

ENGN 2020 Course Notes:

Numerical Methods in Engineering and Physics

Andrew Peterson, Brown University

Copyright © 2018–current, Andrew Peterson, All Rights Reserved

Contents

Preface	5
1 Linear algebra as a computational tool	6
1.1 Basics of matrix algebra	6
1.1.1 Motivation	6
1.1.2 Basic matrix manipulation rules	7
1.1.3 Solving our example	9
1.2 Understanding computational costs	9
1.2.1 The “flop”	9
1.2.2 Matrix multiplication flops	10
1.2.3 Computational order, \mathcal{O}	10
1.3 Linear systems: “inverting a matrix”	10
1.3.1 Solving by hand	10
1.3.2 Row operations	12
1.3.3 Gauss elimination	12
1.3.4 Matrix inversion	13
1.3.5 Solving linear systems in practice	14
1.3.6 LU decomposition	14
1.3.7 Iterative approaches	15
1.3.8 Understanding solution space (rank)	15
1.3.9 Rank, and singular matrices	17
1.3.10 Relation between rank and determinant	17
1.4 Eigenvalue problems: “diagonalizing a matrix”	17
1.4.1 What is an eigenvalue problem?	18
1.4.2 Example: oscillating springs	18
1.4.3 Solving eigenvalue problems the textbook way	20
1.4.4 Can we apply the textbook approach to larger systems? (Absolutely not.)	23
1.4.5 Bounding eigenvalues with Gerschgorin’s Theorem	23
1.4.6 Diagonalization (and similarity)	24
1.4.7 Coaxing a matrix into diagonal form: QR and tridiagonalization	25
1.4.8 Google PageRank	27
1.4.9 Power method	29
1.4.10 Eigenvalues in practice	31
1.5 Odds & ends	32
1.5.1 Norms of vectors and matrices	32
1.5.2 Sparse matrices	33
1.5.3 Iterative methods in linear systems, ill conditioning	34
1.5.4 Basis vectors, and transformations between bases	35

2	Nonlinear algebraic equations	40
2.1	Example: a reactor	40
2.2	Fixed-point iteration	41
2.3	Newton–Raphson Iteration	42
2.3.1	Extension to systems of nonlinear equations	44
2.3.2	Form	44
2.3.3	Convergence behavior	45
2.3.4	Secant method	45
2.3.5	Linesearch	46
2.3.6	Numerical packages, and practical approaches	47
2.4	Polynomial roots	47
2.5	Root behavior	49
2.5.1	Example	49
2.5.2	Factoring out roots	50
2.5.3	Bifurcation	52
3	Optimization of continuous functions	57
3.1	Optimization	57
3.1.1	Problem formulation	59
3.2	Gradient-based methods	59
3.2.1	General step.	60
3.2.2	Search direction.	60
3.2.3	Step size	61
3.3	Hessian-based methods	61
3.3.1	Approximate Hessians (Quasi-Newton)	61
3.4	Non-linear regression: parameter estimation	62
3.5	Constrained optimization	63
3.5.1	Bounded variables	63
3.5.2	Mathematically constrained	63
3.6	Gradient-free methods	65
3.6.1	Nelder–Mead (downhill simplex)	65
4	Graph theory	67
4.1	Formalism	69
4.2	Exploring graphs / shortest-distance problem	70
4.2.1	Breadth-first algorithm: Moore’s algorithm.	70
4.2.2	Dijkstra’s algorithm	71
4.3	Shortest spanning trees	74
4.3.1	Kruskal’s greedy algorithm	74
4.3.2	Prim’s algorithm	75
4.4	Other types of graph theory problems	75
5	Mixed optimization	78
5.1	Pareto optimality	78
5.2	Mixed problems	79
5.2.1	Simple problems: combinatorics	79
5.2.2	Treating as continuous; adding penalties	79
5.2.3	Branch and bound	80
5.2.4	Mixed-integer non-linear programming	80
5.3	Global optimization	80

6	Integration of differential equations	82
6.1	Models	82
6.2	Initial value problems	83
6.2.1	Example: Heat loss	83
6.2.2	General form and numeric approach	86
6.2.3	Explicit Euler	87
6.2.4	Predictor–corrector method (Improved Euler)	89
6.2.5	Runge–Kutta	90
6.2.6	Systems of ODEs	91
6.2.7	Time steps	93
6.2.8	Stiff systems	94
6.2.9	Implicit methods	95
6.3	Boundary value problems	96
6.3.1	Example	96
6.3.2	Non-dimensionalization	97
6.3.3	Finite-difference approximation	98
6.3.4	Multidimensional boundary value problems	99

Preface

This document is a work-in-progress that contains a summary of the in-class lecture notes from ENGN 2020, taught at Brown University in the Spring of 2019. Note that the lecture notes are added by the instructor to this document after each lecture, but there will always be a lag between when the material is presented in lecture and when it appears in this document, so this does not serve as a substitute for your handwritten notes, but rather as a reference once the class is complete.

References. This course relies primarily on two textbooks:

1. Erwin Kreyszig, *Advanced Engineering Mathematics (Tenth Edition)*, Wiley, 2011. [1]
2. Kenneth Beers, *Numerical Methods for Chemical Engineering: Applications in Matlab*, Cambridge, 2007. [2]

Whenever possible, references are provided within this document to where you can find additional reading. When the reading is in either of the two textbooks above, it is simply denoted with a K (Kreyszig) or a B (Beers). In the case of Kreyszig, a specific section is typically provided (*e.g.*, K19.2); since the Beers textbook does not provide section-level headings, it instead is listed as a chapter:page (*e.g.*, B2:79 means the reading starts on page 79, which is in Chapter 2).

Vector and matrix notation. Most texts use bold characters to denote vectors and matrices (typically with vectors as lower-case and matrices as upper-case). This is difficult to do on the whiteboard in lecture, so instead we use underlines to denote vectors and matrices, and here in the readings we'll also bold them for good measure (overkill). Thus, a matrix will look like **A** and a vector like **x**. Note that vectors are assumed to be column vectors, and a row vector will typically be marked as **x**^T.

Code. The concepts discussed in this course are, by and large, independent of the specific software package chosen, such as Matlab, python, or Julia. The examples and homework in this class use python routines (especially those in the packages numpy and scipy), but we try to keep the reliance on a particular code's functionality to a minimum in the lecture notes.

Topic 1

Linear algebra as a computational tool

Most scientific computing relies heavily upon the computational manipulation of vectors, matrices, and higher dimensional arrays of numbers. Many numerical problems ultimately reduce to common operations such as inverting or diagonalizing a matrix. We'll begin the course with a review of linear algebra, focusing on numerical techniques to solve common problems using the appropriate trade-off between speed and accuracy.

1.1 Basics of matrix algebra

1.1.1 Motivation

We first begin by reminding ourselves where linear algebra comes from, and we'll start with a simple algebraic problem with only the single variable x :

$$3x = -2$$

which we can immediately solve to get $x = -2/3$. If we found ourselves solving this problem frequently, we could solve it once generically and then have a formula to plug into; that is,

$$ax = b \Rightarrow \boxed{x = \frac{b}{a}}$$

where a and b are parameters (known numbers) and x is the unknown variable.

Now, let's extend this so that we have two equations and two unknown variables, (x_1, x_2) , such as

$$\begin{aligned} 3x_1 + 4x_2 &= 8 \\ 2x_1 + 9x_2 &= 5 \end{aligned}$$

We learned strategies in our early algebra course to approach this: for example, re-arrange the top equation to isolate x_1 , then plug this expression for x_1 into the bottom equation to express it only in terms of x_2 . Solve the resulting equation for x_2 , then plug this back into the x_1 equation to complete. This works for small systems, but is a bit tedious and error-prone.

Using matrices, we can instead look for systematic ways to approach this. First, let's re-write the equation in the language of matrices:

$$\begin{bmatrix} 3 & 4 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 8 \\ 5 \end{bmatrix} \tag{1.1}$$

or more compactly

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$$

In this course, we'll note matrices and vectors in bold, with two lines under matrices and one line under vectors. Written in this form ($\underline{\underline{\mathbf{A}}}\underline{\mathbf{x}} = \underline{\mathbf{b}}$) our system of equations looks very similar to our single equation ($ax = b$). It would be great if we could find a systematic way to solve for $\underline{\mathbf{x}}$, just as we found a systematic solution ($x = b/a$) in our scalar system. (For example, could we say $\underline{\mathbf{x}} = \underline{\mathbf{b}}/\underline{\underline{\mathbf{A}}}$?¹) To do so, we'll need to develop some mathematical rules for dealing with matrices.

1.1.2 Basic matrix manipulation rules

Reference: K7.1, K7.2, K7.8

We'll review some basic rules of matrix construction and manipulation; see Chapter 7 of the textbook [1] for more complete descriptions or consult any linear algebra textbook.

Vectors, matrices, and arrays. For our purposes, we can think of a vector as a list of numbers and a matrix as a table of numbers. In scientific computing, these are often just taken to be *arrays* of different dimensions; *e.g.*, a vector is a one-dimensional array and a matrix is a two-dimensional array. In our example of equation (1.1), our vectors ($\underline{\mathbf{x}}$ and $\underline{\mathbf{b}}$) each have shape (2), while our matrix has shape (2,2).

By convention, if a matrix's shape is given as (m, n) , it has m rows and n columns, where the rows run horizontally and the columns run vertically. Vectors can also be thought of as column vectors (as shown in equation (1.1)) or row vectors. In this class, we'll tend to assume vectors are column vectors and will represent row vectors as the *transpose* of column vectors; *e.g.*, $\underline{\mathbf{b}}^T = [8 \ 5]$.

It can also be useful to refer to column vectors as matrices having shape $(m, 1)$, while row vectors have shape $(1, m)$. In this way, the rules we develop below for basic operations, such as addition and multiplication, apply equally to matrices and vectors.

Indexing. a_{ij} refers to row i and column j of the $m \times n$ matrix $\underline{\underline{\mathbf{A}}}$:

$$\underline{\underline{\mathbf{A}}} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

where we often compactly refer to the matrix indexing as $\underline{\underline{\mathbf{A}}} = [a_{ij}]$.

Matrix addition. Adding two matrices gives another matrix:

$$\underline{\underline{\mathbf{A}}} + \underline{\underline{\mathbf{B}}} = \underline{\underline{\mathbf{C}}}$$

Matrix addition is done element-by-element, that is $c_{ij} = a_{ij} + b_{ij}$ for all i, j . Therefore, addition is only defined for matrices of identical size, and the resulting matrix $\underline{\underline{\mathbf{C}}}$ has the same size as either $\underline{\underline{\mathbf{A}}}$ or $\underline{\underline{\mathbf{B}}}$.² By its definition, we can easily see that this operation *commutes*:³

$$\underline{\underline{\mathbf{A}}} + \underline{\underline{\mathbf{B}}} = \underline{\underline{\mathbf{B}}} + \underline{\underline{\mathbf{A}}}$$

¹No.

²You should be careful with this computationally, as often scientific computing software will allow you to add two arrays of different shape together (*e.g.*, an $m \times n$ matrix plus an $m \times 1$ vector); be sure to understand what convention it is adopting if you use this.

³Wikipedia's definition: "In mathematics, a binary operation is commutative if changing the order of the operands does not change the result."

Multiplication by a scalar. Multiplying a matrix by a scalar gives another matrix:

$$c \underline{\underline{\mathbf{A}}} = \underline{\underline{\mathbf{D}}}$$

In this operation, c is multiplied by each element of the matrix to give the elements of $\underline{\underline{\mathbf{D}}}$; that is, $d_{ij} = c a_{ij}$. This rule, combined with the rule above, implies how we handle *subtraction*:

$$\underline{\underline{\mathbf{A}}} - \underline{\underline{\mathbf{B}}} = \underline{\underline{\mathbf{A}}} + (-1)\underline{\underline{\mathbf{B}}}$$

That is, subtraction is done with the operation $a_{ij} - b_{ij}$.

Matrix multiplication. The multiplication of two matrices is slightly more cumbersome, and is defined as follows:

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{B}}} = \underline{\underline{\mathbf{C}}}$$

$$c_{jk} = \sum_{l=1}^n a_{jl} b_{lk}$$

where the second line gives the resulting elements of the new matrix $\underline{\underline{\mathbf{C}}}$. From this definition, we can immediately see that if $\underline{\underline{\mathbf{A}}}$ is of size $m \times n$ and $\underline{\underline{\mathbf{B}}}$ is of size $n \times p$, then $\underline{\underline{\mathbf{C}}}$ is $m \times p$. (Further, this means that the second dimension of $\underline{\underline{\mathbf{A}}}$ and the first dimension of $\underline{\underline{\mathbf{B}}}$ must have the same length; that is, n .)

This is often most easily understood with a graphical representation, which is shown nicely in the textbook [1] in Figure K7-3 as

$$m = 4 \left\{ \begin{array}{c} \overbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix}}^{n=3} \end{array} \right\} \begin{array}{c} \overbrace{\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}}^{p=2} \end{array} = \begin{array}{c} \overbrace{\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \\ c_{41} & c_{42} \end{bmatrix}}^{p=2} \end{array} \left\{ m = 4 \right.$$

As seen above, we can interpret the elements of $\underline{\underline{\mathbf{C}}}$ to be the result of a dot-product of the rows of $\underline{\underline{\mathbf{A}}}$ with the columns of $\underline{\underline{\mathbf{B}}}$. Note that this does *not* commute! In general,

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{B}}} \neq \underline{\underline{\mathbf{B}}} \underline{\underline{\mathbf{A}}}$$

We can now also see that our example of equation (1.1) was correctly represented in matrix form by:

$$\begin{bmatrix} 3 & 4 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3x_1 + 4x_2 \\ 2x_1 + 9x_2 \end{bmatrix}$$

That is, our expression $\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}}$ is equivalent to the left-hand side of our original equations.

Identity matrix. The identity matrix is defined as the special matrix $\underline{\underline{\mathbf{I}}}$ which has 1's on the diagonal and 0's at all other locations. For example, the 3×3 identity matrix is

$$\underline{\underline{\mathbf{I}}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

From the matrix multiplication rules, you can work out that the identity matrix has the special property that when it multiplies a matrix $\underline{\underline{\mathbf{A}}}$ it just gives back that same matrix:

$$\underline{\underline{\mathbf{I}}} \underline{\underline{\mathbf{A}}} = \underline{\underline{\mathbf{A}}}$$

This also works for vectors, where $\underline{\underline{\mathbf{x}}}$ is a length- n vector and $\underline{\underline{\mathbf{I}}}$ is the $n \times n$ identity matrix:

$$\underline{\underline{\mathbf{I}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{x}}}$$

Matrix inversion. Matrix “division” is not really defined; instead we speak of matrix “inversion”, which carries the following definition:

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{A}}}^{-1} = \underline{\underline{\mathbf{A}}}^{-1} \underline{\underline{\mathbf{A}}} = \underline{\underline{\mathbf{I}}}$$

where $\underline{\underline{\mathbf{A}}}^{-1}$ is the inverse of matrix $\underline{\underline{\mathbf{A}}}$. Since the identity matrix is always square, this means that matrix inversion is only defined for square matrices $\underline{\underline{\mathbf{A}}}$; that is, where the number of rows and columns is the same. In general, it is not guaranteed that every (square) matrix $\underline{\underline{\mathbf{A}}}$ can be inverted.⁴

$\underline{\underline{\mathbf{A}}}^{-1}$ can be computed by various means, such as Gauss–Jordan elimination, which we’ll discuss in more detail later.

1.1.3 Solving our example

Now we have enough rules of matrix manipulation to find a general solution to our problem from the motivating section:

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$$

First, we’ll pre-multiply both sides by $\underline{\underline{\mathbf{A}}}^{-1}$:

$$\underline{\underline{\mathbf{A}}}^{-1} \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{A}}}^{-1} \underline{\underline{\mathbf{b}}}$$

Then we substitute $\underline{\underline{\mathbf{I}}}$ for $\underline{\underline{\mathbf{A}}}^{-1} \underline{\underline{\mathbf{A}}}$:

$$\underline{\underline{\mathbf{I}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{A}}}^{-1} \underline{\underline{\mathbf{b}}}$$

and since $\underline{\underline{\mathbf{I}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{x}}}$ we can write our solution

$$\underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{A}}}^{-1} \underline{\underline{\mathbf{b}}}$$

Therefore, if we can figure out how to invert our matrix $\underline{\underline{\mathbf{A}}}$, we can solve this problem generically. This will be discussed in §1.3.4.

1.2 Understanding computational costs

Although computational power has grown tremendously over the past decades, most scientific computing tasks are still computationally bottlenecked. Fundamentally, this is because as we increase the size of a system⁵ the computational demand typically does not grow linearly, but often grows with some power of the system size. Here, we will look at how we can quantify the computational demands of various routines.

1.2.1 The “flop”

First, we’ll define a unit of computing arithmetic; we typically define one operation as the combination of one multiplication and one addition, although it doesn’t matter much precisely how we define it. E.g., one operation is

$$2.789 \times 0.3982 + 4.134$$

We’ll refer to this as a “floating-point operation”, or a “flop”.⁶

⁴A square matrix that cannot be inverted is called singular.

⁵When we say increase the size of a system, we can be referring to different aspects. For example, in modeling a physical system we might be increasing the size in a single dimension. When we model things computationally, we tend to model points on a grid; another way to think of increasing the system size is refining the grid for higher accuracy.

⁶And we will refer to the plural of this as flops. Note that it’s common to use the term FLOPS, or floating-point operations per second, to describe the power of a computer. Here, we’ll use the lower-case version to refer to an individual operation, and reserve the capitalized version for computational power.

1.2.2 Matrix multiplication flops

Let's determine how many flops are required to multiply two matrices:

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{B}}} = \underline{\underline{\mathbf{C}}}$$

Sizes:

- $\underline{\underline{\mathbf{A}}}$: $m \times n$
- $\underline{\underline{\mathbf{B}}}$: $n \times p$
- $\underline{\underline{\mathbf{C}}} = \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{B}}}$: $m \times p$

For each element created in $\underline{\underline{\mathbf{C}}}$, we have n flops. There are $m \times p$ elements in $\underline{\underline{\mathbf{C}}}$. So the general answer is we require $m \times n \times p$ flops. Let's also examine two very common cases:

Case 1: Both matrices are square. If both matrices are square, that is, they are both $n \times n$, then this means we require n^3 flops.

Case 2: Matrix times vector. It's very common to multiply a square matrix by a vector: $\underline{\underline{\mathbf{A}}} \underline{\mathbf{b}}$. Then $\underline{\underline{\mathbf{A}}}$ is $n \times n$ and $\underline{\mathbf{b}}$ is $n \times 1$, so we require n^2 flops.

1.2.3 Computational order, \mathcal{O}

If we examine the two cases above, we see that $\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{B}}}$ scales with n^3 , while $\underline{\underline{\mathbf{A}}} \underline{\mathbf{b}}$ scales with n^2 . Consider what this means: if we want to make our system twice as large (double n), then the operation $\underline{\underline{\mathbf{A}}} \underline{\mathbf{b}}$ takes 4 times as long and the operation $\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{B}}}$ takes 8 times as long. This is typically the useful information we need—whatever operation has the highest exponent will dominate the computational demand as system sizes get large, and it really didn't matter much if our two terms had different prefactors—like $0.01n^3$ and $50n^2$ —as the system size gets large the cubic term will dominate. So we are typically just interested in the power of n involved.

We use the term computational order for this, and denote the two cases above as $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$, which we pronounce as “order n-cubed” and “order n-squared”. As we proceed through various algorithms in this course, we'll often come back to examining the demands of any particular calculation by looking at its order.

1.3 Linear systems: “inverting a matrix”

Earlier, we focused on the simple linear system $\underline{\underline{\mathbf{A}}} \underline{\mathbf{x}} = \underline{\mathbf{b}}$, where $\underline{\underline{\mathbf{A}}}$ and $\underline{\mathbf{b}}$ are known and we want to know the vector $\underline{\mathbf{x}}$. This is called a linear system, and solving a linear system (sometimes called inverting a matrix) is one of the two dominant matrix operations that are needed in computations. (The other being the eigenvalue problem, often referred to as diagonalizing a matrix.)

In this course, we'll go over how these are solved in practice, using computational tools. In formal linear algebra math courses, there will be many elegant proofs and derivations; we will have comparatively little of that here, instead focusing on the pragmatic solutions. Please take a good linear algebra course if you want to become expert in symbolic matrix manipulation.

Let's examine how this can be solved, starting from the pen-and-paper way and building up to more generic and efficient computational approaches.

1.3.1 Solving by hand

We will start by examining how we would systematically solve such systems “by hand”; that is, just using basic algebra. The rules and procedures we develop will directly lead to definitions of *row operations*, formalized in the next section, which are key to many matrix simplification algorithms. By recalling the origins of row operations, it is quite simple to remember the rules.

Let's look at the same system of equations we had last time.

$$\underline{\underline{\mathbf{A}}} \mathbf{x} = \underline{\mathbf{b}}$$

$$\begin{bmatrix} 3 & 4 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 8 \\ 5 \end{bmatrix}$$

Using matrix multiplication rules, we see that this is equivalent to our original set of two equations:

$$\begin{aligned} 3x_1 + 4x_2 &= 8 \\ 2x_1 + 9x_2 &= 5 \end{aligned}$$

We'll first solve this the old-school way: just by working with the equations themselves; then we'll look at how this is generalized into matrix manipulations, and look at practical computational methods.

