General Instructions

- Each week hand in your assignment in the form of an “executive summary”. Imagine this to be the
document you would give to your manager, if you were in an industrial setting, or to your advisor if
you were in graduate school. Make your report succinct and focused on what you did. Don’t bother
copying much from the text (I wrote it), but do try to express the point of the work using your own
words.

- Each report can be on paper or placed on the Web with the instructor notified by email (preferred).
   If Web-based, email rubin@physics.orst.edu when each project is ready and include its Web-accessible
   location.

- To save you time in the long run, do it right. Before you begin working on your assignment or running
code, write down (and save this as part of your report):
  1. the equations you are to solve or simulate (the math)
  2. the numerical method being used (the algorithm)
  3. the code listing (the modified code that you ran)

- For an A grade, answer all parts of the assigned problem including these 5 items:
  1. the equations you are to solve or simulate
  2. the numerical method being used
  3. the code listing
  4. edited results (just the edited good stuff): graphs, short tables, or something visual
  5. critical analysis: what did you learn, are you convinced, how could we do this better.

- No credit will be given for just running a code we give you; you should be modifying, extending,
  applying, or rewriting it as part of understanding the problem.

- You may discuss your projects with other people, but the report you submit must reflect your own
  work. If you submit it, you have agreed that you are prepared to explain it to the instructor.

Week 1  Reading 1.1-1.3, 2.1-2.15,
2.6
1. To gain some experience with your computer system, use an editor to key into a file one of the programs area.c, areas.f or Area.java. Then save your file to disk by saving it in your home/personal directory for this week.

```java
/*
 ***********************************************************
 * Area.java: Area of a circle, sample program()
 * copyright RH Landau, Oregon State University Physics()
 *********************************************************** */

public class Area
{
    public static void main(String[] argv) // Begin main program 
    {
        double radius, circum, area, PI = 3.141593; // Declare, assign doubles
        int modelNumber = 1; //Declare. assign integer
        radius = 1.;
        circum = 2.* PI* radius; // Calculate circumference
        area = Math.pow(radius,2) * PI; // Calculate area
        System.out.println("Program Number = " + modelNumber); // Print 1 variable
        System.out.println("radius = " + radius + ", length, area = " + circum + " , " +area); //Print 3S variables 
    }

    (a) Compile and execute area or Area.

    (b) Check that the program is running correctly by running a number of trial cases. Good input
datum for testing is $r = 1$, because then $A = \pi$, as well as $r = 10$.

    (c) Experiment with your program. For example, see what happens if you leave off decimal points, if
you feed in blanks, if you feed in a letter, and whatever you think may crash the program.

2. The programs given here take input from the keyboard (or from the code itself for Java) and place the
output onto the terminal screen. Revise your program so that its output is placed into a file. (You
can do this explicitly in the coding or with output redirection.) You may want to look at some sample
programs or Landau’s Handouts www.physics.orst.edu/ rubin/COURSES/Handouts/. For Java programmers with
some experience, we have a Java version of scanf and printf to install (see Web page); it’s takes some
work to get going, but is an excellent tool.

3. Revise your version of Area so that it uses a main program/method (which does the input and output)
and a function/method (that does the calculation). Check that the program still runs properly.
Table 1: The Java primitive data types and their precision and ranges for the IEEE 754 standard.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Bits</th>
<th>Bytes</th>
<th>Range &amp; Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>logical</td>
<td>1</td>
<td>-</td>
<td>true or false</td>
</tr>
<tr>
<td>char</td>
<td>string</td>
<td>16</td>
<td>2</td>
<td>'\u0000' ↔ '\uFFFF' (ISO Unicode characters)</td>
</tr>
<tr>
<td>byte</td>
<td>integer</td>
<td>8</td>
<td>1</td>
<td>−128 ↔ +127</td>
</tr>
<tr>
<td>short</td>
<td>integer</td>
<td>16</td>
<td>2</td>
<td>−32,768 ↔ +32,767</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
<td>32</td>
<td>4</td>
<td>−2,147,372,036,854,775,808 ↔ +2,147,483,648</td>
</tr>
<tr>
<td>long</td>
<td>integer</td>
<td>64</td>
<td>8</td>
<td>−9,223,372,036,854,775,808 ↔ +9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>float</td>
<td>floating point</td>
<td>32</td>
<td>4</td>
<td>±1.401298 × 10^{−45} ↔ ±3.402923 × 10^{+38}</td>
</tr>
<tr>
<td>double</td>
<td>floating point</td>
<td>64</td>
<td>8</td>
<td>±4.94065645841246544 × 10^{−324} ↔ ±1.7976931348623157 × 10^{+308}</td>
</tr>
</tbody>
</table>

**IEEE Floating Point Arithmetic**

Scientific computations mainly use floating-point numbers, and for reliability, they tend to use double-precision floating point numbers. The floating-point representation on computers is a binary version of what is commonly known as “scientific” or “engineering” notation. For example, in scientific and in engineering notation, respectively, the speed of light \( c \) is

\[
c = +2.99792458 \times 10^{+10} \text{cm/sec}, \quad \text{or} \quad c = +0.299792458 \times 10^{+11} \text{cm/sec}.
\]  

In both cases, the number out front, 2.99792458 or 0.299792458, is called the *mantissa* and contains nine *significant figures*. In engineering notation, the number of significant figures is the number of digits to the right of the *decimal point* (the 0 does not count), while in scientific notation, the nonzero digit in front of the decimal point is also significant. The power to which 10 is raised, in turn, is called the *exponent*. The explicit + sign before the mantissa and exponent is a reminder that these numbers may also be negative.

Floating point numbers are stored on a computer as a concatenation (juxtaposition) of the sign bit, the exponent and the mantissa. Yet the relation between what’s stored and the actual value of the number is somewhat indirect, with a number of different prescriptions for the relation used over the years. In fact, it used to be that each computer operating system and each computer language would define its own standards for floating point numbers. Different standards meant that the same program running correctly on different computers could give different results. While the results usually were only slightly different, the user could never be sure if the lack of reproducibility of a test case was due to the particular computer being used or to an error in the program’s implementation.

In 1987, the Institute of Electrical and Electronics Engineers (IEEE) and the American National Stan-
Table 2: Representation Scheme for IEEE Singles.

<table>
<thead>
<tr>
<th>Number Name</th>
<th>Values of s, e and f</th>
<th>Value of Single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0 &lt; e &lt; 255</td>
<td>((-1)^s \times 2^{e-127} \times 1.f)</td>
</tr>
<tr>
<td>Subnormal</td>
<td>e = 0, f ≠ 0</td>
<td>((-1)^s \times 2^{-126} \times 0.f)</td>
</tr>
<tr>
<td>Signed Zero</td>
<td>e = 0, f = 0</td>
<td>((-1)^s \times 0.0)</td>
</tr>
<tr>
<td>+∞</td>
<td>s = 0, e = 255, f = 0</td>
<td>+\text{INF}</td>
</tr>
<tr>
<td>−∞</td>
<td>s = 1, e = 255, f = 0</td>
<td>-\text{INF}</td>
</tr>
<tr>
<td>Not a Number</td>
<td>s = u, e = 255, f ≠ 0</td>
<td>NaN</td>
</tr>
</tbody>
</table>

When the standard is followed, you can expect the primitive data types to have the precision and ranges given in Table 1. In addition, when computers and software adhere to this standard, and most now do, you are guaranteed that your program will produce identical results on different computers. (However, because the IEEE standard may not produce the most efficient code or the highest accuracy for a particular computer, sometimes you may have to invoke compiler options to demand that IEEE standard is strictly followed for your test cases; after that you should run with greater efficiency and higher precision.)

There are actually a number parts to the IEEE standard, and some computer or chip manufacturers may adhere to only some of these parts. In the floating-point scheme, the number \(x\) is stored as three separate entities. There is the sign \(s\), the fractional part \(f\) of the mantissa, and an exponential field (or biased exponent) \(e\). All parts are stored in binary form and occupy different segments of a single 32-bit word or two adjacent 32-bit words (for singles and doubles). Normally, the floating point number is reconstituted as

\[
x_{\text{float}} = (-1)^s \times 1.f \times 2^{e - \text{bias}}. \tag{2}
\]

In the representation (2), the sign \(s\) is stored as a single bit \(s = 0, 1\) for positive, negative signs respectively. Normal numbers have \(0 < e < 255\), and for them the convention assumes that the mantissa’s first bit is a 1, and so only the fractional part \(f\) after the binary point is stored. (Subnormal numbers and special cases have \(e = 0\) or \(e = 255\), and their representations are given in Table 2.) This representations ensures that all normal floating-point numbers have the same relative precision, as well as giving an extra bit of precision. Accordingly, the first bit does not have to be stored and the computer only needs to recall that there is a phantom bit there. During the processing of numbers in a calculation, the first bit of an intermediate result may become zero, but this will be corrected before the final number is stored. In summary, for normal cases, the actual mantissa (1.f in binary notation) contains an implied 1 preceding a binary point.

