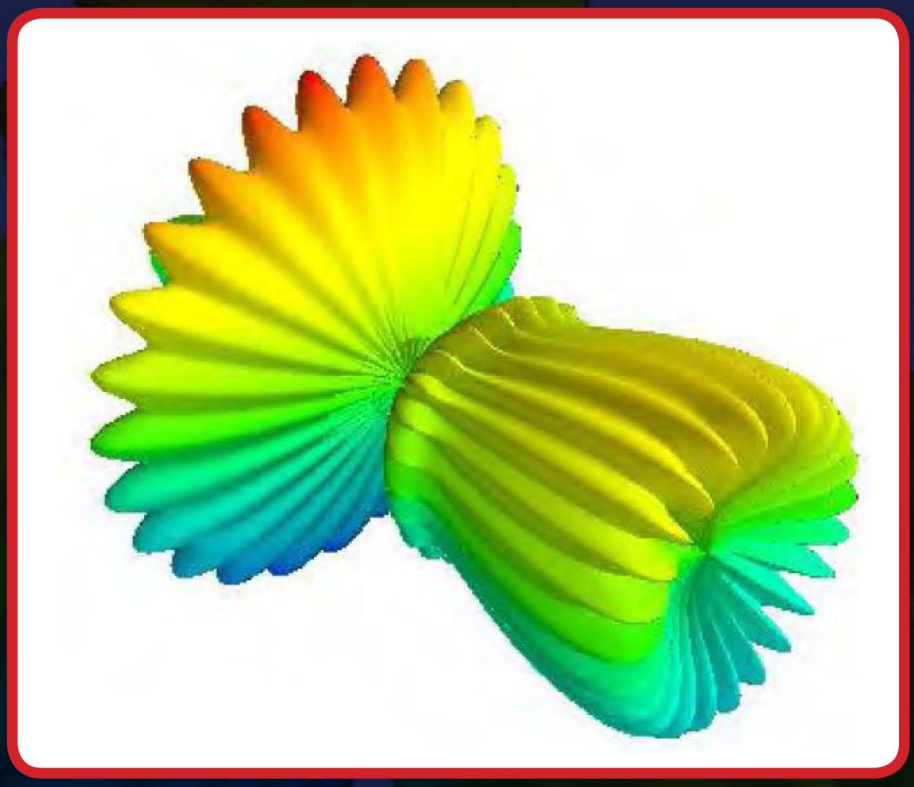
SERIES IN COMPUTATIONAL PHYSICS

Steven A. Gottlieb and Rubin H. Landau, Series Editors

Computational Problems for Physics

With Guided Solutions Using Python



Rubin H. Landau

Manuel José Páez

CRC Press

Taylor & Francis Group

Computational Problems for Physics

Computational Problems for Physics

With Guided Solutions Using Python

Rubin H. Landau, Manuel José Páez

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20180414

International Standard Book Number-13: 978-1-1387-0541-8 (Paperback)
International Standard Book Number-13: 978-1-1387-0591-3 (Hardback)

Contents

Acknowledgments

Thank you, former students who were our experimental subjects as we developed computational physics courses and problems over the years. And special thanks to you who have moved on with your lives, but found the time to get back to tell us how much you have benefitted from our Computational Physics courses; bless you!

We have tried to give credit to the authors whose books have provided motivation and materials for many of this book's problems. Please forgive us if we have forgotten to mention you; after all, 20 years is a long time. In particular, we have probably made as our own materials from the pioneering works of Gould & Tobochnik, Koonin, and Press et al..

Our lives have been enriched by the invaluable friendship, encouragement, helpful discussions, and experiences we have had with many colleagues and students over the years. With uncountable sadness we particularly remember our deceased colleagues and friends Cristian Bordeianu and Jon Maestri, two of the most good-natured, and good, people you could have ever hoped to know. We are particularly indebted to Paul Fink (deceased), Hans Kowallik, Guillermo Avendaño-Franco, Saturo S. Kano, David McIntyre, Shashi Phatak, Oscar A. Restrepo, Jaime Zuluaga, Viktor Podolskiy, Bruce Sherwood, C. E. Yaguna, and Zlatco Dimcovic for their technical help and most of all their friendship.

The authors wish to express their gratitude to Lou Han, for the encouragement and support throughout the realization of this book, and to Titus Beu and Veronica Rodriguez who have helped in the final production.

Finally, we extend our gratitude to our families, whose reliable support and encouragement are lovingly accepted, as always.

Series Preface

There can be little argument that computation has become an essential element in all areas of physics, be it via simulation, symbolic manipulations, data manipulations, equipment interfacing, or something with which we are not yet familiar. Nevertheless, even though the style of teaching and organization of subjects being taught by physics departments have changed in recent times, the actual content of the courses has been slow to incorporate the new-found importance of computation. Yes, there are now speciality courses and many textbooks in *Computational Physics*, but that is not the same thing as incorporating computation into the very heart of a modern physics curriculum so that the physics being taught today more closely resembles the physics being done today. Not only will such integration provide valuable professional skills to students, but it will also help keep physics alive by permitting new areas to be studied and old problems to be solved.

This series is intended to provide undergraduate and graduate level textbooks for a modern physics curriculum in which computation is incorporated within the traditional subjects of physics, or in which there are new, multidisciplinary subjects in which physics and computation are combined as a "computational science." The level of presentation will allow for their use as primary or secondary textbooks for courses that wish to emphasize the importance of numerical methods and computational tools in science. They will offer essential foundational materials for students and instructors in the physical sciences as well as academic and industry professionals in physics, engineering, computer science, applied math, and biology.

Titles in the series are targeted to specific disciplines that currently lack a textbook with a computational physics approach. Among these subject areas are condensed matter physics, materials science, particle physics, astrophysics, mathematical methods of computational physics, quantum mechanics, plasma physics, fluid dynamics, statistical physics, optics, biophysics, electricity and magnetism, gravity, cosmology, and high-performance computing in physics. We aim for a presentation that is concise and practical, often including solved problems and examples. The books are meant for teaching, although researchers may find them useful as well. In select cases, we have allowed more advanced, edited works to be included when they share the spirit of the series — to contribute to wider application of computational tools in the classroom as well as research settings.

Although the series editors had been all too willing to express the need for change in the physics curriculum, the actual idea for this series came from the series manager, Lou Han, of Taylor & Francis Publishers. We wish to thank him sincerely for that, as well as for encouragement and direction throughout the project.

Steve Gottlieb, *Bloomington*
Rubin H. Landau, *Corvallis*
Series Editors

Preface

As seems true in many areas, practicing scientists now incorporate powerful computational techniques as key elements in their work. In contrast, physics courses often include computational tools only to illustrate the physics, with little discussion of the method behind the tools, and of the limits to a simulation's reliability and precision. Yet, just as a good researcher would not believe a physics results if the mathematics behind it were not solid, so we should not believe a physics results if the computation behind it is not understood and reliable. While specialty courses and textbooks in Computational Physics are an important step in the right direction, we see an additional need to incorporate modern computational techniques throughout the Physics curriculum. In addition to enhancing the learning process, computational tools are valuable tools in their own right, especially considering the broad areas in which physics graduates end up working.

The authors have spent over two decades trying to think up computational problems and demonstrations for physics courses, both as part of the development of our Computational Physics texts, and as material to present as tutorials at various professional meetings and institutions. This book is our effort at collecting those problems and demos, adding to them, and categorizing them so that they may extend what has traditionally been used for homework and demonstrations throughout the physics curriculum.

Our assumed premise is that learning to compute scientifically requires you to get your hands dirty and to open up that black box of a program. Our preference is that the reader use a compiled language since this keeps her closer to the basic algorithms, and more likely to be able to estimate the numerical error in the answer (essential for science). Nevertheless, programming from scratch can be time consuming and frustrating, and so we provide many sample codes as models for the problems at hand. However, readers or instructors may prefer to approach our problems with a problem solving environment such as Sage, Maple, or Mathematica, in which case our codes can serve as templates.