In our first algebra courses, we probably learned that we should start by isolating x_1 , in terms of x_2 , in either equation, then plug that back into the other equation. As we got more advanced, we learned that we can also add together the equations or multiples thereof; we can always add one equation to another since both sides of an equation are equal—we are adding the same thing to both sides. So let's make a new equation by adding together double the first equation and triple the second:

$$\begin{array}{r} 6x_1 + 8x_2 = 16 \\ -6x_1 - 27x_2 = -15 \\ \hline -19x_2 = 1 \end{array}$$

This is perhaps a bit more efficient than the isolation approach, as we got to the answer for x_2 in a single step.

Now we have a third equation ($19x_2 = 1$); but it is not independent of the other two. That is, we have not added any new information to our system. So, we can and should drop an equation and we can choose to drop either of the parent equations. Let's drop the second, and our system of equations is now

$$\begin{aligned} 3x_1 + 4x_2 &= 8 \\ -19x_2 &= 1 \end{aligned}$$

Once our equations are in this form, we can immediately get numerical numbers. (The bottom equation gives $x_2 = -\frac{1}{19}$, then if we plug this into the top equation we get $x_1 = \frac{52}{19}$.) We can deduce that if we had a larger set of equations, we could perform additional similar operations until we could stack them in such a “triangular” form, with only one unknown variable in the bottom equation, two in the next, three above that, etc. Once we achieve this “triangular”, our system is effectively solved.

In linear algebra, we might express this as an *augmented matrix*, where $\underline{\underline{\mathbf{A}}}$ and $\underline{\mathbf{b}}$ are listed adjacent to one another as $[\underline{\underline{\mathbf{A}}} | \underline{\mathbf{b}}]$. Our original set of equations, in augmented form, would be

$$\left[\begin{array}{cc|c} 3 & 4 & 8 \\ 2 & 9 & 5 \end{array} \right]$$

After our algebra, our “solved” system would be

$$\left[\begin{array}{cc|c} 3 & 4 & 8 \\ 0 & -19 & 1 \end{array} \right]$$

As above, we can see that this has a “triangular form”, and specifically we would call the left-hand side of this augmented matrix an “upper triangular” matrix, which means it has only zeros below the diagonal. Once our matrix is in upper triangular form, the system is effectively solved.

1.3.2 Row operations

Reference: K7.3

Since the rows of our matrix correspond to equations, any operations that are allowed between two equations are allowed between two rows. In matrix algebra, we call these “row operations”. Specifically, this means that in our matrix form we are allowed to:

- Multiply any row by a (non-zero) constant.
- Replace any row by its sum with another row.
- Swap any two rows.⁷

1.3.3 Gauss elimination

Reference: K7.3

The procedure we performed in §1.3.1, when performed with row operations on an augmented matrix, is known as *Gauss elimination*. Here, we’ll demonstrate the procedure through an example, which is a modified version of Problem K7.3.4. We’ll use nomenclature like R_2 to mean row 2, and the arrow means replace with.

$$\left[\begin{array}{ccc|c} 4 & 1 & 0 & 4 \\ 5 & -3 & 1 & 2 \\ -9 & 2 & 1 & 5 \end{array} \right]$$

- $R_2 \leftarrow 4R_2 - 5R_1$
- $R_3 \leftarrow 4R_3 + 9R_1$

$$\left[\begin{array}{ccc|c} 4 & 1 & 0 & 4 \\ 0 & -17 & 4 & -12 \\ 0 & 17 & 4 & 56 \end{array} \right]$$

- $R_3 \leftarrow R_3 + R_2$

$$\left[\begin{array}{ccc|c} 4 & 1 & 0 & 4 \\ 0 & -17 & 4 & -12 \\ 0 & 0 & 8 & 44 \end{array} \right]$$

It is now in a form that it can be rapidly solved for \underline{x} . This is called upper-triangular form.

What we’ve shown here is the most basic form, in practice, if zeros appear in the original matrix (or at any point) the rows (equation order) or columns (variable order) should be re-organized to put the zeros below the diagonal. Additionally, partial pivoting should be implemented, as noted above. These refinements are included in any modern computational package that performs Gauss elimination.

⁷This can increase the stability of an algorithm. If two rows are swapped to place the largest on the diagonal, this is called partial pivoting, and makes the method more numerically stable.

Computational cost of Gauss elimination

How many flops are required for Gauss elimination? Assume the matrix size is $n \times n$.

To clear one value and change it to zero I do a flop; that is, I multiply the value in the row I care about by something to scale it to the same value as my pivot row, then I subtract them. However, I have to do this for every other value in my row as well. So, roughly, to change a value into a zero requires n calculations.

My matrix has n^2 values. Below the diagonal are $n^2/2 - n$ values, which we can approximate as $n^2/2$ for large n . So to clear below the diagonal we need something proportional to n^3 ; that is, $\mathcal{O}(n^3)$. This can become quite expensive for large systems.

Recall that our system is $\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$. Interestingly, solving for $\underline{\underline{\mathbf{b}}}$ (matrix multiplication) is $\mathcal{O}(n^2)$, but solving for $\underline{\underline{\mathbf{x}}}$ is $\mathcal{O}(n^3)$. This is quite different than the analogous (scalar) algebraic problem ($ax = b$), where the cost of solving for b or x is equivalent.

1.3.4 Matrix inversion

Reference: K7.8

Earlier, we determined that we can compactly write a solution to this problem as

$$\underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{A}}}^{-1} \underline{\underline{\mathbf{b}}}$$

To use this in practice, we first must find $\underline{\underline{\mathbf{A}}}^{-1}$, (an $n \times n$ matrix) and then multiply that by $\underline{\underline{\mathbf{b}}}$. (This is why we say that matrix inversion and solving a linear system are equivalent.) How do we find the inverse of a matrix?

A common method is Gauss–Jordan elimination. A concise derivation of why this works is given in K7.8; we won't reproduce the derivation here. However, the general method is to construct an augmented matrix $\underline{\underline{\tilde{\mathbf{A}}}}$:

$$\underline{\underline{\tilde{\mathbf{A}}}} = [\underline{\underline{\mathbf{A}}} \mid \underline{\underline{\mathbf{I}}}]$$

Next, we perform row operations until we have turned the left side of this augmented matrix into the identity matrix. When that occurs, the right-hand side is the inverse:

$$[\underline{\underline{\mathbf{A}}} \mid \underline{\underline{\mathbf{I}}}] \xrightarrow{\text{row operations}} [\underline{\underline{\mathbf{I}}} \mid \underline{\underline{\mathbf{A}}}^{-1}]$$

Hand-wavy derivation

To help justify what is given more rigorously in the textbook, we offer a brief explanation here. In short, we can think of trying to find the matrix $\underline{\underline{\mathbf{X}}}$ in

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{X}}} = \underline{\underline{\mathbf{I}}}$$

This looks very similar to our Gauss elimination problem $\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$; we solved that one using an augmented matrix $[\underline{\underline{\mathbf{A}}} \mid \underline{\underline{\mathbf{b}}}]$. By analogy, we can solve this one as

$$[\underline{\underline{\mathbf{A}}} \mid \underline{\underline{\mathbf{I}}}]$$

If we do row operations to make the left-hand side turn into the identity matrix, we get

$$[\underline{\underline{\mathbf{I}}} \mid \underline{\underline{\mathbf{K}}}]$$

We can re-construct what equation this came from by going in reverse:

$$\underline{\underline{\mathbf{I}}} \underline{\underline{\mathbf{X}}} = \underline{\underline{\mathbf{K}}}$$

Since $\underline{\underline{\mathbf{I}}} \underline{\underline{\mathbf{X}}} = \underline{\underline{\mathbf{X}}}$, then $\underline{\underline{\mathbf{X}}} = \underline{\underline{\mathbf{K}}}$, and since $\underline{\underline{\mathbf{X}}}$ is what we are looking for we are done.

Computational cost

The order of this is $\mathcal{O}(n^3)$, but carries a higher pre-factor because we need to put zeros on both sides of the diagonal, and do row operations across a wider augmented matrix. So, in practice, this method is slower than Gaussian elimination, but scales the same.

Why bother?

If it's more expensive than Gauss elimination, why bother with this method? It can still be useful if you encounter a situation where you need to solve many related problems; e.g., solving $\underline{\underline{\mathbf{A}}}\underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$, then $\underline{\underline{\mathbf{A}}}\underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{c}}}$, etc. If you just compute $\underline{\underline{\mathbf{A}}}^{-1}$ once, then the subsequent solutions are nearly free.

1.3.5 Solving linear systems in practice

In practice, you should *not* code your own software to solve $\underline{\underline{\mathbf{A}}}\underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$; many more-expert people than us have spent much time developing optimized algorithms for this problem! In this class, we use `numpy.linalg.solve` to solve linear systems.

If we check the documentation for this function, we find that it is actually using a standardized library call from “LAPACK”. These are highly optimized Fortran/C libraries that solve this problem, and are called upon by many numerical programs. However, if we check the documentation for LAPACK's solve function, we find that it's actually using neither Gaussian elimination nor matrix inversion; instead, it's using something called LU decomposition. In the following section, we'll look briefly at that method.

1.3.6 LU decomposition

Reference: K20.2

The name comes from the fact that $\underline{\underline{\mathbf{L}}}$ and $\underline{\underline{\mathbf{U}}}$ are lower and upper triangular matrices, respectively. That is, they look like:

$$\underline{\underline{\mathbf{L}}} = \begin{bmatrix} \# & 0 & 0 & 0 & 0 \\ \# & \# & 0 & 0 & 0 \\ \# & \# & \# & 0 & 0 \\ \# & \# & \# & \# & 0 \\ \# & \# & \# & \# & \# \end{bmatrix} \quad \underline{\underline{\mathbf{U}}} = \begin{bmatrix} \# & \# & \# & \# & \# \\ 0 & \# & \# & \# & \# \\ 0 & 0 & \# & \# & \# \\ 0 & 0 & 0 & \# & \# \\ 0 & 0 & 0 & 0 & \# \end{bmatrix}$$

Starting with our problem:

$$\underline{\underline{\mathbf{A}}}\underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$$

Let's assume the matrix $\underline{\underline{\mathbf{A}}}$ can be composed into $\underline{\underline{\mathbf{A}}} = \underline{\underline{\mathbf{L}}}\underline{\underline{\mathbf{U}}}$; as noted in K20, any matrix that can be inverted can also be decomposed into this form. Then we can write

$$\underline{\underline{\mathbf{L}}}\underline{\underline{\mathbf{U}}}\underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$$

The product $\underline{\underline{\mathbf{U}}}\underline{\underline{\mathbf{x}}}$ gives a vector, let's refer to it as $\underline{\underline{\mathbf{y}}}$; that is, we'll substitute $\underline{\underline{\mathbf{U}}}\underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{y}}}$:

$$\underline{\underline{\mathbf{L}}}\underline{\underline{\mathbf{y}}} = \underline{\underline{\mathbf{b}}}$$

That is, the procedure is to first decompose $\underline{\underline{\mathbf{A}}}$ into $\underline{\underline{\mathbf{L}}}$ and $\underline{\underline{\mathbf{U}}}$. Then, solve $\underline{\underline{\mathbf{L}}}\underline{\underline{\mathbf{y}}} = \underline{\underline{\mathbf{b}}}$ for $\underline{\underline{\mathbf{y}}}$. From this, solve $\underline{\underline{\mathbf{U}}}\underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{y}}}$ to find the desired result, $\underline{\underline{\mathbf{x}}}$.

The computationally intensive step, scaling with $\mathcal{O}(n^3)$ (but with a prefactor about half that of Gaussian elimination) is decomposing $\underline{\underline{\mathbf{A}}}$ into $\underline{\underline{\mathbf{L}}}$ and $\underline{\underline{\mathbf{U}}}$. After this, since the matrices are already triangular, the subsequent solving for $\underline{\underline{\mathbf{y}}}$ and then $\underline{\underline{\mathbf{x}}}$ turns out to be comparatively cheap ($\mathcal{O}(n^2)$).

1.3.7 Iterative approaches

Reference: K20.3

The above approaches all give exact solutions to $\underline{\mathbf{A}} \underline{\mathbf{x}} = \underline{\mathbf{b}}$. You can save a lot of computational expense by only solving for an *approximate* solution, in an iterative manner. Iterative solutions, in principle, tend towards the exact solution as the number of iterations increases.

For linear systems, a technique known as Gauss–Seidel iteration (K20.3) can be used. We will delay discussion of iterative techniques until our discussion of eigenvalue problems, where they are essential, as exact solutions are computationally intractable.

1.3.8 Understanding solution space (rank)

Reference: K7.4

There have been a couple of questions on how we can know if the system $\underline{\mathbf{A}} \underline{\mathbf{x}} = \underline{\mathbf{b}}$ has a solution. It's relatively straightforward to understand this if we just think in terms of the underlying algebraic equations—here, we'll do that, and then translate this into the language of matrices: namely *rank* and *singular matrices*. We'll look at things that can happen in the smallest systems possible, and try to generalize to larger matrices. Here, we will always assume that our matrix $\underline{\mathbf{A}}$ is square ($n \times n$), and therefore both our vectors ($\underline{\mathbf{x}}$ and $\underline{\mathbf{b}}$) are column vectors of length n .

Case 1: redundant equations

Let's illustrate this with an example. Say we have the following system of equations:

$$\begin{aligned}x_1 + 2x_2 &= 4 \\2x_1 + 4x_2 &= 8\end{aligned}$$

Obviously, those two equations are identical; that is, they give us the same information about the relationship between x_1 and x_2 . However, let's pretend we don't know this and try to solve the system. If we do our standard Gauss elimination row operation—that is, subtracting $2 \times$ the first equation from the second, our system reduces to

$$\begin{aligned}x_1 + 2x_2 &= 4 \\0 + 0 &= 0\end{aligned}$$

Generally, we say that these equations are not *linearly independent*. Although it was obvious from the start in this case, this won't always be immediately obvious in large systems. Another example in which the equations can violate linear independence is if we have three equations and the third equation can be assembled by summing the first two; we would also get a row of zeros from Gauss elimination in that case.

In this system, we can conclude that there is not a *unique* solution (x_1, x_2) , but rather there is an *infinite family* of solutions. That is, if we pick x_1 (let's pick 3), then we know x_2 (which will be $\frac{1}{2}$ in this case). But we are not restricted in what we pick for x_1 , so this is an infinite set of solutions. In this 2-dimensional example, we can think of the infinite solution space as being constrained to a single line on the x_1 - x_2 plane.

Matrix interpretation. In augmented matrix form our (reduced) system is:

$$\left[\begin{array}{cc|c} 1 & 2 & 4 \\ 0 & 0 & 0 \end{array} \right]$$

This is a general feature if we have equations that are not linearly independent: we get rows of zeros at the bottom of our augmented matrix. We say that the *rank* is defined as the number of non-zero rows of this

matrix; in this case, our rank is 1. If the rank of the augmented matrix is less than the length⁸ of $\underline{\mathbf{x}}$; then our system is under-determined, and we have an infinite family of solutions. That is, this situation occurs when:

$$\text{rank } [\underline{\mathbf{A}}|\underline{\mathbf{b}}] < \text{len } \underline{\mathbf{x}}$$

Note that if we had performed row operations on $\underline{\mathbf{A}}$ (as opposed to the augmented matrix $[\underline{\mathbf{A}}|\underline{\mathbf{x}}]$), we would also have found that the rank of $\underline{\mathbf{A}}$ was less than the length of $\underline{\mathbf{x}}$, which is an indication of a problem, but doesn't distinguish between this case and the next case.

Case 2: conflicting equations

Now, let's instead say we have the following equations:

$$x_1 + 2x_2 = 4$$

$$x_1 + 2x_2 = 3$$

Obviously, both equations cannot be simultaneously satisfied, as the left-hand side of both equations is identical. If we tried to do row operations, we would get the equation $0 = 1$, which is nonsensical. Therefore, this system has *no solution*.

Matrix interpretation. Note that the matrix $\underline{\mathbf{A}}$ is the same as the previous example so again has rank 1, which is less than the length of $\underline{\mathbf{x}}$ —which indicates we have a problem. In the previous case, this indicated that we had infinite solutions; in this case, it indicates we have no solutions.

We already determined that $\text{rank}(\underline{\mathbf{A}}) = 1$ and $\text{len}(\underline{\mathbf{x}}) = 2$. Let's examine our augmented matrix:

$$\left[\begin{array}{cc|c} 1 & 2 & 4 \\ 1 & 2 & 3 \end{array} \right]$$

Perform Gauss elimination to find the rank:

$$\left[\begin{array}{cc|c} 1 & 2 & 4 \\ 0 & 0 & 1 \end{array} \right]$$

Here, the bottom row is not all zeros,⁹ so $\text{rank}([\underline{\mathbf{A}}|\underline{\mathbf{x}}]) = 2$. This turns out to be the criterion we can use: if the rank of the augmented matrix is larger than the rank of $\underline{\mathbf{A}}$, then we have no solution:

$$\text{rank}([\underline{\mathbf{A}}|\underline{\mathbf{x}}]) > \text{rank } \underline{\mathbf{A}}$$

Since we're typically interested in finding a unique solution to $\underline{\mathbf{A}} \underline{\mathbf{x}} = \underline{\mathbf{b}}$, we can use the simple criterion that if $\text{rank}(\underline{\mathbf{A}}) < \text{len}(\underline{\mathbf{x}})$, we do not have a unique solution. That is, it is entirely determined by the matrix $\underline{\mathbf{A}}$; the make-up of $\underline{\mathbf{b}}$ will only determine if we have no solutions or infinite solutions.

Case 3: homogeneous systems

There is one special case that comes up often that we should examine: when $\underline{\mathbf{b}} = \underline{\mathbf{0}}$; that is, when our system is

$$\underline{\mathbf{A}} \underline{\mathbf{x}} = \underline{\mathbf{0}}$$

where $\underline{\mathbf{0}}$ is a column vector that only contains zeros; $\underline{\mathbf{0}} = [0 \ 0 \ \dots \ 0]^T$. In this case, we can immediately see that the solution $\underline{\mathbf{x}} = \underline{\mathbf{0}}$ always works. This is often referred to as the trivial solution.

However, is there another solution beyond this? Let's examine a couple of scenarios.

⁸Here, by length of $\underline{\mathbf{x}}$ we just mean the number of unknown variables; here $\underline{\mathbf{x}} = [x_1 x_2]^T$ so the length is two. No mathematical operations are used here.

⁹(Although it is nonsensical if put into equation form.)

- First, let's examine $\underline{\underline{\mathbf{A}}} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$, plugging this into $\underline{\underline{\mathbf{A}}} \mathbf{x} = \underline{\mathbf{0}}$ gives the equations:

$$\begin{aligned} x_1 + 0 &= 0 \\ 0 + 0 &= 0 \end{aligned}$$

Therefore, x_1 must be zero, but x_2 is unconstrained! That is, we can choose anything for x_2 . So we have found a non-trivial solution to this problem beyond $\mathbf{x} = \underline{\mathbf{0}}$.

- For the second case, let's just swap the 0's and 1's and examine $\underline{\underline{\mathbf{A}}} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$, which gives

$$\begin{aligned} 0 + x_2 &= 0 \\ x_1 + x_2 &= 0 \end{aligned}$$

The only values of x_1 and x_2 that solve this system are 0 and 0, that is, our trivial solution. So this system has no non-trivial solution. (The only solution is $\mathbf{x} = \underline{\mathbf{0}}$.)

Understanding in terms of matrix properties. Again, we can use the rank to differentiate these systems; but now we don't have to worry about the augmented matrix. Specifically, if the $\text{rank}(\underline{\underline{\mathbf{A}}}) < \text{len}(\mathbf{x})$, then we have non-trivial solutions. If the $\text{rank}(\underline{\underline{\mathbf{A}}}) = \text{len}(\mathbf{x})$, we only have the trivial solution.

1.3.9 Rank, and singular matrices

We can summarize the conclusions from the previous section, and use this to define singular matrices.

First, if the rank of $\underline{\underline{\mathbf{A}}}$ is less than the number of unknowns (n , which is the length of \mathbf{x}), then there is something funny. That is, there is not a unique solution, whether our system is $\underline{\underline{\mathbf{A}}} \mathbf{x} = \underline{\mathbf{b}}$ or $\underline{\underline{\mathbf{A}}} \mathbf{x} = \underline{\mathbf{0}}$. In the general case $\underline{\underline{\mathbf{A}}} \mathbf{x} = \underline{\mathbf{b}}$, this can either mean there is no solution or that there are infinite solutions. In the homogeneous case $\underline{\underline{\mathbf{A}}} \mathbf{x} = \underline{\mathbf{0}}$, this means that there are other solutions in addition to the trivial solution.

Generally, if $\text{rank}(\underline{\underline{\mathbf{A}}}) < n$, where $\underline{\underline{\mathbf{A}}}$ is an $n \times n$ matrix, then we call this a *singular matrix*. As we noted earlier, singular matrices are ones which cannot be inverted; that is, $\underline{\underline{\mathbf{A}}}^{-1}$ does not exist.