Finally, in order to guarantee that the stored biased exponent \(e\) is always positive, a fixed number, called the bias is added to the actual exponent \(p\) before it is stored as the biased exponent \(e\). The actual exponent,
which of course can be negative, is

\[ p = e - \text{bias}. \quad (3) \]

**IEEE Single and Double Precision**

To be more specific about the actual storage of floating point numbers, we need to examine the two basic floating-point formats: *singles* and *doubles*. Singles or *floats* is shorthand for *single precision floating point numbers*, and doubles is shorthand for *double precision floating point numbers*. Singles occupy 32 bits overall, with one for the sign, eight bits for the exponent, and 23 bits for the fractional mantissa (which gives 24-bit precision when the phantom bit is included). Doubles occupy 64 bits overall, with one for the sign, 10 for the exponent, and 53 for the fractional mantissa (for 54-bit precision). This means that the exponents and mantissas for doubles are not simply double those of floats, as we see in Table 1. In addition, the IEEE standard also permits *extended precision* that goes beyond doubles, but we won’t get into that here.

To see how this scheme all works, let’s look at a 32 bit (4 B) float or single precision number:

<table>
<thead>
<tr>
<th>bit position</th>
<th>s</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>23</td>
<td>22</td>
</tr>
</tbody>
</table>

The sign bit \( s \) is in position 31, the biased exponent \( e \) in bits 30-23, and the fractional part of the mantissa \( f \) is in bits 0-22. Since eight bits are used to store the exponent \( e \) in (2) and \( 2^8 = 256 \), \( e \) has a range

\[ 0 \leq e \leq 256. \quad (4) \]

Since the

\[ \text{bias} = 127_{10}, \quad (5) \]

the actual exponent

\[ p = e_{10} - 127, \quad (6) \]

and the full exponent \( p \) for singles has the range

\[ -127 \leq p \leq 128, \quad (7) \]

as indicated in Table 1.

The mantissa \( f \) for singles is stored by the 23 bits in positions 0-22. For *normal numbers*, that is, numbers with \( 0 < e < 255 \), \( f \) is the fractional part of the mantissa. In other words, the actual number represented by these 32 bit is

\[ \text{normal floating point number} = (-1)^s \times 1.f \times 2^{e-127}. \quad (8) \]
For subnormal numbers, that is, numbers with $e = 0$, $f \neq 0$, $f$ is the entire mantissa. In other words, the actual number represented by these 32 bit is

\[ \text{subnormal numbers} = (-1)^s \times 0.f \times 2^{e-126}. \quad (9) \]

The 23 bits $m_0 - m_{22}$ used to store the mantissa of normal singles correspond to the representation

\[ \text{mantissa} = 1.f = 2 + m_0 \times 2^{-1} + m_1 \times 2^{-2} + \cdots + m_{22} \times 2^{-23}, \quad (10) \]

with $0.f$ used for subnormal numbers. (The special representations used to store $\pm 0, \pm \infty$ are given in Table 2.)

To see how this works in practise, the largest positive normal floating-point number possible for a 32-bit machine has the maximum value for $e$ (254), and the maximum value for $f$,

\[ f_{\text{max}} = 01111 1110 1111 1111 1111 1111 1111 111 = (0)(1111 1110)(1111 1111 1111 1111 1111 111), \quad (11) \]

where we have grouped the bits for clarity. This corresponds to

\[ s = 0, \quad (12) \]
\[ e = 1111 1110 = 254 \quad (13) \]
\[ \Rightarrow p = e - 127 = 127 \quad (14) \]
\[ f = 1.1111 1111 1111 1111 1111 1111 1111 111 = 1 + 0.5 + 0.25 + \ldots \simeq 2 \quad (15) \]
\[ \Rightarrow (-1)^s \times 1.f \times 2^{p-e-127} = 2 \times 2^{127} \simeq 3.4 \times 10^{38}. \quad (16) \]

This is the value shown in Table 1.

Likewise, smallest positive floating-point number possible is subnormal ($e = 0$) with a single significant bit in the mantissa:

\[ 0 0000 0000 0000 0000 0000 0000 0000 001. \]

This corresponds to

\[ s = 0, \quad (17) \]
\[ e = 0 \quad (18) \]
\[ \Rightarrow p = e - 126 = -126 \quad (19) \]
\[ f = 0.0000 0000 0000 0000 0000 0000 0000 001 = 2^{-23} \quad (20) \]
\[ \Rightarrow (-1)^s \times 0.f \times 2^{p-e-126} = 2^{-149} \simeq 1.4 \times 10^{-45}. \quad (21) \]

In summary, single-precision (32 bit or 4-byte) numbers have 6-7 decimal places of precision (significance) and magnitudes in the range

\[ 1.4 \times 10^{-45} \leq \text{single precision} \leq 3.4 \times 10^{38}. \quad (22) \]
Table 3: The IEEE single precision standard for special cases.

<table>
<thead>
<tr>
<th>Number</th>
<th>s</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>0</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>-0</td>
<td>1</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>+∞</td>
<td>0</td>
<td>1111 1111</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>-∞</td>
<td>1</td>
<td>1111 1111</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

Table 4: Representation Scheme for IEEE Doubles.

<table>
<thead>
<tr>
<th>Number Name</th>
<th>Values of s, e and f</th>
<th>Value of Single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>$0 &lt; e &lt; 2047$</td>
<td>$(-1)^s \times 2^{e-1023} \times 1.f$</td>
</tr>
<tr>
<td>Subnormal</td>
<td>$e = 0$, $f \neq 0$</td>
<td>$(-1)^s \times 2^{-1022} \times 0.f$</td>
</tr>
<tr>
<td>Signed Zero</td>
<td>$e = 0$, $f = 0$</td>
<td>$(-1)^s \times 0.0$</td>
</tr>
<tr>
<td>+∞</td>
<td>$s = 0$, $e = 2047$, $f = 0$</td>
<td>INF</td>
</tr>
<tr>
<td>−∞</td>
<td>$s = 1$, $e = 2047$, $f = 0$</td>
<td>−INF</td>
</tr>
<tr>
<td>Not a Number</td>
<td>$s = u$, $e = 2047$, $f \neq 0$</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Specific values are given in Table 1 and the special cases ±0 and ±∞ in Table 3.

Doubles are stored as two 32–bit words, for a total of 64 bits (8B) overall. The sign occupies one bit, the biased exponent $e$ 11 bits, and the fractional mantissa, 52 bits:

<table>
<thead>
<tr>
<th>bit position</th>
<th>s</th>
<th>e</th>
<th>f</th>
<th>f (cont)</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>62</td>
<td>52</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

As we see here, the fields are stored contiguously, with part of the mantissa $f$ stored in a separate 32-bit word. The order of these words, and whether the second word with $f$ is the most–, or least–significant part of the mantissa, is machine dependent. For doubles, the bias is quite a bit larger than for singles,

$$bias = 111111111111_2 = 1023_{10}$$

so the actual exponent

$$p = e - 1023.$$  

The bit-patterns for doubles is given in Table 4, and the range and precision are given in Table 1. To summarize, if you write a program with doubles, then 64-bit (8-byte) will be used to store your floating point numbers. You will have about 16 decimal places of precision (1 part in $2^{52}$) and magnitudes in the range

$$4.9 \times 10^{-324} \leq \text{double precision} \leq 1.8 \times 10^{308}.$$
1. Consider the 32-bit single-precision floating point number

<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>23</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>1110 1010 0000 0000 0000 0000 000</td>
</tr>
</tbody>
</table>

(a) What are the (binary) values for the sign $s$, the biased exponent $e$, and the fractional mantissa $f$. *(hint: $e_{10} = 13$).*
(b) What is the decimal value for the biased exponent $e$ and the actual exponent $p$?
(c) Show that the mantissa of $A$ equals 1.625 000.
(d) Show that $A \approx 1.2344 \times 10^{-64}$.

2.13:

2. Check where under- and overflow occur for single-precision floating-point numbers (floats).
3. Check where under- and overflow occur for double-precision floating-point numbers (doubles).
4. Check where under- and overflow occur for integers. Note, there is no exponent stored for integers, so the smallest integer corresponds to the most negative one. To determine the largest and smallest integer, you need to observe the results as you explicitly pass through the limits by continually adding and subtracting 1. (Integer arithmetic uses *two’s compliment* arithmetic, so you should expect some surprises.)

2.15:

1. Check the machine precision (number of places in mantissa) for single-precision floating-point numbers (floats).
2. Check the machine precision for double-precision floating-point numbers (doubles).

Week 2 Reading 3,

3.14:
The mathematical definition of the exponential function $e^{-x}$ is

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!}.$$  \hspace{1cm} (26)

As long as $x^2 < \infty$ this series converges in the mathematical sense. Accordingly, a good guess for a direct algorithm to compute the exponential would be

$$e^{-x} \approx \sum_{n=0}^{N} \frac{(-x)^n}{n!}.$$ \hspace{1cm} (27)
Clearly for (27) to be a good algorithm, we need to have the first ignored term \((-x)^{N+1}/(N+1)!\) to be small as compared to the sum we are keeping. This is the desired regime in which the series is converging. However, we also need to have the sum of all the ignored term to be small as compared to the sum we are keeping and the sum to be computed accurately.

