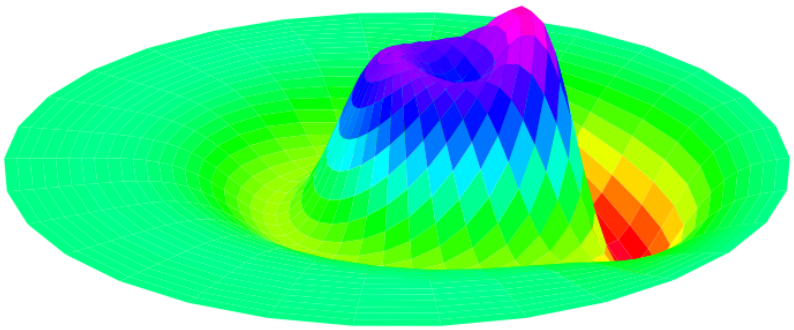
PH385: Numerical Methods in Physics



Lance J. Nelson

Department of Physics

PH385: Numerical Methods in Physics

Lance J. Nelson

Department of Physics

Brigham Young University–Idaho

Last Revised: January 21, 2020

Preface

This is a lab notebook intended to give you experience solving ordinary and partial differential equations numerically. The objectives of this course are

- to help you learn how to solve a differential equation numerically; in situations when a paper-and-pencil solution is impossible or impractical.
- to help you gain greater skills programming a computer and using loops, logic, functions, and classes.
- that your ability to produce a high-quality, professional scientific document will increase.

Text with a bold P designation (**P1.1** for example) indicate tasks that will be done together in class. Text with a bold H designation are homework problems and should be completed out of class. (working in groups is encouraged.)

Python is the programming language that we will be using You can obtain a free copy of Python here. Any computer code that you create should be uploaded to the Google Drive folder provided.

There is a companion book to this one entitled, "Introduction to Python". It is intended to help you learn to use Python to do the tasks contained herein.

Contents

Chapter 1

A short introduction to LaTeX

As a scientist (or engineer), one of your most important tasks will be to produce quality written documents detailing your work. The quality of the document that has your name attached to it speaks volumes about the type of professional you are. LaTeX is your best friend in this arena.

You can think of LaTeX as almost like a programming language for generating documents. You can create variables, loops, function, and even use if/else statements, all in the context of document production. We won't focus on the most complicated parts of LaTeX here but if you are interested, please come and talk to me. At first glance, LaTeX may seem like a complicated version of Microsoft Word, but as you gain experience using it you will find that it actually simplifies many complicated tasks. It is an especially useful tools when:

1. Your document contains a lot of math equations and you are referring to those equations frequently in the body of your document.
2. Your document has a lot of citations and a bibliography.
3. Your document has a lot of figures/graphics and you are frequently referencing them in the body of your document.
4. You'd like to use your programming skills to build nifty automations when building a document. Here are a few examples
   (a) When I build an exam, I include the questions and the solutions in the same document. By modifying a single variable I can turn those solutions on to build the key and off to build the exam.
   (b) When I build a schedule/calendar, I have a file that lists the correct dates and other date-specific information. When the schedule is built, the dates are read from the file. If I need to modify the schedule, I simply modify the list of dates and recompile.

LaTeX is used heavily in research environments and is a skill that will be an asset to you.

Getting Started

The first thing we ought to learn is how to create a simple document using LaTeX.

P1.1 (5 pts) Open your editor and type the following into the window and press the `Typeset` button.

```
\documentclass{article}
\begin{document}
This is a new document.  I can type whatever I want and it will appear
in the body of the text
\end{document}
```

You probably wouldn't use LaTeX to produce a document this simple, but at least you can see how simple documents are produced. Please note that every document you produce must have these lines:

```
\documentclass{article}
\begin{document}

\end{document}
```

Math

The fun starts when you need to add math to your document. LaTeX is great when it comes to producing great-looking, numbered math equations that can easily be referenced from within the body of the text. Literally any math symbol you could want can be produced if you know the correct syntax. A full listing of the syntax for all of the math symbols will not be provided here but can be easily found with the help of Google.

There are several different ways that you may want to incorporate math into your document. You may just want to add a math equation in the middle of a sentence

P1.2 (5 pts) Type the following into your editor window and press the `Typeset` button.

```
\documentclass{article}
\begin{document}
To find the electric field, we need to evaluate the integral:
$\frac{1}{\alpha} \int_0^{10} \ln(2 x) dx$ and then take the limit as
$\alpha \rightarrow \infty$.
\end{document}
```

Take a second to look at the output and to digest the code. Ask any questions that you may have.

Let me highlight a few things that you should have noticed:

1. When typing math in the middle of a sentence, you must enclose the math expression in $ symbols.

2. Sub- and super- scripts are done just as you would expect. If the sub- or super-script is longer than one character, you must enclose it in curly braces ({}).

3. Common math symbols can be easily produced if you know the syntax. Here we see that `\int` is the syntax for the integral symbol, `\ln` is the syntax for ln, and `\frac{}{}` is the symbol for making fractions (`{num \over denom}` also works). The syntax for other commonly-used math symbols is provided in table 1.1.

4. Greek letters can also be produced if you know the correct syntax. In this example we see that `\alpha` is the syntax for α. The syntax for a few of the other common greek letters is given in table 1.2

Sometimes the math that you want to write down is a little longer and you'd like it to be on it's own line. That's no problem in LaTeX

P1.3 (5 pts) Type the following into your editor windown and push the `Typset` button:

```
\documentclass{article}
\usepackage{amsmath} % Needed to use \eqref
\begin{document}
To find the electric field, we need to evaluate the integral:
\begin{equation}\label{eq:integralEquation}
 \frac{1}{\alpha} \int_0^{10} \ln(2 x) dx
\end{equation}

When we evaluate equation \eqref{eq:integralEquation}, we find that it
equals $0$ because $\alpha = \infty$.

\end{document}
```

Take a second to look at the output and to digest the code. Ask any questions that you may have.

`\frac{1}{5}`	$\frac{1}{5}$
`\sqrt{5}`	$\sqrt{5}$
`\int`	$\int$
`\ln`	$\ln$
`\oint`	$\oint$
`\sum`	$\sum$
`\infty`	∞
`\nabla`	∇
`\partial`	∂

Table 1.1 Commonly-used LaTeX math symbols.

Once again, let me highlight a few things that you should have noticed:

1. When you want an equation to be numbered and located on it's own line you can use the

```
\begin{equation} \label{myLabel}
<equation>
\end{equation}
```

2. The `\label` command can be used to assign a name to your equation.

3. You can use the name you gave to your equation to reference it in the text. This means that you don't need to know the number that it was assigned.

This is done like this `\eqref{equationLabel}`. Some commands require that you import an external package. In this case, the `\eqref` command needed the `amsmath` package.

Often you will have multiple lines of math and you want to be very careful about how the lines line up. Let's explore that a little bit.

P1.4 (5 pts) Type the folowing into the editor window and press the `Typeset` button [1]

[1] A copy/paste may save you some time.

```
\documentclass{article}
\usepackage{amsmath}
\begin{document}

\begin{equation}\label{kRadius}
k = \sqrt{\frac{2mE_f}{\hbar^2}}
\end{equation}

Now we can re-arrange to solve for $E_f$:

\begin{align}
3 \pi^2 \frac{N}{V} &= \left(\frac{2 m E_f}{\hbar^2}\right)^{3/2}\\
3 \pi^2 n &= \left(\frac{2 m E_f}{\hbar^2}\right)^{3/2}\\
\left(3 \pi^2 n\right)^{1/3} &= \left(\frac{2 m E_f}{\hbar^2}\right)^{1/2}\\
\left(3 \pi^2 n\right)^{1/3} &= \sqrt{\frac{2 m E_f}{\hbar^2}}\\
\end{align}

Comparing to equation \eqref{kRadius} we can conclude that:

\begin{equation}
k = \left(3 \pi^2 n\right)^{1/3}
\end{equation}


\end{document}
```

As before, digest what you see and ask any questions that you may have. What useful bits of information can you extract from this example

`\alpha`	α
`\beta`	β
`\gamma`	γ
`\chi`	χ
`\Delta`	Δ

Table 1.2 Commonly-used greek letters in LaTeX.

These are just a few ways to produce math equations in your document. As you progress in your abilities, more questions will undoubtedly arise. We'll handle those situations case by case in this class. In conclusion, LaTeX produces beautiful math that is formatted in exactly the way that you want it and can be easily referenced from within the text. It should be your go-to tool anytime you need to produce a document with math in it.

Figures

A key element of any scientific document are graphics. This could be a plot or a chart, or just an image. Regardless, we need to figure out how to include them in our LaTeX document.

P1.5 (5 pts) Download a picture of an elephant and save it to your computer. Then type the folowing into the editor window and press the `Typeset` button [2] [3]

```
\documentclass{article}
\usepackage{graphicx}  % You need this anytime you want to include graphics
\begin{document}

In figure \ref{figLabel} you will find an image of an elephant.
\begin{figure}
\includegraphics[scale = 1.2]{path/to/figure/of/elephant}
\caption{This is the caption to the figure \label{figLabel}}
\end{figure}

\end{document}
```

Look over the code and the output until things start to make sense. Ask any questions that you may have.

[2] A copy/paste may save you some time.

[3] You may have to fiddle with the `scale=1.2` to get an appropriate size.

Once again, let me highlight a few things that you should have noticed:

1. Anytime you are including graphics in your document you will need the `graphicx` package.
2. Including a graphic is done with the `\includegraphics` command. The required argument to this function (found in curly braces) is the path to the image file. One of the optional arguments (found in square brackets) is `scale = 1.2`, which allows you to specify the size of the image.
3. To label a figure (for referencing in the text) and creating captions you need to place your `\includegraphics` statement in

```
\begin{figure}

\end{figure}
```

Citations and Bibiliography

A key element to any scientific paper are citations and a bibliography page. It is not uncommon for a published paper to have 30-40 citations. Tracking and managing all of these citations manually would be a mind-numbing task. Luckily, LaTeX can handle all of this for you. There are two things that need to be discussed regarding citations: i) finding the information for a source to be cited, ii) citing a source and creating the bibliography.

Finding source information

Google Scholar is a great place to do literature searches and it can help you gather the bibliography information too. But there are some settings that need to be altered.

P1.6 (5 pts) Follow the steps below to enable tex-friendly citation information:

(a) Go to www.google.com/scholar

(b) In the upper left corner, click the drop down menu and click on `Settings`.

(c) Under Bibliography manager, click "Show link to import citations into BibTeX" and click "Save"

(d) Perform a search for "Superalloys", or some other interesting topic of your choosing.

(e) Near the bottom of the first hit, there should be a link entitled "Import into BibTex". Click it. This is the source information that you will need

Citing a source and creating the bibliography

For a scientist that is actively and frequently publishing papers, it is quite common for them to cite the same source(s) in multiple publications. To simplify the citation process, a file that contains all of his frequently-cited sources (commonly referred to as a "bib" file) is maintained.

P1.7 (5 pts) Follow the steps below to create a simple bib file.

(a) Create a new file named `refs.bib`. It needs to have the `.bib` postfix.

(b) Using Google Scholar, search for a few publications and copy their bibTex entry into the file refs.bib. (See previous section) My refs.bib looks like this:

```
@article{caron2000high,
  title={High ysolvus new generation nickel-based superalloys for single crys
  author={Caron, P},
  journal={Superalloys},
  volume={2000},
  pages={737--746},
  year={2000}
}

@article{pettit1984oxidation,
  title={Oxidation and hot corrosion of superalloys},
  author={Pettit, FS and Meier, GH and Gell, M and Kartovich, CS and Bricknel
  journal={Superalloys},
  volume={85},
  pages={65},
  year={1984}
}
```

```
@article{pollock2006nickel,
  title={Nickel-based superalloys for advanced turbine engines: chemistry, microstructure, and prope
  author={Pollock, Tresa M and Tin, Sammy},
  journal={Journal of propulsion and power},
  volume={22},
  number={2},
  pages={361--374},
  year={2006}
}
```

(c) Save the file

(d) Create another file in the same directory as the bib file. Put the following into the file

```
\documentclass{article}
\begin{document}

As I discuss superalloys I may need to make a citation
\cite{pollock2006nickel}, or maybe even two at a time \cite{pettit1984oxidation,caron2000high}.

\bibliographystyle{ieeetr}  % There are various styles to choose from.
\bibliography{refs}
\end{document}
```

[4] Study the code and the output until things make sense. Ask any questions that you may have. **Note: If you are compiling your code at the command line, you will need to execute the following four commands:**

```
pdflatex texFile
bibtex texFile (no .tex extension here)
pdflatex texFile
pdflatex texFile
```

[4] The names of your citations will likely be different from mine. You are also free to modify the names in the refs.bib file to be whatever you want.

Miscellaneous

The possible topics relating to LaTeX functionality fills entire books. I will not try to be complete in my coverage. Rather, let me give you a few more handy tidbits that come up frequently. Below you will find

Section Headings

When organizing a document, you will want to use sections and subsections. Here's how to do it.

```
\section{Name of Section}  % Section with numbering
\subsection{Name of Subsection} % Subsection with numbering
\section*{Name of Section}  % Section with no numbering
\subsection*{Name of Subsection} % Subsection with no numbering
```

As mentioned in the comments, adding a * will supress the numbering of the sections.

Abstract and Title

Every scientific document has an abstract, or short summary, of the document. Beneath the title of every paper is listed the authors names and affiliations. Here is how all of this is generated in latex:

```
\documentclass{article}
\usepackage{authblk}

            \title{The effect of variable air density on the trajectory of a cannon shell.}
            \author[1]{Lance J. Nelson}
            \affil[1]{Brigham Young University - Idaho}
            \author[2]{SECONDARY AUTHOR}
           \affil[2]{SECONDARY AUTHOR affiliation}
            \date{\today}
\begin{document}
    \maketitle
    \begin{abstract}
      Most projectile motion problems assume that the air density remains
      constant for the duration of the motion.  This is not a bad
      asssumption when the projectiles maximum altitude is relatively
      small.  However, for high altitude projectiles this may be a poor
      assumption.  In this work, we will investigate how a variable air
      density changes the trajectory of a high-altitude projectile.  We
      will also explore the effect of ground temperature on the range of
      these projectiles.
    \end{abstract}


This is my article


\end{document}
```

Including code in your document

In this class, you may want to include some or all of your code and explain what you did. Instead of copying your code into LaTeX (ughh), LaTeX can read your code file and place it into the document, complete with text highlighting specific to the language you are coding in. Here is how you do it: [5]

[5] Pay special attention to the comments for help understanding

```
\documentclass{article}
\pdfoutput=1

\usepackage{fancyvrb}
\usepackage{color}

\definecolor{purple}{rgb}{0.625,0.125,0.9375}
\usepackage{listings}
\lstset{
  frame=lines,  % top and bottom rule only
  framesep=2em, % separation between frame and text
    keepspaces=true,
    aboveskip=0in,
    belowskip=0.2in,
    language=Python, % What language are you coding in
    fancyvrb=true,
    breaklines=false,
    basicstyle=\footnotesize\ttfamily,
    numbers=left,
    stepnumber=1,
    keywordstyle=\color{blue},  %What color for keywords
    identifierstyle=,
    commentstyle=\color{red}, % What color do you want comments.
    stringstyle=\ttfamily\color{purple},
    columns=fullflexible,
    showstringspaces=False,
    caption = {  The following is an example code},
    captionpos = b  % Where do you want the caption located
    }

\begin{document}


\lstinputlisting{testPython.py}  % This is where you specify
                                 % the location of your code file.
\end{document}
```

To say that we have only scratched the surface of what LaTeXcan do would be a huge understatement. However, what we have given you will suffice for the requirements of this class. If your curiosity overwhelms you and you must have more, please feel free to come talk with me one-on-one.

Homework

H1.8 Recreate the document called exampleWriteUp.pdf found on iLearn. The figures used in the document are also found on iLearn. You are free to use copy/paste to help speed up the process.

When constructing your document, make sure that you reference all figures and math equations using the `\label` tag rather than hard-coding a specific number.

Note: The abstract and title code from above will not produce a document that looks identical to mine. To make your document look exactly like mine, use the following code block at the top of your document :

```
\documentclass[aps,prb,twocolumn,amsfonts,showpacs,letterpaper]{revtex4-1}

\pdfoutput=1
%\usepackage{natbib}
\usepackage{siunitx}  % <- You need this package to use the 'align' environment
\usepackage{amsmath}  % <- You need this package to use the 'align' environment
\usepackage{graphicx}  %<-  You need this package to insert graphics
%\usepackage{authblk}


%\bibliographystyle{apsrev4-1}
\begin{document}
        \title{The effect of variable air density on the trajectory of a cannon shell.}
        \author{Lance J. Nelson}
        \affiliation{Brigham Young University - Idaho}
%       \author{SECONDARY AUTHOR}
%       \affiliation{SECONDARY AUTHOR's affiliation}
        \date{\today}

\begin{abstract}
  Most projectile motion problems assume that the air density remains
  constant for the duration of the motion.  This is not a bad
  assumption when the projectile's maximum altitude is relatively
  small.  However, for high altitude projectiles this may be a poor
  assumption.  In this work, we will investigate how a variable air
  density changes the trajectory of a high-altitude projectile.  We
  will also explore the effect of ground temperature on the range of
  these projectiles.
\end{abstract}

\maketitle
```

Chapter 2

A short introduction to Python

One goal of this class is that you become proficient with Python in a scientific setting. This will not happen all at once, but gradually over the course of the entire semester. You have already been exposed to python in PH295 (previously PH291) but if you are like most students, you might have struggled with loops, logic and functions. These basic programming concepts are pretty crucial in this class. In this chapter, we'll review some python syntax and practice with some problems.

Data Types

The most commonly used types of data that will be used for this class are: integers, floats, strings, lists, and arrays.

Read sections 3.1 – 3.6 in the Python manual to learn about the first four data types and sections 5.1–5.3 to learn about arrays. If something you read is confusing or unfamiliar to you, I suggest that you try to recreate the calculation yourself. Take good notes so you can ask good questions during class.

Try the problem below to test your ability

P2.1 (5 pts)

1. Build an array containing all multiples of 3 starting at 3 and ending at 2000.
2. Calculate $3x^2\sin(x) + \ln(x)$, where x is the list you just created. In other words, for every number in the array you just created, calculate the quantity above. This will yield an array of calculated values.
3. Sum the elements of the list to get a single final answer.

Ans: -736905.545292

Loops and Logic

A loop is a set of instructions to be performed until some pre-determined criteria is satisfied. Loops are used heavily in this class and you must grasp what they are as soon as possible. Logic refers to the act of checking the truthfulness of a statement and is used heavily in conjunction with loops.

Read chapter 6 in the Python manual to get a better feel for what we mean. If something you read is confusing or unfamiliar to you, I suggest that you try to

recreate the calculation yourself. Take good notes so you can ask good questions during class.

Try the following problem to test your understanding of `for` loops

P2.2 (10 pts) On iLearn, you will find a file entitled "massdata.txt" which contains the mass of a collection of objects and their (x, y, z) coordinates.

(a) In Python, read in the data from the file (see chapter 8 in the python book).

(b) Use a 'for' loop to loop over the data points and calculate the center of mass coordinates for the collection of particles. You may remember that the equation find the center of mass is

$$x_{\text{cm}} = \frac{1}{M}\sum_{i}^{N} x_i m_i$$

(c) Can you find the center of mass without using a loop?

Try the following problem to test your understanding of `while` loops

P2.3 (10 pts) Perform the summation below

$$\sum_{n=1}^{\infty} nx^n$$

using a `while` loop. Make your own counter for n by using $n = 0$ outside the loop and $n = n + 1$ inside the loop. Have the loop execute until the current term in the sum, nx^n has dropped below 10^{-8}. Verify that the sum only converges for $|x| < 1$ and that when it does converge, it converges to $x/(1-x)^2$.
Caution: When building `while` loops, you can easily get caught in an infinite loop, one that never ends. It's always wise to build in a fail safe: A variable to count the number of iterations and that breaks out of that loop if the number gets too high.

Functions

A function is like a black box. The user passes some needed information into the box and the box uses that information to perform some useful calculation and spits a result out.

Read chapter 4 in the Python manual to see how to build functions in Python. If something you read is confusing or unfamiliar to you, I suggest that you try to recreate the calculation yourself. Take good notes so you can ask good questions during class.

Try the following problem to test your abilities

P2.4 (10 pts) Build a function that contains a loop that sums the integers from 1 to N, where N is an integer value that is passed into the function. By calling the function at least 5 times, verify that the formula

$$\sum_{n=1}^{N} n = \frac{N(N+1)}{2}$$

is correct.

Plotting

Computers can't plot functions. Computers can only evaluate functions and plot data points. If you choose to evaluate a function on a very dense grid of points, plot those points, and connect them with a line, the resulting plot will resemble the function. This way of thinking will be increasingly important as the semester progresses. Read chapter 7 in the Python manual to get a better idea of how to construct a plot in Python. If something you read is confusing or unfamiliar to you, I suggest that you try to recreate the calculation yourself. Take good notes so you can ask good questions during class.

P2.5 (5 pts) Plot the function

$$y(x) = e^{-0.25x}\cos(x)$$

from $x = 0$ to $x = 10$.

Classes

You can think of a class as an object that contains both data and functions. Since you are already familiar with variables and functions, the best way to learn about classes is to see one and then ask questions. Here is a simple class used to analyze ideal gas processes:

```
class idealGas():
    def __init__(self,R=8.314,kB=1.38e-23):
        self.R = R
        self.kB = kB

    #Define what kind of ideal gas process we are dealing with
    def setProcessType(self,process, n,cV):
        self.n = n  #Set the number of moles of the gas
        self.process = process  #Indicate what kind of process it is:
                                # Isothermal, Isobaric, Adiabatic, Isochoric
        self.cV = cV
        self.cP = self.cV + self.R
        # If the process is adiabatic, we might need to know what gamma is.
        if process == 'Adiabatic':
             self.gamma = self.cP/self.cV

    # Set the initial and final conditions for the process.
    def setConditions(self,pi=None,Vi=None,Ti=None,pf=None,Vf=None,Tf=None):
        self.pi = pi
        self.Vi = Vi
        self.Ti = Ti
        self.pf = pf
        self.Vf = Vf
        self.Tf = Tf

    def isothermal(self):
        from math import log
        self.work = -self.n * self.R * log(self.Vf/self.Vi)
        self.deltaE = 0
        self.heat = - self.work

    def isochoric(self):
        self.work = 0
        self.heat = self.n * self.cV * (self.Tf - self.Ti)
        self.deltaE = heat

    def isobaric(self):
        self.work = - self.pI * (self.Vf - self.Vi)
        self.heat = self.n * self.cP * (self.Tf - self.Ti)
        self.deltaE = self.n * self.cV * (self.Tf - self.Ti)

    def adiabatic(self):
        self.heat = 0
        self.deltaE = self.n * self.cV * (self.Tf - self.Ti)
        self.work = self.deltaE

    # Depending on what type of process it is, calculate the work
    # done, heat absorbed, and change in internal energy for the process.
    def calculate(self):


        # These three lines can replace the lines below and its a really
        # cool use of dictionaries. Try it.
        #         processes = {'Isothermal':self.isothermal, 'Adiabatic':
        #   self.adiabatic, 'Isochoric':self.isochoric, 'Isobaric':self.isobaric}
```

```
            # processes[self.process]()

             if self.process == 'Isothermal':
                 self.isothermal()
             elif self.process == 'Isochoric':
                 self.isochoric()
             elif self.process == 'Isobaric':
                 self.isobaric()
             else:  #Adiabatic
                 self.adiabatic()

# Class definition done. What follows below illustrates how to use the class.
gasOne = idealGas()  #Initialize the class.
gasOne.setProcessType('Isothermal',3,20.8)  # Set the type of process and number of moles
gasOne.setConditions(pi = 2.0e5, pf = 1.0e5,Ti = 200, Tf = 200, Vi = 2, Vf = 5 )  # Set initial and final condi
gasOne.calculate()  # Calculate thermodynamic properties.
print(gasOne.work)  # Print results
print(gasOne.deltaE)
```

Homework

H2.6 (10 pts) Build a function that takes argument x and performs the sum

$$\sum_{n=1}^{1000} nx^n$$

using a `for` loop. By calling the function with different values of x, verify that the sum only converges for $|x| < 1$ and that when it does converge, it converges to $x/(1-x)^2$.

H2.7 (10 pts) Verify, by numerically experimenting with a loop that uses the `break` command to exit the loop at the appropriate time, that the following infinite-product relation is true:

$$\prod_{n=1}^{\infty}\left(1+\frac{1}{n^2}\right) = \frac{\sinh\pi}{\pi}$$

H2.8 (10 pts) (**Transcendental Equations**) A transcendental equation is one that cannot be solved analytically. Try solving the following equation for x:

$$\frac{\sin(x)}{x} = 1$$

to see what I mean. One numerical method for solving this kind of problem involves first rearranging the equation to look like this:

$$x = \sin(x)$$

and then repeatedly evaluating the r.h.s using the result of the previous evalution until subsequent evaluations differ very little. [6]

[6] This method is called successive evaluation

Use a `while` loop to verify that the following three iteration processes converge. Execute the loops until convergence at the 10^{-8} level is achieved.

$$x_{n+1} = e^{-x_n} \quad ; \quad x_{n+1} = \cos x_n \quad ; \quad x_{n+1} = \sin 2x_n$$

Note: iteration loops are easy to write. Just give x an initial value (any number will do) and then inside the loop replace x by the formula on the right-hand side of each of the equations above. To watch the process converge you will need to call the new value of x something like `xnew` so you can compare it to the previous x.

Finally, try iteration again on this problem:

$$x_{n+1} = \sin 3x_n$$

Convince yourself that this process isn't converging to anything. We will see in Lab 19 what makes the difference between an iteration process that converges and one that doesn't.

H2.9 (15 pts) Write a class to calculate the trajectory of a projectile. You should have at least three member functions to: i) set the launch conditions (initial velocity, position, etc. ii) calculate the landing location and iii) plot the trajectory.

[1]

[1] **giordano1997computational**.

Part I

Preliminaries

Chapter 3

Discrete Grids

Python skills that you will need for today:
Plotting (Chapter 7), numpy array construction (section 5.3)

Analytical solutions to differential equations yield analytic functions; functions that can be evaluated at arbitrary points. Transitioning from analytical solutions to differential equations to numerical solutions requires a shift in mentality. When you solve a differential equation numerically, the solution function is represented as an array of function values on a discrete grid. It is important now to develop the intuition and skills for representing a function as a discrete set of values on a grid. Before we proceed to solving differential equations, let's spend some time getting comfortable working with spatial grids.

Spatial grids

Figure 3.1 shows a graphical representation of three types of spatial grids for the region $0 \le x \le L$. We divide this region into spatial *cells* (the spaces between vertical lines) and functions are evaluated at N discrete *grid points* (the dots). In a *cell-edge* grid, the grid points are located at the edge of the cell. In a *cell-center* grid, the points are located in the middle of the cell. Another useful grid is a cell-center grid with *ghost points*. The ghost points (unfilled dots) are extra grid points on either side of the interval of interest and are useful when we need to consider the derivatives at the edge of a grid.

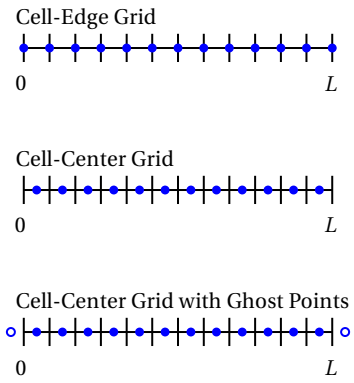


Figure 3.1 Three common spatial grids

P3.1 (a) (5 pts) Make a Python script that creates a cell-edge spatial grid in the variable `x` over the interval $0 \le x \le \pi$ with $N = 500$. Plot the function $y(x) = \sin(x)\sinh(x)$ on this grid. Explain the relationship between the number of cells and the number of grid points in a cell-edge grid. Then verify that the number of points in this x-grid is N (using Python's `len` command).

(b) (5 pts) Repeat part (a) but now use a cell-centered grid. Notice that your grid doesn't start at `a` or end at `b`. Rather, your grid starts at $a + \frac{dx}{2}$ and ends at $b - \frac{dx}{2}$, where dx is the grid spacing. This means that you will need to add a line that calculates the grid spacing. Explain the relationship between the number of cells and the number of grid points in a cell-centered grid and decide how you should calculate dx to get the same grid spacing as for the cell-edge grid.

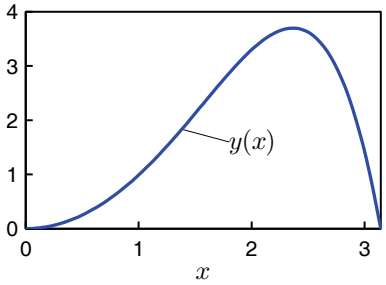


Figure 3.2 Plot from 3.1(a)

(c) (5 pts) Now write a script like the one in part (b) to build a cell-center grid over the interval $0 \le x \le 2$ with $N = 5000$. Evaluate the function $f(x) = \cos x$ on this grid and plot this function. Then estimate the area under the curve by summing the products of the centered function values f_j with the widths of the cells h like this (midpoint integration rule):

```
sum(f)*h;
```

Verify that this result is quite close to the exact answer obtained by integration:

$$A = \int_0^2 \cos x \, dx.$$

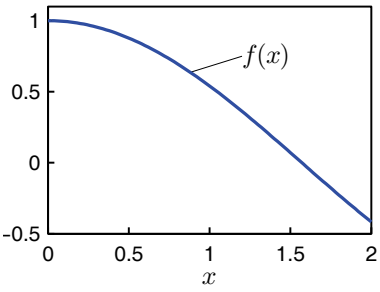


Figure 3.3 Plot from 3.1(b)

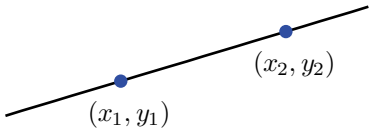


Figure 3.4 The line defined by two points can be used to interpolate between the points and extrapolate beyond the points.

Interpolation and extrapolation

Grids only represent functions at discrete points, and there will be times when we want to find good values of a function *between* grid points (interpolation) or *beyond* the last grid point (extrapolation). We will use interpolation and extrapolation techniques fairly often during this course, so let's review these ideas.

The simplest way to estimate these values is to use the fact that two points define a straight line. For example, suppose that we have function values (x_1, y_1) and (x_2, y_2). The formula for a straight line that passes through these two points is

$$y - y_1 = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) \tag{3.1}$$

Once this line has been established it provides an approximation to the true function $y(x)$ that is pretty good in the neighborhood of the two data points. To linearly interpolate or extrapolate we simply evaluate Eq. (3.1) at x values between or beyond x_1 and x_2.

P3.2 Use Eq. (3.1) to do the following special cases:

(a) (5 pts) Find an approximate value for $y(x)$ halfway between the two points x_1 and x_2. Does your answer make sense?

(b) (5 pts) Find an approximate value for $y(x)$ 3/4 of the way from x_1 to x_2. Do you see a pattern?

Note: You should find that

$$y(x_1 + ph) = y_1 + p(y_2 - y_1) \tag{3.2}$$

, where p is a fraction of the grid spacing. ($\frac{1}{2}$ for part a and $\frac{3}{4}$ for part b.)

(c) (5 pts) If the spacing between grid points is h (i.e. $x_2 - x_1 = h$), show that the linear extrapolation formula for $y(x_2 + h)$ is

$$y(x_2 + h) = 2y_2 - y_1 \tag{3.3}$$

This provides a convenient way to estimate the function value one grid step beyond the last grid point.

(d) (5 pts) Also show that

$$y(x_2 + \frac{h}{2}) = \frac{3}{2}y_2 - \frac{1}{2}y_1. \tag{3.4}$$

We will use both of these formulas during the course.

A fancier technique for finding values between and beyond grid points is to use a parabola instead of a line. It takes three data points to define a parabola, so we need to start with the function values (x_1, y_1), (x_2, y_2), and (x_3, y_3). The general formula for a parabola is

$$y = a + bx + cx^2 \tag{3.5}$$

where the coefficients a, b, and c need to be chosen so that the parabola passes through our three data points. To determine these constants, you set up three equations that force the parabola to match the data points, like this:

$$y_j = a + bx_j + cx_j^2 \tag{3.6}$$

with $j = 1, 2, 3$, and then solve for a, b, and c.

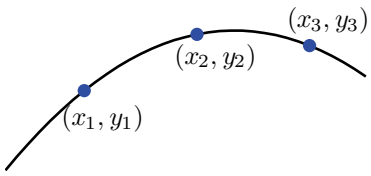


Figure 3.5 Three points define a parabola that can be used to interpolate between the points and extrapolate beyond the points.