We often present simple pseudocodes in the text, with full Python code listings at the end of each chapter (most numeric, but some symbolic).[1] The Python language

[1] Please note that copying and pasting a code from a pdf listing is not advisable because the formatting, to which Python is sensitive, is not preserved in the process. One needs to open the `.py`

plus its family of packages comprise a veritable ecosystem for computing [CiSE(07,11)]. Python is free, robust, portable, universal, and provides excellent visualization via the *MatPlotLib* and *VPython* packages (the latter also called by its original name *Visual*). We find Python the easiest compiled language for education, with excellent applications in research and development. Further details are provided in Chapter 1.

Each chapter in the text contains a Chapter Overview with highlights as to what is to follow. Chapters 1 and 2 review background materials used throughout the rest of the book. Chapter 1 covers basic computational methods, including floating point numbers and their errors, integration, differentiation, random numbers generation, and the solution to ordinary and partial differential equations. Chapter 2 covers fundamental numerical analysis tools, including Fourier analysis, noise reduction, wavelet analysis, principal components analysis, root searching, least-squares fitting, and fractal dimension determination. Although some of the problems and demos in Chapters 1 and 2 may find use in Mathematical Methods of Physics courses, those chapters are meant as review or reference.

This book cover multiple areas and levels of physics with the chapters organized by subject. Most of the problems are at an upper-division undergraduate level, which should by fine for many graduate courses as well, but with a separate chapter aimed at entry-level courses. We leave it to instructors to decide which problems and demos may work best in their courses, and to modify the problems for their own purposes. In all cases, the introductory two chapters are important to cover initially.

We hope that you find this book useful in changing some of what is studied in physics. If you have some favorite problems or demos that you think would enhance the collection, or some suggestions for changes, please let us know.

RHL, rubin@science.oregonstate.edu Tucson, November 2017
MJP, mpaezenator@gmail.com Medellín, November 2017

version of the code with an appropriate code editor.

About the Authors

Rubin Landau is a Distinguished Professor Emeritus in the Department of Physics at Oregon State University in Corvallis and a Fellow of the American Physical Society (Division of Computational Physics). His research specialty is computational studies of the scattering of elementary particles from subatomic systems and momentum space quantum mechanics. Dr. Landau has taught courses throughout the undergraduate and graduate curricula, and, for over 20 years, in computational physics. He was the founder of the OSU Computational Physics degree program, an Executive Committee member of the APS Division of Computational Physics, and the AAPT Technology Committee. At present Dr. Landau is the Education co-editor for AIP/IEEE *Computing in Science & Engineering* and co-editor of this Taylor & Francis book series on computational physics. He has been a member of the XSEDE advisory committee and has been part of the Education Program at the SuperComputing (SC) conferences for over a decade.

Manuel José Páez-Mejia has been a Professor of Physics at Universidad de Antioquia in Medellín, Colombia since January 1969. He has been teaching courses in Modern Physics, Nuclear Physics, Computational Physics, Numerical Methods, Mathematical Physics, and Programming in Fortran, Pascal, and C languages. He has authored scientific papers in nuclear physics and computational physics, as well as texts on the C Language, General Physics, and Computational Physics (coauthored with Rubin Landau and Cristian Bordeianu). In the past, he and Dr. Landau conducted pioneering computational investigations of the interactions of mesons and nucleons with few-body nuclei. Professor Paez has led workshops in Computational Physics throughout Latin America, and has been Director of Graduate Studies in Physics at the Universidad de Antioquia.

Web Materials

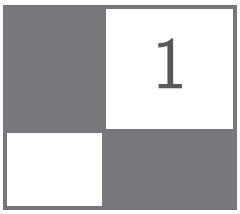
The Python codes listed in the text are available on the CRC Press website at
https://www.crcpress.com/Computational-Problems-for-Physics-With-Guided-Solutions-Using-Python/Landau-Paez/p/book/9781138705418.
We have also created solutions to many problems in a variety of computer languages, and they, as well as these same Python codes, are available on the Web at
http://science.oregonstate.edu/~landaur/Books/CPbook/eBook/Codes/.
Updates of the programs will be posted on the websites.

Background material for this book of problems is probably best obtained from Computational Physics text books (particularly those by the authors!). In addition, Python notebook versions of every chapter in our CP text [LPB(15)] are available at
http://physics.oregonstate.edu/~landaur/Books/CPbook/eBook/Notebooks/.
As discussed in the documentation there, the notebook environment permits the reader to run codes and follow links while reading the book electronically.

Furthermore, most topics from our CP text are covered in video lecture modules at
http://science.oregonstate.edu/~landaur/Books/CPbook/eBook/Lectures/.

General System Requirements

The first chapter of the text provides details about the Python ecosystem for computing and the packages that we use in the text. Basically, a modern version of Python and its packages are needed.



Computational Basics for Physics

1.1 Chapter Overview

There is no way that a single chapter or two can provide the background necessary for the proper use of computation in physics. So let's hope that this chapter is just a review. (If not, you may want to look at some of the related video lecture modules at http://physics.oregonstate.edu/~landaur/Books/CPbook/eBook/Lectures/.) In this chapter we cover computing basics, starting with some of the tools available as part of the Python ecosystem, and particularly for visualization and matrix manipulations. There follows a discussion of number representations and the limits and consequences of using floating-point numbers (often absent in Computer Science classes). We then review some basic numerical methods for differentiation, integration, and random number generation. We end with a discussion of the algorithms for solving ordinary and partial differential equations, techniques used frequently in the text. Problems are presented for many of these topics, and doing them would be a good way to get started!

Most every problem in this book requires some visualization. Our sample programs tend to do this with either the Matplotlib or VPython (formerly Visual) package, or both (§1.2.1). Including visualization in the programs does make them longer, though having embedded visualization speeds up the debugging and learning processes, and is more fun. In any case, the user always has the option of outputting the results to a data file and then visualizing them separately with a program such as gnuplot or Grace.

1.2 The Python Ecosystem

This book gives solution codes in Python, with similar codes in other languages available on the Web. Python is free, robust, portable, and universal, and we find it the easiest compiled language for education. It contains high-level, built-in data types, which make matrix manipulations and graphics easy, and there are a myriad of free packages and powerful libraries which make it all around excellent for scientific work, even symbolic manipulation. For learning Python, we recommend the

online tutorials [Ptut(14), Pguide(14), Plearn(14)], the books by Langtangen [Langtangen(08), Langtangen(09)], and the *Python Essential Reference* [Beazley(09)].

The Python language plus its family of packages comprise a veritable ecosystem for computing [CiSE(07,11)]. To include package `PackageName` in your program, you use either an `import PackageName` statement, which loads the entire package, or to load a specific method include a `from PackageName` statement at the beginning of your program; for example,

```
from vpython import *
y1 = gcurve(color = blue)
```

In our work we use the **packages**

Matplotlib (Mathematics Plotting Library)	`http://matplotlib.org`
NumPy (Numerical Python)	`http://www.numpy.org/`
SciPy (Scientific Python)	`http://scipy.org`
SymPy (Symbolic Python)	`http://sympy.org`
VPython (Python with Visual package)	`http://vpython.org/`

Rather than search the Web for packages, we recommend the use of *Python Package Collections*, which are collections of Python packages that have been engineered and tuned to work well together, and that can be installed in one fell swoop. We tend to use

Anaconda	`https://store.continuum.io/cshop/anaconda/`
Enthought Canopy	`https://www.enthought.com/products/canopy/`
Spyder (in Anaconda)	`https://pythonhosted.org/spyder/`

1.2.1 Python Visualization Tools

VPython, the nickname for Python plus the Visual package, is particularly useful for creating 3-D solids, 2-D plots, and animations. In the past, and with some of our programs, we have used the "classic" version of VPython, which is accessed via importing the module `visual`. That version will no longer be supported in the future and so we have (mostly) converted over to VPython 7, but have included the classic (VPython 6) versions in the `Codes` folder as well[1]. The Visual package is accessed in VPython 7 by importing the module `vpython`. (Follow VPython's online instructions to load VPython 7 into Spyder or notebooks.)

In Figure 1.1 we present two plots produced by the program `EasyVisualVP.py` given in Listing 1.1.[2] Notice that the plotting technique with VPython is to create first a

[1] For some programs we provide several versions in the Codes folder: those with a "Vis" suffix use the classic Visual package, those with a "VP" suffix use the newer VPython package, and those with "Mat", or no suffix, use Matplotlib.

