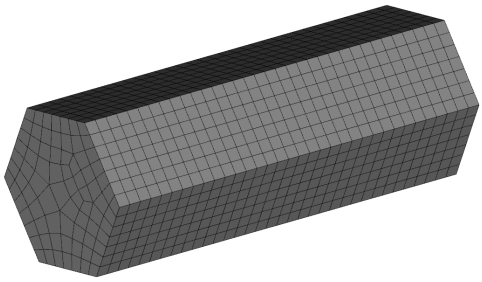
Chemical and Biological Engineering Calculations using Python

Jeffrey J. Heys

Formatting originally based on The Legrand Orange Book LaTeX template by Mathias Legrand.
First printing, June 2014



Contents

1 Problem Solving in Engineering 7

1.1 Textbook and Course Format 9

1.2 Equation Identification and Categorization 9

1.2.1 Algebraic vs. Differential Equations 9

1.2.2 Linear vs. Nonlinear Equations 9

1.2.3 Ordinary vs. Partial Differential Equations 10

1.2.4 Interpolation vs. Regression 12

1.3 Problems 12

1.4 Additional Resources 12

2 Programming with Python 15

2.1 Why Python? 15

2.1.1 Compiled vs. Interpreted Computer Languages 16

2.1.2 A Note on Python Versions 17

2.2 Getting Python 17

2.2.1 Installation of Python 18

2.3 Python Variables and Operators 19

2.4 External Libraries 21

2.5 Loops and Conditionals in Python 22

2.6 Functions 25

2.7 Numpy Basics 26

2.7.1 Array and Vector Creation 26

2.7.2 Array Operations 28
2.8 Matplotlib Basics 29
2.9 Additional Resources 31

3 Symbolic Mathematics 33
3.1 Introduction 33
3.2 Symbolic Mathematics Packages 34
3.3 An Introduction to SymPy 34
3.4 Factoring and Expanding Functions 38
3.5 Derivatives and Integrals 40

4 Linear Systems 43
4.1 Example Problem 44
4.2 A Direct Solution Method 46
4.2.1 Computational Cost 50
4.3 Iterative Solution Methods 51
4.3.1 Vector Norms 52
4.3.2 Jacobi Iteration 52
4.3.3 Gauss-Seidel Iteration 54
4.3.4 Convergence of Iterative Methods 56

5 Regression 59
5.1 Motivation 59
5.2 Fitting Vapor Pressure Data 60
5.3 Linear Regression 61
5.3.1 Alternative derivation of the normal equations 64
5.4 Nonlinear Regression 64

6 Nonlinear Equations 69
6.1 Introduction 69
6.2 Bisection Method 71
6.3 Newton's Method 74
6.4 Broyden's Method 75
6.5 Multiple Nonlinear Equations 77

7 Statistics 83
7.1 Introduction 83

7.2 Reading Data from a File **83**
7.2.1 Parsing an Array 86
7.3 Statistical Analysis **86**
7.4 Advanced Linear Regression **89**

8 Numerical Differentiation and Integration **93**
8.1 Introduction **93**
8.2 Numerical Differentiation **93**
8.2.1 First Derivative Approximation 94
8.2.2 Second Derivative Approximation 96
8.3 Numerical Integration **97**
8.3.1 Numerical Integration Using Scipy 100

9 Initial Value Problems **103**
9.1 Introduction **103**
9.2 Biochemical Reactors **103**
9.3 Forward Euler **104**
9.4 Modified Euler Method **107**
9.5 Systems of Equations **108**
9.5.1 Second-Order Initial Value Problems 110
9.6 Stiff Differential Equations **110**

10 Boundary Value Problems **115**
10.1 Introduction **115**
10.2 Shooting Method **116**
10.3 Finite Difference Method **120**

11 Partial Differential Equations **125**
11.1 Finite Difference Method for Steady-State PDEs **125**
11.1.1 Setup 126
11.1.2 Matrix Assembly 127
11.1.3 Solving and Plotting 129
11.2 Including Convection **130**
11.3 Finite Difference Method for transient PDEs **133**

12 Finite Element Method **139**
12.1 A Warning **139**

12.2 Why FEM? 139

12.3 Laplace's Equation 140

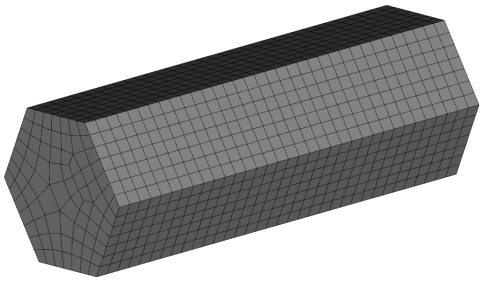
12.3.1 The Mesh 140
12.3.2 Discretization 140
12.3.3 Wait! Why are we doing this? 141
12.3.4 FEniCS implementation 141

12.4 Further Reading 142

Bibliography 145

Index 147

Textbook and Course Format
Equation Identification and Categorization
Algebraic vs. Differential Equations
Linear vs. Nonlinear Equations
Ordinary vs. Partial Differential Equations
Interpolation vs. Regression
Problems
Additional Resources



1. Problem Solving in Engineering

For most engineering problems, students find that the following sequence of steps is an effective problem solving approach to the vast majority of the problems they encounter:

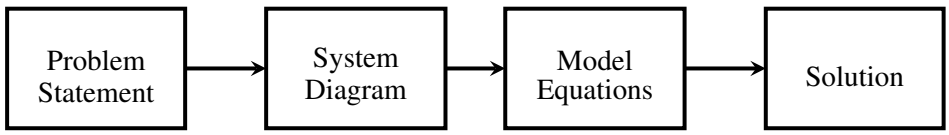


Figure 1.1: Engineering problem solving process.

In most courses, students practice all the steps outlined in figure 1.1, but the focus is usually on the construction of the system diagram and developing the mathematical equations for every unique type of process that is the focus of a particular course. Only limited attention is usually given to solving the mathematical equations that arise in a particular course because the assumption is that the student should have learned how to do that in their mathematics courses or some other course. Many chemical or biological engineering curricula have a course that is focused on the use of computers to solve the many different types of equations that arise in a student's engineering courses. The focus of this textbook is just that: using computers to solve the equation(s) that students typically encounter throughout the chemical and biological engineering curriculum.

The timing of a course on computational or numerical methods for solving engineering problems varies considerably from one curriculum to the next. One approach is to schedule the course near the end of the curriculum. As a upper level course, students are able to review most of the engineering principles and mathematics that they learned previously and develop a new set of tools (specifically, computational tools) for solving those same problems. Two disadvantages are associated with this approach. First, students do not have the computational tools when they first learn a new engineering principle, which limits the scope of problems they can solving to problems that can be largely solved without a computer (i.e., problems that can be solved with paper and pencil.) The second disadvantage is that the third- and fourth-years of many chemical or biological engineering curricula are already filled with other required courses and it is difficult to find time for yet another course.

A second approach is to schedule the computational methods course early in the curriculum, before students have taken most of the engineering courses in which they learn to derive, construct, and identify the mathematical equations they need to solve and that sometime require a computational approach. There are also two problems with this approach. First, the students have typically not taken all the required mathematics courses, and, as a result, it is difficult to teach a computational approach to solving a differential equation when a student has not yet learned what a differential equation is or techniques for solving it. The second disadvantage is that the student has not taken courses on separations, kinetics, transport, etc. in which they learn to derive or identify the appropriate mathematical equation(s) for their particular problem. It is, of course, difficult to teach a computational approach to solving an equation when the importance or relevance of that equation is not known.

A third approach for addressing this dilemma is to simply not teach a stand-alone computational methods course and instead cover the relevant computational approaches as they are needed in each individual course. We will continue our listing of the 'top two challenges' and identify two potential difficulties with this approach. First, instead of learning and becoming comfortable with 2 or 3 computational tools (i.e., mathematical software packages), students under this format often need to learn 4 or 5 computational tools because everyone of their instructors prefers a different tool, and the students never really become proficient with any single tool. The second difficult is that there are a few important concepts that play a role in many of the various computational methods, e.g., rounding error, logical operators, accuracy, etc., that may never be taught if there is not a single course focused on computational methods.

This textbook, and the course that it was originally written to support, are focused on the second approach – a course that appears in the first year or early in the second year of a chemical or biological engineering curriculum. The main reason for adopting this approach is simply the belief it is critical for students to understand both the potential power and flexibility of computational methods and also the important limitations of these methods before using them to solve problems in chemical and biological engineering. For a student to use a computational tool in a course and blindly trust that tool because they don't understand the algorithms behind the tool is probably more destructive than never learning the tool at all. Further, to limit a student to only problems that can be solved with paper and pencil for most of their undergraduate education is similarly unacceptable. Addressing the limitations associated with teaching computational methods before most of the fundamental engineering and some mathematics courses is difficult. The basic strategy employed by this book is to teach students to recognize the type of mathematical equation they need to solve, and, once they know the type of equation, they can take advantage of the appropriate computational approach that was covered in the course associated with this book (or, more likely, refer back to this book for the appropriate algorithm for their particular equation).

This textbook advocates that students develop the following skills: (1) recognize the type of mathematical equation that needs to be solved – algebraic or differential? linear or nonlinear? interpolation or regression? ordinary or partial differential equation?, and (2) select and implement the appropriate algorithm. If students are able to develop these two skills, they will be equipped with a set of tools that will serve them well in their later engineering courses. These tools can be used by a student to check their work, even when they are primarily using paper and pencil to solve a problem. It is not optimal that students learn how to approximately solve mathematical equations before they know why the equation is relevant, but ever effort is made in this book to at least try and explain the relevance of equations when possible.

1.1 Textbook and Course Format

The course that motivated the creation of this textbook is one semester of approximately 15 weeks. It is the author's belief that most of this material can be covered in that length of time. Each chapter in the textbook covers a different topic and was constructed so that the material in that chapter could be covered in approximately one week. There are, of course, some exceptions. For example, chapter 2 is one of the longest and may require two or even three weeks to cover. Conversely, chapter 3 is one of the shortest and can probably be covered in less than a week or omitted entirely. The large number of topics and short amount of time associated with a single semester, may encourage instructors using this book to consider a slightly different format than the traditional lecture format. For example, if two class times per week are available, an instructor may want to consider requiring students to read the book or watch an online lecture that presents the material to be covered before coming to the first class meeting time each week. The two class periods could then be used to cover example problems (the first class each week) and a 'working class' could be used for the second class meeting of the week. When students are trying to complete the homework, they often need support to overcome a difficult error message or unexpected and unphysical numerical answer from the computer, and allowing students to work on problems for one class time per week is often very beneficial.

1.2 Equation Identification and Categorization

We identified two categories of skills that we wish to develop: (1) recognizing the type of equation(s) and (2) selecting and implementing an appropriate computational method. The first skill will be covered in this first chapter and then the remainder of the book is for developing the second set of skills.

1.2.1 Algebraic vs. Differential Equations

The distinction between algebraic and differential equations is trivial – a differential equation is a relationship between the derivatives of a variable and some function. Differential equations described the rate of change of a variable; typically the rate of change with respect to space or time. Equations can have both independent and dependent variables. It is usually simplest to identify the dependent variables because their value depends upon the value of another variable. For example, in both $v(t) = 2\pi + t^2$ and $\frac{dv}{dt} = 3 + v \cdot t$, v is the dependent variable because its value depends on the value of t and t is the independent variable. There can be multiple independent variables, e.g., multiple spatial dimensions and time. In summary, differential equations have at least one derivative and algebraic equations do not. The presence of a derivative has a significant impact on the computational method used for solving the problem of interest.

1.2.2 Linear vs. Nonlinear Equations

A linear function, $f(x)$ is one that satisfies both of the following properties:
additivity: $f(x+y) = f(x) + f(y)$
homogeneity: $f(c \cdot x) = cf(x)$.

In practice, this means that the dependent variables cannot appear in polynomials of degree two or higher (i.e., $f(x) = x^2$ is nonlinear), in nonlinear arguments within the function (i.e., $f(x) = x + sin(x)$ is nonlinear), or as products of each other (i.e., $f(x,y) = x + xy$ is nonlinear).

For algebraic equations, it is typically straightforward to solve linear systems of equations, even very large systems consisting of millions of equations and millions of unknowns. Two different methods for solving linear systems of equations will be covered in chapter 4. Nonlinear algebraic equations can sometimes be solved exactly using techniques learned in algebra or using symbolic mathematics algorithms, especially when there is only a single equations. However, if we have more than one nonlinear equation or even a single, particularly complex nonlinear algebraic equation (or if we are simply feeling a little lazy) we may need to take advantage of a computational technique to try and find an approximate solution. Algorithms for solving nonlinear algebraic equations are described in chapter 6.

It is important to note that the distinction between linear and nonlinear equations can also be extended to differential equations and all of the same principles apply. For example, $\frac{dc}{dt} = 4c$ and $\frac{d^2c}{dt^2} = 2\sin(\pi t)$ are linear while $\frac{dc}{dt} = c^2$ is nonlinear. In some cases, the nonlinearity will not significantly increase the computational challenge, but, in other cases like the Navier-Stokes equations, the nonlinearity can significantly increase the difficulty in obtaining even an approximate solution.

1.2.3 Ordinary vs. Partial Differential Equations

An ordinary differential equation (ODE) has a single independent variable. For example, if a differential equation only has derivatives with respect to time, t, or a single spatial dimension, x, it is an ordinary differential equation. A differential equation with two or more independent variables is a partial differential equation (PDE). The following are examples of ordinary differential equations.

■ **Example 1.1**

$$t \cdot \frac{dc}{dt} + \frac{d^2c}{dt^2} = \sin(t) \quad \text{(linear, second-order ODE)}$$

If you have not taken a differential equations course, this equation may look a little intimidating or confusing. To solve this equation, we need to find a function $c(t)$ where the first derivative of the function, multiplied by t, plus the second derivative of the function is equal to $\sin(t)$. If that sounds difficult, don't worry, by the end of this textbook you will know how get an approximate solution, i.e., a numerical approximation of the function $c(t)$. It is also important to emphasize that multiplying the dependent variable c by the dependent variable t did not make the equation nonlinear. A nonlinearity only arises if, for example, c is multiplied by itself. ■

Exercise 1.1 Even though you may not have taken a differential equations course, you might be able to solve a simplified version of the previous example. Try to solve:

$$\frac{d^2p}{dt^2} = \sin(t)$$

Notice that we have eliminated the difficult term with t multiplied by the first derivative. Let us start by integrating both sides of the equation with respect to t. Recalling that an integral is just an anti-derivative, so we get:

$$\frac{dp}{dt} + c_1 = -\cos(t) + c_2 \ .$$

The two constants of integration can simply be combined into a single constant, c_0, which can be placed on the right-hand side giving:

$$\frac{dp}{dt} = -\cos(t) + c_0 \ .$$

Now, let us integrate both sides once more with respect to t:

$$p(t) + c_3 = -\sin(t) + c_0 t + c_4$$

which we can simplify once again by combining the two new constants of integration to a single constant c, to give:

$$p(t) = -\sin(t) + c_0 t + c \ .$$

In order to fully determine our unknown function $p(t)$, we need two additional conditions to solve for the value of our two remaining unknown constants, c_0 and c. Typically, this additional information would be initial conditions, i.e., the value of $p(0)$, i.e., the value of p when $t = 0$, and the value of $\frac{dp}{dt}$ at $t = 0$.

It is always a good idea to check the solution to your problem by substituting $p(t)$ back into the original differential equation and checking to make sure that you get the desired right-hand side. ■

■ **Example 1.2**

$$\frac{dx}{dt} = x^2 + 3\cos(t) \ \text{ (nonlinear, first-order ODE)} \ .$$

Again, if you have not had a differential equations course, solving this equation requires finding a function $x(t)$ that has a derivative equal to $(x(t))^2$ plus $3\cos(t)$. Do not worry if that makes your head spin, we will also cover the solution of this class of problems. ■

Some examples of partial differential equations are included below.

■ **Example 1.3**

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \ \text{ (linear, second-order PDE),}$$

This is an equation that describes unsteady, conductive heat transport in one spatial dimension. You could use this equation to describe, for example, the warming of the ground when the sun comes up in the morning, among many other examples. Solving this equation requires finding a function $T(x,t)$ of both time t and space x where the first derivative with respect to time is equal to α times the second derivative with respect to space. ■

■ **Example 1.4**

$$m\frac{\partial m}{\partial x}+\frac{\partial m}{\partial y}=0 \text{(nonlinear, first-order PDE).}$$

By now it is probably obvious that the standard mathematical convention is to use ∂ for derivatives in a PDE while ODEs use d. The order of the equation is determined by the order of the highest derivative. ■

1.2.4 Interpolation vs. Regression

Within engineering, it is often necessary to obtain an equation, usually a polynomial equation, that 'fits' a given set of data. If we want an equation that exactly matches the data, then we must interpolate the data so that we obtain a function (i.e., a polynomial) that has the same value as the data for a given value of the independent variable. In order to determine an interpolant, the number of adjustable parameters that we determine in the equation must equal the number of data points. For example, if we want to interpolate 3 data points, we must use an equation that has three adjustable parameters, such as a quadratic polynomial, ax^2+bx+c.

In practice, it is actually pretty rare that we want to exactly interpolate a given set of data because we hopefully have a large amount of data (and we don't want to use a very high-order polynomial) and that data contains some amount of error. In most cases, we want to approximately fit our data with an equation of some form. In order to do this, we must first decide how we want to measure the 'goodness' of a fit. Maybe we want to fit an equation so that the sum of the distances from the best fit equation to each and every point is minimized. Another option (the option that is almost always selected) is to minimize the sum of the *square* of the distance between every data point and the 'best' fit approximation. This is the so called least-squares regression approach. The function that gives us the best fit based on our chosen criteria is called the regression function and the process of determining the regression function is called regression analysis. The most popular type of regression, linear regression (Figure 1.2) using least-squares, and nonlinear polynomial regression are both covered in chapter 5.

1.3 Problems

Problem 1.1 Determine the type of differential equation:

(a) $\frac{d^2x}{dt^2}=-g$ (Newton's first law)

(b) $\frac{\partial C_A}{\partial t}+v\cdot\frac{\partial C_A}{\partial z}+kC_A=\frac{\partial}{\partial z}\left(\mathscr{D}\frac{\partial C_A}{\partial z}\right)$

Problem 1.2 If you want to determine the polynomial that interpolates 6 data points, what order polynomial is required?

Problem 1.3 You are asked to use regression to determine the best linear polynomial fit for a given set of data. A colleague encourages you to determine the best fit by minimizing the sum of the distance between each point and the line instead of minimizing the sum of the square of the distance, which is the standard practice. The colleague claims this will reduce the influence of a few outlying data points. Is the colleague correct?

1.4 Additional Resources

An understanding of how to solve differential equation problems is not required for understanding the material in this book. However, an ability to classify or recognize the type of equation that

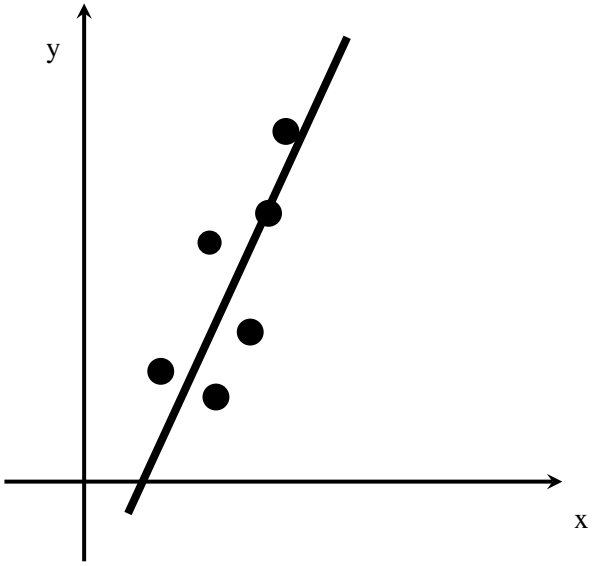
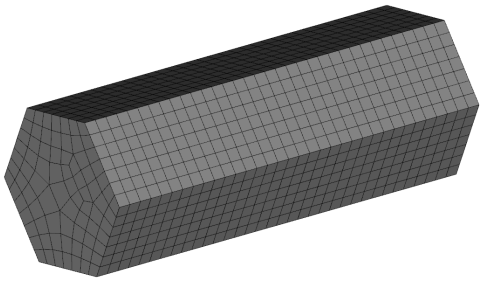


Figure 1.2: An example of linear regression for a given set of data.

one is trying to solve is required. Most differential equation textbooks include a comprehensive set of definitions that enable the classification of mathematical equations. Two popular differential equation textbooks for engineers are:

- Differential Equations for Engineers and Scientists by Çengel and Palm [ÇP13]
- Advanced Engineering Mathematics by Zill and Cullen [ZC06]

Why Python?
Compiled vs. Interpreted Computer Languages
A Note on Python Versions
Getting Python
Installation of Python
Python Variables and Operators
External Libraries
Loops and Conditionals in Python
Functions
Numpy Basics
Array and Vector Creation
Array Operations
Matplotlib Basics
Additional Resources



2. Programming with Python

The objective of this chapter is to motivate the use of the Python programming language for solving problems in chemical and biological engineering and then to present a few basic principles associated with programming in Python. It is important to emphasize that the goal is not to cover all aspects of programming in Python because that would require an entire book by itself. Instead, the goal is to present a few important principles and then slowly add additional Python programming knowledge throughout the remainder of the book.

2.1 Why Python?

When it comes to solving the many different mathematical problems that arise in chemical and biological engineering, many different software options exist for obtaining an exact or approximate solution. Some options, such as HYSYS and Aspen, are very user-friendly and they hide most of the details of the calculations from the user. While these software packages represent an important resource for engineers, our goal here is, in fact, to learn and understand the calculations that are happening in the background of these commercial packages. We will not discuss these high-level software packages here simply because we want to focus on and understand the actual computational details.

The next set of software options for solving engineering problems are mathematical software packages such as MATLAB, Mathematica, or MathCAD. These packages give the user more control over the calculations, but they also require more specialized knowledge than the process simulation software described previously. These mathematical software packages are probably the most popular options for a college-level course on chemical or biological engineering calculations. They have one major disadvantage, however, they can be quite expensive, especially if the various supporting libraries and add-on packages are also required. It is true that many institutions have a site license for these software packages, but the license may require students to be on the school's network to use the software. It also means that the student is unlikely to have access to the software after they graduate.

The final option for the computational solution of engineering problems is to simply write

your own computer code. Unfortunately, this option requires significant specialized knowledge — knowledge that is rarely retained beyond the course in which it is taught. Writing computer code can also be a very frustrating experience when subtle errors in the code are difficult to identify due to obscure error messages. The result is that students spend most of their time looking for errors in the computer code instead of learning about computations and algorithm development.

There is not a perfect solution to the dilemma of selecting an optimal computer environment for learning computational techniques for solving engineering problems. However, the Python programming language has many advantages that make it the platform of choice here. These advantages include:

1. It is freely available and runs on most major computer platforms including Windows, MacOS, and Linux.
2. It has a tremendous number of additional libraries that are also free and add computational mathematics capabilities. For example, the Numpy library provides Python with capabilities that are similar to those of MATLAB.
3. It is an interpreted language (defined below) and is easier and faster for developing new algorithms than compiled languages.
4. Many libraries of previously compiled algorithms can be imported into Python, which allows for very fast and efficient computations.
5. It is worth repeating – it is free!

2.1.1 Compiled vs. Interpreted Computer Languages

The first high-level programming languages that were developed, such as Fortran or C, were compiled languages. This meant that the programmer would type source code into the computer, this code was compiled into assembler code, and this was ultimated linked to produce a final executable file (see figure 2.1). The advantage of this approach is that the executable that was produced was relatively optimized and efficient for the platform on which it was built. Even today, most numerical software that requires significant computations, e.g., meteorological software, is written in a compiled language. The disadvantage of this approach is that significant expertise and training is required to write computer programs in a compiled language, identifying errors in the source code is often a very difficult and time consuming process, and the resulting program can only be run on the platform or operating system for which it is compiled.

These disadvantages associated with compiled programming languages can largely be addressed through the use of interpreted languages. Common interpreted programming languages include Java, Python, and JavaScript. Even MATLAB can be seen as an interpreted programming language. The source code for these languages is not compiled and linked to form a platform specific executable but is, instead, compiled to an intermediate language (or bytecode) that is run on a 'virtual machine'. The virtual machine is a piece of software that interprets the bytecode and executes the instructions contained in the original source code. One obvious advantage of this approach is that the source code can be run on any computer that has the required virtual machine. Since Python and many associated libraries are available for all the major operating systems, you can execute Python source code almost anywhere. Interpreted languages also tend to be easier to program with because the syntax is more forgiving and the error messages are more informative (although you will still see cryptic error messages and frustrating syntax requirements in all computer languages). The disadvantage of interpreted languages is that they tend to execute instructions more slowly than compiled languages – often by a factor of 10 or more. If we need to multiply 10^{14} numbers by π, a factor of 10 can mean

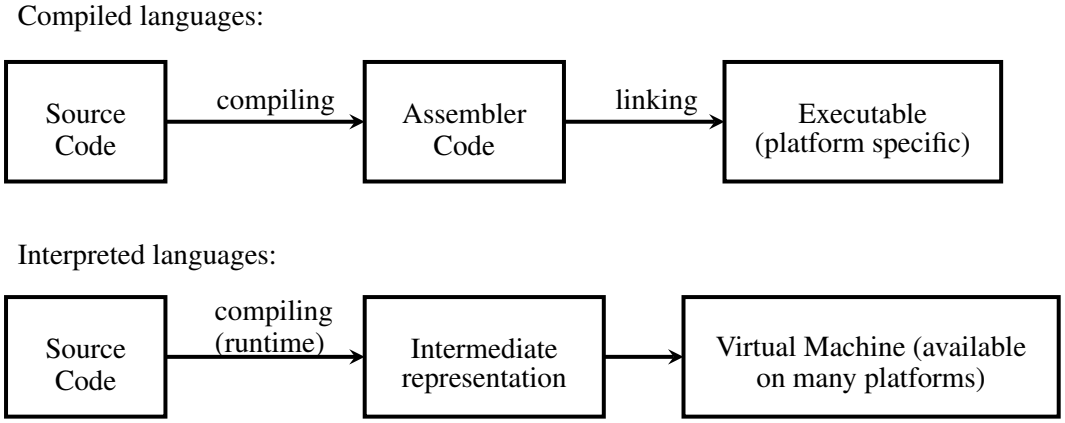


Figure 2.1: The process of going from source code (i.e., a set of instructions) into a running computer program is different for compiled programming languages (top) versus interpreted programming languages (bottom).

the difference between a 1 hour computation and a 10 hour computation. Interpreted languages are getting faster all the time, however, and they are starting to close the gap between compiled and interpreted languages. One common strategy is 'Just-in-time' (JIT) compilation. The basic idea here is that the virtual machine can actually compile important and frequently run source code all the way to a platform specific executable (just like a compiled language). Of course this 'on-the-fly' compiling slows down the execution of the rest of the computer program, but, if a particular set of instructions is executed frequently, it may be more than worth the cost of JIT compilation.

2.1.2 A Note on Python Versions

In 2008, a new version of Python, Python 3.0, was released. This new version contained significant changes from the previous Python 2.x series. In particular, programs written for the Python 3.x series would normally not run on the Python 2.x series of virtual machines, and existing programs written for Python 2.x virtual machines would only occasionally run on Python 3.x series virtual machines. Because of these significant changes, the process of moving existing Python 2.x code so that it can execute on the newer Python 3.x series of virtual machines has progressed relatively slowly. As of 2014, most numerical python libraries are still based on Python 2.7 – the last version of Python in the 2.x series. Most of the examples in this textbook were originally written for a Python 2.7 series virtual machine. It is inevitable that all Python computations will eventually transition to Python 3.x or later virtual machines. In the meantime, it is important to recognize the version of Python that use are using and select the appropriate virtual machine for the code that is being executed.

2.2 Getting Python

The process of learning numerical methods for chemical or biological engineering requires writing and executing computer programs. This book advocates the use of Python for writing and executing these computer programs so it is highly recommended that the reader have access to at least Python 2.7 plus the following libraries:

- Numpy (`www.numpy.org`) – array operation library

- Scipy (`www.scipy.org`) – scientific algorithm library that uses numpy
- Matplotlib (`www.matplotlib.org`) – provides the pyplot and pylab plotting libraries
- SymPy (www.sympy.org) – symbolic mathematics library (optional, used primarily in Chapter 3)

It is also recommended that an integrated development environment (IDE) be used to facilitate the writing of Python Source code. One particularly good IDE is called Spyder (`code.google.com/p/spyderlib`). Figure 2.2 shows the basic layout of the Spyder IDE interface. The input window on the left side of the Spyder program window shows the Python source code that is currently being edited. The code in the source window can be executed or run by selecting 'Run' from the 'Run' menu or simply pressing F5 on most platforms. The upper right-hand screen usually shows documentation when it is available for different functions included with Python or imported libraries. The lower high-hand screen shows a Python console or Python prompt, '>>>'. Basically, the Python prompt is an actively running Python virtual machine and different Python commands can be tested at the prompt. The following exercise illustrates the power of and flexibility of the Python prompt.

Exercise 2.1 At the Python prompt, set the variable 'a' equal to the string 'hello', set the variable 'b' equal to the string 'world' (note the space and the beginning), and then ask Python to 'print a+b'. The exact sequence of instructions should give:

```
>>> a='hello'
>>> b=' world'
>>> print a+b
hello world
>>>
```

Congratulations if you just executed your first Python program! ■

When writing a new Python program, it is often helpful to 'try out' a command or line of code at the Python prompt to observe the result. Having an active virtual machine for testing ideas helps to make Python an efficient language for writing new programs.

2.2.1 Installation of Python

For computers running Windows, two good options for installing Python include:

- pythonxy (code.google.com/p/pythonxy)
- winpython (winpython.sourceforge.net)

Both of these packages include Python plus all the required libraries such as numpy and scipy plus they include the Spyder IDE.

For computers running MacOS, it is easiest to install Macports (www.macports.org) and then Python and all of the required libraries can be easily downloaded and installed with this command in a terminal:

```
sudo port install py27-numpy py27-scipy py27-matplotlib py27-ipython py27-sympy
```

For computers running a Debian-based version of Linux, the following command will install all required libraries:

```
sudo apt-get install python-numpy python-scipy python-matplotlib ipython python-sympy
```

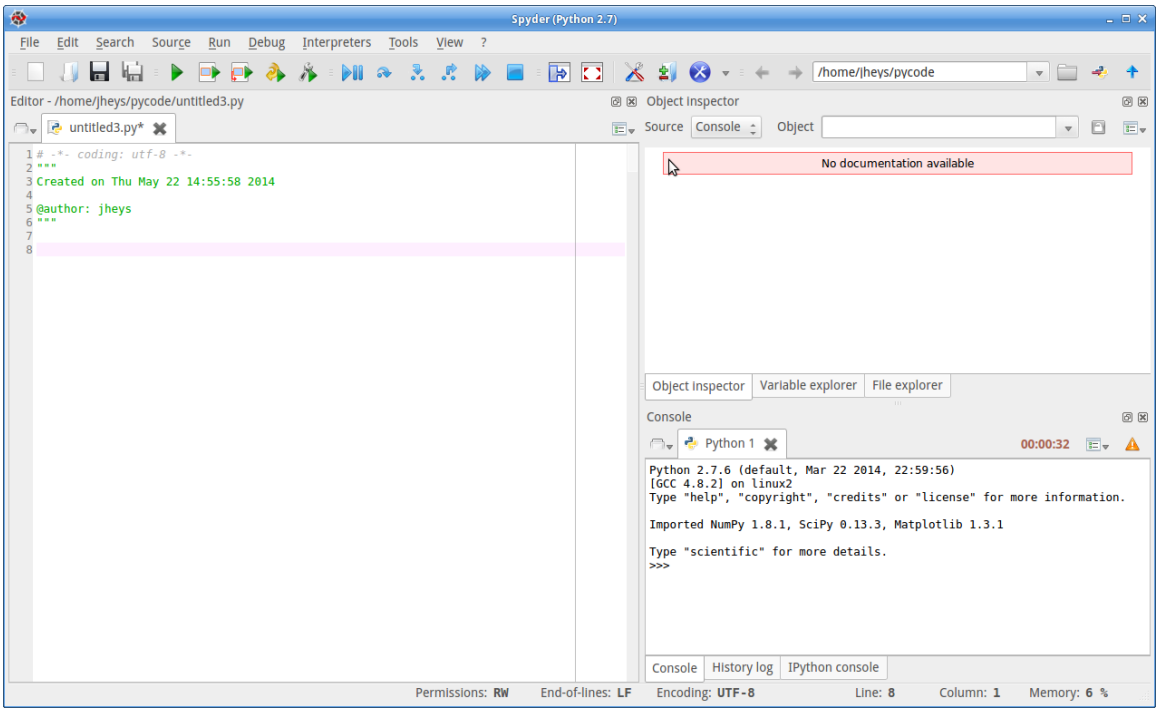


Figure 2.2: A screenshot of the Spyder IDE for Python programming including source code window on the left size, documentation window on the upper right side and Python console for rapid testing and executing the source code in the lower right side.

The FEniCS program, which is used in Chapter 12, is only available for Debian-based versions of Linux (i.e., Ubuntu or Mint Linux) or MacOS, and can be installed on Debian systems using

```
sudo apt-get install fenics
```

2.3 Python Variables and Operators

Programming frequently requires us to assign a variable to a specific piece of data (or something more complex). For example, typing:

```
a = "hello"
```

into the console or a Python script file results in the variable 'a' being *assigned* to character string 'hello'. The word *assigned* is emphasized here because it better reflects the role being played by the equal sign. Whenever Python code contains '=', the object on the right is being assigned to the variable on the left.

In Python (and most other programming languages) we should see:

a = "hello"

as

a ← “hello”

The role of the assignment operator may seem obvious, but many programmers have struggled when the following code did not work:

```
>>> a=4
>>> a=b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>>
```

The novice programmer may believe that the second line (‘a=b’) will result in ‘b’ being set to 4. This will not happen, and, instead, we get an error because the result of executing the code is that ‘a’ is assigned to something that is not defined (the variable ‘b’ has not been assigned). Notice that the end of the Python error message is telling us the problem.

It is often useful in programming to collect multiple objects together into a single container and assign them to a variable. Python includes a number of different types of containers including tuples, lists, and dictionaries. The focus here is on numerical computations and the most useful type of container for these algorithms is a list container. In Python, a list has one or more objects (usually numbers for numerical computations) separated by commas and surrounded by square brackets. Lists should remind us of vectors. The construction of lists is illustrated below.

```
>>> vec1 = [2, 3, 5]
>>> vec2 = [24, 2, 10]
>>> vec1.append([8,2])
>>> print vec1
[2, 3, 5, [8, 2]]
>>> vec1[3] = 87
>>> print vec1
[2, 3, 5, 87]
>>> print vec1+vec2
[2, 3, 5, 87, 24, 2, 10]
>>> print vec1+"what?"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

Two vectors are defined at the console and assigned to the variable ‘vec1’ and ‘vec2’. A nested list is then appended onto the end of ‘vec1’ and then the nested list is replaced with 87 (programmers like to say that Python lists are mutable and can be changed). Lists of the same type can be concatenated as ‘vec1’ and ‘vec2’ are combined. Lists of different types of objects cannot be concatenated as evidenced by the error message at the end.

Now that we know how to assign variables, we are in position to explore operators like '+' and '*', which allow us to effectively use Python like a calculator. The following example illustrates some features of operators.

■ **Example 2.1** The following can be typed into the Python console:

```
>>> a=4
>>> b=2
>>> print a-b
2
```

multiplication and division:

```
>>> print a*b
8
>>> print a/b
2.0
```

exponent and remainder:

```
>>> print a**2
16
>>> print b%a
2
>>> print a%b
0
```

■

2.4 External Libraries

Imagine that we want to calculate sin(1.2). If we try typing that into the Python console or a simple piece of source code, this is what we are likely to see:

```
>>> sin(1.2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
```

We get an error message because the sin() function is not a built-in function in Python. In order to use the sin() function, we need to import the 'math' library into Python. This can be accomplished two different ways and both are shown in the example below.

■ **Example 2.2** To import a library, use the **import** command

```
>>> import math
>>> math.sin(1.2)
0.9320390859672263
```

This is the preferred approach. The first command imports the entire math library , and any functions, methods, or data contained within that library can be accessed by typing: math.name or math.name() where name() is the name of a function in the library. A complete list of functions and values within the math library can be found at `docs.python.org/2/library/math`.

The other approach to importing a library uses the **from** command

```
>>> from math import *
>>> sin(1.2)
0.9320390859672263
```

This approach loads the entire math library into the global Python namespace, which can be thought of as a list of reserved words that are already defined. For example, **import** is a reserved word that is part of the global Python namespace and we should never use **import** for any other purpose. For example, do not try to use 'import' as a variable name. Whenever we load a library into this same global namespace, we greatly increase the number of global terms and invite the possibility of conflict. For example, if we tried to load two libraries that both contain the sin() function (both the math library and the Numpy library, which we use frequently both contain a sin() function), Python would give us an error. There are times when it is easier to use the **from** name **import** $*$ option for loading libraries, but it is usually better to use **import** name. ■

If you tried to run >>> sin (1.2) at the console and were successful, this was a result of using an IDE that was smart enough to load the math library for you.

There are hundreds of libraries that have been written by others that increase the power of Python and save us from having to rewrite code that has already been written by others. In this book, we will use the *math*, *numpy*, *scipy* and *pyplot* libraries extensively.

2.5 Loops and Conditionals in Python

As described previously, the equal sign does not actually compare two objects to see if they are equal. If we wish to compare two things for equality, we need to use == as illustrated:

```
>>> a=4
>>> b=4
>>> a==b
True
>>> a<b
False
```

Beyond the equality comparator, the less than and greater than comparators are frequently helpful. In all cases, the comparator should return a boolean, True or False.

■ **Example 2.3** While the focus in this textbook is on numerical programming, it can be interesting to try out some of the same principles on strings of characters. Consider the following:

```
>>> a="hello"
```

```
>>> b="world"
>>> a==b
False
>>> a<b
True
>>> b<a
False
```

Here the comparator compares two strings to determine which is first alphabetically. ■

Comparators can be extremely helpful in constructing conditional statements. For example, if we want a block of code to only execute when a certain condition is true, we can use an **if** statement:

```
>>> a=4
>>> if a<5:
...     print "smaller"
...
smaller
```

or, in the form of a script:

```
a=4
if a<3:
    print "smaller"
else:
    print "larger"
```

where the code will, of course, print 'larger' upon execution. From these two examples, we can make an *INCREDIBLY IMPORTANT OBSERVATION* (note that I wish that I could make the next sentence flash). In Python, blocks of code are designated using indentation. Every **if** statement has a condition followed by a colon. If the conditional is True the following block of text is executed, and the scope or length of the block is determined by the fact that all lines of code within the block MUST be indented EXACTLY the SAME amount. If the first line in the block is indented 4 spaces (and 4 spaces is the standard Python style), then every line must be indented 4 space. If you mess up and indent one line with only 3 spaces or a tab, error messages and chaos will follow. This same requirement extends to nested conditional statements, as shown.

■ **Example 2.4** An example of nested conditional statements:

```
a=4
if a>3:
    print "a␣is␣greater␣than␣3"
    print "adding␣1"
    a += 1 # this is identical to a=a+1
```

```
    if a>4:
        print "a␣is␣greater␣than␣4"
        print "subtracting␣1"
        a -= 1
print "a␣=␣", a
```

Upon execution, this code generates: a is greater than 3
adding 1
a is greater than 4
subtracting 1
a = 4 ■

Another situation where we have to be careful about indentation is loops. Loops allow us to repeatedly run a block of code for either a set number of iterations or until some condition is met. The **for** loop is the most common type of loop and is illustrated below.

■ **Example 2.5** Let us construct a loop that executes 5 times and has a counter that incrementally increases in value every time through the loop.