While, in principle, it should be faster to see the effects of error accumulation in this algorithm by using single precision numbers (floats) in your programming, C and Java tend to use double-precision mathematical libraries and so it is hard to do a pure single-precision computation. Accordingly, do these exercises in double precision, even if the text suggest single precision.

1. Write a program that calculates \(e^{-x}\) as the finite sum (27).

2. Calculate your series for \(x \leq 1\) and compare it to the built-in function \(\exp(x)\) (you may assume that the built-in exponential function is exact). You should pick an \(N\) for which
\[
\frac{(-x)^{N+1}}{(N+1)!} \leq 10^{-7} \sum_{n=0}^{N} \frac{(-x)^n}{n!}.
\]

3. Examine the terms in the series for \(x \approx 10\) and observe the significant subtractive cancellations that occur when large terms add together to give small answers. In particular, print out the near-perfect cancellation at \(n \approx x - 1\).

4. See if better precision is obtained by being clever and using \(\exp(-x) = 1/\exp(x)\) for large \(x\) values. This eliminates subtractive cancellation, but not other types of round-off error.

5. By progressively increasing \(x\) from 1 to 10 and then from 10 to 100, use your program to determine experimentally when the series starts to lose accuracy, and when the series no longer converges.

6. Make a series of graphs of the error vs \(N\) for different values of \(x\). We recommend \(xmgr\) or \(gnuplot\) on unix and \(gnuplot\) on windows (see our Web links). (Note that \(xmgr\) has a built-in tutorial under the Help menu.)

**Week 3 Reading 4,**

4.8: Compute the relative error
\[
\epsilon = \left| \frac{\text{numeric-exact}}{\text{exact}} \right|,
\]
for the trapezoid rule, Simpson’s rule, and Gaussian quadrature.

1. Print out your data directly into the following tabular form, or use an editor to place it into it:

<table>
<thead>
<tr>
<th>(N)</th>
<th>(\epsilon_T)</th>
<th>(\epsilon_S)</th>
<th>(\epsilon_G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
Note that there are spaces or tabs separating the fields. Try $N$ values of 2, 10, 20, 40, 80, 160, ....  
(*Hint: These are even numbers, which may not be the assumption of every rule.*)

2. Make a plot similar to Fig.4.3 of $\log_{10} \epsilon$ versus $\log_{10} N$. The ordinate will then tell you immediately the number of number of decimal places of precision obtained for each value of $N$.

3. Use your plot or table to estimate the power-law dependence of the error $\epsilon$ on the number of points $N$. Do this for both the trapezoid and Simpson rules and for both the algorithmic and round-off error regimes. (Note that it may be hard to reach the roundoff error regime for the trapezoid rule because the approximation error is so large.)

**Week 4  Reading 18, 19**

Even though the materials in Chapters 18 and 19 of Landau & Páez were written with Fortran and C in mind, they are also relevant for Java. Java is a good language for scientific programming and its code is much more portable and universal than Fortran and C. However, at present compiled Java code runs slower than Fortran or C code. In part, this is a consequence of the Fortran and C compilers having been around longer and thereby having been better refined to get the most out of a computer’s hardware, and, in part, this is also a consequence of Java being designed for portability and not speed. Since modern computers are so fast, whether a program takes one second or three seconds usually doesn’t matter much, especially in comparison to the hours of your time that it might take to modify a program for different computers. However, you may want to convert the code to C if you are running a computation that takes hours or days to complete and will be doing it many times.

Especially when asked to, good Fortran and C compilers will look at your entire code as a single entity and rewrite it for you so that it runs faster. (The rewriting is at a fairly basic level, so there’s not much use in your studying the compiler’s output as a way of improving your programming skills.) In particular, Fortran and C compilers are very careful with how large arrays are stored in different parts of the computer’s memory. They also are careful with how numbers are fed into cache so as not keep the CPU waiting around for numbers to crunch. There is no fundamental reason why a program written in Java cannot be compiled to produce an equally efficient code, and indeed such compilers are now being developed and becoming available. However, such optimized code is not portable, but instead runs on the single computer architecture for which it was compiled.

As we have said, the source code in a file such as `Prog.java` gets compiled into the byte code present in the file `Prog.class`. This class file can be interpreted line by line by another program known as the Java Virtual Machine. When you change from a Unix to a Windows computer, for example, the Java Virtual Machine program changes, but the byte code that runs via the Virtual Machine stays the same. This is the essence of Java's portability.

In order to improve the performance of Java, many computers and browsers now run a Just-In-Time
(JIT) Java compiler. If a JIT is present, the Java Virtual Machine feeds your byte code `Prog.class` to the JIT so that it can be recompiled into native code explicitly tailored to the machine on which you are running. Although there is an extra step involved here, the total time it takes to run your program is usually 10-30 times faster with the JIT as compared to line-by-line interpretation. Because the JIT is an integral part of the Java Virtual Machine on each operating system, this usually happens automatically.

In the exercises below we will investigate techniques to optimize both a Fortran and a Java program. In the process we shall compare the speeds of Fortran versus Java for the same computation. If you run your Java code on a variety of machines (easy to do with Java), you should also be able to compare the speed of one computer to another. Note, you can do this exercise even if you do not know Fortran.

The type of optimization often associated with “High Performance” or “Numerically Intensive” computing (and discussed in Landau and Páez) is one in which sections of code are rewritten and reorganized in order to decrease execution time. The overall value of doing this, especially as computers have become so fast and so available, is often a subject of controversy between computer scientists and computational scientists. Both camps would agree that using the optimization options of the compilers is a good idea, while computational scientists tend to believe that you can’t rely on the compiler to do it all.

**Collected Expert Opinions on Optimization**

**Rules of Optimization:**

**Rule 1:** Don't do it.

**Rule 2 (for experts only):** Don't do it yet. - M.A. Jackson

“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.” - W.A. Wulf

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” - Donald Knuth

“The best is the enemy of the good.” - Voltaire

**Don’t optimize as you go:** Write your program without regard to possible optimizations, concentrating instead on making sure that the code is clean, correct, and understandable. If it’s too big or too slow when you’ve finished, then you can consider optimizing it.

**Remember the 80/20 rule:** In many fields you can get 80% of the result with 20% of the effort (also called the 90/10 rule - it depends on who you talk to). Whenever you’re about to optimize code, use profiling to find out where that 80% of execution time is going, so you know where to concentrate your effort.

**Always run “before” and “after” benchmarks:** How else will you know that your optimizations actually made a difference? If your optimized code turns out to be only slightly faster or smaller than the original version, undo your changes and go back to the original, clear code.

**Use the right algorithms and data structures:** Don’t use an $O(n^2)$ bubblesort algorithm to sort a thousand elements when there’s an $O(n \log n)$ quicksort available. Similarly, don’t store a thousand items in an array that requires an $O(n)$ search when you could use an $O(\log n)$ binary tree, or an $O(1)$ Java hash table.

---

Week 4 Problem

The various versions of the program **tune** solve the matrix eigenvalue equation

\[ [H]|c \rangle = E|c \rangle \]  

for the eigenvalues \( E \) and eigenvectors \( |c \rangle \) of a Hamiltonian matrix \( [H] \). Here the individual Hamiltonian matrix elements are assigned the values

\[ H_{i,j} = \begin{cases} i, & \text{for } i = j, \\ 0.3|i-j|, & \text{for } i \neq j \end{cases} \]

\[ [H] = \begin{bmatrix} 1 & 0.3 & 0.09 & 0.027 & \ldots \\ 0.3 & 2 & 0.3 & 0.9 & \ldots \\ 0.09 & 0.3 & 3 & 0.3 & \ldots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \]  

This means that the Hamiltonian is almost diagonal, so the eigenvalues should be easy to find. For the present problem, the \( H \) matrix has dimension \( N \times N \approx 2000 \times 2000 = 4,000,000 \). With so many elements, matrix manipulations should take enough time so that we can see the effects of optimization.

The solution to (30) is found via a variation of the power method. We start off with an arbitrary first guess for the eigenvector \( |c \rangle \) which we represent as the column vector:

\[ |c_0 \rangle \simeq \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \]

Note that since the Hamiltonian matrix \( [H] \) is nearly diagonal and with \( H_{1,1} \) the smallest diagonal element, then this guess should be close to the eigenvector with the smallest eigenvalue. We next calculate the energy corresponding to this eigenvector as

\[ E \simeq \frac{\langle c_0 | H | c_0 \rangle}{\langle c_0 | c_0 \rangle}, \]

where \( \langle c_0 | \) is the row vector representation of \( |c_0 \rangle \). We then guess that an improved eigenvector has the \( k^{th} \) component

\[ |c_1 \rangle_k \simeq |c_0 \rangle_k + \frac{(H-E)|c_0 \rangle_k}{E - H_{k,k}}, \]

where \( k \) ranges over the length of the eigenvector. If repeated, this method converges to the eigenvector with the smallest eigenvalue since it gets the largest weight (smallest denominator) each time. For the present case, six places of precision in the eigenvalue is usually obtained after 11 iterations.

