$\epsilon_{\rm M} \;=\; 2^{-52} \;\simeq 2\times 10^{-16} \quad .$$

To see this type the following into the command line

```
>> a = 1 + 1.0e-16
>> a - 1
```

Why does $a-1=0$? Compare that to the statement

```
>> a = 1 + 2.0e-16
>> a - 1
```

While the result is not very accurate (why not?) it is clear that the computer has enough precision to distinguish between 1 and $1+2\times 10^{-16}$. For convenience `Matlab` has a variable `eps` already set to $\epsilon_{\rm M}$. You can see its value by typing

```
>> eps
```

Problem A.1 For the mantissa `Matlab` uses 52 out of the 64 bits in the double precision word. How much does that leave for the exponent? What would be the largest possible number `Matlab` could represent? Test this.

What to worry about

There are many things in life which should worry you. Machine precision is not one of them.

Your task, as a computational physicist, is to calculate accurate answers on the computer. You may wonder, "Will my final answer be compromised by the finite precision with which the computer can represent a number?" The reply is almost always "*Not by a long shot.*" It is true that each number stored on a computer has a small error of relative size $\epsilon_{\rm M}$. It is also true that as these numbers are used in a sequence of arithmetic operations the errors accumulate. This can often result in cumulative errors many times $\epsilon_{\rm M}$. Still, there are numerous other errors involved in doing computational physics, almost all are vastly larger than $\epsilon_{\rm M}$. We encounter these types of errors in numerical integration, for instance. Until we discuss this you might also consider the kinds of numbers you feed into the computer: initial velocities, masses of balls, masses of elementary particles. How accurately you can any of these be known? In most cases using three significant digits is all that is warranted. The fact that `Matlab` stores 16 digits should strike you as naively optimistic.

There is one time where it pays to be aware of machine precision: at *subtraction* time. When subtracting two numbers which are very nearly equal, the result will be greatly compromised. Let's consider this in an example with decimal numbers, stored to seven decimal places. Consider the subtraction

$$\begin{array}{rlll} & 4.3874734 & \times & 10^{-5} \\ - & 4.3874621 & \times & 10^{-5} \\ \hline & 0.0000113 & \times & 10^{-5} \end{array} \;=\; 1.13 \underbrace{00000}_{\text{not signif.}} \times 10^{-10}$$

Note that the result contains 5 zeros which were "shifted" into place when the exponent was changed from -5 to -10. These digits no longer represent real information; they are not significant.

It is always wisest to avoid such pathological subtractions whenever possible. There are certain problems, however, in which they are inevitable. Inverting certain types of matrices is one example. Double precision exists at all, only because of such subtraction problems.

Appendix C

Using Arrays in Matlab

We are already familiar with variables containing numerical values, such as

```
>> x = 19
```

which creates a variable `x` and assigns it the value `19`. This is an example of a *scalar* variable; it addresses a single eight-byte memory location. An *array* is a variable which addresses *many* memory locations. The command

```
>> a = [ 1, 4.4, -2.4, 18, 7 ]
```

creates the variable `a` which is a 5 element array. The name `a` refers to the entire collection. The values can be addressed individually using an array *index*; the command

```
>> a(2)
```

will print out the second element in the array: `4.4`. `Matlab` indexes the array elements 1, 2,... up to the number of elements in the array (in this case 5).[1] Attempting to use an index larger than the size of the array will produce an error

```
>> a(7)
???  Index exceeds matrix dimensions.
```

The array `a` above is an example of a *one-dimensional* array, otherwise called a *vector*. It is possible to define arrays with any number of dimensions, however, one and two are the most popular choices. A two-dimensional array is sometimes called a *matrix*. The command

```
>> b = [ 4, 3, 6, 5; 0, 8, 3, 2 ]
```

defines a 2×4 matrix,

$$b = \begin{bmatrix} 4 & 3 & 6 & 5 \\ 0 & 8 & 3 & 2 \end{bmatrix} \quad .$$

Addressing individual elements requires two indices:

```
>> b(1, 3)
```

[1] Some programming languages, such as C, choose to begin indexing at 0, instead of 1. You might recall the dozens of news stories leading up to New Years 2000, which pointed out that it was *not* actually the new millennium. The endless discussion of this point hinged on the difference between these two indexing schemes. Similar confusion reigns in computer languages since some start indexing at 0 while others, such as `Matlab` and FORTRAN beginning at 1. The only advice we can offer is that if you plan to be multi-lingual you must be adaptable.

returns the element in row 1 column 3, the matrix element $b_{13} = 6$.

Matlab's forté is its ability to manipulate arrays (hence the name Matlab, derived from MATrix). A few handy tidbits are given below. Unfortunately, this only scratches the surface of Matlab's many array operations. You should consult the online Help facility for the rest.

- A ":" be used to refer to a sub-range of indices

  ```
  >> d = a(2:4)
  ```

  defines d to be a 3-element vector containing a(2), a(3) and a(4). In other words it contains [4.4, -2.4, 18]. If you omit the first or second index in the range Matlab will assume you mean the beginning or end of the array:

  ```
  >> e = a( 3: )
  ```

  means the elements of a() from a(3) to the end; in this case a(3), a(4) and a(5).

- Accessing a specific row or column of a matrix will produce a vector. This is done with the character ":" alone (i.e. all the way from the beginning to the end). Typing the command

  ```
  >> c = b(:, 3)
  ```

  defines the variable c to be a vector containing all rows in the third column of the matrix b. So c will be a two-element vector containing [6, 3].

- The colon can also be used to create arrays which contain a range of increasing values. The command

  ```
  >> f = 5:10
  ```

  defines f to be a 6-element vector containing the numbers 5 through 10. This command is equivalent to typing

  ```
  >> f = [ 5, 6, 7, 8, 9, 10 ]
  ```

  but much easier.

- With arrays in the picture the concept of multiplication becomes much trickier. Suppose we have two 3×3 matrices

  ```
  >> w = [ 1, 2, 3; 4, 5, 6; 7, 8, 9 ]
  >> u = [ 3, 7, 1; 6, 2, 5; 0, 2, 3 ]
  ```

  which result in the assignments

  $$w = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad , \quad u = \begin{bmatrix} 3 & 7 & 1 \\ 6 & 2 & 5 \\ 0 & 2 & 3 \end{bmatrix} \quad .$$

  Typing a command to multiply the arrays

```
>> v = w*u
```

will perform the operation of *matrix multiplication*. That is to say, `v` will be defined to be a 3×3 matrix whose elements are set to

$$v_{ij} \;=\; \sum_{m=1}^{3} w_{im} v_{mj} \quad , \quad v = \begin{bmatrix} 15 & 17 & 20 \\ 42 & 50 & 47 \\ 69 & 83 & 74 \end{bmatrix}$$

There are situations where we would like to multiply w and v *element-by-element*, rather than by matrix multiplication. This second type of multiplication is denoted by "`.*`" rather than the simple "`*`". So typing the command

```
>> s = w.*u
```

defines `s` to be a 3×3 matrix whose elements are set to

$$s_{ij} \;=\; w_{ij}\, u_{ij} \quad , \quad s = \begin{bmatrix} 1 \times 3 & 2 \times 7 & 3 \times 1 \\ 4 \times 6 & 5 \times 2 & 6 \times 5 \\ 7 \times 0 & 8 \times 2 & 9 \times 3 \end{bmatrix} = \begin{bmatrix} 3 & 14 & 3 \\ 24 & 10 & 30 \\ 0 & 16 & 27 \end{bmatrix}$$

It is important to see the distinctions between these two types of multiplications.