[2] We remind the reader that copying and pasting a program from a pdf listing is not advisable because the formatting, to which Python is sensitive, is not preserved in the process. One needs to open the `.py` version of the program with an appropriate code editor.

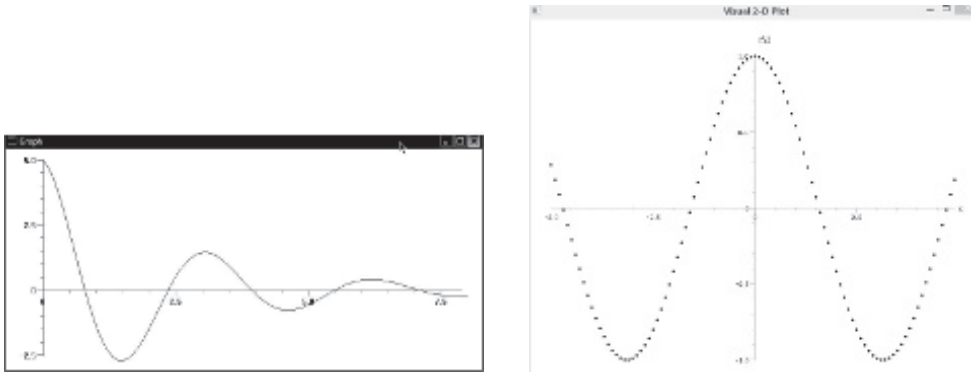


Figure 1.1. Screen dumps of two x-y plots produced by our program `EasyVisualVP.py` using the VPython package. The *left* plot uses default parameters while the *right* plot uses user-supplied options.

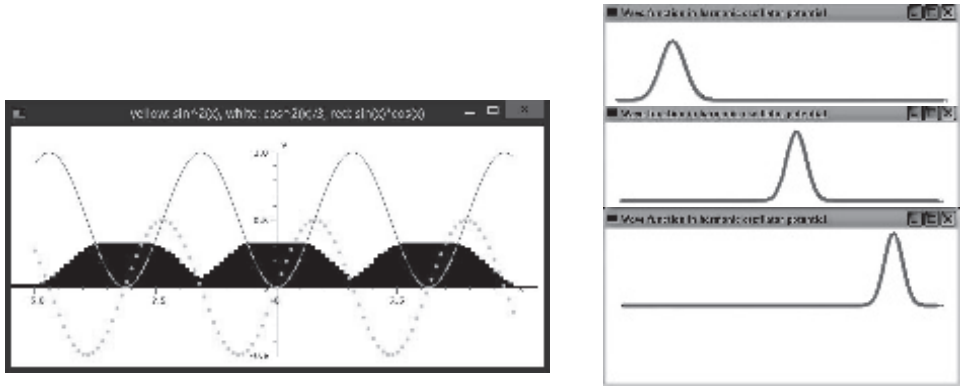


Figure 1.2. *Left:* Output from the program 3GraphVP.py that places three different types of 2-D plots on one graph using VPython. *Right* Three frames from a VPython animation of a quantum mechanical wave packet produced with HOmov.py.

plot object, and then to add the points one at a time to the object. In contrast, Matplotlib creates a vector of points and plots the entire vector in one fell swoop.

The program `3GraphVP.py` in Listing 1.2 places several plots in the same figure and produces the graph on the left of Figure 1.2. There are vertical bars created with `gvbars`, dots created with `gdots`, and a curve created with `gcurve` (colors appear only as shades of gray in the paper text). Creating animations with VPython is essentially just making the same 2-D plot over and over again, with each one at a slightly differing time. Three frames produced by `HOmov.py` are shown on the right of Figure 1.2. The part which makes the animation is simple:

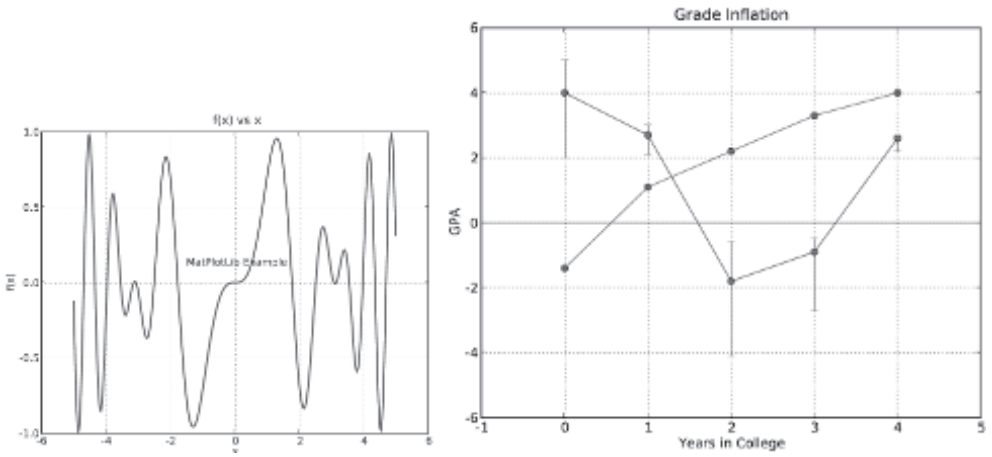


Figure 1.3. Matplotlib plots. *Left:* Output of EasyMatPlot.py showing a simple, x-y plot. *Right:* Output from GradesMatPlot.py that places two sets of data points, two curves, and unequal upper and lower error bars, all on one plot.

```
PlotObj= curve(x=xs, color=color.yellow, radius=0.1)
...
while True:
   rate(500)
   RePsi[1:-1] =...
   ImPsi[1:-1] =..
   PlotObj.y = 4*(RePsi**2 + ImPsi**2)
```

The package Matplotlib is a powerful plotting package for 2-D and 3-D graphs and data plots of various sorts. It uses the sophisticated numerics of NumPy and LAPACK [Anderson et al.(113)] and commands similar to MATLAB^{TM}. It assumes that you have placed the x and y values you wish to plot into 1-D arrays (vectors), and then plots these vectors with a single call. In `EasyMatPlot.py`, given in Listing 1.3, we import Matplotlib as the `pylab` library:

```
from pylab import *
```

Then we calculate and input arrays of the x and y values

```
x = arange(Xmin, Xmax, DelX)   #  x array in range + increment
y =  -sin(x)*cos(x)    # y array as function of x array
```

where the # indicates the beginning of a comment. As you can see, NumPy's `arange` method constructs an array covering "a range" between `Xmax` and `Xmin` in steps of `DelX`. Because the limits are floating-point numbers, so too will be the individual x_i's. And because `x` is an array, `y = -sin(x)*cos(x)` is automatically one too! The actual plotting is performed with a dash '-' used to indicate a line, and `lw`=2 to set its width.

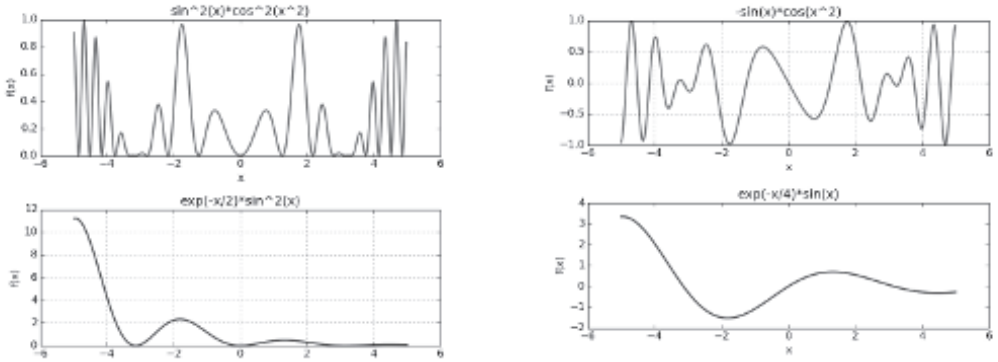


Figure 1.4. *Left* and *Right* columns show two separate outputs, each of two figures, produced by `MatPlot2figs.py`. (We used the slider button to add some space between the red and blue plots.)

The result is shown on the left of Figure 1.3, with the desired labels and title. The `show()` command produces the graph on your desktop.