1. Vary the precision variable and note how it affects the number of iterations required.

2. Try a variation in the eigenvector guess algorithm (35) so that the method converges to some of the other eigenvectors.
3. Keep a table like Table 19.1 in the *Computational Physics* text of your execution times versus technique.

4. On page 264 of the text and on the Web, you will find the source code for the Fortran program `tune.f`. Compile and execute `tune.f`, recording the time it takes to run. On Unix systems, the compiled program will be placed in the file `a.out`. From a Unix shell, the compilation, timing and execution can all be done with the commands:

```
> f77 tune.f
> time a.out
```

Here the compiled Fortran program is given the (default) name `a.out` and the `time` command gives you the execution (user) time and system time in seconds to execute `a.out`.

5. Even if you do not know (or care) what’s going on inside of a program, compilers are smart and caring enough to figure it out for you and to rewrite your program as needed. You control how completely the compiler does this when you compile optimization options, for example,

```
> f77 -O tune.f
```

where the `-O` turns on optimization. Now run and time the optimized code, checking that it still gives the same answer, and note the speed up.

6. Actually, there are usually a number of levels of optimization that the compiler can attempt, with each level giving the possibility of a greater speedup as well as a concordant greater risk of being wrong. With the Forte compilers on the Suns, the possible options are:

```
-0: Use default optimization level (-O3)
-01: Provide minimum statement-level optimizations
-02: Enable basic block-level optimizations
-03: Add loop unrolling and global optimizations
-04: Add automatic inlining of routines from same source file
-05: Attempt aggressive optimizations (with profile feedback)
```

Note that the `-O` option is equivalent to `-O3`. Try out `-O3` and `-O4` and note the time and accuracy. Usually O3 is pretty good, especially for as simple a program as `tune` (there is only a main program/method). Since we have only one subroutine/method in our program, we would not expect `-O4` or `-O5` to be an improvement over `-O3`. However, we do expect `-O3`, with its loop unrolling, to be an improvement over `-O2`.

7. As explained in the text, `tune4` does some loop unrolling (we will explore that soon). To see the best we can do with Fortran, record the time for the most optimized version of `tune4.f`. 

13
8. The program `Tune.java` is the Java equivalent of the Fortran program `tune.f`. This is what it looks like:

```java
/** Tune.java: matrix program to be tuned for performance */
public class Tune {
    public static void main(String[] argv) {
        final int Ldim = 2051;
        int i, j, iter = 0;
        double [][] ham = new double [Ldim] [Ldim];
        double [] coef = new double [Ldim];
        double [] sigma = new double [Ldim];
        double err, ener, ovlp, step=0.,time;
        // Store initial time
        time = System.currentTimeMillis();
        // set up Hamiltonian & starting vector
        for ( i = 1; i <= Ldim-1; i++) {
            for ( j=1; j <= Ldim-1; j++) {
                if (Math.abs(j-i) >10) {ham[j][i] = 0.0;}
                else ham[j][i] = Math.pow(0.3, Math.abs(j-i));
            }
            ham[i][i] = i ; // Diagonal H elements
            coef[i] = 0.0 ; // Eigenvector elements
        }
        // start iterating towards the solution
        coef[1] = 1.0 ; // Guess 1st eigenvector = 1, 0, ...0
        err = 1.0 ; // Arbitrary err to begin
        // iterate until converge or 15 tries
        while (iter <15 && err > 1.0e-6) {
            iter = iter + 1 ;
            // compute current energy \& norm, \& normalize
            ener = 0.0 ;
            ovlp = 0.0 ;
            for ( i= 1; i <= Ldim-1; i++) // outer loop
            {
                ovlp = ovlp + coef[i]*coef[i] ;
                sigma[i] = 0.0;
                for ( j= 1; j <= Ldim-1; j++) // inner loop
                {
                    sigma[i] = sigma[i] + coef[j]*ham[j][i];
                    ener = ener + coef[i]*sigma[i] ;
                }
            }
        }
    }
}
```
for ( i = 1; i<= Ldim-1; i++ ) //Normalize 
{
    coef[i] = coef[i]/Math.sqrt(ovlp) ;
    sigma[i] = sigma[i]/Math.sqrt(ovlp) ;
}

// compute update and error
err = 0.0 ;
for ( i = 2; i <= Ldim-1; i++) 
{
    step = (sigma[i] - ener*coef[i])/(ener-ham[i][i]) ;
    coef[i] = coef[i] + step ;
    err = err + Math.pow(step,2) ;
}
err = Math.sqrt(err) ;
System.out.println ("iter, ener, err = " + iter + ", " + ener + ", " + err);

// output elapsed time
time = (System.currentTimeMillis() - time)/1000;
System.out.println("time= " + time + "s");
}

9. To get an idea of what Tune.java does (and give you a feel for how hard life is for the poor computer), assume ldim =3 and run through one iteration of Tune4 by hand. Assume that the program has converged, and follow the code to completion. Write down and hand in the different assignment statements actually executed by the computer.

10. Compile and execute Tune.java. You do not have to issue the time command since we built a timer into the Java program (however there is no harm in trying it). Check that you still get the same answer as you did with Fortran, and note how much longer it takes with Java.

11. Try the -O option with the Java compiler and note if the speed changes (since this just inlines methods, it should not affect our one-method program).

12. You might be surprised how much slower is Java than Fortran and that the Java optimizer doesn't seem to do much good. To see what the actual Java byte code does, invoke the Java profiler with the command

> javap -c Tune

This should produce a file, java.prof for you to look at with an editor. Look at it and see if you agree with us that scientists have better things to do with their time than understand such files!
13. We now want to perform a little experiment in which we see what happens to performance as we fill up the computer’s memory. In order for this experiment to be reliable, it is best for you to not to be sharing the computer with any other users. On Unix systems, the `who -a` command will show you the other users (we leave it up to you to figure out how to negotiate with them).

14. To get some idea of what aspect of our little program is making it so slow, compile and run `Tune.java` for the series of matrix sizes $ldim = 10, 100, 250, 500, 750, 1025, 2500,$ and $3000$. You may get an error message that Java is out of memory at 3000. This is because you have not turned on the use of virtual memory. In Java, the memory allocation pool for your program is called the heap and it is controlled by the -Xms and -Xmx options to the Java interpreter java:
   - `-Xms256m` Set initial heap size to 256 Mbytes
   - `-Xmx512m` Set maximum heap size to 512 Mbytes

15. Make a graph of the run time versus matrix size. It should be similar to Fig. 1, although if there are more than one user on your computer while you run, you may get erratic results. You will note that as our matrix gets larger than $\sim 1000 \times 1000$ in size, the curve has a sharp increase in slope with execution time, in our case increasing like the third power of the dimension. Since the number of elements to compute increases like the second power of the dimension, something else is happening here. It is a good guess that the additional slowdown is due to page faults in accessing memory. In particular, accessing two dimensional arrays, with their elements scattered all through memory, can be very slow.
16. Repeat the previous experiment that gauges the effect of increasing the ham matrix size, only now for tune.f for matrix size \(ldim = 10, 100, 250, 500, 1025, 3000, 4000, 6000, \ldots\). You should get a graph like ours. Although our implementation of Fortran has automatic virtual memory, its use will be exceedingly slow, especially for our problem (like a 50-fold increase in time!). So, if you submit your program and you get nothing on the screen (but the disk may sound or flash busy), then you are probably in the virtual memory regime. If you can, let the program run for one iteration and then kill it (\texttt{control+c} or \texttt{kill -9 pid}). You can then multiply that time by 11 to approximate the time it would have taken for a full computation.