1.3.10 Relation between rank and determinant

Finally, we can note that the rank can also be determined from the determinant of a matrix. Specifically, it can be shown (K7.7) that for an $n \times n$ matrix $\underline{\underline{\mathbf{A}}}$, the rank is equal to n iff¹⁰

$$\det(\underline{\underline{\mathbf{A}}}) \neq 0$$

See K7.7 if you need a re-fresher on how to evaluate determinants. For a 2×2 matrix, it is just the down-diagonal minus the up-diagonal:

$$\det \left(\begin{bmatrix} a & b \\ c & d \end{bmatrix} \right) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - cb$$

where we conventionally use the straight brackets to indicate a determinant.

Thus, we can equivalently state that for a homogeneous system, if $\det \underline{\underline{\mathbf{A}}} = 0$, we have non-trivial solutions. If $\det \underline{\underline{\mathbf{A}}} \neq 0$, we have only the trivial solution. This is known as (part of) *Cramer's rule*.

1.4 Eigenvalue problems: “diagonalizing a matrix”

Reference: K8, B3

¹⁰‘iff’ means ‘if and only if’.

The eigenvalue problem is typically a very abstract concept when it is first encountered in undergraduate linear algebra courses. However, the problem is extremely common and important, and has a massive number of applications. In class, we discussed its application in music, chemistry, global warming, Google's success, process control, quantum mechanics, electronics, computational fluid mechanics, and special relativity. Further, we noted that it's perhaps one of the most compute-consuming operations in all of computations, since it bottlenecks quantum chemistry computations (which are the dominant user of large scientific computing resources) and is used extensively in Silicon Valley-style search optimizations.

1.4.1 What is an eigenvalue problem?

First, consider the operation of a matrix on a vector. That is, if we multiply a square matrix by a vector we get another vector:

$$\underline{\underline{\mathbf{A}}} \underline{\mathbf{x}} = \underline{\mathbf{y}}$$

Therefore, the matrix can be thought of as transforming one vector into another. And if the matrix is square, it transforms one vector into another of the same size. For example, a matrix might rotate a vector into a new direction, convert a vector from one coordinate system to another (like from Cartesian to polar coordinates), or even take the derivative of a vector.

However, occasionally when a matrix operates on a vector we get the same vector back again! (with a multiplicative constant, λ)

$$\underline{\underline{\mathbf{A}}} \underline{\mathbf{x}} = \lambda \underline{\mathbf{x}}$$

When this happens, it is called an eigenvalue problem. λ is known as an eigenvalue, and $\underline{\mathbf{x}}$ is known as an eigenvector.

This is conventionally re-written with the rules of matrix manipulation (§1.1.2) as

$$(\underline{\underline{\mathbf{A}}} - \lambda \underline{\underline{\mathbf{I}}}) \underline{\mathbf{x}} = \underline{\mathbf{0}}$$

We should emphasize that eigenvalues/eigenvectors are properties of the matrix $\underline{\underline{\mathbf{A}}}$ —conventionally we think of starting with a matrix $\underline{\underline{\mathbf{A}}}$ and finding possible eigenvalue/eigenvector pairs.

By Cramer's rule, this has a non-trivial solution iff the determinant of the quantity in parentheses is zero. We probably learned in a linear algebra course that this is used to find the eigenvalues:

$$\det(\underline{\underline{\mathbf{A}}} - \lambda \underline{\underline{\mathbf{I}}}) = 0$$

1.4.2 Example: oscillating springs

Eigenvalue problems can arise all over the place. However, to ground us, let's look at a physical example of where an eigenvalue problem arises, and derive it from first principles. We will examine a system of masses coupled with springs as shown in the bottom of Figure 1.1.

Let's first recall how the spring constant is defined for a single spring, as shown in the top of the figure. Here, q is the displacement of the mass from its equilibrium (zero-force) position. Hooke's Law gives us:

$$F = -kq$$

$$\frac{dF}{dq} = -k$$

where the second line is the derivative of the first. We can say that dF/dq is a good definition of the spring constant, k —it is how much the force changes when we displace the spring away from its equilibrium position.

Now let's examine the system with many masses and many springs; for simplicity in the figure we show them each only connected to their nearest neighbor, but we'll write the equations generically to allow for springs between any pair of masses. The effective force constants, which we'll just call $k_{i,j}$'s, are defined

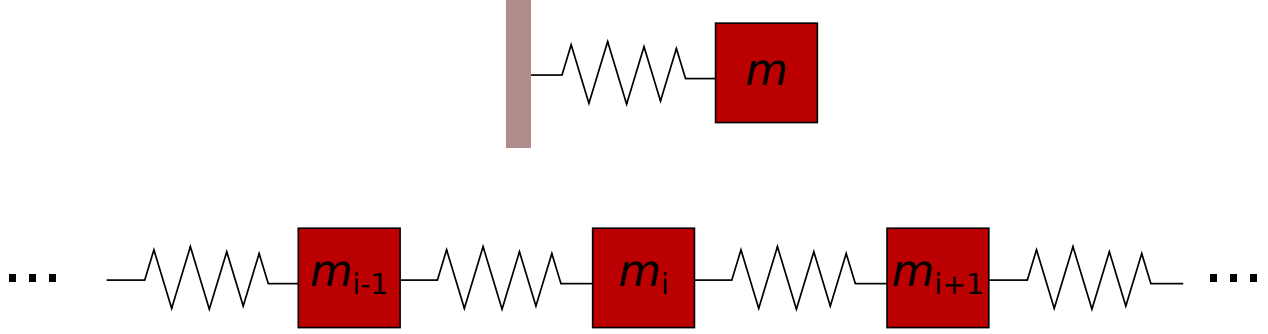


Figure 1.1: (Top) A single mass oscillating on a spring. (Bottom) A system of masses, vibrating on springs that interact with neighbors.

under the scenario of keeping all the masses fixed except for one, and perturbing that one. Then, we look at the force not just on that mass, but on all the other masses as well in order to define force constants.

Similar to the one-mass problem, let's define q_i as the displacement on mass i away from its equilibrium position. Mathematically, we can say that each spring constant is defined as

$$k_{i,j} \equiv - \left(\frac{\partial F_j}{\partial q_i} \right)_{q_j \neq i} \quad (1.2)$$

From equation (1.2), we can write the force on mass j as

$$F_j = - \sum_{i=1}^N k_{i,j} q_i$$

Using Newton's second law, $F = ma$,

$$m_j \frac{d^2 q_j}{dt^2} = - \sum_{i=1}^N k_{i,j} q_i$$

To make the equations a bit cleaner we fold the mass into the q 's and k 's, with

$$\begin{aligned} \tilde{q}_i &\equiv q_i \sqrt{m_i}, & \tilde{k}_{i,j} &\equiv \frac{k_{i,j}}{\sqrt{m_i m_j}} \\ \frac{d^2 \tilde{q}_j}{dt^2} &= - \sum_{i=1}^N \tilde{k}_{i,j} \tilde{q}_i \end{aligned} \quad (1.3)$$

You might recall from your differential equations class that this is a system of first-order, homogeneous differential equations. This system has a generic¹¹ solution of

$$\tilde{q}_j = A_j \cos \omega t \quad (1.4)$$

If we plug this into our differential equation, (1.3), this leads directly¹² to

$$-\omega^2 A_i = - \sum_j \tilde{k}_{i,j} A_j$$

or

¹¹When this material was presented in class, we used the trial solution of $\tilde{q}_j = A_j e^{i\omega t}$. Because we are trying to minimize the use of imaginary numbers in this course, I have changed the trial solution to the cosine form. The math works out the same, but we avoid imaginary numbers and Euler's formula.

¹²Note that we switched the order of the indices i and j when we plugged into the differential equation.

$$0 = \sum_j \tilde{k}_{i,j} A_j - \omega^2 A_i$$

which is our eigenvalue problem. We can see that the matrix elements are $\tilde{k}_{i,j}$, the column vector is A_j , and the eigenvalues are ω^2 .

E.g., for $N = 3$ we get

$$\begin{bmatrix} \tilde{k}_{1,1} - \omega^2 & \tilde{k}_{1,2} & \tilde{k}_{1,3} \\ \tilde{k}_{2,1} & \tilde{k}_{2,2} - \omega^2 & \tilde{k}_{2,3} \\ \tilde{k}_{3,1} & \tilde{k}_{3,2} & \tilde{k}_{3,3} - \omega^2 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

or

$$(\underline{\mathbf{k}} - \omega^2 \underline{\mathbf{I}}) \underline{\mathbf{A}} = \underline{\mathbf{0}}$$

Therefore, if we can find the eigenvalues ω^2 of this problem we have found the characteristic frequencies of our system of coupled oscillators. Whereas the eigenvectors will give the directions of motion, $\underline{\mathbf{A}}$, in which each spring vibrates. That is, for every eigenvalue/eigenvector pair that we find, we can use equation (1.4) to write the vibration's motion as

$$\underline{\tilde{\mathbf{q}}} = \underline{\mathbf{A}} \cos \omega t$$

where $\underline{\tilde{\mathbf{q}}}$ has components $[\tilde{q}_i]$. Note that the magnitude of $\underline{\mathbf{A}}$ is arbitrary in this treatment.

1.4.3 Solving eigenvalue problems the textbook way

Reference: K8.1

Typically, we are introduced to eigenvalue problems in a linear algebra course, and the solution method is as described in this section. Here, we'll go over this method, and note that it really isn't applicable to anything other than the most trivial-sized eigenvalue problems; that is, perhaps 4×4 matrices. Whereas in typical computational applications, our matrices might have millions of rows and columns, and we'll quickly see that the "textbook" approach does not scale.

To remind us, the eigenvalue problem is formulated as:

$$\underline{\underline{\mathbf{A}}} \underline{\mathbf{x}} = \lambda \underline{\mathbf{x}}$$

$$(\underline{\underline{\mathbf{A}}} - \lambda \underline{\mathbf{I}}) \underline{\mathbf{x}} = \underline{\mathbf{0}}$$

As we noted earlier, there will always be the solution $\underline{\mathbf{x}} = \underline{\mathbf{0}}$; if the system is "well-behaved" this will be the only solution and this is not a very interesting problem. Therefore, for this system to have more than a single solution we require that the matrix $(\underline{\underline{\mathbf{A}}} - \lambda \underline{\mathbf{I}})$ be singular—which means that we will either have infinite solutions or zero solutions. Since we know that it won't be zero solutions (we already have found one!) we know that this will result in infinite solutions.

One way to state that a matrix is singular is if its determinant is zero:

$$\det(\underline{\underline{\mathbf{A}}} - \lambda \underline{\mathbf{I}}) = 0$$

This then becomes the governing equation of the classic, textbook way to solve for eigenvalue problems, which we'll show by means of an example.

Worked example on a 2×2 matrix

Let's examine Example 1 of Section K8.1 in the textbook, in which we are given a 2×2 matrix and asked to find its eigenvalues and eigenvectors. The matrix $\underline{\underline{\mathbf{A}}}$ is

$$\underline{\underline{\mathbf{A}}} = \begin{bmatrix} -5 & 2 \\ 2 & -2 \end{bmatrix}$$

The term $\lambda \underline{\underline{\mathbf{I}}}$ is:

$$\lambda \underline{\underline{\mathbf{I}}} = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix}$$

We want to solve for λ in the determinant:

$$|\underline{\underline{\mathbf{A}}} - \lambda \underline{\underline{\mathbf{I}}}| = \begin{vmatrix} -5 - \lambda & 2 \\ 2 & -2 - \lambda \end{vmatrix} = 0$$

which gives the algebraic expression below, the roots of which are the eigenvalues (λ).

$$(-5 - \lambda)(-2 - \lambda) - 4 = 0$$

$$10 + 7\lambda + \lambda^2 - 4 = 0$$

$$\lambda^2 + 7\lambda + 6 = 0$$

$$(\lambda + 6)(\lambda + 1) = 0$$

$$\boxed{\lambda = -1, -6}$$

So now we found two solutions, in terms of λ , that appear to solve our eigenvalue expression (or at least for which $\det(\underline{\underline{\mathbf{A}}} - \lambda \underline{\underline{\mathbf{I}}}) = 0$). However, didn't we expect from the previous discussion that we would find infinite solutions? Before we resolve that question, let's find the associated eigenvectors.

Finding the eigenvectors. Now that we have the eigenvalues, we can find the eigenvectors by returning to the original equation

$$(\underline{\underline{\mathbf{A}}} - \lambda \underline{\underline{\mathbf{I}}}) \underline{\underline{\mathbf{x}}} = 0$$

so let's now find the $\underline{\underline{\mathbf{x}}}$'s by plugging in the numbers to the matrices:

$$\begin{bmatrix} -5 - \lambda & 2 \\ 2 & -2 - \lambda \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

That is, we have to solve a linear system to find the eigenvector associated with each eigenvalue. Plugging in $\lambda = -1$:

$$\begin{bmatrix} -4 & 2 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

You may notice already that something funny is going on—we have the same equation listed twice! (That is, the equations are as below.)

$$-4x_1 + 2x_2 = 0$$

$$2x_1 - 1x_2 = 0$$

This was actually to be expected—actually, we *required* this by our original statement that the matrix must be singular.

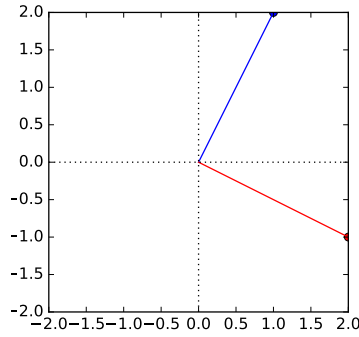


Figure 1.2: The eigenvectors of a simple eigenvalue problem.

Now, we only have one equation, and the best we can do is plug in something arbitrary for either x_1 or x_2 . Let's choose $x_1 = 1$. From either of our equations, we have $x_2 = 2x_1$, which means $x_2 = 2$. We then get

$$\underline{\mathbf{x}}^T = [1 \quad 2]$$

If we had chosen $x_1 = 2$, we would have gotten $x_2 = 4$, or $\underline{\mathbf{x}}^T = [2 \quad 4]$. It seems we can only get this vector to within an arbitrary constant; that is, all vectors $c\underline{\mathbf{x}}$ are also apparently eigenvectors. But perhaps if we look at the original form of the eigenvalue equation this is not surprising.

$$\underline{\underline{\mathbf{A}}} \underline{\mathbf{x}} = \lambda \underline{\mathbf{x}}$$

Let's say we had chosen $c\underline{\mathbf{x}}$ instead of $\underline{\mathbf{x}}$, then we have

$$\underline{\underline{\mathbf{A}}} c\underline{\mathbf{x}} = \lambda c\underline{\mathbf{x}}$$

The constant comes out of both sides and we recover $\underline{\underline{\mathbf{A}}} \underline{\mathbf{x}} = \lambda \underline{\mathbf{x}}$; that is, $c\underline{\mathbf{x}}$ is also an eigenvector associated with the eigenvalue λ . So we *do* get an infinite set of solutions, as predicted!

If we think of vectors as having directions and magnitudes, this means that only the direction of an eigenvector is meaningful.

We still have one more eigenvalue to examine: $\lambda = -6$. If we plug in the -6 root, we get $\underline{\mathbf{x}} = [2 \quad -1]$, with the same issue, and our analysis is the same: any constant times this eigenvector is also an eigenvector.

Full solution to this example problem. Thus, we *do* in fact find infinite solutions, as we had stated that we expected to above. Precisely, we find the following sets of solutions to our eigenvalue problem:

- $\underline{\mathbf{x}} = \underline{\mathbf{0}}$, where λ can be anything (except perhaps ∞). This is our trivial solution, and is typically not interesting.
- $\lambda = -1$, $\underline{\mathbf{x}}^T = c [1 \quad 2]$, where c is any finite, non-zero constant.
- $\lambda = -6$, $\underline{\mathbf{x}}^T = c [2 \quad -1]$, where c is any finite, non-zero constant.

Plotting the solution. Let's plot these—see Figure 1.2. Note they are at right angles—that is, they are orthogonal. Will this always be the case? It turns out, if the matrix is symmetric then yes, this will always be the case. This is a very useful property we will discuss later.

Normalizing the eigenvectors. Since the magnitude of the eigenvectors is arbitrary (only the direction matters), it is common to normalize them to have magnitude of 1. Here, we define¹³ the length of a vector to be

$$\sqrt{x_1^2 + x_2^2}$$

and we can see that for both eigenvectors we get $\sqrt{1+4} = \sqrt{5}$. Thus, we might report the eigenvectors as $\frac{1}{\sqrt{5}} \begin{bmatrix} 1 & 2 \end{bmatrix}$ and $\frac{1}{\sqrt{5}} \begin{bmatrix} 2 & -1 \end{bmatrix}$.

1.4.4 Can we apply the textbook approach to larger systems? (Absolutely not.)

Let's pause here and note a couple of things about the computation.

1. The procedure is straightforward. That is to say, we just follow the same pattern as above for any other 2×2 matrix we encounter.
2. However, matrices with dimensions in the millions are common in computational problems. So we need to figure out if this works for larger systems.
3. First, note we need to take a determinant. Computationally, that scales¹⁴ with $\mathcal{O}(n!)$. That is about as bad as it gets. Note this is much worse than n^2 or n^3 ...¹⁵ So in practice, we have to throw an immense amount of processor power at this problem to solve it.
4. If we are patient enough to solve the determinant, we get an algebraic problem to solve for λ ; that is, we get an n th-order polynomial, where n is the size of our square matrix.

In our example, it's just a simple quadratic (λ^2), which we have standard formulae to solve. This is also true for 3rd- and 4th-order polynomials, so we can solve 3×3 and 4×4 matrices in this manner.

However, for a general $n \times n$ matrix we have an n th-order polynomial to solve, but unfortunately there is no method to solve polynomials greater than 4th order.¹⁶

Therefore, we absolutely cannot take this general approach, and must use methods that find approximate solutions to the eigenvalue problem.

1.4.5 Bounding eigenvalues with Gerschgorin's Theorem

Reference: K20.7, B3:111

To motivate the computational approaches of estimating eigenvalues, we'll start with a simple "trick" that can be used to quickly estimate eigenvalues, as well as give uncertainty. Here, we'll teach the trick itself, and leave the theoretical background to the reading.

The trick is that the diagonal elements of a matrix give an estimate of the eigenvalues, and the remainder of each row gives an estimate of the uncertainty. That is, we can estimate each eigenvalue from each diagonal element from

$$|a_{jj} - \lambda| \leq \sum_{k \neq j} |a_{jk}|$$

In words: the estimate is the diagonal entry; the error bounds are the sum of the rest of the row.

In general, this applies for both real and complex matrices. (And a matrix consisting of only real numbers can have complex eigenvalues!) So these are interpreted as Gerschgorin discs in the complex plane: every eigenvalue lies inside a disc.

¹³Note that this is our common definition of a vector magnitude (or length) from high school, but other definitions are possible. We'll mention those other definitions in a future section; §1.5.1.

¹⁴https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations

¹⁵It can be easiest to understand if we just think of the power: n^2 is power 2, meaning $\log_{10} n^2 = 2$, and n^3 is power 3. So to compare, we can take a log of $n!$. Using Stirling's approximation $\ln n! \approx n \ln n$. Ouch.

¹⁶In fact, if you look at how numpy finds the roots of a polynomial, you will see that it does so by estimating the eigenvalues of the "companion matrix". Thus, finding the roots of a polynomial and the eigenvalues of a matrix are really equivalent problems, and both are impossible to do exactly for large systems.

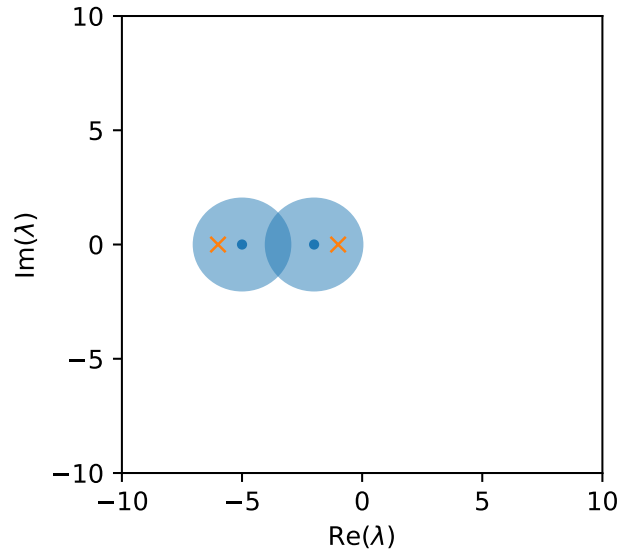


Figure 1.3: Gerschgorin discs.

Let's try this for our example problem from before to see if it worked. Our matrix was:

$$\underline{\underline{\mathbf{A}}} = \begin{bmatrix} -5 & 2 \\ 2 & -2 \end{bmatrix}$$

So we would predict that we have a disk centered at -5, with a radius of 2, and another at -2, with a radius of 2. See Figure 1.3 for the disks in the complex plane. Also drawn are our actual eigenvalues (as \times 's), which were -1 and -6, and we can see that they fall within the disks. It works!

It turns out that symmetric matrices (containing only real numbers) always give real eigenvalues; therefore, in this case we could have further isolated that our numbers lie just on the real number line.