In Listing 1.4 we give the code `GradesMatplot.py`, and on the right of Figure 1.3 we show its output. Here we repeat the `plot` command several times in order to plot several data sets on the same graph and to plot both the data points and the lines connecting them. We import Matplotlib (pylab), and then import NumPy, which we need for the `array` command. Because we have imported two packages, we add the `pylab` prefix to the `plot` commands so that Python knows which package to use. A horizontal line is created by plotting an array with all y values equal to zero, unequal lower and upper error bars are included as well as grid lines.

Often the science is clearer if there are several curves in one plot, and, several plots in one figures. Matplotlib lets you do this with the `plot` and the `subplot` commands. For example, in `MatPlot2figs.py` in Listing 1.5 and Figure 1.4, we have placed two curves in one plot, and then output two different figures, each containing two plots. The key here is repetition of the `subplot` command:

```
figure(1)                                                    #  1st figure
subplot(2,1,1)        # 1st subplot, 2 rows, 1 column
subplot(2,1,2)                                        # 2nd subplot
```

If you want to visualize a function like the dipole potential

$$V(x,y) = [B + C/(x^2 + y^2)^{3/2}]x, \tag{1.1}$$

you need a 3-D visualization in which the mountain height $z = V(x,y)$, and the x and y axes define the plane below the mountain. The impression of three dimensions is obtained by shading, parallax, and rotations with the mouse, and other tricks. In

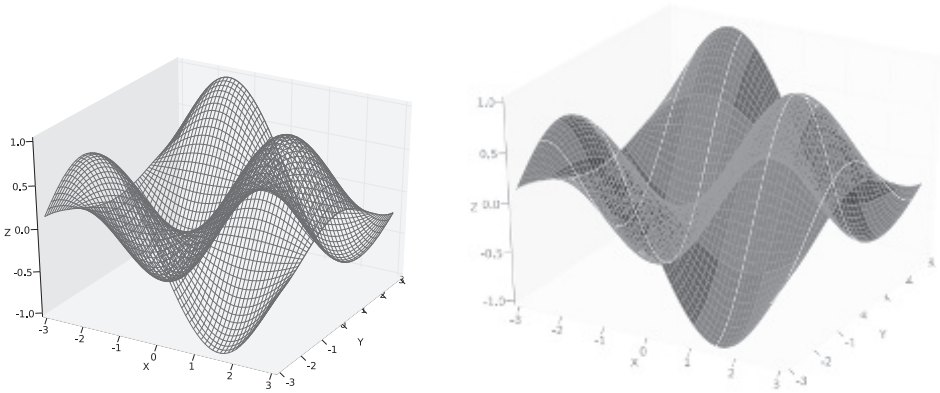


Figure 1.5. *Left:* A 3-D wire frame. *Right:* a surface plot with wire frame. Both are produced by the program `Simple3Dplot.py` using Matplotlib.

Figure 1.5 left we show a wire-frame plot and in Figure 1.5 right a surface-plus-wire-frame plot. These are obtained from the program `Simple3Dplot.py` in Listing 1.6. The `meshgrid` method sets up grid matrix from the x and y coordinate vectors, and then constructs the $Z(x, y)$ surface with another vector operation.

A *scatter plot* is a useful way to visualize individual points (x_i, y_j, z_k) in 3-D. In Figure 1.6 left we show two such plots created with `PondMatPlot.py` in Listing 1.7 and `Scatter3dPlot.py` in Listing 1.8. Here the 111 indicates a $1 \times 1 \times 1$ grid.

1. As shown in Figure 1.7, a beam of length $L = 10$ m and weight $W = 400$ N rests on two supports a distance $d = 2$ m apart. A box of weight $W_b = 800$ N, initially above the left support, slides frictionlessly to the right with a velocity v=7 m/s.
   a. Write a program that calculates the forces exerted on the beam by the right and left supports as the box slides along the beam.
   b. Extend your program so that it creates an animation showing the forces and the position of the block as the box slides along the beam. In Listing 1.10 we present our code `SlidingBox.py` that uses the Python Visual package, and in Figure 1.7 left we present a screen shot captured from this code's animation. Modify it for the problem at hand.
   c. Extend the two-support problem to a box sliding to the right on a beam with a third support under the right edge of the beam.

2. As shown on the left of Figure 1.8, a two kilogram mass is attached to two 5-m strings that pass over frictionless rollers. There is a student holding the end of each string. Initially the strings are vertical, but then move apart as the students move at a constant 4 m/s, one to the right and one to the left.

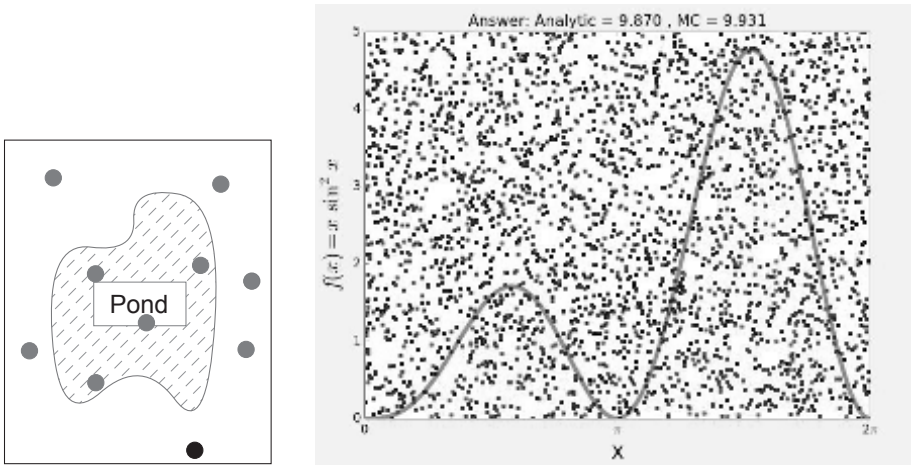


Figure 1.6. *Left:* Throwing stones into a pond as a technique for measuring its area. The ratio of "hits" to total number of stones thrown equals the ratio of the area of the pond to that of the box. *Right:* The evaluation of an integral via a Monte Carlo (stone throwing) technique of the ratio of areas.

.

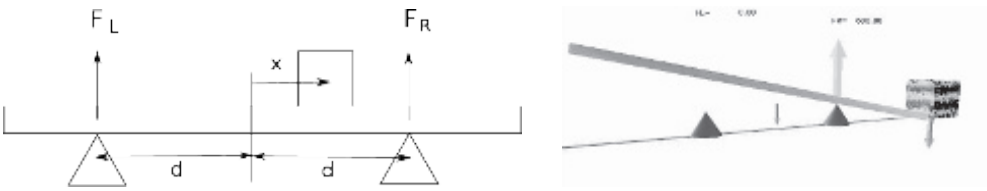


Figure 1.7. *Left:* A beam and a box supported at two points. *Right:* A screen shot from the animation showing the forces on the beam as the weight moves.

(a) What is the initial tension of each cord?

(b) Compute the tension in each cord as the students move apart.

(c) Can the students ever get both strings to be horizontal?

(d) Extend your program so that it creates an animation showing the tensions in the string as the students move apart. In Listing 1.9 we present our code using the Python Visual package, and in Figure 1.8 right we present a screen shot captured from this code's animation. Modify the code as appropriate for your problem.

3. As shown on the right of Figure 1.8, masses $m_1 = m_2 = 1$kg are suspended by two strings of length $L = 2.5m$ and connected by a string of length $s = 1m$.

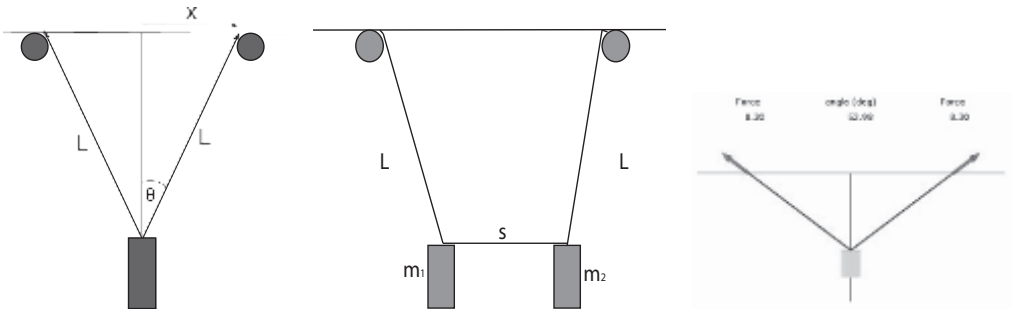


Figure 1.8. *Left:* A mass is suspended from two strings on frictionless rollers, with students pulling horizontally at the end of each string. *Center:* A mass m_1 and a second m_2 are suspended by two strings of length L and connected by a string of length s. *Right:* A screen shot from animation showing the forces on two strings as the students pulling on each mass move apart.