17. To test our hypothesis that the access of the elements in our two-dimensional array \(\text{ham}[i][j]\) is slowing us down, we have modified Tune.java into Tune4.java (this corresponds to tune.f in the text):

```java
/** Tune4.java: matrix program with loop unrolling */
public class Tune4
{
    public static void main(String[] argv)
    {
        final int Ldim = 2051;
        int i, j, iter = 0;
        double [][] ham = new double [Ldim] [Ldim];
        double [] diag = new double [Ldim];
        double [] coef = new double [Ldim];
        double [] sigma = new double [Ldim];
        double err, ener, ovlp, ovlp1, ovlp2, step=0., fact, time, t,t1,t2,u;
        // Store initial time
        time = System.currentTimeMillis();
        // set up non-diagonal Hamiltonian
        for ( i = 1; i < Ldim-1; i++)
        {
            for ( j=1; j < Ldim-1; j++)
            {
                if (Math.abs(j-i) >10) {ham[j][i] = 0.0;}
                else ham[j][i] = Math.pow(0.3, Math.abs(j-i));
            }
        }
        // set up diagonal Hamiltonian and starting vector
        for ( i=1; i < Ldim-1; i++)
        {
            ham[i][i] = i ;
            coef[i] = 0.0 ;
            diag[i] = ham [i][i];
            coef[1] = 1.0 ;
        }
```
err = 1.0;
// start iterating towards the solution
iter = 0;
while (iter < 15 && err > 1.0e-6)
{
  iter = iter + 1;
  // compute current energy & norm, & normalize
  ener = 0.0;
  ovlp1 = 0.0;
  ovlp2 = 0.0;
  // NB loop partially unrolled so steps of 2
  for (i = 1; i <= Ldim-2; i = i+2) // outer loop
  {
    ovlp1 = ovlp1 + coef[i]*coef[i];
    ovlp2 = ovlp2 + coef[i+1]*coef[i+1];
    t1 = 0.0;
    t2 = 0.0;
    for (j = 1; j <= Ldim-1; j++) // inner loop begins
    {
      t1 = t1 + coef[j]*ham[j][i];
      t2 = t2 + coef[j]*ham[j][i+1];
    }
    sigma[i] = t1;
    sigma[i+1] = t2;
    ener = ener + coef[i]*t1 + coef[i+1]*t2;
  }
  ovlp = ovlp1 + ovlp2;
  ener = ener / ovlp;
  fact = 1.0 / Math.sqrt(ovlp);
  coef[1] = fact * coef[1];
  err = 0.0;
  // compute update and error norm
  for (i = 2; i <= Ldim-1; i++)
  {
    t = fact * coef[i];
    u = fact * sigma[i] - ener * t;
    step = u / (ener - diag[i]);
    coef[i] = t + step;
    err += step * step;
  }
  err = Math.sqrt(err);
  System.out.println("iter, ener, err " + iter + ", " + ener + ", " + err);
}
// output elapsed time
time = (System.currentTimeMillis() - time) / 1000;
System.out.println("time= "+ time + "s");
18. Compile and execute Tune4.java. Look at Tune4.java and note where the nested for loop over i and j now takes step of $\Delta i = 2$ rather the unit steps in Tune.java. If thing work as expected, the better memory access of Tune4.java should cut the runtime nearly in half. Record your answer in your table.

19. In order to have half as many calls to the two-dimensional array as before, we employed a technique know as loop unrolling in which we explicitly wrote out some of the lines of code which would otherwise be executed implicitly as the for loop went through all the values of its indices. This is not as clear a piece of code as before, but, evidently, it permits the compiler to speed up the code. To check that Tune and Tune4 actually do the same thing, assume $ldim = 3$ and run through one iteration of Tune4.java by hand. Hand in your output.

20. There are often lots of little changes people make to (try to) make programs run faster. Some of these are hand-me-down techniques from the times when people actually paid for every millisecond of computer time, and some, like the loop unrolling we just did, can have significant effects. The hard question is whether the investment in your time and the less-clear coding is worth the savings of computer time. To give you a feel of the changes made and the resulting speedups, note the effect on the speed of replacing the “expensive” (time consuming) square root function $\text{Math.sqrt(ovlp)}$ by just $ovlp$ (of course the answers will differ).

Week 5  Reading 15.1-15.7

Root Finding by Trial and Error

Many computer techniques, such as multiplying matrices, are accomplished directly according to a set procedure. It may get repeated until all elements in the matrix are computed, but we can say ahead of time exactly how many steps will be involved in the process. In contrast, other computer techniques employ a “trial and error” algorithm in which the program goes through some numerical procedure, and then tests to see if the result is acceptable. (We already did some of this when we summed a power series and quit summing when either the additional terms were small compared to the total sum, or when we though we had summed enough.) This type of program is usually interesting to write as we have to think hard in order to make the computer act “intelligently” for all possible cases. These “trial and error” programs are also interesting to run because, like experiments, they depend very much on the initial conditions, and it is hard to predict exactly what the computer will come up as it searches for a solution.
Figure 2: A graphical representation of the steps involved in solving for a zero of $f(x)$ using the bisection algorithm (left), and the Newton-Raphson method (right). The bisection algorithm takes the midpoint of the interval as the new guess for $x$, and so each step reduces the interval size by one half. The Newton-Raphson method takes the new guess as the zero of the line tangent to $f(x)$ at the old guess. Four steps are shown for the bisection algorithm, but only two for the more-rapidly-convergent Newton-Raphson method.

The “trial and error” technique for root finding looks for a solution $x$ of the equation

$$f(x) = 0. \quad (36)$$

Having the RHS of the equation $= 0$ is just a convention. Any equation, such as $10 \sin x = 3x^3$, can easily be converted to this form; for example, $10 \sin x - 3x^3 = 0$. The general procedure is one in which we start with a guess value for $x$, substitute that guess into $f(x)$ (the “trial”), and then see how far your guess is from a true solution (the “error”). You then change $x$ appropriately (a new guess) and try it out in $f(x)$. The procedure continues until $f(x) \simeq 0$ to some desired level of precision, or until the changes in $x$ are insignificant. (As a safeguard, you may also set a maximum value for the number of trials.)

### Bisection Algorithm

The most elementary trial-and-error technique is the bisection algorithm. It is quite reliable, but slow. In other words, if you start with some interval in which $f(x)$ changes sign, then the bisection algorithm will always converge to the root, but it will take many, progressively smaller, steps to get there. Other techniques, such as the Newton-Raphson method we describe next, may converge much more quickly and with bigger steps, but if the initial guess is not close enough to the zero, these more intelligent techniques may become unstable and fail to find any root.
The basis behind the bisection algorithm can be seen on the left of Fig. 2. We start with two values of \( x \), between which we know a zero occurs. (You can determine these by making a graph or by stepping through different \( x \) values and calculating the value of \( f(x) \) for each.) To be specific, let’s say that \( f(x) \) is negative at \( x_- \) and positive at \( x_+ \):

\[
f(x_-) < 0, \quad f(x_+) > 0.
\] (37)

Note that it may well be that \( x_- > x_+ \) if the function changes from positive to negative as \( x \) increases. So we start with an interval of length \( |x_+ - x_-| \) within which we know a zero occurs. The bisection algorithm then bisects this interval and tests to see in which half interval the sign change occurs. It then uses that new half-interval as the interval and continues.

Explicitly, we calculate the midpoint of the interval

\[
x = \frac{x_+ + x_-}{2},
\] (38)

and then check if there is a sign change in the right half of the interval. If there is, then we move our search over the right half-interval; if there is not, then we move our search over to the left half-interval. Explicitly in pseudocode:

```java
if ( f(x)f(x+) > 0 ) x+ = x
else x- = x
```

The process continues until the value of \( f(x) \) is less than some constant you have defined as the level of precision. To be safe, you may want to check that there is a sign change and stop the process if the \( x \) values stop changing or after some large number of steps.

The example on the left of Fig. 2 shows the first interval extending from \( x_- = x_{+1} \) to \( x_+ = x_{-1} \). We then bisect that interval at \( x \), and since \( f(x) < 0 \) at the midpoint, we set \( x_- = x_{-2} = x \) and label it as \( x_{-2} \) to indicate the second step. We then use \( x_{+2} = x_{+1} \) and \( x_{-2} \) as the next interval and continue the process. We see in the figure that only \( x_- \) changes for the first three steps in this example, but that for the fourth step, \( x_+ \) finally changes. The changes then get too small for us to show.

Although it is easy enough to write your own version, the program `Bisection.java` on the Web is there to help get you started. You may want to modify it to check that each range of \( x \) values contains a sign change, and to input \( x_- \) and \( x_+ \) values from the keyboard as a way of searching for an initial interval containing a sign change.

1. To get a feel for the use of this technique, try out `Bisection.java` and then apply it to solve for some bound states of a quantum square well. As discussed in introductory quantum mechanics texts, the bound state energies for a square well are solutions of an equation of the form

\[
\sqrt{10 - E} \tan(\sqrt{10 - E}) = \sqrt{E},
\] (39)

were we have chosen the constants to make the equation simple.
2. As a first step in finding the numerical solution, make a plot of

\[ f(E) = \sqrt{10 - E} \tan(\sqrt{10 - E}) - \sqrt{E} \tag{40} \]

versus \( E \). Note from your plot some approximate \( E \) values at which \( f(E) = 0 \). You will need to find more exact values for these zeros.

3. \textit{Warning:} since the tan function has singularities, you have to be careful. In fact, your graphics program (or Maple) may not function very accurately near these singularities. Accordingly show that an equivalent form of (40) is

\[ \sqrt{E} \cot(\sqrt{10 - E}) = \sqrt{10 - E}. \tag{41} \]

Make a second plot of this function (which also has singularities, but at different places). Note if the zeros are in the same places as before, and, if not, pick out the values of the more accurate values.

4. Feel free to see what kind of roots \textit{Maple} or \textit{Mathematica} finds for equations (40) and (41).

5. Use the bisection algorithm to find some energies \( E \) that are solutions of both (40) and (41). Comment on the agreement between the two.

\textbf{Newton-Raphson}

The Newton-Raphson technique is an algorithm for solving

\[ f(x) = 0, \tag{42} \]

Figure 3: Two examples of how the Newton-Raphson method may fail if the initial guess is not close enough to the zero for \( f(x) \) to be well approximated by a straight line. In the top figure, the second guess for \( x \) occurs at a zero of the derivative, and consequently predicts \( x = \infty \) and the next guess. In the bottom figure, the search has fallen into an infinite loop.
more quickly than the bisection algorithm. As we see graphically on the right of Fig. 2, the Newton-Raphson algorithm is the equivalent of drawing a straight line tangent to the curve \( f(x) \) at some guessed value of \( x \), and then using the intercept of the line with the \( x \) axis as the next guess. If the “curve” were a straight line, the answer would be exact, otherwise it is a good approximation when \( f(x) \) is nearly linear. The process continues until some set level of precision is reached. As we see in Fig. 2, if a guess is in a region where \( f(x) \) is nearly linear, then the convergence is much more rapid than the bisection algorithm.