P3.3 Use Eq. (3.6) to create a set of three equations in Mathematica. For simplicity, assume that the points are on an evenly-spaced grid and set $x_2 = x_1 + h$ and $x_3 = x_1 + 2h$. Solve this set of equations to obtain some messy formulas for a, b, and c that involve x_1, y_1, y_2, y_3, and h. Then use these formulas to solve the following problems:

(a) (10 pts) Estimate $y(x)$ half way between x_1 and x_2, and then again halfway between x_2 and x_3.

(b) (10 pts) Show that the quadratic extrapolation formula for $y(x_3 + h)$ (i.e. the value one grid point beyond x_3) is

$$y(x_3 + h) = y_1 - 3y_2 + 3y_3 \tag{3.7}$$

Also find the formula for $y(x_3 + h/2)$.

Homework

H3.4 (a) Build a cell-centered grid in the variable x over the interval $0 \le x \le \pi$ with $N = 500$.

(b) Evaluate the function $\sinh(x)$ on the grid.

(c) Use linear extrapolation to determine the value of the function one grid point beyond the last one. Compare to the true value of the function at that point.

(d) Use quadratic extrapolation to determine the value of the function one grid point beyond the last one. Compare to the true value of the function at that point.

(e) Use linear interpolation to determine the value of the function between the last and second-to-last grid point. Compare to the true value of the function.

(f) Use quadratic interpolation to determine the value of the function between the last and second-to-last grid point. Compare to the true value of the function.

Chapter 4

Numerical Derivatives

In calculus class (long ago now) you were taught analytical techniques for calculating derivatives and integrals. In other words, given a function $f(x)$ you were taught how to find the function: $\frac{df}{dx}$. These techniques are great if you have $f(x)$ but aren't very helpful if you don't. In numerical physics you typically don't have $f(x)$ but you probably have a way to gather samples from $f(x)$. How can we use these discrete function values to calculate derivatives and integrals. This will be our topic for this chapter.

Numerical Derivatives

In your introductory calculus book, the derivative was probably introduced using the *forward difference* formula

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{4.1}$$

The word "forward" refers to the way this formula reaches forward from x to $x+h$ to calculate the slope. The exact derivative represented by Eq. (4.1) in the limit that h approaches zero. However, we can't make h arbitrarily small when we represent a function on a grid because (i) the number of cells needed to represent a region of space becomes infinite as h goes to zero; and (ii) computers represent numbers with a finite number of significant digits so the subtraction in the numerator of Eq. (4.1) loses accuracy when the two function values are very close. But given these limitation we want to be as accurate as possible, so we want to use the best derivative formulas available. The forward difference formula isn't one of them.

The best first derivative formula that uses only two function values is usually the *centered difference* formula:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{4.2}$$

It is called "centered" because the point x at which we want the slope is centered between the places where the function is evaluated. The corresponding centered second derivative formula is

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{4.3}$$

You will derive both of these formulas a little later, but for now we just want you to understand how to use them.

To see the importance of centering, consider Fig. 4.2. In this figure we are trying to find the slope of the tangent line at $x = 0.4$. The usual calculus-book

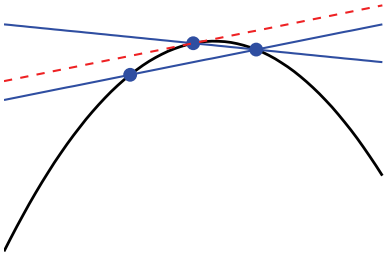


Figure 4.1 The forward and centered difference formulas both approximate the derivative as the slope of a line connecting two points. The centered difference formula gives a more accurate approximation because it uses points before and after the point where the derivative is being estimated. (The true derivative is the slope of the dotted tangent line).

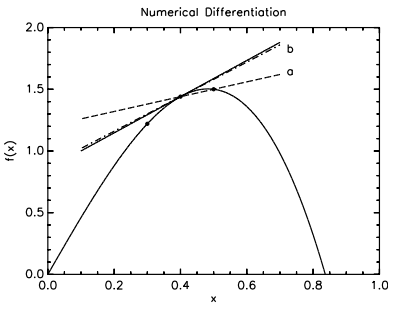


Figure 4.2 The centered derivative approximation works best.

formula uses the data points at $x = 0.4$ and $x = 0.5$, giving tangent line a. It should be obvious that using the "centered" pair of points $x = 0.3$ and $x = 0.5$ to obtain tangent line b is a much better approximation. **When approximating derivatives using finite difference we'll always want to use the centered difference (if possible).**

P4.1 To experiment with the different numerical derivative formulas and to see why the centered-difference is better, do the following:

(a) Differentiate $\sin(x)$ at $x = 1$ using the forward-difference formula and the centered-difference formula.

(b) The true value of the derivative is $\cos(1)$. Compute the ratio of the numerical derivative with the true answer for both methods.

(c) Can you see which method is better?

Derivatives on Grids

Equations (4.2) and (4.3) are great for evaluating a derivative at a single point. But what if we have an entire array of function values and we'd like to evaluate the derivative at all of the points. In other words, what if we want to determine the shape of the derivative function. Luckily, when grids are stored in Numpy arrays, the colon operator provides a compact way to evaluate Eqs. (4.2) and (4.3) on a grid.

P4.2 Perform the following to see how to compute a numerical derivative of a grid of function values:

(a) Create a cell-edge grid with $N = 100$ on the interval $0 \le x \le 5$. Call the array `x`

(b) For the grid that you just created, create an array of function values for the following function: $\sin(x)\cosh(x)$. Call the array `y`.

(c) Using only a single line of code, evaluate the derivative of the function that you just created.

(d) Using only a single line of code, evaluate the second derivative of the function that you just created.

(e) Plot the original function and it's first derivative on the same graph.

If you did the problem right, you probably encountered a problem when you tried to plot the derivatives. That challenge arose because the derivative at the first and last points on the grid can't be calculated using Eqs. (4.2) and (4.3) since there aren't grid points on both sides of the endpoints. About the best we can do

is to extrapolate the interior values of the two derivatives to the end points. If we use linear extrapolation then we just need two nearby points, and the formulas for the derivatives at the end points are found using Eq. (3.3):

P4.3 (a) Use the linear extrapolation formulas (Eq. (3.3) and (3.4)) to calculate the value of the first and second derivative at the endpoints.[7]

(b) Re-plot the function and it's derivatives to verify that the extrapolations look right.

Note: There are two ways to add the extrapolated values into the array storing the derivative values.

- Build your array to be the right size at the beginning: This can be done using `numpy`'s `zeros_like` function. This will build an array of zeros. The middle entries of this array can be set using the derivative formula and the endpoints can be set with the extrapolation formulas.
- Build your array to be initially shorter than needed and later augment that array with the extrapolated values. This can be done using `numpy`'s `append` and `insert` functions.

[7] Rember that Python arrays are zero-indexed

Errors in the approximate derivative formulas

We'll conclude this lab with a look at where the approximate derivative formulas come from and at the types of the errors that pop up when using them. The starting point is Taylor's expansion of the function f a small distance h away from the point x

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots + f^{(n)}(x)\frac{h^n}{n!} + \cdots \tag{4.4}$$

Let's use this series to understand the forward difference approximation to $f'(x)$. If we apply the Taylor expansion to the $f(x+h)$ term in Eq. (4.1), we get

$$\frac{f(x+h)-f(x)}{h} = \frac{\left[f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots\right] - f(x)}{h} \tag{4.5}$$

The higher order terms in the expansion (represented by the dots) are smaller than the f'' term because they are all multiplied by higher powers of h (which we assume to be small). If we neglect these higher order terms, we can solve Eq. (4.5) for the exact derivative $f'(x)$ to find

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} - \frac{h}{2}f''(x) \tag{4.6}$$

From Eq. (4.6) we see that the forward difference does indeed give the first derivative back, but it carries an error term which is proportional to h. But, of course, if

h is small enough then the contribution from the term containing $f''(x)$ will be too small to matter and we will have a good approximation to $f'(x)$.

Now let's perform the same analysis on the centered difference formula to see why it is better. Using the Taylor expansion for both $f(x+h)$ and $f(x-h)$ in Eq. (4.2) yields

$$\frac{f(x+h)-f(x-h)}{2h} = \frac{\left[f(x)+f'(x)h+f''(x)\frac{h^2}{2}+f'''(x)\frac{h^3}{6}+\cdots\right]}{2h} - \frac{\left[f(x)-f'(x)h+f''(x)\frac{h^2}{2}-f'''(x)\frac{h^3}{6}+\cdots\right]}{2h} \tag{4.7}$$

If we again neglect the higher-order terms, we can solve Eq. (4.7) for the exact derivative $f'(x)$. This time, the f'' terms exactly cancel to give

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} - \frac{h^2}{6}f'''(x) \tag{4.8}$$

Notice that for this approximate formula the error term is much smaller, only of order h^2. To get a feel why this is so much better, imagine decreasing h in both the forward and centered difference formulas by a factor of 10. The forward difference error will decrease by a factor of 10, but the centered difference error will decrease by a factor of 100. This is the reason we try to use centered formulas whenever possible in this course.

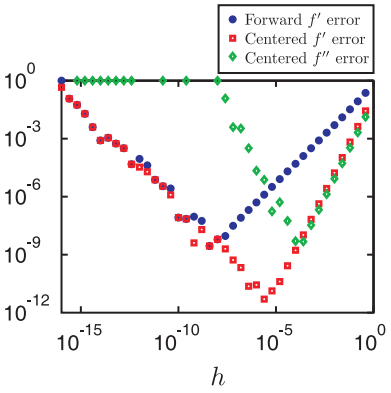


Figure 4.3 Error in the forward and centered difference approximations to the first derivative and the centered difference formula for the second derivative as a function of h. The function is e^x and the approximations are evaluated for $x = 0$.

P4.4 (a) Let's find the second derivative formula using an approach similar to what we did for the first derivative.

i. In Mathematica, write out the Taylor's expansion for $f(x+h)$ using Eq. (4.4), but change the derivatives to variables that Mathematica can do algebra with, like this:

```
eqOne = fplus == f + fp*h + fp2*h^2/2 + fp3*h^3/6 + fp4*h^4/24
```

where `fp` stands for f', `fp2` stands for f'', etc.

ii. Make a similar equation called `eqminus` for $f(x-h)$ that contains the same derivative variables `fp`, `fpp`, etc.

iii. Now use the `Solve` command to solve these two equations for the first derivative `fp` and the second derivative `fpp`.

iv. Verify that the first derivative formula matches Eq. (4.8), including the error term, and that the second derivative formula matches Eq. (4.3), but now with the appropriate error term.

v. What order is the error in terms of the step size h?

P4.5 (a) Construct a loop that calculates the forward first derivative, centered first derivative, and centered second derivative of the function $f(x) = e^x$ at $x = 0$ for various values of the step size. Let the step size range

from $h = 1$ to $h = 1 \times 10^{-5}$ decreasing by powers of 2 at each iteration. Calculate the error of each calculation and save them to lists.

Note that at $x = 0$ the exact values of both f' and f'' are equal to 1, so just subtract 1 from your numerical result to find the error.

(b) Make a log-log plot of error vs. step size for all three lists.

(c) By looking at the plot, verify that the error estimates in Eqs. (4.6) and (4.8) agree with the numerical testing.

In problem 4.5, you should have found that $h = 0.001$ in the centered-difference formula gives a better approximation than $h = 0.01$. These errors are due to the finite grid spacing h, which might entice you to try to keep making h smaller and smaller to achieve any accuracy you want. This doesn't work. Figure 4.3 shows a plot of the error you calculated in problem 4.5 as h continues to decrease (note the log scales). For the larger values of h, the errors track well with the predictions made by the Taylor's series analysis. However, when h becomes too small, the error starts to increase. Finally (at about $h = 10^{-16}$, and sooner for the second derivative) the finite difference formulas have no accuracy at all—the error is the same order as the derivative.

The reason for this behavior is that numbers in computers are represented with a finite number of significant digits. Most computational languages (including Python) use a representation that has 15-digit accuracy. This is normally plenty of precision, but look what happens in a subtraction problem where the two numbers are nearly the same:

$$\begin{array}{rr} & 7.38905699669556 \\ - & 7.38905699191745 \\ \hline & 0.00000000477811 \end{array} \tag{4.9}$$

Notice that our nice 15-digit accuracy has disappeared, leaving behind only 6 significant figures. This problem occurs in calculations with real numbers on all digital computers, and is called *roundoff*. You can see this effect by experimenting with the Python command

```
h = 1e-17
diff = (1+h)
print(diff-1)
```

for different values of h and noting that you don't always get h back. Also notice in Fig. 4.3 that this problem is worse for the second derivative formula than it is for the first derivative formula. The lesson here is that it is impossible to achieve arbitrarily high accuracy by using arbitrarily tiny values of h. In a problem with a size of about L it doesn't do any good to use values of h any smaller than about $0.0001L$.

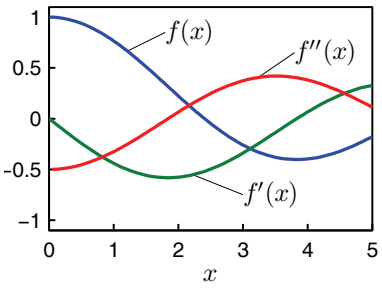


Figure 4.4 Plots from 4.6

Homework Problems

H4.6 Create a cell-edge grid with $N = 100$ on the interval $0 \le x \le 5$. Load $f(x)$ with the Bessel function $J_0(x)$ and numerically differentiate it to obtain $f'(x)$ and $f''(x)$. Use both linear and quadratic extrapolation to calculate the derivative at the endpoints. Compare both extrapolation methods to the exact derivatives and check to see how much better the quadratic extrapolation works. Then make overlaid plots of the numerical derivatives with the exact derivatives:

$$f'(x) = -J_1(x)$$

$$f''(x) = \frac{1}{2}\left(-J_0(x) + J_2(x)\right)$$

Note: Bessel functions can be found in `scipy.special` library. Here is the usage:

```
from scipy.special import jv

f = jv(order,domain)
```

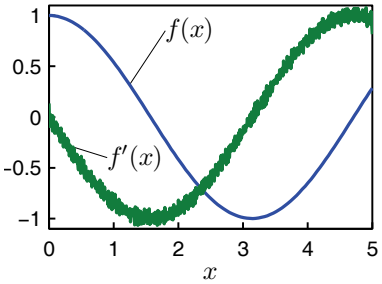


Figure 4.5 Plots of $f(x)$ and $f'(x)$ from 4.7 with 1000 points. $f''(x)$ has too much error to make a meaningful plot for this number of points.

H4.7 Finally, let's learn some wisdom about using finite difference formulas on experimental data. Suppose you had acquired some data that you needed to numerically differentiate. Since it's real data there are random errors in the numbers. Let's see how those errors affect your ability to take numerical derivatives. Make a cell-edge grid for $0 \le x \le 5$ with 1000 grid points. Then model some data with experimental errors in it by using Python's random number function `rand` like this:

```
from numpy import cos
from numpy.random import uniform

f=cos(x)+.001*uniform(0,1,len(x))
```

So now f contains the cosine function, plus experimental error at the 0.1% level. Calculate the first and second derivatives of this data and compare them to the "real" derivatives (calculated without noise). Reduce the number of points to 100 and see what happens.

Differentiating your data is a bad idea in general, and differentiating twice is even worse. If you can't avoid differentiating your data, you had better work pretty hard at reducing the error, or perhaps fitting your data to a smooth function, then differentiating it.

Chapter 5

Overview

Most of your experience with math and physics to this point in your education has involved the use of analytical techniques. In other words, you use your pencil and paper (and occasionally Mathematica) to perform algebra and calculus until you arrive at the solution, a solution that you can write down on paper. It turns out that there are a whole bunch of real-world problems that have no analytical solution: you can't write down the solution on a piece of paper. In these cases, we must use a numerical approach. What that means exactly is probably not very clear at this point, but it will soon.

More specifically, this class will focus on numerical solutions to differential equations. It seems prudent to begin our study with a broad overview of differential equations. What are they? What are some common ones that arise in physics? Can we categorize them into groups?

What is a differential equation?

So what is a differential equation? When we say the word equation, we probably think of something like this

$$5x^2 + 3x - 2 = 0. \tag{5.1}$$

"Solving" this equation entails performing some algebra until x is isolated and the solution is then found on the other side of the equals sign. In this case, the algebra would lead to the following solution

$$x = -1 \text{ or } x = \frac{2}{5} \tag{5.2}$$

Notice that the solution to this equation was a *number*. Now let's look at a differential equation:

$$m\frac{d^2y(t)}{dt^2} + c\frac{dy(t)}{dt} + ky(t) = 0 \tag{5.3}$$

This equation emerges when you write down Newton's second law for a damped oscillator. Unlike the solution to equation (5.1), the solution to this equation is not a number, it's a function. It turns out that one solution[8] to this differential equation is

$$y(t) = Ae^{-\frac{c}{2m}t}\sin(\omega t + \phi) \tag{5.4}$$

where

$$\omega = \sqrt{\frac{k}{m} - \frac{c^2}{4m^2}} \tag{5.5}$$

P5.1 Verify, using Mathematica (or by hand), that this equation really does satisify the differential equation in eq (5.3).

[8] This is the solution to the underdamped case where $\frac{c}{2m} \ll \sqrt{\frac{k}{m}}$

How do we categorize differential equations?

Differential equations are categorized based on the highest derivative found in the equation and on the number of independent variable that the solution function has. For example, notice that the solution to equation (5.3) was a function of *one* variable: t. Differential equations whose solutions are functions of one and only one variable are called **ordinary differential equations**. They are ordinary because the differential equation only has ordinary derivatives and no partial derivatives.

If the solution function has multiple dependent variables, then the differential equation could involved derivatives with respect to each of them. We call these partial derivatives and therefore the differential equation is called a partial differential equation because. Here is an example of a partial differential equation

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0 \tag{5.6}$$

Notice the ordinary derivative ($\frac{d}{dt}$) in equation (5.3) replaced with the partial derivative symbol ($\frac{\partial^2}{\partial t^2}$) in equation (5.6). The solution to this equation is a function of two variables: $u(x,t)$.

Boundary and Initial Conditions

It turns out that the differential equation alone is not sufficient to fully determine the solution. To see why this is so, let's take a very simple differential equation:

$$\frac{dy}{dt} = -\frac{k}{y} \tag{5.7}$$

This equation can be solved by rearranging so that all ys and dys are on one side and all ts and dts are on the other:

$$ydy = -cdt \tag{5.8}$$

Now integrating both sides:

$$\int ydy = -\int kdt \tag{5.9}$$

$$\frac{y^2}{2} + C_1 = -kt + C_2 \tag{5.10}$$

$$\frac{y^2}{2} = -kt + C_3 \tag{5.11}$$

$$y^2 = -2kt + 2C_3 \tag{5.12}$$

$$y = \sqrt{-2kt + C_4} \tag{5.13}$$

$$\tag{5.14}$$

The integrating constant C_4 must be determined to finish the solution. This constant is the initial value of the function, or the value of the function at the initial time ($t = 0\text{s}$). Without this information, the solution is not complete. In general, the number of extra pieces of information needed to solve a differential equation is equal to the order of the differential equation. For example, to solve the following second-order ordinary differential equation

$$\frac{d^2y}{dt^2} = -g \tag{5.15}$$

would require two extra pieces of information, usually the initial position and initial velocity (derivative of position), would be needed to solve the equation.

There are two types on "extra" information that are typically used: initial conditions and boundary conditions.

A second-order differential equation as two first-orders

Let's consider a familiar differential equation corresponding to a one-dimensional projectile experiencing a constant acceleration:

$$\frac{d^2y}{dt^2} = -g \tag{5.16}$$

We call it a second order equation because the highest derivative found is of second order. A second order differential equation can always be written as two first order equations by using an intermediate variable. For example by defining v:

$$v = \frac{dy}{dt} \tag{5.17}$$

we can now write equation (5.16) as:

$$v = \frac{dy}{dt} \qquad \text{and} \qquad -g = \frac{dv}{dt} \tag{5.18}$$

which is a set of *coupled, first-order* differential equations.

Part II

Ordinary Differential Equations

Chapter 6

Euler's Method

Today we get our first taste of what it means to solve a differential equation numerically. In fact, most of our time for the remainder of the semester will be some form of this task: finding a numerical solution to all kinds of differential equations. We begin with an ordinary differential equation and we'll use a familiar (albeit inaccurate) technique to solve it: Euler's method. You should have seen this method in PH150 and possibly in other classes.

Solving Differential Equations Numerically

Last time we mastered the art of taking derivatives on a grid. Now, let's see how we can use that knowledge to numerically solve differential equations. Consider the motion of a projectile near the surface of the earth with no air resistance. The differential equations that describe the projectile are

$$\frac{d^2x}{dt^2} = 0 \qquad \frac{d^2y}{dt^2} = -g \tag{6.1}$$

along with some initial conditions, $x(0)$, $y(0)$, $v_x(0)$, and $v_y(0)$. [9] These are two, second-order, ordinary differential equations. [10] Of course, we can always write a second-order differential equation as a set of two, coupled, first-order differential equations. Let's do that now:

$$\begin{aligned} \frac{dx}{dt} &= v_x & \frac{dy}{dt} &= v_y \\ \frac{dv_x}{dt} &= 0 & \frac{dv_y}{dt} &= -g \end{aligned} \tag{6.2}$$

This set of equations is easily solved analytically, but imagine that we didn't have an analytic solution. How could we numerically model the motion of the projectile?

The basic idea in a numerical solution for a system like this is to think of time as being a discrete grid rather than a continuous quantity. It is easiest to have an evenly spaced time grid $[t_0, t_1, t_2, ...]$ with $t_0 = 0$, $t_1 = \tau$, $t_2 = 2\tau$, etc. We label the variables on this time grid using the notation $x_0 \equiv x(0)$, $x_1 \equiv x(\tau)$, $x_2 \equiv x(2\tau)$, etc. To formulate a numerical solution to these equations, we first must write the derivatives using the (inaccurate) forward difference approximation of the derivative that you learned about in the reading:

$$\begin{aligned} \frac{x_{n+1} - x_n}{\tau} &= v_{x,n} & \frac{y_{n+1} - y_n}{\tau} &= v_{y,n} \\ \frac{v_{x,n+1} - v_{x,n}}{\tau} &= 0 & \frac{v_{y,n+1} - v_{y,n}}{\tau} &= -g \end{aligned} \tag{6.3}$$

[9] Remember, since they are second-order differential equations, I must have two extra pieces of information to solve them.

[10] It's really important that you are able to identify the type of differential equation that you are solving. The numerical method that is most appropriate depends on it.

Notice that the left sides of these equations are centered on the time $t_{n+1/2}$, but the right sides are centered at time t_n. This makes this approach inaccurate[11], but if we make τ small enough it can work well enough to see the principles involved.

[11] Fixing this problem will be our topic for next time.

By solving the equations in (6.3) we can obtain a simple algorithm for stepping our solution forward in time:

$$\begin{aligned} x_{n+1} &= x_n + v_{x,n}\tau & y_{n+1} &= y_n + v_{y,n}\tau \\ v_{x,n+1} &= v_{x,n} & v_{y,n+1} &= v_{y,n} - g\tau \end{aligned} \tag{6.4}$$

This method of approximating solutions is called Euler's method. In general, it's not very good, especially over many time steps. However, it provides a foundation for learning other better methods.

☞ The name Euler does not rhyme with "cooler"; it rhymes with "boiler". You will impress your fellow students and your professors if you give this important name from the history of mathematics its proper pronunciation.

P6.1 Make a program in Python to model the motion of a ball bouncing on the floor using Euler's method. In your script, define the initial position of the ball with `x=0` and `y=1`, and the initial velocity with `vx=1` and `vy=0`. Then write a while loop to step the position and velocity forward in time using Eq. (6.4). Have your while loop exit when the ball has traveled 10 meters horizontally.

(a) To simulate bouncing, put an `if` statement in your loop that checks if `y` is less than zero. When it is, make `vy` positive like this

```
vy=abs(vy)
```

Make a movie by plotting the position of the ball as a dot each time the loop iterates, like this:

```
pyplot.scatter(x[-1],y[-1])
pyplot.xlim(0,10)
pyplot.ylim(0,1.2)
pyplot.draw()
```

(b) Our bouncing condition in part (a) is lousy. Make it better by adding some more logic that does the following:

(i) Test to see if `y` will go less than zero on this time step, but don't actually change `y` yet.

(ii) If `y` won't go less than zero this step, just do a regular Euler step.

(iii) If it will go negative this time step, determine a smaller time step τ_1 such that an Euler step will take the ball to $y = 0$. Then take an Euler step with τ_1. After taking this small step, make the y-velocity positive as before

```
vy=abs(vy)
```

and then take an Euler step of $\tau_2 = \tau - \tau_1$ to finish off the time interval. Play with different values of τ and notice the behaviour of the simulation.

(c) Make your model look more realistic by adding some loss to the code that represents the bounce process, like this

```
vy=0.95*abs(vy)
```

This damping will mask the growth of Euler's method for a suitably small τ.

You should have noticed that no matter what we tried, Euler's method is always unstable (i.e. the amplitude of the bounce continues to grow with each subsequent bounce). This is an inadequacy of Euler's method and must be addressed at some point. For now, you should remember that Euler's method does not conserve energy. Stay tuned for a method that does conserve energy.

P6.2 You may have heard that it's easier to hit a homerun in Denver than in Atlanta. Let's use Euler's method to simulate the motion of a batted baseball and investigate the claim.

(a) We'll need to include the effect of air drag for this situation so let's think about this force for a second. The force of drag is given by:

$$\mathbf{F} = \frac{1}{2}\rho A C v^2 \frac{\mathbf{v}}{|\mathbf{v}|} \tag{6.5}$$

The first thing we need to do is decompose this force into it's components. Look at figure 6.1 to convince yourself that the equations below are the correct components.

$$F_x = \frac{1}{2}\rho A C v v_x \tag{6.6}$$

$$F_y = \frac{1}{2}\rho A C v v_y \tag{6.7}$$

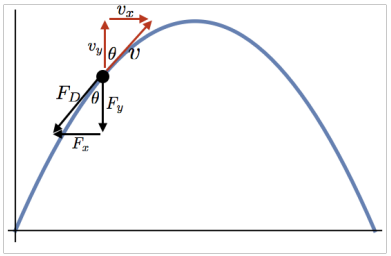


Figure 6.1 Illustration of the decomposition of the drag force.

(b) Add the force of air drag into your Euler's model and plot the trajectory of a baseball($C = 0.5$) batted at an angle of 35° with an initial speed of 49 m/s (110 mph). Use google to lookup the dimensions and mass of a baseball and choose the air density corresponding to sea level ($\rho = 1.29$). Play with the air density and cross-sectional area to convince yourself that you did it correctly.

(c) Now make two plots[12]: one trajectory for the ball batted in Denver ($\rho = 0.96$) and another for the ball batted in Atlanta ($\rho = 1.22$). Determine the difference in ranges for these two batted balls. Would you rather bat in Denver or Atlanta?

(d) It turns out that the drag coefficient (C) of a baseball varies with the speed of the ball. At low speed $C \approx 0.5$ but at higher speed as the air

[12] This should be easy if you have designed your class structure well

flow becomes turbulent, the drag coefficient decreases. The following formula is a good approximation[1]

$$C(v) = .198 + \frac{.295}{1 + e^{\frac{v-35}{5}}} \tag{6.8}$$

Incorporate this change into your code and determine how it affects the difference in ranges between Atlanta and Denver.

Homework

H6.3 Let's use Euler's method to model the motion of a cannon shell fired to a very high altitude.

(a) Use your code from the previous problem as a starting point for this problem. [13] Choose a launch angle $0° < \theta < 90°$ and set the launch speed to be 750 m/s. Choose $C = 0.5$, $A = 0.007$ m^2, $\rho = 1.29$ kg/m^3, and $m = 50$ kg. Plot the trajectory.

[13] You'll want to make a copy so there are two seperate files.

(b) In part (a) you should have found that the maximum altitude of the cannon shell is on the order of kilometers. At these altitudes, the density of the air changes appreciably. We can model the change in air density with the following function:

$$\rho = \rho_0 \left(1 - \frac{ay}{T_0}\right)^{\alpha} \tag{6.9}$$

with $a \approx 6.50 \times 10^{-3}$ K/m, T_0 is the sea-level temperature and the exponent $\alpha \approx 2.5$. Plot this function and verify that it behaves as you would expect.

(c) Now incorporate the changing air density into your Euler's model and plot the resulting trajectory.

(d) Plot the constant-air density trajectory on top of the trajectory that accounts for the variation to see how big of an affect it is. (Hint: Since you coded this using a class, this should be no big deal. Just initiate a second instance of this class and turn off the density correction function.)

---

[1] **giordano1997computational**.

Chapter 7

Second-order Runge-Kutta

Last time we got our first taste of solving an ordinary differential equation using Euler's method. We admitted that Euler's method is not very accurate but we were content to have something to work with. Our goal today will be to improve upon Euler's method and to explore just how good our improvement is.

Second Order Runge-Kutta

Let's start by recalling how we got to Euler's method last time. Consider a general single, first-order, ordinary differential

$$\frac{dx}{dt} = f(x,t) \tag{7.1}$$

Where $f(x,t)$ is any function of x and t. Euler's method consists of first writing the derivative in equation (7.1) with the forward-difference numerical approximation to get:

$$\frac{x_{n+1} - x_n}{\tau} = f(x_n, t_n) \tag{7.2}$$

and rearranging to solve for x_{n+1}:

$$x_{n+1} = x_n + f(x_n, t_n)\tau \tag{7.3}$$

One might suspect that a good way to improve the accuracy would be to include more terms from the Taylor expansion when we write down the numerical derivative on the left hand side of equation (7.2) (see equation (4.6) to remind yourself). However, these extra terms involve second and higher derivatives ($x''(t)$, $x'''(t)$, etc) and it's most likely that we won't have any information about those derivatives which means this won't help much.

Recall that one reason for the inaccuracies in Euler's method is the fact that the right hand side of equation (7.2) is centered at t_n and the left hand side is centered at $t_{n+\frac{1}{2}}$. Using the centered-difference version of the derivative would fix this and equation (7.2) would become:

$$\frac{x_{n+1} - x_{n-1}}{2 \times \tau} = f(x_n, t_n) \tag{7.4}$$

Solving for x_{n+1}:

$$x_{n+1} = x_{n-1} + 2 \times \tau \times f(x_n, t_n) \tag{7.5}$$

or, equivalently:

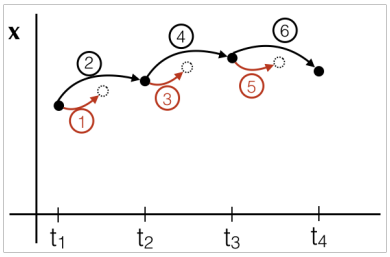


Figure 7.1 Illustration of the second-order Runge Kutta algorithm.

$$x_{n+1} = x_n + \tau \times f(x_{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) \tag{7.6}$$

This looks exactly like Euler's method except that instead of evaluating the right hand side of the differential equation at x_n, t_n, we'd like to evaluate it at $x_{n+\frac{1}{2}}, t_{n+\frac{1}{2}}$. We could have probably guessed this from the outset since the objective was to have both sides of the differential equation centered at the same point.

One question remains: How will we calculate $x_{n+\frac{1}{2}}$ so that we can use it to evaluate $f(x_{n+\frac{1}{2}}, t_{n+\frac{1}{2}})$. As contrary as it may sound, we choose to use an Euler's step for this calculation. This means that each time step is a two-step process: (i) first we advance half a time step using Euler's method and then (ii) use the result of that half step to make the full time step by evaluating equation (7.6) as depicted in figure 7.1. In other words:

$$x_{n+\frac{1}{2}} = x_n + \frac{\tau}{2} f(x_n, t_n) \quad \text{(Euler's step)} \tag{7.7}$$

$$x_{n+1} = x_n + \tau f(x_{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) \quad (2^{\text{nd}} \text{ Order Runge} - \text{Kutta}) \tag{7.8}$$

We can write this algorithm in a way that turns out to be a little easier to implement. It looks like this. [14]

[14] Pay close attention during practice problem 2 to see why.

$$k_1 = \tau f(x_n, t_n) \tag{7.9}$$

$$k_2 = \tau f(x_n + \frac{1}{2}k_1, t_n + \frac{1}{2}\tau) \tag{7.10}$$

$$x_{n+1} = x_n + k_2 \tag{7.11}$$

You should take a minute to verify that these equations are equivalent to equations (7.7)

P7.1 You should recall that Euler's method did not conserve energy. (remember the bouncy ball that gained height the more it bounced). Now that we have a new method, the first thing we ought to investigate is whether it conserves energy. Let's investigate using a simple pendulum. The differential equation for the simple pendulum is:

$$\frac{d^2 s}{dt^2} = -g \sin\theta \tag{7.12}$$

Notice that the left hand side involves x, and the right hand side involves θ. We need to pick one or the other and stick with it. Let's choose to use angular quantities.