```
import math

for i in range(5):
    j = math.sin(i)
    print i, j
print "finished"
```

Upon execution, the output from this code should be:

```
0 0.0
1 0.841470984808
2 0.909297426826
3 0.14112000806
4 -0.756802495308
finished
```

■

Exercise 2.2 The **range** function in Python supports the following arguments:
range(start , stop , step). If only one input is given, it is treated as the stop value and the function returns the integers from 0 to stop−1. If two values are given, they are treated as start and stop values, and 3 values are treated as start, stop, and step size. Using this information, try to construct a loop that prints out the odd integers from 7 to 17, including 17.

We need to set start to 7, step should be set to 2 so that we only have odd integers, but how do we get the loop to include 17, but exclude 19?

The following will not work:

```
for i in range(7,17, 2):
```

```
    print i
print "finished"
```

This code will stop at 15. Instead, because **range** does not include the value of stop in the sequence, we need to set stop to 18 or 19. You may also want to try setting the value of start to 7.5 or some other non-integer. The result will not be good because **range** requires all input values to be integers. ■

2.6 Functions

We have already used a number of built-in functions and functions from external libraries, include the math.sin and **range** functions. In many situations, it is very helpful to write our own functions. Advantages of writing functions include the fact that it becomes easier in reuse the code you have written previously, and functions help to break our programs up into manageable pieces, which makes programming easier. The keyword **def** is used to define a function in Python. This keyword should be followed by the name of the function and variable names for any inputs. The set of instructions that make up the function appear in the block below the fist line. The construction of a function is illustrated through the following examples.

■ **Example 2.6** We want to write a function that will print out the area of a triangle given the size of the base and the height.

```
def triangle(base,height):
    area = 0.5*base*height
    print area

triangle(2,3)
```

Upon execution, this code should print 3.0 to the console output. ■

Of course having a function that just prints something to the screen after a calculation is probably not that useful. Instead, we should try to construct a function that returns the results of the calculations whenever possible. This can be illustrated by rewriting the function in the above example so that it returns the area (and then prints it to the screen).

```
def triangle(base,height):
    area = 0.5*base*height
    return area

size = triangle(2,3)
print size
```

Note the use of the keyword **return** at the end of the triangle function.

Exercise 2.3 Write a function that can be given the number of people at a table. Then, the function calculates the arc length for a single slice of pizza if a single 16-inch diameter pizza is to be divided evenly among the people at the table and everyone receives just one slice.

```
import math
def arclength(numPeople):
    circumference = 16*math.pi
    if(numPeople < 1):
        print "Error: must have at least one person"
        return 0
    else:
        length = circumference/numPeople
        return length

print arclength(6)
```

The code above should return 8.3776, indicating that each of the 6 people in the test problem should receive a slice with an arc length of 8.4 inches. You may want to try modifying the code so that it calculates the arc length when everyone receives more than one slice, but they all still receive the same number of slices. ■

As we will see later, it is often useful to combine related functions together into a single file with a *filename*.py extension. These functions can then be imported into other programs later on using **import** filename and called using filename.function. This is a great way to recycle code we have already written.

Exercise 2.4 The vapor pressure of a pure liquid, written p^*, is a strong function of temperature. To calculate the vapor pressure at a given temperature, T, it is common to use Antoine's equation:

$$\log_{10} p^* = A - \frac{B}{T+C}$$

where A, B, and C are constants that can be looked up for different liquids. Write a function that has A, B, C, and T (in ^oC) as inputs and returns the vapor pressure, p^*. Hint: $10^{\log_{10} x} = x$. ■

Exercise 2.5 Write a function that has A, B, C, and p^* (in mm Hg) as inputs and returns the temperature, T. ■

2.7 Numpy Basics

The numpy library adds powerful linear algebra data structures to Python. It allows us to construct and manipulate vectors and tensors very efficiently, and it is also widely used by other libraries that provide, for example, plotting and linear algebra solvers. The goal of this section is to provide a very brief introduction to a few important features of numpy. Throughout the rest of the book, additional features and options will be demonstrated. The online documentation and tutorials for numpy are also a very valuable source of information about the numpy library.

2.7.1 Array and Vector Creation

The are a number of different interfaces provided by numpy for constructing vectors and arrays. The simplest approach is to simply pass the numpy.array() method a Python list:

```
import numpy

myvector = numpy.array([5,3,7],dtype=numpy.float64)
myarray = numpy.array([[2,3],[6.7,1.0]])
print myvector
print myarray
```

If a single, 1-dimensional list is passed into the numpy.array() method, a 1-dimensional vector is created. It is optional to tell numpy the type of numbers that are stored in the array, integers, floats, or complex numbers. Using 'float64' causes every number in the array to be stored as a 64-bit floating point number. This is the default number type in numpy, so we will usually not set the 'dtype' option. If a nested list is passed into the numpy.array() method, a 2-dimensional tensor is created.

This approach to array construction works great for very small arrays where we already know the values. In practice, however, we will usually construct an array of zeros of the size we want, and then use loops to insert the desired values into the array. This approach is illustrated in the example below.

■ **Example 2.7** Array construction by first initializing an array to zero and then using a loop.

```
import numpy

size = 3
myarray = numpy.zeros([size,size])

for i in range(size):
    for j in range(size):
        myarray[i,j]=1.0/(i*j+1.0)

print myarray
```

The program constructs a square, 2-dimensional array to whatever size is specified by the size variable. The zeros within the array are then replaced by the values calculated inside the nested loop. This approach to first allocating space for an array and then overwriting the initial values is typically more computationally efficient than first constructing an empty array and then appending values. ■

We need to make a very important observation from the previous example: *in Python, lists, vectors, and arrays are indexed starting with zero!* In other words, if we have a vector that contains 5 numbers (i.e., it has length 5), those numbers are accessed with the indices 0, 1, 2, 3, and 4. Notice how if we loop from 0 to a number less than the size of the vector (in this case we loop from 0 to 4), we loop through all the indices without including 5. This important observation is important, and forgetting how vectors are indexed leads to many troublesome bugs in the code.

The final method that is sometimes helpful for array construction is to build an array where the values increase incrementally. For example, we might wish to construct and array of length 10 that contains the integers from 1 to 10. One nice feature of such a vector, is that it also allows us to demonstrate how to access a subsection of the vector.

■ **Example 2.8** A sequential array and slicing:

```
import numpy

myarray = numpy.arange(1,11)
print myarray
myarray[3:6] = numpy.array([300,400,500])
print myarray
```

The code in this example begins by constructing a vector from 1 to 11, but then, we replace 3 of the values within the vector with much larger values. Notice how we replace the values stored at indices 3,4, and 5 because the slicing command – 3:6 – does not include the last index (6) in the slice. Also recall the first entry in the vector (in this case '1') is at index zero. So index 3 initially contains the value 4, which is replaced with 300. Then, the value at index 4, which is initially 5, is replaced with 400. The result of running the example code should be:

```
[  1   2   3   4   5   6   7   8   9  10]
[   1    2    3  300  400  500    7    8    9   10]
```

■

2.7.2 Array Operations

To review, we have discussed the construction of numpy vectors and arrays, and we have discussed how to access the various entries within the vectors and arrays. Before closing out this very brief introduction to numpy, let us explore a few different operations that can be performed on matrices and vectors with the following example.

■ **Example 2.9** Try running the code below:

```
import numpy

myarray = numpy.arange(5)
print myarray
print myarray.shape
myarray = myarray*4
print myarray
yourarray = numpy.ones(5)
theirarray = myarray - 3*yourarray
print theirarray
print numpy.dot(myarray,theirarray)
itsarray = numpy.outer(myarray,theirarray)
print itsarray
print itsarray.shape
```

This code segment begins by building a sequential vector of length 5. The 'shape' property of that vector should return '(5,)', which tells use that the array has 5 rows and no additional columns. This initial vector is then multiplied by 4 and a second and third array are built. The third array,

called 'theirarray' is actually built by taking the first array, 'myarray', and subtracting an array of 3's from it. Please note that adding or subtracting one array from another requires that the arrays have the same size and shape – that is why it is often a good idea to print the value of size and shape to the screen so that you can confirm the sizes are the same. The example ends by taking a dot product of two vectors (again, they must have the same size) and an outer product. ■

This example barely scratched the surface of what is possible with numpy. For example, we have not yet covered matrix inversion or eigenvalue calculations, but many of these topics will be discussed in later chapters.

2.8 Matplotlib Basics

Matplotlib is one of many libraries that adds plotting capabilities to Python. It is a particular good choice here because it is well integrated with numpy and it provides relatively high quality output. To use Matplotlib within our Python scripts, we will import the either pylab library, which gives an interface similar to MATLAB's plotting functionality, or the pyplot library. Pylab and pyplot provide the same functionality, but pylab is slightly easier to use from the Python prompt. Let us begin with a simple example.

■ **Example 2.10** The following script builds a vector with 100 entries that span from 0 to 10.0 using numpy's linspace() function. Then, pylab is used to generate a scatter plot where x is from 0 to 10 and y is $\cos(x)$.

```
import numpy
import pylab

x=numpy.linspace(0,10, num=100)
y = numpy.cos(x)
pylab.plot(x,y)
pylab.show()
```

The resulting figure is shown on the left of Figure 2.3. The final line of example code (pylab.show()) is required because it causes the figure to persist on the screen and become interactive. If this line is omitted, the figures is either never plotted to the screen or is plotted for a fraction of a second before it disappears. Think of the pylab.show() function as causing the program to pause and wait for the user to decided what they want to do with the plot. The pylab.show() function is not required on all operating systems.

If the function call to pylab.plot(x,y) is replaced with pylab.plot(x,y,'bo'), the plot is constructed with blue circles instead of a solid line, as shown in the right half of figure 2.3.

■

Matplotlib is a comprehensive plotting library, large enough that an entire book has been written to document all of the many different style of figures and options. For more information about the library as well as documentation describing the interfaces into the library, please see the tutorials posted on the library website: `matplotlib.org`.

Before ending this chapter, the power of Matplotlib is illustrated through a slightly more complex example.

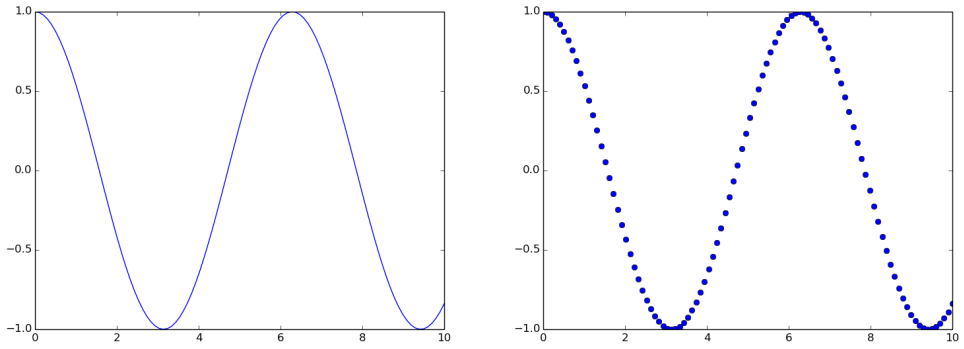


Figure 2.3: A figure of a cos()-wave generated using pylab with a solid line (left) and blue circles (right).

■ **Example 2.11** This example shows the contour plot of a function with two independent variables.

```
import pylab
import numpy

def f(x,y):
    return (1-x/2+x**2+y**3)*numpy.exp(-x**2-y**2)

n = 256
x = numpy.linspace(-2,4,n)
y = numpy.linspace(-2,4,n)
X,Y = numpy.meshgrid(x,y)

C = pylab.contour(X, Y, f(X,Y), 8)
pylab.clabel(C, inline=1)
pylab.colorbar(C, orientation='vertical')
pylab.show()
```

The function that is plotted is defined by the function $f(x,y)$. The independent variables, $x \in (-2,4)$ and $y \in (-2,4)$, are stored in vectors created using numpy's linspace() function. The numpy meshgrid function extends the 1-dimensional vectors over a 2 dimensional array. The contour plot consists of 8 contour lines, which are labeled, and a colorbar is added to the right of the plot. Colorbars are largely unnecessary and unattractive for this style of contour plot, but one is included here to illustrate the simplicity with which it can be added. The resulting figure is shown below (figure 2.4).

■

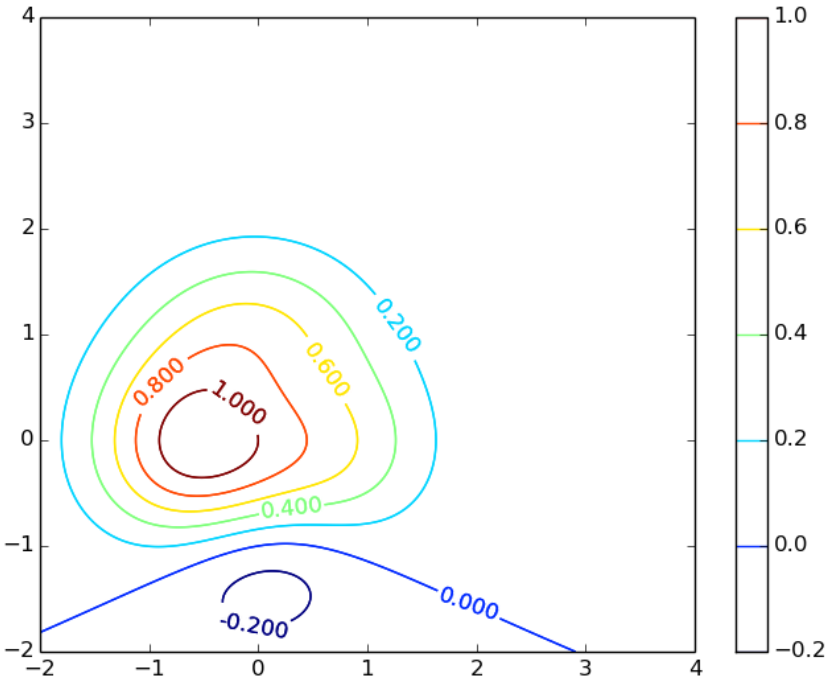


Figure 2.4: A contour plot of a function, $f(x,y) = (1 - x/2 + x^2 + y^3)\exp(-x^2 - y^2)$ for $x \in (-2,4)$ and $y \in (-2,4)$

2.9 Additional Resources

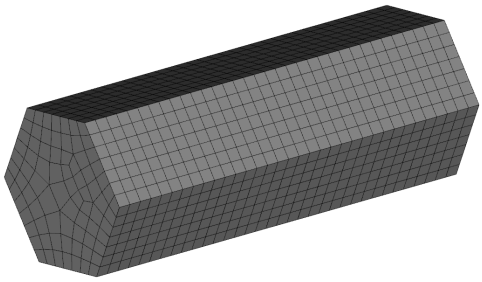
Recommended books on general programming in Python:

- Learning Python by Mark Lutz [Lut13]
- Python in a Nutshell by Alex Martelli [Mar09]
- Think Python by Allen Downey [Dow12]

Recommended book on numerical programming in Python:

- Numerical Methods in Engineering with Python by Jaan Kiusalaas [Kiu10]
- Python Scripting for Computational Science by Hans Petter Langtangen [Lan10]
- Matplotlib for Python Developers by Sandro Tosi [Tos09]

Introduction
Symbolic Mathematics Packages
An Introduction to SymPy
Factoring and Expanding Functions
Derivatives and Integrals



3. Symbolic Mathematics

3.1 Introduction

When we have a mathematical equation or equations that describe some phenomena, there are basically two approaches that we can adopt to solve the equation. First, the method that we are probably most familiar with involves using the principles we learned in Algebra, Calculus, and other mathematics courses to manipulate the equations to determine the variable(s) of interest. For example, you have hopefully learned previously that when faced with the equation $2+x=5$, you simply subtract 2 from each side of the equation and establish that $x=3$. This algebraic process that we have previously learned requires us to symbolically manipulate and equation until we arrive at the desired solution, hopefully[1] This approach has the advantage of giving us an exact solution, but the disadvantage that it is limited to the set of problems where it is possible to obtain a solution through symbolic manipulations. The other approach involves determining an approximate solution, usually through an automatic iterative process on a computer. This approach is typically referred to as obtaining a numerical solution (although the careful reader may note that a better name is numerical approximate solution). The advantage of a numerical approach is that approximate solutions to a larger range of equations is possible, but the disadvantage is that the solutions are only approximate and usually require a computer.

It may be unusual to include a chapter on symbolic mathematics in a book that is focused on numerical methods, but for equations that can be solved by a symbolic approach, it is usually the preferred approach. Experienced engineers and mathematicians can usually determine relatively quickly if a set of mathematical equations is likely to be solvable using a symbolic approach. For novices, however, it is usually a good idea to tryout a symbolic approach, such as the one described in this chapter just to check-and-see if a symbolic solution is easily available. This chapter on symbolic computations also provides a good review of some Python principles that were covered previously, including the use of external libraries.

[1] An interesting historical example of mathematicians trying to use algebra and symbolic manipulation to solve the quintic equation can be found in 'The Equation that Couldn't Be Solved' by Mario Livio [Liv06]. The roots of a quintic equation are typically easy to determine using a numerical process.

3.2 Symbolic Mathematics Packages

A large number of software packages have been developed for symbolic mathematics, and the capabilities of the various packages are not the same. As of 2014, Wikipedia listed over 30 different software packages for symbolic mathematics. The packages listed below are all commercial software, but they are among the most popular and site licenses are available on many university campuses.

Maple One of the oldest software packages for symbolic mathematics, it was originally written at the University of Waterloo in the early 1980's. The name is a reference to Maple's Canadian heritage. While it was quite popular before 1995, its popularity declined due to a user-interface that was difficult to use. The new user-interface, introduced in 2005, is significantly better and similar to the other packages available for symbolic mathematics.

Mathcad One of the first mathematics software packages with a graphical user interface and support for SI units. The software is popular for producing reports and documentation that include mathematical calculations. The symbolic mathematics capabilities are sufficient for most purposes, but not as strong as many of the other packages listed here.

Mathematica Initially released in 1988, Mathematica was one of the first symbolic mathematics packages with a graphical user interface, which is referred to as the 'front end'. While the creation of custom algorithms remains difficult in Mathematica, it is still one of the most popular platforms for computational mathematics. Some of the functionality is available free-of-charge through the Wolfram Alpha website.

MATLAB's Symbolic Mathematics Toolbox MATLAB is primarily used for numerical computing, but the Symbolic Mathematics Toolbox provides some symbolic capabilities. Depending on the type of license, this is one of the most expensive options listed here.

One additional package that should be highlighted here is the Sage (previously SAGE, System for Algebra and Geometry Experimentation) mathematics software, which is free and licensed under a GNU General Public License. Of particular interest here is the fact that Sage uses the Python programming language so individuals familiar with Python with have a more modest learning curve. Sage makes extensive use of Python libraries, including NumPy, SciPy, and SymPy, in order to avoid having to reimplement large amounts of existing code. While Sage is an excellent resource for mathematical computing, it is not covered in detail in this book because of the large size of the platform. Many of the pieces of Sage, including the libraries listed previously are covered here, but the curious reader is encourage to explore the Sage software and its features. The browser-based notebook interface available for Sage may be of particular interest.

The focus in this chapter will be on the use of the SymPy library, which adds support for symbolic mathematics to Python [Tea14]. SymPy is written entirely in Python and does not require any external libraries. SymPy is included with many distributions of Python that are focused on scientists or engineers including Pythonxy. Installation on Linux systems or MacOS systems using MacPorts is also straightforward. Further information related to downloading and installing SymPy as well as comprehensive documentation is available on the SymPy website: `www.sympy.org`.

3.3 An Introduction to SymPy

The SymPy library is imported into Python using the commmand: **import** sympy. As a result, all methods associated with the library are accessed using standard sympy.method() format. Alternatively, the entire library maybe imported as **from** sympy **import** $*$ command, but use of this format is discouraged.

Once the SymPy library has been imported, the next step is to declare the symbolic variable or parameters that will be present in the equations that we plan to manipulate or solve symbolically. The sympy.symbols() class transforms a string that lists the variables or parameters into instances of the Symbol class. For example, the command: E, m, c = sympy.symbols('E␣m␣c') or E, m, c = sympy.symbols('E,␣m,␣c') converts the string 'E m c' into 3 different symbolic variables that maybe used in later to define mathematical expressions or equations or in future symbolic mathematics functions. It is strongly recommended that the Symbol name (i.e., the variable on the left side of the '=' sign) be the same as the variable name in the string that is passed into the symbols() function.

Let us begin by demonstrating the SymPy library on a classic algebraic problem, factoring a quadratic polynomial, $a \cdot x^2 + b \cdot x + c = 0$, to determine its roots. As taught in a typical algebra course, the roots of a quadratic polynomial can be determined using the quadratic equation, $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. The derivation of the quadratic equation only requires straightforward algebraic manipulation of the quadratic polynomial to solve for x. The quadratic equation, i.e., the roots of the quadratic polynomial, can also be derived using SymPy as is illustrated in the example below.

■ **Example 3.1** Determine the roots of the quadratic polynomial: $a \cdot x^2 + b \cdot x + c = 0$.

```
import sympy

a,b,c,x = sympy.symbols('a␣b␣c␣x')
expr = a*x**2 + b*x + c
print sympy.solve(expr,'x')
```

The quadratic polynomial is stored in the variable expr, and using the solve() function in the SymPy library allows for the determination of the roots of the polynomial. The output from the example code should be: [(−b + sqrt(−4∗a∗c + b∗∗2))/(2∗a), −(b + sqrt(−4∗a∗c + b∗∗2))/(2∗a)] , which is a Python list containing the two roots of the polynomial.

If the values of a, b, and c are known, then the SymPy library may still be used. For example, the following modification to the code above solves for the roots of $3x^2 + 4x + 5 = 0$.

```
import sympy

x = sympy.symbols('x')
expr = 3*x**2 + 4*x + 5
print sympy.solve(expr,'x')
```

In this case, the output should be: [−2/3 − sqrt(11)∗I/3, −2/3 + sqrt(11)∗I/3] where $I = \sqrt{-1}$. ■

Another very helpful function in SymPy is the subs() method, which substitutes one expression for another. A simple example of this is replacing x in a polynomial with a specific value as illustrated below.

```
import sympy
```

```
x = sympy.symbols('x')
expr = 3*x**2 + 4*x + 5
print expr.subs(x,2.0)
```

Here, 2.0 is substituted for x in the expression and the result is simplified to 25.000. The conversion of expressions into floating point numbers can use evalf(), and the desired precision can be passed into the function. For example replacing the last line in the previous example with **print** expr.subs(x,2.0). evalf(16) evaluates the result after substitution to 16 digits of precision.

The solve() function was previously used for solving an algebraic equation. The syntax of this function is solve(equations, variables) where equations may be a single equation or a list of equations (a list in enclosed in square brackets, [item, item, item]). The number of variables list, of course, must equal the number of equations. In the example below, the solve() function is used to solve a common problem in describing the behavior of gases.

■ **Example 3.2** The *van der Waals* equation of state is a common equation for relating the temperature (T), pressure (P), and specific volume ($\hat{V}$ of a non-ideal gas. The equation may be written as

$$P = \frac{RT}{\hat{V} - b} - \frac{a}{\hat{V}^2}$$

where

$$a = \frac{27R^2T_c^2}{64P_c}$$

$$b = \frac{RT_c}{8P_c}$$

$$R = 0.08206\, L \cdot atm/(mol \cdot K)$$

and T_c and P_c are the critical temperature and pressure of the gas, respectively. Our goal is to calculate the specific volume of ammonia ($T_c = 405.5K$ and $P_c = 111.3atm$) at $T = 420K$ and $P = 43.4atm$. Before using SymPy to solve for the specific volume, we need to rewrite our equation(s) so that all terms are on one side of the '=' sign. Thus, we will write van der Waals equation as

$$0 = P - \frac{RT}{\hat{V} - b} + \frac{a}{\hat{V}^2}\ .$$

We are now ready to solve for the specific volume.

```
import sympy

R = 0.08206
P = 43.4
T = 420.0
Tc = 405.5
```

```
Pc = 111.3
a = 27*(R**2 * Tc**2 / Pc)/64
b = R * Tc / (8 * Pc)
V = sympy.symbols('V')
f = P - R*T/(V-b) + a/(V**2)
print sympy.solve(f,V)
```

The output from this example is [0.70088, 0.06531 − 0.02985∗I, 0.06531 + 0.02985∗I]. The equation is cubic with respect to $\hat{V}$ so we should not be surprised by getting 3 solutions (i.e., 3 roots). In this case, it is simple to determine the correct solution as two of the solutions are complex and obviously not physical. We should also note that neither Python or SymPy support units so the onus is on the user of the software to ensure that consistent and correct units are used in all calculations. ■

The solve() function is not limited to a single algebraic equation; it also supports multiple equations and unknowns. However, large systems of equations and unknowns are typically solved more efficiently using a numerical approach. The use of solve() for a relatively small and simple system of equations is demonstrated in the example below.

■ **Example 3.3** Use the solve() function to solve the following system of equations for x and y.

$$x^3 + y + 1 = 0$$

$$y + 3x + 1 = 0$$

It is important to note that both equation already have all terms on one side of the '=' sign.

```
import sympy

x, y = sympy.symbols('x y')
eq1 = x**3 + y + 1
eq2 = y + 3*x + 1
sol = sympy.solve([eq1, eq2], [x,y])
print sol[0]
print sol[1][0].evalf(), sol[1][1].evalf()
print sol[2][0].evalf(), sol[2][1].evalf()
```

We should first observe that the solution is a list-of-lists: it is a list of 3 solutions and each solution is a list of the x,y-pair that satisfies the equations. Also, note the use of evalf() to simplify the 3 solutions to this system of equations: ((0,1), (-1.732, 4.196), (1.732, -6.196)). We might be surprised that there are 3 different (x,y) pairs that satisfy this system of equations, but if we rearrange the equations slightly into

$$y_1 = -x^3 - 1$$

$$y_2 = -3x - 1$$

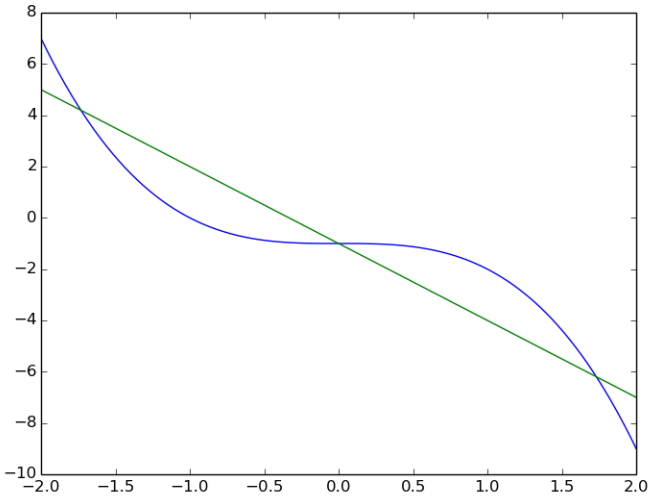


Figure 3.1: x, y-diagram of the two equation that were simultaneously solved in the previous example.

and plot the 2 curves using pylab, we can see that the two curves cross 3 times at the 3 solutions given previously.

■

3.4 Factoring and Expanding Functions

One of the more tedious and error-prone routine tasks in mathematics is expanding and factoring polynomial equations. SymPy can usually perform this task automatically. We can use SymPy to expand the function $f(x) = (x+2)^3 + 3$ using the following code.

```
import sympy

x = sympy.symbols('x')
f = (x+2)**3 + 3
print sympy.expand(f)
```

Running this short program gives us the expanded polynomial: $x^3 + 6x^2 + 12x + 11$. This expansion can also be performed by humans, but the error-rate and time lost are both high.

An even more useful feature of SymPy is its ability to factor polynomials. A student could spend hours trying to factor the polynomial $27x^3 + 135x^2 + 225x + 125$, but SymPy can factor it in a few seconds using the following code.

```
import sympy

x = sympy.symbols('x')
```

```
f = 27*x**3 + 135*x**2 + 225*x + 125
print sympy.factor(f)
```

Running this code block prints the factorization, $(3x+5)^3$ to the standard output. Using SymPy to expand and factor polynomials is simple enough that it can often be done straight from the Python prompt with the construction of a complete script.

■ **Example 3.4** In equilibrium kinetics, it is often necessary to expand and factor polynomial. Consider the water-gas shift reaction

$$CO + H_2O \rightleftharpoons CO_2 + H_2$$

proceeding to equilibrium at a temperature where the equilibrium coefficient is

$$K = \frac{y_{CO_2} y_{H_2}}{y_{CO} y_{H_2O}} = 1.4 \, .$$

(This problem is adapted from an example in Felder and Rousseau [FR05].) If the feed to this reactor is 2.0 mol of CO and 2.0 mol of H_2O and the extent that this reaction proceeds to the right (i.e., the extent of reaction) is ξ, then we can write the mole fraction of each species as

$$y_{CO} = 2 - \xi$$

$$y_{H_2O} = 2 - \xi$$

$$y_{CO_2} = \xi$$

$$y_{H_2} = \xi$$

and the equilibrium equation as

$$K = \frac{\xi \cdot \xi}{(2-\xi)(2-\xi)} = 1.4 \, ,$$

or

$$\xi^2 - 1.4(2-\xi)^2 = 0.0 \, .$$

The goal is to determine the extent of reaction, ξ, which is relatively straightforward but requires expanding out the polynomial. The following code illustrates the use of SymPy to either help us expand the polynomial or solve for ξ.

```
import sympy

xi = sympy.symbols('xi')
f = xi**2/((2-xi)*(2-xi)) - 1.4
g = xi**2 - 1.4*(2-xi)*(2-xi)
print sympy.expand(g)
print sympy.simplify(g)
print sympy.factor(g)
print sympy.solve(f)
```

The first 3 lines printed by the program all generate the simplified quadratic polynomial associated with the equation for ξ, specifically, they give $-0.4\xi^2 + 5.6\xi - 5.6$, which can be solved for ξ using the quadratic equation. The last line prints out the solution, 1.84 or 12.91. Since we start with only 2.0 mols of CO and H_2O, the only physically possible root is 1.84 mol. ■

Exercise 3.1 In the field of Process Controls, it is sometimes necessary to perform what is called a partial fraction decomposition. Consider the equation

$$F(s) = \frac{1}{s(s+0.5)} .$$

The process of partial fraction decomposition requires that we determine the constants c_1 and c_2 such that the following equation is satisfied:

$$\frac{1}{s(s+0.5)} = \frac{c_1}{s} + \frac{c_2}{s+0.5} .$$

Fortunately, SymPy includes the function sympy.apart() that can usually take a partial fraction decomposition automatically. Write a script that determines the value of c_1 and c_2 using a partial fraction decomposition.

Solution: $c_1 = 2.0$ and $c_2 = -2.0$. ■

3.5 Derivatives and Integrals

It is also possible, and often very helpful, to use symbolic mathematics software when taking derivatives and integrals. Symbolic derivatives can be obtained using the sympy.diff() function. Let us begin by taking the derivative of a $\sin(x)$ function.

```
import sympy

x=sympy.symbols('x')
print sympy.diff(sympy.sin(x),x)
```

The output from this code is $\cos(x)$, as expected. Passing in additional symbols (i.e., variables) into the sympy.diff() function causes additional derivatives to be taken. For example, if the last line in the previous code block is replaced with **print** sympy.diff(sympy.sin(x),x,x), the output is $-\sin(x)$. Alternatively, adding 'y' to the symbols list and asking for the derivative with respect to 'y', **print** sympy.diff(sympy.sin(x),x,y), gives the expected result of zero.

■ **Example 3.5** When designing a chemical reactor, we sometimes have a mathematical expression that relates the concentration of a species in the reactor to time. For example, assume that we know that

$$C_A = C_{A0} \exp(-k \cdot t)$$

where C_A is the concentration of species A, C_{A0} is the initial concentration of A, k is a constant, and t is time. We would like to take the derivative of C_A with respect to time to determine the rate of the reaction. The following Python code will determine the derivative:

```
import sympy

Ca0, k, t = sympy.symbols('Ca0 k t')
Ca = Ca0 * sympy.exp(-k*t)
print sympy.diff(Ca, t)
```

The output from the script is $-C_{A0} \cdot k \cdot \exp(-k \cdot t)$, which the observant reader will recognize can be simplified to $-k \cdot C_A$. ■

Symbolic integration tends to be even more helpful than symbolic differentiation, probably because integration by hand is more difficult than differentiation. The sympy. integrate () function is used for symbolic integration. The code block below demonstrates both the single and double integration of a simple $\sin(x)$ function.

```
import sympy

x,y=sympy.symbols('x y')
print sympy.integrate(sympy.sin(x),x)
print sympy.integrate(sympy.sin(x),x,x)
print sympy.integrate(sympy.sin(x),x,y)
```

Upon execution, the code outputs the expected results of $-\cos(x)$ and $-\sin(x)$. What should we expect from the final print statement, which integrates $\sin(x)$ first against x and then against y? Well, the first integration will yield $-\cos(x)$, and the second integration will treat any function of x as a constant and, as a result, integration against y will give $-y \cdot \cos(x)$.

■ **Example 3.6** As students learn in a reactor design course, the sizing of a batch reactor containing an irreversible second-order reaction $A \rightarrow B$ requires that we evaluate the integral:

$$\int_0^X \frac{dX}{(1-X)^2} .$$

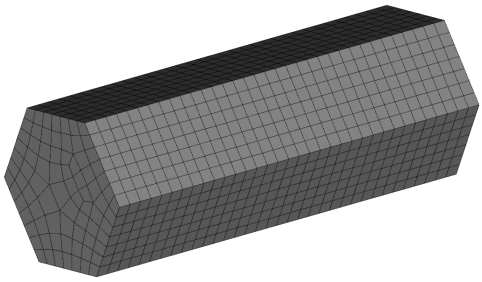
While this integral is still relatively simple, it never hurts to check our work with a symbolic mathematics program. The following script will evaluate this integral using SymPy.