Two examples of the problem with Newton-Raphson are shown in Fig. 3. In the top we see a case where the search takes us to a point on \( f(x) \) where \( df(x)/dx = 0 \), that is to a horizontal tangent. The next guess is \( x = \infty \), which is not very helpful. In the bottom of Fig. 3 we see a case where the search gets into an infinite loop around the solution, but never gets there.

The problem in both these cases is that the initial guess was not close enough to the region where \( f(x) \) is approximately linear. So again, a good plot would have helped produce a good guess. Alternatively, some search routines start with the bisection algorithm in order to get close to the linear region, and then switch to Newton-Raphson to zero in on the root.

The analytic form of Newton Raphson is equivalent to expanding \( f(x) \) in a Taylor series around the old guess and keeping only the linear term. Let:

\[
\begin{align*}
x_0 &= \text{old guess}, \\
x &= x_0 + \Delta x = \text{new guess}, \\
\Delta x &= \text{correction},
\end{align*}
\]

where the correction \( \Delta x \) is unknown. The major assumption is that we are in the linear regime, that is, that the Taylor expansion is accurate with just two terms:

\[
f(x = x_0 + \Delta x) \simeq f(x_0) + \frac{df}{dx}(x_0)\Delta x.
\]

The next guess is the \( x \) value at which the straight line crosses the \( x \) axis, that is, where

\[
f(x_0) + \frac{df}{dx}(x_0)\Delta x \simeq 0,
\]

\[
\Rightarrow \Delta x = -\frac{f(x_0)}{\frac{df}{dx}(x_0)}.
\]

The procedure is repeated starting at the improved \( x \) until some set level of precision is obtained.

The Newton-Raphson algorithm (48) evidently requires evaluation of the derivative \( df/dx \) at each value of \( x \). In many cases you may have an analytic expression for the derivative and can built that into
the algorithm. However, since derivatives are simple to evaluate on the computer using the forward
difference or central difference algorithms, it most cases it’s less error prone to evaluate (48) using
the approximate derivative. Accordingly, we give NewtonRHL_cd.java and NewtonRHL_fd.java on the
Web with the two different approximations for the derivatives. (Central difference is more accurate,
but forward difference is simpler, and once you find a zero it doesn’t matter how you got there.)

6. Use the Newton-Raphson algorithm to find some energies \( E \) that are solutions of (40) and (41). Com-
ment.

7. The "10" in these equations is proportional to the strength of the potential that causes the binding.
See if making the potential deeper, say by changing the 10 to a 20 or a 30, produces more or deeper
bound states.

Matrix Problems

8. (15.11) Find the inverse of

\[
A = \begin{pmatrix}
4 & -2 & 1 \\
3 & 6 & -4 \\
2 & 1 & 8
\end{pmatrix}.
\]

(a) As a general procedure, applicable even if you do not know the analytic answer, check your inverse
in both directions, that is, check that

\[
AA^{-1} = A^{-1}A = 1.
\]

(b) Verify that your numerical answer agrees with exact one:

\[
A^{-1} = \frac{1}{263} \begin{pmatrix}
52 & 17 & 2 \\
-32 & 30 & 19 \\
-9 & -8 & 30
\end{pmatrix}.
\]

9. Consider the same matrix \( A \) as in (49), now used to describe three simultaneous linear equations,

\[
AX = B,
\]

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{pmatrix} \begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix} = \begin{pmatrix}
b_1 \\
b_2 \\
b_3
\end{pmatrix}.
\]

Here the vector \( B \) on the RHS is assumed known, and the problem is to solve for the vector \( X \). Use an
appropriate subroutine to solve these equations for the three different \( X \) vectors appropriate to these
three different \( B \) values on the RHS:

\[
B_1 = \begin{pmatrix}
+12 \\
-25 \\
+32
\end{pmatrix}, \quad B_2 = \begin{pmatrix}
+4 \\
-10 \\
+22
\end{pmatrix}, \quad B_3 = \begin{pmatrix}
+20 \\
-30 \\
+40
\end{pmatrix}.
\]
Figure 4: Two masses connected by four pieces of string, and suspended from a horizontal bar of length $L$. The four angles and the tensions in the strings are unknown.

The solutions should be

$$
X_1 = \begin{pmatrix}
+1 \\
-2 \\
+4
\end{pmatrix}, \quad X_2 = \begin{pmatrix}
+0.312 \\
-0.038 \\
+2.677
\end{pmatrix}, \quad X_3 = \begin{pmatrix}
+2.319 \\
-2.965 \\
+4.790
\end{pmatrix}.
$$

10. **15.4:** Two spheres are hung from three pieces of string as shown in Fig. 4. The weights of the spheres and the lengths of the strings are given: $(W_1, W_2) = (10, 20) \text{ Nt}$, $(L, L_1, L_2, L_3) = (8, 3, 4, 4) \text{ cm}$. The **problem** is to find the angles that the string make and the tensions in the strings when the system is in equilibrium. While this is a simple problem requiring no more than first-year physics to formulate, the coupled transcendental equations cannot be solved by hand, and a computer is needed to find a numerical solution. However, even the computer cannot solve this directly, but rather must solve it by trail and error.

Below we set up the solution for the two mass problem. Your problem is to solve the three mass problem and check out the reasonableness of your solution. You should check that the deduced tensions are positive and that the deduced angles correspond to a physical geometry (for example, with a sketch). Since this is a physics-based problem, we know that the sine and cosine functions must be less than 1 in magnitude, and that the tensions should be of similar magnitude to the weights of the spheres. Some of the **Exploration** you can do as you solve this problem is see at what point your initial guess gets so bad that the computer is unable to find a physical solution.

This physics problem is easy to convert to equations using the laws of statics, yet it is inhumanely painful to find analytic or numeric solution to the equations. However, the computer can usually find a numeric solution by trial and error if given a reasonable (physics-motivated) guess with which to start. Of course this is a very simple example, but for realistic applications with more masses, the analytic solutions may be impossible to find or too complicated to be useful. The numerical solution is usually quite easy.
We start by writing down the geometric constraints that the horizontal length of the structure is $l$ and that the strings begin and end at the same height:

$$L_1 \cos \theta_1 + L_2 \cos \theta_2 + L_3 \cos \theta_3 = L, \quad (56)$$

$$L_1 \sin \theta_1 + L_2 \sin \theta_2 - L_3 \sin \theta_3 = 0. \quad (57)$$

The basics physics says that since there are no accelerations, the sum of the forces on each mass must equal zero. The equations resulting from balancing the horizontal and vertical components of the forces on each mass are

$$T_1 \sin \theta_1 - T_2 \sin \theta_2 - W_1 = 0, \quad (58)$$

$$T_1 \cos \theta_1 - T_2 \cos \theta_2 = 0, \quad (59)$$

$$T_2 \sin \theta_2 + T_3 \sin \theta_3 - W_2 = 0, \quad (60)$$

$$T_2 \cos \theta_2 - T_3 \cos \theta_3 = 0. \quad (61)$$

where $W_i = m_i g$ is the weight of mass $i$, and $T_i$ is the tension in string $i$.

**Method: Newton-Raphson For Nonlinear Equations**

Actually, there is no good and general way to solve a system of simultaneous nonlinear equations—even on a computer! [Press, Fannery, Teukolsky, and Vettering, (1994), Numerical Recipes, Cambridge University Press] If you can guess approximate values for the solutions, then you can use the computer to search for progressively better and better solutions until you are satisfied with the results. (Computers are very good at quickly doing laborious calculations over and over again, but they are rather lacking in the intuition needed to make good guesses.)

As was true for a single equation, the Newton-Raphson method starts by rewriting the $N$ equations to be solved in a general form with zeros on the right-hand sides:

$$f_i(x_1, x_2, ..., x_N) = 0, \quad i = 1, N, \quad (62)$$

or in vector form

$$f(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_N(x) \end{pmatrix} = 0. \quad (63)$$

Here $x_1, x_2, \ldots x_N$ are the unknowns, and the $N$ $f_i$ functions get combine into the vector $f$. For our problem, we take the first six unknowns to be the sines and cosines of the angles, and the last three to be the tensions:
Even though the three $\sin \theta$'s and $\cos \theta$'s are not independent unknowns since $\sin^2 \theta_i + \cos^2 \theta_i = 1$, we treat them as independent and add in the trigonometric identities as additional equations to be solved. This makes the equations quadratic rather than trigonometric, and, as we shall see shortly, this makes them easier to linearize.