(a) Rewrite the differential equation in eq (7.12) so that the left hand side involves θ, not x.

(b) Express the second order differential equation as a set of two first order equations.

(c) Now use 2nd order Runge-Kutta to make a plot of θ vs. time for several cycles of the motion. At $t = 0$ set $\theta = \frac{\pi}{4}$ and $\omega = 0$.

(d) Does 2nd order Runge-Kutta conserve energy?

In the above problem you should have found that the 2nd order Runge-Kutta method [15] does not conserve energy just like Euler's method didn't.

[15] Note: Runge-Kutta is prounounced "Roong-uh Koo-tah", not "Roonja-Cutta"

Convergence

If the Runge-Kutta algorithm doesn't conserve energy, then there has to be something else about it that makes it better than Euler. As you will soon see, the RK algorithm converges more quickly than the Euler algorithm. This means that we can use a larger value of τ with RK and achieve a higher level of accuracy than we could obtain with Euler.

P7.2 To compare the rate of convergence between RK and Euler, let's return to the batted-ball problem (problem 6.2).

(a) Make a copy of your batted-baseball code from problem 6.2

(b) Add a member function that use 2nd order Runge-Kutta to find the trajectory of the baseball.

(c) To investigate the convergence behavior of this method, construct a loop over step sizes starting at $\tau = 1$, ending at $\tau = 1 \times 10^{-5}$. Decrease the step size by a factor of 2 at each iteration. Each time through the loop, use 2nd order Runge-Kutta to calculate the range of the particle and save that range to a list. When you are done, plot the ranges vs. step size with the horizontal axis scaled logarithmically.

(d) Do the same thing using Euler's method and overlay the two plots.

(e) How do the convergences compare?

Error on 2nd-order Runge-Kutta

It's important to quantify the error associated with any numerical method. To do this, Let's use a Taylor expansion to estimate a $x(t+\tau)$.

$$x(t+\tau) = x(t) + \tau x'(t) + \frac{1}{2}\tau^2 x''(t) + \frac{1}{6}\tau^3 x'''(t) + \frac{1}{24}\tau^4 x''''(t) \cdots + x^{(n)}(t)\frac{\tau^n}{n!} \quad (7.13)$$

[16] Mathematicians often have the benefit of hindsight when presenting a set of mathematical steps like this. If you feel like you couldn't have done this from scratch, don't feel bad. The mathematicians who derived it orignally no doubt tried several things before arriving at the desired result.

Next, let's use the Taylor series to estimate $x(t-\tau)$ [16]

$$x(t-\tau) = x(t) - \tau x'(t) + \frac{1}{2}\tau^2 x''(t) - \frac{1}{6}\tau^3 x'''(t) + \frac{1}{24}\tau^4 x''''(t) \cdots + x^{(n)}(t)\frac{\tau^n}{n!} \quad (7.14)$$

Next, let's subtract equation (7.14) from equation (7.13):

$$x(t+\tau) - x(t-\tau) = 2x'(t)\tau - \frac{1}{3}\tau^3 x'''(t)\tau^3 + \cdots \quad (7.15)$$

and now let's solve for $x(t+\tau)$:

$$x(t+\tau) = x(t-\tau) + 2x'(t)\tau - \frac{1}{3}\tau^3 x'''(t) \cdots \quad (7.16)$$

which is equivalent to

$$x(t+\tau) = x(t) + x'(t+\frac{1}{2}\tau)\tau - \frac{1}{3}\tau^3 x'''(t+\frac{1}{2}\tau) \cdots \quad (7.17)$$

or, in index notation, this equation can be written as:

$$x_{n+1} = x_n + f(x_{n+\frac{1}{2}}, t_{n+\frac{1}{2}})\tau - \frac{1}{3}\tau^3 f'(x_{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) \cdots \quad (7.18)$$

Comparing this with equation (7.6), we can see that the 2^{nd}-order Runge-Kutta is neglecting terms of order τ^3 and higher. This is better than Euler, which neglects terms of order τ^2 and higher. In other words, 2^{nd}-order Runge Kutta is good up to second order, which is why we call it a 2^{nd}-order method.

Homework

H7.3 (**LaTeX Problem**) It is well-known that baseballs and golfballs can be made to curve in directions perpendicular to their direction of motion. The force that causes this deflection is called the Magnus force and is due to differences in the force of air drag between opposite sides of the ball. The Magnus force can be calculated as:

$$F_M = S_0 \omega \times \mathbf{v} \quad (7.19)$$

where S_0 is 6.11×10^{-5} kg

(a) Use second-order Runge Kutta to model a major-league baseball pitch. You'll need to look up the mass of a baseball, and a typical throwing speed for an MLB pitcher. Assume that the pitcher throws the pitch exactly horizontal. You may assume that the air density remains constant during the flight of the ball.

(b) Now incorporate the Magnus force into your simulation. Assume that the pitcher throws the ball with side-spin with a rate of rotation equal to $\omega = 30$ revolutions/second.

Please note that including the Magnus force requires that you track all three coordinates of the position and velocity. Let +x be the direction from pitcher to batter, +y be the vertical direction, and +z be perpendicular to both of them. You'll need to add in the necessary code to calculate the z-coordinates of the ball's position and velocity.

Hint: `numpy` has a function called `cross` that might help you calculate the Magnus force.

(c) Make a plot of x vs. z for the baseball (horizontal plane) to see how big of a deflection from straight-line motion occurs.

(d) Now assume that the pitcher gives the ball top spin. In what direction will the Magnus force point now. Make a plot of x vs y for the baseball (vertical plane) and investigate.

Chapter 8

Fourth-order Runge-Kutta and LeapFrog

So far we have used Euler's method and second-order Runge-Kutta. Neither of these methods conserve energy, but we have seen that second-order RK is more accurate than Euler. Today we will improve upon the second-order method and finally learn a method that conserves energy.

Fourth-Order Runge-Kutta

Recall that one way to arrive at the equations for second-order Runge-Kutta is by performing a Taylor expansion to estimate forward in time and subtracting from it a Taylor expansion that estimates backwards in time (see section 28). This results in the cancelation of higher order terms (h^2 in the case of second-order RK) and hence leads to a more accurate method. By performing more Taylor expansions about different points and strategically adding/subtracting them we can force even higher order terms to cancel out and hence arrive at better and better numerical methods. This comes at the expense of greater and greater algorithmic complexity.

The fourth-order Runge-Kutta algorithm is widely accepted to be the best balance between numerical accuracy and algorithmic complexity. We won't give all of the mathematical detail here although I hope my description in the preceding paragraph gives you a good idea of how we might do it. For us it will suffice to state the equations without proof:

$$k_1 = \tau f(x_n, t_n) \tag{8.1}$$

$$k_2 = \tau f(x_n + \frac{1}{2}k_1, t_{n+\frac{1}{2}}) \tag{8.2}$$

$$k_3 = \tau f(x_n + \frac{1}{2}k_2, t_{n+\frac{1}{2}}) \tag{8.3}$$

$$k_4 = \tau f(x_n + k_3, t_{n+1}) \tag{8.4}$$

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)) \tag{8.5}$$

If your implementation of second-order Runge-Kutta from last time was the more elegant version, then modifying your code for this fourth-order method is very simple. You'll only need to add lines (8.3) and (8.4) and modify line (8.5)

P8.1 Let's investigate how much better this approach is compared to Euler's and second-order RK.

(a) Make a copy of your code from problem 7.2.

(b) Add a function that uses the fourth-order Runge-Kutta method to calculate the trajectory of the batted ball. This function will be nearly identical to your elegant version of second-order RK, with only a few modifications/additions.

(c) In the loop over step sizes add a few lines to call the fourth-order function and save the range that it predicts.
Note: To obtain an exact replica of figure 8.1, let $dt = 0.7$ initially and decrease by factors of 1.5 in the loop. Also, use the following initial conditions:

$$v = 400 \text{ m/s} \qquad m = 0.145 \text{ kg}$$
$$\theta = 45^\circ \qquad C = 0.5$$
$$d = 0.075 \text{ m} \qquad \rho = 1.22 \text{ kg/m}^3$$

(d) Plot range vs. step size for all three methods that we have learned so far (Euler, Second-order RK, and Fourth-order RK). Use Logarithmic scales for the horizontal axis and modify the vertical size of the plot so as to capture a 20 meter window centered on the range. Your plot should look something like figure 8.1

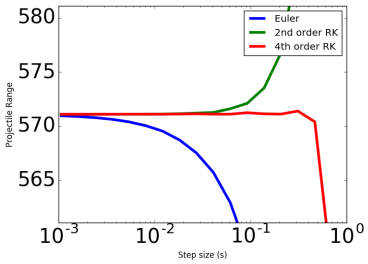


Figure 8.1 Comparison of the rate of convergence for Euler's method, second-order Runge-Kutta, and fourth-order Runge-Kutta.

As you can see in figure 8.1, each method gets successively more accurate. For example, you would have to use $dt = 0.05$ with second-order RK to get results of the same accuracy as you would get with $dt = 0.5$ with fourth-order Runge-Kutta. In other words, a fourth-order Runge-Kutta approach could achieve higher-quality results using less computational resources and time. It's just better.

The leapfrog method

It's now time to learn a method that conserves energy. It's worth mentioning that even though the methods that we have learned do not conserve energy this doesn't mean that they aren't useful. There are plenty of situations where the rate at which the energy increases will be so small that it won't affect your results appreciable. Generally speaking, you'll want your numerical method to conserve energy for motion that is repeating or that runs for long periods of time.

The leapfrog method is very similar to second-order Runge-Kutta. We start by stepping forward one time step using the RK2 algorithm:

$$x_{n+\frac{1}{2}} = x_n + \frac{\tau}{2} f(x_n, t_n) \quad \text{(Euler's step)} \tag{8.6}$$
$$x_{n+1} = x_n + \tau f(x_{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) \quad (2^{\text{nd}} \text{ Order Runge} - \text{Kutta}) \tag{8.7}$$

At this point, the Runge-Kutta method uses x_{n+1} with an Euler step to perform the next half step.

$$x_{n+\frac{3}{2}} = x_{n+1} + \frac{\tau}{2} f(x_{n+1}, t_{n+1}) \tag{8.8}$$

$$\tag{8.9}$$

The algorithm continues this pattern, using an Euler step followed by full step at each iteration. We've already learned that Euler steps should be avoided if we can help it. We obviously needed an Euler step to make our first half step but it might be possible to avoid using Euler to make the remaining half steps. What if instead we used the previous half step to calculate the next half step. In other words:

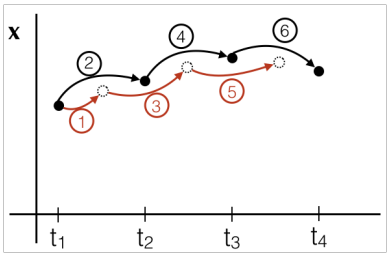


Figure 8.2 Diagram of the leapfrog method for solving first-order differential equations.

$$x_{n+\frac{3}{2}} = x_{n+\frac{1}{2}} + \tau f(x_{n+1}, t_{n+1}) \tag{8.10}$$

$$\tag{8.11}$$

This would be a better approach since the derivative is being evaluated at the halfway point (i.e. Both sides of this equation are centered at the same point.) The remainder of the steps are all full time steps alternating between the midpoint calculations and the on-grid calculations. A diagramatic depiction of the leapfrog method is given in figure 8.2

P8.2 Let's use leapfrog to study the orbit of the earth around the sun.

(a) Use figure 8.3 to convince yourself that the components of gravity on the earth are given by:

$$\frac{d^2x}{dt^2} = -\frac{GM_s r_x}{r^3} \tag{8.12}$$

$$\frac{d^2y}{dt^2} = -\frac{GM_s r_y}{r^3} \tag{8.13}$$

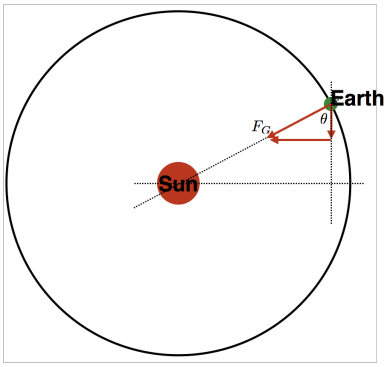


Figure 8.3 Decomposition of the graviational forces on the Earth

(b) The fourth-order Runge-Kutta algorithm is for first-order differential equations, not second order ones. So you'll need to write each of these second-order differential equations as two, coupled, first-order equations. Convince yourself that the equations below are the correct set:

$$\frac{dv_x}{dt} = -\frac{GM_s r_x}{r^3} \qquad \frac{dv_y}{dt} = -\frac{GM_s r_y}{r^3} \tag{8.14}$$

$$\frac{dx}{dt} = v_x \qquad \frac{dy}{dt} = v_y \tag{8.15}$$

(c) Use the leapfrog method to calculate the trajectory of Earth. Appropriate length and time scales for this problem are AU (astronomical units) and years. It turns out that $GM_S = 4\pi^2$ in these units. Note also that the distance from the sun to the earth is 1 AU.

(d) Add some code that will calculate the total energy of earth throughtout it's orbit. Make a plot of total energy vs. time. What did you learn?

(e) What if gravity was not an inverse square law? What if the force of gravity was:

$$F_g = \frac{GM_sM_e}{r^{2.5}} \tag{8.16}$$

Modify your code to investigate this. To see the effect you should modify your intial conditions so that the orbit is eliptical. You can do this by decreasing your initial velocity by a small amount.

(f) Investigate what Earth's orbit would look like if gravity was an inverse-cube law.

You should have noticed that although the total energy does fluctuate during the orbit, it always returns to the same value once the orbit returns to it's starting point. An algorithm like this would be a better choice if you need a solution over long time intervals. In contrast, the energy would slowly increase if we used a method like Runge-Kutta.

Homework

H8.3 Investigate the effect of Jupiter on Earth's orbit. Here are some steps to follow:

(a) You'll need to track the motion of both Earth and Jupiter. This means that you'll need to consider more forces:

i. Gravity of Sun on Earth
ii. Gravity of Sun on Jupiter
iii. Gravity of Jupiter on Earth
iv. Gravity of Earth on Jupiter

Add the necessary forces into your function `getDervis` (or whatever you called it)

(b) Previously, when you calculated the force due to the Sun on the earth, you did:

$$\frac{GM_sM_e}{r^2} = M_e a \tag{8.17}$$

$$\frac{GM_s}{r^2} = a \tag{8.18}$$

$$\frac{4\pi^2}{r^2} = a \tag{8.19}$$

$$\tag{8.20}$$

When you calculate the force between Earth and Jupiter, the mass of the sun does not appear. However, so that you can continue to use the $GM_s = 4\pi^2$ you can do something like:

$$\frac{GM_J M_e}{r^2} = M_e a \tag{8.21}$$

$$\frac{GM_J}{r^2} = a \tag{8.22}$$

$$\frac{GM_S}{r^2}\frac{M_J}{M_S} = a \tag{8.23}$$

$$\frac{4\pi^2}{r^2}\frac{M_J}{M_S} = a \tag{8.24}$$

$$\tag{8.25}$$

(c) The period of Jupiter's orbit is 11.86 years and it's distance from the sun is ≈ 5.2 AU. Jupiter's mass is 1.90×10^{27} kg. Use this information to initialize the position and velocity of Jupiter.

(d) Plot the orbit of both planets for 12 years and observe the effect on Earth's orbit. To really see the effect you'll have to zoom in on Earth's orbit. You can turn the effect of Jupiter on and off by changing the mass of Jupiter to a small number

(e) How would Earth's orbit change if Jupiter's mass was bigger by a factor of 10?, 100?, 1000?

Chapter 9

Chaos

Today we'll not introduce a new numerical method. Rather we'll use the methods that we've already learned to investigate the topic of chaos. People often associate chaos or chaotic systems with undeterminism, or a system (object) whose future state cannot be determined from information about its past state. **This is not what chaos is.** As we see some examples in action, hopefully you will start to get a feel for the true definition of chaos. But just in case you need a formal definition to fall back on, here it is: **A chaotic system is one that is extremely sensitive to initial conditions**. In fact, it's so sensitive to the I.C. that it's virtually unpredicatable.

Let me give an example to illustrate. Consider the following conversation between two scientists:
Scientist #1: I'm going to launch a cannon shell at a 45° angle at a speed of 400 m/s. Can you predict where it will land?
Scientist #2: Sure, I'll just use Euler's method and calculate the trajectory. Your cannon shell will land exactly 25.1 kilometers away.
Scientist #1: Great! Thanks. What if I changed the launch angle to 45.01°. How would that change the landing location?

This last question should have struck you as odd. Surely this scientist knows that changing the launch angle by .01° can't affect the trajectory that much, right? The landing location will still be very close to 25.1 km? At least close enough that you wouldn't need to recalculate the trajetory. Suprisingly, the answer to this question is a clear and definitive"**No**" if the system is chaotic. An appropriate response from scientist #2 if the system is chaotic would be:
Scientist #2: I have no idea where your cannon shell will land. It could land anywhere. If you're going to change the initial conditions on me I have no idea what's going to happen.

Luckily for us, projectiles do not behave chaotically. However, there are plenty of physical systems that do behave this way. Let's investigate further.

P9.1 Our first example of a chaotic system is the driven, damped pendulum. Let's investigate further

(a) The differential equation for the driven, damped, pendulum is:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin(\theta) - q\frac{d\theta}{dt} + F_D\sin(\Omega_D t) \tag{9.1}$$

Write this second-order differential equation as two coupled first order equations.

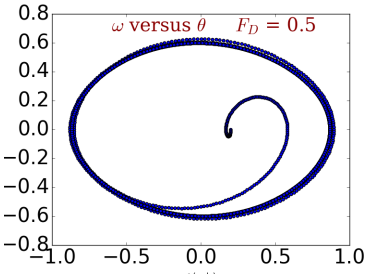


Figure 9.1 Phase space plot for the normal pendulum ($F_D = 0.5$)

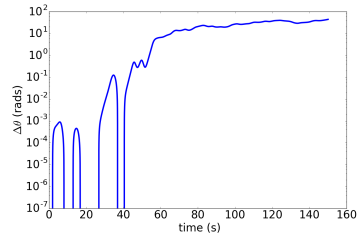


Figure 9.2 $\Delta\theta$ for two chaotic pendululms ($F_D = 1.2$).

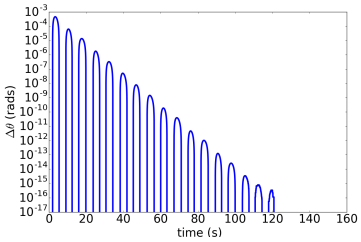


Figure 9.3 $\Delta\theta$ for two normal pendululms ($F_D = 0.5$).

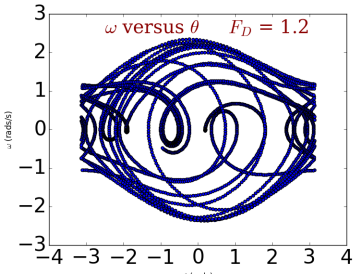


Figure 9.4 Phase space plot for the chaotic pendulum ($F_D = 1.2$)

(b) Use fourth-order Runge-Kutta to determine $\theta(t)$ Use the following variable values:

$$g = 9.8\ \text{m/s}^2 \qquad l = 9.8\ \text{m} \qquad q = \frac{1}{2}\ \text{s}^{-1} \tag{9.2}$$

$$\Omega_D = \frac{2}{3}\ \text{rads} \qquad F_D = 0\ \text{s}^{-2} \qquad dt = 0.04\ \text{s} \tag{9.3}$$

$$\theta(0) = 0.2\ \text{rads} \qquad \omega(0) = 0\ \text{rads/s} \tag{9.4}$$

$$\tag{9.5}$$

Plot θ vs. time for $0 < t < 60$ s. Play with different values of q to convince yourself that your simulation is behaving correctly.

(c) Now turn the driver on and set $F_D = 0.5$. Observe the motion and convince yourself that it is behaving appropriately.

(d) Increase the magnitude of the drive force to $F_D = 1.2$ and observe the motion. Explain the behavior.

(e) So far, nothing seems too out of the ordinary except that when we increased the driving force we got motion that wasn't periodic anymore. Let's take a closer look. Initialize a second pendulum with $\theta(0) = 0.201$ and plot θ vs. time for $0 < t < 60$ s. Compare this plot, side by side, to the plot with $\theta(0) = 0.2$. What do you notice?

(f) Increase the run time of both pendulums to 150 s, 300 s, and 600 s and compare.

(g) Maybe our time step is too big and numerical errors are simply compounding over time. Decrease the time step (dt) and plot θ vs. time to investigate this possibility.

(h) Finally, let's get a visual on how quickly the position of these two pendulums diverge from each other. Make a plot of $\Delta\theta$ vs. time where $\Delta\theta$ is the difference in angular position of the two pendulums. Do this for the normal pendulumm ($F_D = 0.5$) and for the chaotic pendulum ($F_D = 1.2$). Scale your vertical axis to a log scale. Your graphs should look like figures 9.2 and 9.3

As you can see, changing the initial position of the oscillator by a very small amount (.001 rads $\approx \frac{1}{2}^\circ$) resulted in very different motion. That's how a chaotic system behaves! The future motion (state of the system) is unpredictable, but it's not indeterminable. The pendulum still obeys the deterministic laws of physics (which allows us to use a numerical algorithm to calculate future motion) but is unpredictable due to the extreme sensitivity to the initial conditions.

P9.2 A plot of θ vs. time for the chaotic pendulum looks pretty structureless, with no clear pattern to the motion. Scientists like to look for patterns anyway they can.

(a) Construct a phase-space diagram for the driven,damped pendulum for $F_D = 1.2$ and $F_D = 0.5$. Compare the plots. A phase-space diagram is a plot of ω vs. θ. You probably don't have much intuition for this kind of plot but it is a useful way to spot structure and trends in a seemingly random and unpredictable system. Your graphs should look similar to figures 9.1 and 9.4.

(b) Instead of plotting every (ω,θ) pair, only plot those data points that are in-phase with the driving frequency. In other words, only plot those points that satisfy

$$\Omega_D t = 2\pi n \tag{9.6}$$

where n is an integer. You can think of this plot as if you put a strobe light on your phase-space diagram from part (a) and you only plotted those points that you see when the strobe appears. This graph is called a *Poincaré section* and is useful for spotting structure in chaotic systems. Your graph should look like figure 9.5

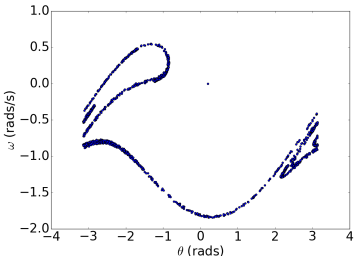


Figure 9.5 *Poincaré section* for the chaotic pendulum ($F_D = 1.2$). The pendulum ran for 15000 s.

Homework

H9.3 Weather systems are well-known to behave chaotically. Edward Lorenz was an atmospheric scientist in the mid 1900s that discovered this. He built atmospheric models and noticed that his models would predict very differently when the initial conditions were rounded (in an seemingly inconsequential fashion) compared to the results the emerged from the unrounded initial conditions. He famously hypothesized that a butterfly could flap his wings in Africa and the disturbance would cause a tornado in North America. This is well known as the butterfly effect today.[17]The differential equations that emerge from Lorenz's model are:

$$\frac{dx}{dt} = \sigma\left(y - x\right) \tag{9.7}$$

$$\frac{dy}{dt} = -xz + rx - y \tag{9.8}$$

$$\frac{dz}{dt} = xy - bz \tag{9.9}$$

$$\tag{9.10}$$

The variables x, y, and z can loosely be compared to temperature, density, and velocity of a fluid although this is a pretty drastic oversimplification

(a) Solve this set of first-order differential equations using fourth-order Runge-Kutta. Set $\sigma = 10$, $b = \frac{8}{3}$ and $r = 10$, $x(0) = 1$, $y(0) = 0$, and $z(0) = 0$. Plot z vs. time.

[17] The term "butterfly effect" has come to be a more general term since Lorenz originally coined it in reference to weather systems. Today, the butterfly effect is used to denote any situation where a small cause can have large effects.

(b) Now set $r = 25$ and plot z vs. time. Compare to the previous plot.

(c) Plot z vs. x.

(d) Plot a full three dimensional plot of x, y, and z (Look in the python book to see how to generate a 3D plot.)

Chapter 10

Differential Equations with Boundary Conditions

So far, we have only studied the behavior of systems where the initial conditions were specified and we calculated how the system evolved forward in time (e.g. the flight of a baseball given its initial position and velocity). The situation becomes somewhat more complicated if instead of having initial conditions, a differential equation has boundary conditions specified at both ends of the interval. This seemingly simple change in the boundary conditions changes everything about how we solve the problem. In this section we develop a method for using a grid and the finite difference formulas we developed in Lab 4 to solve ordinary differential equations with linear algebra techniques.

Solving differential equations with linear algebra

Consider the differential equation

$$y''(x) + 9y(x) = \sin(x) \quad ; \quad y(0) = 0, \quad y(2) = 1 \tag{10.1}$$

Notice that this differential equation has boundary conditions at both ends of the interval instead of having initial conditions at $x = 0$. If we represent this equation on a grid, we can turn this differential equation into a set of algebraic equations that we can solve using linear algebra techniques. Before we see how this works, let's first specify the notation that we'll use. We assume that we have set up a cell-edge spatial grid with N grid points, and we refer to the x values at the grid points using the notation x_j, with $j = 1..N$. We represent the (as yet unknown) function values $y(x_j)$ on our grid using the notation $y_j = y(x_j)$.

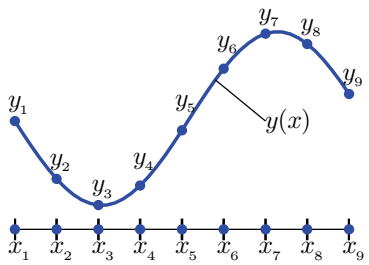


Figure 10.1 A function $y(x)$ represented on a cell-edge x-grid with $N = 9$.

Now we can write the differential equation in finite difference form as it would appear on the grid. The second derivative in Eq. (10.1) is rewritten using the centered difference formula (see Eq. (4.2)), so that the finite difference version of Eq. (10.1) becomes:

$$\frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} + 9y_j = \sin(x_j) \tag{10.2}$$

Now let's think about Eq. (10.2) for a bit. First notice that it is not *an* equation, but a system of *many* equations. We have one of these equations *at every grid point* j, except at $j = 1$ and at $j = N$ where this formula reaches beyond the ends of the grid and cannot, therefore, be used. Because this equation involves y_{j-1}, y_j, and y_{j+1} for the interior grid points $j = 2 \ldots N-1$, Eq. (10.2) is really a system of $N-2$ coupled equations in the N unknowns $y_1 \ldots y_N$. If we had just two more equations we could find the y_j's by solving a linear system of equations. But we do have two more equations; they are the boundary conditions:

$$y_1 = 0 \quad ; \quad y_N = 1 \tag{10.3}$$

which completes our system of N equations in N unknowns.

Before Python can solve this system we have to put it in a matrix equation of the form

$$\mathbf{Ay} = \mathbf{b}, \tag{10.4}$$

where **A** is a matrix of coefficients, **y** the column vector of unknown y-values, and **b** the column vector of known values on the right-hand side of Eq. (10.2). For the particular case of the system represented by Eqs. (10.2) and (10.3), the matrix equation is given by

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & 0 & \ldots & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & \ldots & 0 & 0 & 0 \\ . & . & . & . & \ldots & . & . & . \\ . & . & . & . & \ldots & . & . & . \\ . & . & . & . & \ldots & . & . & . \\ 0 & 0 & 0 & 0 & \ldots & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} \\ 0 & 0 & 0 & 0 & \ldots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ . \\ . \\ . \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} 0 \\ \sin(x_2) \\ \sin(x_3) \\ . \\ . \\ . \\ \sin(x_{N-1}) \\ 1 \end{bmatrix}. \tag{10.5}$$

Convince yourself that Eq. (10.5) is equivalent to Eqs. (10.2) and (10.3) by mentally doing each row of the matrix multiply by tipping one row of the matrix up on end, dotting it into the column of unknown y-values, and setting it equal to the corresponding element in the column vector on the right.

Once we have the finite-difference approximation to the differential equation in this matrix form (**Ay** = **b**), a simple linear solve is all that is required to find the solution array y_j. If **A** and **b** are matrix objects, Python solves this with this command: `y=A.I * b`. (See Chapter 11 in the book *Intro. to Python*)

P10.1 (a) Set up a cell-edge grid with $N = 30$ grid points from $x = 0$ to $x = 2$

(b) The exact solution to this differential equation is given by:

$$y(x) = -0.2236\,(\cos(6-x) + \cos(2+3x) + 16\sin(3x) - \cos(2-3x) - \cos(6+x)) \tag{10.6}$$

Plot this function on the cell-edge grid that we defined above.

(c) Now load the matrix in Eq. (10.5) and do the linear solve to obtain y_j and plot it on top of the exact solution with red dots (`'r.'`) to see how closely the two agree. Experiment with larger values of N and plot the difference between the exact and approximate solutions to see how the error changes with N. We think you'll be impressed at how well the numerical method works, if you use enough grid points.

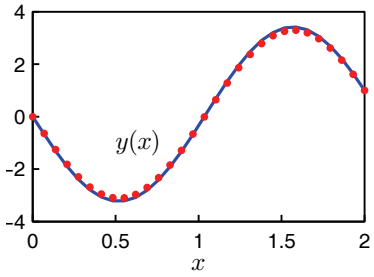


Figure 10.2 The solution to 10.1(c) with $N = 30$

Let's pause and take a minute to review how to apply the technique to solve a problem. First, write out the differential equation as a set of finite difference equations on a grid, similar to what we did in Eq. (10.2). Then translate this set of finite difference equations (plus the boundary conditions) into a matrix form analogous to Eq. (10.5). Finally, build the matrix **A** and the column vector **y** in

Python and solve for the vector **y** using `y=A.I * b`. Our example, Eq. (10.1), had only a second derivative, but first derivatives can be handled using the centered first derivative approximation, Eq. (4.2).

P10.2 Let's practice this procedure with another differential equation:

$$y'' + \frac{1}{x}y' + (1 - \frac{1}{x^2})y = x \quad ; \quad y(0) = 0, \quad y(5) = 1 \tag{10.7}$$

(a) Write out the finite difference equations on paper for the differential equation

(b) Write down the matrix **A** and the vector **b** for this equation.

(c) Now build these matrices in a Python script and solve the equation using the matrix method. Compare the solution found using the matrix method with the exact solution

$$y(x) = \frac{-4}{J_1(5)}J_1(x) + x$$

($J_1(x)$ is the first order Bessel function.)

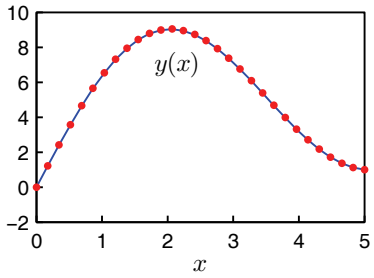


Figure 10.3 Solution to 10.2 with $N = 30$ (dots) compared to the exact solution (line)

Derivative boundary conditions

Now let's see how to modify the linear algebra approach to differential equations so that we can handle boundary conditions where derivatives are specified instead of values. Consider the differential equation

$$y''(x) + 9y(x) = x \quad ; \quad y(0) = 0 \quad ; \quad y'(2) = 0 \tag{10.8}$$

We can satisfy the boundary condition $y(0) = 0$ as before (just use $y_1 = 0$), but what do we do with the derivative condition at the other boundary?

P10.3 (a) A crude way to implement the derivative boundary condition is to use a forward difference formula

$$\frac{y_N - y_{N-1}}{h} = y'|_{x=2} \ . \tag{10.9}$$

In the present case, where $y'(2) = 0$, this simply means that we set $y_N = y_{N-1}$. Solve Eq. (10.8) in Python using the matrix method with this boundary condition. (Think about what the new boundary conditions will do to the final row of matrix **A** and the final element of vector **b**). Compare the resulting numerical solution to the exact solution obtained from Mathematica:

$$y(x) = \frac{x}{9} - \frac{\sin(3x)}{27\cos(6)} \tag{10.10}$$

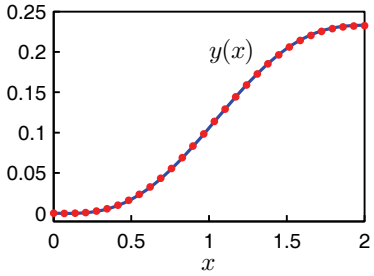


Figure 10.4 The solution to 10.3(a) with $N = 30$. The RMS difference from the exact solution is 8.8×10^{-4}

Nonlinear differential equations

Finally, we must confess that we have been giving you easy problems to solve, which probably leaves the impression that you can use this linear algebra trick to solve all second-order differential equations with boundary conditions at the ends. The problems we have given you so far are easy because they are *linear* differential equations, so they can be translated into *linear* algebra problems. Linear problems are not the whole story in physics, of course, but most of the problems we will do in this course are linear, so these finite-difference and matrix methods will serve us well in the labs to come.