The textbook reading gives more details on Gerschgorin disks, including subtleties about overlapping sets of disks. However, the general principle is obvious: the more diagonal¹⁷ if it a matrix is, the better estimate we have of the eigenvalues. The idea of making a matrix more diagonal as a way of making more and more refined estimates of the eigenvalues will be discussed in subsequent sections of these notes.

1.4.6 Diagonalization (and similarity)

From the concept of Gerschgorin disks, we can see that a more diagonal matrix has more obvious eigenvalues. (That is, we become more and more certain of them for more and more diagonal matrices.) The question is: if we start with a matrix $\underline{\underline{\mathbf{A}}}$, can we somehow transform it into a more diagonal form while keeping the same spectrum¹⁸ of eigenvalues?

Here, we'll review two concepts of linear algebra that allow us to develop methods in this regard: *similarity*, which tells us when two matrices have the same spectrum, and *diagonalization*, which tells us a specific relation between a matrix $\underline{\underline{\mathbf{A}}}$ and its full set of eigenvalues and eigenvectors.

Similarity

Reference: K8.4, K20.6, B3:118

Two matrices are called *similar* if they obey:

$$\underline{\underline{\hat{\mathbf{A}}}} = \underline{\underline{\mathbf{P}}}^{-1} \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{P}}} \quad (1.5)$$

¹⁷A diagonal matrix is one that only has non-zero values on the diagonal. We'll loosely say that a matrix tends to be "more diagonal" as the values on the diagonal grow (in absolute value) relative to those in the rest of the matrix.

¹⁸"Spectrum" of a matrix means its set of eigenvalues.

Importantly, it can be shown that $\underline{\underline{\mathbf{A}}}$ and $\hat{\underline{\underline{\mathbf{A}}}}$ have the same set of eigenvalues. We can prove this in a few lines:

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}} = \lambda \underline{\underline{\mathbf{x}}}$$

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{I}}} \underline{\underline{\mathbf{x}}} = \lambda \underline{\underline{\mathbf{x}}} \quad (\text{from } \underline{\underline{\mathbf{I}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{x}}} \text{ or } \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{I}}} = \underline{\underline{\mathbf{A}}})$$

$$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{P}}} \underline{\underline{\mathbf{P}}}^{-1} \underline{\underline{\mathbf{x}}} = \lambda \underline{\underline{\mathbf{x}}} \quad (\text{from } \underline{\underline{\mathbf{P}}} \underline{\underline{\mathbf{P}}}^{-1} = \underline{\underline{\mathbf{I}}})$$

$$\underline{\underline{\mathbf{P}}}^{-1} \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{P}}} \underline{\underline{\mathbf{P}}}^{-1} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{P}}}^{-1} \lambda \underline{\underline{\mathbf{x}}}$$

$$\hat{\underline{\underline{\mathbf{A}}}} \underline{\underline{\mathbf{P}}}^{-1} \underline{\underline{\mathbf{x}}} = \lambda \underline{\underline{\mathbf{P}}}^{-1} \underline{\underline{\mathbf{x}}}$$

This is new eigenvalue problem (of $\hat{\underline{\underline{\mathbf{A}}}}$) with eigenvectors $\underline{\underline{\mathbf{P}}}^{-1} \underline{\underline{\mathbf{x}}}$ and eigenvalues λ . That is, $\hat{\underline{\underline{\mathbf{A}}}}$ has the same eigenvalues as $\underline{\underline{\mathbf{A}}}$, and has closely related eigenvectors.

This means we can do unlimited operations of the type in equation (1.5) without changing the eigenvalues. If these operations lead towards a more diagonal matrix, we gain more certainty on our eigenvalues.

Diagonalization

Reference: K8.4

If we could perfectly transform $\underline{\underline{\mathbf{A}}}$ into a diagonal form $\underline{\underline{\mathbf{D}}}$, it would of course have the eigenvalues on the diagonal (and zeros elsewhere). $\underline{\underline{\mathbf{D}}}$ would be *similar* to $\underline{\underline{\mathbf{A}}}$, and thus would follow equation (1.5). Specifically, its similarity transform would be:

$$\underline{\underline{\mathbf{D}}} = \underline{\underline{\mathbf{X}}}^{-1} \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{X}}}$$

where $\underline{\underline{\mathbf{X}}}$ is a matrix with the eigenvectors in the columns. That is, there is a similarity relationship between the complete set of eigenvectors, matrix $\underline{\underline{\mathbf{A}}}$, and its eigenvalues. This is another reason why we often call the eigenvalue problem “diagonalizing a matrix”.

We can show this is the case briefly as follows. Let $\underline{\underline{\mathbf{x}}}_1, \underline{\underline{\mathbf{x}}}_2, \dots$ be the eigenvectors of $\underline{\underline{\mathbf{A}}}$, where $\underline{\underline{\mathbf{X}}} = [\underline{\underline{\mathbf{x}}}_1 \quad \underline{\underline{\mathbf{x}}}_2 \quad \dots \quad \underline{\underline{\mathbf{x}}}_n]$. Then,

$$\begin{aligned} \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{X}}} &= [\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}}_1 \quad \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}}_2 \quad \dots \quad \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}}_n] \\ &= [\lambda_1 \underline{\underline{\mathbf{x}}}_1 \quad \lambda_2 \underline{\underline{\mathbf{x}}}_2 \quad \dots \quad \lambda_n \underline{\underline{\mathbf{x}}}_n] \\ &= \underline{\underline{\mathbf{X}}} \underline{\underline{\mathbf{D}}} \end{aligned}$$

which is equivalent to $\underline{\underline{\mathbf{D}}} = \underline{\underline{\mathbf{X}}}^{-1} \underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{X}}}$.

1.4.7 Coaxing a matrix into diagonal form: QR and tridiagonalization

Reference: K20.9, B3:129

From the Gerschgorin disk discussion, it should be obvious that if we can coax our matrix into a more diagonal form that we will have a better and better estimate of the eigenvalues. Here, we’ll briefly describe one algorithm by which this occurs in practice. We won’t go deeply into the theory behind this, leaving such derivations to a linear algebra course. (You can see your textbook for references; section K20.9.)

Specifically, we’ll look at the QR factorization method with tridiagonalization. These methods are among those that are used by routines like numpy’s eigenvalue solver.

QR factorization

This particular application only applies¹⁹ to symmetric matrices $\underline{\underline{\mathbf{A}}}$, which are very common in eigenvalue problems. This, and its variants, are very common methods to coax a matrix into diagonal form.

Let's start with a matrix $\underline{\underline{\mathbf{A}}}$. We are going to develop an iterative procedure, so we'll call the matrix $\underline{\underline{\mathbf{A}}}_0$; subsequent refinements will bear subscripts 1, 2, etc. First, we factor our matrix $\underline{\underline{\mathbf{A}}}_0$ into an orthogonal ($\underline{\underline{\mathbf{Q}}}_0$) and an upper triangular ($\underline{\underline{\mathbf{R}}}_0$) matrix. This is called QR decomposition, which is possible for any non-singular matrix. (Many procedures exist for doing this decomposition; we will not cover those procedures here; only noting that such a decomposition is possible.)

$$\underline{\underline{\mathbf{A}}}_0 = \underline{\underline{\mathbf{Q}}}_0 \underline{\underline{\mathbf{R}}}_0 \quad (1.6)$$

Note the definitions:

- “Orthogonal” ($\underline{\underline{\mathbf{Q}}}$): the rows are orthogonal to one another (we talked about this last time) and each has a Euclidean norm of 1 (magnitude of 1); that is the rows are an orthonormal basis set.²⁰ This has some useful properties:

$$\underline{\underline{\mathbf{Q}}}^T \underline{\underline{\mathbf{Q}}} = \underline{\underline{\mathbf{I}}} = \underline{\underline{\mathbf{Q}}} \underline{\underline{\mathbf{Q}}}^T$$

which implies that the inverse and transpose are equivalent:

$$\underline{\underline{\mathbf{Q}}}^T = \underline{\underline{\mathbf{Q}}}^{-1}$$

- Upper triangular ($\underline{\underline{\mathbf{R}}}$): This has elements only on or above the diagonal.

Recall that *similar* matrices have the same spectrum of eigenvalues, and that a truly diagonal matrix has the eigenvalues on the diagonal. Then, we use $\underline{\underline{\mathbf{Q}}}$ to create a similar matrix to $\underline{\underline{\mathbf{A}}}$:

$$\underline{\underline{\mathbf{B}}}_1 = \underline{\underline{\mathbf{Q}}}_0^{-1} \underline{\underline{\mathbf{A}}}_0 \underline{\underline{\mathbf{Q}}}_0 \left(= \underline{\underline{\mathbf{R}}}_0 \underline{\underline{\mathbf{Q}}}_0 \right)$$

where we use $\underline{\underline{\mathbf{Q}}}_0^{-1} \underline{\underline{\mathbf{A}}}_0 = \underline{\underline{\mathbf{R}}}_0$ from equation (1.6). It can be proven (see the textbook), that in general $\underline{\underline{\mathbf{A}}}_{s+1}$ is more diagonal than $\underline{\underline{\mathbf{A}}}_s$, and that this converges towards a fully diagonal matrix. That is,

$$\lim_{s \rightarrow \infty} \underline{\underline{\mathbf{A}}}_s = \underline{\underline{\mathbf{D}}}$$

In other words, this is a way to coax a matrix into a diagonal form.

Summary. In net, what was done above is just two steps, repeated over and over:

1. Decompose $\underline{\underline{\mathbf{A}}}_s$ into $\underline{\underline{\mathbf{A}}}_s = \underline{\underline{\mathbf{Q}}}_s \underline{\underline{\mathbf{R}}}_s$.
2. Multiply to get $\underline{\underline{\mathbf{A}}}_{s+1} = \underline{\underline{\mathbf{R}}}_s \underline{\underline{\mathbf{Q}}}_s$.

Checking progress. Of course, we can use the Gerschgorin method from earlier to see if we are within a specified tolerance of our true eigenvalues.

¹⁹As noted earlier, symmetric matrices (of real numbers) always have exclusively real eigenvalues—that is, no imaginary components. It should be obvious that in these diagonalization procedures we would never create complex numbers on the diagonal; thus, this doesn't apply to general non-symmetric matrices where complex eigenvalues may be encountered.

²⁰Note that we could call this matrix “orthonormal”, which means both orthogonal and normalized; however, in practice these matrices are just called orthogonal.

Example. Let's try this on our example from last time; try running the below python code.

```
import numpy as np
np.set_printoptions(suppress=True, precision=2)

A = np.array([[ -5.,  2.], [ 2., -2.]])
print(A)

for _ in range(5):
    print(_)
    Q, R = np.linalg.qr(A)
    A = R @ Q
    print(A)
```

Recall that our eigenvalues were -6 and -1. we see it converge to this quickly.

Tridiagonalization

We need to use iterative procedures to diagonalize a matrix, since there is no general procedure to make a matrix diagonal in a finite number of steps. However, there are finite-step techniques to make a matrix *tridiagonal*, meaning having entries only down the diagonal and in the two adjacent diagonals. This has two advantages: first, it makes the matrix become “close to” diagonal, perhaps reducing the number of iterations needed. However, probably more significantly, this reduces an $n \times n$ matrix to a $3 \times n$ matrix and reduces the computational workload, as the remainder of the methods can use sparse optimized codes—that is, operating on $3n$ matrix entries instead of n^2 matrix entries. Chapter 20.9 describes the Householder method to accomplish this; this is typically used before embarking on a method like QR factorization,

1.4.8 Google PageRank

Reference: Bryan and Leise [3]

A very famous example of eigenvectors in practice—and perhaps one that drives an enormous computational demand—is google. Google was founded by two graduate students who thought of a better way to rank web pages on the internet, which became known as the PageRank algorithm.²¹ At the time, the internet was a vast trove of information; and many attempts to organize and search it existed, such as gopher (an alternative, hierarchical interface to sites), yahoo (a web-based directory), and many web search engines, like webcrawler, jeeves, lycos, altavista, etc. However, it was typically difficult to find what you wanted, and when using a search engine you had to comb through pages of results.²²

Here, we'll roughly follow an article called “The \$25,000,000,000 Eigenvector” [3], and look at only the simplest case discussed in that article. I encourage you to read the article to learn many more details, including two common cases where the simple case fails.

Building a search engine for the internet involves three main parts: (1) building an engine to explore websites by following links in a branching manner (we'll cover how this is done in the graph theory part of the course, §[link to come...](#)), (2) indexing the pages that are found, in order that search terms can be found, and (3) presenting a ranked order of relevant websites to the user.

The first two are relatively straightforward; Google's secret sauce was a better way to do the third. Here, we'll discuss how they did this in the context of an eigenvalue problem.

Example. We'll illustrate this with an example; this is the same introductory example as in Bryan *et al.* [3]. Consider the mini-internet, of four pages, shown in Figure 1.4; the four rectangles are webpages, and an arrow from one webpage to another indicates the first page has a link to the other.

²¹Apparently a pun on web page and Larry page.

²²Which was often a fun process, because the internet was fresh and new.

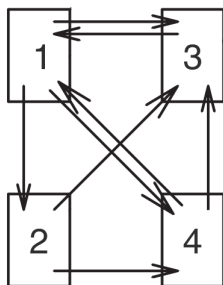


Figure 1.4: Web page connectivity; image from reference [3].

We should try to rank these in terms of which is the most important. Perhaps, whichever page is getting the most links from other pages should be the most important? A ranking system could just count incoming links from other pages, and the one with the most incoming links is the most important. If x_i is the importance of page i , this would be

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \\ 2 \end{bmatrix} \quad ?$$

where each link counts as a “vote” for a website’s importance.

This is a good starting point, but perhaps could be improved. The basic idea is that not all incoming links (votes) should be counted equally. For example, if the *New York Times* puts a link to my academic website, perhaps that should count more than if the *Brown Daily Herald* does so. Google’s founders were in an academic setting, and knew how journal rankings worked—where a factor that is used to assess the impact of journals is where the links to them come from. That is—if we’ve just established that certain sites are more important than others, shouldn’t we give more importance to their links? The basis assertion is that each website has a certain importance (x_i), and “spends” that importance on its links to other sites, in a formula such as

$$\begin{aligned} x_1 &= \frac{x_3}{1} + \frac{x_4}{2} \\ x_2 &= \frac{x_1}{3} \\ x_3 &= \frac{x_1}{3} + \frac{x_2}{2} + \frac{x_4}{2} \\ x_4 &= \frac{x_1}{3} + \frac{x_2}{2} \end{aligned}$$

That is, we divide each site’s influence by the number of outgoing links. In matrix form, this is

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

This is written compactly as:

$$\underline{\mathbf{x}} = \underline{\mathbf{A}}\underline{\mathbf{x}}$$

which by inspection is an eigenvalue problem ($\underline{\mathbf{A}}\underline{\mathbf{x}} = \lambda\underline{\mathbf{x}}$) with $\lambda = 1$. Note the structure of our $\underline{\mathbf{A}}$ matrix: the columns represent the outgoing links from each page, while the rows represent the page to which the links are going. If we keep this in mind, this matrix is easy to build without writing explicit algebraic equations

first. (We can do a quick check on the matrix by summing the columns; they should each be one; unless that page made no links, in which case it would sum to zero.)

How do we solve this quickly, with big data? Also, consider that $\underline{\underline{\mathbf{A}}}$ is changing by the second on the internet—can we solve this in such a manner that when we update $\underline{\underline{\mathbf{A}}}$ we do not need to start over from scratch?

1.4.9 Power method

Reference: K20.8, B3:126

Here, we'll discuss a simple and practical method for solving for a single eigenvector, as in the PageRank example of the previous section, §1.4.8.

Non-linear algebra example

Let's start with a crude algebraic trick that you can sometimes use to iterate to find the root of an equation. This is known as fixed-point iteration.²³ Say we have the equation

$$x^2 + 10x - 1 = 0$$

and we'd like to find a value of x . Let's pretend we don't know the quadratic formula, so we iterate. One way we can do so is to re-arrange for x :

$$x = \frac{1 - x^2}{10}$$

Now, we have an equality for x , although we haven't isolated x . But, we could try plugging in a value on the RHS²⁴, and see if we get the same thing back on the LHS. If we don't (we won't), we can plug in what we get from the LHS into the RHS.

We can write this algorithm simply; in python this would be:

```
import numpy as np

def get_new_x(old_x):
    return (1. - old_x**2) / 10.

x = 0.
for iteration in range(10):
    x = get_new_x(x)
    print(iteration, x)
```

We should see it going to 0.0990, which we can compare to the quadratic formula to see is (an approximation to) the correct answer. This will sometimes, but not always, work. But when it does, it's very simple and in this case was reasonably fast.

PageRank example

Does this work if $\underline{\mathbf{x}}$ is a vector, not just a number? For simple PageRank example of §1.4.8 our equation is:

$$\underline{\mathbf{x}} = \underline{\underline{\mathbf{A}}} \underline{\mathbf{x}}$$

In principle we guess an $\underline{\mathbf{x}}$, plug it in, and see if we converge to something. In python, with the example from our previous section, we would write this procedure as

²³Discussed in K19.2, and we'll cover it in §2.2.

²⁴RHS: right-hand side, LHS: left-hand side.

```

import numpy as np
np.set_printoptions(suppress=True, precision=2)

A = np.array([[0., 0., 1., 1./2.],
              [1./3., 0., 0., 0.],
              [1./3., 1./2., 0., 1./2.],
              [1./3., 1./2., 0., 0.]])

x = np.array([[.25, .25, .25, .25]]).T

for iteration in range(5):
    print(iteration)
    x = A @ x
    print(x)

```

(Note that in recent versions of python, the `@` symbol means matrix multiplication; that is, `np.matmul`.) If you run the code above, we should converge to a steady value of \underline{x} .

Extension to eigenvalue problems

Let's extend the above procedure to a general eigenvalue problem; that is,

$$\underline{\underline{A}} \underline{x} = \lambda \underline{x}$$

So maybe we could try something similar? We could solve for \underline{x} the same way:

$$\underline{x} = \frac{1}{\lambda} \underline{\underline{A}} \underline{x} \quad (1.7)$$

How could we use this in practice? We would have to guess both \underline{x} and λ , as opposed to the earlier problem, when we only guessed \underline{x} . It would be better to only have to guess \underline{x} , and deduce λ .

Recall that if we have an eigenvector of $\underline{\underline{A}}$, then $\underline{\underline{A}} \underline{x}$ will give us \underline{x} times some constant; that is, if we were at our true solution, then it would be equivalent to state that

$$\|\underline{\underline{A}} \underline{x}\| = \lambda \|\underline{x}\|$$

where the double bars indicate the magnitude or norm of the vector. ($\|\underline{a}\| = \sqrt{\underline{a}^T \underline{a}}$ is also known as the ℓ_2 norm; which we'll compare to other definitions of vector length in §1.5.1.) So

$$\lambda = \frac{\|\underline{\underline{A}} \underline{x}\|}{\|\underline{x}\|}$$

That is, λ is how much our vector has grown by the transformation. Recall that eigenvectors are only good to an arbitrary constant anyway, so let's just assert that in our iterative scheme that $\|\underline{x}\| = 1$, that is, we always normalize \underline{x} before we plug it in.²⁵ Then, we can approximate λ as $\|\underline{\underline{A}} \underline{x}\|$ and re-write equation (1.7) as

$$\underline{x} = \frac{\underline{\underline{A}} \underline{x}}{\|\underline{\underline{A}} \underline{x}\|} \quad (1.8)$$

This is the *power method*, and for an arbitrary initial guess will deliver the dominant eigenvector of an eigenvalue problem. Simply repeat equation (1.8) until the vector converges to your satisfaction.

²⁵ You can scale this in multiple ways; the book does this by just taking each \underline{x} and dividing by its maximum value.

When does this apply?

- This returns only a single eigenvector, and it can be shown that it reliably returns the dominant eigenvector; that is, the one with the greatest absolute value (eigenvalue). (Of course, if you start *exactly* at another eigenvector, you will return that one instead.)
- If your matrix is sparse; that is, if it has lots of zeros, this method can be a good choice because it only relies on matrix multiplication. There are good sparse methods for matrix multiplication.
- According to Wikipedia, this is used by Google in its PageRank algorithm and Twitter in its recommended followers algorithm.
- These iterative methods have a nice property that if $\underline{\mathbf{A}}$ changes subtly, then we can update it and keep on iterating, since $\underline{\mathbf{x}}$ will also change subtly in response to these dynamic changes in $\underline{\mathbf{A}}$. In the context of PageRank, as websites add or delete links this means they don't have to completely restart the algorithm every time a single website changes its links.

1.4.10 Eigenvalues in practice

In general, the method you choose to solve an eigenvalue problem will depend heavily on the nature of your problem, and as with most numerical methods, it's more efficient to use existing, optimized libraries rather than code your own solver, unless you have a very specialized or very small problem. As you choose your method, you should ask the following questions

- *Is my system small enough to consider the exact solution?*

An exact solution of an eigenvalue problem is, in general, impossible for matrices larger than 4×4 because there is no general solution strategy for polynomials higher than 4th order. However, if your matrix is 4×4 or less, you have the option to solve it exactly, if precision or computational speed is important in your application. (However, for typical applications, a simple call to a numerical eigenvalue solver like `np.linalg.eig` will be much easier.)