We now generalize the Newton-Raphson method we used for a single equation to the set of equations:

\[
\begin{align*}
\mathbf{x} &= \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix}, \\
\begin{pmatrix} \sin \theta_1 \\ \sin \theta_2 \\ \sin \theta_3 \\ \cos \theta_1 \\ \cos \theta_2 \\ \cos \theta_3 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix}.
\end{align*}
\]

\[ (64) \]

Even though the three $\sin \theta$'s and $\cos \theta$'s are not independent unknowns since $\sin^2 \theta_i + \cos^2 \theta_i = 1$, we treat them as independent and add in the trigonometric identities as additional equations to be solved. This makes the equations quadratic rather than trigonometric, and, as we shall see shortly, this makes them easier to linearize.

We now generalize the Newton-Raphson method we used for a single equation to the set of equations:

\[
\begin{align*}
 f_1(x) &= 3x_4 + 4x_5 + 4x_6 - 8 = 0, \\
 f_2(x) &= 3x_1 + 4x_2 - 4x_3 = 0, \\
 f_3(x) &= x_7x_1 - x_8x_2 - 10 = 0, \\
 f_4(x) &= x_7x_4 - x_8x_5 = 0, \\
 f_5(x) &= x_8x_2 + x_9x_3 - 20 = 0, \\
 f_6(x) &= x_8x_5 - x_9x_6 = 0, \\
 f_7(x) &= x_1^2 + x_4^2 - 1 = 0, \\
 f_8(x) &= x_2^2 + x_5^2 - 1 = 0, \\
 f_9(x) &= x_3^2 + x_6^2 - 1 = 0.
\end{align*}
\]

The solution is a set of nine $x_i$ values which make all nine $f_i$'s vanish simultaneously. Although these equations are not very complicated (the physics after all is elementary), the terms quadratic in $x$ make them nonlinear and this makes it hard or impossible to find an analytic solution. In fact, it is even hard to find a direct numerical solutions except by trial and error.

Accordingly, we use a trial and error approach. We expand the nonlinear equations into linear form and then solve the linear equations. We then take the $x$'s that solve the linear equation and substitute them into the nonlinear $f$'s and see how small the $f$'s are. We then use the new guess to generate an even-better guess, and continue the process until all the $f$'s are smaller than some level of precision.

The Newton-Raphson method expands the $f$'s in a Taylor series about the old values guessed for the $x_i$'s, keeps only the linear terms in the series, and then solves for the new $x_i$ values. Let the approximate solution
at any one stage be the set \( \{x_i, i = 1, 9\} \). We then assume that there is a set of corrections \( \{\Delta x_i, i = 1, 9\} \) for which
\[
f_i(x_1 + \Delta x_1, x_2 + \Delta x_2, ..., x_9 + \Delta x_9) = 0, \quad i = 1, 9.
\] (74)

We solve for the approximate values of the \( \Delta x_i \)'s by assuming that our previous solution is close enough to the actual solution for two terms in the Taylor series to be accurate:
\[
f_i(x_1 + \Delta x_1, x_2 + \Delta x_2, ..., x_9 + \Delta x_9) \approx f_i(x_1, x_2, ..., x_9) + \sum_{j} \frac{\partial f_i}{\partial x_j}(\{x_i\}) \Delta x_j = 0, \quad i = 1, 9,
\] (75)

We thus have a solvable set of nine linear equations in the nine unknowns \( \Delta x_i \). To make this clearer, we write them out as a matrix equation:
\[
\begin{pmatrix}
  f_1 \\
  f_2 \\
  \vdots \\
  f_9
\end{pmatrix} + \begin{pmatrix}
  \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_9} \\
  \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_9} \\
  \vdots & \vdots & \ddots & \vdots \\
  \frac{\partial f_9}{\partial x_1} & \frac{\partial f_9}{\partial x_2} & \cdots & \frac{\partial f_9}{\partial x_9}
\end{pmatrix} \begin{pmatrix}
  \Delta x_1 \\
  \Delta x_2 \\
  \vdots \\
  \Delta x_9
\end{pmatrix} = 0.
\] (76)

Since the derivatives and the \( f \)'s are all evaluated at known values of the \( x_i \)'s, only the vector of \( \Delta x_i \) values is unknown. Accordingly, the matrix equation has the standard linear-algebra form for simultaneous linear equations
\[
[A] \vec{x} = \vec{b},
\] (77)

with
\[
[A] = \begin{pmatrix}
  \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_9} \\
  \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_9} \\
  \vdots & \vdots & \ddots & \vdots \\
  \frac{\partial f_9}{\partial x_1} & \frac{\partial f_9}{\partial x_2} & \cdots & \frac{\partial f_9}{\partial x_9}
\end{pmatrix}
\] (78)

\[
\vec{x} = \begin{pmatrix}
  \Delta x_1 \\
  \Delta x_2 \\
  \vdots \\
  \Delta x_9
\end{pmatrix}, \quad \vec{b} = -\begin{pmatrix}
  f_1 \\
  f_2 \\
  \vdots \\
  f_9
\end{pmatrix}.
\] (79)

While for our two-mass problem we can derive analytic expressions for the derivatives \( \frac{\partial f_i}{\partial x_j} \), there are \( 9 \times 9 = 81 \) such derivatives and entering them all is time consuming and error prone. In contrast, and especially for more complicated problems, it is straightforward to program up a forward difference approximation for the derivatives:
\[
\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(x_j + \Delta x_j) - f_i(x_j)}{\Delta x_j}
\] (80)

While a central difference approximation for the derivative would be more accurate, it would require additional evaluations of the \( f \)'s. Note that since these are partial derivatives, each individual \( x_j \) is varied independently (easy to do within a loop).
Because these equations are linear, they can be solved for the $\Delta x_i$'s using the techniques of linear algebra. In fact, they are of the standard form,

$$[A][X] = [B].$$

(81)

The technique is repeated until the changes reach some tolerance limit. If the initial guess is close to a solution, then convergence is usually found after just a few iterations; if the initial guess is not close, the technique is likely to fail.

**Week 6  Reading 6, 7**

### 6.10 (2), 6.12:

Start at the origin and go take a two-dimensional random walk with your computer.

1. To build in as maximum randomness, choose independently random $\Delta x$'s and $\Delta y$'s in the range $[-1, 1]$ and then normalize the step size to unit length, $\Delta x^2 + \Delta y^2 = 1$.

2. Use your plotting program to draw maps of several independent random walks each of 1000 steps. Comment on whether these look like what you expect of a random walk.

3. If you have your walker taking $N$ steps in a single trial, then conduct a total of $K \simeq \sqrt{N}$ trials each with $N$ steps and each with a different seed.

4. Calculate $R^2$ for each trial and then take the average of $R^2$ for all your $K$ trials:

$$\langle R^2(N) \rangle = \frac{1}{K} \sum_{k=1}^{K} R^2_{(k)}(N).$$

(82)

5. Plot $R_{rms} = \sqrt{\langle R^2(N) \rangle}$ as a function of $\sqrt{N}$. Values of $N$ should start with a small number where $R = \sqrt{N}$ is not expected to be accurate, and end at a quite large value where 2-3 places of accuracy might be obtained on the average.

### 7.6, 7.7: Radioactive Decay

The simulation program for radioactive decay is surprisingly simple, but not without its subtleties. We have time progress in discrete steps of $\Delta t$, and then we count how many nuclei decay during each $\Delta t$. The simulation quits if there are no nuclei left. Accordingly, we have an outer loop over the time steps $\Delta t$, and an inner loop over the remaining nuclei. Some simple pseudocode is:

```plaintext
Input N, lambda
DeltaN = t = 0
Print t, n, DeltaN
Do over t
```

29
\[ t = t + \Delta t \]
\[ \Delta N = 0 \]

Do over \( N \)

\[ \text{if } r_i < \lambda, \Delta N = \Delta N + 1 \]

End do over \( N \)

\[ N = N - \Delta N \]

Print \( t, N, \Delta N \)

if \( N = 0 \), break

End do over \( t \)

Note that our pseudocode is meant to illuminate the logic of the program, and so is simple and without many details. Accordingly, a detailed pseudocode which is close to actual source code is more time consuming to write and may not be as useful.

At some point in the simulation the scale, or units, for the time is set. Since the decay rate constant \( \lambda = 1/\tau \), where \( \tau \) is the lifetime of the nucleus or level, picking a value of \( \lambda \) essentially sets the time scale for \( \Delta t \). Since we compare \( \lambda \) to a random number, and since random numbers are usually in the range \( 0 \leq r_i \leq 1 \), it is convenient to pick time units for which \( 0 \leq \lambda \leq 1 \) (for example, \( \lambda \approx 0.3 \) is a good place to start). This means that while the do loop may take \( \Delta t = 1 \), there is some unit for time assumed (about which you do not have to worry).

For those readers who do want to worry about this, let’s say that we have a nucleus with a lifetime \( \tau = 10 \) seconds. This corresponds to a decay rate constant \( \lambda = 1/\tau = 0.1/sec \). So if we use \( \lambda = 0.1 \) in the simulation, each time step would correspond to one second. Likewise, if we have a nucleus with a lifetime \( \tau = 10 \times 10^{-6} \) seconds, then this corresponds to a decay rate constant \( \lambda = 1/\tau = 0.1/10^{-6} \) sec. So if we use \( \lambda = 0.1 \) in the simulation, each time step would correspond to \( 10^{-6} \) of a second. The actual value of the decay rate \( \Delta N/\Delta t \) in particles per second would then be \( \Delta N/10^{-6} \).