P10.4 (Extra Credit)

(a) Here is a simple example of a differential equation that isn't linear:

$$y''(x) + \sin\big[y(x)\big] = 1 \quad ; \quad y(0) = 0, \quad y(3) = 0 \tag{10.11}$$

Work at turning this problem into a linear algebra problem to see why it can't be done, and explain the reasons someone around you.

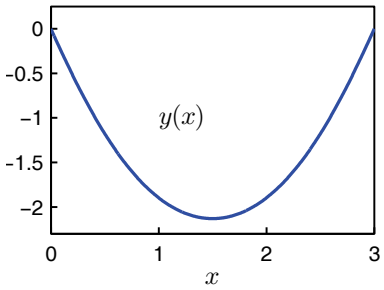


Figure 10.5 The solution to 10.4(b).

(b) Find a way to use a combination of linear algebra and iteration (initial guess, refinement, etc.) to solve Eq. (10.11) in Python on a grid.

HINT: Write the equation as

$$y''(x) = 1 - \sin\big[y(x)\big] \tag{10.12}$$

Make a guess for $y(x)$. (It doesn't have to be a very good guess. In this case, the guess $y(x) = 0$ works just fine.) Then treat the whole right side of Eq. (10.12) as known so it goes in the **b** vector. Then you can solve the equation to find an improved guess for $y(x)$. Use this better guess to rebuild **b** (again treating the right side of Eq. (10.12) as known), and then re-solve to get an even better guess. Keep iterating until your $y(x)$ converged to the desired level of accuracy. This happens when your $y(x)$ satisfies (10.11) to a specified criterion, *not* when the change in $y(x)$ from one iteration to the next falls below a certain level. An appropriate "error vector" would be $\mathbf{Ay} - (1 - \sin y)$.

(c) : Check your answer by using Mathematica's built-in solver to plot the solution.

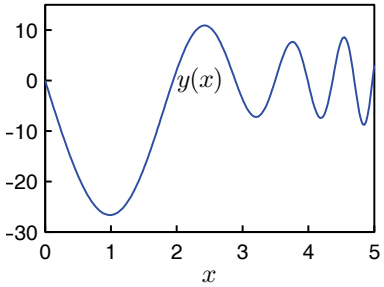


Figure 10.6 Solution to 10.5 with $N = 200$

Homework

H10.5 Solve the differential equation

$$y'' + \sin(x)\,y' + e^x y = x^2 \quad ; \quad y(0) = 0, \quad y(5) = 3 \tag{10.13}$$

in Python using the matrix method. This differential equation does not have an analytic solution. If it weren't for our numerical method, we would not know the solution.

H10.6 Let's improve the boundary condition formula from problem 10.3 using quadratic extrapolation.

(a) Use Mathematica to fit a parabola of the form

$$y(x) = a + bx + cx^2 \tag{10.14}$$

to the last three points on your grid. To do this, use (10.14) to write down three equations for the last three points on your grid and then solve these three equations for a, b, and c. Write the x-values in terms of the last grid point and the grid spacing ($x_{N-2} = x_N - 2h$ and $x_{N-1} = x_N - h$) but keep separate variables for y_{N-2}, y_{N-1}, and y_N. (You can probably use your code from Problem 3.3 with a little modification.)

(b) Now take the derivative of Eq. (10.14), evaluate it at $x = x_N$, and plug in your expressions for b and c. This gives you an approximation for the $y'(x)$ at the end of the grid. You should find that the new condition is

$$\frac{1}{2h}y_{N-2} - \frac{2}{h}y_{N-1} + \frac{3}{2h}y_N = y'(x_N) \tag{10.15}$$

Modify your program from problem 10.3 to include this new condition and show that it gives a more accurate solution than the crude technique from problem 10.3.

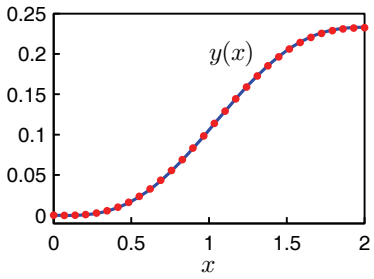


Figure 10.7 The solution to 10.6 with $N = 30$. The RMS difference from the exact solution is 5.4×10^{-4}

Part III

Partial Differential Equations

Chapter 11

The Wave Equation: Steady State and Resonance

To see why we did so much work in Lab 10 on ordinary differential equations when this is a course on partial differential equations, let's look at the wave equation in one dimension. For a string of length L fixed at both ends with a force applied to it that varies sinusoidally in time, the wave equation can be written as

$$\mu \frac{\partial^2 y}{\partial t^2} = T \frac{\partial^2 y}{\partial x^2} + f(x)\cos\omega t \quad ; \quad y(0,t) = 0, \quad y(L,t) = 0 \tag{11.1}$$

where $y(x,t)$ is the (small) sideways displacement of the string as a function of position and time, assuming that $y(x,t) \ll L$. [1] This equation may look a little unfamiliar to you, so let's discuss each term. We have written it in the form of Newton's second law, $F = ma$. The "ma" part is on the left of Eq. (11.1), except that μ is not the mass, but rather the linear mass density (mass/length). This means that the right side should have units of force/length, and it does because T is the tension (force) in the string and $\partial^2 y/\partial x^2$ has units of 1/length. (Take a minute and verify that this is true.) Finally, $f(x)$ is the amplitude of the driving force (in units of force/length) applied to the string as a function of position (so we are not necessarily just wiggling the end of the string) and ω is the frequency of the driving force.

Before we start calculating, let's train our intuition to guess how the solutions of this equation behave. If we suddenly started to push and pull on a string under tension we would launch waves, which would reflect back and forth on the string as the driving force continued to launch more waves. The string motion would rapidly become very messy. Now suppose that there was a little bit of damping in the system (not included in the equation above, but in a future lab we will add it). Then what would happen is that all of the transient waves due to the initial launch and subsequent reflections would die away and we would be left with a steady-state oscillation of the string at the driving frequency ω. (This behavior is the wave equation analog of damped transients and the steady final state of a driven harmonic oscillator.)

Steady state solution

You might remember from PH123 that there are two forms for the solution to the wave equation. They are:

$$y(x,t) = A\cos(kx - \omega t + \phi) \tag{11.2}$$

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 87-110.

and

$$y(x,t) = g(x)\cos(\omega t) \tag{11.3}$$

Take a second to notice the differences between these two functions. Waves described by equation (11.2) are called traveling waves and those described by equation (11.3) are called standing waves. Let's look for the steady-state (or standing wave) solution by guessing that the solution has the form of equation (11.3). Substituting this "guess" into the wave equation to see if it works yields (after some rearrangement)

$$Tg''(x) + \mu\omega^2 g(x) = -f(x) \quad ; \quad g(0) = 0, \quad g(L) = 0 \tag{11.4}$$

This is just a two-point boundary value problem of the kind we studied in Lab 10, so we can solve it using our matrix technique.

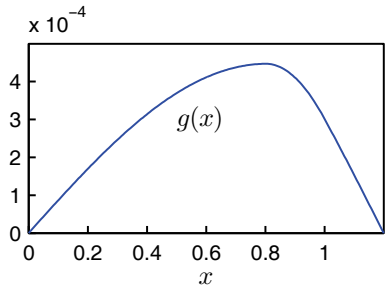


Figure 11.1 Solution to 11.1(a)

P11.1 (a) Modify one of your Python scripts from Lab 10 to solve Eq. (11.4) with $\mu = 0.003$, $T = 127$, $L = 1.2$, and $\omega = 400$. (All quantities are in SI units.) Find the steady-state amplitude associated with the driving force density:

$$f(x) = \begin{cases} 0.73 & \text{if } 0.8 \le x \le 1 \\ 0 & \text{if } x < 0.8 \text{ or } x > 1 \end{cases} \tag{11.5}$$

(b) Repeat the calculation in part (a) for 100 different frequencies between $\omega = 400$ and $\omega = 1200$ by putting a loop around your calculation in (a) that varies ω. Use this loop to load the maximum amplitude as a function of ω and plot it to see the resonance behavior of this system. Can you account qualitatively for the changes you see in $g(x)$ as ω varies? (Use `pyplot.pause(.1)` after the plot of $g(x)$ and watch what happens as ω changes.)

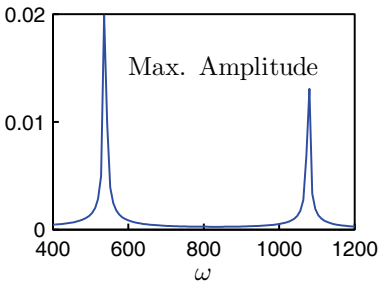


Figure 11.2 Solution to problem 11.1(b).

In problem 11.1(b) you should have noticed an apparent resonance behavior, with resonant frequencies near $\omega = 550$ and $\omega = 1100$ (see Fig. 11.2). Now we will learn how to use Python to find these resonant frequencies directly (i.e. without solving the differential equation over and over again).

Resonance and the eigenvalue problem

The essence of resonance is that at certain frequencies a large steady-state amplitude is obtained with a very small driving force. To find these resonant frequencies we seek solutions of Eq. (11.4) for which the driving force is zero. With $f(x) = 0$, Eq. (11.4) takes on the form

$$-\mu\omega^2 g(x) = Tg''(x) \quad ; \quad g(0) = 0, \quad g(L) = 0 \tag{11.6}$$

If we rewrite this equation in the form

$$g''(x) = -\left(\frac{\mu\omega^2}{T}\right) g(x) \tag{11.7}$$

then we see that it is in the form of a classic eigenvalue problem:

$$Ag = \lambda g \tag{11.8}$$

where A is a linear operator (the second derivative on the left side of Eq. (11.7)) and λ is the eigenvalue ($-\mu\omega^2/T$ in Eq. (11.7).)

Equation (11.7) is easily solved analytically, and its solutions are just the familiar sine and cosine functions. The condition $g(0) = 0$ tells us to try a sine function form, $g(x) = g_0 \sin(kx)$. To see if this form works we substitute it into Eq. (11.7 and find that it does indeed work, provided that the constant k is $k = \omega\sqrt{\mu/T}$. We have, then,

$$g(x) = g_0 \sin\left(\omega\sqrt{\frac{\mu}{T}}x\right) \tag{11.9}$$

where g_0 is the arbitrary amplitude. But we still have one more condition to satisfy: $g(L) = 0$. This boundary condition tells us the values that resonance frequency ω can take on. When we apply the boundary condition, we find that the resonant frequencies of the string are given by

$$\omega = n\frac{\pi}{L}\sqrt{\frac{T}{\mu}} \tag{11.10}$$

where n is an integer. Each value of n gives a specific resonance frequency from Eq. (11.10) and a corresponding spatial amplitude $g(x)$ given by Eq. (11.9). Figure **??** shows photographs of a string vibrating for $n = 1, 2, 3$.

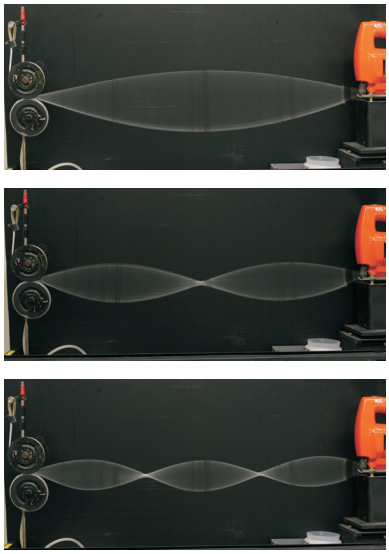


Figure 11.3 Photographs of the first three resonant modes for a string fixed at both ends.

For this simple example we were able to do the eigenvalue problem analytically without much trouble. However, when the differential equation is not so simple we will need to do the eigenvalue calculation numerically, so let's see how it works in this simple case. Rewriting Eq. (11.6) in matrix form, as we learned to do by finite differencing the second derivative, yields

$$\mathbf{Ag} = \lambda\mathbf{g} \tag{11.11}$$

which is written out as

$$\begin{bmatrix} ? & ? & ? & ? & \dots & ? & ? & ? \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & \dots & 0 & 0 & 0 \\ . & . & . & . & \dots & . & . & . \\ . & . & . & . & \dots & . & . & . \\ . & . & . & . & \dots & . & . & . \\ 0 & 0 & 0 & 0 & \dots & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ ? & ? & ? & ? & \dots & ? & ? & ? \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} = \lambda \begin{bmatrix} ? \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ ? \end{bmatrix} \tag{11.12}$$

where $\lambda = -\omega^2 \frac{\mu}{T}$. The question marks in the first and last rows remind us that we have to invent something to put in these rows that will implement the correct boundary conditions. Note that having question marks in the g-vector on the right is a real problem because without g_1 and g_N in the top and bottom positions, we don't have an eigenvalue problem (i.e. the vector **g** on left side of Eq. (11.12) is not the same as the vector **g** on the right side).

The simplest way to deal with this question-mark problem and to also handle the boundary conditions is to change the form of Eq. (11.8) to the slightly more complicated form of a *generalized eigenvalue problem*, like this:

$$\mathbf{Ag} = \lambda \mathbf{Bg} \tag{11.13}$$

where **B** is another matrix, whose elements we will choose to make the boundary conditions come out right. To see how this is done, here is the generalized modification of Eq. (11.12) with **B** and the top and bottom rows of **A** chosen to apply the boundary conditions $g(0) = 0$ and $g(L) = 0$.

$$\underset{\mathbf{A}}{\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & \dots & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix}} \underset{\mathbf{g}}{\begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix}} = \lambda \underset{\mathbf{B}}{\begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}} \underset{\mathbf{g}}{\begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix}} \tag{11.14}$$

Notice that the matrix **B** is very simple: it is just the identity matrix (made in Python with `eye(N,N)`) except that the first and last rows are completely filled with zeros. **Take a minute now and do the matrix multiplications corresponding the first and last rows and verify that they correctly give $g_1 = 0$ and $g_N = 0$, no matter what the eigenvalue λ turns out to be.**

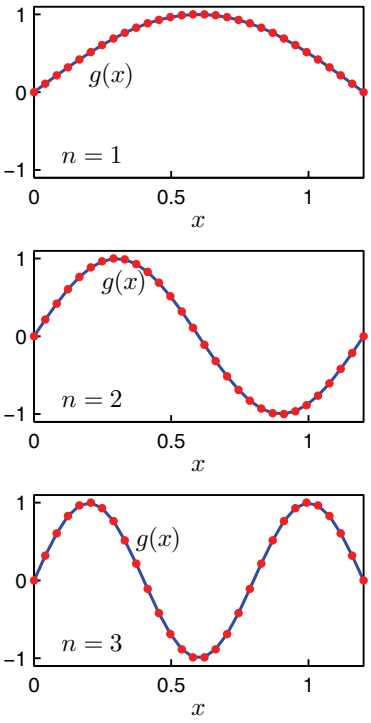


Figure 11.4 The first three eigenfunctions found in 11.2. The points are the numerical eigenfunctions and the line is the exact solution.

P11.2 Let's solve equation (11.6) as an eignevalue problem in Python.

(a) Load the matrix **A** with the matrix on the left hand side of equation (11.14) and the matrix **B** on the right side.

(b) Solve this problem using the `eig` function inside of scipy.linalg

(c) Convert the eigenvalues into frequencies via $\omega^2 = -\frac{T}{\mu}\lambda$, sort the squared frequencies in ascending order.

(d) Now plot some of the eigenfunctions, displaying the corresponding frequency as the title of the plot

(e) The exact frequencies are given by equation (11.10). Calculate the exact frequencies for $n = 1, 2, 3$ and compare them to your numerical results. Now calculate the exact frequency for $n = 20$ and compare to your numerical results. What trend do you notice and can you explain why?

(f) Now that you have the exact resonant frequencies, return to your code from problem 11.1 and make plots of the steady state amplitude for driving frequencies near these resonant values. You should find very large amplitudes, indicating that you have found the resonant modes.

Homework

H11.3 Let's explore what happens to the eigenmode shapes when we change the boundary conditions.

(a) Change your program from problem 11.2 to implement the boundary condition

$$g'(L) = 0$$

Use the approximation you derived in problem 10.6 for the derivative $g'(L)$ to implement this boundary condition, i.e.

$$g'(L) \approx \frac{1}{2h} g_{N-2} - \frac{2}{h} g_{N-1} + \frac{3}{2h} g_N$$

Explain physically why the resonant frequencies change as they do.

(b) In some problems mixed boundary conditions are encountered, for example

$$g'(L) = 2g(L)$$

Find the first few resonant frequencies and eigenfunctions for this case. Look at your eigenfunctions and verify that the boundary condition is satisfied.

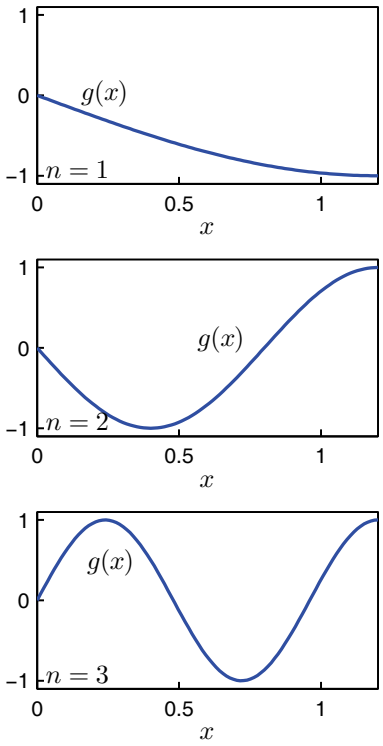


Figure 11.5 The first three eigenfunctions for 11.3(a).

Chapter 12

The Hanging Chain

The resonance modes that we studied in Lab 11 were simply sine functions. We can also use these techniques to analyze more complicated systems. In this lab we first study the problem of standing waves on a hanging chain. This problem was first solved in the 1700's by Johann Bernoulli and is the first time that the function that later became known as the J_0 Bessel function showed up in physics. Then we will jump forward several centuries in physics history and study bound quantum states using the same techniques.

Resonance for a hanging chain

Consider the chain hanging from the ceiling in the classroom.[1] We are going to find its normal modes of vibration using the method of Problem 11.2. The wave equation for transverse waves on a chain with varying tension $T(x)$ and constant linear mass density μ[2] is given by

$$\mu\frac{\partial^2 y}{\partial t^2} - \frac{\partial}{\partial x}\left(T(x)\frac{\partial y}{\partial x}\right) = 0 \tag{12.1}$$

Let's use a coordinate system that starts at the bottom of the chain at $x = 0$ and ends on the ceiling at $x = L$.

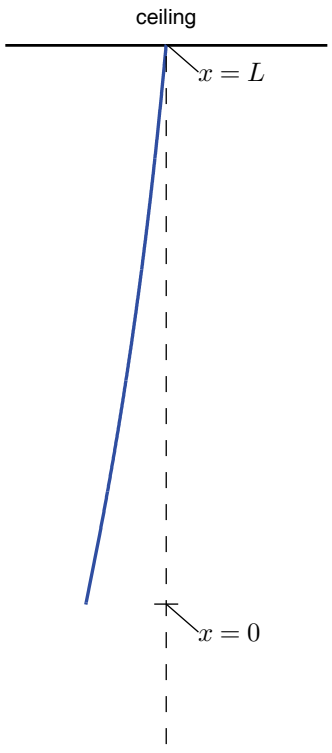


Figure 12.1 The first normal mode for a hanging chain.

P12.1 By using the fact that the stationary chain is in vertical equilibrium, show that the tension in the chain as a function of x is given by

$$T(x) = \mu g x \tag{12.2}$$

where μ is the linear mass density of the chain and where $g = 9.8\ \text{m/s}^2$ is the acceleration of gravity. Then show that Eq. (12.1) reduces to

$$\frac{\partial^2 y}{\partial t^2} - g\frac{\partial}{\partial x}\left(x\frac{\partial y}{\partial x}\right) = 0 \tag{12.3}$$

for a freely hanging chain.

As in Lab 11, we now look for normal modes by separating the variables: $y(x,t) = f(x)\cos(\omega t)$. We then substitute this form for $y(x,t)$ into (12.3) and simplify to obtain

$$x\frac{d^2 f}{dx^2} + \frac{df}{dx} = -\frac{\omega^2}{g}f \tag{12.4}$$

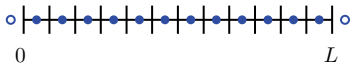


Figure 12.2 A cell-centered grid with ghost points. (The open circles are the ghost points.)

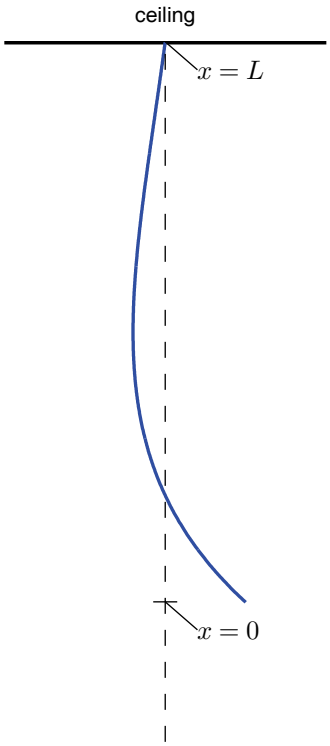


Figure 12.3 The shape of the second mode of a hanging chain

which is in eigenvalue form with $\lambda = -\omega^2/g$.

The boundary condition at the ceiling is $f(L) = 0$ while the boundary condition at the bottom is obtained by taking the limit of Eq. (12.4) as $x \to 0$ to find

$$f'(0) = -\frac{\omega^2}{g} f(0) = \lambda f(0) \tag{12.5}$$

In the past couple labs we have dealt with derivative boundary conditions by fitting a parabola to the last three points on the grid and then taking the derivative of the parabola (see Problem 10.6). This time we'll handle the derivative boundary condition by using a cell-centered grid with ghost points, as discussed in Lab **??**.

Recall that a cell-center grid divides the computing region from 0 to L into N cells with a grid point at the center of each cell. We then add two more grid points outside of $[0, L]$, one at $x_1 = -h/2$ and the other at $x_{N+2} = L + h/2$. The ghost points are used to apply the boundary conditions. Notice that by defining N as the number of interior grid points (or cells), we have $N+2$ total grid points, which may seem weird to you. We prefer it, however, because it reminds us that we are using a cell-centered grid with N physical grid points and two ghost points. You can do it any way you like, as long as your counting method works.

Notice that there isn't a grid point at either endpoint, but rather that the two grid points on each end straddle the endpoints. If the boundary condition specifies a value, like $f(L) = 0$ in the problem at hand, we use a simple average like this:

$$\frac{f_{N+2} + f_{N+1}}{2} = 0 \ , \tag{12.6}$$

and if the condition were $f'(L) = 0$ we would use

$$\frac{f_{N+2} - f_{N+1}}{h} = 0 \ . \tag{12.7}$$

When we did boundary conditions in the eigenvalue calculation of Problem 11.2 we used a **B** matrix with zeros in the top and bottom rows and we loaded the top and bottom rows of **A** with an appropriate boundary condition operator. Because the chain is fixed at the ceiling ($x = L$) we use this technique again in the bottom rows of **A** and **B**, like this (after first loading **A** with zeros and **B** with the identity matrix):

$$A[-1,-2] = \frac{1}{2} \quad A[-1,-1] = \frac{1}{2} \quad B[-1,-1] = 0 \tag{12.8}$$

P12.2 (a) Verify that these choices for the bottom rows of A and B in the generalized eigenvalue problem

$$\mathbf{A}f = \lambda \mathbf{B}f \tag{12.9}$$

give the boundary condition in Eq. (12.6) at the ceiling no matter what λ turns out to be.

(b) Now let's do something similar with the derivative boundary condition at the bottom, Eq. (12.5). Since this condition is already in eigenvalue form we don't need to load the top row of **B** with zeros. Instead we load **A** with the operator on the left ($f'(0)$) and **B** with the operator on the right ($f(0)$), leaving the eigenvalue $\lambda = -\omega^2/g$ out of the operators so that we still have $\mathbf{A}f = \lambda\mathbf{B}f$. Verify that the following choices for the top rows of **A** and **B** correctly produce Eq. (12.5).

$$A[0,0] = -\frac{1}{h} \quad A[0,1] = \frac{1}{h} \quad B[0,0] = \frac{1}{2} \quad B[0,1] = \frac{1}{2} \tag{12.10}$$

(c) Write the finite difference form of Eq. (12.4) and use it to load the matrices **A** and **B** for a chain with $L = 2$ m. (Notice that for the interior points the matrix **B** is just the identity matrix with 1 on the main diagonal and zeros everywhere else.) Use Python to solve for the normal modes of vibration of a hanging chain. As in Lab 11, some of the eigenvectors are unphysical because they don't satisfy the boundary conditions; ignore them.

(d) Solve Eq. (12.4) analytically using Mathematica without any boundary conditions. You will encounter the Bessel functions J_0 and Y_0, but because Y_0 is singular at $x = 0$ this function is not allowed in the problem. Apply the condition $f(L) = 0$ to find analytically the mode frequencies ω and verify that they agree with the frequencies you found in part (c).

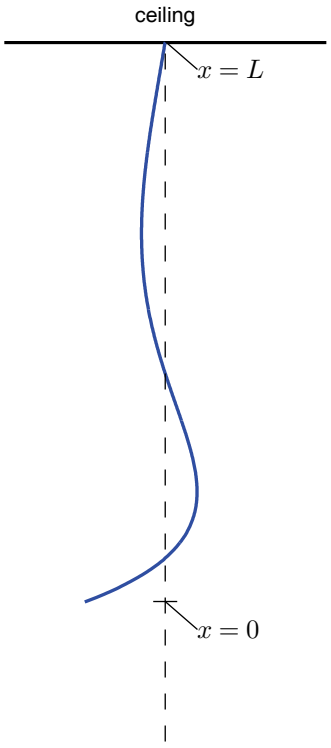


Figure 12.4 The shape of the third mode of a hanging chain

[1] For more analysis of the hanging chain, see N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 299-305.

[2] Equation (12.1) also works for systems with varying mass density if you replace μ with a function $\mu(x)$, but the equations derived later in the lab are more complicated with a varying $\mu(x)$.

Chapter 13

Lattice Vibrations

Hopefully you're beginning to see that normal modes of vibration show up everywhere, and it's worth your time to understand the numerical techniques we've studied to solve these kinds of problems. As you might already know, the atoms that make up a solid vibrate about their equilibrium locations. The collective motion of all atomic vibrations creates waves, sometimes traveling waves and sometimes standing (stationary) waves. These waves determine many of the critical properties of a material, including: i)heat capaticy, ii) thermal conductivity, iii) electrical conductivity, iv) speed of sound and more. In this lab we will study a simplified version of real lattice vibrations: a one-dimensional chain of particles connect via ideal springs.

Coupled Oscillators

Consider the situation depicted in figure 13.1. All of the masses are equal and all of the springs that connect the masses are identical. (We'll change that soon!) Each mass experiences two forces: one from the spring on the right and one from the spring on the left. We'll assume that both spring forces are given by Hooke's law:

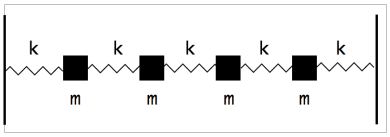


Figure 13.1 A collection of masses connected by springs.

$$F = -ku \tag{13.1}$$

where u is the distance away from equilibrium.

P13.1 (a) Write down Newton's second law for all of the masses in the figure. Once you are done, compare to the equations below and fix any differences between what you wrote and the correct solution. (Pay close attention to all of the signs in the equations)

$$-ku_1 + k(u_2 - u_1) = m\frac{d^2u_1}{dt^2} \tag{13.2}$$

$$-k(u_2 - u_1) + k(u_3 - u_2) = m\frac{d^2u_2}{dt^2} \tag{13.3}$$

$$-k(u_3 - u_2) + k(u_4 - u_3) = m\frac{d^2u_3}{dt^2} \tag{13.4}$$

$$-ku_4 + k(u_3 - u_4) = m\frac{d^2u_4}{dt^2} \tag{13.5}$$

(b) Now let's assume that the motion of each mass is harmonic. In other words, that it's displacement is given by:

$$u_j = x_j e^{-i\omega t} \tag{13.6}$$

where u_j is the displacment of particle j and x_j is the amplitude of the motion. These are standing wave solutions, not traveling wave. Plug this expression in for all of the u's in equation 13.2 - 13.5 and reduce so that all constants appear on the right hand sides. You should arrive at the following set of equations:

$$x_1 - (x_2 - x_1) = \frac{m\omega^2}{k} x_1 \tag{13.7}$$

$$(x_2 - x_1) - (x_3 - x_2) = \frac{m\omega^2}{k} x_2 \tag{13.8}$$

$$(x_3 - x_2) - (x_4 - x_3) = \frac{m\omega^2}{k} x_3 \tag{13.9}$$

$$x_4 - (x_3 - x_4) = \frac{m\omega^2}{k} x_4 \tag{13.10}$$

$$\tag{13.11}$$

(c) Hopefully you are starting to see an eigenvalue problem emerge from this math. In other words, can you see that the set of equations above can be represented as a linear algebra problem of this form:

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{x} \tag{13.12}$$

where the vector of amplitudes $\mathbf{x}$ are the eigenvectors and $\frac{m\omega^2}{k}$ are the eigenvalues. Look at the matrix below and see if you can determine what the entries with question marks should be?

$$\begin{bmatrix} ? & ? & 0 & 0 \\ ? & ? & ? & 0 \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{13.13}$$

You should find that the entries in the matrix look like this:

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{13.14}$$

Notice that since we don't have boundary conditions to consider, we don't need to write this equation as a generalized eigenvalue problem, just a normal eigenvalue problem.

(d) By noticing the pattern in the middle rows of matrix **A**, build a code to solve **the more general problem** involving any number of equal masses (not restricted to 4) connected by identical springs.

(e) Instead of plotting the eigenvectors like we've done in the past, make a movie of the masses as they vibrate. Remember, making a movie in python is little more than repeatedly plotting on the same canvas and then deleting the previous plot before you proceed to the next one.

(f) Is the wavelength of the standing wave different for the different modes or does it stay the same for all?

Clearly what we have studied today is a simplification of a real solid. In real solids:

1. the atoms vibrate in all three dimensions.
2. the forces between neighboring atoms may not be well-approximated by Hooke's Law.
3. Non-negligible forces from atoms located further away than nearest neighbor may exist making the forcing function more complicated.

Despite the simplifications that we have made here, hopefully you are beginning to understand the basics of lattice vibrations.

Homework

H13.2 (**LaTeX problem**) Now let's consider a system of masses where every other mass is different (m_1 and m_2) and every other spring is different (k_1 and k_2). You choose the values of the masses and the spring constants, and then convince yourself that your results make sense. Make an animation of the 3rd and 5th normal mode and verify that it looks correct. (Note: You'll need to rework the problem from the beginning to see how things change when the springs and masses are different. Don't try to modify your code without working out the details on paper (or whiteboard) first.)

Chapter 14

Animating the Wave Equation: Staggered Leapfrog

Up to this point, we have not solved a partial differential equation. We flirted with the idea in chapters 11 and 12 by using separation of variables to turn the partial differential equation into a set of ordinary differential equations. But separating the variables and expanding in orthogonal functions is not the only way to solve partial differential equations, and in many situations this technique is awkward, ineffective, or both. Today we will study another way of solving partial differential equations using a spatial grid and stepping forward in time. And as an added attraction, this method automatically supplies a beautiful animation of the solution. We will only show you one of several algorithms of this type that can be used on wave equations, so this is just an introduction to a larger subject. The method we will show you here is called *staggered leapfrog*; it is the simplest good method that we know.