- *Is my Gerschgorin estimate good enough?*

You can quickly estimate your eigenvalues, complete with uncertainty bounds, using the method of §1.4.5. If this is good enough, you don't need to further solve your system.

- *Do I need all eigenvalues or just one? Or something in between?*

If you only need the dominant eigenvector, you might choose the power method, which just relies on repeated matrix multiplication to coax the initial guess vector into the eigenvector. There are other methods, such as the Davidson algorithm, that allow you to return a larger number of eigenvectors without returning the complete spectrum.

- *Is $\underline{\mathbf{A}}$ static, or will it change over time?*

In applications such as PageRank, the matrix $\underline{\mathbf{A}}$ changes subtly when websites change their links. The eigenvectors will also change only subtly as a result, and this makes the power method well-suited for this application, as one can just change the matrix $\underline{\mathbf{A}}$ on-the-fly, and $\underline{\mathbf{x}}$ will evolve.

In principle, nearly all iterative methods, such as QR factorization, will converge much faster if starting from a good initial guess, so re-using an old solution as an initial guess when $\underline{\mathbf{A}}$ changes is generally a good strategy.

- *Is my problem sparse?*

It is very common for matrices to be sparse—that is, made up with a relatively tiny number of non-zero entries. In this case, you should create the matrix as a “sparse” object, and use special sparse solvers. In python, these are found in the `scipy.sparse` module. More will be said about sparse problems in §1.5.2.

- *Is my matrix symmetric? Is my matrix Hermitian?*

Symmetric and Hermitian matrices are easier to solve than general matrices, and are quite common in eigenvalue problems. Practically, you should choose an eigenvalue solver that is optimized for these systems if you know your matrix to be Hermitian/symmetric; in python this means choosing `np.linalg.eigh` instead of `np.linalg.eig`.

Hermitian matrix definition. If your matrix contains complex numbers, we tend to speak in terms of Hermitian, rather than symmetric, matrices. A Hermitian matrix is equal to its own *conjugate transpose*:

- Transpose: $[a_{ij}] \rightarrow [a_{ji}]$
- Complex conjugate: $b + ci \rightarrow b - ci$
- \therefore Conjugate transpose: $[b_{ij} + c_{ij}i] \rightarrow [b_{ji} - c_{ji}i]$

From this perspective, a symmetric matrix is a special case of a Hermitian transpose containing only real numbers, since there are no imaginary numbers to flip.

1.5 Odds & ends

1.5.1 Norms of vectors and matrices

Reference: K20.4

We'll now briefly define the concept of a "norm", which is another way of saying the length of a vector or a matrix.

Vector norm

We are used to thinking about the length of a vector, probably with the "shortest distance" framework. It turns out there are different ways we can think about this length, listed below and shown in Figure 1.5.

- What is the shortest distance between the origin and the point; that is, $\sqrt{x_1^2 + x_2^2}$? (ℓ_2 norm or Euclidean length)
- What is the length to walk along all the segments, $|x_1| + |x_2|$? (ℓ_1 norm, or Manhattan distance)
- What is the largest segment? $\max(x_1, x_2)$, ℓ_∞ norm

Interestingly, we can write one simple formula that can mathematically express any of these; this is known as the ℓ_p norm:

$$\|\underline{\mathbf{x}}\|_p \equiv (|x_1|^p + |x_2|^p + \dots)^{1/p}$$

Matrix norm

There is also something called a matrix norm, which we'll quickly define. The definition is a little more confusing than a vector norm, but is

$$\|\underline{\mathbf{A}}\| \equiv \max_{\underline{\mathbf{x}}} \frac{\|\underline{\mathbf{A}}\underline{\mathbf{x}}\|}{\|\underline{\mathbf{x}}\|}$$

If we think of a matrix as something that stretches a vector, then the norm of $\underline{\mathbf{A}}$ is the maximum amount by which it can stretch any vector. In other words, we can also say that

$$\underbrace{\|\underline{\mathbf{A}}\underline{\mathbf{x}}\|}_{\text{the amount our vector is stretched by } \underline{\mathbf{A}}} \leq \|\underline{\mathbf{A}}\| \|\underline{\mathbf{x}}\|$$

Here, the degree of the norm of the matrix norm is such that it matches that of the vector norm. Relatively simple formulae turn out to exist for common values of p , as described in the textbook.

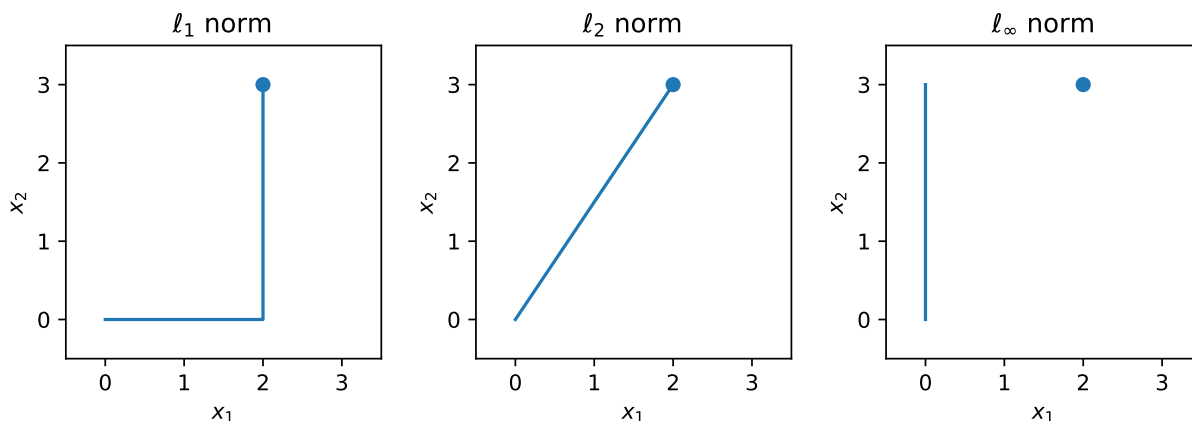


Figure 1.5: Common norms, expressed in two-dimensional space.

1.5.2 Sparse matrices

Reference: Beers 1:46

In practice, a great many matrices encountered in numerical approaches are “sparse”; that is, they consist mostly of zeros. Let’s look at the two example applications we discussed earlier. First, we had the problem of vibrating masses from §1.4.2. In this case, our matrix (\mathbf{k}) was made up of spring constants, the elements of which k_{ij} told us about the forces between any two masses i and j . In this problem, we might consider that springs are only between masses that are close to one another; for a large system, most masses will not be neighbors and this matrix will be mostly zeros. As another example, we discussed Google’s PageRank algorithm (§1.4.8), where the matrix \mathbf{A} had elements A_{ij} which represented a hyperlink between webpage i and webpage j . Since most websites on the internet do not link to one another, this again is a matrix that mostly contains zeros, with non-zero elements only where a link exists. (In this example, the matrix will be *huge*, but mostly full of nothing.)

As another example, we talked about changing a matrix into a similar tri-diagonal matrix (in §1.4.7). Once it is tridiagonalized, if we have an $n \times n$ matrix with $n = 1,000,000$, this means we have 3,000,000 actual numbers. Yet if we use the entire matrix, we have 1,000,000,000,000 numbers in memory, but 99.9997% of the numbers are zero. This is an incredible waste of memory and computation, if you have the memory / computational power for this problem in the first place.

What to do about this? First, we can note that it will be a waste of memory to store all those zeros in memory, so we can come up with more efficient schemes to store the values. For example, we can envision just storing a list of locations and values; e.g., the 3×3 identity matrix might be stored as

$$\begin{aligned} A[0,0] &= 1 \\ A[1,1] &= 1 \\ A[2,2] &= 1 \end{aligned}$$

where we’re using python’s numbering scheme (starting at 0).

Second, we can note that most of the computational work can be simplified significantly if we have sparse systems. If we are performing matrix multiplications, remember that each entry is created by multiplying pairs of numbers from the two parent matrices. Obviously, in a sparse matrix there’s usually a pretty good chance that one of the two entries being multiplied is zero, and we should only actually multiplying the numbers (and subsequently adding them) if there is something in both places. Thus, if your system is sparse, we might employ a very different way of multiplying two matrices than we would do if we expected the matrices to be full of (non-zero) numbers! We should be able to multiply two sparse matrices together relatively cheaply.

To rectify this problem, most scientific software packages have “sparse” methods, that let you create sparse matrices (in memory-efficient form) and manipulate them (in compute-efficient manners). Most of

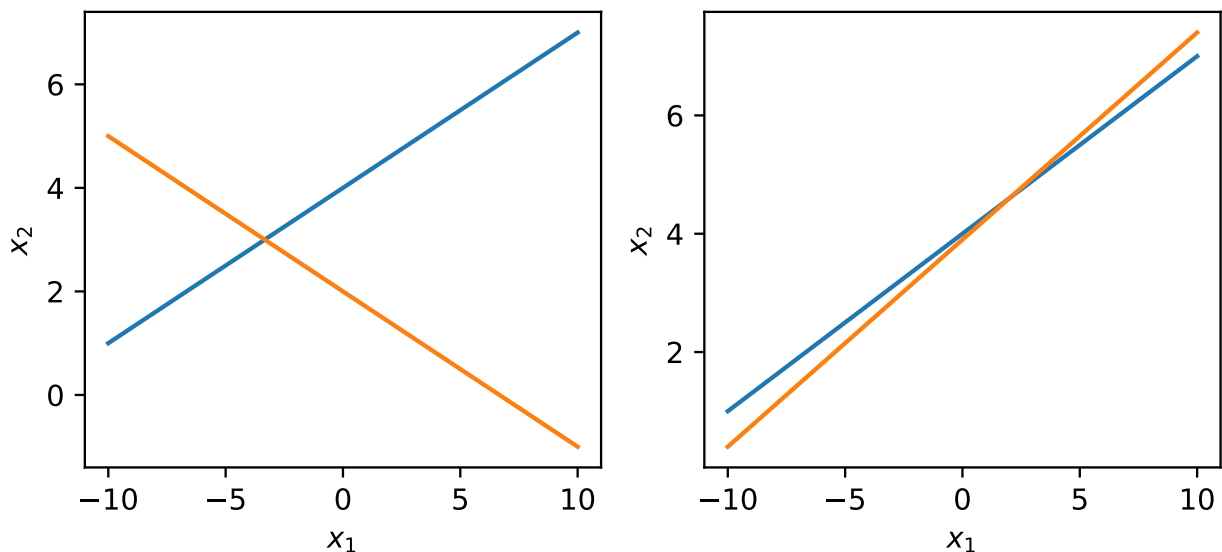


Figure 1.6: In the right plot, our solution is ill-conditioned.

the methods that we have discussed to date have sparse analogs, and if you have a sparse matrix you should use them.

In python/scipy, you can find them with `scipy.sparse`. E.g.,

```
from scipy.sparse.linalg import eigsh # Sparse eigenvalue-problem solver
from scipy.sparse.linalg import eig   # Sparse Hermitian eigenvalue-problem solver
```

1.5.3 Iterative methods in linear systems, ill conditioning

Reference: K20.3, K20.4

Gauss-Seidel Iteration

As we discussed earlier, linear systems ($\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$) can be solved exactly, for example by inverting $\underline{\underline{\mathbf{A}}}$. However, it can sometimes be faster to also use an iterative approach. We didn't cover it in this class, but a common method is called *Gauss-Seidel Iteration*; this is in principle similar to the fixed-point iteration method and power method in its simplicity. Essentially, the system of linear equations is re-arranged so that $x_1 \cdots x_n$ appears explicitly on the left-hand side of the equation, and the values are updated at each iteration. See Example 1 of K20.3 or the corresponding derivation in that section for details on this method, which we won't cover in class due to its similarity to other iterative methods.

Ill-conditioning

We'll motivate the concept of *ill-conditioned* matrices by examining how we can determine when a linear system ($\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{x}}} = \underline{\underline{\mathbf{b}}}$) is solved, but will note that the concept of an ill-conditioned matrix is much more meaningful than this, and indicates a system where a numerical system may be very sensitive to small changes in parameters.

A linear system can be considered to be solved when the residuals, $\underline{\underline{\mathbf{r}}} \equiv \underline{\underline{\mathbf{b}}} - \underline{\underline{\mathbf{A}}} \underline{\underline{\tilde{\mathbf{x}}}}$ are very small, where $\underline{\underline{\tilde{\mathbf{x}}}}$ is the current approximation of the solution. An ill-conditioned system means that even if the residuals $\underline{\underline{\mathbf{r}}}$

are very small,²⁶ then we may still be long ways from our “true” answer.

For example, consider the system of two equations shown in Figure 1.6. We know that the true answer is at the point of intersection of the two lines; or alternatively the difference between the two lines. The left plot is a good system; there’s a clear answer. The right plot shows an ill-conditioned system: there is a large region where the two lines are almost intersecting, so if we had guessed values of $\underline{\mathbf{x}}$ we might think we have solved our system, but still be a long ways off in terms of the true $\underline{\mathbf{x}}$.

Condition number. We can generalize this for solving $\underline{\mathbf{A}} \underline{\mathbf{x}} = \underline{\mathbf{b}}$ as follows. The prediction residuals can be defined as

$$\underline{\mathbf{r}} \equiv \underline{\mathbf{b}} - \underline{\mathbf{A}} \tilde{\underline{\mathbf{x}}} = \underline{\mathbf{A}} \underline{\mathbf{x}} - \underline{\mathbf{A}} \tilde{\underline{\mathbf{x}}} = \underline{\mathbf{A}}(\underline{\mathbf{x}} - \tilde{\underline{\mathbf{x}}})$$

where $\tilde{\underline{\mathbf{x}}}$ is my approximate solution and $\underline{\mathbf{x}}$ is the true solution. Or,

$$\underline{\mathbf{x}} - \tilde{\underline{\mathbf{x}}} = \underline{\mathbf{A}}^{-1} \underline{\mathbf{r}}$$

From the definition of a matrix norm; that is, $\|\underline{\mathbf{A}} \underline{\mathbf{x}}\| \leq \|\underline{\mathbf{A}}\| \|\underline{\mathbf{x}}\|$ we can write

$$\|\underline{\mathbf{x}} - \tilde{\underline{\mathbf{x}}}\| \leq \|\underline{\mathbf{A}}^{-1}\| \|\underline{\mathbf{r}}\|$$

Dividing with the norm of $\underline{\mathbf{x}}$ (so that we get relative inaccuracies), we find

$$\frac{\|\underline{\mathbf{x}} - \tilde{\underline{\mathbf{x}}}\|}{\|\underline{\mathbf{x}}\|} \leq \frac{1}{\underbrace{\|\underline{\mathbf{x}}\|}_{\leq \frac{\|\underline{\mathbf{A}}\|}{\|\underline{\mathbf{b}}\|}}} \|\underline{\mathbf{A}}^{-1}\| \|\underline{\mathbf{r}}\| \leq \underbrace{\|\underline{\mathbf{A}}\| \|\underline{\mathbf{A}}^{-1}\|}_{\equiv \kappa(\underline{\mathbf{A}})} \frac{\|\underline{\mathbf{r}}\|}{\|\underline{\mathbf{b}}\|}$$

That is,

$$\frac{\|\underline{\mathbf{x}} - \tilde{\underline{\mathbf{x}}}\|}{\|\underline{\mathbf{x}}\|} \leq \kappa(\underline{\mathbf{A}}) \frac{\|\underline{\mathbf{r}}\|}{\|\underline{\mathbf{b}}\|}$$

Note that in this form, our error in $\underline{\mathbf{x}}$ is normalized (by $\|\underline{\mathbf{x}}\|$) and our residuals are normalized (by $\|\underline{\mathbf{b}}\| = \|\underline{\mathbf{A}} \underline{\mathbf{x}}\|$).

Therefore, if κ is large, this means that even though our residuals are low our actual error itself in $\underline{\mathbf{x}}$ could still be very large. κ is called the “condition number” of the matrix $\underline{\mathbf{A}}$.

$$\boxed{\kappa(\underline{\mathbf{A}}) \equiv \|\underline{\mathbf{A}}\| \|\underline{\mathbf{A}}^{-1}\|}$$

which gives the criteria:

- Large: numerical difficulties in finding true solution; ill-conditioned problem.
- Small: not so bad: well-conditioned problem.

This is a general property of a matrix $\underline{\mathbf{A}}$ —for example, it can be shown (if the ℓ_2 norm is used in the definition) that the condition number also indicates the spread in the eigenvalues, as it is equal to the ratio of the largest to the smallest (absolute value) eigenvalues.

1.5.4 Basis vectors, and transformations between bases

Reference: K8.4, B3:117

A vector often defines the state of a system, and there are often multiple equivalent ways to express the system state—for example, you probably remember from a physics course that you can equivalently choose any frame of reference you like, even moving ones, to describe your problem. Here, we’ll discuss the concept of sets of basis vectors, and how we can transform between representations. We’ll start with a couple of examples before going through the mathematics.

²⁶When we say very small, we should properly say relatively small with respect to something. In this case, we’ll find that something to be $\underline{\mathbf{b}}$, and the relative quantity that expresses the residual size to be $\|\underline{\mathbf{r}}\| / \|\underline{\mathbf{b}}\|$

- **Water molecule.**

Let's say I'd like to describe the motion of a single water molecule. The obvious way we would represent things is a 9-element vector that contains the cartesian positions of each of the three atoms, and describe how this vector evolves over time (perhaps through Newton's laws of motion).

However, as we acted out in class one day, the water molecule has certain natural vibrations. Specifically, there are three natural vibrations²⁷ in which the water molecule prefers to move (which is found by an eigenvalue problem). So perhaps it makes sense to represent the position of the water molecule in terms of displacements along these three vibrational modes?

However, we'd have lost some information if we did so—our original system had 9 numbers required to describe the water's position, and we only have 3 vibrational motions to describe the water's movement with. To complete this representation, we need to supply an additional 6 numbers (or, degrees of freedom); it turns out that supplying the position and rotation²⁸ of the molecule as a whole does the trick.

Thus, we might want to equivalently represent the water molecule in the Cartesian basis or the more natural basis, and we need a way to transform between these two representations:

$$\begin{bmatrix} x_O \\ y_O \\ z_O \\ x_{H1} \\ y_{H1} \\ z_{H1} \\ x_{H2} \\ y_{H2} \\ z_{H2} \end{bmatrix} \rightarrow \begin{bmatrix} x_{COM} \\ y_{COM} \\ z_{COM} \\ \theta_x \\ \theta_y \\ \theta_z \\ \text{vib}_1 \\ \text{vib}_2 \\ \text{vib}_3 \end{bmatrix}$$

The latter representation can be useful for a number of numerical and physical reasons; let's consider the physical ones.

- If we are describing how gravity acts on this, we can operate only on the z_{COM} component.
- For how diffusion operates, perhaps we only examine the $x_{COM}, y_{COM}, z_{COM}$ components.
- If we are interested in how microwaves interact with this we could focus on the θ terms.
- If we are studying spectroscopy by how IR interacts we can focus on the vibrational terms. We might also use these terms as a starting point to analyze chemical reactions.

- **Principle component analysis.**

This is a rather famous example, which we'll discuss in the last part of the course [\[link to come...\]](#). We won't draw this out fully in these typed notes, but the general concept is reproduced.

If we'd like to draw a picture of a face, we can do so efficiently using a lot of building block pieces (eigenfaces). That is, we multiply each eigenface by a coefficient, and sum them to get back an approximation of our original face. That is,

$$\square = c_1 \square + c_2 \square + c_3 \square + \dots$$

Where the \square 's are various pictures containing facial components. Then we can represent a picture just by a vector of these coefficients, that is, $[c_1 \ c_2 \ c_3 \ \dots]^T$.

²⁷The symmetric stretch, the bend, and the antisymmetric stretch.

²⁸Actually, we can also use an eigenvalue analysis to find the three principle moments of inertia to describe the natural rotations of the system.

We'll now return to the mathematical background, which is rather straightforward. When we express something in Cartesian coordinates, like a number $(1.5, 1.6)$, we can also think of this as coefficients of two unit vectors; that is,

$$\begin{bmatrix} 1.5 \\ 1.6 \end{bmatrix} = 1.5 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 1.6 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Therefore, in general, if we have a basis set given by the vectors $\{\underline{\mathbf{u}}, \underline{\mathbf{w}}\}$ we can express any 2-d vector $\underline{\mathbf{v}}$ as

$$\underline{\mathbf{v}} = c_1 \underline{\mathbf{u}} + c_2 \underline{\mathbf{w}}$$

where $\underline{\mathbf{u}}$ and $\underline{\mathbf{w}}$ are not necessarily our Cartesian unit vectors $[1 \ 0]^T$ and $[0 \ 1]^T$.

Note that we can examine the independence of a basis set by checking dot products. Let's examine this for the Cartesian unit vectors:

$$\underline{\mathbf{v}}^T \underline{\mathbf{v}} = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1$$

$$\underline{\mathbf{u}}^T \underline{\mathbf{v}} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0$$

$$\underline{\mathbf{u}}^T \underline{\mathbf{u}} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1$$

These unit vectors form an orthonormal basis set, and the criteria are:

- Orthogonal. The dot product of a each basis vector with all other unit vectors is 0. This means there is no overlap in the basis vectors.
- Normalized. The dot product of each basis vector with itself is 1; that is, the ℓ_2 -norm of each vector is 1. (§1.5.1)

If both of the above criteria are met, we have an *orthonormal set* of basis vectors.