```
import sympy

X = sympy.symbols('X')
t = sympy.integrate(1/((1-X)**2), (X, 0, X))
print t
print sympy.simplify(t)
```

A couple interesting observations can be made from this block of code. First, notice the function interface for definite integrals: (variable , lower_bound, upper_bound), which, for the problem of interest here is (X, O, X). The result of symbolic integration is initially (i.e., the first print statement): $-1 - \frac{1}{X-1}$, which is correct, but not the simplest form possible. To obtain the more common, and

simpler result, the sympy.simplify () function is used to get the standard result: $-\frac{X}{X-1}$. ■

Example Problem
A Direct Solution Method
Computational Cost
Iterative Solution Methods
Vector Norms
Jacobi Iteration
Gauss-Seidel Iteration
Convergence of Iterative Methods



4. Linear Systems

A single, linear, algebraic equation is trivial to solve. In engineering, however, we are often faced with the more difficult challenge of solving for multiple unknowns (e.g., x_1, x_2, x_3, etc.) that are related by multiple, linear algebraic equations. In the previous chapter on symbolic mathematics, we explored an approach, sympy.solve(), that gave an exact solution. This approach, however, is limited to problems with only a few equations and a few unknowns (typically less than 10). Our goal in this chapter is to learn methods that can handle 1000's or even million's of unknowns.

If the equations are truly linear – the unknowns are not multiplied by each other or themselves, nor are their nonlinear terms within the equations, such as $\sin(x_1)$, then we can write the system of equations as

$$\begin{aligned} a_1x_1 + a_2x_2 + \ldots + a_nx_n &= f_1 \\ b_1x_1 + b_2x_2 + \ldots + b_nx_n &= f_2 \\ \vdots &= \vdots \\ z_1x_1 + z_2x_2 + \ldots + z_nx_n &= f_n \end{aligned}$$

where a, b, …, z, and f each represent n-constants. The system has n-equations and n-unknowns. It is often simpler to write our this system of equations in the form of a matrix:

$$\begin{bmatrix} a_1 & a_2 & \ldots & a_n \\ b_1 & b_2 & \ldots & b_n \\ \vdots & & \ddots & \vdots \\ z_1 & z_2 & \ldots & z_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \tag{4.1}$$

The matrix on the left side and the right-hand-side vector both contain given constants, but the vector, x, in the middle contains the unknowns. This problem is often written $\mathbf{A} \cdot \mathbf{x} = \mathbf{f}$. Our goal in this chapter is to learn different approaches for solving for $\mathbf{x}$ whenever we have a system of linear equations.

> **Note** Throughout this book, a bold lowercase variable (e.g., **x**) is used to represent a vector. A bold uppercase variable represents a matrix (e.g., **A**).

4.1 Example Problem

Distillation columns are used to separate mixtures of compounds based on differences in boiling points. The development of a mathematical model of a distillation column typically results in 100's or 1000's of linear and nonlinear equations. Let us explore a simplified mathematical model for a distillation column where the input is known: 30 kg/s of methane, 25 kg/s of ethane, and 10 kg/s of propane. The input mixture is separated into 3 outflow streams: a overhead stream that is rich in methane (90%) and does not contain any propane, a middle stream that is rich in ethane (50%) and a bottom stream that is rich in propane (70%). Propane is the least volatile of the 3 components in the distillation column, and, hence, is the most likely to be separated into the bottom stream. Figure 4.1 contains additional information for the composition of the outflow streams – note that x is used for mass fractions (i.e., the fraction of a total stream that is a specific compound) and m is used for mass flow rates (in kg/s). Subscripts denote specific compounds – methane (M), ethane (E), and propane (P), or numerical subscripts represent different stream numbers so m_1 is the total mass flow rate of the entire stream 1.

Ideally, distillation columns are operated at steady state, and every kg of each compound that enters the column is matched by a kg of that same compound leaving the column. This must be true due to the Conservation of Mass. Using this principle, an equation that equates the mass flow rate of methane into the column to the mass flow rate of methane leaving the column can be written.

$$m_{in} = m_{out}$$

$$30kg/s = 0.9m_1 + 0.3m_2 + 0.1m_3$$

The second equation utilizes the fact that the mass flow rate of methane in stream 1 must equal the total mass flow rate of that stream (m_1) multiplied by the fraction of the stream that is methane ($x_M = 0.9$). Since there are 3 outflow streams, the mass flow rate into the column must equal the combined mass flow rate from each of the 3 outflow streams – mass must be conserved!

Similarly, we can write mass conservation equations on ethane and propane also:

$$25kg/s = 0.1m_1 + 0.5m_2 + 0.2m_3 \text{(ethane)}$$

$$10kg/s = 0.0m_1 + 0.2m_2 + 0.7m_3 \text{(propane)}.$$

Note that there is no propane in stream 1, but we can still include stream 1 in the balance of propane by setting the fraction of the stream that is propane to 0.0. The result is a final system of 3 linear, algebraic equations with the same 3 unknowns. For simplicity, the system of equations can be written in matrix form as:

$$\begin{bmatrix} 0.9 & 0.3 & 0.1 \\ 0.1 & 0.5 & 0.2 \\ 0.0 & 0.2 & 0.7 \end{bmatrix} \cdot \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} 30.0 \\ 25.0 \\ 10.0 \end{bmatrix} \tag{4.2}$$

For review, let us use SymPy to solve for the flow rates of the 3 outflow streams.

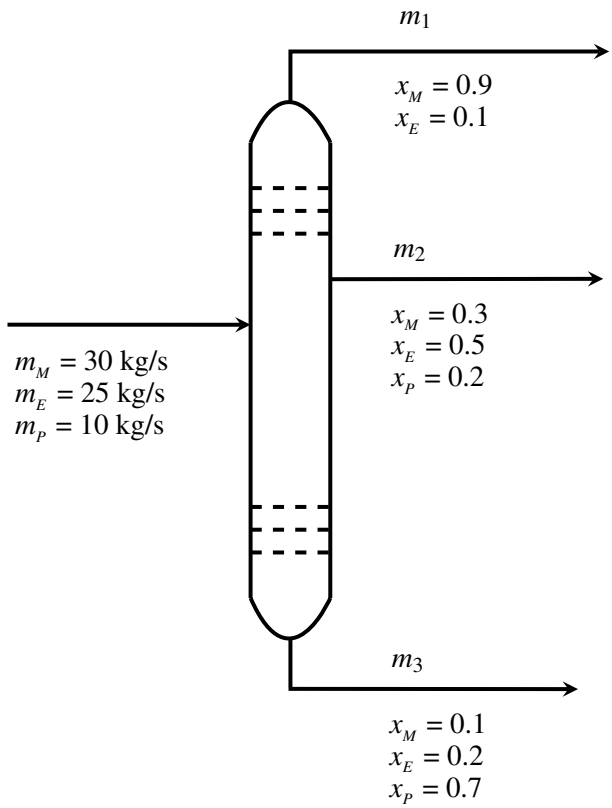


Figure 4.1: Diagram of a distillation column for the coarse separation of methane (M), ethane (E), and propane (P). The composition of the product streams is given, but the mass flow rate of each stream is unknown.

```
import sympy

m1, m2, m3 = sympy.symbols('m1 m2 m3')
eq1 = 0.9*m1 + 0.3*m2 + 0.1*m3 - 30
eq2 = 0.1*m1 + 0.5*m2 + 0.2*m3 - 25
eq3 = 0.2*m2 + 0.7*m3 - 10
print sympy.solve([eq1,eq2,eq3],[m1,m2,m3])
```

SymPy symbolically solves this small system of equations and gives a solution of $m_1 = 17.9$ kg/2, $m_2 = 46.0$ kg/s, and $m_3 = 1.2$ kg/s. It is always a good idea to check a solution to make sure that the original equations are indeed satisfied.

Instead of symbolically solving this system, which is something that does not scale well to larger systems of equations, let us instead solve the system numerically. We will use the Numpy library (`www.numpy.org`) to build the required matrices and vectors for this process. The first step is to build the matrix **A**, and right-hand-side, **f**, and store them as numpy arrays.

```
import numpy

A = numpy.array([[0.9,0.3,0.1],[0.1,0.5,0.2],[0.0,0.2,0.7]])
f = numpy.array([30.0,25.0,10.0])
print A
print f
```

Notice that the entire **A** matrix is contained in a list, and the individual rows of the matrix are sub-lists (or nested lists) within the larger list.

To solve this system of equations numerically, we need to import an additional library that contains common linear algebra functions. There are a number of linear algebra libraries available for Python, but one easy to use library is distributed with numpy and can be imported using **import** numpy.linalg. Note that this library is not automatically loaded when we **import** numpy and must be imported separately. Most people tire of typing numpy.linalg.function() over-and-over, so it is common to import this library using a shorter name such as 'nl'. This is possible using the command **import** numpy.linalg as nl. The following code using the nl.solve() function to solve the matrix problem and determine the unknown flow rates.

```
import numpy
import numpy.linalg as nl

A = numpy.array([[0.9,0.3,0.1],[0.1,0.5,0.2],[0.0,0.2,0.7]])
f = numpy.array([30.0,25.0,10.0])
x = nl.solve(A,f)
print x
```

The output is a vector containing the 3 unknown flow rates: [17.88 45.96 1.15].

The nl.solve() function computes the 'exact' solution of a well-determined linear matrix equation, $\mathbf{A} \cdot \mathbf{x} = \mathbf{f}$. The term 'exact' is in quotes because the solution is only 'exact' up to computer round-off error. In other words, the solution will typically have 8-12 digits of accuracy depending on the condition of the matrix, type of computer, and other factors. Methods that compute an 'exact' solution to a linear matrix equation are called *direct* methods. In the next section, we will examine some of the principles behind direct methods and discuss their scaling. Direct methods are a good choice for systems of 2 to 10,000 equations (although this range changes with available computational power). The computational algorithm used by numpy is a common LAPACK routine that is written in FORTRAN. Because the underlying algorithm is written in a compiled language instead of Python, it is more computationally efficient and scalable.

4.2 A Direct Solution Method

The goal of this section is to briefly examine a simple algorithm for directly calculating the 'exact' solution to a matrix problem. Even though the algorithm presented here is significantly simpler than the more complex algorithms contained in LAPACK and used by numpy, it will still be the most complex Python code written up to this point in this book. The direct solver will actually be split into

two different functions – a Gaussian elimination function and a backward substitution function. It is recommended that the reader create an empty Python file (a suggested filename is bobcatSolver.py) that contains both of the functions. This file or *module* can then be imported into other Python codes and the functions within it called using **import** bobcatSolve and then bobcatSolve.functionname().

To illustrate the process of Guassian elimination, recall the example matrix problem solved previously.

$$\begin{bmatrix} 0.9 & 0.3 & 0.1 \\ 0.1 & 0.5 & 0.2 \\ 0.0 & 0.2 & 0.7 \end{bmatrix} \cdot \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} 30.0 \\ 25.0 \\ 10.0 \end{bmatrix} \tag{4.3}$$

In the linear equation system, each row of **A** and **f** represents an equation or *equality*. It is perfectly mathematically reasonable to multiply an entire equation by a constant or add or subtract one equation from another without changing the solution. Our goal is to multiply equations by a constant and then add or subtract equations from each other so that the lower triangular part of the matrix is zero – in other words, we want a matrix that is all zero's below the main diagonal. Let's start by eliminating the value in the first column that is directly below the main diagonal – the value is currently 0.1. Observe that if we multiple the first equation (row 1) by $\frac{0.1}{0.9}$ and then subtracting the resulting equation from the second row, we will eliminate the 0.1 value in the first column and below the main diagonal. Specifically, replacing R_2 with $R_2 - R_1 \cdot \frac{0.1}{0.9}$ gives:

$$\begin{bmatrix} 0.9 & 0.3 & 0.1 \\ 0.0 & 0.4667 & 0.1888 \\ 0.0 & 0.2 & 0.7 \end{bmatrix} \cdot \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} 30.0 \\ 21.667 \\ 10.0 \end{bmatrix} \tag{4.4}$$

Notice that row 1 (representing equation 1) did not change at all; the only change was to row 2. This process can now be repeated for all non-zero terms below the main diagonal – a process called Gaussian elimination. A similar process is called LU-decomposition, which refers to the decomposition of a matrix into a lower triangular matrix (L) and an upper triangular matrix (U). The terms Gaussian elimination and LU-decomposition are frequently used interchangeably, although they are not exactly the same algorithm. For the current example problem, the result after Gaussian elimination is

$$\begin{bmatrix} 0.9 & 0.3 & 0.1 \\ 0.0 & 0.4667 & 0.1888 \\ 0.0 & 0.0 & 0.619 \end{bmatrix} \cdot \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} 30.0 \\ 21.667 \\ 0.714 \end{bmatrix} \tag{4.5}$$

Examination of the linear matrix system 4.5 shows that solving for the vector of unknowns, $[m_1, m_2, m_3]$, is now relatively trivial. Starting with the last equation, which is now $0.619m_1 = 0.714$, we can easily solve for $m_1 = 1.15$. Once m_3 is determined, it becomes trivial to solve for $m_2 = (21.667 - 0.188m_3)/0.4667$. This process of solving for the final solution after Gaussian elimination is referred to as backward substitution.

Python code for an extremely simple Gaussian elimination function is given below.

```
import numpy

def bobcatLU(A,f):
    n = f.size
```

```
    # Begin by check for compatible matrix and rhs sizes
    if (A.shape[0] != n or A.shape[1] != n):
        print "Error! Incompatible array and vector sizes."
        return f
    # Loop through the columns of the matrix
    # eliminate lower triangular part
    for i in range(0,n-1):
    # Loop through the rows below the diagonal for each column
        for j in range(i+1,n):
            if A[i,i] == 0:
                print "Zero on diagonal, need pivoting!"
                return f
            m = A[j,i]/A[i,i]
            A[j,:] = A[j,:] - m*A[i,:]
            f[j] = f[j] - m*f[i]
    return A,f
```

The function receives a matrix, **A**, and a right-hand-side, **f**, as inputs. The first few lines of code check to ensure that the matrix and right-hand-side have a compatible size. Next, a loop through the columns of the matrix (with the exception of the last column that does not have any terms below the main diagonal) is initiated. For each column, i, there is a second loop (j) through the rows below the main diagonal. For every terms below the main diagonal, the term is eliminated using equation i multiplied by the appropriate multiplier, m. After Gaussian elimination is complete, the modified matrix **A** and right-hand-side, **f** are returned to the calling function.

Backward substitution is an algorithm of similar complexity and is given below.

```
import numpy

def bobcatBS(A,f):
    n = f.size
    # Begin by check for compatible matrix and rhs sizes
    if (A.shape[0] != n or A.shape[1] != n):
        print "Error! Incompatible array and vector sizes."
        return f
    # initialize the solution vector, x, to zero
    x = numpy.zeros((n,1))
    # solve for last entry first
    x[n-1] = f[n-1]/A[n-1,n-1]
    # loop from the end to the beginning
    for i in range(n-2,-1,-1):
        sum = 0
        # for all known x values, sum and move to rhs
        for j in range(i+1,n):
            sum = sum + A[i,j]*x[j]
```

```
        x[i] = (f[i] - sum)/A[i,i]
    return x
```

The backward substitution algorithm begins by checking the dimensions of the input parameters and initializing a vector, **x**, that will ultimately hold the solution. Then, starting with the last row in the linear matrix system, the algorithm calculates the corresponding value for the **x**-vector. The algorithm proceeds from the last row to the first row before completing the substitution step.

It is simplest to combine the Gaussian elimination and backward substitution algorithms into a single file. Note that only a single **import** numpy command is required at the start of the file. The resulting file is called a *module* in Python programming, and it can be imported and used with other code. This is a very simple and efficient mechanism for recycling code. As an example, if the bobcatLU() and bobcatBS() algorithms are saved in a file called bobcatSolve.py, then the algorithms can be used to solve the previous distillation column example in a straightforward manner as illustrated in the example below.

■ **Example 4.1** Use the bobcatLU() and bobcatBS() functions to solve the distillation column example problem.

```
import numpy
import bobcatSolve as bS

A = numpy.array([[0.9, 0.3, 0.1],[0.1,0.5,0.2],[0.0,0.2,0.7]])
f = numpy.array([30.0,25.0,10.0])
A,f = bS.bobcatLU(A,f)
x = bS.bobcatBS(A,f)
print x
```

The solution should be the same as obtained using the numpy.linalg.solve() function: [17.9, 46.0, 1.2].
■

Exercise 4.1 Use the bobcatLU() and bobcatBS() functions to solve the linear matrix system:

$$\begin{bmatrix} 4.0 & -1.0 & 1.0 \\ 2.0 & 5.0 & 2.0 \\ 1.0 & 2.0 & 4.0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8.0 \\ 3.0 \\ 11.0 \end{bmatrix} \tag{4.6}$$

Further, compare the solution to the solution obtained using numpy.linalg.solve(). In both cases, the solution should be $[1.0, -1.0, 3.0]$. ■

Exercise 4.2 Use the bobcatLU Gaussian elimination algorithm on the following linear matrix

system:

$$\begin{bmatrix} 1.0 & -1.0 & 2.0 & -1.0 \\ 2.0 & -2.0 & 3.0 & -3.0 \\ 1.0 & 1.0 & 1.0 & 0 \\ 1.0 & -1.0 & 4.0 & 3.0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -8.0 \\ -20.0 \\ -2.0 \\ 4.0 \end{bmatrix} \tag{4.7}$$

The simple Gaussian elimination algorithm implemented in bobcatLU() fails with this matrix because during the elimination process, a zero is produced on the main diagonal, which in the best case triggers an error message stating that pivoting is required, and in the worst case cases a program crash.

Rewrite the bobcatLU() algorithm to use pivoting and avoid such difficulties. A description of a pivoting algorithm can be found in a number of numerical methods books including the book by Burden and Faires [BF01]. ■

4.2.1 Computational Cost

Gaussian elimination and backward substitution are much more computationally efficient than symbolic computing, but the computational scalability is still not optimal. A very rough approximation of the computational cost can be made by examining the Gaussian elimination algorithm. The elimination of all non-zero terms below the main diagonal requires looping through on the order of n-columns and n-rows. For each entry there are approximately n-multiplications so the total computational cost is on the order of n^3 operations.

To test this estimate of computational cost, the code below was used to measure the computational time for an increasing number of linear equations. The code uses the Python time library to determine the solve time by calculating the different between the start time and stop time of a calculation. The problem was based on a dense matrix of random numbers, and a random right-hand-side vector, and the smallest problem was 100 equations and the largest was 6400 equations.

```
import numpy
import numpy.linalg as nl
import time
import pylab

mag = 4
cputime = numpy.zeros((mag,1))
cpusize = numpy.zeros((mag,1))
n=100
for i in range(mag):
    print n
    A = numpy.random.random((n,n))
    b = numpy.random.random((n,1))
    start = time.clock()
    x = nl.solve(A,b)
    stop = time.clock()
    cputime[i]=stop-start
```

```
    cpusize[i]=n
    n = n*4
pylab.loglog(cpusize,cputime)
pylab.xlabel('Number of Equations')
pylab.ylabel('CPU time (sec.)')
```

The CPU time measurements versus number of equations are summarized in Figure 4.2, which was obtained on a Dell laptop with a Core i5 CPU. The smallest problem size (100 equations) required only 0.0006 sec. Assuming that the scaling of the algorithm is n^3, increasing the problem size by a factor of 4 should increase the computational time by a factor of $4^3 = 64$. The observed CPU time increase is closer to a factor of 36 when going from 100 to 400 equations, but the observed increase is exactly a factor of 64 when going from 1600 to 6400 equations. For this particular computer, 6400 equations required about 1 minute, which is why direct methods are rarely used for problems larger than approximately 10,000 equations (unless the matrix is sparse, i.e., contains mostly zeros).

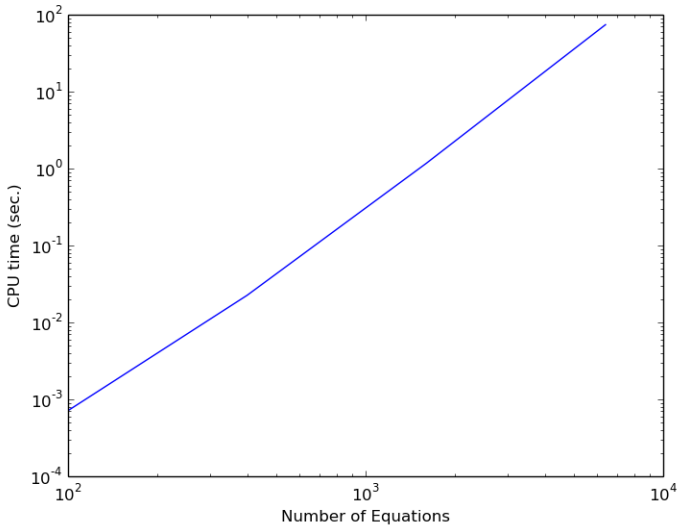


Figure 4.2: The CPU time required to solve a dense system of linear equations using numpy.linalg.solve(). If n is the number of equations, the CPU time scales with n^3.

The n^3 scaling of direct methods motivates the development of alternative approaches that give up the goal of obtaining an 'exact' solution in exchange for improved scaling. In the final section of this chapter, iterative methods that can, in some cases, improve the scaling of CPU time relative to problem size will be briefly examined.

4.3 Iterative Solution Methods

The basic motivation behind iterative methods is the observation that the computational cost of multiplying a matrix and vector is on the order of n^2-multiplications for a dense matrix, and on the order or n-multiplications for a matrix that is sparse (i.e., a matrix that contains mostly zeros).

If we have a guess at the solution vector, which we can refer to as $\mathbf{x}_0$, then it is computationally inexpensive to calculate what is referred to as the residual: $\mathbf{res} = \mathbf{f} - \mathbf{A} \cdot \mathbf{x}_0$. Notice that the residual is measure of how close our guess, $\mathbf{x}_0$, is to satisfying the original matrix problem, $\mathbf{A} \cdot \mathbf{x} = \mathbf{f}$. If the values in the residual vector are 'small', our guess is close to the solution. In order to define 'small' in more specific terms, we need to briefly discuss norms.

4.3.1 Vector Norms

A norm is a single number that reflects the size of a vector. The most commonly used norm is the L^2-norm or Euclidean norm and it is calculated as:

$$|x| = \sqrt{\sum_{k=1}^{n} (x_k)^2} \, .$$

The L^2-norm of a Numpy vector, $\mathbf{x}$, is calculated using numpy.norm(x,2), where the '2' denotes the L^2-norm. One other norm of notable relevance to the calculations of interest in this book is the infinity norm, which is calculated by finding the term in a vector with the largest absolute value. The infinity norm of a Numpy vector is calculated using numpy.norm(x,numpy.inf).

4.3.2 Jacobi Iteration

To illustrate our first iterative method, let us return to the system of equations that represent mass balances around a distillation column.

$$30kg/s = 0.9m_1 + 0.3m_2 + 0.1m_3$$

$$25kg/s = 0.1m_1 + 0.5m_2 + 0.2m_3$$

$$10kg/s = 0.0m_1 + 0.2m_2 + 0.7m_3$$

One approach to determine $m_1 - m_3$ would be to make an initial guess, for example, that $m_1 = 20kg/s$, $m_2 = 20kg/s$, and $m_3 = 20kg/s$. This is not the most reasonable of guesses since total mass is not conserved (i.e., more mass is flowing into the column than out based on our crude guess), but the goal is to illustrate that our guess does not necessarily need to be *really* close to the actual solution. Now, let us solve the first equation for m_1, using our guess for the values of m_2 and m_3. It is trivial to calculate a new guess for $m_{1,new} = (30 - 0.3 \cdot 20 - 0.1 \cdot 20)/0.9 = 24.44$. Repeating this process and solving for a new guess for m_2 using the second equation and the old guess for both m_1 and m_3 results in $m_{2,new} = 38$ and, finally, $m_{3,new} = 8.6$ using the third equation. Notice that the new guess is indeed closer to the solution determined previously in this chapter than our initial guess of 20 kg/s for every stream. If we repeated this process a 'few' more times, always using our improved guess, we might converge towards the 'exact' solution.

The Python script below will help us to perform these calculations quickly and automatically (and we will learn some new Python programming practices as well!).

```
import numpy
import numpy.linalg as nl

def jacobi(A,f,x,maxIter = 100, tol = 1.0e-4):
```

```
    #  inputs:
    #  A is a nxn matrix
    #  f is a right-hand-side vector of length n
    #  x is initial guess at the solution to A x = f
    #  maxIter (optional) is maximum number of Jacobi iterations
    #  tol (optional) is the desired accuracy in terms of the
    #  L2-norm of the residual (= f - Ax)
    n = f.size
    # Begin by checking for compatible matrix and rhs sizes
    if (A.shape[0] != n or A.shape[1] != n):
        print "Error! Incompatible array and vector sizes."
        return f
    # Set up a loop to iterate until we converge to an solution
    # or we reach the maximum number of iterations
    xnew = numpy.copy(x)
    for iter in range(maxIter):
        # calculate residual
        res = f - numpy.dot(A,x)
        # check L2-norm of the residual for convergence
        if (nl.norm(res,2) < tol):
            print 'Converged after ', iter, ' iterations'
            return x
        # start of Jacobi iteration
        for i in range(n):
            sum=0.0
            for j in range(n):
                if(i != j):
                    sum += A[i,j]*x[j]
            xnew[i] = (f[i] - sum)/A[i,i]
        x = numpy.copy(xnew)
    print 'Failed to converge after ', iter, ' iterations'
    return x

A = numpy.array([[0.9, 0.3, 0.1],[0.1,0.5,0.2],[0.0,0.2,0.7]])
f = numpy.array([30.0,25.0,10.0])
x = numpy.array([20.0, 20.0, 20.0])
sol = jacobi(A,f,x)
print sol
```

First, there are two new Python programming techniques used here that have not been covered previously. The code begins with the definition of a function, called jacobi, but, it is important to emphasize that when we execute or run this code, execution actually begins with the line that constructs the matrix **A**. The function definition is read by Python and stored for later use, but the function is not executed until is it called in the second to last line of the script. The function definition must appear before the function is first called because, otherwise, Python will return an error stating

the function has not been defined when it is first called. The second new Python program technique involves the arguments or variables that are passed into the function when it is called. The function itself requires that at least 3 variables be passed into the function, a matrix, a right-hand-side vector, and a vector containing a guess at the solution. However, the function allows two additional, optional argument to be passed when it is called. The first optional argument is the maximum number of Jacobi iterations, and the default value is set to 1000 if another value is not passed into the function. The second optional argument is the desired tolerance, i.e., the desired L^2-norm of the residual vector. The jacobi function iterates until either the maximum number of iterations is reached or the desired tolerance is achieved, whichever is reached first. A helpful comment at the top of the function reminds the user of the input variable requirements.

The lines of Python code for the first half of the function are largely comments or code that we have used before. An iteration loop is initiated to run for at most the maximum number of iterations allowed, and then the residual vector and its L^2-norm are calculated. Before performing the calculations associated with the Jacobi iteration, the L^2-norm of the residual is always calculated to test for convergence. If the norm is less than the desired tolerance, the current guess is returned and the function execution ends. If the norm is not less than the tolerance, the coefficients in $\mathbf{A}$ are multiplied by the current guess at the solution (except for the coefficient on the diagonal associated with the unknown we are determining) and these are subtracted from the right-hand-side and divided by the coefficient along the diagonal. The reader is encouraged to revisit the process described above for solving for one unknown for each of the mass balance equations and to observe the connection to the Jacobi iteration in the Python script. If the desired solution tolerance is not achieved after the maximum number of iterations has been reached, the function prints an error message and simply returns the (incorrect) vector $\mathbf{x}$ after the final iteration.

Testing the Jacobi iterative method on the distillation column mass balances results in 14 iterations being required to achieve the default tolerance for the L^2-norm of the residual. It is interesting to test the method with different initial guesses for the solution. For example, if our initial guess had been x = numpy.array([10.0, 10.0, 10.0]), the algorithm would have required 16 iterations to achieve a solution satisfying the same tolerance.

Exercise 4.3 Test the Jacobi iterative method on the following matrix and right-hand-side:

$$\begin{bmatrix} 1.0 & -1.0 & 2.0 & -1.0 \\ 2.0 & -2.0 & 3.0 & -3.0 \\ 1.0 & 1.0 & 1.0 & 0 \\ 1.0 & -1.0 & 4.0 & 3.0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -8.0 \\ -20.0 \\ -2.0 \\ 4.0 \end{bmatrix} \tag{4.8}$$

The desired solution is $[-7.0, 3.0, 2.0, 2.0]$, but for almost any initial guess (other than the exact solution), the Jacobi iteration fails to converge. ■

4.3.3 Gauss-Seidel Iteration

The Jacobi iteration calculates a new guess for the vector $\mathbf{x}$ based *only* on the previous guess. It is therefore possible to compute each entry in the new guess vector simultaneously. An obvious alternative to this approach is to calculate a new value for the first entry in the unknown vector $\mathbf{x}$, but then use this new value for calculating the second entry in the vector $\mathbf{x}$. Continuing in this manner, each new value in $\mathbf{x}$ is always calculated using the most recent information available. For the first

iteration of the distillation column example, the calculation of $m_1 = 24.4$ would be identical to the Jacobi iteration, but the calculation of m_2 using the new value for m_1 would result in $m_2 = 37.11$ instead of $m_2 = 38$.

The implementation of the Gauss-Seidel iteration is nearly identical to the implementation of the Jacobi iteration, except the vector xnew is no longer required since all calculations involve only the most recent information that is already stored in x. The Python script that implements the Gauss-Seidel iteration for the distillation column mass balances is given below.

```
import numpy
import numpy.linalg as nl

def gaussSeidel(A,f,x,maxIter = 100, tol = 1.0e-4):
    # inputs:
    # A is a nxn matrix
    # f is a right-hand-side vector of length n
    # x is initial guess at the solution to A x = f
    # maxIter (optional) is maximum number of Jacobi iterations
    # tol (optional) is the desired accuracy in terms of the
    # L2-norm of the residual (= f - Ax)
    n = f.size
    # Begin by checking for compatible matrix and rhs sizes
    if (A.shape[0] != n or A.shape[1] != n):
        print "Error! Incompatible array and vector sizes."
        return f
    # Set up a loop to iterate until we converge to an solution
    # or we reach the maximum number of iterations
    for iter in range(maxIter):
        # calculate residual
        res = f - numpy.dot(A,x).flatten()
        # check L2-norm of the residual for convergence
        if (nl.norm(res,2) < tol):
            print 'Converged after ', iter, ' iterations'
            return x
        # start of Gauss-Seidel iteration
        for i in range(n):
            sum=0.0
            for j in range(n):
                if(i != j):
                    sum += A[i,j]*x[j]
            x[i] = (f[i] - sum)/A[i,i]
    print 'Failed to converge after ', iter, ' iterations'
    return x

A = numpy.array([[0.9, 0.3, 0.1],[0.1,0.5,0.2],[0.0,0.2,0.7]])
```

```
f = numpy.array([30.0,25.0,10.0])
x = numpy.array([20.0, 20.0, 20.0])
sol = gaussSeidel(A,f,x)
print sol
```

Applying the Gauss-Seidel iteration to the test problem results in 8 iterations being required for convergence to the approximate solution with the same default tolerance used previously (recall that 10 Jacobi iterations were required). The Gauss-Seidel iterative method typically converges with significantly fewer iterations and over a greater range of initial guesses than the Jacobi iterative method. The only reason to use Jacobi iterations instead of Gauss-Seidel iterations is that calculations involved in the Jacobi iteration may be executed in parallel, which may result in shorter computational times on some computer architectures even with a larger total number of iterations. Before closing this section on iterative methods, it is helpful to look at the factors that impact whether or not these methods converge and the rate of convergence.

4.3.4 Convergence of Iterative Methods

The goal of the iterative methods described here is to determine an approximate solution to the problem $\mathbf{A}\cdot\mathbf{x}=\mathbf{f}$. To examine the implications of the differences between the methods described previously, it is useful to add and subtract $\mathbf{I}\cdot\mathbf{x}$ from the right-hand-side (note that $\mathbf{I}$ is the identity matrix, which is the same size as $\mathbf{A}$ but just has ones on the diagonal and zeros everywhere else), giving: $\mathbf{I}\cdot\mathbf{x}+\mathbf{A}\cdot\mathbf{x}-\mathbf{I}\cdot\mathbf{x}=\mathbf{f}$. This equation can be rearrange to give a potential iterative method: $\mathbf{x_{new}}=\mathbf{f}-\mathbf{A}-\mathbf{I}\cdot\mathbf{x_{old}}$. It turns out that this is a really slow iterative method that should never be used.

We can use this same framework to write down the Jacobi iteration. We begin by decomposing $\mathbf{A}$ into a matrix $\mathbf{D}$ that just has the main diagonal terms from $\mathbf{A}$ with zero everywhere else, a matrix $\mathbf{L}$ that just has the values from $\mathbf{A}$ that are in the lower triangular section strictly below the main diagonal, and a matrix $\mathbf{U}$ that contains the values from $\mathbf{A}$ that are above the main diagonal. With these new matrices, we can rewrite $\mathbf{A}=\mathbf{D}+\mathbf{L}+\mathbf{U}$. An example of this decomposition for our distillation example is:

$$\begin{bmatrix} 0.9 & 0.3 & 0.1 \\ 0.1 & 0.5 & 0.2 \\ 0.0 & 0.2 & 0.7 \end{bmatrix} = \begin{bmatrix} 0.9 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.0 \\ 0.0 & 0.0 & 0.7 \end{bmatrix} + \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.1 & 0.0 & 0.0 \\ 0.0 & 0.2 & 0.0 \end{bmatrix} + \begin{bmatrix} 0.0 & 0.3 & 0.1 \\ 0.0 & 0.0 & 0.2 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \quad (4.9)$$

Recalling that in the Jacobi iteration, all the off diagonal terms in $\mathbf{A}$ were effectively moved to the right side of the equation and we then divided by the diagonal terms of $\mathbf{A}$, the Jacobi iteration can be written as:

$$\mathbf{x_{new}}=\mathbf{D^{-1}}(\mathbf{f}-(\mathbf{L}+\mathbf{U})\cdot\mathbf{x_{old}})\ .$$

Using the same strategy, the Gauss-Seidel iteration can be written as:

$$\mathbf{x_{new}}=(\mathbf{D}+\mathbf{L})^{-1}(\mathbf{f}-\mathbf{U}\cdot\mathbf{x_{old}})\ .$$

In either case, we are required to calculate the inverse of a matrix ($\mathbf{D^{-1}}$ or $(\mathbf{D}+\mathbf{L})^{-1}$), which is normally the same computational cost as Gauss Eliminate (i.e., order n^3 operations) but is very inexpensive for the two matrices listed here because they are strictly diagonal or lower triangular

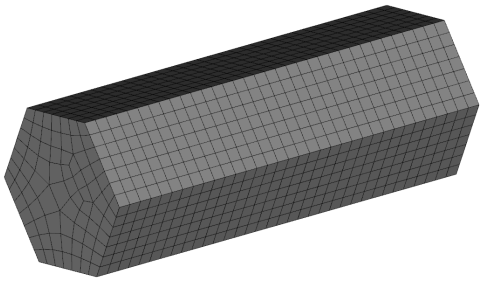
(i.e., the same cost as backward substitution, order n^2 or less). As a result, each iteration is relatively inexpensive from a computational standpoint.

The rate at which these iterative methods converge depends how well the *preconditioner*: $\mathbf{D^{-1}}$ or $(\mathbf{D}+\mathbf{L})^{-\mathbf{1}}$) for Jacobi and Gauss-Seidel, respectively, approximates $\mathbf{A^{-1}}$. If the diagonal matrix, $\mathbf{D}$ contains the largest terms in $\mathbf{A}$, then $\mathbf{D^{-1}}$ is a good preconditioner and convergence is rapid. If the largest magnitude terms are not along the main diagonal, i.e., not contained in $\mathbf{D}$, then it is a poor preconditioner and convergence is unlikely.

The field of iterative methods for linear equation system is very broad, and the presentation of convergence rate here is very simplified. There are a number of iterative method such as the conjugate gradient method, Krylov methods, and multigrid methods that are beyond the scope of this book. However, in all cases, the availability or absence of a good and inexpensive preconditioner has a significant effect. Interestingly, one common precondition is to use Gauss Elimination but to throw away any small terms that arise during the computations. This is referred to as incomplete elimination and it provides a robust and inexpensive preconditioner for some problems. For more information on iterative methods, the interested reader is referred to the books:

- Numerical Analysis by Burden and Faires [BF01]
- Iterative Methods for Solving Linear Systems by Greenbaum [Gre97]
- A Multigrid Tutorial by Briggs, Henson, and McCormick [BHM00]

Motivation
Fitting Vapor Pressure Data
Linear Regression
Alternative derivation of the normal equations
Nonlinear Regression



5. Regression

5.1 Motivation

Frequently in Chemical or Biological Engineering, a series of measurements are taken while one or more parameters for a system are varied. In some case, such as the sample data shown on the left of figure 5.1, the measured system parameter (also called the dependent variable) varies linearly with the parameter that is being systematically varied. For example, if the pressure of a rigid tank containing an ideal gas is measure while the temperature is changing, the relationship will be linear as long as the gas behaves like an ideal gas. In this example, pressure is the dependent variable and temperature is the independent (controlled) variable. The standard practice is to plot the data with the dependent variable on the 'y-axis' and the independent variable on the 'x-axis'. The other possible case, which is much more common in real world, is that the measured variable changes nonlinearly as the controlled parameter is being varied. An example of this occurs if we measure the vapor pressure of water while varying the temperature of the liquid. Example data of such an experiment is shown on the right of figure 5.1. The careful observer will note that in the event nonlinear behavior is observed, one can simply reduce the range over which the independent variable is changed to recover a small region where the relationship is approximately linear. All curves are approximately linear if you zoom in far enough.

Data tends to be much more useful if we are able to obtain a mathematical expression that approximately matches or 'fits' the data. If only two data points are available, then it is trivial to determine the equation for the line that passes through those two points. (It is extremely dangerous to do this because we typically do not know if additional data will be linear and fall close to this line.) If more than two data points are available, then all three points are extremely unlikely to fall on the same line so we need to develop an approach for obtaining the equation for a line that 'fits' the data.

This chapter covers linear regression in some detail, including the mathematics used to find the equation for a line that minimizes the square of the distance between the data the linear regression line. Polynomial regression, where data is fit with a polynomial of a user specified degree is also covered using tools available in Scipy. However, before getting into the mathematics and programming, let

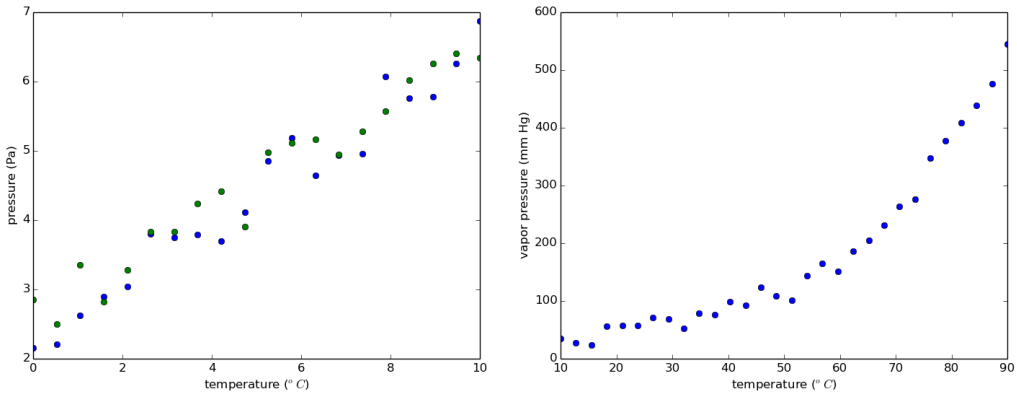


Figure 5.1: Example of two different data sets that might require regression to fit the data with either a line (left) or some other nonlinear function (right). The data on the left has two colors representing to different sets of data – possibly from two separate experiments.

us examine a common problem in vapor-liquid equilibria that requires linear regression.

5.2 Fitting Vapor Pressure Data

Vapor pressure is the pressure exerted by a pure liquid into the vapor phase. In other words, it is a measure of the volatility of the liquid, i.e., a measure of how badly the liquid wants to be a vapor. If the vapor pressure reaches or exceeds the total pressure in the vapor phase, the liquid boils. If the vapor pressure of water in a pot on the stove exceeds approximately 760 mm Hg, i.e., atmospheric pressure, then the water boils. The vapor pressure of water at 100^oC is, of course, 760 mm Hg. The right panel of figure 5.1 shows a typical set of vapor pressure measurements for water over a range of temperatures.

According to the Clausius-Clayperon equation, the relationship between vapor pressure, p^*, and temperature, T is an equation of the form:

$$\ln(p^*) = c_1\left(\frac{1}{T}\right) + c_2$$

where c_1 and c_2 are constant that must be determined from experimental measurements. Looking at the data show on the right in figure 5.1 it may not be apparent that we could ever use linear regression to fit the data, but the Clausius-Clapeyron relationship shows us the way. Recalling that the equation for a line is

$$y = c_1 \cdot x + c_2 \ ,$$

we notice that if we plot $\ln(p^*)$ (instead of just p^*) and if we plot $\frac{1}{T}$ instead of T, the data should be approximately linear allowing us to fit the data with a line and determine c_1 and c_2. When vapor pressure data, such as that shown on the right in figure 5.1, is replotted after taking the natural log of the vapor pressure and the inverse of temperature, the data falls on an approximately straight line as shown in figure 5.2. Once the constants are known, we can use the Clausius-Clapeyron relationship to determine the vapor pressure at any temperature we desire, or, equivalently, determine the temperature at which a desired vapor pressure can be achieved.

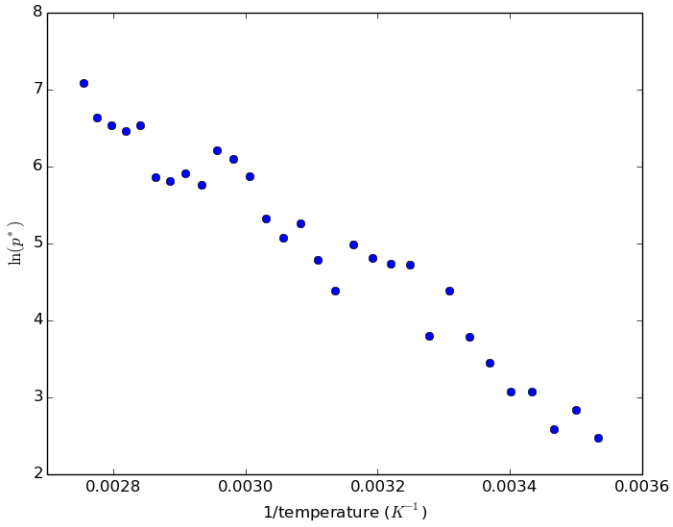


Figure 5.2: When the natural log of vapor pressure, $\ln(p^*)$ is plotted against the inverse temperature, $1/T$, then the points fall on a straight line and linear regression can be used to fit the data.

5.3 Linear Regression

To begin the regression process, we need data. For now, assume that we have n data points that consist of one independent variable and one dependent variable: (x_i, y_i), $i = 1, \ldots, n$. The goal is to obtain an equation for a polynomial, $y = p(x)$, that approximately matches the data. The first step is the selection of a measure that tells us how well the polynomial matches the data. For example, we may want to minimize the absolution value of the distance between the polynomial, $p(x)$, and the data, y_i. If this is the goal, then we need to minimize:

$$E_{abs} = \sum_{i=1}^{n} |y_i - p(x_i)| \ .$$

In practice, it is difficult to minimize E because the derivative of the absolute value function is not defined at the origin. Least-squares regression is an easier and much more common choice. It is based on minimizing:

$$E_{ls} = \sum_{i=1}^{n} (y_i - p(x_i))^2 \ .$$

In order to understand the process of minimizing E_{ls}, let us assume for simplicity that we are interested in linear polynomials, i.e., $p(x_i) = c_1 x_i + c_0$. It is relatively straightforward to extend this analysis to higher-order polynomials, but this is often unnecessary because helpful software has been developed to automate the process.

The goal of linear regression analysis is to determine the two unknowns in the equation for the line: c_0 and c_1 that minimize E_{ls}. As taught in first semester calculus courses, the local minimum of a function occurs when the derivative is equal to zero. Further, since E_{ls} is a quadratic function, the local minimum is also the global minimum. Taking the derivative of E_{ls} with respect to the two

unknowns gives:

$$\frac{\partial E_{ls}}{\partial c_0} = -2\sum_{i=1}^{n}(y_i - c_1 x_i - c_0) = 0\ ,$$

$$\frac{\partial E_{ls}}{\partial c_1} = -2\sum_{i=1}^{n}((y_i - c_1 x_i - c_0)\cdot x_i) = 0\ .$$

Simplifying these equations (note that a constant can be factored out of a sum) gives:

$$\sum_{i=1}^{n}(y_i) = c_1\sum_{i=1}^{n}(x_i) - n\cdot c_0\ ,$$

$$\sum_{i=1}^{n}(y_i x_i) = c_1\sum_{i=1}^{n}(x_i^2) - c_0\sum_{i=1}^{n}(x_i)\ ,$$

which are commonly referred to as the normal equations. Recall that x_i and y_i are *known* or given by the data. The only unknowns are c_0 and c_1. In terms of these unknowns, the normal equations are a linear system of equations and can be rewritten as a matrix problem:

$$\begin{bmatrix} n & \sum_{i=1}^{n}(x_i) \\ \sum_{i=1}^{n}(x_i) & \sum_{i=1}^{n}(x_i^2) \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n}(y_i) \\ \sum_{i=1}^{n}(x_i y_i) \end{bmatrix} \tag{5.1}$$

Fortunately, we learned how to solve linear matrix problems in the previous chapter.

A Python script that performs linear regression on eight data points, which are given near the start of the script, is shown below.