The wave equation with staggered leapfrog

Consider again the classical wave equation with wave speed c. (For instance, for waves on a string $c = \sqrt{T/\mu}$.)

$$\frac{\partial^2 y}{\partial t^2} - c^2 \frac{\partial^2 y}{\partial x^2} = 0 \tag{14.1}$$

The boundary conditions to be applied are usually either of *Dirichlet* type (values specified):

$$y(0,t) = f_{\text{left}}(t) \quad ; \quad y(L,t) = f_{\text{right}}(t) \tag{14.2}$$

or of *Neumann* type (derivatives specified):

$$\frac{\partial y}{\partial x}(0) = g_{\text{left}}(t) \quad ; \quad \frac{\partial y}{\partial x}(L) = g_{\text{right}}(t) \tag{14.3}$$

Some mixed boundary conditions specify a relation between the value and derivative (as at the bottom of the hanging chain). These conditions tell us what is happening at the ends of the string. For example, maybe the ends are pinned ($f_{\text{left}}(t) = f_{\text{right}}(t) = 0$); perhaps the ends slide up and down on frictionless rings attached to frictionless rods ($g_{\text{left}}(t) = g_{\text{right}}(t) = 0$); or perhaps the left end is fixed and someone is wiggling the right end up and down sinusoidally ($f_{\text{left}}(t) = 0$ and $f_{\text{right}}(t) = A\sin\omega t$). In any case, some set of conditions at the ends are required to be able to solve the wave equation.

It is also necessary to specify the initial state of the string, giving its starting position and velocity as a function of position:

$$y(x, t=0) = y_0(x) \quad ; \quad \frac{\partial y(x,t)}{\partial t}|_{t=0} = v_0(x) \tag{14.4}$$

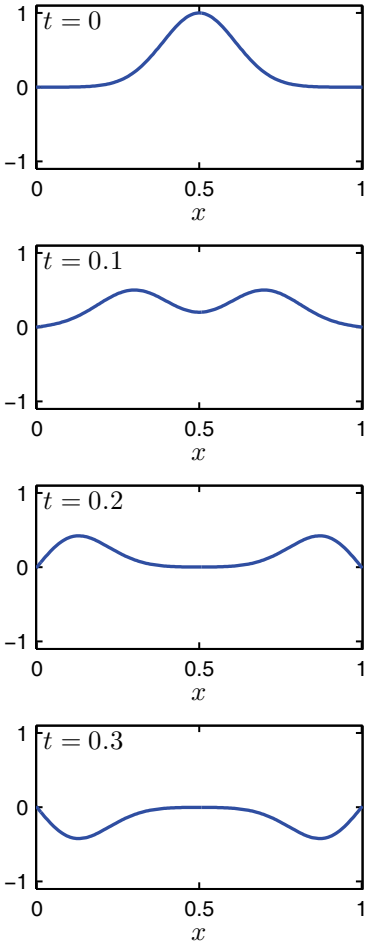


Figure 14.1 Snapshots of the evolution of a wave on a string with fixed ends and an initial displacement but no initial velocity. (See Problem 14.3(a))

Both of these initial conditions are necessary because the wave equation is second order in time, just like Newton's second law, so initial displacements and velocities must be specified to find a unique solution.

To numerically solve the classical wave equation via staggered leapfrog we approximate both the time and spatial derivatives with centered finite differences. In the notation below spatial position is indicated by a subscript j, referring to grid points x_j, while position in time is indicated by superscripts n, referring to time steps t_n so that $y(x_j,t_n)=y_j^n$. The time steps and the grid spacings are assumed to be uniform with time step called τ and grid spacing called h.

$$\frac{\partial^2 y}{\partial t^2}\approx\frac{y_j^{n+1}-2y_j^n+y_j^{n-1}}{\tau^2} \tag{14.5}$$

$$\frac{\partial^2 y}{\partial x^2}\approx\frac{y_{j+1}^n-2y_j^n+y_{j-1}^n}{h^2} \tag{14.6}$$

The staggered leapfrog algorithm is simply a way of finding y_j^{n+1} (y_j one time step into the future) from the current and previous values of y_j. To derive the algorithm just put these two approximations into the classical wave equation and solve for y_j^{n+1}: [1]

$$y_j^{n+1}=2y_j^n-y_j^{n-1}+\frac{c^2\tau^2}{h^2}\left(y_{j+1}^n-2y_j^n+y_{j-1}^n\right) \tag{14.7}$$

P14.1 Derive Eq. (14.7) from the approximate second derivative formulas. (You can use mathematica if you like, but this is really simple to do by hand.)

Equation (14.7) can only be used at interior spatial grid points because the $j+1$ or $j-1$ indices reach beyond the grid at the first and last grid points. The behavior of the solution at these two end points is determined by the boundary conditions. Since we will want to use both fixed value (Dirichlet) and derivative (Neumann) boundary conditions, let's use a **cell-centered grid with ghost points** (with N cells and $N+2$ grid points) so we can easily handle both types without changing our grid. If the values at the ends are specified (Dirichlet boundary conditions) we have

$$\frac{y_1^{n+1}+y_2^{n+1}}{2}=f_{\text{left}}(t_{n+1})\quad\Rightarrow\quad y_1^{n+1}=-y_2^{n+1}+2f_{\text{left}}(t_{n+1}) \tag{14.8}$$

$$\frac{y_{N+2}^{n+1}+y_{N+1}^{n+1}}{2}=f_{\text{right}}(t_{n+1})\quad\Rightarrow\quad y_{N+2}^{n+1}=-y_{N+1}^{n+1}+2f_{\text{right}}(t_{n+1}) \tag{14.9}$$

If the derivatives are specified (Neumann boundary conditions) then we have

$$\frac{y_2^{n+1}-y_1^{n+1}}{h}=g_{\text{left}}(t_{n+1})\quad\Rightarrow\quad y_1^{n+1}=y_2^{n+1}-hg_{\text{left}}(t_{n+1}) \tag{14.10}$$

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 421-429.

$$\frac{y_{N+2}^{n+1} - y_{N+1}^{n+1}}{h} = g_{\text{right}}(t_{n+1}) \quad \Rightarrow \quad y_{N+2}^{n+1} = y_{N+1}^{n+1} + h g_{\text{right}}(t_{n+1}) \tag{14.11}$$

To use staggered leapfrog, follow the following steps:

1. Advance the solution at all interior points to the next time step using Eq. (14.7)
2. Apply the boundary conditions using the appropriate equation from Eqs. (14.8)-(14.11) to find the values of y at the end points
3. Repeat to take another step forward in time.

The staggered leapfrog algorithm in Eq. (14.7) requires not just y at the current time level y_j^n but also y at the previous time level y_j^{n-1}. This means that we'll need to keep track of three arrays: an array `y` for the current values y_j^n, an array `yold` for the values at the previous time step y_j^{n-1}, and an array `ynew` for the values at the next time step y_j^{n+1}. At time $t = 0$ when the calculation starts, the initial position condition gives us the current values y_j^n, but we'll have to make creative use of the initial velocity condition to create an appropriate `yold` to get started. To see how this works, let's denote the initial values of y on the grid by y_j^0, the values after the first time step by y_j^1, and the unknown previous values (`yold`) by y_j^{-1}. A centered time derivative at $t = 0$ turns the initial velocity condition from Eq. (14.4) into

$$\frac{y_j^1 - y_j^{-1}}{2\tau} = v_0(x_j) \tag{14.12}$$

This gives us an equation for the previous values y_j^{-1}, but it is in terms of the still unknown future values y_j^1. However, we can use Eq. (14.7) to obtain another relation between y_j^1 and y_j^{-1}. Leapfrog at the first step ($n = 0$) says that

$$y_j^1 = 2y_j^0 - y_j^{-1} + \frac{c^2\tau^2}{h^2}\left(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0\right) \tag{14.13}$$

If we insert this expression for y_j^1 into Eq. (14.12), we can solve for y_j^{-1} in terms of known quantities:

$$y_j^{-1} = y_j^0 - v_0(x_j)\tau + \frac{c^2\tau^2}{2h^2}\left(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0\right) \tag{14.14}$$

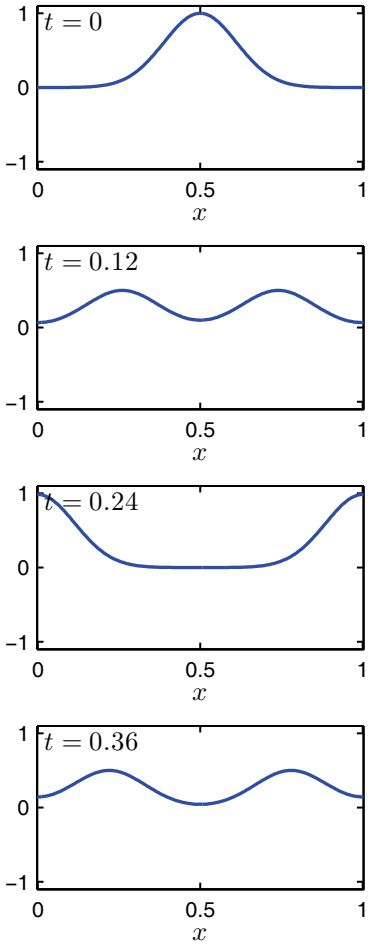


Figure 14.2 Snapshots of the evolution of a wave on a string with free ends and an initial displacement but no initial velocity. (See Problem 14.3(c))

P14.2 Derive Eq. (14.14) from Eqs. (14.12) and (14.13).

Now we have all the pieces we need to start coding.

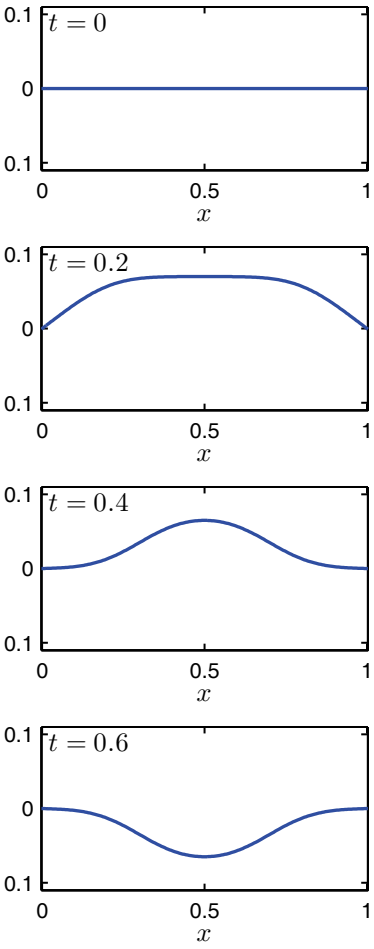


Figure 14.3 Snapshots of the evolution of a wave on a string with fixed ends and no initial displacement but with an initial velocity. (See Problem 14.3(d))

P14.3 Consider the motion of a guitar string, fixed at both ends:

$$y(0) = 0 \quad ; \quad y(L) = 0$$

The string is plucked so that the initial waveform is given by:

$$y(x,0) = e^{-\frac{160(x-\frac{L}{2})^2}{L^2}} - e^{-\frac{160\frac{L}{2}^2}{L^2}}$$

(a) Run an animation of the guitar string long enough that you can see the reflection from the ends and the way the two pulses add together and pass right through each other.

(b) Start experimenting with the time step (τ). Show by numerical experimentation that if $\tau > h/c$ the algorithm blows up spectacularly. This failure is called a *numerical instability* and we will be trying to avoid it all semester. This limit is called the *Courant-Friedrichs-Lewy condition*, or sometimes the *CFL condition*, or sometimes (unfairly) just the *Courant condition*.

(c) Now change the boundary conditions so that $\frac{\partial y}{\partial x} = 0$ at each end and watch how the reflection occurs in this case.

(d) Now change the initial conditions from initial displacement with zero velocity to initial velocity with zero displacement. Use an initial Gaussian velocity pulse just like the displacement pulse you used earlier and use fixed-end boundary conditions. Watch how the wave motion develops in this case. Then find a slinky, stretch it out, and whack it in the middle to verify that the math does the physics right.

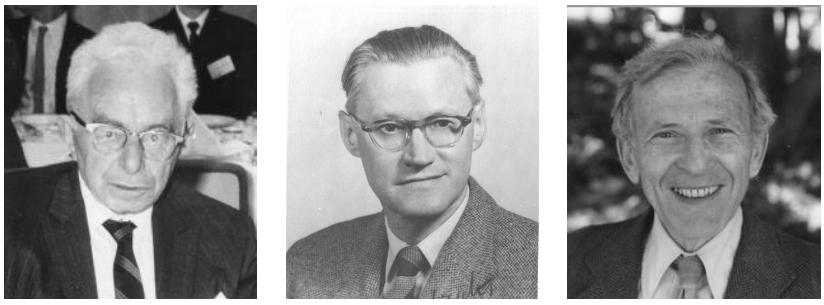


Figure 14.4 Richard Courant (left), Kurt Friedrichs (center), and Hans Lewy (right) described the CFL instability condition in 1928.

Homework

H14.4 Modify your leapfrog program to use $T = 1$ N and $\mu(x) = 0.1 + \frac{x}{L}$ so that the linear mass density of the guitar string increases toward the right. (This makes the wave speed be a function of position, $c = c(x)$.) Watch how the pulses propagate and explain qualitatively why they behave as they do.

Chapter 15

The driven, damped wave equation

Today we will continue solving the wave equation but we will add more physics to the equation, making the simulation more realistic. This will come at the cost of algorithmic complexity. Let's start by adding some damping to the wave equation using a linear damping term, like this:

$$\frac{\partial^2 y}{\partial t^2} + \gamma \frac{\partial y}{\partial t} - c^2 \frac{\partial^2 y}{\partial x^2} = 0 \tag{15.1}$$

with c constant. The staggered leapfrog method can be used to solve Eq. (15.1) also. To do this, we use the approximate first derivative formula

$$\frac{\partial y}{\partial t} \approx \frac{y_j^{n+1} - y_j^{n-1}}{2\tau} \tag{15.2}$$

along with the second derivative formulas in Eqs. (14.5) and (14.6) and find an expression for the values one step in the future:

$$y_j^{n+1} = \frac{1}{2+\gamma\tau}\left(4y_j^n - 2y_j^{n-1} + \gamma\tau y_j^{n-1} + \frac{2c^2\tau^2}{h^2}\left(y_{j+1}^n - 2y_j^n + y_{j-1}^n\right)\right) \tag{15.3}$$

P15.1 (a) Derive Eq. (15.3).

(b) Find a new formula for `yold` using Eqs. (14.12) and (15.3), similar to what we did to arrive at Eq. (14.14).

Your answer should look like this:

$$y_j^{-1} = -v_0(x_j)\frac{2\tau(2+\gamma\tau)}{4} + \left(y_j^0 + \frac{c^2\tau^2}{2h^2}\left(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0\right)\right) \tag{15.4}$$

(c) Modify your staggered leapfrog code to include damping with $\gamma = 0.2$. Then run your animation with the initial conditions in Problem 14.3(d) and verify that the waves damp away. You will need to run for about 25 s and you will want to use a big skip factor so that you don't have to wait forever for the run to finish.

(d) Include some code to record the maximum value of $y(x)$ over the entire grid as a function of time and then plot it as a function of time at the end of the run so that you can see the decay caused by γ.

(e) The decay of a simple harmonic oscillator is exponential, with amplitude proportional to $e^{-\gamma t/2}$. Scale this time decay function properly and lay it over your maximum y plot to see if it fits. Can you explain why the fit is as good as it is? (Hint: think about doing this problem via separation of variables.)

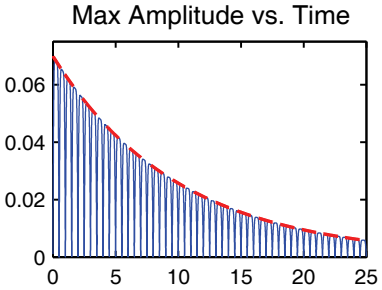


Figure 15.1 The maximum amplitude of oscillation decays exponentially for the damped wave equation. (Problem 15.1(d))

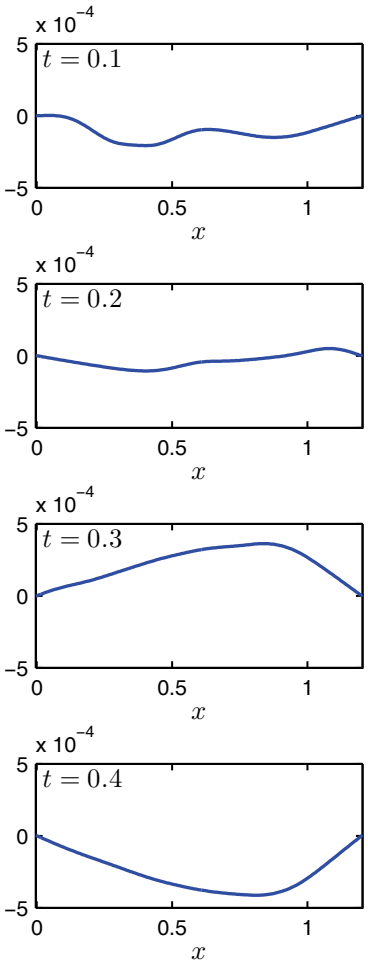


Figure 15.2 Snapshots of the evolution a driven and damped wave with $\omega = 400$. As the transient behavior dies out, the oscillation goes to the resonant mode. To make the pictures more interesting, the string was not started from rest in these plots. (In Problem 15.2 you start from rest for easier coding.)

The damped and driven wave equation

Finally, let's look at what happens when we add an oscillating driving force to our string, so that the wave equation becomes

$$\frac{\partial^2 y}{\partial t^2} + \gamma \frac{\partial y}{\partial t} - c^2 \frac{\partial^2 y}{\partial x^2} = \frac{f(x)}{\mu} \cos(\omega t) \tag{15.5}$$

At the beginning of Lab 11 we discussed the qualitative behavior of this system. Recall that if we have a string initially at rest and then we start to push and pull on a string with an oscillating force/length of $f(x)$, we launch waves down the string. These waves reflect back and forth on the string as the driving force continues to launch more waves. The string motion is messy at first, but the damping in the system causes the the transient waves from the initial launch and subsequent reflections to eventually die away. In the end, we are left with a steady-state oscillation of the string at the driving frequency ω.

P15.2 Model the motion of a damped, driven guitar string, clamped at both ends. Since we are driving the oscillation with an external force, we can start with an undisplaced string with zero velocity. This simplifies the initial conditions (just set $y = 0$ and $y_{\text{old}} = 0$ and enter the time-stepping loop). Use the following values for the string parameters: $T = 127$ N , $\mu = 0.003$ kg/m, and $L = 1.2$ m and remember that $c = \sqrt{T/\mu}$. Use the same driving force as in Problem 11.1(a)

$$f(x) = \begin{cases} 0.73 & \text{if } 0.8 \le x \le 1 \\ \\ 0 & \text{otherwise} \end{cases} \tag{15.6}$$

and set the driving frequency at $\omega = 400$.

(a) Derive an equation similar to Eq. (15.3) but that includes damping and driving as in Eq. (15.5). Your answer should look like this:

$$y_j^{n+1} = \frac{1}{2+\gamma\tau}\left(4y_j^n - 2y_j^{n-1} + \gamma\tau y_j^{n-1} + \frac{2c^2\tau^2}{h^2}\left(y_{j+1}^n - 2y_j^n + y_{j-1}^n\right)\right) + \frac{f(x_j)}{\mu}\cos(\omega t)\frac{2\tau^2}{(2+\gamma\tau)} \tag{15.7}$$

(b) Modify your code from Problem 15.1 to use this new algorithm.

(c) Choose a damping constant γ that is the proper size to make the system settle down to steady state after 20 or 30 bounces of the string. (You will have to think about the value of ω that you are using and about your damping rate result from problem 15.1 to decide which value of γ to use to make this happen.)

(d) Run the model long enough that you can see the transients die away and the string settle into the steady oscillation at the driving frequency. You may find yourself looking at a flat-line plot with no oscillation at all. If this happens look at the vertical scale of your plot and remember that we are doing real physics here. If your vertical scale goes from -1 to 1, you are expecting an oscillation amplitude of 1 meter on your guitar string. Compare the steady state mode to the shape found in Problem 11.1(a) (see Fig. 11.1).

Then run again with $\omega = 1080$, which is close to a resonance (see Fig. 11.2), and again see the system come into steady oscillation at the driving frequency.

Homework

H15.3 Use the staggered leapfrog algorithm to study how pulses propagate along a hanging chain. The wave equation for transverse waves on a chain with a varying tension ($T(x)$), constant linear mass density (μ), including damping and an external driving force is given by:

$$\frac{\partial^2 y}{\partial t^2} + \gamma \frac{\partial y}{\partial t} - \frac{1}{\mu}\frac{\partial}{\partial x}\left(T(x)\frac{\partial y}{\partial x}\right) = \frac{f(x)}{\mu}\cos(\omega t) \tag{15.8}$$

Here are some steps that you can follow to help you accomplish this:

(a) Recall that $T(x) = \mu g x$ for a hanging chain. Insert this into equation (15.8) and evaluate the third term on the left.

(b) Now discretize the equation and rearrange to solve for y_j^{n+1} just as we did in equation (14.7) (Don't lose track of your superscripts and subsripts). Your answer should look like this:

$$y_j^{n+1} = \frac{2\tau^2}{(2+\gamma\tau)}\frac{f(x_j)}{\mu}\cos(\omega t_j) + \frac{2\tau^2 g}{(2+\gamma\tau)}\left(\frac{y_{j+1}^n - y_{j-1}^n}{2dx}\right)$$
$$+\frac{2\tau^2 g}{(2+\gamma\tau)}x_j\left(\frac{y_{j+1}^n - 2y_j^n + y_{j-1}^n}{dx^2}\right) + \frac{2\tau^2}{(2+\gamma\tau)}\left(\frac{2y_j^n - y_j^{n-1}}{\tau^2}\right) + \frac{2\tau^2\gamma}{(2+\gamma\tau)}\left(\frac{y_j^{n-1}}{2\tau}\right) \tag{15.9}$$

(c) The initial conditions are easy enough to include (we'll just set the initial displacement and velocity to 0 and let the driving force create the disturbance) but we'll have to think a bit harder about the boundary conditions. At $x = L$, the rope is fixed to the ceiling so $y(L,t) = 0$. The harder one is at $x = 0$ where the chain is allowed to flap around as it

pleases. To get the boundary condition at $x = 0$ start with the equation that you arrived at in part (a) and evaluate it at $x = 0$. Then discretize this equation and solve it for y_0^{n+1} to get the desired boundary condition. Your answer should look like this:

$$y_0^{n+1} = \left(\frac{4\tau^2}{2+\gamma\tau}\right)\frac{f(x_j)}{\mu}\cos(\omega t_j) - \left(\frac{2-\gamma\tau}{2+\gamma\tau}\right)\left(y_1^{n-1} + y_0^{n-1}\right) + \left(\frac{4}{2+\gamma\tau}\right)\left(y_1^n + y_0^n\right) + \left(\frac{4\tau^2 g}{2+\gamma\tau}\right)\left(\frac{y_1^n - y_0^n}{dx}\right) - y_1^{n+1} \quad (15.10)$$

(d) With the two initial condition, the two boundary conditions and the differential equation all discretized, we are ready to code. Code up the staggered leapfrog for this situation. Let the animation run for several periods of the motion to verify that it is behaving in a way that you would expect.

Chapter 16

The 2-D Wave Equation With Staggered Leapfrog

Two dimensional grids

In this lab we will do problems in two spatial dimensions, x and y, so we need to spend a little time thinking about how to represent 2-D grids. For a simple rectangular grid where all of the cells are the same size, 2-D grids are pretty straightforward. We just divide the x-dimension into equally sized regions and the y-dimension into equally sized regions, and the two one dimensional grids intersect to create rectangular cells. Then we put grid points either at the corners of the cells (cell-edge) or at the centers of the cells (cell-centered). On a cell-center grid we'll usually want ghost point outside the region of interest so we can get the boundary conditions right.

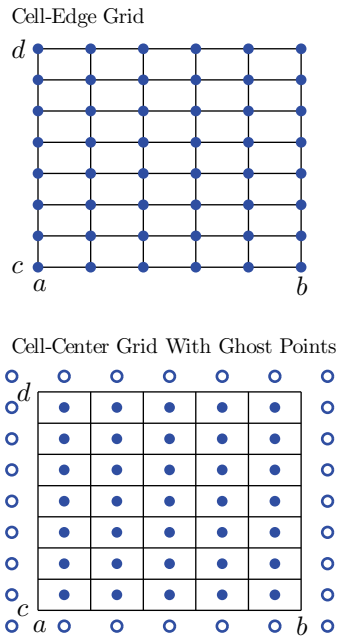


Figure 16.1 Two types of 2-D grids.

P16.1 Numpy has a nice way of representing rectangular two-dimensional grids using the `meshdgrid` command. Let's take a minute to learn how to make surface plots using this command. Consider a 2-d rectangle defined by $x \in [a,b]$ and $y \in [c,d]$. To create a 30-point cell-edge grid in x and a 50-point cell-edge grid in y with $a = 0$, $b = 2$, $c = -1$, $d = 3$, we use the following code:

```
from numpy import linspace,meshgrid
a = 0
b = 2
Nx = 30
x,dx = linspace(a,b,Nx,retstep = True)

c = 1
d = 3
Ny = 50
y,dy = linspace(c,d,Ny,retstep = True)


X,Y=meshgrid(x,y);
```

(a) Put the code fragment above in a script and run it to create the 2-D grid. Examine the contents of `X` and `Y` thoroughly enough that you can explain what the `meshgrid` command does. See the first section of chapter 9 in *Introduction to Python* for help understanding the indexing.

(b) Using this 2-D grid, evaluate the following function of x and y:

$$f(x,y) = e^{(-(x^2+y^2))} \cos(5\sqrt{x^2+y^2}) \tag{16.1}$$

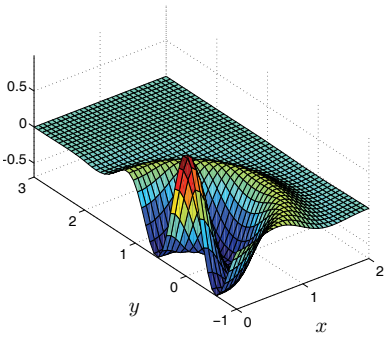


Figure 16.2 Plot from Problem 16.1

Use Python's `plot_surface` or `plot_wireframe` command to make a surface plot of this function. Then properly label the x and y axes with the symbols x and y, to get a plot like Fig. 16.2.

The two-dimensional wave equation

The wave equation for transverse waves on a rubber sheet is [1]

$$\mu \frac{\partial^2 z}{\partial t^2} = \sigma \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right) \tag{16.2}$$

In this equation μ is the surface mass density of the sheet, with units of mass/area. The quantity σ is the surface tension, which has rather odd units. By inspecting the equation above you can find that σ has units of force/length, which doesn't seem right for a surface. But it is, in fact, correct as you can see by performing the following thought experiment. Cut a slit of length L in the rubber sheet and think about how much force you have to exert to pull the lips of this slit together. Now imagine doubling L; doesn't it seem that you should have to pull twice as hard to close the slit? Well, if it doesn't, it should; the formula for this closing force is given by σL, which defines the meaning of σ.

We can solve the two-dimensional wave equation using the same staggered leapfrog techniques that we used for the one-dimensional case, except now we need to use a two dimensional grid to represent $z(x, y, t)$. We'll use the notation $z^n_{j,k} = z(x_j, y_k, t_n)$ to represent the function values. With this notation, the derivatives can be approximated as

$$\frac{\partial^2 z}{\partial t^2} \approx \frac{z^{n+1}_{j,k} - 2z^n_{j,k} + z^{n-1}_{j,k}}{\tau^2} \tag{16.3}$$

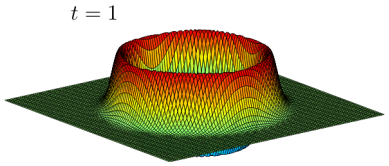
$$\frac{\partial^2 z}{\partial x^2} \approx \frac{z^n_{j+1,k} - 2z^n_{j,k} + z^n_{j-1,k}}{h_x^2} \tag{16.4}$$

$$\frac{\partial^2 z}{\partial y^2} \approx \frac{z^n_{j,k+1} - 2z^n_{j,k} + z^n_{j,k-1}}{h_y^2} \tag{16.5}$$

where h_x and h_y are the grid spacings in the x and y dimensions. We insert these three equations into Eq. (16.2) to get an expression that we can solve for z at the next time (i.e. $z^{n+1}_{j,k}$). Then we use this expression along with the discrete version of the initial velocity condition

$$v_0(x_j, y_k) \approx \frac{z^{n+1}_{j,k} - z^{n-1}_{j,k}}{2\tau} \tag{16.6}$$

to find an expression for the initial value of $z^{n-1}_{j,k}$ (i.e. `zold`) so we can get things started.



[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 129-134.

P16.2 (a) Derive the staggered leapfrog algorithm for the case of square cells with $h_x = h_y = h$. You should arrive at the following equation:

$$z_{j,k}^{n+1} = 2z_{j,k}^{n} - z_{j,k}^{n-1} + \frac{\sigma\tau^2}{\mu h^2}\left(z_{j+1,k}^{n} - 4z_{j,k}^{n} + z_{j-1,k}^{n} + z_{j,k+1}^{n} + z_{j,k-1}^{n}\right) \tag{16.7}$$

(b) Notice that to find the state of the wave at a future moment, you are going to need the state of the wave at two previous moments. Use equation (16.6) together with your result from part (a) to find an expression for `zold`. You've done this several times before so this should not feel too foreign to you. You should find that:

$$z_{j,k}^{-1} = z_{j,k}^{0} - v_{j,k}^{0}\tau + \frac{\sigma\tau^2}{2\mu h^2}\left(z_{j+1,k}^{0} - 4z_{j,k}^{0} + z_{j-1,k}^{0} + z_{j,k+1}^{0} + z_{j,k-1}^{0}\right) \tag{16.8}$$

(c) Write a Python script that animates the solution of the two dimensional wave equation on a square region that is $[-5,5] \times [-5,5]$ and that has fixed edges. Use a **cell-edge** square grid with the edge-values pinned to zero. Choose $\sigma = 2$ N/m and $\mu = 0.3$ kg/m^2 and use a displacement initial condition that is a Gaussian pulse with zero velocity

$$z(x, y, 0) = e^{-5(x^2+y^2)} \tag{16.9}$$

This initial condition doesn't strictly satisfy the boundary conditions, so you should pin the edges to zero.

Run the simulation long enough that you see the effect of repeated reflections from the edges.

(d) You will find that this two-dimensional problem has a Courant condition similar to the one-dimensional case, but with a factor out front:

$$\tau < f\frac{h}{c} \tag{16.10}$$

where $c = \sqrt{\frac{\sigma}{\mu}}$. Determine the value of the constant f by numerical experimentation. (Try various values of τ and discover where the boundary is between numerical stability and instability.)

(e) Also watch what happens at the center of the sheet by making a plot of $z(0, 0, t)$ there. In one dimension the pulse propagates away from its initial position making that point quickly come to rest with $z = 0$. This also happens for the three-dimensional wave equation. But something completely different happens in two (and higher) even dimensions; you should be able to see it in your plot by looking at the behavior of $z(0, 0, t)$ before the first reflection comes back.

(f) Finally, change the initial conditions so that the sheet is initially flat but with the initial velocity given by the Gaussian pulse of Eq. (16.9).

In one dimension when you pulse the system like this the string at the point of application of the pulse moves up and stays up until the reflection comes back from the ends of the system. Does the same thing happen in the middle of the sheet when you apply this initial velocity pulse? Answer this question by looking at your plot of $z(0,0,t)$. You should find that the two-dimensional wave equation behaves very differently from the one-dimensional wave equation.

Homework

H16.3 Consider the same square membrane from problem 16.2 but instead of a constant mass density, let's use the following function:

$$\mu(x,y) = 2.3 - 0.2x - 0.2y \tag{16.11}$$

Modify your code to model the motion of this membrane. Think about how you will calculate your Courant condition.

Elliptic, hyperbolic, and parabolic PDEs and their boundary conditions

Now let's take a step back and look at some general concepts related to solving partial differential equations. Probably the three most famous PDEs of classical physics are

(i) Poisson's equation for the electrostatic potential $V(x,y)$ given the charge density $\rho(x,y)$

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = \frac{-\rho}{\epsilon_0} \quad + \text{Boundary Conditions} \tag{16.12}$$

(ii) The wave equation for the wave displacement $y(x,t)$

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} = 0 \quad + \text{Boundary Conditions} \tag{16.13}$$

(iii) The thermal diffusion equation for the temperature distribution $T(x,t)$ in a medium with diffusion coefficient D

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \quad + \text{Boundary Conditions} \tag{16.14}$$

To this point in the course, we've focused mostly on the wave equation, but over the next several labs we'll start to tackle some of the other PDEs.