Example. In §1.4.3, we solved an eigenvalue problem and found the two eigenvectors of $\begin{bmatrix} 1 & 2 \\ -1 & 2 \end{bmatrix}^T$ and $\begin{bmatrix} 2 \\ -1 \end{bmatrix}^T$. As we have discussed, the eigenvectors often make a natural set of basis vectors, so let's try using these as our basis set. First, let's examine the set of vectors itself via the dot product:

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \end{bmatrix} = 2 - 2 = 0$$

Therefore, these two vectors are orthogonal. However, they are not normalized: $\left\| \begin{bmatrix} 1 & 2 \end{bmatrix}^T \right\|_2 = \sqrt{5}$. If we wanted to, we could divide each by $\sqrt{5}$ to normalize them.²⁹ These two eigenvectors are plotted in Figure 1.2; next, we'll express our Cartesian point $[1.5, 1.6]$ in this new basis.

$$\begin{bmatrix} 1.5 \\ 1.6 \end{bmatrix} = c_1 \begin{bmatrix} 1 \\ 2^* \end{bmatrix} + c_2 \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 1c_1 + 2c_2 \\ 2^*c_1 - 1c_2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2^* & -1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

In the above equation, I marked one of the 2's with a star, so that we can keep track of where the numbers go as we re-arrange this. We see that we get a linear system ($\underline{\mathbf{A}} \underline{\mathbf{x}} = \underline{\mathbf{b}}$), where the columns of our matrix are our two new basis vectors. That is, we just solve

$$\begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 1.6 \end{bmatrix}$$

²⁹Recall from our eigenvalue problem: the eigenvectors of a symmetric matrix are orthogonal. However, the magnitude of the eigenvectors from an eigenvalue problem are arbitrary, so we are free to normalize them if we like. Here, we won't bother.

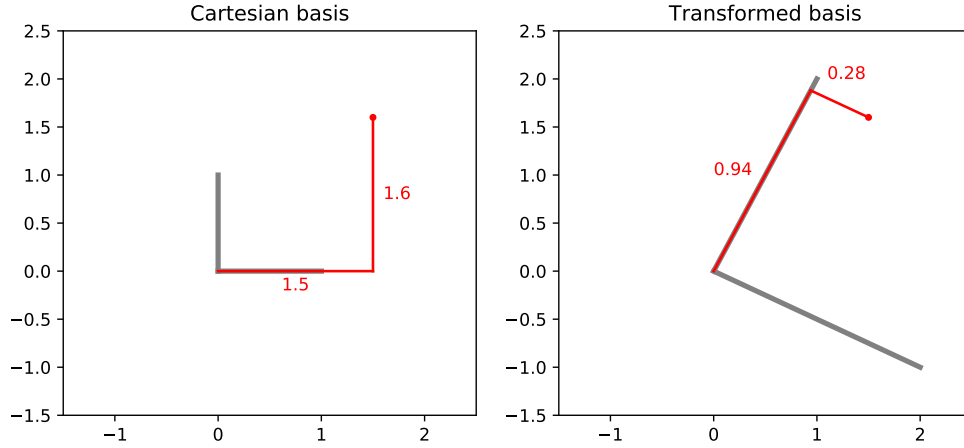


Figure 1.7: A point expressed in the Cartesian basis (1.5, 1.6) and a transformed basis (0.94, 0.28).

We can solve this with our standard techniques³⁰ from §1.3 and find that our vector is now expressed as [0.94, 0.28]; that is these are the coefficients of our new set of basis vectors. The new and old bases are shown in Figure 1.7.

General form

Let's do this generically for a transformation between any two sets of basis vectors, but we'll still constrain this example to 2-dimensional space. (For vectors of length N , you can do the same procedure but with more basis vectors.)

Say we have two sets of basis vectors:

$$B = \{\underline{\mathbf{u}}, \underline{\mathbf{w}}\}$$

$$B' = \{\underline{\mathbf{u}'}, \underline{\mathbf{w}'}\}$$

In our prior example, B was the set of Cartesian unit vectors and B' was the set of eigenvectors, but this is treatment is general. Say that we have a vector $\underline{\mathbf{v}}$ in base B

$$\underline{\mathbf{v}} = c_1 \underline{\mathbf{u}} + c_2 \underline{\mathbf{w}} \quad (\text{known } c\text{'s})$$

and we want to express it in B' . Perhaps a way to do this is to express $\underline{\mathbf{u}}$ and $\underline{\mathbf{w}}$ in terms of $\underline{\mathbf{u}'}$ and $\underline{\mathbf{w}'}$ and plug them in. Let's try this:

$$\begin{aligned} \underline{\mathbf{u}} &= \alpha \underline{\mathbf{u}'} + \beta \underline{\mathbf{w}'} \\ \underline{\mathbf{w}} &= \gamma \underline{\mathbf{u}'} + \delta \underline{\mathbf{w}'} \end{aligned} \tag{1.9}$$

(Our Greek letters, $\alpha, \beta, \gamma, \delta$ are unknown coefficients; we're using these to avoid having too many subscripts and primes, etc.) We'll assume we can find $\alpha, \beta, \gamma, \delta$ as before—but we'll come back to this.

Now we plug in for $\underline{\mathbf{u}}$ and $\underline{\mathbf{w}}$ and we can express $\underline{\mathbf{v}}$ with respect to our new coordinates.

$$\underline{\mathbf{v}} = c_1 (\alpha \underline{\mathbf{u}'} + \beta \underline{\mathbf{w}'}) + c_2 (\gamma \underline{\mathbf{u}'} + \delta \underline{\mathbf{w}'})$$

³⁰If you are dealing in big systems, it's worth thinking about the most efficient way to do this. If we are doing this operation only once, we should use a standard routine like `np.linalg.solve`, which uses LU decomposition, as this is faster than inverting a matrix, perhaps by a factor of two or so. However, if we have many vectors that we need to transform, we should just invert $\underline{\mathbf{A}}$ once, then use this to transform each of our vectors using a simple matrix multiplication.

$$\underline{\mathbf{v}} = \underbrace{(\alpha c_1 + \gamma c_2)}_{c'_1} \underline{\mathbf{u}}' + \underbrace{(\beta c_1 + \delta c_2)}_{c'_2} \underline{\mathbf{w}}'$$

$$\underline{\mathbf{v}} = c'_1 \underline{\mathbf{u}}' + c'_2 \underline{\mathbf{w}}'$$

And so we can see that

$$\begin{bmatrix} c'_1 \\ c'_2 \end{bmatrix} = \begin{bmatrix} \alpha & \gamma \\ \beta & \delta \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

$$\underline{\mathbf{v}}_{B'} = \underline{\underline{\mathbf{T}}} \underline{\mathbf{v}}_B$$

where $\underline{\underline{\mathbf{T}}}$ is called the transformation matrix to take us from the B to the B' basis. Recall that a matrix acts to transform a vector, and that's exactly what we've done here.

We still need to find $\underline{\underline{\mathbf{T}}}$. The components of this are α , etc., and they were defined up in equation (1.9).

$$\underline{\mathbf{u}} = \alpha \underline{\mathbf{u}}' + \beta \underline{\mathbf{w}}'$$

To put in numbers to make the next step easier to see, we can write as

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \alpha \begin{bmatrix} 1 \\ 2^* \end{bmatrix} + \beta \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2^* & -1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

$$\underline{\mathbf{u}} = [\underline{\mathbf{u}}' \quad \underline{\mathbf{w}}'] \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Analogously,

$$\underline{\mathbf{v}} = [\underline{\mathbf{u}}' \quad \underline{\mathbf{w}}'] \begin{bmatrix} \gamma \\ \delta \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2^* & -1 \end{bmatrix} \begin{bmatrix} \gamma \\ \delta \end{bmatrix}$$

This is a linear system, and we can solve the above to get the components of $\underline{\underline{\mathbf{T}}}$.

Topic 2

Nonlinear algebraic equations

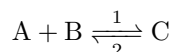
Reference: K19.2, B2

Although linear algebra serves as the backbone of many computational methods, most problems we encounter are not (natively) linear algebra problems. If we are lucky enough that our problems can be expressed algebraically, they are typically non-linear algebraic equations, and we need to develop methodologies to search for the answer(s) to these problems. In this section, we'll be discussing solving systems of non-linear equations, which will lead us to numerical optimization, which is a closely related problem. We'll start off with an example.

2.1 Example: a reactor

To physically motivate this, we'll start with an example of a chemical reactor; this is a "continuous stirred-tank reactor" (CSTR), as drawn in Figure 2.1. This is an idealized chemical reactor in which everything is instantaneously perfectly mixed.

Let's assume we have the reaction:



This is a *reversible* reaction, meaning that it can go forward (1) or backward (2). We can write rate equations for the rates at which these two reactions occur as:

$$r_1 = k_1 C_A C_B$$

$$r_2 = k_2 C_C$$

where the units of r_i are typically $\text{mol s}^{-1} \text{ m}^{-3}$ and the units of C_i are mol m^{-3} .

We can derive differential equations for each species by simply writing conservation equations; that is,

$$\text{Accumulation} = \text{In} - \text{Out} + \text{Generation} - \text{Consumption}$$

$$\frac{dC_A V}{dt} = vC_{A,0} - vC_A - k_1 C_A C_B V + k_2 C_C V$$

$$\frac{dC_B V}{dt} = vC_{B,0} - vC_B - k_1 C_A C_B V + k_2 C_C V$$

$$\frac{dC_C V}{dt} = vC_{C,0} - vC_C + k_1 C_A C_B V - k_2 C_C V$$

If we let the reactor run for a long time, with a constant flow rate in, eventually the concentrations at the outlet will stabilize. This is known as "steady state". At steady state, these are all equal to zero. We

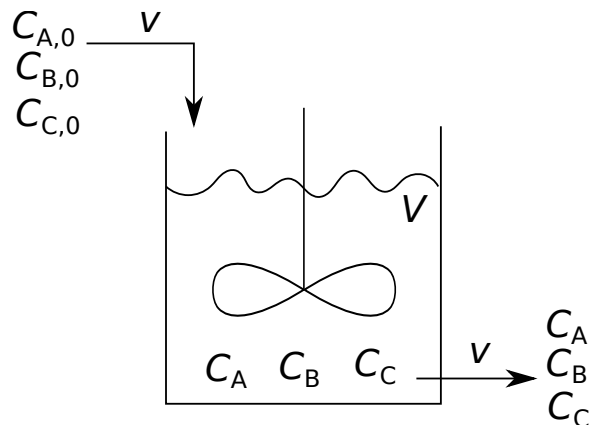


Figure 2.1: A continuous stirred-tank reactor. V is the volume of the reactor, while v is the volumetric flow into the reactor, which contains A, B, and C at initial concentrations of $C_{A,0}$, $C_{B,0}$, and $C_{C,0}$, respectively. The concentrations inside the reactor are C_A , C_B , and C_C ; these are also the exit concentrations since it is perfectly stirred.

conventionally define the ratio of the reactor volume to the volumetric flow rate as $\tau \equiv V/v$ (the residence time), and we get

$$0 = C_{A,0} - C_A - k_1 C_A C_B \tau + k_2 C_C \tau$$

$$0 = C_{B,0} - C_B - k_1 C_A C_B \tau + k_2 C_C \tau$$

$$0 = C_{C,0} - C_C + k_1 C_A C_B \tau - k_2 C_C \tau$$

In the above equations, we have three equations and three unknowns: C_A , C_B , and C_C . However, the presence of the $C_A C_B$ terms makes these equations not able to fit into a $\underline{\mathbf{A}} \underline{\mathbf{x}} = \underline{\mathbf{b}}$ form anymore, so we need to find a nonlinear way to solve these equations. In this relatively simple example, we might be able to work with the equations and find an analytical solution. However, a more typical example might have dozens of reactions in which this is not possible, and we need to find general ways to find values of the variables that solve these equations.

The above is a very simple example from reactions, but similar non-linear systems pop up all over engineering and physics.

2.2 Fixed-point iteration

Reference: K19.2

Fixed-point iteration is the simplest iteration technique we can imagine, and provides a very simple demonstration of the iterative technique; however, it's rarely useful in practice. Instead, variations of Newton's method are typically used, as discussed in upcoming sections. We first discussed fixed-point iteration when we introduced the power method for eigenvalue problems (§1.4.9), and will discuss it briefly again here, by means of an example.

Example. Let's try it for a simple one-equation example: suppose we have

$$e^x - 3x - 5 = 0$$

In fixed-point iteration, we isolate our unknown (x):

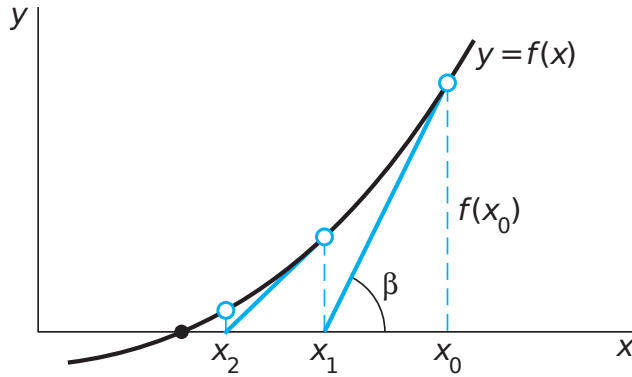


Figure 2.2: Newton's method, from Kreyszig.

$$x = \frac{e^x - 5}{3}$$

Then we guess a value of x and plug it into the right-hand side (RHS) of the above equation. If by some stroke of luck we guessed right, we'd get the same value out of the left-hand side (LHS) of the equation. In practice, the value we get out of the RHS will not be the same. If we are lucky though, the value will be *closer* to the value that solves the equation. This method works sometimes, but is not very robust or fast.

```
def get_x(x):
    return (np.exp(x) - 5.) / 3.

x = 3.
for _ in range(10):
    x = get_x(x); print(x)
```

In the above example, we will see our solution quickly diverge to $+\infty$, which is not an interesting¹ solution to this problem. However, if we had instead guessed 1.5, we would see our solution quickly converge to a root of about -1.6.

2.3 Newton–Raphson Iteration

Reference: K19.2, B2:63

Much better than fixed-point iteration is Newton–Raphson iteration (NRI), often referred to simply as Newton's method. Let's start with one equation, then generalize it to systems. Say we would like to know a value of x that solves:

$$f(x) = 0$$

We can approach this in two ways. The first is graphical; see Figure 2.2. The black line is our function—keep in mind that we don't know this shape, we only can evaluate it at specific points.² Let's say we guess the point x_0 . Our hope is that $f(x)$ will be equal to zero, but we find that it is instead equal to $f(x_0)$. We

¹Actually, it is interesting: at first glance, $x = \infty$ is a solution to $3x = e^x - 5$. But ∞ is a hard thing to define, and we typically think of this answer as $x \rightarrow \infty$. In this regard, we e^x is a much larger number than x . But in most problems in engineering and physics, an answer of ∞ is not very useful, so this debate is best left to mathematicians.

²For a simple function in one-dimensional space, such as this one, we can just plot it and see where it intercepts zero, and this is a solution. If you have a simple function, that is probably the best strategy. However, in engineering and physics we often have more difficult cases. For example, evaluating the function could be extremely slow (for example, in quantum mechanical calculations or something very applied, like a function where we change the pitch on a windmill blade and use a multiphysics model to predict the power output. Also, if you have more than one unknown, this approach breaks down, and we'll address these systems with multiple unknowns in the next section.

now need to make a new guess, x_1 . Clearly, if we know the slope, it tells us the right direction to go, and a logical approach is to assume the slope points to the zero intercept. If we assume this, we get the new point x_1 , and evaluate $f(x_1)$. Here, we see that we have not found the zero-intercept, f has decreased. We can again use the slope, $f'(x_1)$, to take the next step.

To be clear, Newton's method tells us two things. If we guess x_0 and evaluate our function to find $f(x_0)$ (which we'll assume is not zero—if it is, the problem is solved!), we normally wouldn't have an idea of what to do next. f' tells us which way to move. Further, this tells us how big a step to take—under the assumption that the function is linear. This may seem like a bad assumption, but keep in mind that at least we now know the order of magnitude. That is, we otherwise might not know whether we should use a step size of 10^{12} or $.003$; by this approach, we at least have the order-of-magnitude reasonable

Formal derivation

Although we can quickly derive this approach with a graphical method (at least for the one-dimensional problem), we can also work this out systematically, using a Taylor series expansion. Say we have point x_0 , where we know $f(x_0)$, and we want to find $f(x_1)$ such that $f = 0$.

$$\underbrace{f(x_1)}_{\text{want this to be 0}} = \underbrace{f(x_0)}_{\text{known}} + \underbrace{\frac{df}{dx}\bigg|_{x_0}}_{\text{known}} \underbrace{(x_1 - x_0)}_{\text{desired}} + \underbrace{\frac{1}{2}f''(x_0)(x_1 - x_0)^2 + \dots}_{\text{truncate}}$$

We'll truncate this at the first derivative term for now. We want to find the value of x_1 where $f(x_1)$ is zero:

$$0 = f + f' \cdot (x_1 - x_0)$$

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

Example. Let's try again for our example.

$$f(x) = e^x - 3x - 5 = 0$$

$$\frac{df}{dx} = e^x - 3$$

```
def get_f(x):
    return (np.exp(x) - 3. * x - 5.)

def get_fprime(x):
    return (np.exp(x) - 3.)

def get_new_x(x):
    return x - get_f(x) / get_fprime(x)

x = 3.
x = get_new_x(x); print(x)

x = -2.
x = get_new_x(x); print(x)
```

We found another root with the $x = -2$ guess: $x = 2.53$ is a root; even though we guessed near that with fixed-point iteration we didn't find it. We can still get the negative root as well.

2.3.1 Extension to systems of nonlinear equations

Reference: B2:71

It's pretty straightforward to extend this to many dimensions, if we use the Taylor series approach. Consider we have a system of n equations and n unknowns. The many-dimensional TSE is written for a single equation f_i in this group as

$$f_i(\underline{\mathbf{x}}^{(1)}) \approx f_i(\underline{\mathbf{x}}^{(0)}) + \sum_{j=1}^n \left. \frac{\partial f_i}{\partial x_j} \right|_{\underline{\mathbf{x}}^{(0)}} (x_j^{(1)} - x_j^{(0)})$$

where $\underline{\mathbf{x}}^{(0)}$ is known, and we again want the left-hand side to be zero. We have one such equations for each of the n functions f_i . Let $J_{ij} \equiv \frac{\partial f_i}{\partial x_j}$ and $\underline{\underline{\mathbf{J}}} \equiv [J_{ij}]$. This is the *Jacobian* matrix. Named after *Carl Jacobi*, a German mathematician who was pen-pals with Charles Hermite (of Hermitian matrix fame). Let

$$\underline{\mathbf{f}}(\underline{\mathbf{x}}) \equiv [f_1(\underline{\mathbf{x}}) \quad f_2(\underline{\mathbf{x}}) \quad \dots]^T$$

$$\underline{\underline{\Delta}}^{(1)} \equiv \underline{\mathbf{x}}^{(1)} - \underline{\mathbf{x}}^{(0)}$$

Then we can re-write our system of TSE equations as:

$$\underline{\mathbf{0}} = \underline{\mathbf{f}}(\underline{\mathbf{x}}^{(0)}) + \underline{\underline{\mathbf{J}}}^{(0)} \underline{\underline{\Delta}}^{(1)}$$

Or equivalently, our next step is found by solving the now linear ($\underline{\underline{\mathbf{A}}} \underline{\mathbf{x}} = \underline{\mathbf{b}}$) set of algebraic equations shown below

$$\underline{\underline{\mathbf{J}}}^{(0)} \underline{\underline{\Delta}}^{(1)} = -\underline{\mathbf{f}}(\underline{\mathbf{x}}^{(0)})$$

Or at the general step k :

$$\boxed{\underline{\underline{\mathbf{J}}}^{(k)} \underline{\underline{\Delta}}^{(k+1)} = -\underline{\mathbf{f}}(\underline{\mathbf{x}}^{(k)})}$$

That is, in each step we start with a vector $\underline{\mathbf{x}}^{(k)}$ and then

1. Calculate $\underline{\mathbf{f}}(\underline{\mathbf{x}}^{(k)})$. (n function calls.)
2. Calculate $\underline{\underline{\mathbf{J}}}^{(k)}$. (n^2 gradient function calls.)
3. Solve the linear system $\underline{\underline{\mathbf{J}}} \underline{\underline{\Delta}} = -\underline{\mathbf{f}}$.
4. Update $\underline{\mathbf{x}}^{(k+1)} = \underline{\mathbf{x}}^{(k)} + \underline{\underline{\Delta}}$.
5. Iterate until we like where we are.