As before, the strings of length L are being pulled horizontally over frictionless pulleys.

a. Write a program that computes the tension of each cord as the students move apart.
b. Extend your program so that it creates an animation showing the tensions in the strings as the students move apart.
c. How would the problem change if $m_1 \neq m_2$?

1.2.2 Python Matrix Tools

Dealing with many numbers at one time is a prime strength of computation, and computer languages have abstract data types for just that purpose. A *list* is Python's built-in data type for a sequence of numbers or objects kept in a definite order. An *array* is a higher-level data type available with the *NumPy* package, and can be manipulated like a vector. Square brackets with comma separators [1, 2, 3] are used for lists, with square brackets also used to indicate the index for a list item:

```
>>> L = [1, 2, 3]                                    # Create list
>>> L[0]                                           # Print element 0
1
>>> L                                             # Print entire list
 [1, 2, 3]
>>> L[0]= 5                                      # Change element 0
>>> len(L)                                          # Length of list
3
```

NumPy arrays convert Python lists into arrays, which can be manipulated like vectors:

```
>>> vector1 = array([1, 2, 3, 4, 5])              # Fill array wi list
>>> print('vector1 =', vector1)
```

```
vector1 =  [1 2 3 4 5]
>>> vector2 = vector1 + vector1                          # Add 2 vectors
>>> print('vector2=', vector2)
vector2= [ 2  4  6  8 10]
>>> matrix1 = array(([0,1],[1,3])                  # An array of arrays
>>> print(matrix1)
[[0 1]
 [1 3]]
>>> print  (matrix1 * matrix1)                         # Matrix multiply
 [[0 1]
  [1 9]]
```

When describing NumPy arrays, the number of "dimensions", `ndim`, means the number of indices, which can be as high as 32. What might be called the "size" or "dimensions" of a matrix is called the *shape* of a NumPy array:

```
>>> import numpy as np
>>> np.arange(12)                                      # List 12 ints
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> np.arange(12).reshape((3,4))          # Reshape to 3x4
array([[ 0,  1,  2,  3],
            [ 4,  5,  6,  7],
            [ 8,  9, 10, 11]])
>>> a = np.arange(12).reshape((3,4))
>>> a.shape
(3L, 4L)
>>> a.ndim                                              # Dimension?
2
>>> a.size                                       # Number of elements?
12
```

If you want to form the familiar matrix product from two arrays, you use the `dot` function, whereas the asterisk `*` is used for an element-by-element product:

```
>>> matrix1= array( [[0,1], [1,3]])
>>> matrix1
array([[0, 1],
       [1, 3]])
>>> print ( dot(matrix1,matrix1) )              # Matrix or dot product
[[ 1  3]
 [ 3 10]]
>>> print (matrix1 * matrix1)               # Element-by-element product
[[0 1]
 [1 9]]
```

Rather than writing your own matrix routines, for the sake of speed and reliability we recommend the use of well established libraries. Although the array objects of NumPy are not the same as mathematical matrices, there is the `LinearAlgebra` package that treats 2-D arrays as mathematical matrices. Consider the standard solution of linear equations

$$A\mathbf{x} = \mathbf{b}, \tag{1.2}$$

where we have used a bold character to represent a vector. For example,

```
>>> from numpy import *
>>> from numpy.linalg import*
>>> A = array( [ [1,2,3],  [22,32,42],  [55,66,100] ] ) # Array of arrays
>>> print ('A =', A)
A = [[  1   2   3]
        [ 22  32  42]
        [ 55  66 100]]
```

We solve (1.2) with NumPy's `solve` command, and then test the solution:

```
>>> b = array([1,2,3])
>>> from numpy.linalg import solve
>>> x = solve(A, b) # Finds solution
>>> print ('x =', x)
x = [ -1.4057971  -0.1884058   0.92753623] # The solution
>>> print ('Residual =',  dot(A, x) - b)               # LHS-RHS
Residual = [4.44089210e-16   0.00000000e+00  -3.55271368e-15]
```

A direct, if not most efficient, way to solve (1.2) is to calculate the inverse A^{-1}, and then multiply through by the inverse, $\mathbf{x} = A^{-1}\mathbf{b}$:

```
>>> from numpy.linalg import inv
>>>  dot(inv(A), A)                                          # Test inverse
array([[  1.00000000e+00,  -1.33226763e-15,  -1.77635684e-15],
           [  8.88178420e-16,   1.00000000e+00,   0.00000000e+00],
           [ -4.44089210e-16,   4.44089210e-16,   1.00000000e+00]])
>>> print ('x =', multiply(inv(A), b))
x = [-1.4057971  -0.1884058   0.92753623]             # Solution
>>> print ('Residual =',  dot(A, x) - b)
Residual = [  4.44089210e-16   0.00000000e+00  -3.55271368e-15]
```

To solve the eigenvalue problem,

$$I\omega = \lambda\omega, \tag{1.3}$$

we call the `eig` method (as in `Eigen.py`):

```
>>> from numpy import*
>>> from numpy.linalg import eig
>>> I = array( [[2./3,-1./4], [-1./4,2./3]] )
>>> print('I =\n', I)
    I =  [[ 0.66666667 -0.25      ]
            [-0.25        0.66666667]]
>>> Es, evectors = eig(A)                    # Solve eigenvalue problem
>>> print('Eigenvalues =', Es, '\n Eigenvector Matrix =\n', evectors)
    Eigenvalues =   [ 0.91666667  0.41666667]
    Eigenvector Matrix =   [[ 0.70710678  0.70710678]
                                      [-0.70710678  0.70710678]]
>>> Vec = array([ evectors[0, 0], evectors[1, 0] ] )
>>> LHS = dot(I, Vec)
>>> RHS = Es[0]*Vec
>>> print('LHS - RHS =', LHS-RHS) # Test for 0
        LHS - RHS = [  1.11022302e-16  -1.11022302e-16]
```

1. Find the numerical inverse of

$$A = \begin{bmatrix} +4 & -2 & +1 \\ +3 & +6 & -4 \\ +2 & +1 & +8 \end{bmatrix}. \tag{1.4}$$

   a. Check your inverse in both directions; that is, check that $AA^{-1} = A^{-1}A = I$.
   b. Note the number of decimal places to which this is true as this gives you some idea of the precision of your calculation.
   c. Determine the number of decimal places of agreement there is between your numerical inverse and the analytic result:

$$A^{-1} = \frac{1}{263}\begin{bmatrix} +52 & +17 & +2 \\ -32 & +30 & +19 \\ -9 & -8 & +30 \end{bmatrix}. \tag{1.5}$$

2. Consider the matrix A again, here being used to describe three simultaneous linear equations, $A\mathbf{x} = \mathbf{b}$. Solve for three different $\mathbf{x}$ vectors appropriate to the three different $\mathbf{b}$'s:

$$\mathbf{b}_1 = \begin{bmatrix} +12 \\ -25 \\ +32 \end{bmatrix}, \quad \mathbf{b}_2 = \begin{bmatrix} +4 \\ -10 \\ +22 \end{bmatrix}, \quad \mathbf{b}_3 = \begin{bmatrix} +20 \\ -30 \\ +40 \end{bmatrix}.$$

3. Consider the matrix $A = \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix}$, where you are free to use any values you want for α and β. Show numerically that the eigenvalues and eigenvectors are the complex conjugates

$$\mathbf{x}_{1,2} = \begin{bmatrix} +1 \\ \mp i \end{bmatrix}, \quad \lambda_{1,2} = \alpha \mp i\beta. \tag{1.6}$$