Mathematicians have special names for these three types of partial differential equations, and people who study numerical methods often use these names, so let's discuss them a bit. The three names are *elliptic*, *hyperbolic*, and *parabolic*. You can remember which name goes with which of the equations above by remembering the classical formulas for these conic sections:

$$\text{ellipse}: \quad \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{16.15}$$

$$\text{hyperbola}: \quad \frac{x^2}{a^2} - \frac{y^2}{b^2} = 1 \tag{16.16}$$

$$\text{parabola}: \quad y = ax^2 \tag{16.17}$$

Compare these equations with the classical PDE's above and make sure you can use their resemblances to each other to remember the following rules: Poisson's equation is elliptic, the wave equation is hyperbolic, and the diffusion equation is parabolic. These names are important because each different type of equation requires a different type of algorithm and boundary conditions. Fortunately, because you are physicists and have developed some intuition about the physics of these three partial differential equations, you can remember the proper boundary conditions by thinking about physical examples instead of memorizing theorems. And in case you haven't developed this level of intuition, here is a brief review of the matter.

Elliptic equations require the same kind of boundary conditions as Poisson's equation: $V(x,y)$ specified on all of the surfaces surrounding the region of interest. Since we will be talking about time-dependence in the hyperbolic and parabolic cases, notice that there is no time delay in electrostatics. When all of the bounding voltages are specified, Poisson's equation says that $V(x,y)$ is determined instantly throughout the region surrounded by these bounding surfaces. Because of the finite speed of light this is incorrect, but Poisson's equation is a good approximation to use in problems where things happen slowly compared to the time it takes light to cross the computing region.

To understand hyperbolic boundary conditions, think about a guitar string described by the transverse displacement function $y(x,t)$. It makes sense to give end conditions at the two ends of the string, but it makes no sense to specify conditions at both $t = 0$ and $t = t_{\text{final}}$ because we don't know the displacement in the future. This means that you can't pretend that (x,t) are like (x,y) in Poisson's equation and use "surrounding"-type boundary conditions. But we can see the right thing to do by thinking about what a guitar string does. With the end positions specified, the motion of the string is determined by giving it an initial displacement $y(x,0)$ and an initial velocity $\partial y(x,t)/\partial t|_{t=0}$, and then letting the motion run until we reach the final time. So for hyperbolic equations the proper boundary conditions are to specify end conditions on y as a function of time and to specify the initial conditions $y(x,0)$ and $\partial y(x,t)/\partial t|_{t=0}$.

Parabolic boundary conditions are similar to hyperbolic ones, but with one difference. Think about a thermally-conducting bar with its ends held at fixed temperatures. Once again, surrounding-type boundary conditions are inappropriate because we don't want to specify the future. So as in the hyperbolic case, we can specify conditions at the ends of the bar, but we also want to give initial conditions at $t = 0$. For thermal diffusion we specify the initial temperature $T(x,0)$, but that's all we need; the "velocity" $\partial T/\partial t$ is determined by Eq. (16.14), so it makes no sense to give it as a separate boundary condition. Summarizing: for parabolic equations we specify end conditions and a single initial condition $T(x,0)$ rather than the two required by hyperbolic equations.

If this seems like an arcane side trip into theory, we're sorry, but it's important. When you numerically solve partial differential equations you will spend 10% of your time coding the equation itself and 90% of your time trying to make the boundary conditions work. It's important to understand what the appropriate boundary conditions are.

Finally, there are many more partial differential equations in physics than just these three. Nevertheless, if you clearly understand these basic cases you can usually tell what boundary conditions to use when you encounter a new one. Here, for instance, is Schrödinger's equation:

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V\psi \tag{16.18}$$

which is the basic equation of quantum (or "wave") mechanics. The wavy nature of the physics described by this equation might lead you to think that the proper boundary conditions on $\psi(x,t)$ would be hyperbolic: end conditions on ψ and initial conditions on ψ and $\partial\psi/\partial t$. But if you look at the form of the equation, it looks like thermal diffusion. Looks are not misleading here; to solve this equation you only need to specify ψ at the ends in x and the initial distribution $\psi(x,0)$, but not its time derivative.

And what are you supposed to do when your system is both hyperbolic and parabolic, like the wave equation with damping?

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} - \frac{1}{D}\frac{\partial y}{\partial t} = 0 \tag{16.19}$$

The rule is that the highest-order time derivative wins, so this equation needs hyperbolic boundary conditions.

H16.4 Make sure you understand this material well enough that you are comfortable answering basic questions about PDE types and what types of boundary conditions go with them on a quiz and/or an exam.

Chapter 17

The Discrete Fourier Transform

Before we move on from our study of the wave equation and oscillations, we'll cover the topic of Fourier analysis. Fourier analysis is a powerful and widely-used technique. It has applications in signal processing, image processing, and many physics applications. Fourier techniques are helpful for breaking down, anaylzing, smoothing, and filtering any osciallatory function. They are also helpful when solving differential equations.

The Fourier Series

By now you should know that any function can be written as a sum of sine and cosine functions. More precisely, if $f(t)$ is defined on the interval $0 \leq x < \tau$, we can express this function as:

$$f(t) = \sum_{k=0}^{\infty} \alpha_k \cos\frac{2\pi k t}{\tau} + \sum_{k=1}^{\infty} \beta_k \sin\frac{2\pi k t}{\tau} \tag{17.1}$$

We can compress the notation by using the following identities: $\cos\theta = \frac{1}{2}\left(e^{-i\theta} + e^{i\theta}\right)$ and $\sin\theta = \frac{1}{2}i\left(e^{-i\theta} - e^{i\theta}\right)$. Upon collecting like terms, equation 17.1 becomes:

$$f(t) = \sum_{k=-\infty}^{\infty} \gamma_k e^{i\frac{2\pi k t}{\tau}} \tag{17.2}$$

with

$$\gamma_k = \begin{cases} \frac{1}{2}\left(\alpha_{-k} + i\beta_{-k}\right) & \text{if } k < 0 \\ \alpha_0 & \text{if } k = 0 \\ \frac{1}{2}\left(\alpha_k - i\beta_k\right) & \text{if } k > 0 \end{cases} \tag{17.3}$$

P17.1 Derive equation 17.2 by inserting the identities given above into equation 17.1.

The Fourier Transform

Finding the coefficients (γ_k) in equation 17.2 is valuable because it helps us know which of the sine and cosine functions should be included in the expansion. Or, to put it another way, if we knew which coefficients were not zero, we would know which frequencies were present in the function $f(t)$.

The standard approach to finding the γ_k is to exploit the orthogonality of the functions $e^{i\frac{2\pi kt}{\tau}}$. If we multiply both sides of equation 17.2 by $e^{-i\frac{2\pi k't}{\tau}}$ and integrate both sides, we get:

$$\int_0^\tau f(t)e^{-i\frac{2\pi k't}{\tau}}\,dt = \sum_{k=-\infty}^{\infty}\gamma_k\int_0^\tau e^{i\frac{2\pi kt}{\tau}}e^{-i\frac{2\pi k't}{\tau}}\,dt \tag{17.4}$$

$$\int_0^\tau f(t)e^{-i\frac{2\pi k't}{\tau}}\,dt = \sum_{k=-\infty}^{\infty}\gamma_k\int_0^\tau e^{i\frac{2\pi (k-k')t}{\tau}}\,dt \tag{17.5}$$

The integral on the right hand side of Eq. 17.5 will be zero unless $k = k'$.

P17.2 Use Mathematica to verify that the integral on the right hand side of equation 17.5 is zero if $k \neq k'$. Just pick a value for τ. What is the value of the integral when $k = k'$?

Since all of the integrals in the sum evaluate to zero except when $k = k'$, the sum essentially collapses down to one term and equation 17.5 becomes:

$$\int_0^\tau f(t)e^{-i\frac{2\pi k't}{\tau}}\,dt = \tau\gamma'_k \tag{17.6}$$

and we can solve for γ_k:

$$\gamma_k = \frac{1}{\tau}\int_0^\tau f(t)e^{-i\frac{2\pi kt}{\tau}}\,dt \tag{17.7}$$

The Discrete Fourier Transform

Equation 17.7 is great if **you know the function** $f(t)$. However, there are many situations when, instead of knowing the function you simply have samples of the function **at regular intervals**:

$$f(t_n)$$

where

$$t_n = \frac{n}{N}\tau$$

in this case, the integral in equation 17.7 becomes a sum over the function samples:

$$\gamma_k = \frac{1}{N}\sum_0^N f(t_n)e^{-i\frac{2\pi kn}{N}}\,dt \tag{17.8}$$

P17.3 Let's try equation 17.8 out on a few simple examples to get a feel for how it works. We'll start by working with a simple one-frequency function:

$$f(t) = \sin(5\times 2\pi t) \tag{17.9}$$

(a) Even though we have the analytic form of the function, let's pretend that we don't know what it is and just extract some samples from the function. Sample this function at a rate of 15 Hz from $t = 0$ to $t = 1$ s.

(b) Now use these samples to calculate the discrete Fourier transform using equation 17.8.

(c) Your probably noticed that your DFT had two peaks in it: one at 5 Hz and another at 10 Hz. What's going on here? Let's try increasing the sampling rate to 300 Hz and see what happens.

You should have seen two peaks again, but this time one was at 5 Hz and the other at 295 Hz. You expected the one at 5 Hz to be there but the other one is clearly incorrect. It turns out that only half (150) of the coefficients that you calculated were meaningful. The other half were simply complex conjugates of the first 150. In other words, **even though your sampling rate was** 300 **Hz, you were only able to detect frequencies up to** 150 **Hz**. Lock that last statement away, it's very key.

(d) Investigate the validity of this last statement by modifying your sampling function to be:

$$f(t) = \sin(5 \times 2\pi t) + \sin(170 \times 2\pi t) \tag{17.10}$$

and sampling at a rate of 300 Hz. Now you have two frequencies that you'd like to detect. Is your DFT able to correctly detect both of them?

A simple statement will help you remember the key important rule: *A DFT can only detect frequencies up to half the sampling rate!* This is called Nyquist's theorem.

(e) What this all amounts to is that the sum in equation 17.8 shouldn't go to N but rather to $N/2$ so that we don't calculate redundant coefficients. Modify your DFT code to incorporate this change and re-plot the DFT for the sampling function in 17.10

(f) Now modify your sampling rate so that you can detect the 170 Hz signal and recalculate the DFT.

(g) Calculate the DFT for the following function:

$$f(t) = \sin(5 \times 2\pi t) + 3\sin(170 \times 2\pi t) \tag{17.11}$$

Are you able to detect that the 170 Hz signal is stronger than the 5 Hz signal?

(h) Investigate what happens if you change the sampling time interval?

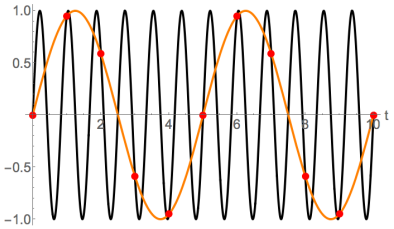


Figure 17.1 Aliasing; If your sampling rate is too low, you won't be able to distinguish the true signal from a lower-frequency version (the alias) of the signal.

One key point from that last exercise was that the sampling frequency is very important when performing a discrete Fourier transform. To help illustrate this, consider the 1.2 Hz signal shown in black in figure 17.1. Sampling this function

regularly using a sampling rate of 1 Hz would extract the sample points shown by the red dots. Furthermore, this same 1 Hz sampling rate would **extract the same exact samples** from a 0.2 Hz signal (shown in orange). How is your DFT algorithm suppose to know which signal is truly present? The answer is that it doesn't know; it can't know. Because the sampling rate (1 Hz) is not equal to twice the frequency of the 1.2 Hz signal, you cannot extract the higher-frequency information. The DFT algorithm will indicate the presence of the 0.2 Hz frequency even if the true signal was actually composed of a 1.2 Hz signal. The "fake" signal is called an alias and this problem is generally referred to as "aliasing".

P17.4 (a) Perform a discrete Fourier transform on the function:

$$f(t) = \cos(5 \times 2\pi t) + 3\sin(170 \times 2\pi t) \tag{17.12}$$

Can you detect the 5 Hz component of the signal?

(b) Now look closer at the γ_k coefficients and compare to equation 17.3.

Homework

H17.5 Let's incorporate our new knowledge about Fourier transforms into some of the previous problems we have done.

(a) Return to your code from problem 15.2 and add a member function that calculates the discrete Fourier transform.

(b) Add some code into your `animate` function that will collect samples in time at $x = 0.5$. Note: Sampling at every iteraion will amount to a very high sampling rate and you will end up collecting way too much data. Your DFT function probably won't be able to handle it. You should find a way to sample less frequently.

(c) Now call the DFT routine from part (a) to calculate the Fourier transform of the samples that you collected.

(d) You probably noticed that if your dataset gets too big, your DFT function crashes or takes a really long time. Numpy has some builtin Fourier transform functions that are optimized to be very fast. You can import them like this:

```
from numpy.fft import rfft
```

(`fft` stands for Fast Fourier Transform) Try using the builtin function to see what kind of a speedup you get.

(e) Now sample the wave at a different spatial location and compare the frequency components to the ones you got from part (c).

Chapter 18

The Diffusion, or Heat, Equation

Now let's attack the diffusion equation [1]

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \quad . \tag{18.1}$$

This equation describes how the distribution T (often associated with temperature) diffuses through a material with diffusion coefficient D. We'll study the diffusion equation analytically first and then we'll look at how to solve it numerically.

Analytic approach to the diffusion equation

The diffusion equation can be approached analytically by assuming that T is of the form $T(x,t) = g(x)f(t)$. If we plug this form into the diffusion equation, we can use separation of variables to find that $g(x)$ must satisfy

$$g''(x) + a^2 g(x) = 0 \tag{18.2}$$

where a is a separation constant. If we specify the boundary conditions that $T(x=0,t)=0$ and $T(x=L,t)=0$ then the solution to Eq. (18.2) is simply $g(x) = \sin(ax)$ and the separation constant can take on the values $a = n\pi/L$, where n is an integer. Any initial distribution $T(x,t=0)$ that satisfies these boundary conditions can be composed by summing these sine functions with different weights using Fourier series techniques.

P18.1 (a) Find how an initial temperature distribution $T_n(x,0) = T_0 \sin(n\pi x/L)$ decays with time, by substituting the form $T(x,t) = T(x,0)f(t)$ into the diffusion equation and finding $f_n(t)$ for each integer n. Do long wavelengths or short wavelengths decay more quickly?

(b) Show that an initial Gaussian temperature distribution like this

$$T(x) = T_0 e^{-(x-L/2)^2/\sigma^2} \tag{18.3}$$

decays according to the formula

$$T(x,t) = \frac{T_0}{\sqrt{1+4Dt/\sigma^2}} e^{-(x-L/2)^2/(\sigma^2+4Dt)} \tag{18.4}$$

by showing that this expression satisfies the diffusion equation Eq. (18.1) and the initial condition. (It doesn't satisfy finite boundary conditions, however; it is zero at $\pm\infty$.) Use Mathematica.

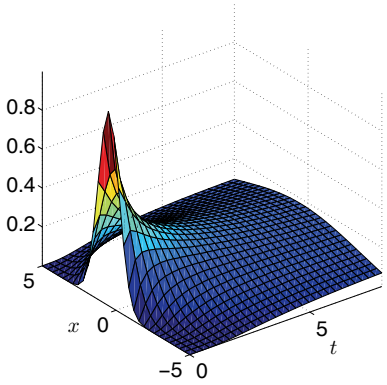


Figure 18.1 Diffusion of the Gaussian temperature distribution given in Problem 18.1(b) with $\sigma = 1$ and $D = 1$.

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 110-129.

Numerical approach: a first try

Now let's try to solve the diffusion equation numerically on a grid as we did with the wave equation. It is similar to the wave equation in that it requires boundary conditions at the ends of the computing interval in x. But because its time derivative is only first order we only need to know the initial distribution of T. This means that the trouble with the initial distribution of $\partial T/\partial t$ that we encountered with the wave equation is avoided. But in spite of this simplification, the diffusion equation is actually more difficult to solve numerically than the wave equation.

If we finite difference the diffusion equation using a centered time derivative and a centered second derivative in x to obtain an algorithm that is similar to leapfrog then we would have

$$\frac{T_j^{n+1} - T_j^{n-1}}{2\tau} = \frac{D}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{18.5}$$

$$T_j^{n+1} = T_j^{n-1} + \frac{2D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{18.6}$$

There is a problem starting this algorithm because of the need to have T one time step in the past (T_j^{n-1}), but even if we work around this problem this algorithm turns out to be worthless because no matter how small a time step τ we choose, we encounter the same kind of instability that plagues staggered leapfrog (infinite zig-zags). Such an algorithm is called *unconditionally unstable*, and is an invitation to keep looking. This must have been a nasty surprise for the pioneers of numerical analysis who first encountered it. It seems almost intuitively obvious that making an algorithm more accurate is better, but in this case the increased accuracy achieved by using a centered time derivative leads to numerical instability.

For now, let's sacrifice second-order accuracy to obtain a stable algorithm. If we don't center the time derivative, but use instead a forward difference we find

$$\frac{T_j^{n+1} - T_j^n}{\tau} = \frac{D}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{18.7}$$

$$T_j^{n+1} = T_j^n + \frac{D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{18.8}$$

You might expect this algorithm to have problems since the left side of Eq. (18.7) is centered at time $t_{n+\frac{1}{2}}$, but the right side is centered at time t_n. This problem makes the algorithm inaccurate, but it turns out that it is stable if τ is small enough. In the next lab we'll learn how to get a stable algorithm with both sides of the equation centered on the same time, but for now let's use this inaccurate (but stable) method.[2]

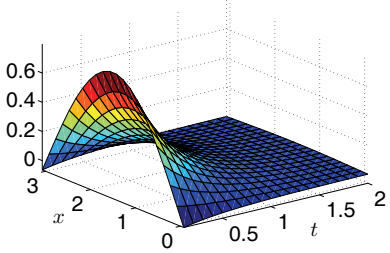


Figure 18.2 Diffusion of the $n = 1$

[2] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 412-421.

P18.2 (a) Modify one of your staggered leapfrog programs that uses a cell-center grid to implement this algorithm to solve the diffusion equation on the interval $[0, L]$ with initial distribution

$$T(x,0) = \sin(\pi x/L) \tag{18.9}$$

and boundary conditions $T(0) = T(L) = 0$. Use $D = 2$, $L = 3$, and $N = 20$. (You don't need to make a space-time surface plot like Fig. 18.2. Just make a line plot that updates each time step as we've done previously.) This algorithm has a CFL condition on the time step τ of the form

$$\tau \le C\frac{h^2}{D} \tag{18.10}$$

Determine the value of C by numerical experimentation.

Test the accuracy of your numerical solution by overlaying a graph of the exact solution found in 18.1(a). Plot the numerical solution as points and the exact solution as a line so you can tell the difference. Show that your grid solution matches the exact solution with increasing accuracy as the number of grid points N is increased from 20 to 40 and then to 80. You can calculate the error using something like

```
error = mean( abs( y - exact ) )
```

(b) Get a feel for what the diffusion coefficient does by trying several different values for D in your code.

(c) Verify your answer to the question in problem 18.1(a) about the decay rate of long versus short wavelengths by trying initial distributions of $T(x,0) = \sin(2\pi x/L)$, $T(x,0) = \sin(3\pi x/L)$, $T(x,0) = \sin(4\pi x/L)$, etc. and comparing decay rates.

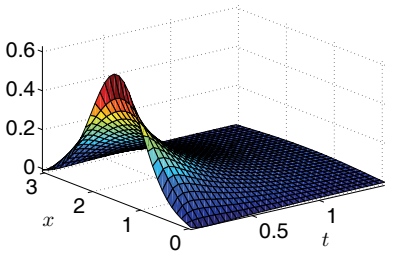


Figure 18.3 Diffusion of the Gaussian temperature distribution given in Problem 18.3(a) with fixed T boundary conditions.

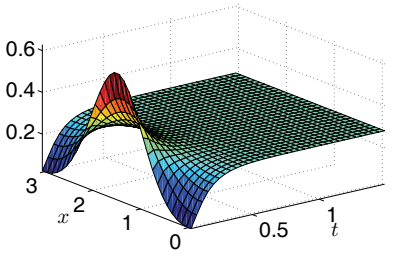


Figure 18.4 Diffusion of the Gaussian temperature distribution given in Problem 18.3(a) with insulating boundary conditions.

Even though this technique can give us OK results, the time step constraint for this method are pretty extreme. The constrain is of the form $\tau < Bh^2$, where B is a constant, and this limitation scales horribly with h. Suppose, for instance, that to resolve some spatial feature you need to decrease h by a factor of 5; then you will have to decrease τ by a factor of 25. This will make your script take forever to run, which is usually intolerable. In the next lab we'll learn a better way.

Homework

H18.3 Let's continue experimenting with the problem we started in 18.2

(a) Use as an initial condition a Gaussian distribution centered at $x = L/2$:

$$T(x,0) = e^{-40(x/L-1/2)^2} \tag{18.11}$$

Use two different kinds of boundary conditions:

(i) $T = 0$ at both ends and

(ii) $\partial T/\partial x = 0$ at both ends.

Explain what these boundary conditions mean by thinking about a watermelon that is warmer in the middle than at the edge. Tell physically how you would impose both of these boundary conditions on the watermelon and explain what the temperature history of the watermelon has to do with your plots of $T(x)$ vs. time.

(Notice how in case (i) the distribution that started out Gaussian quickly becomes very much like the $n = 1$ sine wave. This is because all of the higher harmonics die out rapidly.)

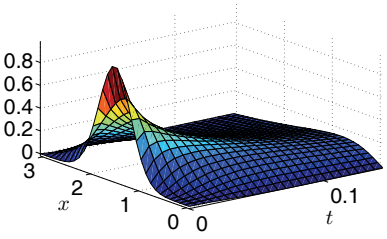


Figure 18.5 Diffusion of an initial Gaussian with D as in Problem 18.3(b).

(b) Modify your program to handle a diffusion coefficient which varies spatially like this:

$$D(x) = D_0 \frac{x^2 + L/5}{(L/5)} \tag{18.12}$$

with $D_0 = 2$. Note that in this case the diffusion equation is

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial T}{\partial x}\right) \tag{18.13}$$

Use the two different boundary conditions of part (a) and discuss why $T(x,t)$ behaves as it does in this case.

Chapter 19

Implicit Methods: the Crank-Nicolson Algorithm

You may have noticed that all of the time-stepping algorithms we have discussed so far are of the same type: at each spatial grid point j you use present, and perhaps past, values of $y(x,t)$ at that grid point and at neighboring grid points to find the future $y(x,t)$ at j. Methods like this, that depend in a simple way on present and past values to predict future values, are said to be *explicit* and are easy to code. They are also often numerically unstable, as we have seen previously. Even when they aren't unstable, there can be severe constraints on the size of the time step. *Implicit* methods are generally harder to implement than explicit methods, but they have much better stability properties. The reason they are harder is that they assume that you already know the future.

Implicit methods

To give you a better feel for what "implicit" means, let's study the simple first-order ordinary differential equation

$$\dot{y} = -\gamma y \tag{19.1}$$

P19.1 (a) We can solve this equation using Euler's method. We start by discretizing the equation:

$$\frac{y_{n+1} - y_n}{\tau} = -\gamma y_n \,. \tag{19.2}$$

and then solving for y_{n+1}:

$$y_{n+1} = y_n \left(1 - \gamma\tau\right) \,. \tag{19.3}$$

The term in parenthesis on the right hand side is sometimes referred to as the amplification factor. It is the amount by which the solution at one step is multiplied to get to the next step. If this term is greater than 1, the algorithm will be unstable and if it is less than 1 it will be stable. **Every instability that we have encountered thus far in the class can be traced to some kind of amplification factor like this.** For the present case we find the stability condition to be

$$\gamma\tau < 2$$

$$\tau < \frac{2}{\gamma} \tag{19.4}$$

Write a simple Python program and use Euler's method to solve this differential equation. By experimenting with τ, show that the stability condition in equation (19.4) holds. This is an example of an explicit method.

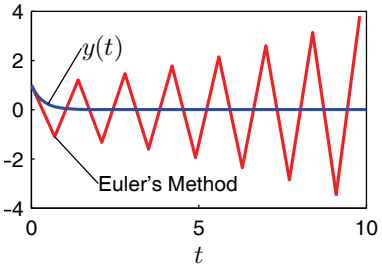


Figure 19.1 Euler's method is unstable for $\tau > 2/\gamma$. ($\tau = 2.1/\gamma$ in this case.)

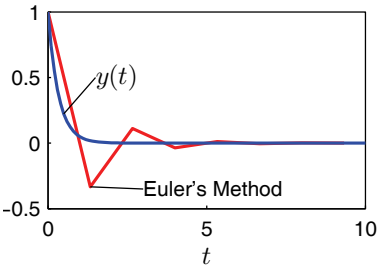


Figure 19.2 The implicit method in 19.1(b) with $\tau = 4/\gamma$.

(b) It may not be obvious how we can fix this problem to create a more stable algorithm. However, notice that the left side of Eq. (19.2) is centered on time $t_{n+\frac{1}{2}}$ but the right side is centered on t_n. Let's try centering the right-hand side at time $t_{n+\frac{1}{2}}$ by using an average of the advanced and current values of y,

$$y_n \Rightarrow \frac{y_n + y_{n+1}}{2} .$$

Please note that for all of the partial differential equations that we have solved thus far, we have **never** had any terms on the right hand side that were not centered at t_n. Having terms like y_{n+1} show up on the right hand side is a definite indication that you are using an implicit method. Now, solving for y_{n+1} is not a **simple** algebraic task, although you can do it. You should find that

$$y_{n+1} = y_n \frac{2-\gamma\tau}{2+\gamma\tau} \tag{19.5}$$

Notice that the presence of τ in the denominator prevents the amplification factor (the fraction on the right hand side) from going above 1 and therefore this algorithm will be more stable. This is an implicit method.

Show by numerical experimentation in a modified script that when τ becomes large this method doesn't blow up. It isn't correct because y_n bounces between positive and negative values, but at least it doesn't blow up.

(c) Now modify Euler's method by making it *fully implicit* by using y_{n+1} in place of y_n on the right side of Eq. (19.2) (this makes both sides of the equation reach into the future). [18]This method is no more accurate than Euler's method for small time steps, but it is much more stable and it doesn't bounce between positive and negative values. Show by numerical experimentation in a modified script that this fully implicit method damps away very quickly when τ is large. Extra damping is usually a feature of fully implicit algorithms.

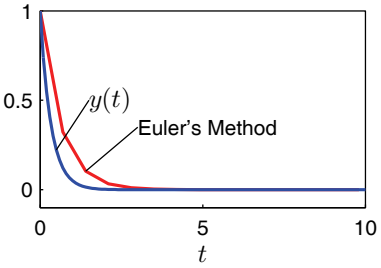


Figure 19.3 The fully implicit method in 19.1(c) with $\tau = 2.1/\gamma$.

[18] We call this method the Euler-Cromer method.

The diffusion equation with Crank-Nicolson

Now let's look at the diffusion equation again, and see how implicit methods can help us. Just to make things more interesting we'll let the diffusion coefficient be a function of x:

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial T}{\partial x}\right) \tag{19.6}$$

We begin by finite differencing the right side, taking care to handle the spatial dependence of D. Rather than expanding the nested derivatives in Eq. (19.6) let's difference it in place on the grid. In the equation below $D_{j\pm\frac{1}{2}} = D(x_j \pm h/2)$.

$$\frac{\partial T_j}{\partial t} = \frac{D_{j+\frac{1}{2}}\left(T_{j+1} - T_j\right) - D_{j-\frac{1}{2}}\left(T_j - T_{j-1}\right)}{h^2} \tag{19.7}$$

P19.2 Show that the right side of Eq. (19.7) is correct by finding a centered difference expressions for $D(x)\frac{\partial T}{\partial x}$ at $x_{j-\frac{1}{2}}$ and $x_{j+\frac{1}{2}}$. Then use these two expressions to find a centered difference formula for the entire expression at x_j. Draw a picture of a grid showing x_{j-1}, $x_{j-\frac{1}{2}}$, x_j, $x_{j+\frac{1}{2}}$, and x_{j+1} and show that this form is centered properly.

Now we take care of the time derivative by doing something similar to problem 19.1(b): we take a forward time derivative on the left, putting that side of the equation at time level $n+\frac{1}{2}$. To put the right side at the same time level (so that the algorithm will be second-order accurate), we replace each occurrence of T on the right-hand side by the average

$$T^{n+\frac{1}{2}} = \frac{T^{n+1} + T^n}{2} \tag{19.8}$$

like this:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = \frac{D_{j+\frac{1}{2}}\left(T_{j+1}^{n+1} - T_j^{n+1} + T_{j+1}^n - T_j^n\right) - D_{j-\frac{1}{2}}\left(T_j^{n+1} - T_{j-1}^{n+1} + T_j^n - T_{j-1}^n\right)}{2h^2} \tag{19.9}$$

If you look carefully at this equation you will see that there is a problem: how are we supposed to solve algebraically for T_j^{n+1}? The future values T^{n+1} are all over the place, and they involve three neighboring grid points (T_{j-1}^{n+1}, T_j^{n+1}, and T_{j+1}^{n+1}), so we can't just solve in a simple way for T_j^{n+1}. This is an example of why implicit methods are harder than explicit methods.

In the hope that something useful will turn up, let's put all of the variables at time level $n+1$ on the left, and all of the ones at level n on the right.

$$-D_{j-\frac{1}{2}}T_{j-1}^{n+1} + \left(\frac{2h^2}{\tau} + D_{j+\frac{1}{2}} + D_{j-\frac{1}{2}}\right)T_j^{n+1} - D_{j+\frac{1}{2}}T_{j+1}^{n+1} = D_{j-\frac{1}{2}}T_{j-1}^n + \left(\frac{2h^2}{\tau} - D_{j+\frac{1}{2}} - D_{j-\frac{1}{2}}\right)T_j^n + D_{j+\frac{1}{2}}T_{j+1}^n \tag{19.10}$$

We know this looks ugly, but it really isn't so bad. To solve for T_j^{n+1} we just need to solve a linear system, as we did in Lab 10 on two-point boundary value problems. When a system of equations must be solved to find the future values, we say that the method is *implicit*. This particular implicit method is called the *Crank-Nicolson algorithm*.

Phyllis Nicolson (1917–1968, English)

John Crank (1916–2006, English)

To see more clearly what we are doing, and to make the algorithm a bit more efficient, let's define a matrix **A** to describe the left side of Eq. (19.10) and another matrix **B** to describe the right side, like this:

$$\mathbf{A}T^{n+1} = \mathbf{B}T^n \tag{19.11}$$

(T is now a column vector). The elements of **A** are given by

$$A_{j,k} = 0 \quad \text{except for:}$$

$$A_{j,j-1} = -D_{j-\frac{1}{2}} \quad ; \quad A_{j,j} = \frac{2h^2}{\tau} + (D_{j+\frac{1}{2}} + D_{j-\frac{1}{2}}) \quad ; \quad A_{j,j+1} = -D_{j+\frac{1}{2}} \tag{19.12}$$

and the elements of **B** are given by

$$B_{j,k} = 0 \quad \text{except for:}$$

$$B_{j,j-1} = D_{j-\frac{1}{2}} \quad ; \quad B_{j,j} = \frac{2h^2}{\tau} - (D_{j+\frac{1}{2}} + D_{j-1/2}) \quad ; \quad B_{j,j+1} = D_{j+\frac{1}{2}} \tag{19.13}$$

Once the boundary conditions are added to these matrices, Eq. (19.11) can easily be solved symbolically to find T^{n+1}

$$T^{n+1} = \mathbf{A}^{-1}\mathbf{B}T^n \,. \tag{19.14}$$

Since we will be repeatedly inverting a (possibly very large) matrix as we step forward in time it is important that we look for every optimization we can find. In our case, you might have noticed that the matrices **A** and **B** have a unique structure. They are what we call banded matrices; only entries along the main diagonal and a few sub-diagonals contain non-zero entries. Numpy has routines that are optimized specifically for solving this kind of problem and we will use them here.

P19.3 (a) Write a Python program that implements the Crank-Nicolson algorithm for a one-dimensional rod held at zero temperature at the ends. Use $D(x) = 2$ and an initial temperature given by $T(x) = \sin(\pi x/L)$. (This is the same problem that you solved in lab 18) As you found in Lab 18, the exact solution for this distribution is:

$$T(x,t) = \sin(\pi x/L)e^{-\pi^2 Dt/L^2} \tag{19.15}$$

Below are some guidelines to help you:

(i) You should build a member function to load your matrices **A** and **B**. The middle rows of these matrices can be constructed using equations (19.12) and (19.13). The top and bottom rows of these matrices are (as usual) the bondary conditions. For Dirichlet boundary condtions with $T = 0$ at the ends, the top and bottom

rows of these matrices would be (assuming a cell-center grid with ghost points):

$$A[0,0] = \frac{1}{2} \quad A[0,1] = \frac{1}{2} \quad B[0,0] = 0 \tag{19.16}$$

$$A[-1,-1] = \frac{1}{2} \quad A[-1,-2] = \frac{1}{2} \quad B[-1,-1) = 0 \tag{19.17}$$

so that the equations for the top and bottom rows are

$$\frac{T_1 + T_2}{2} = 0 \qquad \frac{T_{N+1} + T_{N+2}}{2} = 0 \tag{19.18}$$

(ii) You'll want to use the function `solve_banded` from the library `scipy.linalg`. Read the documentation page for this function until you understand how to use it (Google "scipy solve_banded")

(iii) The animate member function is quite simple. The loop structure itself only has a few statements, one of them being the `solve_banded` function call.