```
import numpy
import numpy.linalg as nl
import pylab

n = 8
x = numpy.array([ 0.1,1.43,2.86,4.29,5.71,7.14,8.57,9.95 ])
y = numpy.array([ 2.33,2.81,3.84,4.41,4.31,5.65,5.68,6.80 ])
if ((n!=x.size) or (n != y.size)):
    print "Error: inconsistent number of data points!"
    print "Results will not be valid."

Xsum = numpy.sum(x)
Ysum = numpy.sum(y)
XYsum = numpy.sum(numpy.dot(x,y))
X2sum = numpy.sum(numpy.dot(x,x))
A = numpy.array([[n, Xsum], [Xsum, X2sum]])
f = numpy.array([Ysum, XYsum])
c = nl.solve(A, f)

yLR = c[0] + c[1]*x
pylab.plot(x,y,'o',x,yLR)
pylab.xlabel('x')
```

```
pylab.ylabel('y')
```

The script begins by constructing two numpy arrays that contain the data. One array, **x**, contains the independent variable and the other array, **y**, contains the dependent variable data. The script then checks to make sure that the arrays are the same size. A useful numpy function, numpy.**sum**() is used to sum the entries in a vector. These sums are stored in different variables for use later in constructing the matrix, **A**, and right-hand-side, **f**. The linear system is solved using the solver included with numpy. In order to plot the regression linear, the solution to the linear problem, which contains the coefficients for the linear polynomial fit to the data, is used to calculate the approximate value of the dependent variable at every independent variable point. The original data for the example problem is shown on the left side of figure 5.3, and the data with the linear regression curve is shown on the right side. Plotting data as separate points and polynomial fits to the data as a solid line is standard practice and highly recommended.

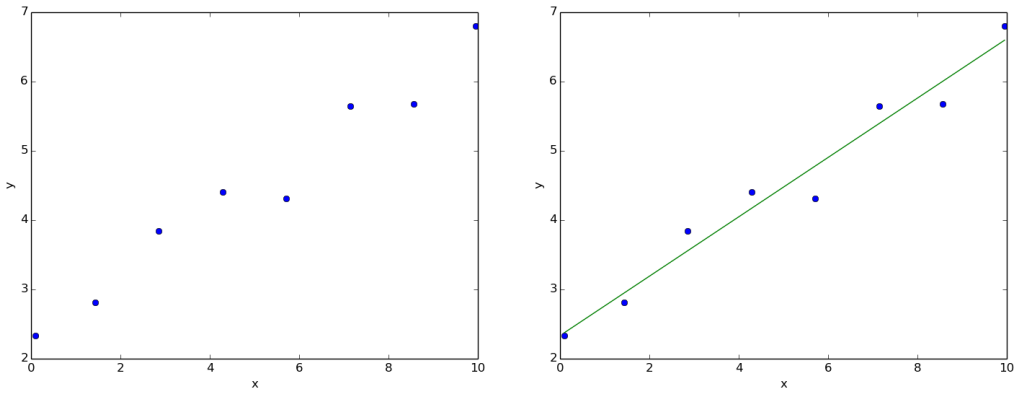


Figure 5.3: Sample data used in regression analysis is shown on the left, and the same data with the calculated regression line is shown on the right.

The data for the previous example was obtained by first selecting evenly distributed points from a straight line: $y = 0.4x + 2.5$, and then adding random noise to the data so that it did not all fall on a straight line. Interestingly, the regression line obtained using least-squares linear regression was $y = 0.4285x + 2.33$. Because only 8 points were used and the random noise was not evenly distributed over such a small number of points, the original line was not completely recovered. Additional testing showed that the slope could be reliably recovered (approximately 2 significant digits) with about 20 data points, but recovering an accurate value for the intercept was more difficult and required at least 100 data points to obtain approximately 2 significant digits.

Exercise 5.1 Repeat the derivation of the normal equations and linear matrix problem associated with the least-squares regression of a quadratic polynomial: $y_i = c_2 x_i^2 + c_1 x_i + c_0$. The final result should be:

$$\begin{bmatrix} n & \sum_{i=1}^{n}(x_i) & \sum_{i=1}^{n}(x_i^2) \\ \sum_{i=1}^{n}(x_i) & \sum_{i=1}^{n}(x_i^2) & \sum_{i=1}^{n}(x_i^3) \\ \sum_{i=1}^{n}(x_i^2) & \sum_{i=1}^{n}(x_i^3) & \sum_{i=1}^{n}(x_i^4) \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n}(y_i) \\ \sum_{i=1}^{n}(x_i y_i) \\ \sum_{i=1}^{n}(x_i^2 y_i) \end{bmatrix} \tag{5.2}$$

■

5.3.1 Alternative derivation of the normal equations

Recall that the goal in regression analysis is to determine the polynomial coefficients give the optimal fit to n data points. This requires that the number of data points be greater than the number of coefficients in the polynomial. The polynomial will have the form: $y_i = c_0 + c_1 x_i + c_2 x_i^2 + \ldots$. Since $i = 1, \ldots, n$, we can think of this polynomial at each data point being written as a linear matrix problem: $\mathbf{Ac} = \mathbf{y}$ where

$$\mathbf{A} = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots \\ 1 & x_2 & x_2^2 & \ldots \\ \vdots & & \ddots & \\ 1 & x_n & x_n^2 & \ldots \end{bmatrix} \tag{5.3}$$

Notice that $\mathbf{A}$ and $\mathbf{y}$ are both given by the data that is being fit. In regression, the goal is to minimize:

$$E_{ls} = ||\mathbf{y} - \mathbf{Ac}||^2 ,$$

which is the L^2-norm of the residual squared. E_{ls} is minimized when the derivative is zero, which leads to:

$$\frac{\partial E_{ls}}{\partial c} = 2||\mathbf{y} - \mathbf{Ac}||\mathbf{A} = 0 .$$

Upon rearrangement [AR05], we obtain the normal equations in matrix form:

$$\left(\mathbf{A^T A}\right)\mathbf{c} = \mathbf{A^T y} .$$

The result is the same linear matrix problem that was derived previously using the more common approach.

5.4 Nonlinear Regression

The fitting of data by a nonlinear function through a least-squares minimization process can be difficult because the process leads to a system of nonlinear equations that must be solved. The solving of nonlinear equations is the focus of the next chapter, but a few words on the process are important here. First, nonlinear regression requires that the user supply a 'guess' for the values of the unknown parameters in the function that is being fit to the data. If data is being fit to a function like $a\sin(\pi x/p)$, where x is the independent variable, then the user must supply an initial guess for the amplitude, a and the period, p, of the data. The nonlinear solution process tends to be much more likely to converge to a function that optimally fits the data in a least-squares norm if the initial guesses for the function parameters are relatively close. The second factor that impacts the solution process is the accuracy of the data. Data with significant levels of noise is very unlikely to lead to a nonlinear regression solution for highly nonlinear functions.

A relatively robust nonlinear least-squares regression routine is included with the Scipy library, it is the `scipy.optimize.curve_fit()` function. This function is based on the Levenberg-Marquardt algorithm for nonlinear regression. The call to this function has the form:
`scipy.optimize.curve_fit(func, xdata, ydata, p0, sigma)`, and the following parameters are passed into this function when it is called:

func : This is a user defined function that is declared before curve fit is called. The function must take the independent data as input (i.e., xdata), and the values for the different unknown parameters that are being determined in the fitting process. The function must return a vector, **y**, containing the dependent variable. The curve_fit function changes the values of the unknown parameters to minimize the difference between the function return values and the dependent variable data, ydata.

xdata : a vector of the independent variable data points.

ydata : a vector of the dependent variable data points.

p0 : a Python list containing guesses for the unknown function parameters. A default value of 1 is used for all parameters if no initial guess is provided. Use of the default value is discouraged.

sigma : an *optional* vector that is used to provide relative weights for the least-squares processes. If the goal is to fit some of the data points more closely than other data, a larger weight can be applied to those data points.

The use of the curve_fit () function is illustrated through the Python script below.

```
import numpy
from scipy.optimize import curve_fit
import pylab

n = 20
# Antoine coefficients for water from Wikipedia
A = 8.07131
B = 1730.63
C = 233.426

# Build some fake data temperature, x, versus
# vapor pressure, y, data
x = numpy.linspace(20,90,num=n)
error = numpy.random.rand(n)
y = numpy.zeros(n)
for i in range(n):
    y[i] = A - B/(x[i]+C)
    y[i] = (10**y[i])+50*(error[i]-0.5)

# Function for Antoine's equation -- used in call
# to curve fit below.
def antoine(temp, a, b, c):
    n = temp.size
    p = numpy.zeros(n)
    for i in range(n):
        p[i] = 10**(a - b/(temp[i]+c))
    return p

# Guesses for the Antoine coefficients and curve_fit call
```

```
params = [10, 2000, 200]
popt, pcov = curve_fit(antoine, x, y, p0=params)

# calculate the dependent variable for plotting the curve
yfit = antoine(x, popt[0], popt[1], popt[2])
# plot data as points and fit as a line
pylab.plot(x,y,'o',x, yfit)
pylab.xlabel('temperature (°C)')
pylab.ylabel('p* (mm Hg)')
```

This function begins by generating some 'fake' data that is curve fit later in the script. Random noise is added to the fake vapor pressure data. The nonlinear function with unknown parameters is defined in the function antoine(temp,a,b,c). This function is passed a vector of temperatures and estimates for the three unknown parameters: a, b, and c. The function returns the vapor pressure based on the estimated parameters. The curve_fit() function repeatedly calls antoine() with different estimates for the parameters in an effort to better fit the dependent variable data. At the end of the script, the original fake data and the curve_fit() result, if one is obtained, are plotted.

A sample result using the script above is shown in figure 5.4. Repeated experimentation with the script and different sets of fake data provided some interesting observations. First, the Antoine parameters from curve_fit() often differed from the actual parameters by 10% or more whenever significant quantities of noise were added to the fake data. Second, the curve_fit() function often failed to converge to a least-squares minimizer if the error level in the data exceeded roughly 10%. Adding small amounts (0.1%) of noise to the data gave accurate values for the Antoine parameters and always led to convergence by curve_fit(), but larger amounts of noise gave poor results or no results. Finally, large quantities of data (50 or more points) gave better results than small quantities of data (10 or fewer points).

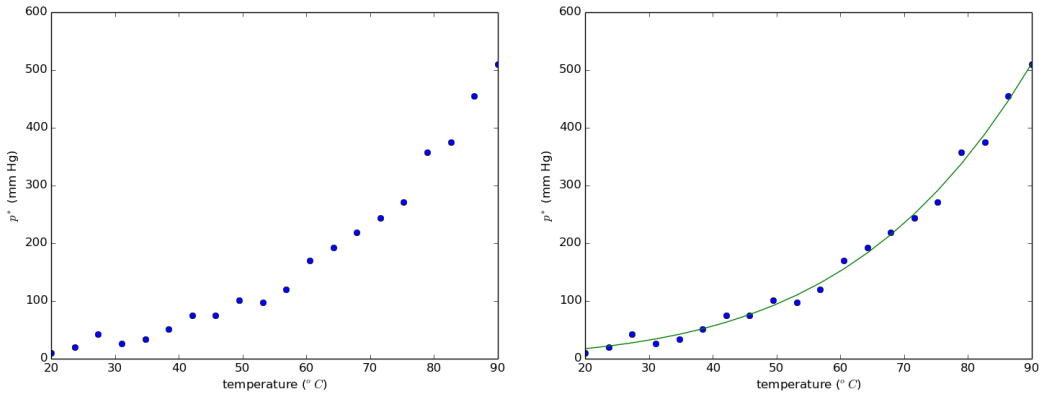


Figure 5.4: Sample data used in nonlinear regression analysis is shown on the left, and the same data with the nonlinear regression line from curve_fit() shown on the right. $p^* = 10^{8.1-1719/(232+T)}$

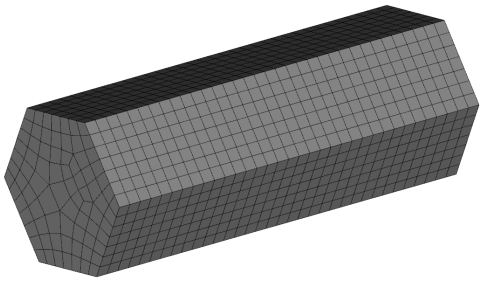
Exercise 5.2 Look up the average monthly high temperature for Bozeman, Montana. Starting with April, list the average monthly high temperature for the next 12 months. Then, fit this data

with a function of the form:

$$a + \sin(\pi x/p) + b$$

where x is the month (starting with April as zero and continuing to March, month 11) and a, b, and p are unknown parameterms that are to be determined to fit the average monthly high in a least-squares minimization. ■

Introduction
Bisection Method
Newton's Method
Broyden's Method
Multiple Nonlinear Equations



6. Nonlinear Equations

6.1 Introduction

Nonlinear equations are frequently encountered in diverse areas of engineering, including chemical reaction rates, phase equilibria, fluid distribution systems, material deformation at large strain, etc. One important nonlinear equation that will be used as an example problem in this chapter is the Soave-Redlich-Kwong (SRK) nonideal equation of state. This equation relates the pressure and temperature of a gas to its specific volume (i.e., the volume per mole of material). The SRK equation of state can be written

$$P = \frac{RT}{\hat{V} - b} - \frac{\alpha a}{\hat{V}(\hat{V} + b}$$

where P is the absolute pressure, T is the absolute temperature, $\hat{V}$ is the specific volume, R is the gas constant, and the remaining parameters are defined as

$$a = 0.42747 \frac{(RT_c)^2}{P_c}$$

$$b = 0.08664 \frac{RT_c}{P_c}$$

$$m = 0.48508 + 1.55171\omega - 0.1561\omega^2$$

$$\alpha = \left[1 + m\left(1 - \sqrt{T/T_c}\right)\right]^2$$

where T_c and P_c are the critical temperature and pressure for the substance of interest.

Equations of state relate T, P, and $\hat{V}$ for a given substance and whenever two of the three parameters are known, the EOS can be used to determine the third, unknown parameter. If T and $\hat{V}$ are given for a known substance, then it is trivial to calculate P after looking up T_c, P_c, and ω for that substance. However, the most common situation is that T and P are known, and we wish to calculate $\hat{V}$. This is a much more difficult problem, requiring the solution to a cubic equation. Our goal in

the first half of this chapter is to examine two different methods for finding solutions to nonlinear, algebraic equations like the SRK equation when $\hat{V}$ is unknown. The first method, bisection, is slow but very robust while the second method, Newton's method, is fast but does not always converge to the solution.

Before examining bisection and Newton's method in detail, we can use some functions included with the Scipy library to solve the SRK problem described above. It is, of course, dangerous to use an algorithm that we do not understand – so please consider this approach with skepticism, but having a solution to our example problem will be helpful going forward. The Scipy library includes a number of functions for solving nonlinear equations and these functions are contained in the optimization section of the library. Some of the nonlinear solvers are designed for a single nonlinear equations and other solvers, examined later in this chapter, are designed for large systems of nonlinear equations. Here we will use the 'broyden1' function, (scipy.optimize.broyden1() to solve the SRK equation for $\hat{V}$. The use of the broyden1 function (and all other nonlinear solvers) requires at least two inputs: (1) the name of a function containing the nonlinear equation, and (2) a guess for the solution. Further, the nonlinear function must be rearranged so that all terms are on one side of the equal sign and it returns zero at the solution. Hence, if we have a general nonlinear equation like the SRK equation that can be written $f(x) = g(x)$ and we are searching for x that satisfies this equality, we must rewrite the function as $f(x) - g(x) = 0$ so that the function returns zero when x is found.

> *The nonlinear function must be rearranged so that it is equal to zero at the solution.*

The SRK equation should be rewritten as

$$P - \frac{RT}{\hat{V}-b} + \frac{\alpha a}{\hat{V}(\hat{V}+b} = 0\,.$$

The Python script below begins with the definition of the SRK function, which is passed a guess for $\hat{V}$ as the only input, and then using that guess and the appropriate constants for the material of interest (carbon monoxide in this case), tests to see if the SRK equation is satisfied. If the function returns zero, then the estimated value for $\hat{V}$ was a root of the function.

```
import math
import scipy.optimize

def SRK(V):
    # Properties of Carbon Monoxide
    T = 300 # K
    P = 10 # atm
    Tc = 133.0 # K
    Pc = 34.5 # atm
    w = 0.049 # Acentric factor for CO
    R = 0.08206 # L atm / (mol K)
    a = 0.42747*(R*Tc)**2 / (Pc)
    b = 0.08664*(R*Tc/Pc)
    m = 0.48508+1.55171*w-0.1561*w**2
    alpha = (1+m*(1-math.sqrt(T/Tc)))**2
```

```
    term1 = R*T/(V-b)
    term2 = alpha*a/(V*(V+b))
    return P-term1+term2

# Properties of Carbon Monoxide
V = 2.0 # L/mol, initial guess

V = scipy.optimize.broyden1(SRK, V, maxiter=100,f_tol=1e-6)
print V
```

The Python script above should print out 2.46 L/mol as a root to the SRK equation, and this is the desired answer. At this point, we do not know how the broyden1 function determines this root, but, it turns out that this function is similar to Newton's method, which is covered later in the chapter.

Finally, it was noted previously that the SRK equation is a cubic equation in terms of $\hat{V}$, which implies that there could be up to 3 possible roots. One method for trying to find the other possible solutions is to change the initial guess for $\hat{V}$. Trying this approach reveals that the broyden1 function converges to 2.46 L/mol for any strictly positive guess. For any strictly negative guess, the method fails to converge to a solution, and a guess of 0.0 gives NaNs ('Not a Number') errors. These errors are usually caused by dividing by zero. This reinforces the issue of robustness with nonlinear solvers. In this simple case of one equation, the function can be plotted to show that there is likely only one root. This plot will be revisited to address the issue of robustness.

6.2 Bisection Method

The bisection method is based on identifying a region for the independent variable that bounds the root that we are trying to find. In figure 6.1 a nonlinear function crosses the x-axis at the solution and points $x = a$ and $x = b$ have been identified, which bound the solution. The algorithm is based on an iteration with the following steps:

1. calculate the midpoint of the region bounding the root, $c = (a+b)/2$,
2. calculate the value of the function, $f(c)$ at the midpoint,
3. determine which previous endpoint, $f(a)$ or $f(b)$, has the same sign (positive or negative) as $f(c)$ and replace that endpoint with c.

In figure 6.1, endpoint b would be replaced by c and the new region that bounds the root would be between a and c. Notice that the region bounding the endpoint is halved every iteration and, eventually, $f(c)$ is close to zero and the magnitude of $f(c)$ is less than a preset tolerance. The bisection method is extremely robust in that once an a and b that bound the root are known, it *always* converges to a solution (i.e., a root of the equation). The weaknesses of this approach are the requirement that a and b be determined manually (fortunately for many problems, we have a rough estimate for the solution) and convergence is slow because we are only halving the domain each iteration.

The Python script below uses the bisection method to find a root, i.e., find a value for $\hat{V}$ that satisfies the SRK equation.

```
import math
```

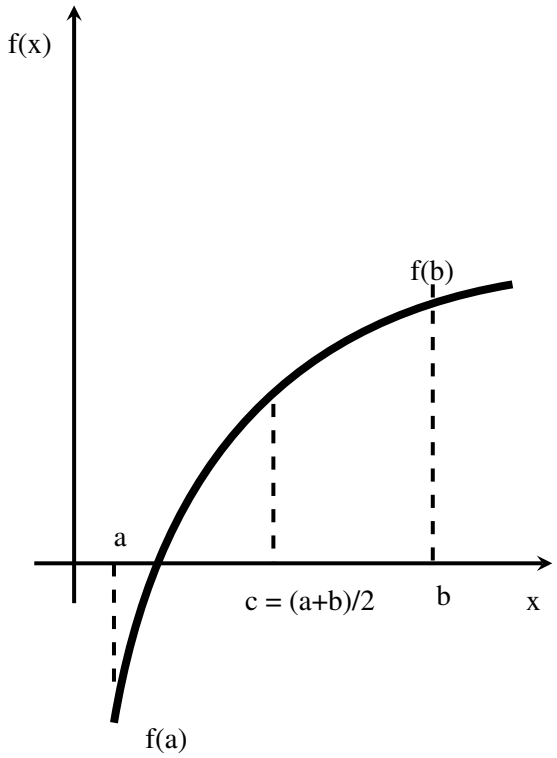


Figure 6.1: The bisection method is used to find the roots or zeros of a nonlinear function. The first step is to identify points a and b such that $f(a) \cdot f(b) < 0$ (i.e., $f(a)$ and $f(b)$ have different signs.) The distance between a and b is halved and then a or b is discarded depending on the sign of $f(c)$. This process is repeated until the location of the root within some tolerance is found.

```
def SRK(V):
    # Properties of Carbon Monoxide
    T = 300 # K
    P = 10 # atm
    Tc = 133.0 # K
    Pc = 34.5 # atm
    w = 0.049 # Acentric factor for CO
    R = 0.08206 # L atm / (mol K)
    a = 0.42747*(R*Tc)**2 / (Pc)
    b = 0.08664*(R*Tc/Pc)
    m = 0.48508+1.55171*w-0.1561*w**2
    alpha = (1+m*(1-math.sqrt(T/Tc)))**2
    term1 = R*T/(V-b)
    term2 = alpha*a/(V*(V+b))
    return P-term1+term2

#Settings
```

```
maxIter = 1000
TOL = 1e-4

# Bisection method
a = 1.0
fa = SRK(a)
b = 5.0
fb = SRK(b)
if(fa*fb >= 0):
    print "ERROR: bad starting bounds"
    exit()
# Bisection iteration
for i in range(maxIter):
    # Calculate midpoint
    c = (a+b)/2
    fc = SRK(c)
    print "c = ", c, " and f(c)= ", fc
    # Check for convergence
    if(math.fabs(fc) < TOL):
        print "Root found at ", c
        exit()
    if(fa*fc < 0):
        b = c
        fb = fc
    else:
        a = c
        fa = fc
print "Reached maxIter.  Current estimate at ", c
```

When this script is run, it prints out the midpoint, c, and $f(c)$ every iteration. Depending on the initial bounds, 15-20 iterations are required for convergence to the tolerance of 1×10^{-4}. With nonideal gas problems like this, it is possible to obtain a good initial estimate for the solution using the ideal gas law, $P\hat{V} = RT$. For carbon monoxide under the conditions listed above, the ideal gas law gives $\hat{V} = 2.46 L/mol$, which is almost identical to the solution obtained using the SRK equation. If the pressure increases or the temperature decreases, the ideal gas law becomes less accurate.

Exercise 6.1 A novice programmer rewrote the last few lines of the bisection script as shown below:

```
    # Check for convergence
    if(math.fabs(fc) < TOL):
        print "Root found at ", c
        exit()
    if(fa*fc < 0):
        c = b
```

```
            fc = fb
        else:
            c = a
            fc = fa
    print "Reached maxIter.  Current estimate at ", c
```

The resulting method failed to converge. Why? ■

6.3 Newton's Method

The most useful equation in computational mathematics may be the Taylor polynomial (also known as Taylor series or Taylor expansion). Newton's method is one of many results given in this book that may be derived using a Taylor polynomial. A Taylor polynomial is an expansion of $f(x)$ about a nearby point, x_0:

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{x - x_0}{2}f''(x_0) + O(x - x_0)^3 \ .$$

Typically, x_0 is chosen to be a point where information about the function $f()$ is known. The last term tells us that the magnitude of the term is on the order of $(x - x_0)^3$, but the exact term is not given. If the distance between x and x_0 is small (i.e., $x - x_0 << 1$) then $(x - x_0)^3$ or even $(x - x_0)^2$ are very small.

In Newton's method, we are trying to find x such that $f(x) = 0$. Assuming we have a guess, called x_0, for the value of x and setting $f(x) = 0$, the Taylor polynomial, ignoring the last two terms given above because they are (hopefully) small, becomes:

$$0 = f(x_0) + (x - x_0)f'(x_0) \ .$$

Rearranging and solving for the unknown x gives:

$$x = x_0 - \frac{f(x_0)}{f'(x_0)} \ .$$

Figure 6.2 illustrates how this equation helps us to find the root of interest. Noticing that we only retained the linear term from the Taylor polynomial, Newton's method approximates the nonlinear function with a line at current estimate for the location of the root, x_0, and that line is then used to calculate a new and hopefully better estimate for the location of the root. A new Taylor polynomial is used to approximate the nonlinear function with a line at this new location, and the process is repeat until the root is found (in the case of convergence) or an iteration limit is reached (failure).

Newton's method has two advantages over the bisection method:

1. Only a single estimate for the location of the root is required.
2. The error in approximating the nonlinear function with a straight line is on the order of $(x - x_0)^2$. As the guess, x_0, gets close to the true solution, x, the linear approximation is very good and converge is extremely fast (i.e., convergence is quadratic, or squared, near the solution).

Newton's method also has two disadvantage compared to the bisection method:

1. We need to have a function that calculates the derivative, $f'(x)$, at a point, x_0. This limitation, however, is somewhat overcome in the next section on Broyden's method.

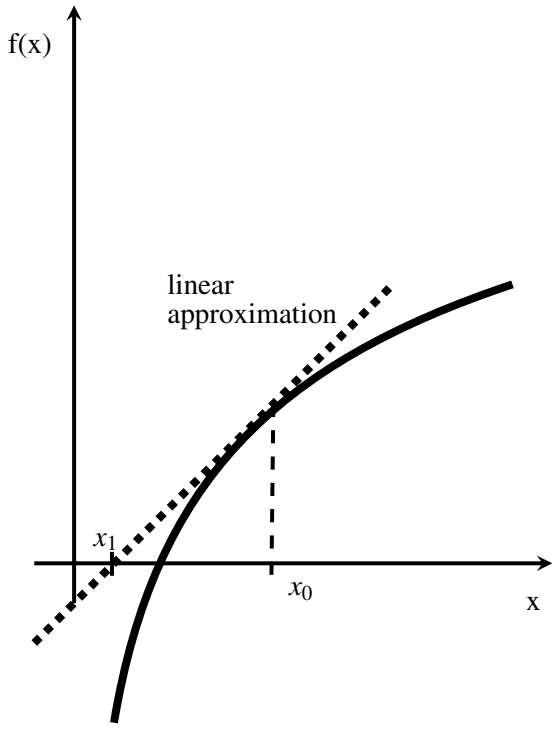


Figure 6.2: Newton's method approximates a nonlinear function with a straight line at the point x_0. The linear approximation is then used to obtain a new estimate for the root (i.e., the intersection of the nonlinear function with the x-axis) of the nonlinear function.

2. If the initial guess for the location of the root is not sufficiently accurate, the iterations can diverge and a solution will not be found. This behavior was seen previously using Broyden's method when the initial estimate of $\hat{V}$ was less than zero.

Exercise 6.2 Use Newton's method to find a solution for $x^3+2x^2-2=0$. This function displays some interesting behavior when using Newton's method, depending on the initial guess. There is only a single solution (near $x=0.8$), but an initial guess between $x\in(-2,0)$ is unlikely to converge to the one solution while other initial guesses will converge to the solution. Explain this behavior (hint: plot the function). ■

6.4 Broyden's Method

One disadvantage associated with Newton's method is the requirement that the derivative of the function of interest also be provided to the algorithm. Having an approximation for the derivative would be a significant benefit if it were available. Fortunately, the Taylor polynomial can once again help us by providing an approximation to the derivative. Recall the Taylor polynomial:

$$f(x)=f(x_0)+(x-x_0)f'(x_0)+\frac{x-x_0}{2}f''(x_0)+O(x-x_0)^3\ ,$$

and notice that it can be rearrange to provide an estimate for the derivative:

$$f'(x_0)\approx\frac{f(x)-f(x_0)}{(x-x_0)}\ .$$

If we substitute this directly into the Newton iteration, everything will cancel out because we just use the same equation twice. Instead of using the current guess, x_0, and the next guess, x to approximate the derivative, we can use the previous two guesses:

$$f'(x_1) \approx \frac{f(x_1) - f(x_0)}{x_1 - x_0} .$$

Notice that the current guess is now x_1 and the previous guess was x_0. Substituting this approximation for the derivative into Newton's method gives:

$$x_{new} = x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)} .$$

This approach is also frequently called the Secant method because the nonlinear function is approximated using a secant line instead of a tangent line.

The Python script below uses Broyden's method to find the solution to the SRK model problem that was solved previously.