1.2.3 Python Algebraic Tools

Symbolic manipulation software represents a supplementary, yet powerful, approach to computation in physics [Napolitano(18)]. Python distributions often contain the symbolic manipulation packages *Sage* and *SymPy*, which are quite different from each other. *Sage* is in the same class as Maple and MATHEMATICA and is beyond what we care to cover in this book. In contrast, the *SymPy* package runs very much like any other Python package from within a Python shell. For example, here we use Python's interactive shell to import methods from SymPy and then take some analytic derivatives:

```
>>> from sympy import *
>>> x, y = symbols('x y')
>>> y = diff(tan(x),x);  y
```

```
     tan^2(x) + 1
>>> y = diff(5*x**4 + 7*x**2, x, 1);  y             #  dy/dx  1 optional
     20 x^3 + 14 x
>>> y = diff(5*x**4+7*x**2, x, 2);  y              #  d^2y/dx^2
     2  (30 x^2 + 7)
```

The `symbols` command declares the variables x and y as algebraic, and the `diff` command takes the derivative with respect to second argument (the third argument is order of derivative). Here are some expansions:

```
>>> from sympy import *
>>> x, y = symbols('x y')
>>> z = (x + y)**8; z
     (x + y)^8
>>> expand(z)
     x^8 + 8 x^7 y + 28 x^6 y^2 + 56 x^5 y^3 + 70 x^4 y^4
                        + 56 x^3 y^5  + 28 x^2 y^6 + 8 x y^7 + y^8
```

SymPy knows about infinite series, and about different expansion points:

```
>>> sin(x).series(x, 0)                                  # Sin x series
     x - x^3/6 + x^5/120 + \mathcal{O}(x^6)$
>>> sin(x).series(x,10)                              #  sin x about x=10
 sin(10) + x cos(10) - x^2  sin(10)/2 - x^3  cos(10)/6
         + x^4  sin(10)/24 + x^5  cos(10)/120 +O(x^6)
>>> z = 1/cos(x); z                            # Division, not an inverse
    $1/\cos(x)$
>>> z.series(x, 0)                              # Expand 1/cos x about 0
    1 + x^2/2 + 5 x^4/24 + O(x^6)
```

A classic difficulty with computer algebra systems is that the produced answers may be correct though not in a simple enough form to be useful. SymPy has functions such as `simplify`, tellfactor, `collect, cancel`, and `apart` which often help:

```
>>> factor(x**2 -1)
    (x - 1) (x + 1)                                        # Well done
>>> factor(x**3 - x**2 + x - 1)
    (x - 1) (x^2 + 1)
>>> simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
    x - 1
>>> factor(x**3+3*x**2*y+3*x*y**2+y**3)
    (x + y)^3                                              # Much better!
>>> simplify(1 + tan(x)**2)
    cos(x)^{(-2)}
```

1.3 Dealing with Floating Point Numbers

Scientific computations must account for the limited amount of computer memory used to represent numbers. Standard computations employ integers represented in *fixed-point* notation and other numbers in *floating-point* or scientific notation. In Python, we usually deal with 32 bit integers and 64 bit floating point numbers (called

double precision in other languages). Doubles have approximately 16 decimal places of precision and magnitudes in the range

$$4.9 \times 10^{-324} \;\leq\; \text{double precision} \;\leq\; 1.8 \times 10^{308}. \tag{1.7}$$

If a double becomes larger than 1.8×10^{308}, a fault condition known as an *overflow* occurs. If the double becomes smaller than 4.9×10^{-324}, an underflow occurs. For overflows, the resulting number may end up being a machine-dependent pattern, not a number (NAN), or unpredictable. For underflows, the resulting number is usually set to zero.

Because a 64-bit floating point number stores the equivalent of only 15–16 decimal places, floating-point computations are usually approximate. For example, on a computer

$$3 + 1.0 \times 10^{-16} = 3. \tag{1.8}$$

This loss of precision is measured by defining the *machine precision* ϵ_m as the maximum positive number that can be added to a stored 1.0 without changing that stored 1.0:

$$1.0_c + \epsilon_m \;\overset{\text{def}}{=}\; 1.0_c, \tag{1.9}$$

where the subscript c is a reminder that this is a computer representation of 1. So, except for powers of 2, which are represented exactly, we should assume that all floating-point numbers have an error in the fifteenth place.

1.3.1 Uncertainties in Computed Numbers

Errors and uncertainties are integral parts of computation. Some errors are computer errors arising from the limited precision with which computers store numbers, or because of the approximate nature of algorithm. An algorithmic error may arise from the replacement of infinitesimal intervals by finite ones or of infinite series by finite sums, such as,

$$\sin(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} \simeq \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} + \mathcal{E}(x, N), \tag{1.10}$$

where $\mathcal{E}(x, N)$ is the approximation error. A reasonable algorithm should have $\mathcal{E}$ decreasing as N increases.

A common type of uncertainty in computations that involve many steps is *round-off* errors. These are accumulated imprecisions arising from the finite number of digits in floating-point numbers. For the sake of brevity, imagine a computer that kept just four decimal places. It would then store 1/3 as 0.3333 and 2/3 as 0.6667, where the computer has "rounded off" the last digit in 2/3. Accordingly, even a simple subtraction can be wrong:

$$2\left(\frac{1}{3}\right) - \frac{2}{3} = 0.6666 - 0.6667 = -0.0001 \neq 0. \tag{1.11}$$

So although the result is small, it is not 0. Even with full 64 bit precision, if a calculation gets repeated millions or billions of times, the accumulated error answer may become large.

Actual calculations are often a balance. If we include more steps then the approximation error generally follows a power-law decrease. Nevertheless the relative round-off error after N steps tends to accumulate randomly, approximately like $\sqrt{N}\epsilon_m$. Because the total error is the sum of both these errors, eventually the ever-increasing round-off error will dominate. As rule of thumb, as you increase the number of steps in a calculation you should watch for the answer to converge or stabilize, decimal place by decimal place. Once you see what looks like random noise occurring in the last digits, you know round-off error is beginning to dominate, and you should probably step back a few steps and quit. An example is given in Figure 1.9.

1. Write a program that determines your computer's underflow and overflow limits (within a factor of 2). Here's a sample pseudocode

```
under = 1.
over = 1.
begin do N times
     under = under/2.
     over = over * 2.
     write out: loop number, under, over
end do
```

   a. Increase N if your initial choice does not lead to underflow and overflow.
   b. Check where under- and overflow occur for floating-point numbers.
   c. Check what are the largest and the most negative integers. You accomplish this by continually adding and subtracting 1.

2. Write a program to determine the machine precision ϵ_m of your computer system within a factor of 2. A sample pseudocode is

```
eps = 1.
begin do N times
  eps = eps/2.
  one = 1. + eps
 end do
```

   a. Determine experimentally the machine precision of floats.
   b. Determine experimentally the machine precision of complex numbers.

1.4 Numerical Derivatives

Although the mathematical definition of the derivative is simple,

$$\frac{dy(t)}{dt} \overset{\text{def}}{=} \lim_{h\to 0} \frac{y(t+h)-y(t)}{h}, \tag{1.12}$$

it is not a good algorithm. As h gets smaller, the numerator to fluctuate between 0 and machine precision ϵ_m, and the denominator approaches zero. Instead, we use the Taylor series expansion of a $f(x+h)$ about x with h kept small but finite. In the *forward-difference* algorithm we take

$$\left.\frac{dy(t)}{dt}\right|_{FD} \simeq \frac{y(t+h)-y(t)}{h} + \mathcal{O}(h). \tag{1.13}$$

This $\mathcal{O}(h)$ error can be cancelled off by evaluating the function at a half step less than and a half step greater than t. This yields the *central-difference derivative*:

$$\left.\frac{dy(t)}{dt}\right|_{\text{CD}} \frac{y(t+h/2)-y(t-h/2)}{h} + \mathcal{O}(h^2). \tag{1.14}$$

The central-difference algorithm for the second derivative is obtained by using the central-difference algorithm on the corresponding expression for the first derivative:

$$\left.\frac{d^2y(t)}{dt^2}\right|_{CD} \simeq \frac{y'(t+h/2)-y'(t-h/2)}{h} \simeq \frac{y(t+h)+y(t-h)-2y(t)}{h^2}. \tag{1.15}$$

1. Use forward- and central-difference algorithms to differentiate the functions $\cos t$ and e^t at $t = 0.1, 1.$, and 100.
   a. Print out the derivative and its relative error $\mathcal{E}$ as functions of h. Reduce the step size h until it equals machine precision $h \simeq \epsilon_m$.
   b. Plot $\log_{10}|\mathcal{E}|$ versus $\log_{10} h$ and check whether the number of decimal places obtained agrees with the estimates in the text.