(b) Try various values of τ and see how it compares with the exact solution. Verify that when the time step is too large the solution is inaccurate, but still stable. To do the checks at large time step you will need to use a long run time and not skip any steps in the plotting.

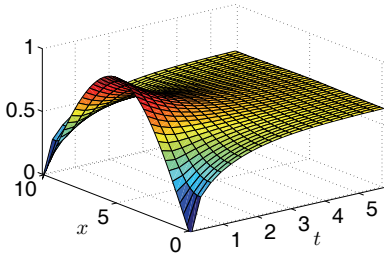


Figure 19.4 Solution to 19.3(d)

(c) Also study the accuracy of this algorithm by using various values of the cell number N and the time step τ. For each pair of choices run for 5 seconds and find the maximum difference between the exact and numerical solutions. You should find that the number of cells N hardly matters at all. The time step τ is the main thing to worry about if you want high accuracy in diffusion problems solved with Crank-Nicolson.

(d) Modify your Crank-Nicolson script to use boundary conditions $\partial T/\partial x = 0$ at the ends (what does this imply physically?). Run with the same initial condition as in part (a) (which does not satisfy these boundary conditions) and watch what happens. Use a "microscope" on the plots early in time to see what happens in the first few grid points during the first few time steps.

(e) Instead of setting $T = 0$ at the endpoints, set the $T = 1$ at $x = 0$ and $T = 5$ at $x = L$. Run with the same initial condition as in part (a) (which does not satisfy these boundary conditions) and watch what happens as the simulation progresses. Here are some hints for implementing the non-zero boundary conditions.

(i) The finite-valued boundary conditions cannot be implemented by modifying matrix $\mathbf{B}$ (can you explain why?). Instead, you should wait until after you have computed $\mathbf{B}T^n$ in your loop to implement them. If we call $r = \mathbf{B}T^n$, then immediately after this calculation we should set the boundary conditions like this:

```
r[0] = 1
r[-1] = 5
```

Homework

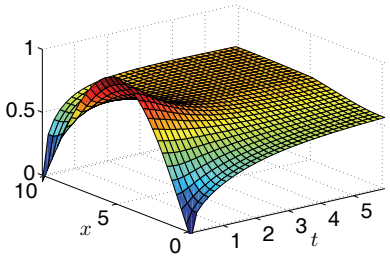


Figure 19.5 Solution to 19.4(a)

H19.4 Let's continue with the insulated ($\frac{\partial T}{\partial x} = 0$ at both boundaries), one-dimensional rod with the initial temperature profile of $T(x,0) = \sin(\pi x/L)$ and make a few modifications.

(a) Modify your code from problem 19.3 so that $D(x)$ is given by:

$$D(x) = \begin{cases} 1 & \text{if } x < \frac{L}{2} \\ 5 & \text{if } \frac{L}{2} \le x \le L \end{cases} \tag{19.19}$$

(b) Explain physically why your results are reasonable. In particular, explain why even though D is completely different, the final value of T is the same as in part (b).

H19.5 Modify your code from problem 19.4 to incorporate the boundary conditions:

$$\frac{\partial T}{\partial x} = -1 \text{ for } x = 0 \tag{19.20}$$

and

$$\frac{\partial T}{\partial x} = 2 \text{ for } x = L \tag{19.21}$$

Does the animation make sense? Can you describe physically what is happening here?

Chapter 20

Random Walks

All systems that we have studied thus far have been deterministic in nature. In other words, once we specify the differential equation along with the boundary and/or initial conditions, the solution is fully determined. There is one an only one correct solution to the problem. This doesn't mean that our numerical solutions for finding these true solutions are perfect, just that there is only one true solution.

In contrast to deterministic systems are stoichastic sytems, which employ some sort of randomness as a means to solve the problem. For example, consider squirting a small drop of perfume into a large room. As you would expect, the perfume molecules, which are initially highly concentrated in a very small space, will slowly move through the air molecules until they are evenly mixed throughout the entire room. The deterministic approach to this problem would involve using Euler's method (or Runge-Kutta) to determine the future positions and velocities of **all of the particles**, being sure to check for molecular collisions at every time step. None of the particles in the room could be omitted ($\sim 10^{27}$ particles) and time steps would need to be very small so that we don't miss a collision. Hopefully, you can see that this approach would be a fool's errand. In fact, your computer wouldn't even have enough memory to store the positions and velocities of this many particles.

Instead of approaching this problem deterministically, what if we just pretend that the displacment of each perfume molecule at each time step is random: random in step size and in step direction. Is it possible that such an approach could capture the true physical behavior of the perfume molecules as they bounce around off of all the other molecules? Let's see.

One-dimensional random walk

P20.1 Model the diffusion of perfume molecules in one dimension by tracking the locations of 500 random walkers. Let the step size for each walker be constant and equal to 1 unit and perform a total of 100 steps for each walker. After moving all of the walkers, calculate and save the average of the squares of the particle displacements:

$$\langle x^2 \rangle \tag{20.1}$$

Plot this average after the simulation is finished. Your plot should look something like figure 20.1

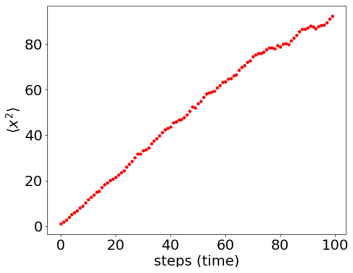


Figure 20.1 $\langle x^2 \rangle$ vs. time for 500 random walkers. The step size was 1 unit and each walker made 100 steps.

(a) The slope of the trend in figure 20.1 is related to the diffusion constant via the equation:

$$\langle x^2 \rangle = 2Dt \tag{20.2}$$

Fit the averages data to a line and determine the value of the diffusion constant. You should find that

(b) Investigate what happens when the probability for a left step is less than the probability for a right step. Is the motion still diffusive?

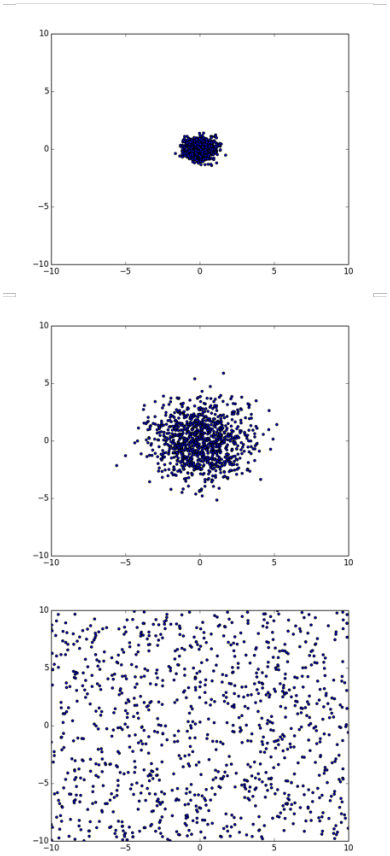


Figure 20.2

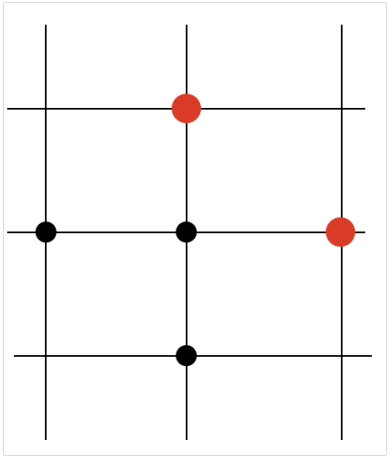


Figure 20.3 Two-dimesional random walker grid. Red dots repre-

Diffusion in three dimensions

We have hinted that a random walker is a good model for real diffusion, but let's make it a little more formal now. Consider a randow walker that is confined to walk on a three dimensional cubic lattice. Let $P^n_{i,j,k}$ be the probability of finding the particle on lattice site (i,j,k) at time n. Since we are on a simple cubic lattice, each site has 6 nearest neighbors. ($(i-1,j,k)$,$(i+1,j,k)$,$(i,j-1,k)$,$(i,j+1,k)$,$(i,j,k-1)$,$(i,j,k+1)$ If there is a walker on one of these neighboring sites at time n, then the probability that there will be a walker at site (i,j,k) at time $n+1$ is given by:

$$P^{n+1}_{i,j,k} = \frac{1}{6}\left(P^n_{i-1,j,k} + P^n_{i+1,j,k} + P^n_{i,j+1,k} + P^n_{i,j-1,k} + P^n_{i,j,k+1} + P^n_{i,j,k-1}\right) \tag{20.3}$$

Rearranging a little bit gives us:

$$P^{n+1}_{i,j,k} - P^n_{i,j,k} = \frac{1}{6}(P^n_{i-1,j,k} - 2P^n_{i,j,k} + P^n_{i+1,j,k} \tag{20.4}$$

$$+ P^n_{i,j+1,k} - 2P^n_{i,j,k} + P^n_{i,j-1,k} \tag{20.5}$$

$$+ P^n_{i,j,k+1} - 2P^n_{i,j,k} + P^n_{i,j,k-1}) \tag{20.6}$$

$$\tag{20.7}$$

Note that we have arranged the $2P^n_{i,j,k}$ terms strategically on the right hand side so that you can recognize them as finite difference version of second derivatives. The left hand side looks a lot like a time derivative. In other words, this equation looks a lot like the finite-difference version of the heat equation:

$$\frac{\partial P}{\partial t} = D\nabla^2 P \tag{20.8}$$

the density of the particles also obeys this differential equation:

$$\frac{\partial \rho}{\partial t} = D\nabla^2 \rho \tag{20.9}$$

P20.2 Model the diffusion of a perfume drop in a two-dimensional box. The perfume molecules should initially be placed very close to one another at the center of the box. Add the following functionality to your code:

(a) Make a simulation of the diffusion by periodically plotting the locations of all of the particles.

(b) Plot the number of particles at each point in the simulation cell at periodic times in the simulation. At the beginning of the simulation, you will see a large peak at the center of the box. As time progresses, that peak will level out until there is a near uniform distribution across the simulation cell. In order for the plot to be smooth, you will need to include at least 10,000 random walkers.

Homework

H20.3 (**LaTeX Problem**) You may recall the second law of thermodynamics, which states that the entropy of the universe never decreases. (It usually increases!) The perfume-squirt problem is a good example of this: the perfume molecules diffuse out and mix with the other molecules in a homogeneous fashion because to do so increases entropy. In this problem, we'll see if we can calculate the entropy of the perfume molecules as they diffuse and verify that the entropy increases.

(a) Begin by initializing a collection of 400 random walkers to be initially located uniformly in the region $-10 < x, y < 10$. The walkers are should not be allowed bo move beyond the location $x = \pm 10$, $y = \pm 10$.

(b) One commonly-stated definition of entropy is

$$S = -\sum_i P_i \ln P_i \tag{20.10}$$

where P_i is the probability of finding the system is state i. To apply this equation we must first overlay a fairly dense, uniform grid on top of our simulation box. (see figure 20.4) The summation in equation (20.10) then becomes a loop over all of the little boxes that you have defined. Define an 8 x 8 grid. (although you could pick a different one) For example, in the figure I have drawn 15 random walkers, and to evaluate equation (20.10) I would first begin at the upper left box. Since there are no particles in the box, I would skip that term in the sum (Don't want to do $\ln 0$). However P_4, which would be needed once you encountered the fourth box on the top row would be: $P_4 = \frac{1}{15}$ because there is a 1 in 15 chance of finding a particle in that cell.

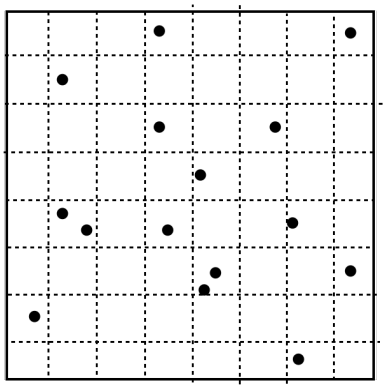


Figure 20.4 To calculate the entropy of the perfume molecules we must first divide our simulation box into smaller, equal sized boxes. Here, we have chosen an 8 x 8 grid.

(c) Inside your loop that executes the random walks, add some code to calculate the entropy using equation (20.10). You won't want to calcualte the entropy at every iteration, but I'll let you play with how frequently you should do it.

(d) Run the simulation sufficiently long so that the walkers spread out uniformly across the simulation cell and then plot entropy vs. time. You should notice that the entropy starts low (when all of the particles are all bunched together) and then increases as the particles diffuse out and assymptotically approaches the value:

$$S = -\ln(\frac{1}{N_c}) \tag{20.11}$$

where N_c is the number of cells that you summed over when you calculated the entropy. (If you used and 8x8 grid, $N_c = 64$) A plot of your entropy should look like figure 20.5.

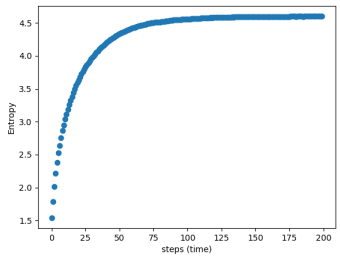


Figure 20.5 Entropy as a function of time for the perfume-squirt problem. The entropy has been calculated using the grid from figure 20.4

Chapter 21

Poisson's Equation I

Now let's consider Poisson's equation in two dimensions:[1]

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = -\frac{\rho}{\epsilon_0} \tag{21.1}$$

Poisson's equation is used to describe the electric potential in a two-dimensional region of space with charge density described by ρ. Note that by studying this equation we are also studying Laplace's equation (Poisson's equation with $\rho = 0$) and the steady state solutions to the diffusion equation in two dimensions ($\partial T/\partial t = 0$ in steady state).

Finite difference form

The first step in numerically solving Poisson's equation is to define a 2-D spatial grid. For simplicity, we'll use a rectangular grid where the x coordinate is represented by N_x values x_j equally spaced with step size h_x, and the y coordinate is represented by N_y values y_k equally spaced with step size h_y. This creates a rectangular grid with $N_x \times N_y$ grid points, just as we used in Lab **??** for the 2-d wave equation. We'll denote the potential on this grid using the notation $V(x_j, y_k) = V_{j,k}$.

The second step is to write down the finite-difference approximation to the second derivatives in Poisson's equation to obtain a grid-based version of Poisson's equation. In our notation, Poisson's equation is the represented by

$$\frac{V_{j+1,k} - 2V_{j,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} - 2V_{j,k} + V_{j,k-1}}{h_y^2} = -\frac{\rho}{\epsilon_0} \tag{21.2}$$

This set of equations can only be used at interior grid points because on the edges it reaches beyond the grid, but this is OK because the boundary conditions tell us what V is on the edges of the region.

Equation (21.2) plus the boundary conditions represent a set of linear equations for the unknowns $V_{j,k}$, so we could imagine just doing a big linear solve to find V all at once. Because this sounds so simple, let's explore it a little to see why we are not going to pursue this idea. The number of unknowns $V_{j,k}$ is $N_x \times N_y$, which for a 100×100 grid is 10,000 unknowns. So to do the solve directly we would have to be working with a $10,000 \times 10,000$ matrix, requiring 800 megabytes of RAM just to store the matrix. Doing this big solve is possible for 2-dimensional problems like this because computers with much more memory than this are

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 138-150.

now common. However, for larger grids the matrices can quickly get out of hand. Furthermore, if you wanted to do a 3-dimensional solve for a $100 \times 100 \times 100$ grid, this would require $(10^4)^3 \times 8$, or 8 million megabytes of memory. Computers like this are not going to be available for your use any time soon. So even though gigabyte computers are common, people still use iteration methods like the ones we are about to describe.

Iteration methods

Let's look for a version of Poisson's equation that helps us develop a less memory-intensive way to solve it.

P21.1 Solve the finite-difference version of Poisson's equation (Eq. (21.2)) for $V_{j,k}$ to obtain

$$V_{j,k} = \left(\frac{V_{j+1,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} + V_{j,k-1}}{h_y^2} + \frac{\rho}{\epsilon_0} \right) \Bigg/ \left(\frac{2}{h_x^2} + \frac{2}{h_y^2} \right) \tag{21.3}$$

With the equation in this form we could just iterate over and over by doing the following. (1) Choose an initial guess for the interior values of $V_{j,k}$. (2) Use this initial guess to evaluate the right-hand side of Eq. (21.3) and then to replace our initial guess for $V_{j,k}$ by this right-hand side, and then repeat. If all goes well, then after many iterations the left and right sides of this equation will agree and we will have a solution. [2]

Carl Friedrich Gauss (1777–1855, German)

It may seem that it would take a miracle for this to work, and it really is pretty amazing that it does, but we shouldn't be too surprised because you can do something similar just by pushing buttons on a calculator. Consider solving this equation by iteration:

$$x = e^{-x} \tag{21.4}$$

If we iterate on this equation like this:

$$x_{n+1} = e^{-x_n} \tag{21.5}$$

we find that the process converges to the solution $\bar{x} = 0.567$. Let's do a little analysis to see why it works. Let $\bar{x}$ be the exact solution of this equation and suppose that at the n^{th} iteration level we are close to the solution, only missing it by the small quantity δ_n like this: $x_n = \bar{x} + \delta_n$. Let's substitute this approximate solution into Eq. (21.5) and expand using a Taylor series. Recall that the general form for a Taylor's series is

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots + f^{(n)}(x)\frac{h^n}{n!} + \cdots \tag{21.6}$$

[2] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 429-441.

When we substitute our approximate solution into Eq. (21.5) and expand around the exact solution $\bar{x}$ we get

$$x_{n+1} = e^{-\bar{x}-\delta_n} \approx e^{-\bar{x}} - \delta_n e^{-\bar{x}} + \cdots \tag{21.7}$$

If we ignore the terms that are higher order in δ (represented by $\cdots$), then Eq. (21.7) shows that the error at the next iteration step is $\delta_{n+1} = -e^{-\bar{x}}\delta_n$. When we are close to the solution the error becomes smaller every iteration by the factor $-e^{-\bar{x}}$. Since $\bar{x}$ is positive, $e^{-\bar{x}}$ is less than 1, and the algorithm converges. When iteration works it is not a miracle—it is just a consequence of having this expansion technique result in an error multiplier that is less than 1 in magnitude.

P21.2 Write a short Python script to solve the equation $x = e^{-x}$ by iteration and verify that it converges. Then try solving this same equation the other way round: $x = -\ln x$ and show that the algorithm doesn't converge. Then use the $\bar{x} + \delta$ analysis above to show why it doesn't.

Well, what does this have to do with our problem? To see, let's notice that the iteration process indicated by Eq. (21.3) can be written in matrix form as

$$V_{n+1} = \mathbf{L}V_n + r \tag{21.8}$$

where $\mathbf{L}$ is the matrix which, when multiplied into the vector V_n, produces the $V_{j,k}$ part of the right-hand side of Eq. (21.3) and r is the part that depends on the charge density ρ. (Don't worry about what $\mathbf{L}$ actually looks like; we are just going to apply general matrix theory ideas to it.) As in the exponential-equation example given above, let $\bar{V}$ be the exact solution vector and let δ_n be the error vector at the n^{th} iteration. The iteration process on the error is, then,

$$\delta_{n+1} = \mathbf{L}\delta_n \tag{21.9}$$

Now think about the eigenvectors and eigenvalues of the matrix $\mathbf{L}$. If the matrix is well-behaved enough that its eigenvectors span the full solution vector space of size $N_x \times N_y$, then we can represent δ_n as a linear combination of these eigenvectors. This then invites us to think about what iteration does to each eigenvector. The answer, of course, is that it just multiplies each eigenvector by its eigenvalue. Hence, for iteration to work we need all of the eigenvalues of the matrix $\mathbf{L}$ to have magnitudes less than 1. So we can now restate the original miracle, "Iteration on Eq. (21.3) converges," in this way: "All of the eigenvalues of the matrix $\mathbf{L}$ on the right-hand side of Eq. (21.3) are less than 1 in magnitude." This statement is a theorem which can be proved if you are really good at linear algebra, and the entire iteration procedure described by Eq. (21.8) is known as *Jacobi iteration*. Unfortunately, even though all of the eigenvalues have magnitudes less than 1 there are lots of them that have magnitudes very close to 1, so the iteration takes forever to converge (the error only goes down by a tiny amount each iteration).

But Gauss and Seidel discovered that the process can be accelerated by making a very simple change in the process. Instead of only using old values of $V_{j,k}$ on the right-hand side of Eq. (21.3), they used values of $V_{j,k}$ as they became available during the iteration. (This means that the right side of Eq. (21.3) contains a mixture of V-values at the n and $n+1$ iteration levels.) This change, which is called *Gauss-Seidel iteration* is really simple to code; you just have a single array in which to store $V_{j,k}$ and you use new values as they become available. (When you see this algorithm coded you will understand this better.)

Successive over-relaxation

Even Gauss-Seidel iteration is not the best we can do, however. To understand the next improvement let's go back to the exponential example

$$x_{n+1} = e^{-x_n} \tag{21.10}$$

and change the iteration procedure in the following non-intuitive way:[19]

$$x_{n+1} = \omega e^{-x_n} + (1-\omega)x_n \tag{21.11}$$

where ω is a number which is yet to be determined.

[19] I can show you where this came from if it bothers you.

P21.3 Insert $x_n = \bar{x} + \delta_n$ into equation (21.11) and use the Taylor series to expand the expression as before. You should be able to show that the error changes as we iterate according to the following

$$\delta_{n+1} = (-\omega e^{-\bar{x}} + 1 - \omega)\delta_n \tag{21.12}$$

Now look: what would happen if we chose ω so that the factor in parentheses were zero? The equation says that we would find the correct answer in just one step! Of course, to choose ω this way we would have to know $\bar{x}$, but it is enough to know that this possibility exists at all. All we have to do then is numerically experiment with the value of ω and see if we can improve the convergence.

P21.4 Write a Python script that accepts a value of ω and runs the iteration in Eq. (21.11). Experiment with various values of ω until you find one that does the best job of accelerating the convergence of the iteration. You should find that the best ω is near 0.64, but it won't give convergence in one step. See if you can figure out why not. (Think about the approximations involved in obtaining Eq. (21.12).)

As you can see from Eq. (21.12), this modified iteration procedure shifts the error multiplier to a value that converges better. So now we can see how to improve Gauss-Seidel: we just use an ω multiplier like this:

$$V_{n+1} = \omega(\mathbf{L}V_n + r) + (1-\omega)V_n \tag{21.13}$$

then play with ω until we achieve almost instantaneous convergence.

Sadly, this doesn't quite work. The problem is that in solving for $N_x \times N_y$ unknown values $V_{j,k}$ we don't have just one multiplier; we have many thousands of them, one for each eigenvalue of the matrix. So if we shift one of the eigenvalues to zero, we might shift another one to a value with magnitude larger than 1 and the iteration will not converge at all. The best we can do is choose a value of ω that centers the entire range of eigenvalues symmetrically between -1 and 1. (Draw a picture of an arbitrary eigenvalue range between -1 and 1 and imagine shifting the range to verify this statement.)

Using an ω multiplier to shift the eigenvalues is called *Successive Over-Relaxation*, or SOR for short. Here it is written out so you can code it:

$$V_{j,k} = \omega \left(\frac{V_{j+1,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} + V_{j,k-1}}{h_y^2} + \frac{\rho}{\epsilon_0} \right) \Bigg/ \left(\frac{2}{h_x^2} + \frac{2}{h_y^2} \right) + (1-\omega) V_{j,k} \tag{21.14}$$

or (written in terms of Rhs, the right-hand side of Eq. (21.3)):

$$V_{j,k} = \omega Rhs + (1-\omega) V_{j,k} \tag{21.15}$$

with the values on the right updated as we go, i.e., we don't have separate arrays for the new V's and the old V's. And what value should we use for ω? The answer is that it depends on the values of N_x and N_y. In all cases ω should be between 1 and 2, with $\omega = 1.7$ being a typical value. Some wizards of linear algebra have shown that the best value of ω when the computing region is rectangular and the boundary values of V are fixed (Dirichlet boundary conditions) is given by

$$\omega = \frac{2}{1 + \sqrt{1 - R^2}} \tag{21.16}$$

where

$$R = \frac{h_y^2 \cos(\pi / N_x) + h_x^2 \cos(\pi / N_y)}{h_x^2 + h_y^2}. \tag{21.17}$$

These formulas are easy to code and usually give a reasonable estimate of the best ω to use. Note, however, that this value of ω was found for the case of a cell-edge grid with the potential specified at the edges. If you use a cell-centered grid with ghost points, and especially if you change to derivative boundary conditions, this value of ω won't be quite right. But there is still a best value of ω somewhere near the value given in Eq. (21.16) and you can find it by numerical experimentation.

Finally, we come to the question of when to stop iterating. It is tempting just to watch a value of $V_{j,k}$ at some grid point and quit when its value has stabilized at some level, like this for instance: quit when $\epsilon = |V(j,k)_{n+1} - V(j,k)_n| < 10^{-6}$. You will see this error criterion sometimes used in books, but *do not use it.* We know of one person who published an incorrect result in a journal because this error criterion lied. *We don't want to quit when the algorithm has quit changing V; we want to quit when Poisson's equation is satisfied.* (Most of the time these

are the same, but only looking at how V changes is a dangerous habit to acquire.) In addition, we want to use a relative (%) error criterion. This is easily done by setting a scale voltage V_{scale} which is on the order of the biggest voltage in the problem and then using for the error criterion

$$\epsilon = \left| \frac{Lhs - Rhs}{V_{\text{scale}}} \right| \tag{21.18}$$

where Lhs is the left-hand side of Eq. (21.3) and Rhs is its right-hand side. Because this equation is just an algebraic rearrangement of our finite-difference approximation to Poisson's equation, ϵ can only be small when Poisson's equation is satisfied. (Note the use of absolute value; can you explain why it is important to use it? Also note that this error is to be computed at all of the interior grid points. Be sure to find the maximum error on the grid so that you only quit when the solution has converged throughout the grid.)

And what error criterion should we choose so that we know when to quit? Well, our finite-difference approximation to the derivatives in Poisson's equation is already in error by a relative amount of about $1/(12N^2)$, where N is the smaller of N_x and N_y. There is no point in driving ϵ below this estimate. For more details, and for other ways of improving the algorithm, see *Numerical Recipes*, Chapter 19. OK, we're finally ready to code this all up.

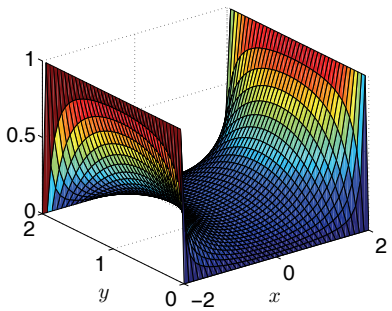


Figure 21.1 The electrostatic potential $V(x,y)$ with two sides grounded and two sides at constant potential.

[20] This is Laplace's equation

P21.5 (a) Solve Poisson's equation on a two dimensional grid $[-2,2] \times [0,2]$ ($N_x = N_y = 30$) with no free charge ($\rho = 0$)[20], and with the boundary conditions $V(-2,y) = V(2,y) = 1$ and $V(x,0) = V(x,2) = 0$. Set the error criteria to 10^{-4} and verify that the optimum value of ω given by Eq. (21.16) is the best one to use.

(b) Using the optimum value of ω in each case, run your program for $N_x = N_y = 20$, 40, 80, and 160. See if you can find a rough power law formula for how long it takes to push the error below 10^{-5}, i.e., guess that Run Time $\approx AN_x^p$, and find A and p. The Scipy fitting tool(`from scipy.optimize import curve_fit`) will be helpful for curve fitting. (see section 12.2 in the python book)

Homework

H21.6 Modify your code from problem 21.5 and place a point charge: $\rho = 1 \times 10^{-10}$ C/m^2 at coordinates $(0,1)$. Plot the potential that results. Try now with $\rho = -1 \times 10^{-10}$ C/m^2

Chapter 22

Poisson's Equation II

In three dimensions, Poisson's equation is given by

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -\frac{\rho}{\epsilon_0} \tag{22.1}$$

You can solve this equation using the SOR technique, but we won't make you do that here. Instead, we'll look at several geometries that are infinitely long in the z-dimension with a constant cross-section in the x-y plane. In these cases the z derivative goes to zero, and we can use the 2-D SOR code that we developed in the last lab to solve for the potential of a cross-sectional area in the x and y dimensions.

In this lab, we'll also learn how to use information about the potential (V) to calculate the electric field. Recall that

$$\mathbf{E} = -\nabla V = -\left(\frac{\partial V}{\partial x}\hat{i} + \frac{\partial V}{\partial y}\hat{j} + \frac{\partial V}{\partial z}\hat{k}\right) \tag{22.2}$$

We can use finite, discrete derivatives to calculate the components of the electric field over the entire domain. `Numpy` also has a `gradient` function that will perform the derivatives for you.

P22.1 Modify your code from last time to model a rectangular pipe with $V(-2, y) = -2$, $V(2, y) = 2$ and the $y = 0$ and $y = L$ edges held at $V = 0$.

(a) Make a plot of $V(x, y)$.

(b) Use `numpy.gradient` to calculate the electric field. Then make a plot of the electric field vectors $\mathbf{E}(x, y)$. See section 9.4 in the python book for help making a vector field plot.

(c) Now modify your boundary conditions so that the boundary condition at the $x = -L_x/2$ edge of the computation region is $\partial V/\partial x = 0$ and the boundary condition on the $y = L_y$ edge is $\partial V/\partial y = 0$. You can do this problem either by changing your grid and using ghost points or by using a quadratic extrapolation technique (see Eq. (10.15)). Both methods work fine, but note that you will need to enforce boundary conditions inside the main SOR loop now instead of just setting the values at the edges and then leaving them alone.

You may discover that the script runs slower on this problem. See if you can make it run a little faster by experimenting with the value of ω that you use. Again, changing the boundary conditions can change the eigenvalues of the operator. (Remember that Eq. (21.16) only works for cell-edge grids with fixed-value boundary conditions, so it only gives a ballpark suggestion for this problem.)

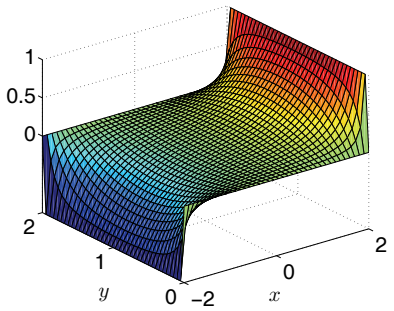


Figure 22.1 The potential $V(x, y)$ from Problem 11.1(a).

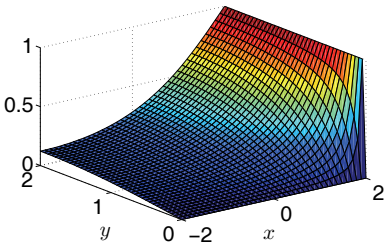


Figure 22.2 The potential $V(x,y)$ with zero-derivative boundary conditions on two sides (Problem 22.1(c).)

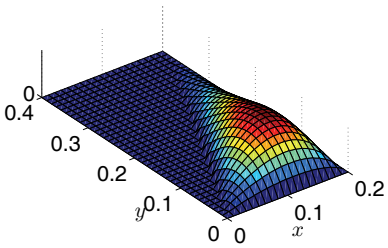


Figure 22.3 The potential $V(x,y)$ with constant charge density on a triangular region grounded at its edges (Problem 22.2(a).)

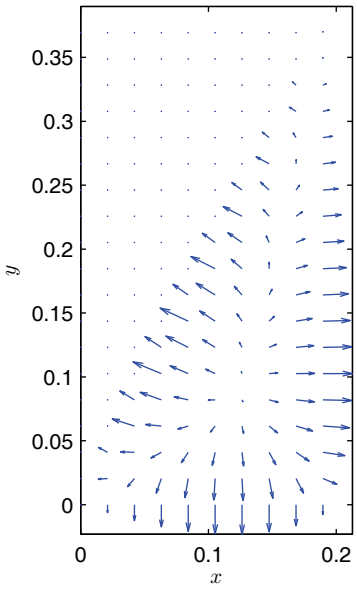


Figure 22.4 The electric field from Problem 22.2(b).)