```
import math

def SRK(V):
    # Properties of Carbon Monoxide
    T = 300 # K
    P = 10 # atm
    Tc = 133.0 # K
    Pc = 34.5 # atm
    w = 0.049 # Acentric factor for CO
    R = 0.08206 # L atm / (mol K)
    a = 0.42747*(R*Tc)**2 / (Pc)
    b = 0.08664*(R*Tc/Pc)
    m = 0.48508+1.55171*w-0.1561*w**2
    alpha = (1+m*(1-math.sqrt(T/Tc)))**2
    term1 = R*T/(V-b)
    term2 = alpha*a/(V*(V+b))
    return P-term1+term2

#Setting
maxIter = 1000
TOL = 1e-4

# Broyden's method
x0 = 1.0
fx0 = SRK(x0)
x1 = x0+1e-2
fx1 = SRK(x1)
for i in range(maxIter):
    xNew = x1 - fx1*(x1-x0)/(fx1-fx0)
```

```
    fxNew = SRK(xNew)
    print "xNew = ", xNew, " and fxNew = ", fxNew
    if(math.fabs(fxNew) < TOL):
        print "Root found at ", xNew
        exit()
    else:
        x0 = x1
        fx0 = fx1
        x1 = xNew
        fx1 = fxNew
print "Reached maxIter.   Current estimate at ", c
```

Using Broyden's method, the solution, $\hat{V} = 2.46$, is obtained in about 7 iterations, depending on the initial guess. The iterations are especially rapid near the solution with $f(x) = -0.00016$ after 6 iterations and $f(x) = 1.7 \times 10^{-}7$ after 7 iterations. Notice that the value of $f(x)$ is converging to zero quadratically (or nearly quadratically since we are approximating the derivative.

Using an initial guess of $x_0 = -1.0$ once again leads to an error message (in this case division by zero) and failure to converge. Now that we understand the principle behind Newton's method and Broyden's method, we can understand why this occurs. Figure 6.3 shows the nonlinear SRK function with a single root at $\hat{V} = 2.46$. For any guess near that root, if we approximate the SRK function with a straight line and then notice where that line cross the x-axis, it will probably be a point closer to the solution, and the iterations will converge. If our initial guess is negative, however, any linear approximation to the function is going to predict that the root lies further to the left. As a result each iteration will take us further and further towards $x \to -\infty$.

6.5 Multiple Nonlinear Equations

Problems that have multiple nonlinear equations that must all be solved simultaneously arise in problems as diverse as fluid piping networks (i.e., water distribution systems) to staged distillation columns. In order to solve these systems of nonlinear equations, we need to use the tools we have already learned for linear systems and single nonlinear equations and combine them together. The first step is to recall Newton's method:

$$x_{new} = x_0 - \frac{f(x_0)}{f'(x_0)} ,$$

which enables the iterative determination of x such that $f(x) = 0$. Now we are interested in the case where we have n nonlinear equations (some prefer to think of this as a *vector* of equations) written $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ and we are trying to determine the vector of unknowns, $\mathbf{x}$, that satisfies this system of equations. For multiple equations, Newton's method can be rewritten:

$$\mathbf{x_{new}} = \mathbf{x_{old}} - \frac{\mathbf{f}(\mathbf{x_{old}})}{\mathbf{f}'(\mathbf{x_{old}})} ,$$

where

$$\mathbf{f}'(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_0}{\partial x_0} & \frac{\partial f_0}{\partial x_1} & \cdots \\ \frac{\partial f_1}{\partial x_0} & \frac{\partial f_1}{\partial x_1} & \cdots \\ \vdots & & \ddots \end{bmatrix} \tag{6.1}$$

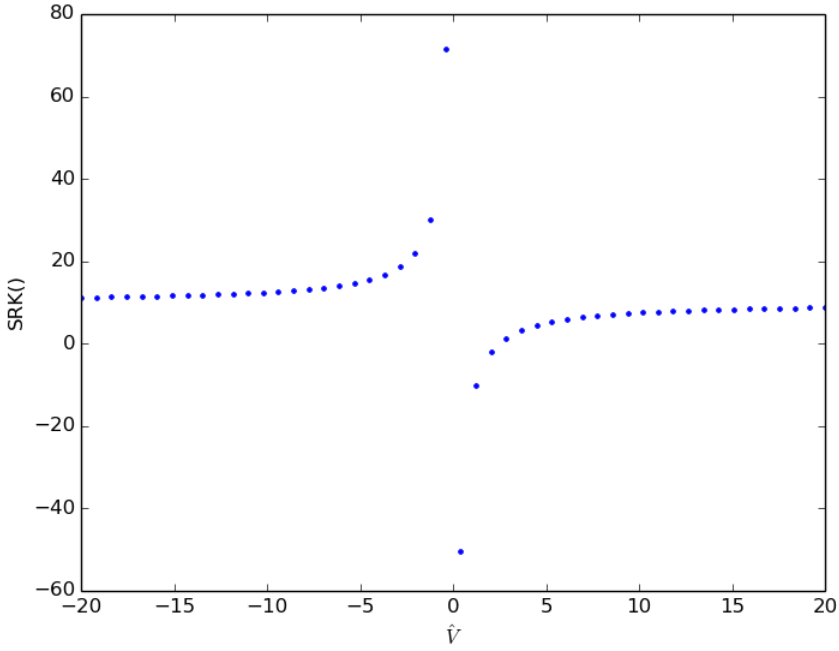


Figure 6.3: The nonlinear SRK function with a single root at $\hat{V} = 2.46$.

and

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_0(\mathbf{x}) \\ f_1(\mathbf{x}) \\ \vdots \end{bmatrix} . \tag{6.2}$$

All of the previous advantages (fast, single initial guess) and disadvantages (calculate derivatives, may not converge) apply when Newton's method is applied to a system of nonlinear equations instead of a single equation. It is common to call $\mathbf{f}(\mathbf{x})$ the *residual* or residual vector and $\mathbf{f}'(\mathbf{x})$ is the *Jacobian* or Jacobian matrix. Calculating the residual divided by the Jacobian requires solving a linear matrix problem: $\mathbf{f}'(\mathbf{x}) \cdot \mathbf{dx} = \mathbf{f}(\mathbf{x})$ for $\mathbf{dx}$. Once $\mathbf{dx}$ is determined, then $\mathbf{x_{new}} = \mathbf{x_{old}} - \mathbf{dx}$. This is wrapped in an iteration and repeated until $||\mathbf{f}(\mathbf{x})||_2 < TOL$. In other words, if $\mathbf{x}$ is close to the solution vector, then each equation in the residual should be a small number and the L^2-norm of the residual should be small.

As an example, consider the system of two nonlinear equations with two unknowns:

$$f_0(x,y) = 2 * x^2 - y^2 ,$$

$$f_1(x,y) = y - 0.5 \cdot (\sin(x) + \cos(y)) .$$

For a system this small, it is possible to use a contour plot to explore properties of the solution. Putting both of the unknowns, x and y into a single vector: $x[0] = x$ and $x[1] = y$ and plotting the contours where each function is equal to zero generates the plot shown in figure 6.4.

A Python script that uses Newton's method to solve this small system of 2 nonlinear equations is shown below.

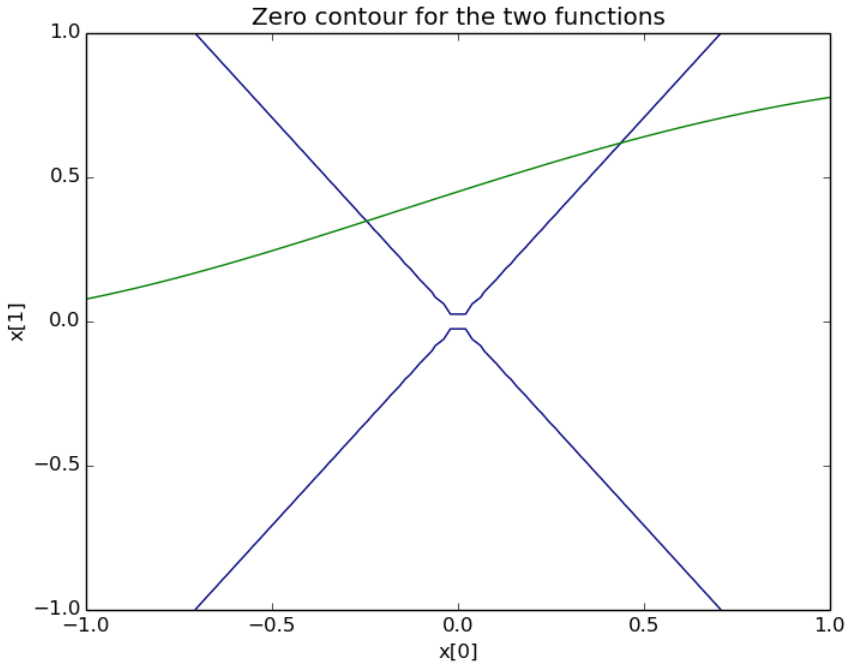


Figure 6.4: A plot of the zero contour lines (i.e., the lines where f_0 and f_1 equal zero) for the example functions. The two points where both functions are zero are solutions to this nonlinear system, and two different solution vectors are seen to exist in this plot.

```
import numpy
import numpy.linalg as nl
import math

def res(x):
    n = x.size
    f = numpy.zeros(n,dtype=numpy.float64)
    f[0] = 2*x[0]**2 - x[1]**2
    f[1] = x[1]-0.5*(math.sin(x[0])+math.cos(x[1]))
    return f

def jac(x):
    n=x.size
    j = numpy.zeros((n,n),dtype=numpy.float64)
    j[0,0] = 4*x[0]
    j[0,1] = -2*x[1]
    j[1,0] = -0.5*math.cos(x[0])
    j[1,1] = 1.0 + 0.5*math.sin(x[1])
    return j
```

```
x = numpy.array([1.0,1.0])
TOL = 1.0e-6
maxIter = 20
for i in range(maxIter):
    f = res(x)
    j = jac(x)
    print "Iteration ",i," has a norm of ",nl.norm(f,2)
    if (nl.norm(f,2) < TOL):
        print "Converged in ", i, " iterations to ", x
        break
    else:
        dx = nl.solve(j,f)
        x = x - dx
if(i == maxIter):
    print "Failed to converge in ", maxIter, " iterations"
```

The residual vector and Jacobian matrix are calculated in separate functions that are called each Newton iteration. The L^2-norm of the residual vector is then calculated to check for convergence. If convergence has not been achieved, then the linear matrix problem is solved and the guess at the solution is updated based on that solution. Using an initial guess of $(1.0, 1.0)$, Newton's method required 5 iterations to converge to one of the possible solutions, $[0.44, 0.62]$. It also displayed rapid, quadratic convergence once the approximate solution was close to the final solution.

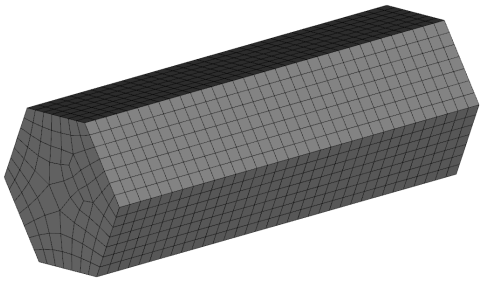
As with Broyden's method, it is possible to avoid having to calculate an exact Jacobian matrix. Numerical approximations for all the different derivatives within the Jacobian matrix can be used instead of writing an explicit Jacobian function. Most of the algorithms in the scipy.optimize library for solving systems of nonlinear equations are capable of calculating an approximate Jacobian matrix automatically. The Python script below demonstrates three different algorithms available in scipy.optimize for solving linear systems of equations.

```
import numpy
import math
from scipy.optimize import fsolve, broyden1, newton_krylov

def res(x):
    n = x.size
    f = numpy.zeros(n,dtype=numpy.float64)
    f[0] = 2*x[0]**2 - x[1]**2
    f[1] = x[1]-0.5*(math.sin(x[0])+math.cos(x[1]))
    return f

x0 = numpy.array([0.0,0.0])
x = fsolve(res,x0)
print "Converged to ", x
```

```
x = newton_krylov(res, x0)
print "Converged to ", x
x = broyden1(res, x0)
print "Converged to ", x
```

Only the original nonlinear functions are provided to these algorithms, and the algorithms automatically approximate the Jacobian. All three of these algorithms,
fsolve(), broyden1(), **and** newton_krylov() are basically Broyden's method, which was discussed previously, extended to multiple equations. Interestingly, using an initial guess of $(0.0, 0.0)$, two of the methods, fsolve() **and** broyden1() converged to the solution at $\mathbf{x} = (0.44, 0.62)$, but newton_krylov() converged to the other solution, $\mathbf{x} = (-0.25, 0.35)$.

Introduction
Reading Data from a File
Parsing an Array
Statistical Analysis
Advanced Linear Regression



7. Statistics

7.1 Introduction

The rigorous statistical analysis of data is a critical aspect of our never ending pursuit of *more* – more efficiency, more consistency, more profitability, etc. There is a large number of outstanding software packages that facilitate the processes of statistical analysis. Commercial packages such as IBM SPSS Statistics and SAS are widely used, and open source (free) packages such as R, are well documented packages with large communities of users. The motivation for *briefly* examining select statistical functions available in the `scipy.stats` library is that it is often helpful to combine statistical analysis with other mathematical computations. For example, we might be using a Python script to model a fermentation reactor, and one input to that model is the dissolved oxygen concentration for the inflow stream. Before running the reactor simulation, we might want to calculate the averaged dissolved oxygen concentration as well as the standard deviation. Being able to perform all these calculation – from solving ordinary differential equations to standard deviation calculations, in one software package or programming environment can help us to be more efficient and make fewer mistakes. Transferring results from one computer to another computer or one software platform to another software platform is a time consuming process that frequently introduces mistakes. Thus, it should be avoided whenever possible.

7.2 Reading Data from a File

Before data can be statistically analyzed, it often needs to be loaded from a file. In this section, a few different options are examined for loading data from a common separated value or csv file. This format is used here because it is a common format for sharing data between different software packages. Most spreadsheets, including Excel, can export the contents of a spreadsheet as a csv file. The goal of the different methods presented in this section is to import the contents of a csv text file into a numpy array for later statistical analysis and plotting.

The first data set that will be imported contains multiple pH, dissolved oxygen (DO), and temperature data from 6 different locations in a river. The locations are distinguished only by a number (1 through 6). Figure 7.1 shows a screen capture of the contents of the first part of the data

file (DOdata.csv). The file contains a header row that contains the names of the various columns of data and the actual data follows below.

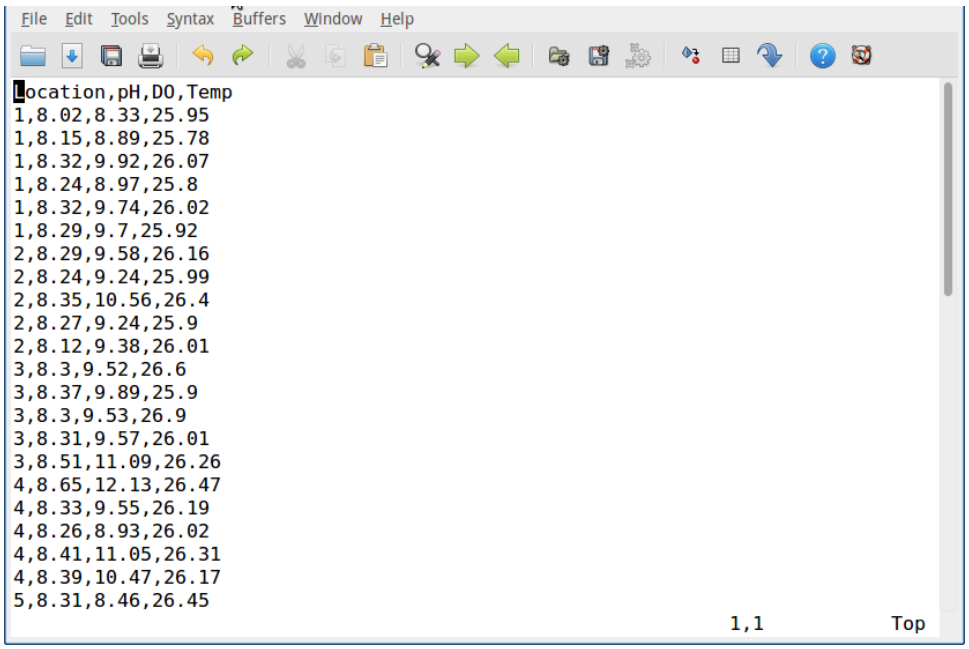


Figure 7.1: A screen capture showing the first few lines of the file 'DOdata.csv'. The file contains a header row followed by the data.

The simplest method for importing data from a file into a numpy array is to use the function numpy.genfromtxt(). This function is passed as inputs the name of the file and the delimiter (i.e., the character separating the unique data points), which is typically a comma or a tab character. The Python script below illustrates the use of this function for importing data from a csv file.

```
import numpy

data = numpy.genfromtxt("DOdataNoHeader.csv",delimiter=",")
numRows, numCols = data.shape
```

One drawback to this approach is that if the file contains a header row, such as the example data shown in figure 7.1, the header text is automatically added to the numpy array, which can lead to problems later. If this approach is used, the csv file should be carefully checked and any header text should be removed.

The second approach for importing data from a file into a numpy array uses standard Python functions for opening a file, and then reading the file contents one row at a time. Every time a new row is read in from the file, it is loaded into the numpy array using the numpy.fromstring() function. This approach requires the construction of an empty numpy array to hold new data as it is read in from the file. The advantage of this approach is that it allows one to load the header information from the file and store it in a separate variable from the numpy array. The Python script below implements

this more manual approach for reading in data from a cvs file.

```
import numpy

# open file containing data
dataFile = open("DOdata.csv")

# read first line with headers and split them up
header = dataFile.readline()
dataHeaders = header.split(',')
numCols = len(dataHeaders)

# set maxRows we are capable of reading
maxRows = 256
data = numpy.zeros((maxRows,numCols),dtype=numpy.float64)

# read in data one row at a time
rowCount = 0
for row in dataFile:
    data[rowCount,:] = numpy.fromstring(row,sep=',')
    rowCount += 1
    if(rowCount >= maxRows):
        break
```

A few important observations can be made from this routine. First, both the split () function and the numpy.fromstring() function are passed a character for delimiting between columns. Second, a parameter limiting the max number of rows of data that can be loaded from the file is set. This is recommended for security (avoids accidentally reading in 10GB worth of data) and computational speed (avoids have to resize the numpy array every time new data is loaded).

The third approach for loading data from a cvs file takes advantage of the cvs library that is a standard part of the Python language. This approach still allows special handling of the header row, and it also requires the creation of an empty numpy array for storing the data as it is read from the file. When each row is read from the file, it is imported as a list of strings, i.e., each number is a unique string. These strings are converted into floating point numbers using the function numpy.astype(**float**). Once all the rows are read from the file, the numpy array can be resized to truncate an empty rows from the end of the array that was originally created. The Python script below implements this third approach.

```
import csv
import numpy

with open('DOdata.csv', 'rb') as dataFile:
    reader = csv.reader(dataFile)
    header=reader.next()
```

```
    numCols = len(header)
    maxRows = 256
    data = numpy.zeros((maxRows,numCols))
    rowCount = 0
    for row in reader:
        data[rowCount,:] = numpy.array(row).astype(float)
        rowCount += 1
        if(rowCount >= maxRows):
            break
    data = numpy.resize(data,(rowCount,numCols))
```

7.2.1 Parsing an Array

Once all the data from the file is imported into a numpy array, it is sometimes desirable to extract out a subset of this data. For example, the data set that has been used in this chapter contains multiple pH, DO, and temperature measurements at 6 different locations. If we want to compare the average DO at location 1 to the average DO at location 2, it is helpful to parse or extract that data from the larger array. This process often requires that we first count the amount of data that is going to be extracted, then allocate space in an empty array for the data, and finally reading and copying the data into the new array.

For the data set shown in figure 7.1, a simple function can be written that parses the larger array and returns a small array with only the location data of interest.

```
def getDOlocation(data, loc):
    countLocation = 0
    for i in range(numRows):
        if(data[i,0] == loc):
            countLocation += 1
    DOdata = numpy.zeros(countLocation)
    countLocation = 0
    for i in range(numRows):
        if(data[i,0] == loc):
            DOdata[countLocation] = data[i,2]
            countLocation += 1
    return DOdata
```

This function is passed the full data set (a numpy array) and the location of interest. It counts the number of data points at that location, creates an empty array of appropriate size, copies the data into the new array, and then returns the data of interest. Notice that two loops through the full data set are required, the first loops counts the number of data points at the desired locations and the second loop copies the data.

7.3 Statistical Analysis

Before conducting any statistical analysis on a data set, it is always a good idea to plot a histogram of the data. Matplotlab (i.e., pylab) has a built in function for generating a histogram plot from a vector

containing the data of interest. The plotting function is called with: pylab.hist(data[:,3],9) where the first argument is a vector with the data of interest (in this case we are asking for a histogram of the temperature data across all locations) and the second, optional argument is number of bins (i.e., the number of temperature ranges). The histogram of the temperature data is shown in figure 7.2 and the x-axis indicates the temperature ranges for each bin and the y-axis is the frequency of data points within each range. Some of the statistical analysis that is covered in this section relies on data that is normal (or Gaussian) distributed. Examination of the histogram can provide some insight into whether or not a particular data set is normally distributed.

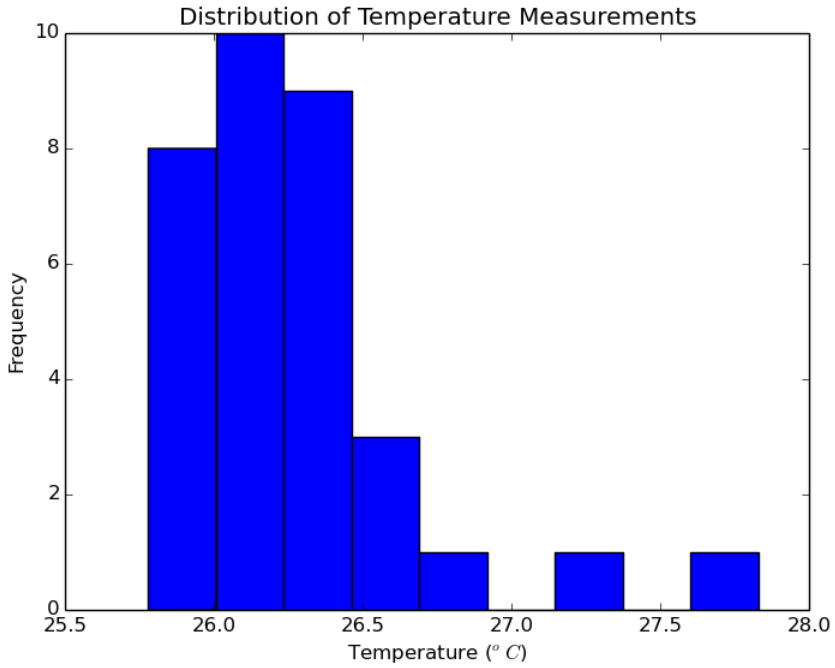


Figure 7.2: A histogram of 33 temperature measurements. The data does not appear to be normally distributed.

Numpy has built in functions for calculating the mean and standard deviation for an array. Using the getDOlocation() function above to extract out the dissolved oxygen (DO) data for locations 1 and 2, the mean and standard deviation can be calulated using the numpy.mean() and numpy.std() function as illustrated in the code below.

```
DOdata1 = getDOlocation(data, 1)
DOdata2 = getDOlocation(data, 2)
print "Location 1: average = ", DOdata1.mean()
print " and std. dev. = ", DOdata1.std()
print "Location 2: average = ", DOdata2.mean()
print " and std. dev. = ", DOdata2.std()
```

For the DOdata.csv file data, the average DO for location one is 9.26 ± 0.57 mg/L and the average

for location two is 9.6 ± 0.50 mg/L. An important question is whether or not these two sample means are statistically different. If additional data is collected, would the means converge to the same mean or different means? This question can be answered using a t-test, which is used to compare two means in order to determine the probability that the means are the same. For these two data sets, DOdata1 and DOdata2, a t-test comparison can be performed using:
t, prob = scipy.stats.ttest_ind (DOdata1,DOdata2,equal_var = True). This function requires two input vectors containing the two different sets of data whose means are being compared in a t-test. A third, optional input is a boolean (i.e., True or False) that indicates whether or not the variance is approximately equal between the two sets of data. If the two data sets are both from experimental measurements, we typically assume that their variances are equal. However, if a set of experimental data is being compared to published data, then equal_var=False should be used instead.

For the DO data from locations 1 and 2 in a river, the value for 't' or the t-stat is -0.95. In order to interpret this value, it needs to be compared to a critical t value from a table. We can avoid this work by simply focusing on the value of 'prob' or probability that is returned. This value reflects the probability that the two means are the same. If $prob < 0.05$, then we can say with 95% confidence that the two means are different. If $prob < 0.01$ then we can be 99% confident that the means are different. These are the two thresholds commonly used in statistically comparing two data sets. For the data examined here, prob equals 0.37 so we cannot say that the two means are different and no statistical conclusion should be made. The means might be the same or they might be different, we simply do not have enough data to have confidence in either result.

In some cases, the data that we collect and want to analyze is subjective. For example, imagine if we asked 100 people to score 20 different movies from this past year on a scale of 1 to 10. Some people are 'easy graders' and would give scores closer to 10 even when they did not really like the movie. Other people are 'tough graders' and they never give a score over 8. Further, some people would use the full range of possible values, giving the worst movie a 1 and the best movie a 10 while other people would use a small, clustered range of scores (i.e., the worst movie gets a 6 and the best movie gets a 9). Whenever we have subjective scores like this with variable distributions, it is helpful to normalize the scores using a z-score, which is also known as the standard score. The z-score is defined as:

$$z = \frac{x - \mu}{\sigma}$$

where μ is the average score and σ is the standard deviation for one set of scores. Thus, each persons z-scores for the 20 movies are centered around zero (good movies have a z-score greater than zero and below average movies have a negative z-score) and the z-score reflects how many standard deviations better or worse than the average, for each person rating movies, a movie end up being rated. A z-score of 2.0 reflects a movie that is 2 standard deviations better than average (i.e., the person ranked the movie in their top 5% of movies).

A data set can be automatically translated into z-scores using the function:
zscore = scipy.stats.zscore(DOdata1). The input into the function is the original data, and the function returns the normalized scores or z-scores. The length of the **zscore** vector should be the same as the vector containing the original data, and the z-scores should average zero. The z-scores for the DO data at location 1 are: -1.63, -0.65, 1.16, -0.51, 0.85, and 0.77. The greatest deviation from the average for the DO data is -1.6 standard deviations below average.

7.4 Advanced Linear Regression

Linear regression was discussed previously, but the scipy.stats library provides a linear regression function that returns additional analysis that can be helpful in interpreting the results. To explore the linear regression function in scipy.stats, a new data set that contains dissolved oxygen and temperature measurements as a function of time is used. Further, the measurements were repeated so two different time dependent data sets are available. The data is stored in two separate csv files: DOdepletion1.csv and DOdepletion2.csv. Whenever such a situation arises, it is helpful to write a Python script to open a file and import the data for a specific trial number – trial 1 or trial 2 in this case. The Python script below is passed a trial number and it then builds the file name using the trial number and string concatenation.

```
def loadData(trial):
    filename = 'DOdepletion' + str(trial) + '.csv'
    with open(filename, 'rb') as dataFile:
        reader = csv.reader(dataFile)
        header=reader.next()
        numCols = len(header)
        maxRows = 10
        data = numpy.zeros((maxRows,numCols))
        rowCount = 0
        for row in reader:
            data[rowCount,:] = numpy.array(row).astype(float)
            rowCount += 1
            if(rowCount >= maxRows):
                break
        data = numpy.resize(data,(rowCount,numCols))
        return data
```

The DO measurements from the first data set are plotted in figure 7.3. It is clear that the DO concentration is reduced over time, but it is not clear what order process is governing the clearance process. Most kinetic processes like this are first-order or second-order. The rate of change (i.e., derivative) for a first-order process is governed by

$$\frac{dDO}{dt} = -k \cdot DO$$

where k is the rate constant and DO is the dissolved oxygen concentration in mg/L. For a first-order process, the rate of depletion is proportional to the concentration and a faster depletion rate is observed at higher concentrations. In a differential equations course or a kinetics course, the solution to this equation is derived. The solution is just presented here:

$$DO = DO_0 \cdot e^{-kt}$$

or

$$\ln(DO) = \ln(DO_0) - kt$$

where DO_0 is the initial dissolved oxygen concentration. If a process is first-order, then these equations should fit the data. On the other hand, if a process is second-order the rate of change is governed by

$$\frac{dDO}{dt} = -k \cdot DO^2$$

and the solution to this equation is given by

$$\frac{1}{DO} = \frac{1}{DO_0} + kt \ .$$

For the DO measurements plotted in figure 7.3, we wish to determine if the process is first-order or second-order using linear regression to fit the data to both rate laws and determining which provides the better fit.

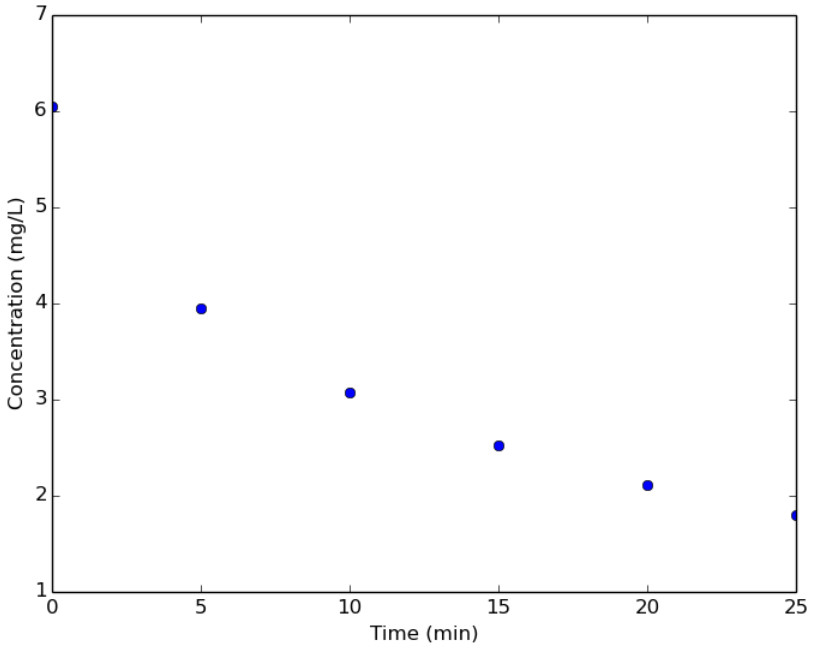


Figure 7.3: Dissolved oxygen (mg/L) measurements as a function of time for an isolated sample. The order of the depletion rate is not clear.

In order to fit the data with a first-order model, we need to perform linear regression on $\ln(DO)$ versus t. Since

$$\ln(DO) = \ln(DO_0) - kt \ ,$$

the slope from regression should be an estimate for k and the intercept should be an estimate for $\ln(DO_O)$. The `scipy.stats.linregress()` function can be called and passed the time and natural log of DO data using

```
slope, intercept, r_value, p_value, std_err =
scipy.stats.linregress(dataSet1[:,0],numpy.log(dataSet1[:,1]))
```

This function returns the slope, intercept, R, and p-value resulting from linear regression analysis. The value of R is a measure of how close the data is to falling on a straight line. An R value close to 1.0 indicates that the data is highly linear and falls close to the regression line. The p-value indicates the probably that slope is zero, i.e., the probability that the data in the second vector (the y-data) is independent of the data in the first vector (the x-data). For the first set of DO data, the value of $R^2 = 0.96$, indicating a good but not great fit, and the p-value is 0.0005, indicating that the DO value has an extremely low probability of being independent of time. Figure 7.4 shows the impact of adding the curve corresponding to the first-order model to the previous plot of the data. The fit is far from perfect and indicates that the first-order model is probably not correct.

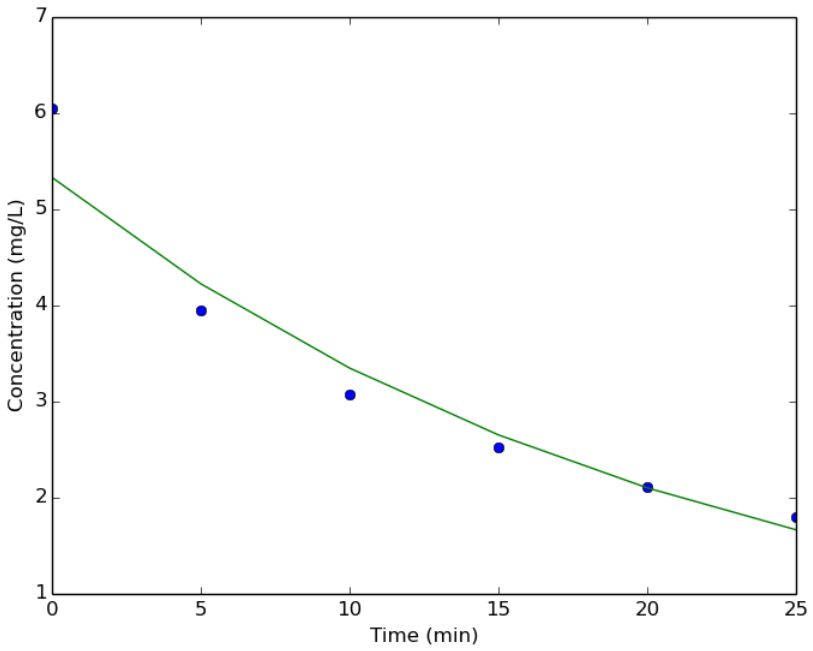


Figure 7.4: Dissolved oxygen (mg/L) measurements as a function of time and the best fit curve based on a first-order depletion model.

Fitting the data with a second-order model requires performing linear regression on $\frac{1}{DO}$ versus t. The resulting slope from linear regression should correspond to k, and the intercept should be equal to $\frac{1}{DO_0}$. The scipy.stats.linregress() function is called using:

```
slope, intercept, r_value, p_value, std_err =
scipy.stats.linregress(dataSet1[:,0],1/dataSet1[:,1])
```

The value of R^2 for the second-order model is 0.999, indicating that the data falls extremely close to a straight line. This is strong evidence that the second-order model is the correct model. Once again, the p-value is very small at 3×10^{-7}. Adding the best-fit curve corresponding to the second-order

model (figure 7.5) to the original data plot shows that the model fits the data very well and the true order of the process is probably second-order.

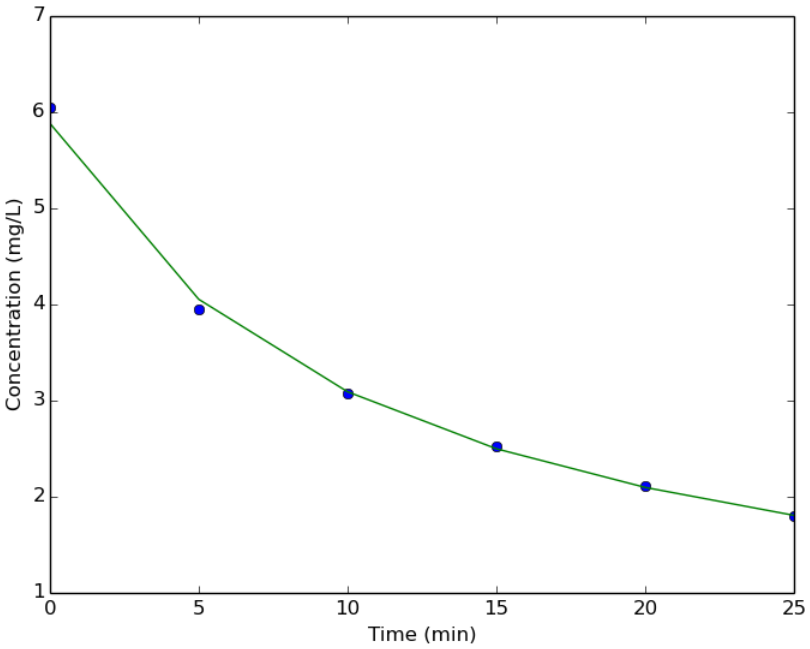
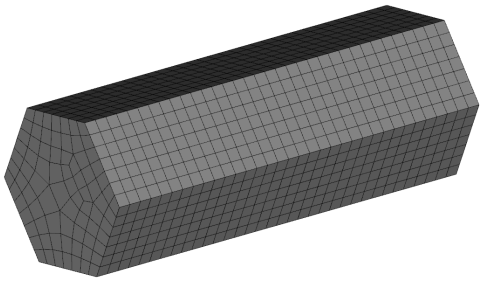


Figure 7.5: Dissolved oxygen (mg/L) measurements as a function of time and the best fit curve based on a second-order depletion model.

Introduction
Numerical Differentiation
First Derivative Approximation
Second Derivative Approximation
Numerical Integration
Numerical Integration Using Scipy



8. Numerical Differentiation and Integration

8.1 Introduction

Most students learn how to calculate the derivative or integral of a given function in a first year calculus course. Computational methods for taking derivatives and integrals of functions were also described in Chapter 3 on symbolic computations. For some complex functions, however, it is simply not possible to calculate the integral or even the derivative, and it maybe necessary to obtain a numerical approximation of derivative or integral. Sometimes it is just easier to use a numerical approximation than to determine the exact derivative or integral. In this chapter, standard approaches for numerically approximating derivatives and integrals are described. The numerical approximation of a derivative is usually not particularly useful until it is used to approximately solve differential equations, as described in later chapters. The numerical approximation of a definite integral, however, is immediately useful for calculating such quantities as enthalpy changes and reactor volumes. This chapter is divided into two parts. The first briefly examines numerical approximations for the first- and second-derivative of a given function at a point. The second part describes a few different methods of numerically approximating definite integrals.

8.2 Numerical Differentiation

The numerical approximation of a derivative has already been briefly explored when Newton's method was discussed earlier. In this section, common choices for numerically approximating a derivative are examined along with an analysis of the error associated with the approximation. These approximations can be derived from the Taylor polynomial, but here, the Taylor polynomial will be written in a slightly different form from that shown in Chapter 6 on nonlinear equations. The Taylor polynomial provides an approximation for a function about the point x_i. The value of the function a distance h away from x_i is:

$$f(x_i+h) = f(x_i) + h\frac{df(x_i)}{dx} + \frac{h^2}{2}\frac{d^2f(x_i)}{dx^2} + \ldots$$

or

$$f(x_i+h) = f(x_i) + h\frac{df(x_i)}{dx} + O(h^2)$$

where $O(h^2)$ represents a term with a size on the order of h^2. This equation can be seen to be equivalent to the previous Taylor polynomial equation by replacing h with $x - x_i$.

8.2.1 First Derivative Approximation

Our goal in this section is to approximate $\frac{df(x_i)}{dx}$, and we can rearrange the Taylor polynomial above to give:

$$\frac{df(x_i)}{dx} = \frac{f(x_i+h) - f(x_i)}{h} + O(h)$$

or

$$\frac{df(x_i)}{dx} \approx \frac{f(x_i+h) - f(x_i)}{h} .$$

This approximation is typically referred to as the forward approximation of the first derivative, and the error associated with the approximation is of order h. As we will see in the next few chapters, numerical approximation of derivatives is often performed on a sequence of points that are separated by a fixed distance h. Hence, it is common to simplify this notation slightly by replacing $f(x_i)$ by f_i and $f(x_i+h)$ with f_{i+1} so that it clearly refers to the next point in a sequence of points. Using this notation, the forward difference approximation of the derivative becomes:

$$\frac{df(x_i)}{dx} \approx \frac{f_{i+1} - f_i}{h} .$$

The use of this approximation is shown in figure 8.1(b). Clearly, the approximation of the slope becomes more accurate as the distance between the two points shrinks towards zero.

The Taylor polynomial to approximate the derivative at $f(x_i - h)$ is

$$f(x_i-h) = f(x_i) - h\frac{df(x_i)}{dx} + O(h^2)$$

and using the same derivation as above, leads to the backward difference approximation of the derivative:

$$\frac{df(x_i)}{dx} \approx \frac{f_i - f_{i-1}}{h} ,$$

which has the same $O(h)$ accuracy as the forward difference approximation. The use of the backward difference approximation is shown in figure 8.1(a). The final method for approximating the derivative of $f(x)$ at x_i is by subtracting the Taylor polynomial for $f(x_i - h)$ from the polynomial for $f(x_i + h)$. Notice that the $f(x_i)$ terms are eliminated and, more importantly, it can be shown that the $O(h^2)$ terms are eliminated leaving a $O(h^3)$ term. The resulting centered difference approximation of the first derivative is:

$$\frac{df(x_i)}{dx} \approx \frac{f_{i+1} - f_{i-1}}{2h} ,$$

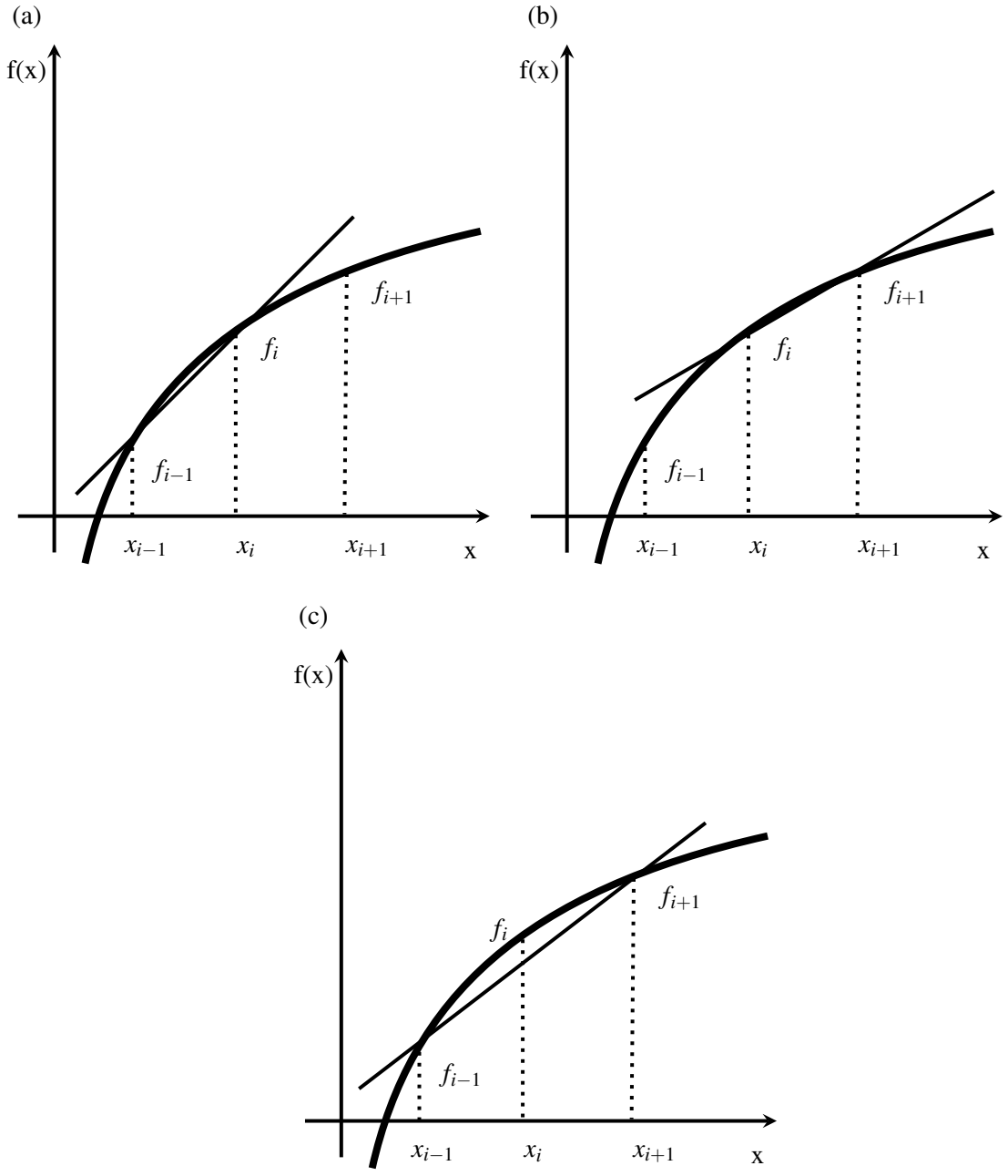


Figure 8.1: Three different finite difference approximations for the derivative at x_i: (a) backward difference, (b) forward difference, and (c) centered difference.

which is $O(h^2)$ accurate. As a result, as $h \to 0$, the centered difference approximation is much more accurate than either the forward or backward difference approximation. The centered difference approximation is illustrated in figure 8.1(c).

To illustrate both the implementation and accuracy of numerical approximations for the first derivative, the Python code below was created. The code has the function $f(x) = x \cdot \sin(x)$ declared near the beginning, and the code has a second function that calculates the exact derivative. The derivative at $x = 1.0$ is approximated using the 3 different approximations described above, and the accuracy of each approximation is calculated.