2. Calculate the second derivative of $\cos t$ using the central-difference algorithms.
   a. Test it over four cycles, starting with $h \simeq \pi/10$ and keep reducing h until you reach machine precision

1.5 Numerical Integration

Mathematically, the Riemann definition of an integral is the limit

$$\int_a^b f(x)\,dx = \lim_{h\to 0} \sum_{i=1}^{(b-a)/h} f(x_i)h. \tag{1.16}$$

Numerical integration is similar, but approximates the integral as the a finite sum over rectangles of height $f(x)$ and widths (or weights) w_i:

$$\int_a^b f(x)\,dx \simeq \sum_{i=1}^{N} f(x_i)w_i. \tag{1.17}$$

Equation (1.17) is the standard form for all integration algorithms: the function $f(x)$ is evaluated at N points in the interval $[a,b]$, and the function values $f_i \equiv f(x_i)$ are summed with each term in the sum weighted by w_i. The different integration algorithms amount to different ways of choosing the points x_i and weights w_i. If you are free to pick the integration points, then our suggested algorithm is Gaussian quadrature. If the points are evenly spaced, then Simpson's rule makes good sense.

The trapezoid and Simpson integration rules both employ $N-1$ boxes of width h evenly-spaced throughout the integration region $[a,b]$:

$$x_i = a + ih, \qquad h = \frac{b-a}{N-1}, \qquad i = 0, N-1. \tag{1.18}$$

For each interval, the trapezoid rule assumes a trapezoid of width h and height $(f_i + f_{i+1})/2$, and, accordingly, approximates the area of each trapezoid as $\frac{1}{2}hf_i + \frac{1}{2}hf_{i+1}$. To apply the trapezoid rule to the entire region $[a,b]$, we add the contributions from all subintervals:

$$\int_a^b f(x)\,dx \simeq \frac{h}{2}f_1 + hf_2 + hf_3 + \cdots + hf_{N-1} + \frac{h}{2}f_N, \tag{1.19}$$

where the endpoints get counted just once, but the interior points twice. In terms of our standard integration rule (1.17), we have

$$w_i = \left\{\frac{h}{2}, h, \ldots, h, \frac{h}{2}\right\} \qquad \text{(Trapezoid Rule)}. \tag{1.20}$$

In `TrapMethods.py` in Listing 1.15 we provide a simple implementation.

Simpson's rule is also for evenly spaced points of width h, though with the heights given by parabolas fit to successive sets of three adjacent integrand values. This leads to the approximation:

$$\int_a^b f(x)dx \simeq \frac{h}{3}f_1 + \frac{4h}{3}f_2 + \frac{2h}{3}f_3 + \frac{4h}{3}f_4 + \cdots + \frac{4h}{3}f_{N-1} + \frac{h}{3}f_N. \tag{1.21}$$

In terms of our standard integration rule (1.17), this is

$$w_i = \left\{\frac{h}{3}, \frac{4h}{3}, \frac{2h}{3}, \frac{4h}{3}, \ldots, \frac{4h}{3}, \frac{h}{3}\right\} \qquad \text{(Simpson's Rule)}. \tag{1.22}$$

Because the fitting is done with sets of three points, *the number of points N must be odd for Simpson's rule.*

In general, you should choose an integration rule that gives an accurate answer using the least number of integration points. For the trapezoid and Simpson rules the errors vary as

$$\mathcal{E}_t = O\left(\frac{[b-a]^3}{N^2}\right)\frac{d^2f}{dx^2}, \qquad \mathcal{E}_s = O\left(\frac{[b-a]^5}{N^4}\right)\frac{d^4f}{dx^4}, \tag{1.23}$$

where the derivatives are evaluated someplace within the integration region. So unless the integrand has behavior problems with its derivatives, Simpson's rule should converge more rapidly than the trapezoid rule and with less error. While it seems like one might need only to keep increasing the number of integration points to obtain better accuracy, relative round-off error tends to accumulate, and, after N integration points, grows like

$$\epsilon_{ro} \simeq \sqrt{N}\epsilon_m, \tag{1.24}$$

where $\epsilon_m \simeq 10^{15}$ is the machine precision (discussed in §1.3). So even though the error in the algorithm can be made arbitrary small, the total error, that is, the error due to algorithm plus the error due to round-off, eventually will increase like $\sqrt{N}$.

1.5.1 Gaussian Quadrature

Gauss figured out a way of picking the N points and weights in (1.17) so as to make an integration over [-1,1] exact if $g(x)$ is a polynomial of degree $2N-1$ or less. To accomplish this miraculous feat, the x_i's must be the N zeros of the Legendre polynomial of degree N, and the weights related to the derivatives of the polynomials [LPB(15)]:

$$P_N(x_i) = 0, \qquad w_i = \frac{2}{(1-x_i^2)[P'_N(x_i)]^2}. \tag{1.25}$$

Not to worry, we supply a program that determines the points and weights. If your integration range is [a,b] and not [-1,+1], they will be scaled as

$$x'_i = \frac{b+a}{2} + \frac{b-a}{2}x_i, \qquad w'_i = \frac{b-a}{2}w_i. \tag{1.26}$$

In general, Gaussian quadrature will produce higher accuracy than the trapezoid and Simpson rules for the same number of points, and is our recommended integration method.

Our Gaussian quadrature code `IntegGaussCall.py` in Listing 1.16 requires the value for precision `eps` of the points and weights to be provided by the user. Overall precision is usually increased by increasing the number of points used. The points and weights are generated by the method `GaussPoints.py`, which will be included automatically in your program via the `from GaussPoints import GaussPoints` statement.

1.5.2 Monte Carlo (Mean Value) Integration

Monte Carlo integration is usually simple, but not particularly efficient. It is just a direct application of the *mean value theorem*:

$$I = \int_a^b dx\, f(x) = (b-a)\,\langle f \rangle\,. \tag{1.27}$$

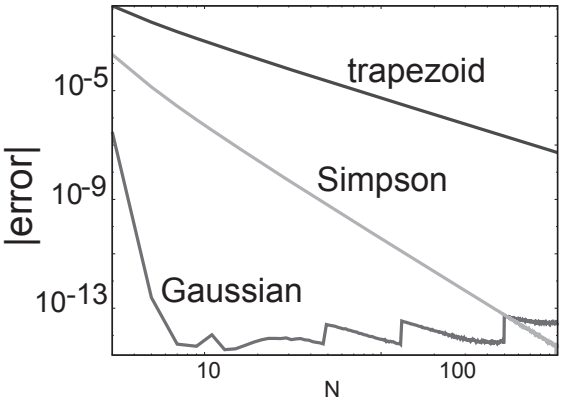


Figure 1.9. A log-log plots of the relative error in an integration using the trapezoid rule, Simpson's rule, and Gaussian quadrature *versus* the number of integration points N.

The mean is determined by *sampling* the function $f(x)$ at random points within the integration interval:

$$\langle f \rangle \simeq \frac{1}{N}\sum_{i=1}^{N} f(x_i) \quad \Rightarrow \quad \int_a^b dx\, f(x) \simeq (b-a)\frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{1.28}$$

The uncertainty in the value obtained for the integral I after N samples of $f(x)$ is measured by the standard deviation σ_I. If σ_f is the standard deviation of the integrand f in the sampling, then for a normal distribution of random number we would have

$$\sigma_I \simeq \frac{1}{\sqrt{N}}\sigma_f. \tag{1.29}$$

So, for large N the error decreases as $1/\sqrt{N}$. In Figure 1.6 left we show a scatter plot of the points used in a Monte Carlo integration by the code `PondMapPlot.py` in Listing 1.7.

Before you actually use random numbers to evaluate integrals, we recommend that you work through §1.6.2 to be sure your random number generator is working properly.