2.3.2 Form

Whether working with a single equation or many, the basic form is:

$$f(x) = 0$$

$$\underline{\mathbf{f}}(\underline{\mathbf{x}}) = \underline{\mathbf{0}}$$

where we can use NRI to find solutions, using techniques such as:

$$\Delta^{(k+1)} = -\frac{f(x^{(k)})}{f'(x^{(k)})}, \text{ where } \Delta^{(k+1)} \equiv x^{(k+1)} - x^{(k)}$$

$$\underline{\underline{\Delta}}^{(k+1)} = -\underline{\underline{\mathbf{J}}}^{-1} \underline{\mathbf{f}}(\underline{\mathbf{x}}^{(k)}), \text{ where } \underline{\underline{\Delta}}^{(k+1)} \equiv \underline{\mathbf{x}}^{(k+1)} - \underline{\mathbf{x}}^{(k)}$$

2.3.3 Convergence behavior

Reference: B2:67

Next we'll look at how fast this method converges to the true solution; this assumes we are “close” to the true solution already. First, we can get an estimate of how much the error reduced as each from the NRI equation itself by subtracting the true solution from both sides of the equation. That is,

$$x^{(k+1)} - \tilde{x} = \underbrace{x^{(k)} - \tilde{x}}_{\equiv -\epsilon^{(k)}} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

$$\epsilon^{(k+1)} = \epsilon^{(k)} + \frac{f(x^{(k)})}{f'(x^{(k)})}$$

Now, we just need to find something reasonable to put in for the ratio of f 's. As always, we start with a TSE; here we are expanding the f evaluated at the true solution \tilde{x} about f at our current value $x^{(k)}$.

$$\underbrace{f(\tilde{x})}_{=0} = f(x^{(k)}) + f'(x^{(k)}) \underbrace{(\tilde{x} - x^{(k)})}_{\equiv \epsilon^{(k)}} + \frac{1}{2} f''(x^{(k)}) \underbrace{(\tilde{x} - x^{(k)})^2}_{(\epsilon^{(k)})^2} + \frac{1}{6} f'''(x^{(k)}) \underbrace{(\tilde{x} - x^{(k)})^3}_{\text{cross out}} + \dots$$

Divide by f' :

$$0 \approx \underbrace{\frac{f(x^{(k)})}{f'(x^{(k)})} + \epsilon^{(k)}}_{=\epsilon^{(k+1)}} + \frac{1}{2} (\epsilon^{(k)})^2 \frac{f''(x^{(k)})}{f'(x^{(k)})}$$

So

$$\epsilon^{(k+1)} \approx -(\epsilon^{(k)})^2 \frac{f''(x^{(k)})}{2f'(x^{(k)})}$$

$$\epsilon^{(k+1)} \approx C(\epsilon^{(k)})^2$$

When we are getting close to the solution, x is not moving much and the ratio of the two functions is roughly a constant. Therefore, we can assume that the error decreases quadratically. E.g., if our error starts at 10^{-1} we can expect

$$10^{-1} \rightarrow 10^{-2} \rightarrow 10^{-4} \rightarrow 10^{-8} \rightarrow 10^{-16}$$

This is pretty good.

2.3.4 Secant method

Reference: K19.2, B2:69, B2:77

What if we don't know an analytical expression for our function, so we don't have a derivative (f' or \mathbf{J})? Perhaps we could approximate a derivative? From the definition of a derivative, if δx is small, then the below should be a decent approximation:

$$\frac{df}{dx} \approx \frac{f(x + \delta x) - f(x)}{\delta x}$$

The question is how small is small enough is a good one. From the standpoint of the definition, the smaller the better. However, we can imagine two problems with making it too small: first, if the function itself has “noise”³ than δx has to be large enough to show a meaningful signal outside the noise. Second, even

³This can occur for lots of reasons. For example, f could be the output of a complicated algorithm or even set of experimental measurements. In the case of a complicated algorithm, it would typically only be converged to within a specified tolerance, so small perturbations δx may not leave the noise of this tolerance.

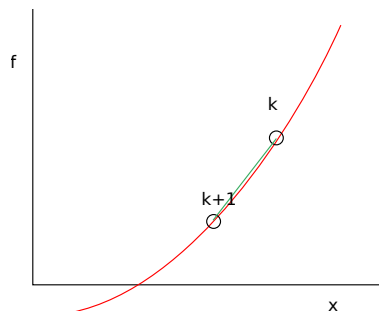


Figure 2.3: The slope estimate in the secant method.

for noiseless functions, at some point we hit precision limits of the computer we are using—that is, we run out of decimal places and we get noise introduced from the truncation/rounding of these small numbers. In the latter case, a recommendation is to choose $\delta \approx \sqrt{\text{eps}}$; where the term in the square root is the precision of the machine. These considerations will become more important in future sections where we solve differential equations.

However, using the above requires an additional function call, which can be the most expensive part of the algorithm. It turns out it's not a bad idea to just use the last step to make this estimate directly; see Figure 2.3. Then,

$$\left. \frac{df}{dx} \right|_{x^{(k)}} \approx \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}$$

Using the above as the estimate of the derivative is the *secant method*. We pay a bit of a penalty for this in the convergence behavior, and some smart people have derived that if we use this approximation the convergence is to the 1.6 power, rather than the 2nd power when we have the analytical derivative available.

Extension to systems: Broyden's method. The secant method only provides a slope estimate if we have a single function of a single unknown; for higher-dimensional systems, Broyden's method is used instead. This, and similar approaches, are used internally in many of the methods that are built into software packages such as *scipy*. We won't cover how Broyden's method works in this course, but it is described in the reading.

2.3.5 Linesearch

Reference: B2:79

A simple method is often added on top of Newton's method, and related approaches, to ensure that the function does not “fly away” to unreasonable solutions. To understand the need for this, let's look at a simple example of finding the roots of the following equation with Newton's method.

$$f(x) = 3x^3 + 4x^2 - 145x + 1.$$

$$f'(x) = 9x^2 + 8x - 145$$

(Since this is a simple polynomial, we can find that a solution exists at about -7.1.) If we use Newton's method and an initial guess of -4.5—which is reasonably close to our solution—we shoot out from an initial function value of 461 to fxn value of -155574898; see Table 2.1. This seems bad—we have been flung very far from our solution, and our function value is now further from 0 than when we started!

There is a simple algorithm for addressing this problem, known as the robust reduced-step linesearch algorithm. In this algorithm, we first propose a step size with Newton's method and evaluate the function. If the function is better than our last step, we accept it. If not, we divide Δx in two and evaluate the function again. This is shown in Figure 2.4. Of course, many variants of this approach can be proposed, and other approaches are discussed in the reading.

Table 2.1: Steps in a poorly performing optimization with standard Newton’s method.

Step	x	f	f'	$\Delta \equiv -f/f'$
0	-4.5	462	1.25	-368.9
1	-373.4	-155 574 898.5	\vdots	\vdots

Algorithm to generate $x^{[k+1]}$ from $x^{[k]}$:

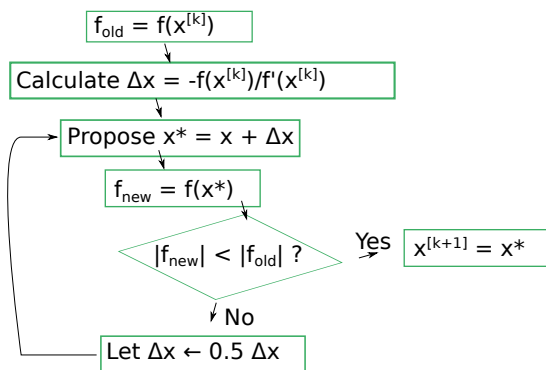


Figure 2.4: A simple linesearch method.

2.3.6 Numerical packages, and practical approaches

Finding the roots of a nonlinear function is a very common problem, and robust implementations of nonlinear algebraic solvers are implemented in all standard numerical solvers. Many sophisticated variants on the above approaches exist, and are implemented in these solvers. In this class, we’ll tend to use `scipy.optimize.fsolve`, which uses a linesearch method known as Powell’s method, and automatically approximates the differential/Jacobian if you do not supply one. Similar to the case with linear algebra methods, you typically should *not* implement your own solver, but rather you should pose your problem in a way that a standard solver can find the solution for you.

In the case of `scipy.optimize.fsolve`, you will create one or two functions, which you feed to the root finder. The first contains the function for which you want to find a root, and the second contains the derivative of the function, if you have it available.

2.4 Polynomial roots

Reference: B3:148

Since we are now examining nonlinear systems of equations, we should examine the special case of the polynomial. Consider a polynomial of the form:

$$p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 = 0$$

(Note that there is no coefficient in front of the first term; if yours has one, just divide it out to make the first coefficient 1.)

As we have discussed before, finding the roots of this problem is a completely equivalent problem to solving the eigenvalue problem. We can, of course, solve this with the nonlinear solvers discussed elsewhere in this topic; however, its equivalence with an eigenvalue problem allows us to use eigenvalue solvers to find the roots. This has some advantages—for example, we can find all the roots at once, rather than the one-by-one approach of the Newton’s method.

You will recall that in the textbook (by-hand) method of solving eigenvalue problems, §1.4.3, we take a determinant of $\underline{\underline{\mathbf{A}}} - \lambda \underline{\underline{\mathbf{I}}}$, which results in a polynomial. Here, we do this process in reverse to re-assemble a matrix from a polynomial. This matrix is called the *companion matrix* of our polynomial, and has the form

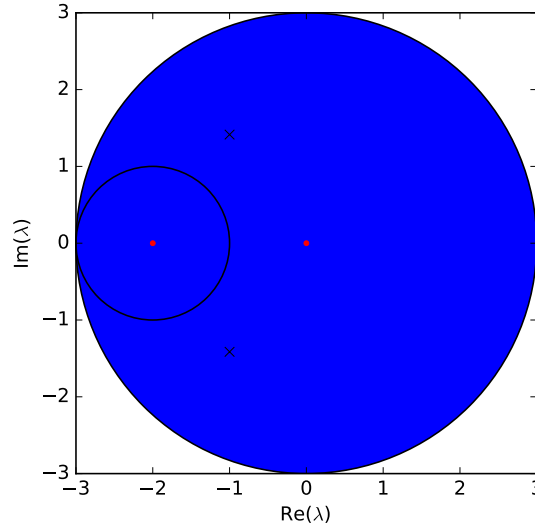


Figure 2.5: Gerschgorin disks for our 2nd-degree polynomial.

$$\mathbf{C}_{\equiv_p} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & -a_0 \\ 1 & 0 & \cdots & 0 & 0 & -a_1 \\ 0 & 1 & \cdots & 0 & 0 & -a_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & -a_{n-2} \\ 0 & 0 & \cdots & 0 & 1 & -a_{n-1} \end{bmatrix}$$

Example. Let's look at an example:

$$p(x) = x^2 + \underbrace{2}_{a_1}x + \underbrace{3}_{a_0} = 0$$

Of course, in this case we can just use the quadratic formula to find the roots. (We'll do that afterward to check.) For now, let's make our matrix.

$$\mathbf{C}_{\equiv_p} = \begin{bmatrix} 0 & -3 \\ 1 & -2 \end{bmatrix}$$

The assertion is that the eigenvalues of the companion matrix are the roots of the polynomial. Knowing this, we can actually go ahead and make Gerschgorin disks to bound our answers. From last time, we said that each eigenvalue must exist in a disk with

$$\lambda = a_{jj} \pm \sum_{k \neq j} |a_{jk}|$$

That is, the center of each disk is the matrix's diagonal, and the disk radius is the sum of the absolute value of the off-diagonal elements. See Figure 2.5; the disk centers are 0, -2; radii are 3, 1. So we already know where to look for our roots without solving our polynomial. Note one subtlety of the Gerschgorin disk theorem: if two (or more) disks overlap then their *combined* area contains at least two (or more) eigenvalues. Conversely, if a single disk sits alone then it contains exactly one eigenvalue. The true roots of this quadratic equation are $x = -1 \pm 1.414i$, which we've added to the plot, and we can see they are in the combined disk areas.

Let's also check the equivalence of the companion matrix to our original polynomial:

$$\underline{\underline{\mathbf{C}}}_p \underline{\mathbf{x}} = \lambda \underline{\mathbf{x}}$$

$$\left(\underline{\underline{\mathbf{C}}}_p - \lambda \underline{\underline{\mathbf{I}}}\right) \underline{\mathbf{x}} = \underline{\mathbf{0}}$$

We take the determinant of the quantity in parentheses to find the eigenvalues λ :

$$\begin{vmatrix} 0 - \lambda & -3 \\ 1 & -2 - \lambda \end{vmatrix} = 0$$

$$-\lambda(-2 - \lambda) + 3 = 0$$

$$\lambda^2 + 2\lambda + 3 = 0$$

That is, we recover the original polynomial.

Summary. The point here is the equivalency of the eigenvalue problem and polynomial root finding; one can be used to inform the other. If you have a massive polynomial, you can actually fairly quickly decide where all the roots must lie by Gerschgorin’s theorem. Similarly, all the other methods available to us from solving the eigenvalue problem are also valid for polynomials. Thus, for polynomials we don’t have to rely on one-by-one root finding, as we do with Newton’s method. We can also use the methods that search only for the dominant root (eigenvalue), or a handful of roots.

In practice, you can use a numeric polynomial solver (like `numpy.roots`), which operates by forming the companion matrix and finding its eigenvalues.

2.5 Root behavior

Here, we’ll look at some practical issues encountered when searching for roots, and what can be done about them.

2.5.1 Example

Reference: B2:64

Let’s play around with Newton’s method with a function we know the exact answer to, in order to examine how it works. Let’s try the cubic polynomial

$$f(x) = x^3 - 6x^2 + 11x - 6 = 0$$

Here, we already know the answer algebraically: this equation can be factored to find the roots 1, 2, and 3.

However, we’ll use Newton’s method to see how it works in this case. We can code⁴ up Newton’s method very simply as:

```
def get_f(x):
    return x**3 - 6. * x**2 + 11. * x - 6.

def get_fprime(x):
    return 3. * x**2 - 12 * x + 11.

def get_step(x):
    return - get_f(x) / get_fprime(x)
```

⁴Although it’s really easy to code up Newton’s method yourself, as mentioned in §2.3.6 you should typically not do this, but instead rely on pre-written codes such as `scipy.optimize.fsolve`. Here, we code up our own solely so we can closely examine how it works.

Table 2.2: Convergence of the indicated problem.

Initial guess	Result
0	1
0.5	1
1	1
1.5	3 ! In one step!
2	2
2.5	1 ! In one step!
3	3
1.4	1
1.6	2
1.55	3, but shoots to 12 first!
2.5	1 ! In one step!

```
x = 0.
for _ in range(10):
    x += get_step(x)
    print(x)
```

When we run this from the initial guess of $x^{(0)} = 0$, we should see it converge to the $x = 1$ root:

```
0.5454545454545454
0.8489532106224531
0.9746740710233017
0.9990915480569483
0.9999987646910549
0.999999999977106
0.9999999999999999
```

We also note the quadratic convergence behavior we expected from §2.3.3: on line 4 we have ~ 3 nines (0.999), the next line has ~ 6 nines (0.999999), and the following line has ~ 12 nines. We double the number of nines each time, as we get very close to our root of one.

Next we'll try varying the initial guess, to see when we find each root. We ran this in class, and the results are tabulated in Table 2.2.

As we see in the table, some roots can be hard to find, and we should always be aware of this in nonlinear systems. Imagine that in this function we had no idea where our roots were, so we just started trying the odd numbers to span space; we would never realize that we were missing 2.

Can we understand this? Figure 2.6 shows the function whose roots we were trying to find, along with its derivative, and the update function (f/f'). We can already understand the weird behavior that we observed with initial guesses of around 1.5 by noting that the slope of f is about 0; this results in division by a very small number, and large values of the update function that change sign at infinity!

2.5.2 Factoring out roots

Reference: B2:66

In the previous case, roots 1 and 3 were relatively easy to find, but the other one was challenging. In general, once we have found a root we can just factor it out of our original function to create a function that is absent of that root. *E.g.*, say we were searching for roots of f and had found the roots at $x = 1$ and $x = 3$, and want to know if we are missing further roots. Then we can define g that has these roots factored out:

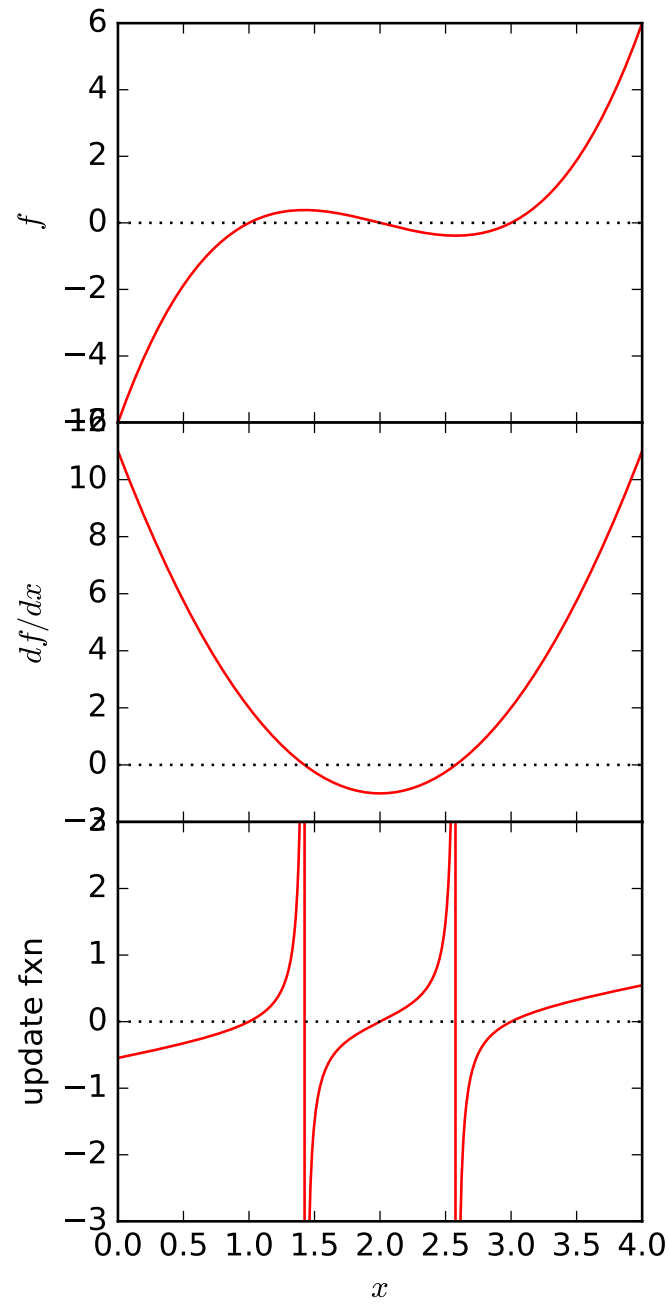


Figure 2.6: The function $x^3 - 6x^2 + 11x - 6$, its derivative, and its update function in Newton's method. This example is taken from B2:65.

$$g(x) = \frac{f(x)}{(x-1)(x-3)} = 0$$

(Where f would have gone to zero we divide by another zero.) Then we just use our root finder with g instead of f . Note that if we are supplying f' , we will need to modify it as well; however, this modification is systematic with the chain rule. In principle, we can keep pulling out roots one-by-one until we find everything; however, in practice there can be noise if the algorithm explores g near our found roots, so we should certainly be careful not to use our found roots (1 and 3 in the above discussion) as initial guesses.

Figure 2.7 shows our function, and the new function g which has the root at $x = 1$ factored out. We see that we are left with only the roots at 2 and 3 in our new function. Of course, in this particular example we started with the simple polynomial $(x-1)(x-2)(x-3)$, so that factoring works is not surprising. To demonstrate that this also works on more complex functions, Figure 2.7 also shows the same procedure on the below equation.

$$f(x) = e^x - 3x - 5$$

2.5.3 Bifurcation

Reference: B2:94

Example: an exothermic reactor (cigarette lighter)

We will start with a famous physical example of bifurcation: a cigarette lighter. If we push the button on a cigarette lighter, fuel flows out and one of two things happens: (1) there is a flame, or (2) there is no flame. This occurs, at steady state, even if all parameters are the same. (We can control which answer we get by either providing, or not providing, a spark.) This is an example of a non-linear system in which there are at least two physically-meaningful roots to our governing equations.

This is a well-known problem in reactions,⁵ and we can express the governing equation for an idealized (perfectly-mixed) exothermic reaction as:

$$\underbrace{\frac{(T - T^0)}{J}}_{\text{heat removal}} = \underbrace{\frac{k(T)}{\frac{v}{V} + k(T)}}_{\text{heat generation}}$$

where T is the reactor temperature. T^0 is the inlet (fuel) temperature; J contains lumped physical parameters of the system (mainly heat capacity and heat of reaction). $k(T)$ is the reaction rate coefficient, which is an exponential function of temperature. v is the volumetric flow rate and V is the reactor volume.

The right side is called the generation curve—which is highly non-linear—and the left side is called the removal curve, which is linear. We can solve this graphically: we plot the left side and the right side separately; where the two intersect we have a solution. This is shown in Figure 2.8.

In this figure, the central plot shows the case with multiple solutions, while the left and right plots show different parameter choices that result in only one solution. First, focus on the multiple-solution plot: there are apparently three roots, but we could only think of two when we looked at a cigarette lighter, that is, “flame on” and “flame off”. Let’s examine the roots:

- Let’s assume the top root—which is at the highest reactor temperature—is “flame on” and examine it. If we imagine perturbing the system slightly by *increasing* the temperature away from the solution value, we have that the removal term is greater than the generation term; thus, there will be a driving force for the temperature to reduce back towards the solution. We can make a similar argument if we perturb the temperature down slightly: the generation curve will be higher than the removal curve, and there is a driving force for the temperature to rise towards the solution. We would say this is a stable solution.