```
import math

def fun(x):
    return x*math.sin(x)

def exactDeriv(x):
    return math.sin(x)+x*math.cos(x)

x0 = 1.0
exact = exactDeriv(x0)

for i in range(5):
    h = 10**(-i-1)
    # forward difference
    df_forward = (fun(x0+h)-fun(x0))/h
    df_backward = (fun(x0)-fun(x0-h))/h
    df_center = (fun(x0+h)-fun(x0-h))/(2*h)
    print "forward error is ", math.fabs(df_forward-exact)
    print "backward error is ", math.fabs(df_backward-exact)
    print "centered error is ", math.fabs(df_center-exact)
```

Running the Python script for numerical differentiation gives the results shown in table 8.1. The forward and backward finite difference approximations give very similar levels of error, and both options give an error that is $O(h)$. The centered difference approximation, on the other hand, has a similar level of error for large values of h, but it converges much faster to the exact solution and has an error that is $O(h^2)$. Interestingly, even though it is not clear at this point why anyone would ever consider using a forward or backward difference approximation, we will later see situations where we need to accept the poor accuracy of the lower order approximations because they demonstrate better numerical stability. So, don't forget about the forward and backward difference approximations because they will have a use later.

8.2.2 Second Derivative Approximation

The second derivative of a function can also be numerically approximated, and, once again, we turn to the Taylor polynomial to derive an equation for the approximation. Recalling that:

$$f(x_i + h) = f(x_i) + h\frac{df(x_i)}{dx} + h^2\frac{d^2f(x_i)}{dx^2} + O(h^3)$$

h	forward difference error	backward difference error	centered difference error
0.1	6.8×10^{-3}	1.7×10^{-2}	5.1×10^{-3}
0.01	1.1×10^{-3}	1.2×10^{-3}	5.1×10^{-5}
0.001	1.2×10^{-4}	1.2×10^{-4}	5.1×10^{-7}
0.0001	1.2×10^{-5}	1.2×10^{-5}	5.1×10^{-9}
1.0×10^{-5}	1.2×10^{-6}	1.2×10^{-6}	5.5×10^{-11}

Table 8.1: Accuracy of different numerical approximations of the first derivative of $f(x) = x \cdot \sin(x)$.

and

$$f(x_i - h) = f(x_i) - h\frac{df(x_i)}{dx} + h^2\frac{d^2f(x_i)}{dx^2} - O(h^3) \ ,$$

we can add these two equations together (note that the first derivative terms cancel and the $O(h^3)$ terms cancel) giving:

$$f(x_i + h) + f(x_i - h) = 2f(x_i) + h^2\frac{d^2f(x_i)}{dx^2} + O(h^4) \ .$$

This equation can be rearranged to solve for the second derivative and the notation simplified to yield:

$$\frac{d^2f(x_i)}{dx^2} \approx \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} \ ,$$

which is $O(h^2)$ accurate. This approximation is also called the centered difference approximation for the second derivative. There are many other approximations that have been derived for the second derivative, but this one approximation is used in the vast major of engineering algorithms in the author's experience.

8.3 Numerical Integration

The calculation of the definite integral of a function in one-dimension is identical to calculating the area between the function and the x-axis. If we were not at all concerned with accuracy, we could approximate the function with a straight line between the bounds on the definite integral, (a, b). The resulting polygon would be a trapezoid and we could easily approximate the area with:

$$\int_a^b f(x)dx \approx \frac{h}{2}[f(a) + f(b)] \ .$$

The error associated with this approximation depends on the nonlinearity of the function (i.e., roughly, the second derivative) but for most problems, the use of a single trapezoid is not sufficiently accurate and much better accuracy is possible at a modest computational cost.

If the region under the function is subdivided into n intervals of width $h = (b - a)/n$, then arbitrarily high accuracy is possible by increasing n. Approaches that use multiple polygons to approximate the area under the curve are called composite method. The simplest composite method is to divide the region under the function into rectangles. Each rectangle typically has the width,

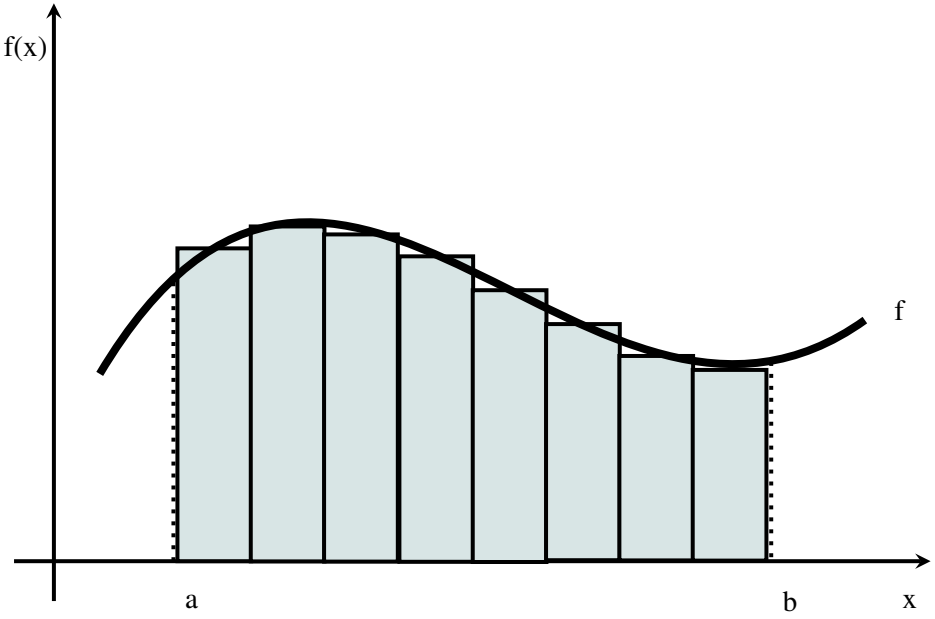


Figure 8.2: The fundamental idea behind the numerical approximation of a definite integral (from a to b) is to estimate the area under the curve using simple subdomains with areas that are easy to calculate. The composite midpoint rule is illustrated here.

h, and the height of each rectangle is determined by evaluating the function at the midpoint of the rectangle's width. This approach is called the composite midpoint rule because the midpoint is used to determine the size of each rectangle. The midpoint rule process is illustrated in figure 8.2. The equation describing this approximate integral is

$$\int_a^b f(x)dx \approx 2h \sum_{j=0}^{n} f(x_j)$$

where x_j is at the midpoint of each subdomain. The error associated with this approximation is $O(h^2)$, which means that using twice as many subdomains (i.e., halving h) results in a factor of 4 reduction in the error.

Instead of approximating the area under the function with a sequence of rectangles, a sequence of trapezoids could be used instead. The process is illustrated in figure 8.3. The equation describing this process is:

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{(} n-1)f(x_j) + f(b)\right] .$$

Interesting, the accuracy of the composite trapezoid rule is the same as the composite midpoint rule, $O(h^2)$, but the trapezoid rule has the same overall computational cost and is typically more accurate by a factor of 2. Whenever we read that the order of accuracy is h^2 (i.e., $O(h^2)$) we can think of this as saying that the error is equal to $k \cdot h^2$ where k is a constant, or we can think of this as saying that the error is proportional to h^2. If the error of the composite midpoint rule is $k \cdot h^2$, then the error

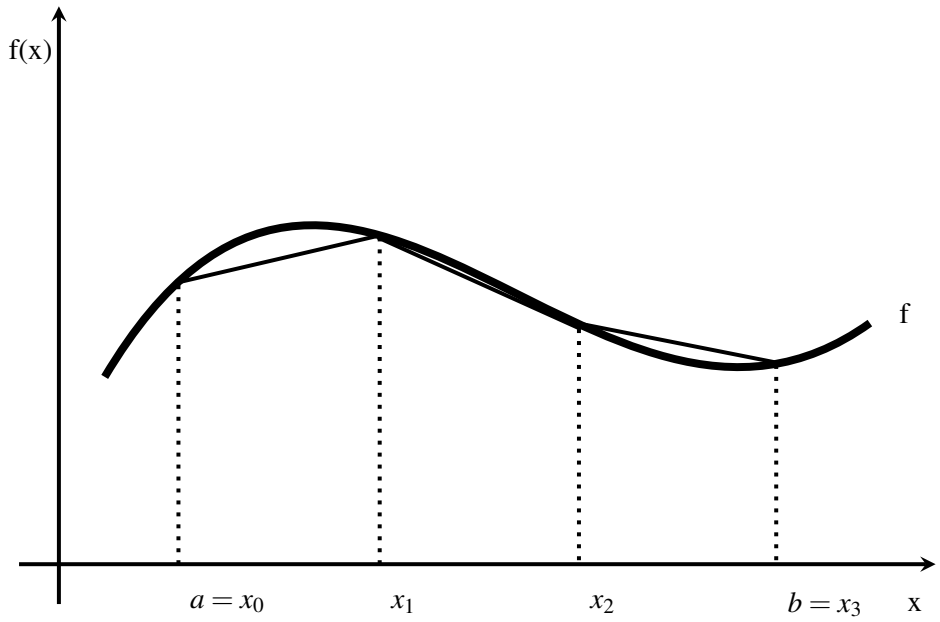


Figure 8.3: The area under a function (i.e., the definite integral of a function) can be estimated by subdividing the area into a sequence of trapezoids.

associated with the composite trapezoid rule for that same problems is $\frac{k}{2}h^2$. Because the composite trapezoid rule provides twice the accuracy for the same computational cost, it is generally preferred over the composite midpoint rule.

The composite trapezoid rule is straightforward to implement in Python, and an (inefficient) implementation is given in the script below. This implementation is inefficient because the function being integrated is evaluated twice at the same location, x. This is obviously a waste, but the algorithm is so fast that this little inefficiency does not really matter.

```
import math
import numpy

def fun(x):
    return x*numpy.sin(x)

def exactIntegral(a, b):
    ans = -b*math.cos(b)+math.sin(b)+a*math.cos(a)-math.sin(a)
    return ans

a = 0.0
b = 2.0

# Trapazoid Rule
```

```
n = 100
h = (b-a)/(n-1)
x = numpy.linspace(a,b,num=n)
area = 0
for i in range(n-1):
    area = area + h*(fun(x[i]) + fun(x[i+1]))/2.0
print "error: ", math.fabs(exact-area)
```

For the test problem used here, using $n = 10$ results in an error of 3.3×10^{-3} and using $n = 100$ results in an error of 2.6×10^{-6}. This error reduction is even larger than expected ($O(h^2)$ is expected) due to the smoothness of the function being integrated.

8.3.1 Numerical Integration Using Scipy

The numerical integration techniques examined thus far are based on evaluating the function being integrated at evenly spaced points, i.e., the midpoint of a subdomain or the endpoints of a subdomain. Higher accuracy can be achieved by evaluating the function being integrated at strategically placed points that minimize the error associate with the approximation. These 'optimal' points are not at the ends of the subdomain, and they are not evenly spaced throughout the subdomain. Fortunately, mathematicians have previously determined the locations of these optimal points, they are called *Gauss points*, and the numerical integration method is called *Gaussian quadrature*.

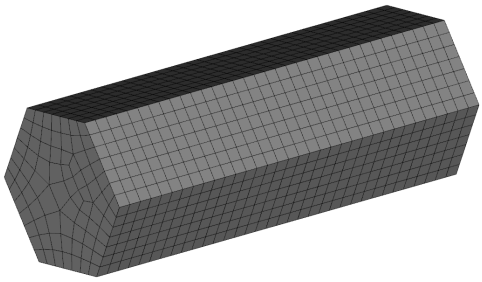
An algorithm that implements the Gaussian quadrature approach for numerically approximating integrals is included in the `scipy.integrate` library. The use of this algorithm is illustrated in the Python script below.

```
import math
import numpy
import scipy.integrate.quadrature as quad

def fun(x):
    return x*numpy.sin(x)

def exactIntegral(a, b):
    ans = -b*math.cos(b)+math.sin(b)+a*math.cos(a)-math.sin(a)
    return ans

a = 0.0
b = 2.0

exact = exactIntegral(a, b)
estimate = quad(fun,a,b)
print "error: ", math.fabs(exact-estimate[0])
```

The Gaussian quadrature function is passed the name of a Python function contain the equation we are approximately integrating, and it is passed the bounds on the definite integral. The use of Gaussian quadrature provides an extremely accurate approximation of the integral at a relatively

modest computational cost in many cases. For the example equation used in the Python script above, $f(x) = x \cdot \sin(x)$, the Gaussian quadrature algorithm in Scipy returns the approximate integral that has an error of only 9×10^{-12}. This is basically as close as possible to the exact integral.

Introduction
Biochemical Reactors
Forward Euler
Modified Euler Method
Systems of Equations
Second-Order Initial Value Problems
Stiff Differential Equations



9. Initial Value Problems

9.1 Introduction

Within the field of engineering, we are often interested in how a system that is at a know state reacts to changes in a parameter that influences that system. For example, what happens to the current in circuit if there is a change in the resistance across device in the circuit? What happens to the temperature of coolant leaving a radiator if the air temperature changes? How fast is the change? What is the magnitude of the change? Many systems can be described by a differential equation that contains derivatives with respect to time. Typically, the *initial conditions* of the system are known and we are interested in modeling the long term behavior of the system. Problems in this important category are call *initial value problems* and they are the focus of this chapter.

9.2 Biochemical Reactors

In biological systems, enzymes catalyze most of the reactions where one compound or substrate is converted into a product. For example, wine contains ethanol. In some cases, an enzyme from a microorganism can be present that converts the ethanol into acetic acid. If this happens, the wine sours. The conversion of ethanol into acetic acid is controlled and facilitated by an enzyme. The first step is for the enzyme, E, and substrate (ethanol), S to combine and form a complex written $E \cdot S$. In some cases this complex falls appart before a product is formed, but in other cases, the enzyme catalyzes the reaction that leads to the substrate-enzyme complex forming a product (acetic acid), P, and the product quickly separates from the enzyme. This process can be summarized as:

$$E + S \overset{k_1}{\rightleftharpoons} k_{-1} E \cdot S \overset{k_2}{\to} E + P \, .$$

Reactions catalyzed by enzymes are central to much of biochemical engineering and bioprocess engineering. As you can learn in almost any course in those fields, if a process is controlled by the above set of reactions, then we can describe the change in the concentration of substrate in a close system (i.e., a system with no inflow and outflow like a sealed wine bottle) with the equation:

$$\frac{dS}{dt} = \frac{-V_{max} S}{K_m + S}$$

where S is the concentration of substrate, V_{max} is a parameter that describes the maximum reaction rate (i.e., full utilization of all enzymes because high concentration of S is present) and K_m is the substrate concentration at which the reaction rate is half of V_{max}. In order to solve an initial value problem, we seek to determine the substrate concentration, S, that satisfied the equation above, i.e., the rate of change of the substrate is equal to a function that depends on the current concentration.

The equation for $\frac{dS}{dt}$ describes the loss or consumption of S. The rate at which S is consumed must be exactly equal to the rate at which P is produced (i.e., every ethanol molecule that is reacted away forms an acetic acid molecule) so we can also write an equation for the formation of P as:

$$\frac{dP}{dt} = \frac{V_{max}S}{K_m + S}$$

(note the sign change). This model of enzyme kinetics is known as the Michaelis-Menten kinetics model, originally proposed in 1913! Given an initial concentration of substrate S and product P, we can solve the initial value problem to determine the change in these respective concentrations over time. Figure 9.1 shows the change in substrate and product concentration for an initial concentration of 1.0 (dimensionless) for the substrate and a concentration of 0.0 for the product. The ODE's were solved using `scipy.integrate.odeint()`, which is described later in the chapter.

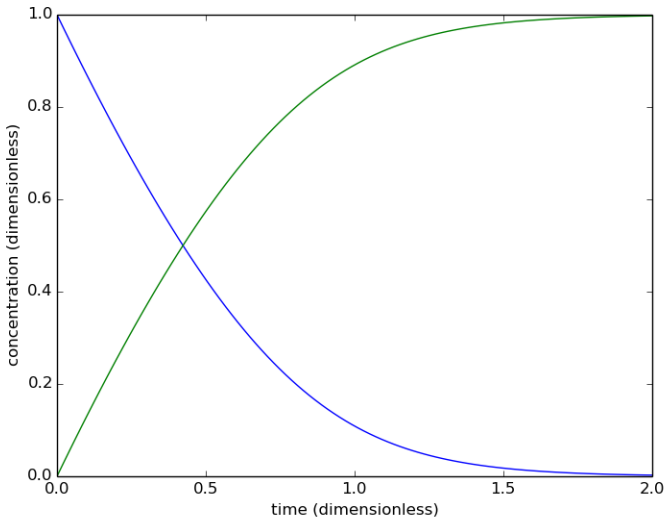


Figure 9.1: Substrate consumption and product formation for a process governed by Michaelis-Menten kinetics.

9.3 Forward Euler

In this section, the forward Euler method is described, and it is the simplest approach for solving an initial value problem. Any initial value problem can be written as:

$$\frac{dy}{dt} = f(y,t) \ ,$$

and the initial condition $y_0 = y(t = 0)$. The key to understanding the process for approximately solving the initial value problem is to simply recognize that the initial value problem gives us: (1) a

starting point, and (2) a slope! We can use these two pieces of information to make an estimate as to where the unknown function, y, is going in the near future. Taking a small step in the direction given by the slope leads to a new time point that is slightly greater than where we started and a new value for y. At this new location, we can again calculate the slope and take a small step in the direction given by the slope. This process is illustrated in figure 9.2. The process described here estimates the future based ONLY on the current conditions. This catagory of time stepping methods for solving initial value problems are called *explicit* methods because the estimates of the future are based explicitly on the present.

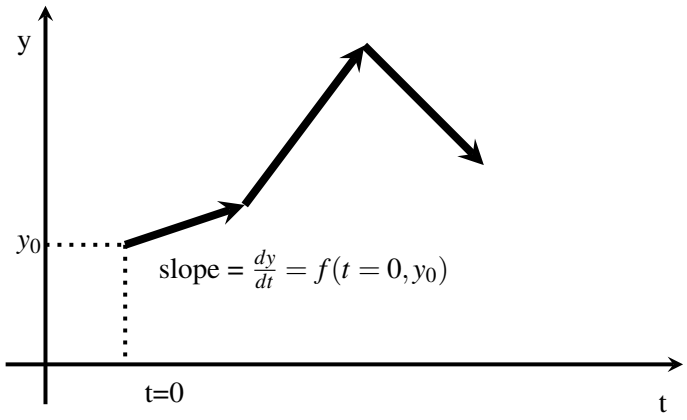


Figure 9.2: An approximate solution to an initial value problem can be obtained by calculating a slope from the ODE equation based on the current values for y and t, and then taking a small step in time to a new set of vaules. This problem is repeated until the desired final time is reached.

The construction of an algorithm that implements the process described above begins with the creation of a function that contains the ODE and returns the slope (i.e., the time derivative) based on a given time, t, and value for the dependent variable, y. The forward Euler process is based on an iteration where the slope is calculated and then, based on the size of the time step, δt, a new value for y is calculated. Recall that the first derivative of a function can be approximated with a finite difference approximation:

$$\frac{dy}{dt} \approx \frac{y_i - y_{i-1}}{\delta t} = f(y_{i-1}, t_{i-1}) \ .$$

The current value of the dependent variable, y_{i-1} and time, t_{i-1}, are used to calculated the slope and solve for the new value of the dependent variable, y_i. The previous equation can be rearranged to give:

$$y_i = y_{i-1} + \delta t \cdot f(y_{i-1}, t_{i-1}) \ ,$$

which is the forward Euler method. The Python script below implements the forward Euler method to predict the change in substrate concentration for an Michaelis-Menten reaction.

```
import numpy
import pylab
```

```
# model kinetic parameters
Vmax = 2.0 # mol/(L s)
Km = 0.5 # mol/L

# ODE definition
def df(s,t):
    dsdt = -Vmax*s/(Km+s)
    return dsdt

# setup time discretization
n = 10 # number of time steps
t = numpy.linspace(0,2.0,n)
dt = t[1]-t[0]

#allocate storage space and set initial conditions
sol = numpy.zeros(n)
sol[0] = 1.0 # initial S in mol/L

for i in range(1,n):
    sol[i] = sol[i-1]+dt*df(sol[i-1],t[i-1])

pylab.plot(t,sol)
pylab.xlabel("time (dimensionless)")
pylab.ylabel("concentration (dimensionless)")
```

The first section of this script contains the ODE function within a separate, callable function. The inputs to this function are the current value for the dependent variable (substrate concentration) and the current time. The function returns the slope, $\frac{ds}{dt}$. The forward Euler algorithm requires that time be discretized into 'small' segments, and, to facilitate this process, the algorithm builds a vector that holds all the time points. The initial condition and storage space for the final solution are then set. The forward Euler iteration is very simple because it calculates a new value for the dependent variable based on the slope and the time step size.

Figure 9.3 shows a plot of the substrate concentration versus time using the forward Euler method with two different time step sizes. The blue curve is based on 100 time steps ($\delta t = 0.02$) and the green curve is based on 4 time steps ($\delta t = 0.5$). The large time steps used for the green curve show how the slope of the approximation is equal to the slope at the beginning of the time step. During the time step, the slope decreases as the substrate concentration is reduced so there is a significant error associated with the large time step. The accuracy of the forward Euler method is of the same order as the forward finite difference approximation of the derivative, i.e., the error is $O(\delta t)$. If the length of a time step is cut in half, the error is also halved. Fortunately, the forward Euler method has a relatively low computational cost so taking a large number of time steps is feasible here, but if we need to simulate a much longer period of time or if we are solving many initial value problems simultaneously, the very small time step requirement quickly becomes an issue. To begin to address this accuracy limitation, we now turn our attention to improving the order of accuracy.

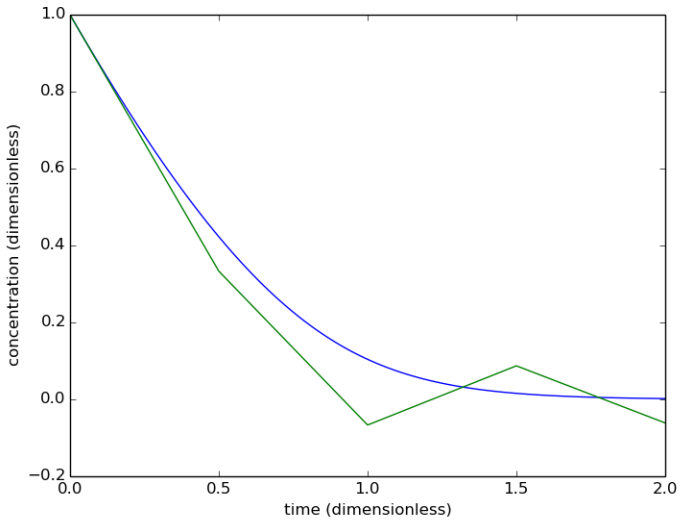


Figure 9.3: Substrate concentration governed by a Michaelis-Menten reaction is modeled using the forward Euler method with two different time step sizes: 100 time steps (blue) and 4 times steps (green).

9.4 Modified Euler Method

The forward Euler method is based on an estimate of the slope based only on the current conditions. One method for improving the accuracy of the forward Euler method is to predict future conditions and then use that prediction to get a better estimate for the slope. For example, we can predict future conditions using the forward Euler method:

$$y_i^* = y_{i-1} + \delta t \cdot f(y_{i-1}, t_{i-1}) \ ,$$

and then use this prediction to estimate the slope at t_i using

$$\frac{dy}{dt} = f(y_i^*, t_i) \ .$$

We now have two estimates for the slope, one at t_{i-1} and one at t_i. Using an average of these two estimates gives us a better estimate of the slope over the time span of interest. Using this principle the modified Euler method calculates a new value for the dependent variable using:

$$y_i = y_{i-1} + 0.5 \cdot \delta t \cdot (f(y_{i-1}, t_{i-1}) + f(y_i^*, t_i)) \ ,$$

where y_i^2 is calculated using the equation above (i.e., forward Euler). The accuracy of modified Euler approach is $O(\delta t^2)$, which leads to significantly smaller errors for small time step sizes.

A Python script that utilizes the modified Euler method for the Michaelis-Menten kinetics problem is given below.

```
import numpy
import pylab

# model kinetic parameters
Vmax = 2.0 # mol/(L s)
Km = 0.5 # mol/L

# ODE definition
def df(s,t):
    dsdt = -Vmax*s/(Km+s)
    return dsdt

# allocate space and set initial condition
n = 100 # number of time steps
t = numpy.linspace(0,2.0,n)
sol = numpy.zeros(n)
sol[0] = 1.0 # initial S in mol/L
dt = t[1]-t[0]

for i in range(1,n):
    est = sol[i-1]+dt*df(sol[i-1],t[i-1])
    sol[i] = sol[i-1]+0.5*dt*(df(sol[i-1],t[i-1])+df(est,t[i]))

pylab.plot(t,sol)
pylab.xlabel("time (dimensionless)")
pylab.ylabel("concentration (dimensionless)")
```

Overall, the algorithm is very similar to the forward Euler method, but it has an extra step in the iteration loop to estimate the value of the dependent variable at the end of the time step.

9.5 Systems of Equations

The enzymatic degradation of a substrate to a product involves changes in concentration to both the substrate and the product. In this and many other systems, a change in one system parameter has an impact on many other system parameters. For these problems, multiple initial value problems must be solved simultaneously. This implies that there is a vector, $\mathbf{y}$, of dependent variable, and a vector of ODEs:

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

Fortunately, the extension of the methods described above to a system of ODEs is trivial. The ODE definition function needs to be modified slightly so that it can receive a vector of dependent variables and it must return a vector of derivatives (or slopes), but otherwise there is little change.

As we move towards systems of equation, the computational costs can increase quickly and it is often advantageous to utilize the initial value problem algorithms available in the `scipy.integrate` library. These algorithms have a number of helpful advantages:

1. high accuracy – often fourth-order ($O(\delta t^4)$) or higher,
2. error checking – comparing the prediction of a fourth-order and fifth-order method (or, in general, comparing the prediction of two methods with different orders of accuracy) each time step allows the algorithm to adjust the time step size in order to maintain a desired level of accuracy, and
3. fast execution – the algorithms are often written in FORTRAN or C and execute faster than a purely Python algorithm.

The simplest initial value problem solver in the Scipy library is `scipy.integrate.odeint()`. This function must be given the name of a function containing the ODEs, the initial condition(s), and the time span for integration. The use of this function for modeling both the substrate and product concentration for a Michaelis-Menten kinetics problem is illustrated in the Python script below.

```
import numpy
import pylab
from scipy.integrate import odeint

# model kinetic parameters
Vmax = 2.0 # mol/(L s)
Km = 0.5 # mol/L

# ODE definition
def df(c,t):
    s = c[0] # substrate concentration
    p = c[1] # product concentration
    dsdt = -Vmax*s/(Km+s)
    dpdt = Vmax*s/(Km+s)
    return numpy.array([dsdt,dpdt])