P22.2 (a) Modify your code to solve the problem of an infinitely long hollow rectangular pipe of x-width 0.2 m and y-height 0.4 m with an infinitely long thin diagonal plate from the lower left corner to the upper right corner. The edges of the pipe and the diagonal plate are all grounded. There is uniform charge density $\rho = 10^{-10}$ C/m^3 throughout the lower triangular region and no charge density in the upper region (see Fig. 22.3). Find $V(x,y)$ in both triangular regions. You will probably want to have a special relation between N_x and N_y and use a cell-edge grid in order to apply the diagonal boundary condition in a simple way.

(b) Make a `quiver` plot of the electric field at the interior points of the grid.

Homework

H22.3 Study electrostatic shielding by going back to the boundary conditions of Problem 22.1 with $L_x = 0.2$ and $L_y = 0.4$, while grounding some points in the interior of the full computation region to build an approximation to a grounded cage. Allow some holes in your cage so you can see how fields leak in.

You will need to be creative about how you build your cage and about how you make SOR leave your cage points grounded as it iterates. One thing that won't work is to let SOR change all the potentials, then set the cage points to $V = 0$ before doing the next iteration. It is much better to set them to zero and force SOR to never change them. An easy way to do this is to use a cell-edge grid with a *mask*. A mask is an array that you build that is the same size as `V`, initially defined to be full of ones like this

```
from numpy import ones
mask=ones([Nx,Ny]);
```

Then you go through and set the elements of `mask` that you don't want SOR to change to have a value of zero. (We'll let you figure out the logic to do this for the cage.) The mask can then be used when you update $V(i,j)$ to ensure that the value of the potential on the cage is never modified from it's original value. There are a few different ways to do this. Feel free to do what makes sense to you.

The use of the mask is powerful because you can calculate the potential for quite complicated shapes just by changing the mask array. Just build the mask once, before the main loop, and the SOR algorithm can handle the rest.

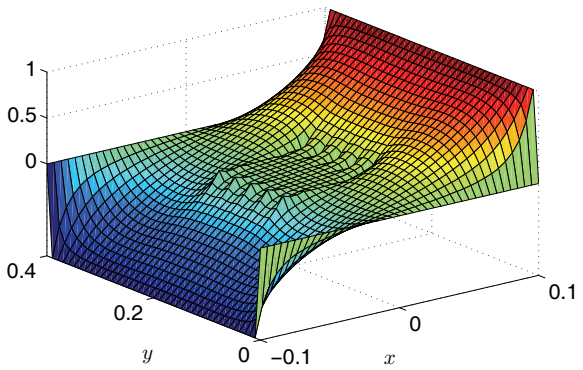


Figure 22.5 The potential $V(x, y)$ for an electrostatic "cage" formed by grounding some interior points. (Problem 22.3.)

Chapter 23

Schrödinger's Equation

Here is the time-dependent Schrödinger equation which governs the way a quantum wave function changes with time in a one-dimensional potential well $V(x)$:[1]

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi \tag{23.1}$$

Note that except for the presence of the imaginary unit i, this is very much like the diffusion equation. In fact, a good way to solve it is with the Crank-Nicolson algorithm. Not only is this algorithm stable for Schrödinger's equation, but it has another important property: it conserves probability. This is very important. If the algorithm you use does not have this property, then as ψ for a single particle is advanced in time you have (after a while) 3/4 of a particle, then 1/2, etc.

P23.1 (a) Derive the Crank-Nicolson algorithm for Schrödinger's equation. It will probably be helpful to use the material in Lab 19 as a guide (beginning with Eq. (19.6)). Schrödinger's equation is simpler in the sense that you don't have to worry about a spatially varying diffusion constant, but make sure the $V(x)\psi$ term enters the algorithm correctly. You should find the following:

$$\psi_j^{n+1}\left(\frac{i\hbar}{\tau} - \frac{\hbar^2}{2mdx^2} - \frac{V(x_j)}{2}\right) + \psi_{j+1}^{n+1}\frac{\hbar^2}{4mdx^2} + \psi_{j-1}^{n+1}\frac{\hbar^2}{4mdx^2} \tag{23.2}$$

$$= \psi_j^{n}\left(\frac{i\hbar}{\tau} + \frac{\hbar^2}{2mdx^2} + \frac{V(x_j)}{2}\right) - \psi_{j+1}^{n}\frac{\hbar^2}{4mdx^2} - \psi_{j-1}^{n}\frac{\hbar^2}{4mdx^2} \tag{23.3}$$

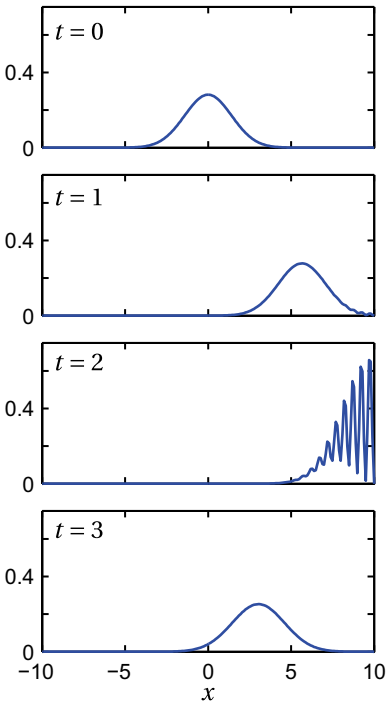


Figure 23.1 The probability density $|\psi(x)|^2$ of a particle in a box that initially moves to the right and then interferes with itself as it reflects from an infinite potential (Problem 23.2(a)).

Particle in a box

Let's use this algorithm for solving Schrödinger's equation to study the evolution of a particle in a box with

$$V(x) = \begin{cases} 0 & \text{if } -L < x < L \\ +\infty & \text{otherwise} \end{cases} \tag{23.4}$$

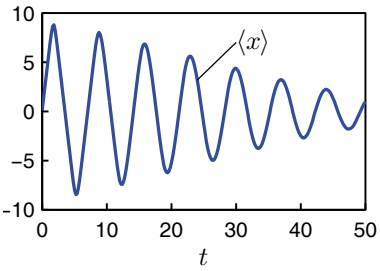


Figure 23.2 The expectation position $\langle x\rangle$ for the particle in Fig. **??** as time progresses and the packet spreads out (Problem 23.2(c)).

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 470-506.

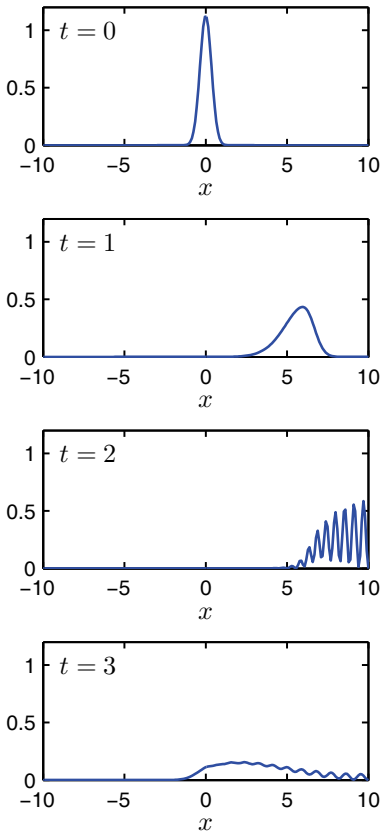


Figure 23.3 The probability density $|\psi(x)|^2$ of a particle that is initially more localized quickly spreads (Problem 23.2(d)).

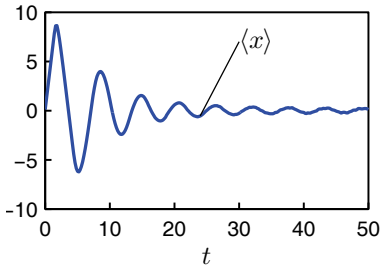


Figure 23.4 The expectation position of the particle in Fig. **??** as time progresses.

The infinite potential at the box edges is imposed by forcing the wave function to be zero at these points:

$$\psi(-L) = 0 \quad ; \quad \psi(L) = 0 \tag{23.5}$$

P23.2 (a) Write a script to solve the time-dependent Schrödinger equation using Crank-Nicolson as discussed in Lab 8. Use natural units in which $\hbar = m = 1$ and $L = 10$. We find that a **cell-edge grid works best**, but you can also do cell-center with ghost points if you like. Start with a localized wave packet of width σ and momentum p:

$$\psi(x,0) = \frac{1}{\sqrt{\sigma\sqrt{\pi}}} e^{ipx/\hbar} e^{-x^2/(2\sigma^2)} \tag{23.6}$$

with $p = 2\pi$ and $\sigma = 2$. This initial condition does not exactly satisfy the boundary conditions, but it is very close.
Run the script with $N = 200$ and watch the particle (wave packet) bounce back and forth in the well. Plot an animation of $\psi^*\psi$ so that you can visualize the probability distribution of the particle. Try switching the sign of p and see what happens.

(b) Verify by doing a numerical integral that $\psi(x,0)$ in the formula given above is properly normalized. Then run the script and check that it stays properly normalized, even though the wave function is bouncing and spreading within the well.

(c) Run the script and verify by numerical integration that the expectation value of the particle position

$$\langle x \rangle = \int_{-L}^{L} \psi^*(x,t)\; x\; \psi(x,t)\, dx \tag{23.7}$$

is correct for a bouncing particle. Plot $\langle x \rangle(t)$ to see the bouncing behavior. Run long enough that the wave packet spreading modifies the bouncing to something more like a harmonic oscillator. (Note: you will only see bouncing-particle behavior until the wave packet spreads enough to start filling the entire well.)

(d) You may be annoyed that the particle spreads out so much in time. Try to fix this problem by narrowing the wave packet (decrease the value of σ) so the particle is more localized. Run the script and explain what you see in terms of quantum mechanics.

Tunneling

Now we will allow the pulse to collide with a non-infinite barrier and study what happens. Classically, the answer is simple: if the particle has a kinetic

energy less than V_0 it will be unable to get over the barrier, but if its kinetic energy is greater than V_0 it will slow down as it passes over the barrier, then resume its normal speed in the region beyond the barrier. (Think of a ball rolling over a bump on a frictionless track.) Can the particle get past a barrier that is higher than its kinetic energy in quantum mechanics? The answer is yes, and this effect is called tunneling.

To see how the classical picture is modified in quantum mechanics we must first compute the energy of our pulse so we can compare it to the height of the barrier. The quantum mechanical formula for the expectation value of the energy is

$$\langle E\rangle = \int_{-\infty}^{\infty} \psi^* H\psi\, dx \tag{23.8}$$

where ψ^* is the complex conjugate of ψ and where

$$H\psi = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi(x) \tag{23.9}$$

In our case $\psi(x)$ is essentially zero at the location of the potential barrier, so we may take $V(x) = 0$ in the integral.

P23.3 Use Mathematica to compute $\langle E\rangle$ for your wave packet. You should find that

$$\langle E\rangle = \frac{p^2}{2m} + \frac{\hbar^2}{4m\sigma^2} \approx 20 \tag{23.10}$$

Since this is a conservative system, the energy remains constant and you can just use the initial wave function in the integral.

HINT: You will need to use the `Assuming` command to specify that certain quantities are real and/or positive to get Mathematica to do the integral.

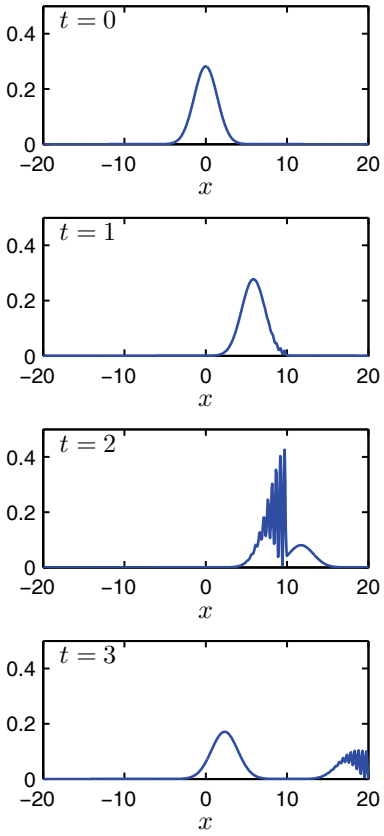


Figure 23.5 The probability distribution $|\psi(x)|^2$ for a particle incident on a narrow potential barrier located just before $x = 10$ with $V_0 > \langle E\rangle$. Part of the wave tunnels through the barrier and part interferes with itself as it is reflected.

H23.4 (a) Modify your script from Problem 23.2 so that it uses a computing region that goes from $-2L$ to $2L$ and a potential

$$V(x) = \begin{cases} 0 & \text{if } -2L < x < 0.98L \\ V_0 & \text{if } 0.98L \le x \le L \\ 0 & \text{if } L < x < 2L \\ \\ +\infty & \text{otherwise} \end{cases} \tag{23.11}$$

so that we have a square potential hill $V(x) = V_0$ between $x = 0.98L$ and $x = L$ and $V = 0$ everywhere else in the well.

Note: Since $V(x)$ was just zero in the last problem, this is the first time to check if your $V(x)$ terms in Crank-Nicolson are right. If you see strange behavior, you might want to look at these terms in your code.

(b) Run your script several times, varying the height V_0 from less than your pulse energy to more than your pulse energy. Overlay a plot of $V(x)/V_0$ on your plot of $|\psi|^2$ and look at the behavior of ψ in the barrier region.

(c) Vary the width of the hill with V_0 bigger than your pulse energy to see how the barrier width affects tunneling.

Chapter 24

Molecular Dynamics

In this lab we will attempt to simulate the motion of a collection of particles that are allowed to interact with each other. To put it rather simply, we're going to put a bunch of particles in a box, give them all velocities and let them bounce around. The particles could be anything, but they are usually atoms or molecules that constitute a solid,liquid, or gas. Molecular dynamic simulations are typically used to investigate the properties of solids and liquids. Firstly, we'll focus our attention on building the simulation. After that is done, we'll try to answer some interesting physics questions

Building this simulation will involve building the biggest and most complex code that you have done all semester. We will attack the problem one step at a time until we have built the entire simulation. Here is a rough outline of what we must accomplish:

1. Initialize the position and velocities of the particles. There are two options here:
   (a) Initialize the particles to have random initial position and velocities.
   (b) Initialize the particles to a periodic arrangement of points (also called a lattice)
2. Iterate over time, updating the positions and velocites of each particle at each iteration. This means that at each iteration we must
   (a) Calculate the force on every particle due to all the other particles that are exerting forces on it.
   (b) Use the force to update the x and y components of the position and velocity vector for each particle.
   (c) Decide what we want to do if a particle moves beyond the simulation box(cell). We could implement periodic boundary conditions (particles leaving one side of the box are entering the other side) or make the walls of the cell be "hard" so that the particles bounce off.

We'll have one section for each point, discussing the relevant issues and building code as we go.

Initialization

There are two ways that you can initialize the positions and velocities of your particles. You can either (i) choose to put the particles at random locations and give them random velocities or you can (ii) place the particles on a regular, periodic array of points. Let's do random first

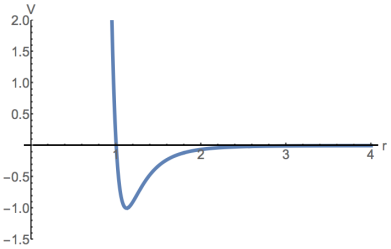


Figure 24.1 The Lennard-Jones potential

Random

The potential that we will be using is called a Lennard-Jones potential (see figure 24.1). Notice that the potential increases very rapidly when the particles get very close. If you decide to give your particles random initial positions and velocites, you'll need to ensure that no two particles are initialized at locations closer than 1 unit apart.

P24.1 Let's start coding

(a) Build a class (call it `MD` or something else that makes sense to you) and construct the class initializer (the `__init__` function) to initialize the following variables

i. the box size (assume a square box)

ii. the number of particles in the box

(b) Now build a function that initilializes the positions to random locations inside of the box, while ensuring that the particles are no closer than 1. Initialize the components of the velocity vectors to be random numbers between $(-10,-10)$. (see figure 24.2) Selecting a random number in a given range can be done like this:

```
from numpy.random import uniform
a = uniform(-10,10,size= 2) # Generates a random vector of length 2 in
                            # the range (-10,10)
```

Warning: There are only so many particles that you can put in a box that satisfy the criteria that no two particles be closer than 1 unit away. If you try putting more particles in your box than is possible, you may get stuck in an infinite loop.

(c) Build a function that plots the locations of all the particles and call the function to visually verify that you accomplished your goal.

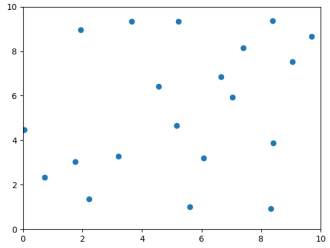


Figure 24.2 A MD simulation cell with particles placed at random locations.

Regular grid of points

We'll also experiment with placing the particles on a regular grid of points. For example, you could choose a square grid, defined by the lattice vectors:

$$\mathbf{v}_1 = 1\hat{i} + 0\hat{j}$$

$$\mathbf{v}_2 = 0\hat{i} + 1\hat{j}$$

If you take multiples of these vectors and add them up, you can arrive at any point on the square grid of points. In other words:

$$\mathbf{r}_{m,m} = n\mathbf{v}_1 + m\mathbf{v}_2 \tag{24.1}$$

The lattice vectors for a triangular lattice are:

$$\mathbf{v}_1 = 1\hat{i} + 0\hat{j}$$

$$\mathbf{v}_2 = 0.5\hat{i} + 0.8666\hat{j}$$

P24.2 Add another function to your class which initializes the positions of your particles to be on a square lattice. (see figure 24.3) The initialization of the velocity vectors is the same as before. **Plot the positions of your particles to see if you have done it correctly.** Here are a few things to watch out for:

(a) Design the function so that you can pass the lattice vectors into the function.

(b) You won't be able to specify how many particles go in the simulation box. Rather the number of particles in your box is determined by the lattice site spacing. So don't try to force a certain number of particles into your box. Just design the function to put particles on the lattice sites until the box is filled.

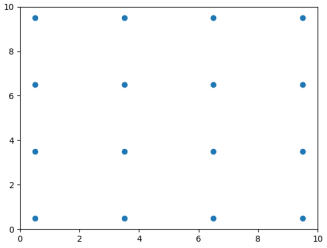


Figure 24.3 An MD simulation cell with particles placed on square lattice sites.

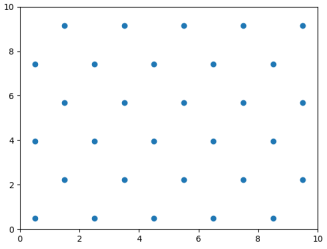


Figure 24.4 An MD simulation cell with particles placed on triangular lattice sites.

Calculating forces

Before we can talk about actually moving particles around, we must understand the forces that the particles are exerting on each other. As mentioned previously, we are using a Lennard-Jones potential (see figure 24.1), which is given by the following function:

$$V(r) = 4\left[\left(\frac{1}{r}\right)^{12} - \left(\frac{1}{r}\right)^{6}\right] \tag{24.2}$$

. As you can see, the potential is attractive for large particle separation and gets very large if the particles ever get very close to one another.

You may recall that force and potential are related via:

$$\mathbf{F} = -\nabla V \tag{24.3}$$

P24.3 Evaluate equation 24.3 by taking a derivative of equation 24.2 with respect to r. You should find that:

$$F(r) = 24\left(\frac{2}{r^{13}} - \frac{1}{r^{7}}\right) \tag{24.4}$$

Equation 24.4 is the magnitude of the force. The components can be expressed as:

$$F_x = 24\left(\frac{2}{r^{13}} - \frac{1}{r^{7}}\right)\frac{x}{|\mathbf{r}|} \qquad F_y = 24\left(\frac{2}{r^{13}} - \frac{1}{r^{7}}\right)\frac{y}{|\mathbf{r}|} \tag{24.5}$$

P24.4 Add a function to your class that calculates the **net** force on a given particle and returns it's force vector . When you are done, **call your function for several particles and visuallly verify that the returned force vectors make sense.** Here are some hints:

(a) Pass in the position vector of the particle of interest as an argument to this function.

(b) You'll need to loop over your list of particles and, one-by-one, calculate 24.5 for the particle in question. You won't want to include the force of the particle on itself (obviously) and you can neglect forces from particles that are more than 4 length units away (force gets small fast).

(c) You'll want to code this function to work for both periodic boundary conditions and hard-wall conditions. For periodic boundary conditions there is the possibility that two particles that seem to be very far apart are actually quite close. Think about how to handle this.

Animating

Now we're ready to make the particles move and interact with each other. It may be tempting to reach for Euler's method to update the positions and velocities of the particles. However, in molecular dynamics situations, you will be performing a very large number of iterations. Numerical errors associated with Euler's method are too large and will compound over the course of the simulation. Instead, we will use a method very similar to the leapfrog method that we studied in chapter 8. Let's start with the leapfrog equations (found in equations (8.6) through (8.8)), but written in terms of position and velocity:

$$x_{n+\frac{1}{2}} = x_n + \frac{\tau}{2} v(x_n, t_n) \quad \text{(Euler's step)} \tag{24.6}$$

$$x_{n+1} = x_n + \tau\, v(x_{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) \quad (2^{\text{nd}} \text{ Order Runge} - \text{Kutta}) \tag{24.7}$$

$$x_{n+\frac{3}{2}} = x_{n+1} + \frac{\tau}{2} v(x_{n+1}, t_{n+1}) \tag{24.8}$$

Recall that with this method, we had to calculate both x and v at every full time step **and** at every half time step. This was because a (the right hand side of $\frac{dv}{dt}$) depended on v. In our case, that dependency is missing, which means we ought to be able to reduce the number of steps needed.

Recall that the Euler step we took was needed so that we could calculate the velocity one half step forward in time. This velocity was then used to calculate the the position one full step forward in time. In our case, the velocity update doesn't involve position and therefore, the first half step for the position is not needed. In other words, because the differential equations that we are solving have the following structure (i.e. no v_x dependence on $\frac{dv_x}{dt}$, no v_y dependence on $\frac{dv_y}{dt}$, no x dependence on $\frac{dx}{dt}$, and no y dependence on $\frac{dy}{dt}$):

$$\frac{dx}{dt} = v_x \qquad \frac{dv_x}{dt} = \frac{F_x}{m} \tag{24.9}$$

$$\frac{dy}{dt} = v_y \qquad \frac{dv_y}{dt} = \frac{F_y}{m} \tag{24.10}$$

we can just iterate over the equations below, finding the velocities at half time steps and positions at full time steps:

$$v_{n+\frac{1}{2}} = v_n + \frac{\tau}{2} a(x_n, t_n) \quad \text{(Euler's step)} \tag{24.11}$$

$$x_{n+1} = x_n + \tau\, v(t_{n+\frac{1}{2}}) \quad (2^{\text{nd}} \text{ Order Runge} - \text{Kutta}) \tag{24.12}$$

$$v_{n+\frac{3}{2}} = v_{n+\frac{1}{2}} + \tau\, a(x_{n+1}, t_{n+1}) \quad (2^{\text{nd}} \text{ Order Runge} - \text{Kutta}) \tag{24.13}$$

$$\tag{24.14}$$

This method is called the **Verlet** method and it is equivalent to the leapfrog method we learned in chapter 8, except that we are not calculating the position at half time steps or the velocities at full time steps because we don't need them. Because we have reduced the number of calculations, this method should be faster. Notice also that we are using a centered derivative for all updates except the very first one, where we needed an Euler step to get us going.

P24.5 Let's build the animate function now

(a) Build a member function called `animate` that implements the Verlet algorithm. Assume that the masses of the particles are 1

(b) After the positions and velocities have been updated, you'll need to enforce the boundary conditions. Write one last function that checks to see if a particle has left the simulation box. Then think about how you want to modify that particles position and velocity for both types of boudary conditions.

Using the model

Now that you have a working molecular dynamics model, we should try to use it to answer fun physics questions. There are many questions that you could ask, but for today, let's try to observe (and detect) phase transitions from gas to liquid and then a solid. To do this, we need a way to cool our collection of particles. One straightforward way to do this is to periodically rescale the velocities according to

$$v \to 0.95 v \tag{24.15}$$

Over time, the velocities of the particles will decrease and hence the temperature will go down.

Homework

H24.6 Modify your code to start out hot and gradually cool down so that you can observe transitions from gas to liquid and finally to solid. Follow the guidelines below:

(a) Begin by randomly placing 16 particles in a 4.3 x 4.3 box and giving them random velocities from the following distribution:

```
velocities = choice([-1,1],size = 2)* normal(100,1,size = 2)
```

Can you figure out what this line of code is doing? Play around with it until you understand.

(b) Use equation (24.15) to cool your system periodically. Be careful to ensure that you don't cool the system too fast. You need to let the system equilibriate at one temperature before you move to the next

(c) One way to detect the phase transition is to watch your simulation and see when particles begin to coalesce. Watch your simulation and see if you can observe the transitions.

(d) We need a more quantitative way to spot the transitions. One way would be to pick any two particles and track the square of their separation over time:

$$\Delta r = |\mathbf{r}_1 - \mathbf{r}_2| \tag{24.16}$$

When the system is a liquid, the separation should flucuation in a large range. Once the system transitions to a solid, the particles will remain nearly stationary and thus you would expect the separation of the two particles to remain fairly constant. By looking at a plot of Δr^2 vs. time, you should be able to spot the transition. Incorporate this idea into your code and see if you can use it to spot the solid-liquid transition.

Chapter 25

Ideas for student projects

H25.1 (Normal Modes) In chapters 11 and 12 we studied eigenvalue problems which typically arise when finding the normal modes of vibration for bound system. We did this for horizontal strings bound at both ends and for the hanging chain. Use the ideas from this chapter to find the normal modes of vibration for the square membranes that we studied in chapter 16. Then return to your code from chapter 16 for animating the square membrane and see if you can observe the standing waves.

H25.2 (Shooting Method) In chapter 10 we studied boundary value problems and chose to solve them by forming the problem into a linear algebra problem. There is an alternate way to solve boundary value problems called the shooting method. In the shooting method, you pretend that a boundary value problem is an initial value problem. Here is an outline of the method:

(a) The value of the function at the left boundary **is known**(it's given) and the slope of the function at the left boundary **is guessed**!

(b) Runge-Kutta (or another integration method) is used to solve the problem out to the right boundary.

(c) The function value at the right boundary is then checked to see if it matches the boundary condition.

(d) If it does, the algorithm terminates.

(e) If it doesn't, the initial slope is modified and another solution is found.

Pick a problem from chapter 10 and use the shooting method to solve it.

H25.3 (3D Random Walker) Simulate a random walk in three dimensions allowing the walker to make steps of unit length but in random directions. (i.e. Don't restrict your walker to only taking steps in the $\hat{x}, \hat{y}, \hat{z}$ directions.) Make a movie of the diffusion. By fitting $\langle r^2 \rangle$ vs t data, find the diffusion constant. The value that you calculated for the diffusion constant should (theoretically) be $\frac{1}{2}$, but it's not. Instead, these calculated values fluctuate following a gaussian distribution. Apply the χ^2 (should know from PH336) test to a set of diffusion constants to verify that they come from a normal distribution

H25.4 (Kepler's second Law) Modify your planetary motion code to verify Kepler's second law. Do it for more than one planet.

H25.5 (KnuckleBall) A knuckleball is a baseball pitch where the the ball is spun very slowly. For example, consider a baseball that is thrown so that the

stitching is on the right side of the ball and no stitching is on the left side. There will be an imbalance of drag forces that will push the ball leftward. If the ball rotates very slowly, for example, just enough so that the stitching on the ball faces left after it has traveled $\frac{1}{4}$ of the way to the plate, then the force on the ball will change directions. The net affect on the ball can be a very erratic back-and-forth motion. Let's assume that the lateral force on the ball is given by:

$$F = \frac{1}{2} mg \left[\sin(4\theta) - 0.25\sin(8\theta) + 0.08\sin(12\theta) - 0.025\sin(16\theta)\right] \quad (25.1)$$

Model the motion of a knuckleball and investigate the effect of ω on the motion.

H25.6 (Circular Membrane) In chapter 16 we solved the wave equation for a square membrane. Now solve the wave equation for a circular membrane. The wave equation in polar coordinates is given by:

$$\mu \frac{\partial^2 u}{\partial t^2} = \sigma \left[\frac{1}{r} \frac{\partial}{\partial r} \left(r \frac{\partial u}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} \right] \quad (25.2)$$

Fix the edges of the membrane and give the membrane an initial displacement in the shape of a Gaussian, just like we did in chapter 16. Here are some hints:

(a) You should use a cell-centered grid with ghost points for this problem. A good exercise would be to try a cell-edge grid and see what problems you run into.

(b) Discretize the differential equation using center-difference derivatives.

(c) For simplicity, you may set `yold` equal to the initial disturbance to get the animation started.

(d) In chapter 16, the array `z` was used to store the displacements of the points on the membrane. The columns of the arrays represented the y-axis and the rows of the array represented the x-axis. Now, since you are working in polar coordinates, let the columns represent θ and the rows represent r. This means that the first row represents $r = 0 - \frac{dr}{2}$ (ghostpoint) and the last row represents $r = R + \frac{dr}{2}$ (ghost point). The first column represents $\theta = 0 - \frac{d\theta}{2}$ and the last column represents $\theta = 2\pi + \frac{d\theta}{2}$.

H25.7 (2D Heat) We solved the **one-dimensional** heat equation in chapters 17 and 18. Now solve it for a two-dimensional square plate. Pick your own initial and boundary conditions.

H25.8 (2D Waves) In chapter 15 we solved the undamped, undriven, two-dimensional wave equation. Add damping and a periodic driving force to this problem and build an animation of the vibration.

H25.9 (**Lightning Rod**) Calculate the electric potential and field near a lightning rod. Model this as a very long and narrow metal rod held at high voltage, with one end near a conducting plane. Of special interest is the field near the tip of the rod.

H25.10 (**Equipartition Theorem**) The equipartition theorem states that, on average, each quadratic degree of freedom in a system has the same amount of energy. Let's consider a system that has nonquadratic degrees of freedom and see how the energy is shared. Consider a system of $N = 32$ masses, each connected to each other by springs. Let the force of the spring on the masses be given by:

$$F(x) = -kx - \beta x^3 \tag{25.3}$$

We call this a non-linear system because the spring forces are not a linear function of the displacement. Perform the following:

(a) Use the leapfrog algorithm (see chapter 8) to build an animation of the masses. Before you begin coding, you'll want to write down Newton's second law for the first few masses so you can begin to see the pattern. The initial displacements of the masses is given by:

$$x_i^0 = A\sqrt{\frac{2}{N+1}}\sin(\frac{im\pi}{N+1}) \tag{25.4}$$

where A is the amplitude of vibration, N is the number of oscillators, and m is the mode number.

(b) The linear system ($\beta = 0$) has a family of normal modes given by equation (25.4) and if the system starts out in one of these modes, it will remain in it indefinitely. How will the nonlinear system evolve in time if we start it out in one of it's normal modes? To answer this question, we need to track the modal spectrum over time. This means we should perform a discrete fourier transform of the oscillator displacements **at each moment in time** to obtain the set of a_k. The energy content of a given mode is given by:

$$E_k = \frac{1}{2}\left[\left(\frac{da_k}{dt}\right)^2 + \omega_k^2 a_k^2\right] \tag{25.5}$$

with

$$\omega_k = 2\sin\left(\frac{k\pi}{2(N+1)}\right) \tag{25.6}$$

Calculate the energy for modes $k = 1...5$ as a function of time and plot.

(c) Invesitgate what happens when you vary β, the constant in front of the non-linear term. Compare to the case when $\beta = 0$ (linear springs).

H25.11 (Violin String) In chapter 15 we used staggered leapfrog to animate the undamped, undriven wave equation with fixed ends. Now try and modify that code to model the motion of a violin string that is driven by a bow. You'll need to consider the force of static and kinetic friction between the bow and the violin string to determine whether the string slips or sticks to the bow

H25.12 (Spectral Methods) In chapter 15 we used staggered leapfrog to animate the undamped, undriven wave equation with fixed end. An alternate approach to animating the wave equation is to perform a discrete fourier transform on the **initial waveform** and then propagate it forward in time using the following equation:

$$y(x,t) = \sum_{k=0}^{N/2} \left[\alpha_k \cos(\frac{2\pi x}{\lambda_k} + \beta_k \sin(\frac{2\pi x}{\lambda_k}) \right] \cos(2\pi f_k t) \tag{25.7}$$

Try this and see if you can get the wave to propagate in exactly the same way that they did with staggered leapfrog. Now try truncating your fourier coefficents (only include those coefficients that are greater than some cutoff) and see how it affects the propagation.

H25.13 (Earth-Sun-Moon-Jupiter system) Model the Earth-Moon-Sun-Jupiter system and investigate the affect of Jupiter on the orbit of Earth's moon.