On the left of Figure 1.6 we show a pond whose area we wish to determine. We can determine the area of such an irregular figure by throwing stones in the air (generating random (x, y) values), and counting the number of splashes $N_{\rm pond}$ as well as the number of stones lying on the ground $N_{\rm box}$. The area of the pond is then given by the simple ratio:

$$A_{\rm pond} = \frac{N_{\rm pond}}{N_{\rm pond} + N_{\rm box}} A_{\rm box}. \tag{1.30}$$

1. Write a program to integrate a function for which you know the analytic answer so that you can determine the error. Use

a. the trapezoid rule,
b. the Simpson rule,
c. Gaussian quadrature, and
d. Monte Carlo integration.

2. Compute the relative error $\epsilon = |(\text{numerical-exact})/\text{exact}|$ for each case, and make a log-log plot of relative error versus N as we do in Figure 1.9. You should observe a steep initial power-law drop-off of the error, which is characteristic of an algorithmic error decreasing with increasing N. Note that the ordinate in the plot is the negative of the number of decimal places of precision in your integral.

3. The algorithms should stop converging when round-off error starts to dominate, as noted by random fluctuations in the error. Estimate the number of decimal places of precision obtained for each of the three rules.

4. Use sampling to compute π (number hits in unit circle/total number hits = π/Area of box).

5. Evaluate the following integrals:

a. $\int_0^1 e^{\sqrt{x^3+5\,x}}\,dx$

b. $\int \frac{1}{x^2+2\,x+4}\,dx$

c. $\int_0^\infty e^{(-x^2)}\,dx$

1.6 Random Number Generation

Randomness or chance occurs in different areas of physics. For example, quantum mechanics and statistical mechanics are statistical by nature, and so randomness enters as one of the key assumptions of statistics. Or, looked at the other way, random processes such as the motion of molecules were observed early on, and this led to theory of statistical mechanics. In addition to the randomness in nature, many computer calculations employ *Monte Carlo* methods that include elements of chance to either simulate random physical processes, such as thermal motion or radioactive decay, or in mathematical evaluations, such as integration.

Randomness describes a lack of predicability or regular pattern. Mathematically, we define a sequence $r_1, r_2, \ldots$ as *random* if there are no short- or long-range correlations among the numbers. This does not necessarily mean that all the numbers in the sequence are equally likely to occur; that is called *uniformity*. As a case in point, 0, 2, 4, 6, ... is uniform though probably not random. If $P(r)\,dr$ is the probability of finding r in the interval $[r, r + dr]$, a uniform distribution has $P(r) =$ a constant.

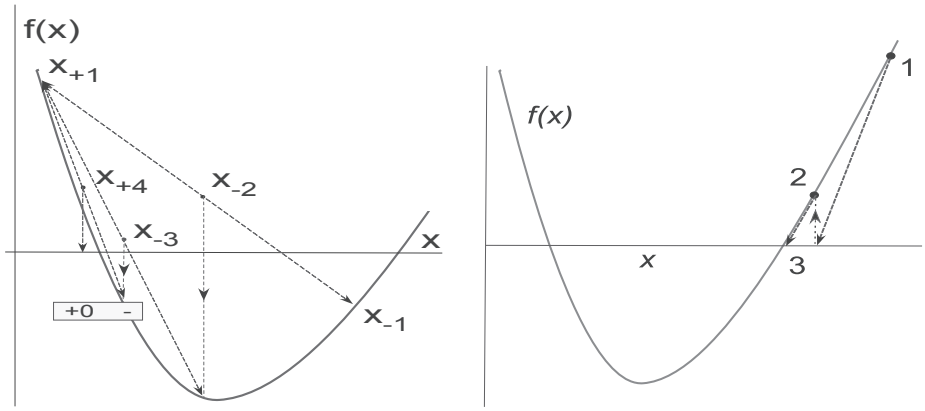


Figure 1.10. *Left:* A graphical representation of three steps involved in solving for a zero of $f(x)$ using the bisection algorithm. *Right:* Two steps shown for the Newton-Raphson method of root finding in which the function is approximated as a straight line, and then the intercept of that line is determined.

Computers, being deterministic, cannot generate truly random numbers. However, they can generate *pseudorandom numbers*, and the built-in generators are often very good at this. The `random` module in Python produces a sequence of random numbers, and can be used after an `import random` statement. The module permits many options, with the simple command `random.random()` returning the next random floating point number, distributed uniformly in the range [0.0, 1.0). But if you look hard enough, you are sure to find correlations among the numbers.

The *linear congruential* or *power residue* method is the common way of generating a pseudorandom sequence of numbers:

$$r_{i+1} \overset{\text{def}}{=} (a\, r_i + c) \bmod M \;= \text{remainder}\left(\frac{a\, r_i + c}{M}\right). \tag{1.31}$$

Wikipedia has a table of common choices, for instance, $m = 2^{48}$, $a = 25214903917$, $c = 11$. Here *mod* is a function (% sign in Python) for modulus or *remaindering*, which is essentially a bit-shift operation that results in the least significant part of the input number and hence counts on the randomness of round-off errors.

Your computer probably has random-number generators that should be better than one computed by a simple application of the power residue method. In Python we use `random.random()`, the Mersenne Twister generator. To initialize a random sequence, you need to plant a seed (r_0), or in Python say `random.seed(None)`, which seeds the generator with the system time, which would differ for repeated executions. If random numbers in the range $[A, B]$ are needed, you only need to scale, for example,

$$x_i = A + (B - A) r_i, \quad 0 \le r_i \le 1, \;\Rightarrow\; A \le x_i \le B. \tag{1.32}$$

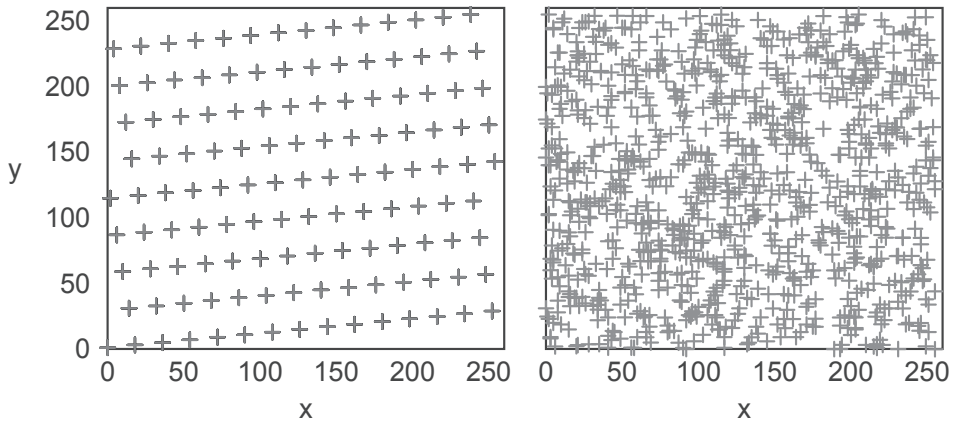


Figure 1.11. *Left:* A plot of successive random numbers $(x, y) = (r_i, r_{i+1})$ generated with a deliberately "bad" generator. *Right:* A plot generated with the built-in random number generator. While the plot on the right is not proof that the distribution is random, the plot on the left is proof enough that the distribution is not random.

1.6.1 Tests of Random Generators

A good general rule, before starting a full calculation, is to check your random number generator by itself. Here are some ways:

- Look at a print out of the numbers and check that they fall within the desired range and that they look different from each other.
- A simple plot of r_i versus i (Figure 1.12) may not prove randomness, though it may disprove it as well as showing the range of numbers.
- Make an x-y plot of $(x_i, y_i) = (r_{2i}, r_{2i+1})$. If your points have noticeable regularity (Figure 1.11 left), the sequence is not random. Random points (Figure 1.11 right) should uniformly fill a square with no discernible pattern (a cloud).
- A simple test of uniformity, though not randomness, evaluates the kth moment of a distribution

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k \simeq \int_0^1 dx\, x^k P(x) \simeq \frac{1}{k+1} + O\left(\frac{1}{\sqrt{N}}\right), \tag{1.33}$$

  where the approximate value is good for a continuous uniform distribution. If the deviation from (1.33) varies as $1/\sqrt{N}$, then you *also* know that the distribution is random since this assumes randomness.