# initial condition
c0 = numpy.array([1.0, 0.0]) # initial S, P in mol/L
t = numpy.linspace(0,2.0,100)
sol = odeint(df, c0, t)
pylab.plot(t,sol)
pylab.xlabel("time (dimensionless)")
pylab.ylabel("concentration (dimensionless)")
```

Notice that a vector of dependent variables is passed into the function containing the ODEs. Care must be taken to ensure that *the order in which unknowns are located in the vector is the same order used for the derivatives that are returned.* Assigning new variable names to the unknowns in the dependent variable vector can be helpful for keeping track of the various unknowns. Of course, these new variable names are limited to the function and cannot be used outside the function. The figure resulting from the this Python script was shown earlier in the chapter (figure 9.1).

Exercise 9.1 One of the most famous initial value problems is the predator-prey problem. If x is the population of prey, it is typical to assume that the population change is governed by:

$$\frac{dx}{dt} = ax - bxy$$

where a is the birthrate per unit of x and b is the death rate due to the predator with population y. The population of predator is typically assumed to be governed by:

$$\frac{dy}{dt} = cxy - dy$$

where c is the growth rate from the consumption of prey and d is the death rate from overpopulation or age.

Solve the predator prey model equations with $a = 2$, $b = 1$, $c = 1.5$, and $d = 2$, and with an initial population of 1.0 for both the predator and the prey. Plot the population over at least 10 time units. ■

9.5.1 Second-Order Initial Value Problems

Although they are rare in chemical or biological engineering, some initial value problems involve second-order ODEs and two initial conditions. The classic problem in this category is the motion of a body under gravitational force. If x is the position of the body, then Newton's first law states:

$$\frac{d^2x}{dt^2} = -g \ ,$$

where g is gravitational acceleration. Solving this equation requires two initial conditions: an initial position, $x(t = 0)$, and an initial velocity, $\frac{dx(t=0)}{dt}$.

Solving second-order initial value problems is relatively straightforward using the algorithms already described because second-order (or higher-order) problems can be rewritten as systems of first-order equations by defining new variables. For the problem above, defining $v = \frac{dx}{dt}$ allows the second-order problem to be rewritten as a system of two first-order equations:

$$\frac{dx}{dt} = v \ ,$$

$$\frac{dv}{dt} = -g \ ,$$

and notice that one initial condition can be used for each equation: $x(t = 0)$ and $v(t = 0)$.

9.6 Stiff Differential Equations

The forward Euler method and other explicit time stepping methods are very simple and computationally efficient for solving initial value problems, but they have an important limitation. Use the forward Euler method to solve the initial value problem:

$$\frac{dy}{dt} = -25y + 25\sin(t) + \cos(t) \ , \ 0 \leq t \leq 2.0$$

$$y(0) = 1.0$$

with 10 time steps ($\delta t = 0.2$). The solution shown in figure 9.4 was obtained when attempting this approach. This is very, very far from the correct solution. In fact, the error associated with this approximation is growing exponentially, and if we integrate beyond $t = 2.0$, the approximation only becomes worse.

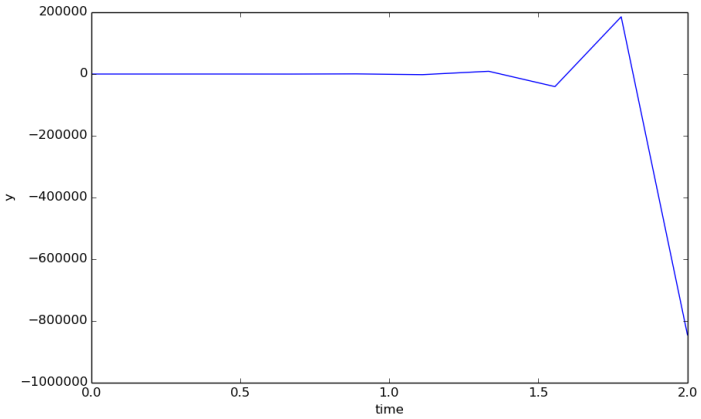


Figure 9.4: The approximate solution resulting from using the forward Euler method on the initial value problem $\frac{dy}{dt} = -25y + 25\sin(t) + \cos(t)$ with $y(0) = 1.0$ and $\delta t = 0.2$. The approximation error is large and growing exponentially.

To help us understand why the forward Euler method failed so badly for this problem, it is helpful to look at the plot of the exact solution shown in figure 9.5. The important thing to recognize about the solution ($y = \sin(t) + e^{-25t}$) is that there are two different time scales present – there is a very fast time scale that causes a rapid, initial decrease in the solution, and then there is a slower time scale associated with the oscillations from the sin() function. *The defining characteristic of stiff differential equations are two or more time scales.* If only the fast time scale existed, the solution would quickly reach steady state and the simulation could focus on the brief period when all the changes occur. If only the slow time scale existed, longer time steps could be used in obtaining an approximate solution. With stiff differential equations, the time step must be small enough to capture the fast time scale events, but those small time steps result in large computational cost associated with simulating the slower, longer time scale.

The use of explicit time stepping methods on a stiff ODE reveal the major weakness of these methods. Recall that explicit methods predict the direction of the solution using only currently available information. An analogy would be to walk around watching only the ground exactly at your feet – based on the topology at your feet, you take a step anticipating that the topology of the ground is not going to change dramatically over that step. For problems with only one time scale, that assumption holds. Stiff ODEs, however, have a slow changing topology (the slow time scale) and a cliff or fast changing topology. When an explicit method hits the fast time scale, it is analogous to stepping off a cliff and exponential error increases result.

A number of different algorithms have been developed to address the challenge of stiff ODEs. The algorithm used by `scipy.integrate.odeint()` is one such algorithm [Hin83], and if this function is used to solve the stiff example problem above, the solution is indistinguishable from the exact solution. Figure 9.5 actually contains two curves – the exact solution and the approximate solution

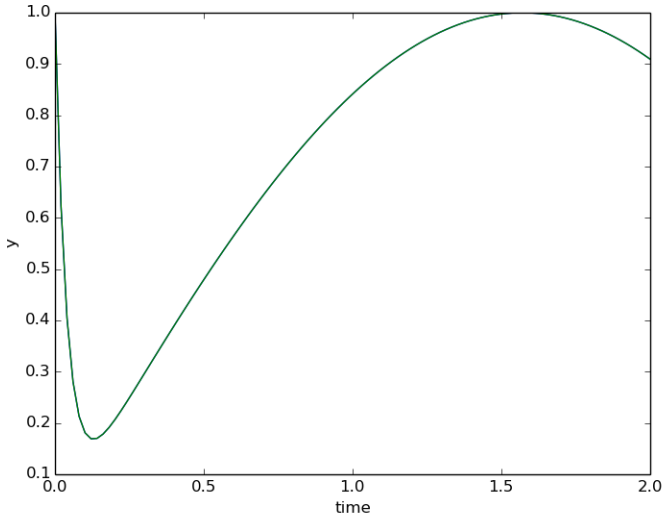


Figure 9.5: The exact solution and the solution resulting from using scipy . integrate . odeint () on the initial value problem $\frac{dy}{dt} = -25y + 25\sin(t) + \cos(t)$ with $y(0) = 1.0$ and $\delta t = 0.2$. The two curves overlap one another.

from scipy . integrate . odeint (). Algorithms designed for stiff ODEs utilize two strategies to handle the multiple time scales within the problem. The first strategy is to continuously check the accuracy of a predicted solution for each time step, and then continuously adapt the size of the time step to the size required to maintain accuracy. By adjusting the time step size, these algorithms can take smaller time steps whenever faster time scales are causing rapid changes and take larger time steps whenever the changes are slow. This first strategy is helpful to minimize the computational costs associated with stiff ODEs, but, by itself, this strategy is not sufficient and a second, critical technique is required.

Explicit time stepping methods for solving initial value problems all have the basic form:

$$y_i = y_{i-1} + F(\delta t, y_{i-1}) \ ,$$

where the predicted solution is only based on current values (or estimates of the future that are still, ultimately, based on current values as is the case of the modified Euler method or the popular, explicit Runge-Kutta methods.) The alternative to explicit time stepping is implicit time stepping where the slope or change in the dependent variables over the next time step is not simply based on current values, but also based on future values. Implicit methods have the basic form:

$$y_i = y_{i-1} + F(\delta t, y_{i-1}, y_i) \ .$$

For example, the simplest implicit method is the backward Euler method, and it has the form:

$$y_i = y_{i-1} + \delta t \cdot f(t_i, y_i$$

for solving an initial value problem of the form: $\frac{dy}{dt} = f(t,y)$. Notice that the unknown variable, y_i, now appears in multiple places within the equation and solving for the unknown likely requires

solving a *nonlinear* equation. For multiple ODEs, we have to solve a system of nonlinear equations. The development of algorithms that use implicit time stepping requires utilization of the methods covered in the chapter on nonlinear equations – typically, these algorithms use Newton's method to solve the nonlinear equations. Fortunately, robust algorithms have been developed by others and should be utilized whenever possible. A number of different algorithms that should meet most requirements are available through the `scipy.integrate.ode()` function (`http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html`).

Problem 9.1 Model the concentration of reactant A in a stirred tank reactor with two inputs. The only reaction is

$$A \rightarrow B \ .$$

This reaction can be described by the first-order reaction equation $-r_A = kC_A$ where $-r_A$ is the rate at which A is consumed in $\frac{molA}{/}L \cdot s$, C_A is the concentration of A in mol/L and $k = 0.35s^{-1}$ is the rate constant. There are two input streams into the reactor: the first input has a flow rate of $Q_1 = 10\ L/min$ and $C_{A,1} = 2molA/L$ and the second input stream is turned on at $t = 0$ and has a flow rate of $Q_2 = 8L/min$ and a concentration of $C_{A,2} = 5molA/L$. The concentration of A in the tank is governed by the equation:

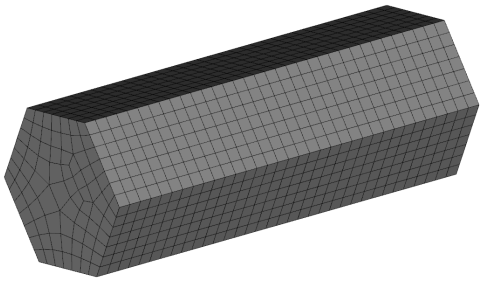
$$\frac{dC_A}{dt} = [C_{A,1}Q_1 + C_{A,2}Q_2 - (Q_1 + Q_2)C_A]/V - kC_A$$

where $V = 50L$ is the volume of the reactor. Before $t = 0$, the reactor is operating at steady-state, i.e., $dC_A/dt = 0$, and $Q_2 = 0$ so the above equation simplifies to

$$(C_{A,1} - C_A)Q_1/V = kC_A \ ,$$

which can be solved to establish that at $t = 0$, $C_A = 0.73mol/L$. This is the initial condition that should be used to model the concentration of A in the reactor after $t = 0$. Write a Python script to model this process and plot C_A as a function of time.

Introduction
Shooting Method
Finite Difference Method



10. Boundary Value Problems

This chapter continues our exploration of numerical methods to solve ordinary differential equations (ODEs), which have derivatives with respect to a single independent variable. The focus here is on problems that have a second-order derivative so two conditions are required to determine a unique solution. In the previous chapter, the derivatives were generally with respect to time, and second-order problems had two initial conditions – one on the dependent variable and one on the first derivative of the dependent variable. In this chapter, the derivatives will typically be with respect to space, and the two conditions used to determine a unique solution will be at either end of the spatial domain that is being modelled.

10.1 Introduction

Boundary value problems (BVPs) frequently arise in chemical and biological engineering. A general form for a linear BVP equation is

$$\frac{d^2y}{dx^2} + a(x)\frac{dy}{dx} + b(x)\cdot y = c(x)\ .$$

Solving a boundary value problem requires finding a function, $y(x)$, that satisfies this equation (i.e., the second-derivative of y plus $a(x)$ times the first derivative plus $b(x)$ times the solution, $y(x)$ is equal to a given right-hand side. Determining a unique solution requires two addition conditions on y. Basically, we need to integrate this equation twice, which gives two constants of integration, and we need the two additional conditions to solve for the two constants of integration. For some problems, $y(x)$ is known at one or both ends of the domain. This boundary condition is referred to as a Dirichlet or essential boundary conditions. For example, if we are solving the boundary value problem above on the domain $[0,1]$ and it is known that $y(0) = 3.2$, this would be considered a Dirichlet condition. If Dirichlet conditions are given for both ends of the domain, i.e., $y(1)$ is also known, then an approximate solution can be determined. The other common possibility is that $\frac{dy}{dx}$ is known at one or both ends of the domain. This type of boundary condition is known as a Neumann boundary condition.

Boundary value problems arise when describing diffusion processes, when modeling conductive heat transport, and when calculating viscous fluid flow. There are many other settings in which these types of equations can potentially arise, but problems involving conservation of mass, energy, and momentum are the most common in chemical and biological engineering. In this chapter, two different numerical approaches will be examined for solving boundary value problems – the shooting method and the finite difference method. In both cases, we will utilize numerical techniques that were covered previously, so be prepared to review material from early chapters as needed.

10.2 Shooting Method

The previous chapter discussed an approach for calculating the location of a projectile under just the force of gravity given two initial condition: an initial location and an initial velocity (recall that velocity is just the first derivative of location, $\frac{dx}{dt}$). The equation describing the motion of the projectile was a second-order ODE, but the problem fell into the category of initial value problems because it included two conditions at the same boundary (i.e., the $t = 0$ boundary) instead of one condition at each boundary. BVPs have a condition at each boundary, analgous to solving the projectile motion problem given a starting location and an ending location but no initial velocity. Without two initial conditions, we cannot simply reuse the methods covered in the previous chapter, but, with a little creativity, we can recycle much of what was developed previously and adapt it to BVPs.

Imaging that we are solving the projectile motion problem, given an initial location and a target location, and we need to determine an initial velocity as well as the particle location and velocity between the launching and target locations. We could guess an initial velocity, take a shot, and determine the final distance from the given target. We could then take a second shot with a different initial velocity, and once again measure the distance to the target. Using these two shots as reference points, we could then interpolate (or extrapolate) to determine a better estimate for the required initial velocity to hit the target. Repeating this process for 3 or 4 shots would hopefully lead to us hitting the target. This process is effectively 'the shooting method' for BVPs. We guess one initial value, use all the initial value methods from the last chapter, check to see if we matched the other boundary condition, and repeat the process until our guess at the unknown initial value results in us matching the second boundary condition.

It is easiest to examine the algorithm for the shooting method through an example problem. We want to solve the BVP:

$$\frac{d^2y}{dx^2} = 4(y-x)$$

on the domain, $0 \leq x \leq 1$, and with the boundary conditions, $y(0) = 0$ and $y(1) = 2$. The first step is to rewrite the equation as a system of first order equations by introducing a new variable:

$$\frac{dy}{dx} = y_1$$

$$\frac{dy_1}{dx} = 4(y-x)$$

where an initial condition is available for the first equation ($y(0) = 0$), but not the second equation. The shooting method requires a guess for the second boundary condition, and then we can solve the system of equations using any of the algorithm presented in the initial value problem chapter

(the scipy . integrate . odeint () function is recommended) and check to see if the other boundary condition ($y(1) = 2$) is satisfied. Figure 10.1 shows the solution using a guess of $y_1(0) = 1.0$. With this guess, the target of 2.0 is missed and $y(1) = 1.0$ is hit instead.

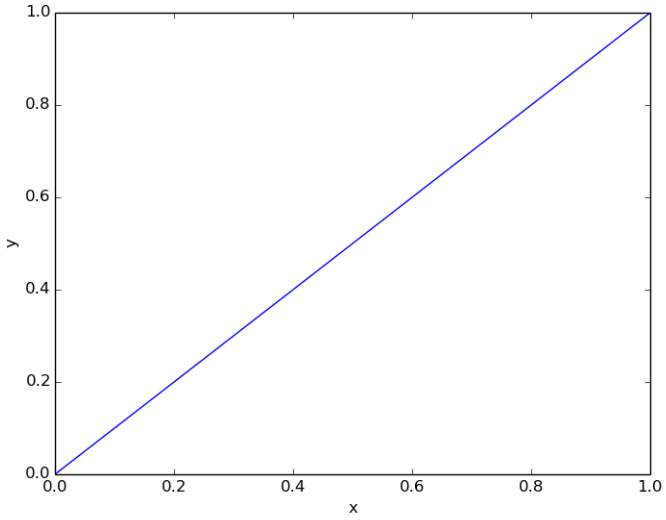


Figure 10.1: Approximate solution to the BPV, $\frac{d^2y}{dx^2} = 4(y-x)$ using a guess of $\frac{dy(0)}{dx} = y_1(0) = 1.0$.

For the second shot, an initial guess of $y_1(0) = 0.0$ is used, and the approximate solution using this guess is shown in figure 10.2. In this case, the target is missed by a greater amount as $y(1) = -0.813$.

Now that two shots have been taken and the two misses have been measured, we need a method for determining a better guess for the second boundary condition on $y_1(0)$. The simplest approach is to fit the two previous results with a line and extrapolate to determine a better guess. If γ_0 is the first guess at the boundary condition, γ_1 is the second guess, and β is the desired target value, then an improved guess is available using:

$$\gamma = \gamma_1 - \frac{\gamma_1 - \gamma_0}{(y_{\gamma_1}(1.0) - y_{\gamma_0}(1.0))}(y_{\gamma_1} - \beta) \; .$$

An iterative process is used where the improved estimate for the boundary condition, γ, replaces the older guess, γ_0, and the process is repeated.

A Python script that uses the shooting method to solve the example problem is given below.

```
import math
import numpy
import pylab
from scipy.integrate import odeint

# Split y'' = 4*(y-t) into
```

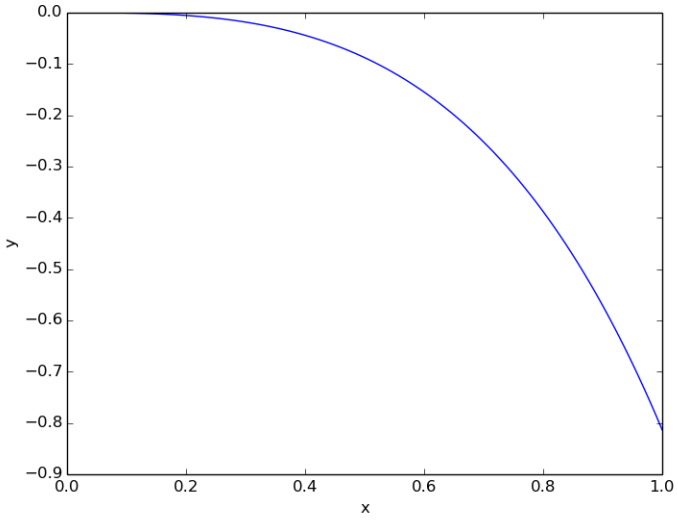


Figure 10.2: Approximate solution to the BPV, $\frac{d^2y}{dx^2} = 4(y-x)$ using a guess of $\frac{dy(0)}{dx} = y_1(0) = 0.0$.

```
# y0' = y1 and
# y1' = 4*(y-t)
def dfdt(y, t):
    dy0dt = y[1]
    dy1dt = 4.0*(y[0]-t)
    return numpy.array([dy0dt, dy1dt])

def exact(t):
    coeff = 0.13786
    sol = coeff*(numpy.exp(2.0*t) - numpy.exp(-2.0*t)) + t
    return sol

TOL = 1e-6
t = numpy.linspace(0.0,1.0,100)
alpha = 0.0
beta = 2.0
gamma0 = 1.0
gamma1 = 0.0

# first shot, use bc for y, set other to 0.0
yinit1 = numpy.array([alpha,gamma0])
y1 = odeint(dfdt, yinit1, t)
# get impact point for first shot
# note this gets the last row, first column entry
end1 = y1[-1,0]
```

```
print "Error with shot: ", math.fabs(beta-end1)

for i in range(20):
    # second shot, set bc for y to 0.0, other uses 1.0
    yinit2 = numpy.array([alpha,gamma1])
    y2 = odeint(dfdt, yinit2, t)
    end2 = y2[-1,0]
    print "Error with shot: ", math.fabs(beta-end2)
    if math.fabs(beta-end2) < TOL:
        break

    gamma = gamma1-((end2-beta)*(gamma1-gamma0)/(end2-end1))
    gamma0 = gamma1
    gamma1 = gamma
    end1 = end2

pylab.plot(t,y2[:,0])
pylab.plot(t,exact(t))
pylab.xlabel('x')
pylab.ylabel('y')
```

The Python script includes an additional function that contains the exact solution so that the approximate solution from the shooting method can be compared to the exact solution. One feature of numpy that is used in this script but has not been covered previously is that it is possible to access the last element in a row or column of an array using the index -1. For example, the reference y[-1] in numpy will return the last entry in the vector. This feature is used here to get the last row entry in the first column of the solution array to determine the value of y at the far boundary, i.e., $y(1)$ so that we can calculate the distance to the target, β. Figure 10.3 shows both the final result from the shooting method and the exact solution. The lines are so close that they are indistinguishable from each other. For linear BVPs like this, only 3 iterations should be required because the linear extrapolation used to determine an improved guess, γ, should yield the exact value to use. Nonlinear BVPs may require 4 or 5 iterations to determine an acceptable value for γ.

The shooting method is a common choice for boundary value problems when an efficient and familiar ODE initial value problem solver is available and Dirichlet boundary conditions are given. In general, however, it is not the most common choice for BVPs in the author's experience. A more intuitive and flexible approach, capable of solving problems with one or two Neumann boundary conditions, is presented in the next section (and the next chapter): the finite difference method.

Exercise 10.1 Use the shooting method to solve the boundary value problem:

$$\frac{d^2y}{dx^2} = -y$$

on the domain $0 \le x \le \frac{\pi}{4}$ with $y(0) = 1$ and $y(\pi/4) = 1.0$. ■

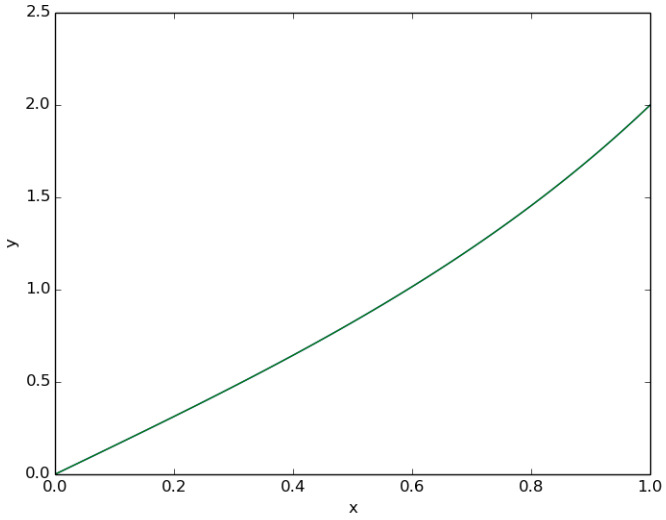


Figure 10.3: Approximate solution to the BPV, $\frac{d^2y}{dx^2} = 4(y-x)$ using a guess of $\frac{dy(0)}{dx} = y_1(0) = 1.551$.

10.3 Finite Difference Method

The finite difference method is based on the idea of replacing the derivatives in a differential equation with algebraic approximations of those derivatives at discrete points distributed throughout the domain of interest. A general form for a linear boundary value problem is

$$\frac{d^2y}{dx^2} + a(x)\frac{dy}{dx} + b(x)\cdot y = c(x) \ ,$$

on the domain $a \leq x \leq b$ with boundary condition given at a and b. The finite difference method begins by dividing the domain, $a \leq x \leq b$ into a sequence of evenly spaced, discrete points called nodes with their location given by x_i, as shown in figure 10.4. If h is the distance between the nodes, $h = x_i - x_{i-1}$, and N is the number of intervals between nodes (i.e., there are $N+1$ nodes that are numbered from 0 to N), then the location of each node can be calculated using $x_i = a + ih$.

Recall that the second derivative can be approximated at location x_i with

$$\frac{d^2y}{dx^2} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \ ,$$

where h is the distance between the nodes, $h = x_i - x_{i-1}$. Similarly, the first derivative can be approximated with

$$\frac{dy}{dx} \approx \frac{y_{i+1} - y_{i-1}}{2h} \ .$$

Using these two approximations for the derivatives, the original BVP equation can be replaced with an algebraic approximation at every node in the domain (i.e., we are replacing a differential equation with many (i.e., $N+1$) algebraic equations). The algebraic approximation is

$$\left(\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}\right) + a(x_i)\left(\frac{y_{i+1} - y_{i-1}}{2h}\right) + b(x_i)y_i = c(x_i)$$

Notice that we have transformed the problem from a differential equation into a large system of linear algebraic equations. The unknowns are the values of y_i at every node.

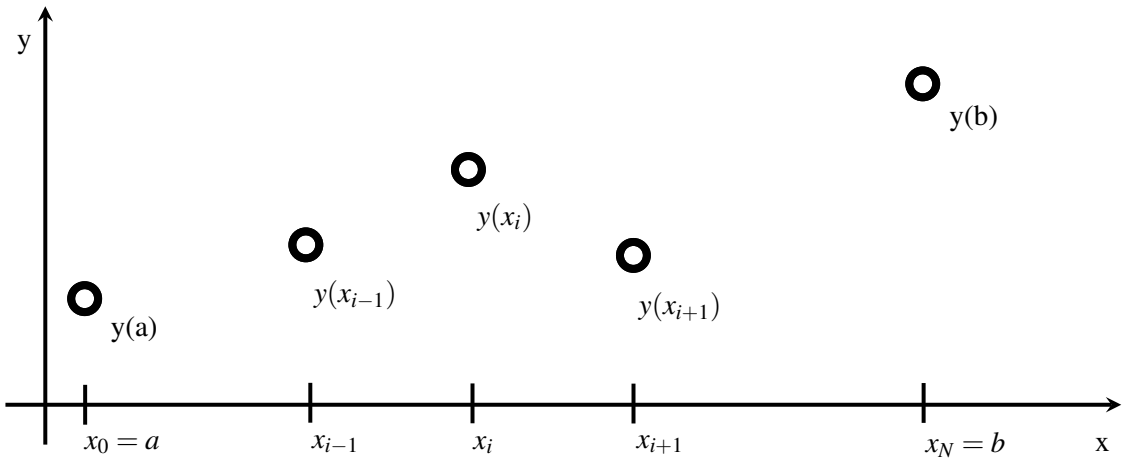


Figure 10.4: Using the finite difference method requires dividing the domain, $a \leq x \leq b$ into a set of discrete points or nodes with their locations given by x_i. The goal of the approach is to determine the approximate solution, y_i, at every node.

The finite difference algorithm for linear boundary value problems has 3 sections:

1. The setup phase involves specifying the number of intervals between nodes, N, which translates into $N+1$ nodes numbered from 0 to N, the size of the domain by specifying a and b, and calculating the node spacing, h. The setup phase also typically includes building a vector containing the locations of the nodes, and allocating space for later storing the matrix and right-hand-side associated with the linear system of equations.
2. The middle section of the code is a loop through each node and adding the appropriate coefficients into the matrix and right hand side. The details of this step are summarized below, but it is important to note that nodes located at the boundary have boundary conditions that need to be handled separately.
3. The last section of the algorithm involves solving the linear matrix problem (using, for example, numpy.linalg . solve ()) and plotting the solution.

The finite difference equation at every node has the form:

$$\left(\frac{y_{i+1}-2y_i+y_{i-1}}{h^2}\right)+a(x_i)\left(\frac{y_{i+1}-y_{i-1}}{2h}\right)+b(x_i)y_i=c(x_i)\ ,$$

which is typically rewritten as

$$(y_{i+1}-2y_i+y_{i-1})+\frac{h}{2}\cdot a(x_i)\,(y_{i+1}-y_{i-1})+h^2b(x_i)y_i=h^2c(x_i)\ .$$

This equation exists at every node, and the resulting system of equations can be written as a matrix

problem with the following form:

$$\begin{bmatrix} -2+h^2b(x_1) & 1.0+\frac{ha(x_1)}{2} & 0 & \cdots \\ 1.0-\frac{ha(x_2)}{2} & -2+h^2b(x_2) & 1.0+\frac{ha(x_2)}{2} & 0 \\ 0 & 1.0-\frac{ha(x_3)}{2} & -2+h^2b(x_3) & \ddots \\ \vdots & 0 & 1.0-\frac{ha(x_4)}{2} & \ddots \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} h^2c(x_1) \\ h^2c(x_2) \\ \vdots \\ h^2c(x_N) \end{bmatrix}$$

It is important to note that the above linear matrix system does NOT include boundary conditions (or an equation for node 0) and is only intended to give the basic structure of the linear system. If the boundary condition $y(x_0) = 1.0$ is given, then instead of having a finite difference equation for the first node, which corresponds to the first row in the matrix equation, the boundary condition equation would be used instead and the matrix problem would become:

$$\begin{bmatrix} 1.0 & 0 & 0 & \cdots \\ 1.0-\frac{ha(x_1)}{2} & -2+h^2b(x_1) & 1.0+\frac{ha(x_1)}{2} & 0 \\ 0 & 1.0-\frac{ha(x_2)}{2} & -2+h^2b(x_2) & \ddots \\ \vdots & 0 & 1.0-\frac{ha(x_3)}{2} & \ddots \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} 1.0 \\ h^2c(x_1) \\ \vdots \\ h^2c(x_N) \end{bmatrix}$$

The Python script below uses the finite difference method to solve the boundary value problem:

$$\frac{d^2y}{dx^2} + y = 0$$

on the domain $0 \leq x \leq \pi/2$ with the Dirichlet boundary conditions $x(0) = 1.0$ and $x(\pi/2) = 1.0$.

```
import numpy
from numpy.linalg import solve
import pylab

N = 9 # number of intervals
x = numpy.linspace(0,numpy.pi/2.0,N+1)
h = x[1]-x[0]

# Allocate space
A=numpy.zeros((N+1,N+1))
b = numpy.zeros(N+1)

# Boundary condition at x=0
A[0,0] = 1.0
b[0] = 1.0

for i in range(1,N):
    A[i,i-1] = 1.0
    A[i,i] = -2.0 + h**2
    A[i,i+1] = 1.0
```

```
    b[i] = 0

# Boundary condition at x = pi/2.0
A[N,N] = 1.0
b[N] = 1.0

y = solve(A,b)
pylab.plot(x,y)

pylab.plot(x, numpy.cos(x)+numpy.sin(x))
pylab.xlabel('x')
pylab.ylabel('y')
```

The equations associated with the boundary conditions receive specially handling in this algorithm, but all the other finite difference equations are handled with an iterative loop. For this particular problem, an exact solution is available and this solution is plotted on the same figure as the approximate solution. Even with only 10 nodes, the finite difference approximation is almost indistinguishable from the exact solution as shown in figure 10.5.

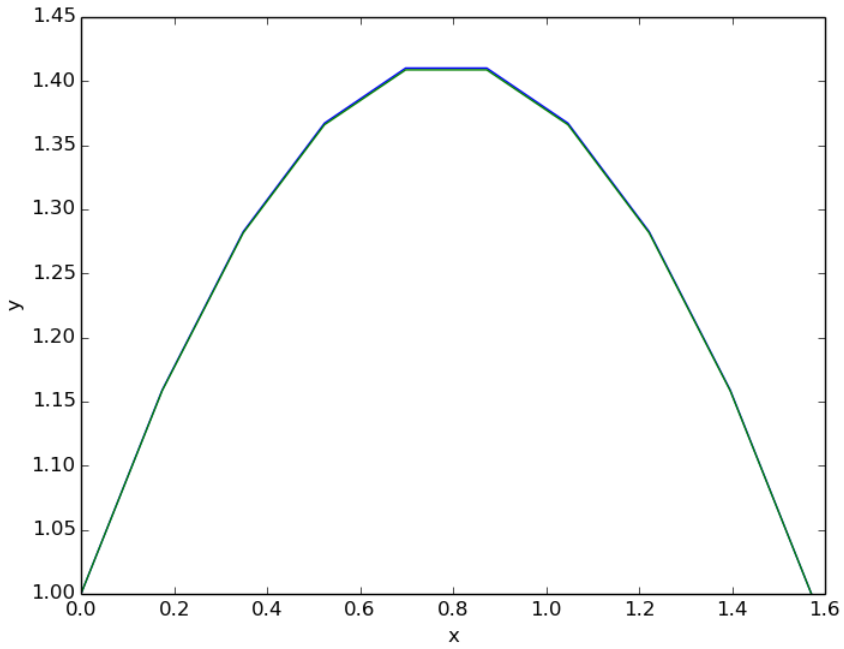


Figure 10.5: Approximate solution to the BPV, $\frac{d^2y}{dx^2} + y = 0$ using the finite difference method with 10 nodes. The exact solution is also plotted (green) and is almost indistinguishable from the approximate solution.

Exercise 10.2 Use the finite difference method to solve the boundary value problem:

$$\frac{d^2y}{dx^2} = -\cos(x) - \sin(x)$$

on the domain $0 \leq x \leq \pi/2$ and with the boundary conditions $y(0) = 0$ and $\frac{dy(\pi/2)}{dx} = -1.0$. Note that this problem has the same solution as the example problem in the finite difference section above, $y = \cos(x) + \sin(x)$.

The one complication with this problem is the Neumann boundary condition at $x = \pi/2$. The derivative in this boundary condition can be approximated as

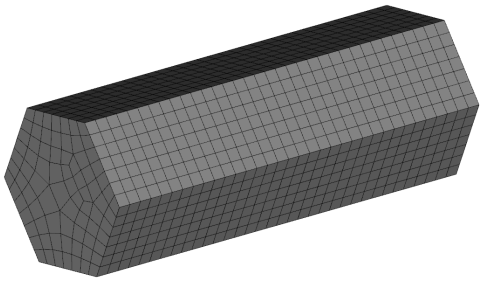
$$\frac{y_{x_N} - y_{x_{N-1}}}{h} = -1.0 \; .$$

This equation becomes the last row in the linear matrix problem, which has the form:

$$\begin{bmatrix} \ddots & 1.0 + \frac{ha(x_{N-2})}{2} & 0 \\ 1.0 - \frac{ha(x_{N-1})}{2} & -2 + h^2 b(x_{N-1}) & 1.0 + \frac{ha(x_{N-1})}{2} \\ 0 & -1.0 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} y_{N-2} \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \\ -h \end{bmatrix}$$

■

Finite Difference Method for Steady-State PDEs
Setup
Matrix Assembly
Solving and Plotting
Including Convection
Finite Difference Method for transient PDEs



11. Partial Differential Equations

A goal in developing a mathematical model is to create the simplest possible model that captures the features of interest in the system. In many cases, the variables we are interested in calculating are primarily changing in time and spatial variation can either be ignored or captured through a lumping parameter. For example, in a well-mixed reactor or even a region of the body, it is often possible to develop useful and informative models that include only time derivatives and not spatial derivatives. These models are typically initial value problems. In other cases, there are important spatial variations in temperature or concentration, but because the system is continuous and nearly steady-state, the small temporal variations can be ignored. For the case of a membrane or a large slab, it is often sufficient to only model spatial changes in one direction, which leads to a boundary value problem. However, in other cases it is necessary to examine variation of the quantity of interest in multiple dimension, which leads to Partial Differential Equations (PDEs). These equations have 2 or more independent variables, and they have derivatives with respect to each of these variables.

11.1 Finite Difference Method for Steady-State PDEs

The finite difference method was developed in the previous chapter for the solution of boundary value problems. The objective of this chapter is to extend this method to multiple dimensions and PDEs. This category of problem is still called boundary value problems, but algorithm development is much more complex and fraught with pitfalls when multiple dimensions are involved. Finite difference algorithms for PDEs are more complex than any algorithm previously developed in this book. The presentation here will break the full algorithm into its major sections and each section will be discussed independently. The primary example used in this chapter is the solution to Laplace's equation in 2-dimensions:

$$\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} = f(x,y) \ .$$

Solving this equation requires the determination of the function c such that its second derivative with respect to x plus its second derivative with respect to y is equal to $f(x,y)$, and c must satisfy

the required boundary conditions. This equation describes diffusion through solids and biological tissues, and it describes conductive heat transport through a stationary medium. At the end of this section, a second example is presented that includes a convective flux term.

11.1.1 Setup

Finite difference algorithms typically begin with a setup phase where the size of the domain and the number of nodes are specified. The algorithm developed here is based on M intervals (i.e., $M+1$ nodes) in the x-direction and N intervals in the y-direction. Ideally, the spacing between the nodes in each direction is equal, but that is not required. What is required is that the spacing between the nodes in each direction be *uniform*. Figure 11.1 shows a typical node in the domain. If i is used as an index for nodes in the x-direction, and j is used as an index for nodes in the y-direction, then the center node in this figure is $node(i, j)$. The four nearest neighboring nodes are also shown in the this figure. As we will see below, each node is effectively coupled to these 4 nearest neighbors when we use the finite difference approximation presented later.

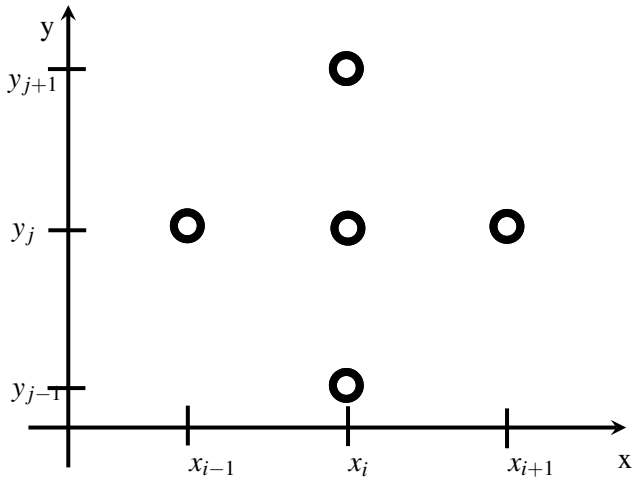


Figure 11.1: If the finite difference method is used for a two dimensional boundary value problem, each node is connected to its four nearest neighbors.

The first section of Python code for the 2-dimension Laplace problem using finite differences is shown below.

```
import numpy
from numpy.linalg import solve
import pylab

M = 15 # intervals in x-direction
N = 15 # intervals in y-direction
east = 0.0 # edge locations
west = 1.0
south = 0.0
north = 1.0
```

```
# node locations
x = numpy.linspace(east,west,M+1)
y = numpy.linspace(south,north,N+1)
h = x[1]-x[0]
k = y[1] - y[0]
h2k2 = h**2 / k**2
coeff = 2*(h2k2+1)
X,Y = numpy.meshgrid(x,y)
#pylab.plot(X,Y,'o')

# Allocate space
totalNodes = (M+1)*(N+1)
A=numpy.zeros((totalNodes,totalNodes))
b = numpy.zeros(totalNodes)
```

The scipy.linalg.solve() function is imported because it is used later for solving the linear matrix problem that is the result of the finite difference approximation. The variables M and N are set to the number of intervals between the nodes in the x- and y-directions, respectively. Larger values for M and N will give a more accurate approximate solution, but values larger than about 100 will have a significant computational cost. The total number of nodes in the domain is $(M+1)(N+1)$ so $M = 100$ and $N = 100$ will result in over 10,000 nodes and a linear matrix problem with over 10,000 unknowns. This is about the largest acceptable matrix size for a solver based on Gaussian elimination. The variables *east*, *west*, *south*, and *north* are used to specify the location of the boundaries of the rectangular domain of the problem. Some authors of numerical algorithms find those variable names to be intuitive while others dislike (hate) those variable names because they cannot remember which direction is east and which is west, for example. Different variable names are, of course, acceptable. For example, some algorithm writers use the variables $left$, $right$, $bottom$, and top instead or even a, b, c, and d.

The next stage in the setup phase of this algorithm is to build vectors holding the locations of the nodes in the x- and y-directions. Based on these node locations, the spacing between the nodes, h and k, can be calculated. The variables $h2k2$ and $coeff$ are needed later in the algorithm. The numpy.meshgrid() function converts the vectors x and y into 2D arrays that hold the x- and y-location of every node. These arrays are only needed for plotting the solution at the end of the algorithm although they can be used here to generate a plot with the locations of the nodes included (this line is currently commented out). The final step in the setup phase is to calculate the total number of nodes and, hence, the total number of unknowns in the linear matrix problem. Once this is known, space for the matrix and right-hand-side vector can be allocated.

11.1.2 Matrix Assembly

The model problem that is used here is

$$\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} = 0$$

on the domain $0 \leq x \leq 1.0$ and $0 \leq y \leq 1.0$ and $c = 0$ on all boundaries except for the east (left) boundary, which has $c = \sin(y)$. Approximating the derivatives with centered finite difference

equations leads to the algebraic equation

$$\frac{c_{i+1,j}-2c_{i,j}+c_{i-1,j}}{h^2}+\frac{c_{i,j+1}-2c_{i,j}+c_{i,j-1}}{k^2}=0$$

and multiplying this equation by k^2 leads to

$$-2\left[\left(\frac{h}{k}\right)^2+1\right]c_{i,j}+c_{i+1,j}+c_{i-1,j}+\left(\frac{h}{k}\right)^2(c_{i,j+1}+c_{i,j-1})=h^2f(x,y)=0\ .$$

This algebraic, finite difference approximation of the original PDE exists at each node and makes up one row in the linear matrix problem. The equation includes coefficients for five unknowns, reflecting that each node, $c_{i,j}$ is connected to its four nearest neighbors (north, east, south and west).

Assembly of the linear matrix problem begins with an outer-loop through each row of nodes in the y-direction followed by an inner-loop through each column of nodes in the x-direction. The combination of these two loops is that each node in the domain is visited once – starting in the lower left corner (southwest corner), proceeding to the right across the first row, then up to the next row, and then next until we end in the upper right corner (northeast corner). At each node, the appropriate coefficients from the equation above are added to the matrix.

Before examining the algorithm for assembling the matrix, it is important to recognize that we need a mechanism for mapping node(i,j) to a unique row in the matrix (i.e., to a unique unknown or node number). In the previous chapter where we examined 1-dimensional problems this was trivial because node i corresponded to the i^{th} unknown and i^{th} row of the matrix. Now, however, we have node (i,j) due to the 2-dimensional nature of the problem. (At this point, some student suggest using a 3-dimensional matrix, but this is not the correct solution.) In order to uniquely map node (i,j) to a particular unknown number, we will use the equation

$$node=j*(M+1)+i$$

where $i=(0,\ldots,M)$ and $j=(0,\ldots,N)$. So for node (0,0) we get $node=0$, for node (0,1) we get $node=M+1$, and for node (M,N) we get $node=N*(M+1)+M=(N+1)*(M+1)-1$. This same equation can be used to calculate the unique node number of the neighboring nodes to the north, east, south, and west.

The section of Python code below assembles the matrix for the test problem.