Accordingly, the product of physical quantities \( \lambda \Delta t \) is dimensionless. Since the simulation uses integer time steps, this effectively sets \( \Delta t = 14 \), in which case the number that the simulation uses for \( \lambda \) is the product of physical quantities \( \lambda \Delta t \). However, unless you plan to compare your simulation to experimental data, you do not have to worry about the scale for time and can output \( \Delta N \) as the decay rate (it’s actually the physics behind the slopes and relative magnitudes of the graphs that we want to show).

Write your own program to simulate radioactive decay (this is really quite simple and you can use the sample program as a guide).

1. Plot the logarithm of the number left \( \ln N(t) \) and the logarithm of the decay rate \( \ln(\Delta N(t)) \) versus time. Note that the simulation measures time in steps (generation number) of \( \Delta t \).

2. Check that for large \( N(0) \) you obtain what looks like exponential decay, and that for small \( N(0) \) you get a stochastic process (the large \( N(0) \) simulation is also stochastic, it just doesn’t look it).
Figure 5: A fit to total cross section data using cubic splines. Individual cubic polynomials are used to interpolate between data points.

3. Create a plot to show that the slopes of the $N(t)$ versus $t$ curves are independent of the value used for $N(0)$, but are proportional to $\lambda$.

4. Create a plot to show that, within the expected stochastic variations, $\ln N(t)$ and $\ln(\Delta N(t))$ have the same time dependence.

5. Explain in your own words how a process that is spontaneous and random at its very heart leads to exponential decay.

6. How does your simulation show that the decay is exponential and not a power law like $N = \beta t^{-\alpha}$?

7.10: Throwing stones in a pond.

**Week 7 Reading 5**

As indicated in the text, cubic splines give pleasant-looking fits to data. In fact, once you have a spline fit you have individual cubic polynomials fits to each interval, and you can use these polynomials to replace a table of numbers by the set of cubics. Fitting a series of cubics to data is a little complicated to program up, so we recommend using a library routine.

While there are a number of Java-based spline applications available on the internet, none seemed appropriate for interpreting a simple set of numbers. Accordingly, with the help of Cristian Bordeianu of the Technological High School in Transylvania (Romania) we have adapted the `splint.c` and the `spline.c` functions from Press et al. for class use. We want you to learn how to use this program for general problems. The source for `SplineApp.java` is:

[SplineRHL.java](#)
import java.io.*;

public class SplineApp
{
    /** Cubic Spline fit to data. Based on Press et al, adapted by
     Cristian Constantin Bordeianu, Romania
     Data table = arrays x[n], y[n], nx = # of tabulated points
     x0 < x1 ... < x(n-1)
     Output = yout for given xout (here xout via loop at end)
     yp1 and ypn = 1st derivatives at endpoints, evaluated internally
     (setting yp1 or ypn >0.99e30 produces natural spline)
     y2[n] = array of second derivatives **/

    public static void main(String[] argv)throws IOException,
    FileNotFoundException
    {
        // save data in Spline.dat
        PrintWriter w =
            new PrintWriter(new FileOutputStream("Spline.dat"), true);
        PrintWriter q =
            new PrintWriter(new FileOutputStream("Input.dat"), true);
        // enter your input data here
        double x[] = {0., 1.2, 2.5, 3.7, 5., 6.2, 7.5, 8.7, 9.9};
        double y[] = {0, 0.93, 0.6, -0.53, -0.96, -0.08, 0.94, 0.66, -0.46};
        // save input data in Input.dat file
        for (i=0; i<n; i++) { q.println("  " + x[i] + "  " + y[i] + ");
        Nfit = 300; // enter the desired number of points to have data fit
        yp1 = (y[1]-y[0])/(x[1]-x[0]) // 1st deriv at initial point
         - (y[2]-y[1])/(x[2]-x[1])
         + (y[3]-y[2])/(x[3]-x[0]);
        ypn = (y[n-1]-y[n-2])/(x[n-1]-x[n-2]) // 1st deriv at end point
         - (y[n-2]-y[n-3])/(x[n-2]-x[n-3])
         + (y[n-1]-y[n-3])/(x[n-1]-x[n-3]);
        if (yp1 > 0.99e30) y2[0] = u[0] = 0.0; //natural spline test
        else {
            y2[0] = (-0.5);
            u[0] = (3.0/(x[1]-x[0]))*((y[1]-y[0])/(x[1]-x[0])-yp1);
        }
    }
}
for (i=1; i<n-2; i++) // decomposition loop; y2, u are temps
{
    sig = (x[i]-x[i-1])/(x[i+1]-x[i-1]);
    p = sig*y2[i-1]+2.0;
    y2[i] = (sig-1.0)/p;
    u[i] = (y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x[i]-x[i-1]);
    u[i] = (6.0*u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;
}

if (ypn > 0.99e30) qn = un = 0.0; //test for natural else
// else evaluate second derivative
{
    qn = 0.5;
    un = ((3.0)/(x[n-1]-x[n-2]))*(ypn-(y[n-1]-y[n-2])/(x[n-1]-x[n-2]));
}

y[n-1] = (un-qn*u[n-2])/(qn*y2[n-2]+1.0); for (k = n-2; k>= 0; k--)
{y2[k] = y2[k]*y2[k+1]+u[k];} // back substitution
// splint (initialization) ends

// Parameters determined, Begin *spline* fit

for( i=1; i<= Nfit; i++ ) // loop over evenly spaced xout values
{
    xout = x[0] + (x[n-1]-x[0])*(i-1)/(Nfit);

    klo = 0; // Bisection alg to find place in table
    khi = n-1; // klo, khi bracket xout value

    while (khi-klo > 1) {
        k = (khi+klo) >> 1;
        if (x[k] > xout) khi = k;
        else klo = k;
    }
    h = x[khi]-x[klo];
    if (x[k] > xout) khi = k;
    else klo = k;
    a = (x[khi]-xout)/h;
    b = (xout-x[klo])/h;
    yout = (1*a*y[klo]+b*y[khi]+((a*a*a-a)*y2[klo] + (b*b*b-b)*y2[khi])*(h*h)/6.0);

    w.println(" data stored in Spline.dat");
}
}
Figure 6: Computed wave function and square well potential. The wave function computed by integration in from the right is matched to the one computed by integration in from the left (dashed) at a point near the left edge of the well. Note how the wave function decays rapidly outside of the well. **Left:** Magnified view of ground state with no nodes. **Right:** Full view of excited state wave function with single node at origin.

5.5 (1) and cubic-spline fit. Extend this also for extrapolations outside of the range of the table. (More advanced: determine derivatives.)

5.17 or 5.18 or least-square fit of Polynomial

Week 8: Reading 8 & 9. We now have an RK2.java and RK4.java on the web as well as an adaptive step size extension in RK45_rhl.java. You should compare all three.

9.12:

8.9: Differentiation

9.13: Nonlinear oscillators

9.14: Energy conservation

Week 9: Eigenvalues of an ODE Reading 10
Chapter 10 in the text describes the method for solving the 1D Schrödinger equation for the energies of the bound states within a potential well. This combines the numerical solution of ordinary differential equations with the eigenvalue condition of finding a value for the energy for which the wavefunction is normalizable. The code `QuantumEigenRHL.java` uses `rk4` to solve the 1D Schrödinger equation,

\[
\frac{d^2 \psi(x)}{dx^2} + \frac{2m}{\hbar^2} [E - V(x)] \psi(x) = 0,
\]

for the wavefunction \( \psi(x) \). It solves for \( \psi_L(x) \) when integrating in from the right, and for \( \psi_R(x) \) when integrating in from the left. It then matches these two functions to each other at the matching point \( x_m \). To match, we use the bisection algorithm to find an energy at which the logarithmic derivative \( \psi'(x)/\psi(x) \) for the right wavefunction matches that of the left wavefunction. Explicitly, the program finds a zero of \( \text{diff}(E) \):

\[
\Delta(E) = \frac{\psi'_R(x_m) \psi_L(x_m) - \psi'_L(x_m) \psi_R(x_m)}{\psi'_L(x_m) \psi_R(x_m) + \psi'_R(x_m) \psi_L(x_m)}.
\]

1. Run `QuantumEigenRHL.java` and plot up the wavefunction and potential from the data files produced (note these have an arbitrary normalization). Deduce, by counting the number of nodes in the wave function, whether this is a ground state or an excited state.

2. Make an energy level diagram showing the depth of the potential (see the bottom of the code) and the energy of the state you found.

3. Change the value of the initial guess energy \( \text{guess} \) and search for other bound states. Make sure to examine the wavefunction for every state found (it must look physically reasonable) and note the number of nodes and symmetry (even/odd) of the wavefunction.

4. Add each new state found to your energy level diagram with labels such as 3E for the three node even state. Find as many of these states as you can.

**Week 10 Review**