```
for j in range(0,N+1):
    for i in range(0,M+1):
        node = j*(M+1)+i
        Enode = j*(M+1)+i+1 # node to the east
        Wnode = j*(M+1)+i-1 # node to the west
        Snode = (j-1)*(M+1)+i # node to the south
        Nnode = (j+1)*(M+1)+i # node to the north

        if (i == 0): # check for west boundary
            A[node,node] = 1.0
            b[node] = numpy.sin(numpy.pi*y[j])
        elif (i == M): # check for east boundary
            A[node,node] = 1.0
```

```
            b[node] = 0.0
        elif (j == 0): # check for south boundary
            A[node,node] = 1.0
            b[node] = 0.0
        elif (j == N): # check for north boundary
            A[node,node] = 1.0
            b[node] = 0.0
        else:
            A[node,node] = -coeff
            A[node,Enode] = 1.0
            A[node,Wnode] = 1.0
            A[node,Snode] = h2k2
            A[node,Nnode] = h2k2
            b[node] = h**2 * 0.0
```

This section of code consists of two nested loops that cause us to ultimately loop through every node in the domain. The numbers of the node and its neighbors are calculated first. Then, the algorithm checks to see if a node is on the boundary. It is important to do this first because if a node is on the boundary, then one or two of its neighbors does not exist and trying to write entries into the matrix for nodes that do not exist will only lead to crashes and error messages. If the node is on the boundary, the appropriate boundary condition can be applied (in this case, all boundary conditions are Dirichlet conditions and all are zero except for the east boundary). Finally, if the node is not on the boundary, the appropriate values, based on the finite difference equation above, are added in to the matrix and right-hand-side.

11.1.3 Solving and Plotting

The final code segment solves the linear matrix problem (scipy.linalg.solve() is recommended), and the result is returned as a 1-dimensional vector. The contour plotting function in Matplotlib typically requires that the solution data be in an array with the same shape as the arrays (X and Y) hold the locations of the nodes for the 2-dimensional finite difference mesh. Each row in the array corresponds to a row of nodes in the domain. The solution vector can be reshaped using the numpy.resphape() function before the plotting function is called. Depending on the plotting routine that is used, additional labels and color bars may be helpful. The Python code segment below finalizes the process of solving the Laplace problem in 2-dimensions using finite differences. The resulting contour plot is shown in figure 11.2

```
z = solve(A,b)
Z = z.reshape(M+1,N+1)
pylab.contour(X,Y,Z)
pylab.colorbar()
pylab.xlabel('x')
pylab.ylabel('y')
```

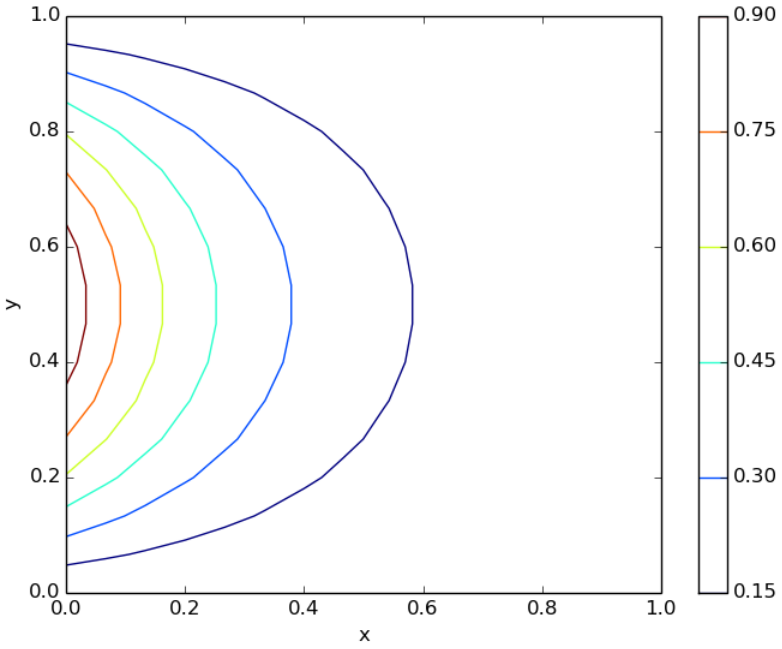


Figure 11.2: Contours of the solution for the model problem.

11.2 Including Convection

The Laplace problem examined in the first part of this chapter is used to model diffusion or heat conduction in stagnant domains. If there is also fluid movement or convection in addition to the diffusion or conduction, then an additional term is required in the model equation. To model steady-state diffusion and convection in 2-dimensions with the fluid velocity being given by the vector (u,v), the following equation is used

$$\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} - u \cdot \frac{\partial c}{\partial x} - v \cdot \frac{\partial c}{\partial y} = 0$$

Using the same boundary conditions and domain as before, and approximating the second-order derivatives with centered finite difference equations and the first-order derivatives with backward finite difference equations leads to the algebraic equation

$$\frac{c_{i+1,j} - 2c_{i,j} + c_{i-1,j}}{h^2} + \frac{c_{i,j+1} - 2c_{i,j} + c_{i,j-1}}{k^2} - u \cdot \frac{c_{i,j} - c_{i-1,j}}{h} - v \cdot \frac{c_{i,j} - c_{i,j-1}}{k} = 0$$

It is very important to recognize that less accurate, backward difference equations were used here instead of the more accurate centered difference equations for first-order derivatives. When applying a finite difference approximation to the convective term, it is important that the approximation of the derivative be done in such a way as to include only the nodes that are upwind of the node of interest. The mathematical justification for this choice is available in a number of excellent books, but here we will simply state that not using an upwind difference approximation usually leads to numerical instability and significant error in the approximate solution. An analogy of questionable

accuracy is that it is difficult to detect changes in smell (concentration) when facing downwind but it is relatively simple when facing upwind. The finite difference equation above is based on the assumption that both components of the wind velocity, (u,v), are positive. If one or both of these velocities is negative, then the finite difference equation must be modified to use first-derivative approximations in the upwind direction.

Adding convection to the previous 2-dimensional finite difference code is relatively simple and only requires: (1) adding a wind velocity variable, and (2) adding additional terms to the matrix for the convective part of the equation. The Python script for the test problem with a fluid velocity vector of $(5.0,5.0)$ is reproduced completely below.

```
import numpy
from numpy.linalg import solve
import pylab

M = 15 # intervals in x-direction
N = 15 # intervals in y-direction
east = 0.0 # edge locations
west = 1.0
south = 0.0
north = 1.0
# node locations
x = numpy.linspace(east,west,M+1)
y = numpy.linspace(south,north,N+1)
h = x[1]-x[0]
k = y[1] - y[0]
h2k2 = h**2 / k**2
coeff = 2*(h2k2+1)
X,Y = numpy.meshgrid(x,y)
#pylab.plot(X,Y,'o')

# Wind
# both MUST be positve due to differencing
wind = numpy.array([5.0,5.0])

# Allocate space
totalNodes = (M+1)*(N+1)
A=numpy.zeros((totalNodes,totalNodes))
b = numpy.zeros(totalNodes)

for j in range(0,N+1):
    for i in range(0,M+1):
        node = j*(M+1)+i
        Enode = j*(M+1)+i+1 # node to the east
        Wnode = j*(M+1)+i-1 # node to the west
```

```
            Nnode = (j+1)*(M+1)+i # node to the north
            Snode = (j-1)*(M+1)+i # node to the south

            if (i == 0): # check for east boundary
                A[node,node] = 1.0
                b[node] = numpy.sin(numpy.pi*y[j])
            elif (i == M): # check for west boundary
                A[node,node] = 1.0
                b[node] = 0.0
            elif (j == 0): # check for south boundary
                A[node,node] = 1.0
                b[node] = 0.0
            elif (j == N): # check for north boundary
                A[node,node] = 1.0
                b[node] = 0.0
            else:
                A[node,node] = -coeff - h*wind[0] - h**2*wind[1]/k
                A[node,Enode] = 1.0
                A[node,Wnode] = 1.0 + h*wind[0]
                A[node,Nnode] = h2k2
                A[node,Snode] = h2k2 + h**2*wind[1]/k
                b[node] = h**2 * 0.0

z = solve(A,b)
Z = z.reshape(M+1,N+1)
pylab.contour(X,Y,Z)
pylab.colorbar()
pylab.xlabel('x')
pylab.ylabel('y')
```

The convective term can have a significant impact on the solution to the model problem, depending on the magnitude of the wind. The contours associated with a fluid velocity of $(5.0, 5.0)$ are shown in figure 11.3. Once the magnitude of the fluid velocity exceeds about 100, the problem quickly becomes more numerically demanding and a much finer mesh (i.e., more nodes) is likely to be required to get an accurate result. An inaccurate result is almost always inexpensive from a computational standpoint, but accuracy can be costly.

Exercise 11.1 A few days every year, it is possible to detect an unpleasant smell on the campus of the University of Colorado at Boulder. The (incorrect) explanation given to students was that this was the "Husker smell" emanating from Nebraska. (The correct explanation identified a company in Greeley, Colorado that occasionally produced bad smells). Develop a model of convection and diffusion over the state of Colorado. Model Colorado as a unit square (feel free to be more accurate if you are so compelled as the state is definitely not a square) and apply zero concentration boundary conditions to all boundaries except the eastern half of the north boundary (i.e. the right half of the top boundary) and the northern half of the east boundary (i.e., the top

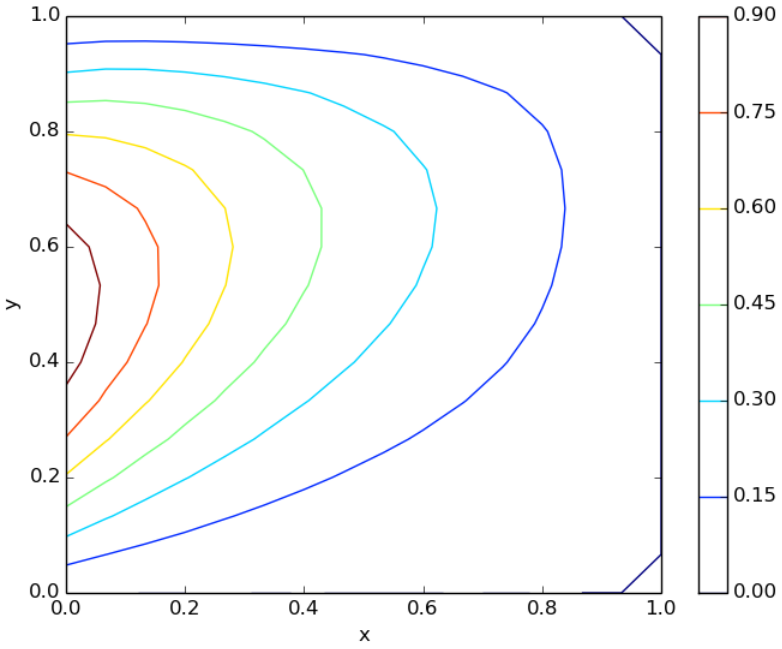


Figure 11.3: Contours of the solution for the model problem with a convective flow of $(5.0, 5.0)$, i.e., towards the northeast corner.

half of the right boundary), which roughly corresponds to the shared border between Nebraska and Colorado. (It may be helpful to consult a map of the United State.) Along the portion of Colorado's border that is shared with Nebraska, apply a concentration of 1.0. Determine the concentration in the center of the state of Colorado on a windless day and then the concentration for wind velocities in the x-direction only between -10.0 and +10.0. ■

11.3 Finite Difference Method for transient PDEs

Partial differential equations have multiple independent variables, and the first part of this chapter examined problems where all the independent variables were spatial variables. These equations described changes in two or more spatial directions. The other possibility is that one of the independent variables is time. The final part of this chapter examines problems where there are derivatives with respect to time and space in the same equation. This class of PDEs is often called 'parabolic' PDEs. Before examining a finite difference algorithm to solve a parabolic PDE problem, let's briefly discuss controlled drug release.

Researchers are increasingly developing devices that gradually release a pharmaceutical drug over time. In most cases, a gradual release is preferred to the burst release that is associated with the injection of a drug. One type of device is based on the embedding of the drug in a polymer and allowing the drug to gradually diffuse out of the polymer over time. The polymeric material is eventually passed through the digestive track after the drug has been gradually released via diffusion. If we assume that device is shaped like a chip (think poker chip), then the diffusion occurs primarily in one spatial direction – the direction that requires the shortest distance for diffusion. In this case,

the concentration is described by the equation

$$\frac{dc}{dt} = \mathscr{D}\frac{d^2c}{dx^2}$$

where $\mathscr{D}$ is the diffusivity and c is the concentration within the device. We assume that the distance from the surface of the device, $x = 0$ to the center is 1.0 (dimensionless). As a result, the spatial domain is $0 \leq x \leq 1.0$. Further, we assume that the concentration at the surface is zero, i.e., $c(x = 0) = 0.0$, implying the any drug that diffuses to the surface is immediately swept away, and the device is assumed symmetric about the center, i.e., $\frac{dc(x=1.0)}{dx} = 0$. Finally, we assume that the initial concentration is 1.0 (dimensionless) everywhere except the surface, i.e., $c(t = 0, 0 < x \leq 1.0) = 1.0$.

We have previously discussed how to replace the derivatives in the PDE with finite difference approximations. For this problem, the time that we wish to simulate is going to be divided into N discrete steps and we are going to approximate the time derivative with a forward difference approximation (i.e., use the forward Euler method). The one spatial dimension is going to be divided into M intervals (or $M+1$ nodes) that span from 0.0 to 1.0, and the second derivative in space will be replaced with a centered difference approximation. The result of the finite difference approximation is that the original PDE is replaced with the finite difference equation:

$$\frac{c_{i,new} - c_{i,old}}{\delta t} = \mathscr{D}\frac{c_{i+1,old} - 2c_{i,old} + c_{i-1,old}}{h^2}$$

where i is the index for the spatial location of the node, $c_{i,old}$ refers to the concentration at node i at the previous time step, and $c_{i,new}$ refers to the concentration at the next time step (i.e., the unknowns). This equation is usually rearrange to give

$$c_{i,new} = c_{i,old} + \delta t \cdot \frac{\mathscr{D}}{h^2}[c_{i+1,old} - 2c_{i,old} + c_{i-1,old}] \ .$$

Notice that explicit time stepping is used so we can calculate the new concentration using only concentrations from the previous time step. This equation also shows how the new concentration at each node depends upon three concentrations from the previous time step – the concentration at the same node and the two neighboring nodes. Figure 11.4 summarizes the connection between the concentration at a node and its neighboring nodes from the previous time step.

A Python script that utilizes the finite difference approximation to solve the transient diffusion problem, i.e., the drug release problem, is shown below. The algorithm is similar to the others shown in this chapter in that the first section of the algorithm performs some basic setup operations. The number of time and space intervals is specified, the size of the domain and duration of the simulation are set, and some vectors containing the spatial node locations and time points are constructed. One additional and important step in the setup phase is that a vector must be constructed that contains the initial conditions.

```
import numpy
import pylab

M = 15 # intervals in x-direction
N = 200 # total number of time steps
left = 0.0 # edge locations
right = 1.0
```

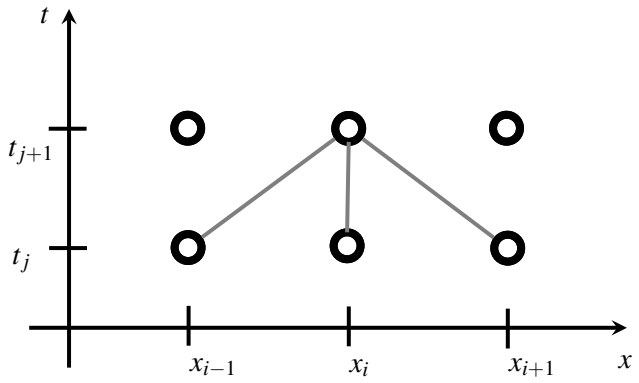


Figure 11.4: Diagram illustrating the finite difference approximation. Time is shown on the y-axis, and space is on the x-axis. When solving for the concentration at a node for the next time step, the concentration will depend on three adjacent nodes from the previous time step.

```
start = 0.0
stop = 20.0
diffh2 = 5.0

# node locations
x = numpy.linspace(left,right,M+1)
t = numpy.linspace(start,stop,N+1)
h = x[1]-x[0]
dt = t[1] - t[0]

# Initial conditions
c_old = numpy.ones(M+1)
c_old[0] = 0.0
c_new = numpy.zeros(M+1)
pylab.plot(x,c_old)
pylab.xlabel('x')
pylab.ylabel('c')

# time loop
for j in range(1,N+1):
    c_new[0] = 0.0
    for i in range(1,M):
        c_new[i] = c_old[i] + dt*diffh2*(c_old[i-1] - 2*c_old[i] + c_old[i+1])
    c_new[M] = c_new[M-1]
    c_old = c_new.copy()
    if j%20 == 0:
        pylab.plot(x,c_new)
```

The main section of the algorithm consists of two loops. The outer loop is a loop through the time steps, and the inner loop cycles through the spatial nodes. The algorithm operates by calculating new concentrations at every spatial location before moving on to the next time step. Care must be taken to set the boundary condition. The Dirichlet boundary condition at $x = 0$ is set before proceeding through the inner loop through the spatial nodes. The Neumann boundary condition at $x = 1$ is enforced by setting the concentration of the edge node to be equal to its nearest neighbor. The justification for doing this comes from the boundary condition equation:

$$\frac{dc}{dx} = \frac{c_i - c_{i-1}}{h} = 0$$

or

$$c_i = c_{i-1} \ .$$

This simulation uses a large number of time steps (200) and plotting the solution at every time step creates a very busy figure. To simplify the figure showing the results, the solution is only plotted every 20th time step by checking to see if the remainder of dividing the time step number by 20 is zero. The results from running the algorithm are shown in figure 11.5.

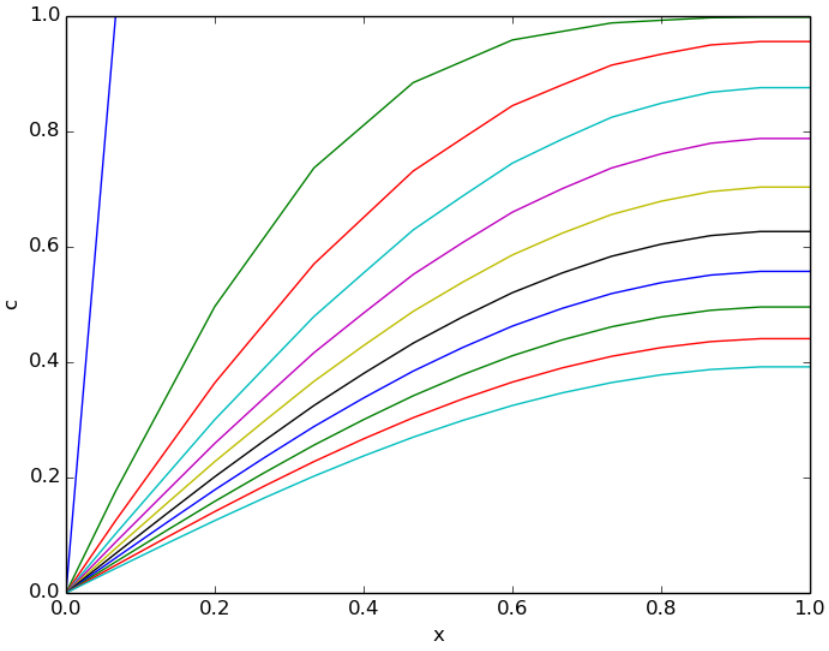


Figure 11.5: Plot of the concentration for the model problem shown every 20th time step. The initial concentration is the highest, and as time increases, the concentration decreases.

Observant readers may be surprised that the algorithm above used 200 time steps and only 16 spatial nodes. This choice can be justified in part by observing that the approximation of the time derivative is first-order (temporal error is $O(\delta t)$ and the spatial derivative approximation is second-order ($O(h^2)$) so we should expect to need a higher temporal resolution. An alternative

justification can be provided by simply re-running the algorithm with only 40 time steps. The result of this numerical experiment is shown in figure 11.6. In this case, the approximation error is seen to grown exponentially — a clear symptom of a stiff differential equation. Recalling that stiff differential equations should be solved using an implicit time stepping method, we simply state here that switching to an implicit time stepping method like backward Euler will avoid the exponential error growth seen here. The disadvantage of switching to implicit time stepping is that a linear (or nonlinear) matrix problem must be solved every time step. For this reason, many algorithms use a sufficiently small step size to maintain stability. The time step size must be kept smaller than a constant times the node spacing squared, i.e., $\delta t < kh^2$. Techniques for determining k are available in most numerical analysis books or k can be determined through numerical experiments.

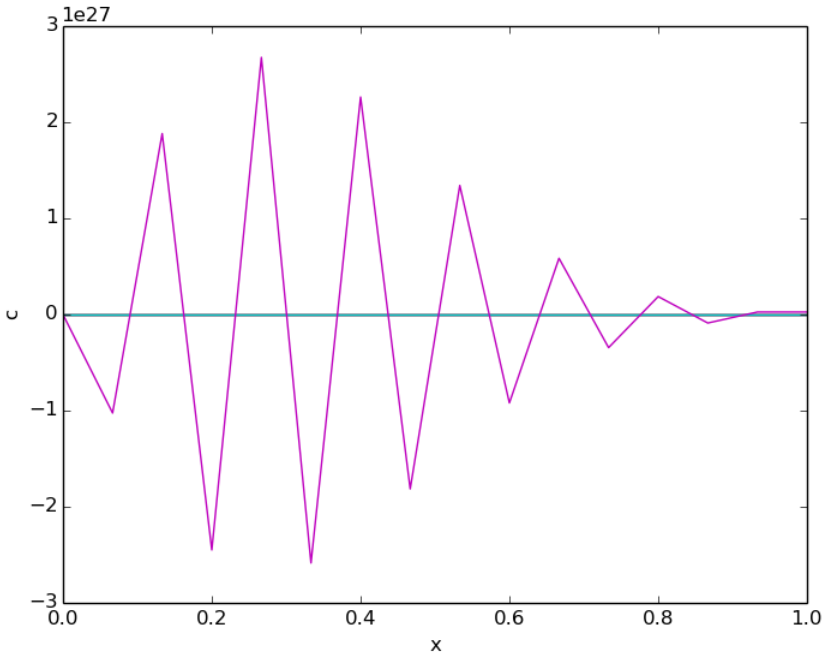
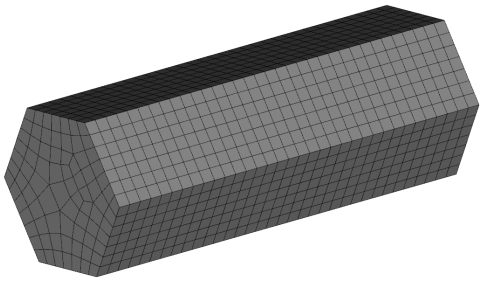


Figure 11.6: Plot of the concentration for the model problem using only 40 time steps. The error grows exponentially, and the final solution is completely without value.

A Warning
Why FEM?
Laplace's Equation
The Mesh
Discretization
Wait! Why are we doing this?
FEniCS implementation
Further Reading



12. Finite Element Method

This is optional chapter on the use of the finite element method (FEM) and the FEniCS library. FEniCS is only available for Linux and MacOS.

12.1 A Warning

The finite element method is an advanced numerical approach to solving partial differential equations and is typically only taught at the graduate level. This chapter only briefly develops the FEM. The interested reader is referred to the suggested reading at the end of the chapter for further information regarding FEM. The primary focus here is to present a tool (FEniCS) that advanced readers may appreciate later in their academic careers for solving PDEs in more complicated geometries and with greater computational efficiency.

12.2 Why FEM?

The FEM is used to solve partial differential equation such as those that were solved earlier using the finite difference method. Why, we might ask, should we discuss anything beyond the finite difference method? Why do people use FEM instead of finite differences? There are a number of reasons for using FEM, but three reasons are typically given more often than any other:

1. The finite element method is easy to use when the shape of the domain is complex. If we wish to solve Laplace's equation on a domain that is shaped like the human brain (and this is a real problem in medical imaging), then we need to use FEM because finite difference cannot accurately capture the shape of complex domains.
2. The finite element method is typically easier to extend to higher-order approximations. Implementing a 4th- or even 8th-order accurate finite element method is relatively straightforward. This level of accuracy is rarely needed, but it can be a real advantage in some situations like modeling blood flow in the aorta that is nearly turbulent.
3. Rigorous mathematical analysis of FEM is much more extensive than analysis of finite differences.

12.3 Laplace's Equation

The use of FEniCS and the FEM will only be briefly introduced here through the lens of an example problem. An excellent book [L+12] and a large library of example problems (`fenicsproject.org`) are available for FEniCS and the interested reader should utilize these resources to gain a much more comprehensive understanding of the FEniCS library and FEM. The FEniCS project is actually a collect of software packages that are used in concert to solve differential equations using FEM. The central software package that to some extent ties everything together is called dolfin. Many of the packages, including dolfin, are written in C++ for faster execution, but wherever necessary, the C++ code has been wrapped so that it can be called from Python. To utilize the various FEniCS packages within a Python code, the first step is to import the dolfin library with **from** dolfin **import** $*$. Since this is the only library that is imported, we do not need to worry about importing multiple objects with the same name.

12.3.1 The Mesh

Whenever we use FEM, we always need to start with a mesh. A mesh is a division of the problem domain into small polyhedral shapes. In two dimensions, we typically divide the domain into triangles or quadrilateral shapes. For simple domains, like squares, rectangles, circles, etc., FEniCS has built in mesh generators that automatically divide the domain into triangles. The function call mesh = UnitSquareMesh(32, 32) divides a unit square into triangles with 32 triangles in each direction. The resulting mesh is shown in figure 12.1. For more complex shapes, a number of software packages are available that provide a CAD-like interface for defining the domain (e.g., our domain might be a mountain bike) and then divide the domain into small polyhedral shapes of a desired type and size. The Cubit software from Sandia National Laboratory is one example.

12.3.2 Discretization

The basic idea behind the FEM is to select a set of mathematical functions called the function space and then determine the exact function on each element that best satisfies the original PDE. The most common function space by a significant margin is polynomial functions of a selected order. While this is a pretty simple idea, the actual implementation is much more difficult. To construct a function space in FEniCS, call V = FunctionSpace(mesh, 'Lagrange', 2) where 'Lagrange' specifies Lagrange polynomials and '2' is the polynomial degree (i.e., quadratic polynomials).

Recall that Laplace's equation is

$$\nabla^2 u = \frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = 0$$

and the same boundary conditions as before will be used here ($u = sin(\pi y)$ on the west boundary, $u = 0$ elsewhere). We begin by integrating both sides of this equation over the entire domain and multiplying both sides of the equation by a 'test' function, v, giving:

$$\int_\Omega (\nabla^2 u)(v)\, d\Omega = \int_\Omega 0 \cdot v\, d\Omega = 0\ .$$

Using integration by parts, this equation becomes:

$$\int_\Omega (\nabla u)(\nabla v)\, d\Omega + \text{boundary terms} = 0\ .$$

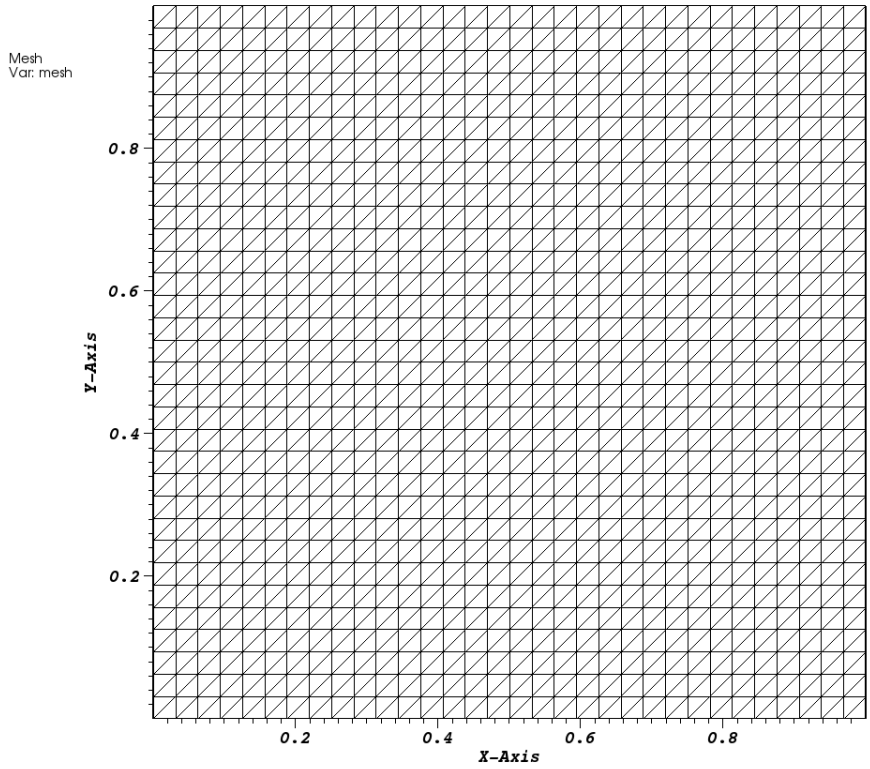


Figure 12.1: Triangular mesh from the UnitSquareMesh() function.

This form of the equation is called the *weak form*, and the boundary terms cancel along all interior boundaries. Along external boundaries, the boundary terms are used to enforce Neumann boundary conditions or they are not present when we have Dirichlet boundary conditions.

12.3.3 Wait! Why are we doing this?

The short answer is to ask a mathematician. The long answer will result when you ask a mathematician. The abbreviated answer is that we are trying to find a polynomial approximation of u on every element. Between elements the approximate solution is continuous, but the derivative is not continuous. By multiplying by a test function, we were able to move one of the two derivatives off of u and onto v. Now we have an equation that we can actually evaluate over the domain to determine an approximate solution. On each element we are trying to determine, essentially, the polynomial coefficients for that element. Since we have many elements, the result is a large linear system where the unknowns are the polynomial coefficients for all the element.

12.3.4 FEniCS implementation

The Python script below uses FEniCS (primarily dolfin) to solve Laplace's equation on the unit square.

```
from dolfin import *

# Create mesh
mesh = UnitSquareMesh(32, 32)

# Create function space
V = FunctionSpace(mesh, 'Lagrange', 2)

# Define boundary conditions
u0 = Expression('sin(pi*x[1])*(1-x[0])')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(0.0)
a = inner(grad(u), grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)

solve(a == L, u, bc,
      solver_parameters={'linear_solver': 'cg',
                         'preconditioner': 'ilu'})

# Dump solution to file in VTK format
file = File('poisson.pvd')
file << u
```

The construction of the matrix problem happens when the matrix object, *a*, is created. The inner product of 'grad(u)' and 'grad(v)' is same as the weak form equation above. The right-hand-side, *L*, is just a vector of zeros. The dolfin solve() function solves the linear matrix problem using an iterative method (conjugate gradient) with an incomplete matrix factorization as the preconditioner. The final result is written to a file and can be visualized using Paraview, Visit, or other visualization software packages. Figure 12.2 was generated using Visit from Lawrence Livermore National Laboratory.

12.4 Further Reading

The following books may be useful for learning more about the finite element method.

- An Introduction to the Finite Element Method by Reddy [Red93]

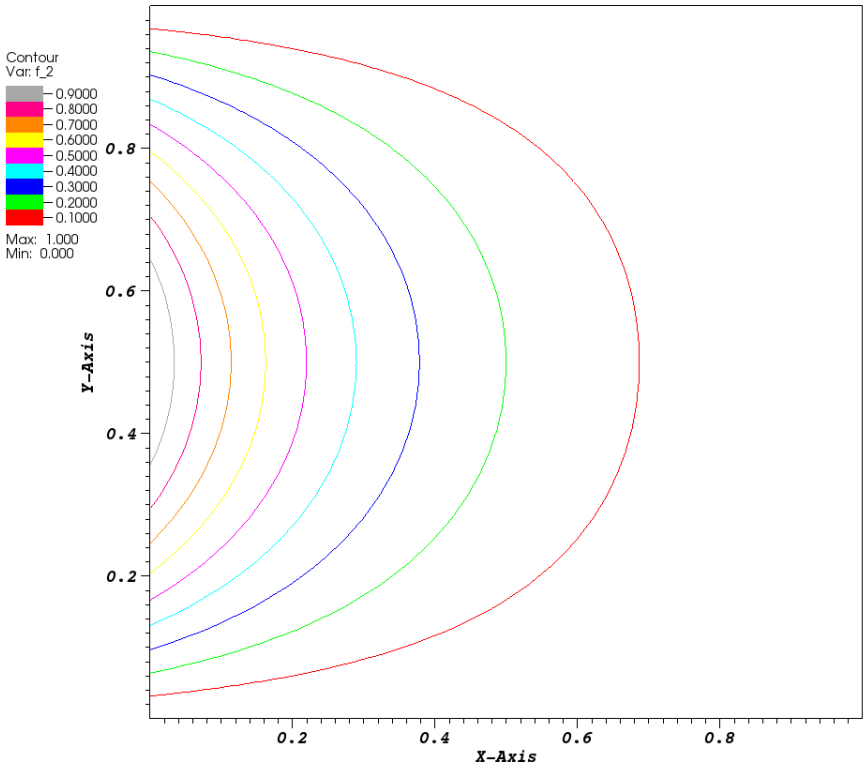
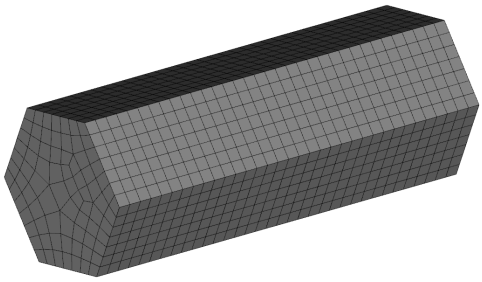


Figure 12.2: FEM solution to the model problem (Laplaces equation) using a second-order Lagrange function space. The solution was visualized using Visit.

- Finite Element Methods for Flow Problems by Donea and Huerta [DH03]
- The Finite Element Method in Heat Transfer and Fluid Dynamics by Reddy and Gartling [RG94]
- The Mathematical Theory of Finite Element Methods by Brenner and Scott [BS02]



Bibliography

[AR05] H. Anton and C. Rorres. *Elementary Linear Algebra*. 9th edition. John Wiley and Sons, 2005 (cited on page 64).

[BS02] S.C. Brenner and L.R. Scott. *The Mathematical Theory of Finite Element Methods*. 2nd edition. New York, NY: Springer-Verlag, 2002 (cited on page 143).

[BHM00] W.L. Briggs, V.E. Henson, and S.F. McCormick. *A Multigrid Tutorial*. 2nd edition. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2000 (cited on page 57).

[BF01] R.L. Burden and J.D. Faires. *Numerical Analysis*. 7th edition. Pine Grove, CA: Brooks/-Cole, 2001 (cited on pages 50, 57).

[ÇP13] Y.A. Çengel and W.J. Palm III. *Differential Equations for Engineers and Scientists*. 1st edition. New York, NY: McGraw-Hill, 2013 (cited on page 13).

[DH03] J. Donea and A. Huerta. *Finite Element Methods for Flow Problems*. West Sussex, England: John Wiley & Sons, 2003 (cited on page 143).

[Dow12] A.B. Downey. *Think Python*. 1st edition. Sebastopol, CA: O'Reilly, 2012 (cited on page 31).

[FR05] R. M. Felder and R. W. Rousseau. *Elementary Principles of Chemical Processes*. 3rd edition. Hoboken, NJ: John Wiley & Sons, 2005 (cited on page 39).

[Gre97] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. Frontiers in Applied Mathematics. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997 (cited on page 57).

[Hin83] A.C. Hindmarsh. "ODEPACK, A Systematized Collection of ODE Solvers". In: *Scientific Computing*. Edited by R.S. Stepleman. North-Holland Amsterdam: Elsevier, 1983 (cited on page 111).

[Kiu10] J. Kiusalaas. *Numercial Methods in Engineering with Python*. 2nd edition. New York, NY: Cambridge Press, 2010 (cited on page 31).

[Lan10] H. P. Langtangen. *Python Scripting for Computational Science*. 3rd edition. Texts in Computational Science and Engineering. Berlin: Springer, 2010 (cited on page 31).

[Liv06] M. Livio. *The Equation that Couldn't be Solved: How Mathematical Genius Discovered the Language of Symmetry*. New York, NY: Simon & Schuster, 2006 (cited on page 33).

[L+12] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8 (cited on page 140).

[Lut13] M. Lutz. *Learning Python*. 4th edition. Sebastopol, CA: O'Reilly, 2013 (cited on page 31).

[Mar09] A. Martelli. *Python in a Nutshell*. 2nd edition. Sebastopol, CA: O'Reilly, 2009 (cited on page 31).

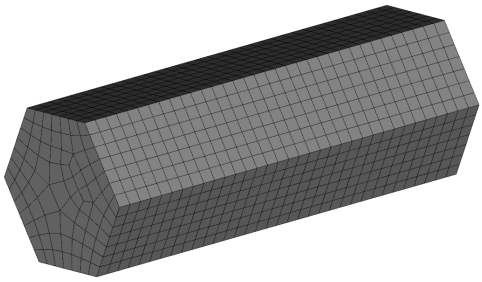
[Red93] J.N. Reddy. *An Introduction to the Finite Element Method*. 2nd edition. Boston, MA: McGraw Hill, 1993 (cited on page 142).

[RG94] J.N. Reddy and D.K. Gartling. *The Finite Element Method in Heat Transfer and Fluid Dynamics*. Boca Raton, FL: CRC Press, 1994 (cited on page 143).

[Tea14] SymPy Development Team. *SymPy: Python library for symbolic mathematics*. 2014. URL: http://www.sympy.org (cited on page 34).

[Tos09] S. Tosi. *Matplotlib for Python Developers*. Birmingham, UK: Packt Publishing, 2009 (cited on page 31).

[ZC06] D.G. Zill and M.R. Cullen. *Advanced Engineering Mathematics*. 3rd edition. Sudbury, MA: Jones and Barlett, 2006 (cited on page 13).



Index

A

Antoine's equation 66
array construction 26
array operations 28
array slicing 27

B

backward difference approximation 94
backward substitution 48
Bisection Method 71
Broyden's Method 75
broyden1 function 70, 81

C

centered difference approximation 94
Clausius-Clayperon equation 60
colorbar 30
comparators 22
conditionals 23
csv files 83
curve fitting 64

D

def keyword 25
direct methods 46
distillation column 44

E

Equation classification 9
explicit time stepping 105

F

file, data import from 83
finite difference method 120
finite element mesh 140
first derivative approximation 94
for loops 24
forward difference approximation 94
forward Euler method 104
fraction decomposition 40
fsolve function 81
function space 140

G

Gauss-Seidel iteration 54
Gaussian elimination 47
Gaussian quadrature 100
Getting Python 17

I

if statements 23
iterative methods 51

J

Jacobi iteration 52

L

Laplace's equation 125
linear regression 61
linear regression statistics 89
lists 20

M

Maple 34
math library 22
Mathcad 34
Mathematica 34
MATLAB 34
Matplotlib 18, 29
meshgrid 30
Michaelis-Menton kinetics 104
midpoint rule 98
modified Euler method 107

N

NaN 71
newton krylov function 81
Newton's Method 74
normal equations 62
numerical integration 97
numpy 17, 26, 45
numpy linear algebra library 46

P

predator-prey models 110
Problems 12
pylab 29
pylab.plot() 29
pyplot 29
Python functions 25
Python installation 18
Python versions 17

R

range() function 24

S

Sage 34
scipy 18
scipy initial value problem 109
Scipy quadrature function 100
secant method 76
second derivative approximation 96
second-order initial value problems 110
Shooting Method 116
Spyder 18
SRK equation of state 69
standard score 88
stiff differential equations 110
symbolic derivatives 40
symbolic integration 41
symbolic mathematics 33
SymPy 18, 34
SymPy factor function 38
SymPy solve function 35
system of linear equations 43
systems of nonlinear equations 77

T

transient PDEs 133
trapezoid rule 98

V

van der Waals equation 36
vapor pressure 60
vector norms 52
vector summation 63

W

weak form 141
Why Python? 15
Wolfram Alpha 34

Z

z-score 88