⁵You can find the derivation in most kinetics textbooks; see, for example Chapter 6 of Schmidt. [4]

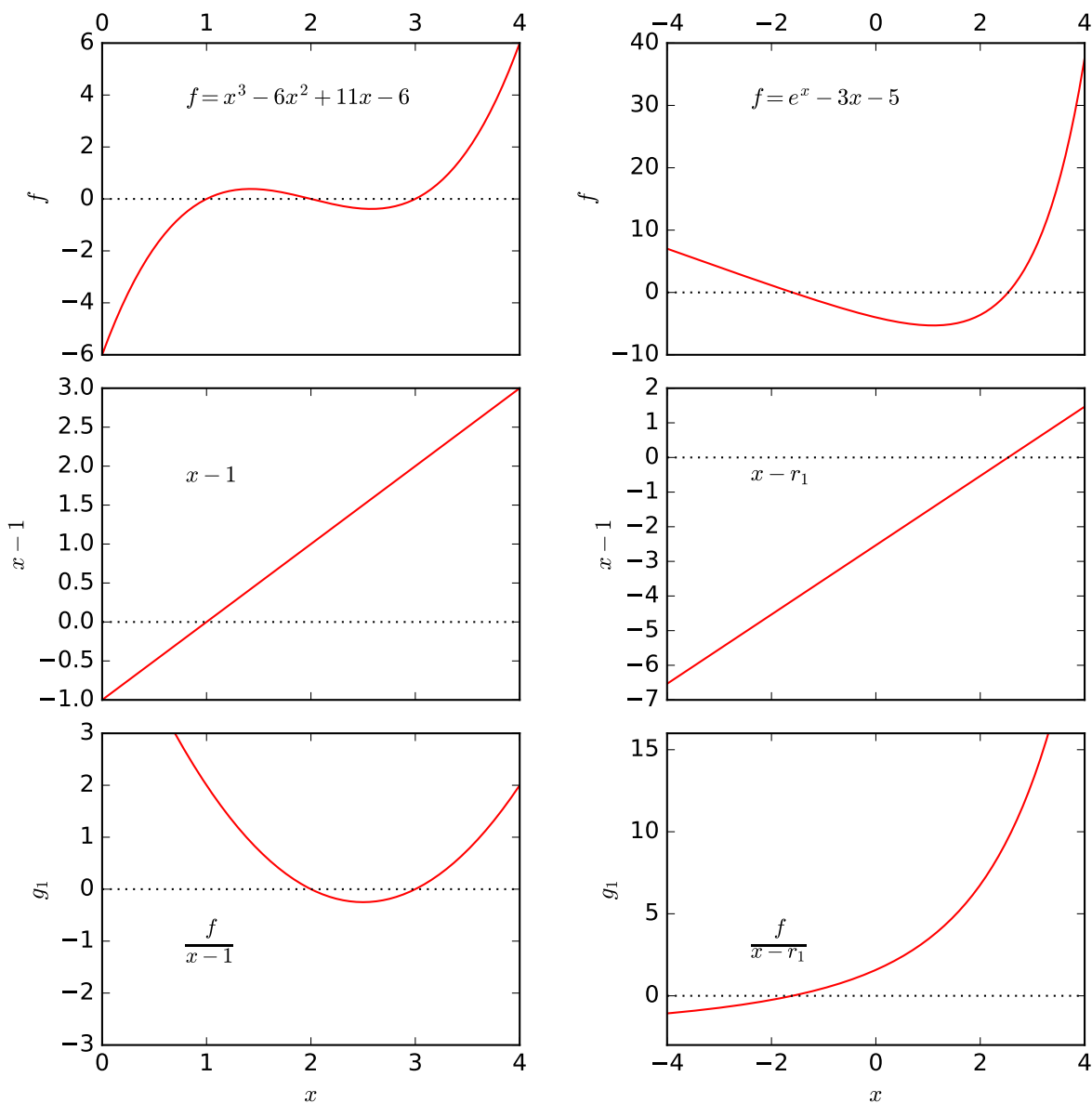


Figure 2.7: Factoring out $x - r_1$ to get $g = f/(x - r_1)$ for two different functions and their roots r_1 .

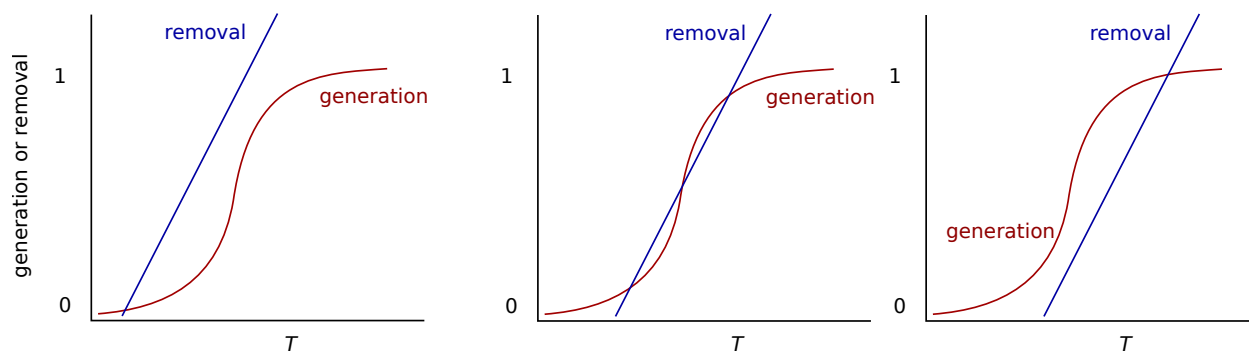


Figure 2.8: Generation and removal curves for a CSTR with exothermic reaction.

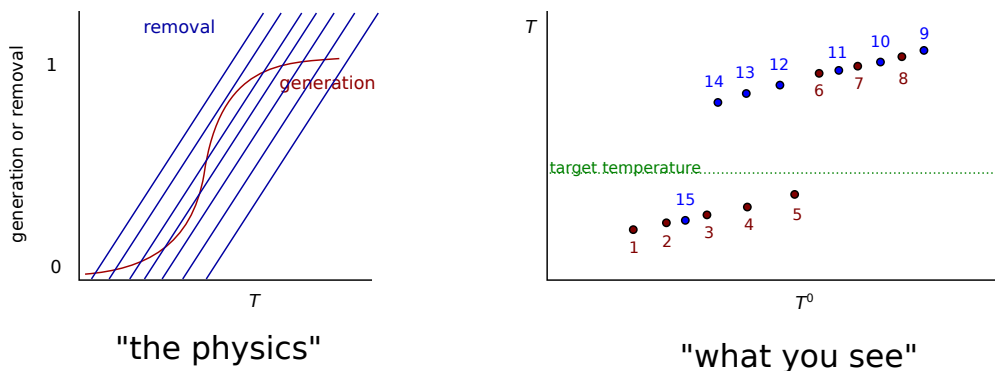


Figure 2.9: Hysteresis curves: the operator controls T^0 .

- The lowest-temperature solution behaves just the same way as the top one: it is a stable solution. We can label this the “flame off” solution.
- However, the middle solution is fundamentally different. Perturb the temperature upwards, and the generation curves is higher than the removal curve. This solution is actually avoided, and is a dividing line between the two stable solutions.

We will explore the stability and behavior in much more depth when we examine differential equations. For now, we’ll keep our attention just on when we can expect the number of roots to change.

We have control over the parameters in the equation above, and if we consider adjusting T^0 ; the feed temperature, we will shift the removal curve while keeping the generation curve in a fixed location. As highlighted in the figure, this means we can move to regions where *only* the low-temperature or *only* the high-temperature solution is possible. Thus, we move in and out of the region of multiple-roots, and this act of moving between regions of solution space is called *bifurcation*. If you consider moving the blue line, we go from a region where we have one solution to a region where we have three solutions. At the dividing line between these two regions, we have one specific *point* where we have two solutions; this dividing point is called a *bifurcation point*.

This can lead to confusing, and even dangerous operations, if this behavior is not understood. See Figure 2.9. Imagine that you are the operator in charge of running this reactor, and you can use T^0 , the feed temperature, to control the reactor temperature. Assume that you have decided to target a certain temperature, and you are currently below that value, at point #1 in the right-hand figure. You increase the T^0 , and you see the reactor temperature steadily rise, as in points 1–5 on the figure. However, suddenly, you increase the temperature further and the reactor temperature jumps, dramatically overshooting your target (points 5–6). Now you decide you should turn the cooling back down, again trying to get to your target temperature. When you get to point #12, you are past the point where the jump occurred on the way up, and you expect it to jump back down. However, it doesn’t, and it’s not until point #15 that it jumps down.

Perhaps surprisingly, we can never dial-in our target temperature, although we can dial in temperatures above and below this value. This would be rather perplexing behavior if you weren’t aware of the governing equations that cause it to occur. This sort of behavior is referred to as hysteresis, and we should really learn to understand when our system is entering regions when we can expect weird stuff to happen.

General solution strategy

One dimension. If we have a single governing equation, as in the previous example, then solution strategies such as the one we undertook in the example are the logical choice. You should be able to come up with similar ways to present the data for your particular problem type.

Multiple dimensions. If we have multiple dimensions, a graphical solution such as above is not feasible, and we need a more generic approach. We’ll start with two functions of two unknowns—this way, we can still track things graphically while developing a general solution strategy. Here, we’ll also assume the functions

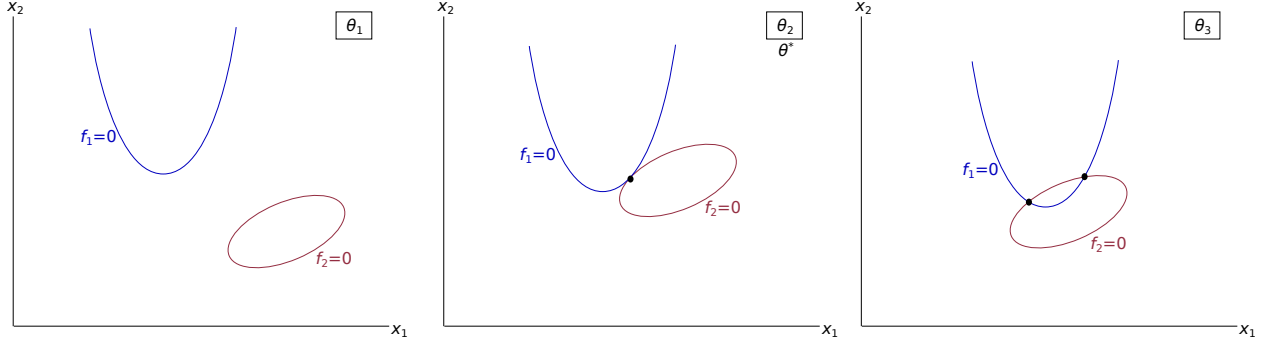


Figure 2.10: Bifurcation diagram for two-dimensional example.

carry some single parameter θ that we'll use to decide which domain we are in—this is analogous to T^0 in the prior example.

$$f_1(x_1, x_2; \theta) = 0 \quad (2.1)$$

$$f_2(x_1, x_2; \theta) = 0 \quad (2.2)$$

Now, we will plot the $f_1 = 0$ and $f_2 = 0$ curves on the x_1 - x_2 plane; the points of intersection of these two curves mark solutions to the system. Let's assume they look like Figure 2.10. Again, we see possibilities to have 0, 1, or 2 solutions; the middle point, which we've marked as θ^* , is the bifurcation point that divides the 0- and 2-solution regions.

We would like to find this special bifurcation point, which we can see occurs when the two curves just “touch”. Mathematically, this occurs when the tangents of the two curves are equal, which means that the slope dx_2/dx_1 of both curves are equal.

We need to find dx_2/dx_1 for both curves; to get these terms we take the total differentials of our above equations:

$$df_1 = \left(\frac{\partial f_1}{\partial x_1} \right)_{x_2} dx_1 + \left(\frac{\partial f_1}{\partial x_2} \right)_{x_2} dx_2 = 0$$

$$df_2 = \left(\frac{\partial f_2}{\partial x_1} \right)_{x_2} dx_1 + \left(\frac{\partial f_2}{\partial x_2} \right)_{x_2} dx_2 = 0$$

We divide both equations by dx_1 and note that dx_2/dx_1 must be equal in the two equations, so we'll call this quantity $S \equiv \frac{dx_2}{dx_1}$. Also, note that our partial derivatives are the elements of the Jacobian matrix, J_{ij} . Then, we can write:

$$J_{11} + J_{12}S = 0$$

$$J_{21} + J_{22}S = 0$$

$$\begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix} \begin{bmatrix} 1 \\ S \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

By Cramer's rule⁶ this has a non-trivial solution only when the determinant of our matrix is equal to zero:

$$\det(\underline{\mathbf{J}}(\theta)) = 0 \quad (2.3)$$

⁶If the determinant is not equal to zero, it has only the trivial solution. Yet the trivial solution is not possible because the “1” cannot equal zero.

We now have enough information to find bifurcation points. Recall that each bifurcation point is defined by three unknowns: x_1 , x_2 , and θ . We originally had only two equations (eqs. (2.1) and (2.2)); this gives us the third equation (eq. (2.3)) to also find θ and locate the bifurcation points. Of course, this is no panacea; we are still left with a system of equations to solve. However, we now have a logical manner to find these points at which the solution behavior changes. An example of finding bifurcation points will be provided on the homework.

Topic 3

Optimization of continuous functions

3.1 Optimization

Reference: B5, K22

Optimization is incredibly common in engineering and physics, and is perhaps the most commonly used numerical method in day-to-day use. Optimization is useful in obvious ways: like optimizing devices or processes or economics, but also in many other numerical tasks, like finding the best-fit parameters for a model or finding a converged solution to a modeled problem. In this section, we'll discuss optimization of continuous functions; the next section will cover graph theory in which optimization involving discrete variables is covered.

Key points of optimization We'll first hit the most salient points:

1. *You can only optimize one variable.* We started by examining Dilbert; Figure 3.1; the bosses are trying to optimize the money flowing in, while Dilbert's trying to optimize on quality. The major problem is that you can only choose to optimize one thing—this is perhaps the most important point of optimization. In business, you can't simultaneously maximize earnings *and* optimize worker happiness *and* optimize environmental performance *and* optimize customer satisfaction.¹

In Figure 3.2, we can see the problem graphically: there is not a value of x that gives both a minimum in f_1 and in f_2 . Note that if we want to optimize in these two variables we need to combine them into a single function, like

$$f = f_1 + cf_2$$

Of course, the choice of c is typically arbitrary—for example, f_1 and f_2 might not even have the same units—and will greatly affect what outcome we get. Note something else we see here: there are multiple local minima; this is incredibly common in optimization. Note also that the location of the global minimum will change depending on our choice of c .

2. *Loss functions in general have many local minima, which are “easy” to find.* As we just saw, it's common to have many local minima; we also show a higher-dimensional version of this in Figure 3.2. We can typically find the local minimum of a function by “walking downhill”, which is the basis of most methods.

¹Fortunately for this example, no business knows perfectly how those factors influence earnings, so they might say that they want to maximize quality in order to maximize customer satisfaction, which will lead to more long-term sales, assuming these things are correlated. However, it's unlikely that this will *also* improve environmental performance or worker happiness. This is the economic justification for regulations, which put prices or constraints on these factors so that they are included when businesses optimize their economics.

DILBERT

BY SCOTT ADAMS

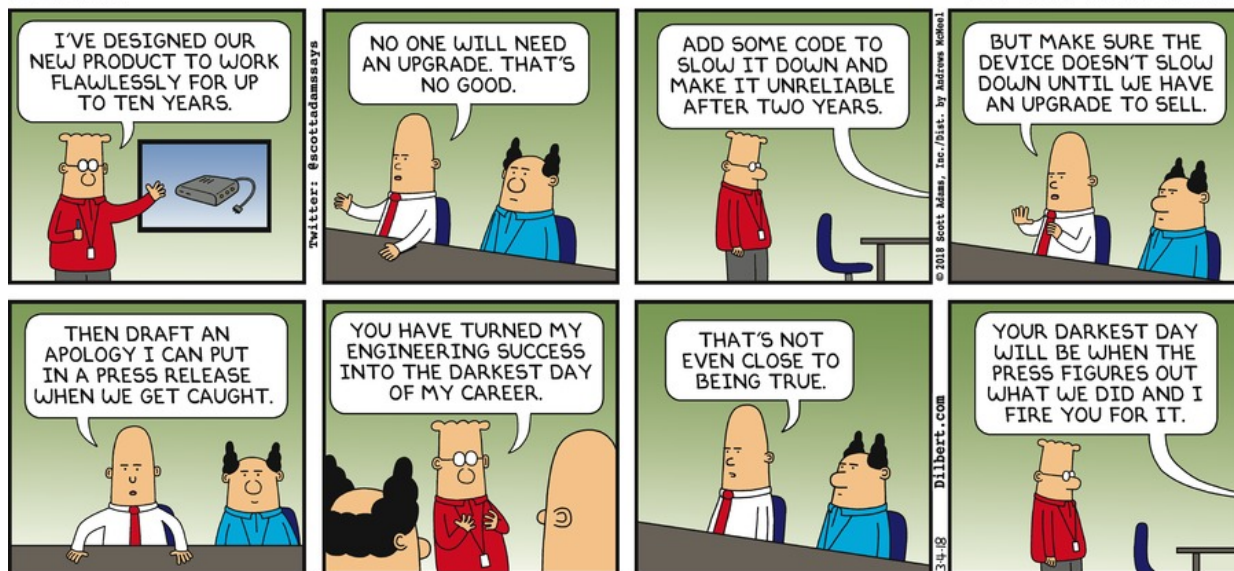


Figure 3.1: Dilbert and his bosses are trying to optimize two different functions. (Copyright Scott Adams.)

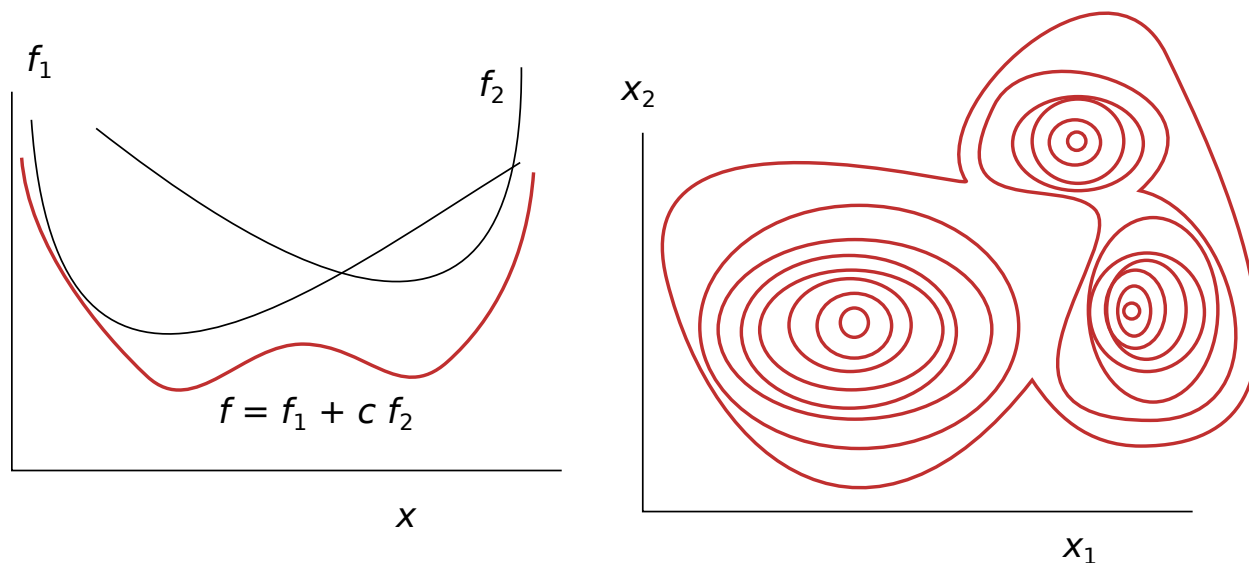


Figure 3.2: (Left) If we have two quantities, f_1 and f_2 , that we want to optimize as a function of x , we cannot find one value of x that optimizes both. We can, however, weight and add them, and look for an optimum of the combined function. (Right) A two dimensional contour map showing a loss function with many local minima.

3. *The global minimum can be hard to find.* Clearly, which local minimum we hit will depend on where we start walking from, and thus finding a local minimum is relatively easy. However, finding the global minimum is typically much harder.
4. *Optimization and NLAE are highly related.* For continuous functions, optimization and systems of nonlinear algebraic equations (NLAE) mathematically reduce to the same problem. (This is why `fsolve` is in `scipy.optimize`!) Most of what we covered in the NLAE section can be directly adapted to optimization.

3.1.1 Problem formulation

The (single!) function that we optimize is typically² called the “cost function”. Since we all want to pay the lowest cost, we conventionally describe optimization as a minimization problem:

$$\underset{\underline{\mathbf{x}}}{\text{minimize}} f(\underline{\mathbf{x}})$$

If you’d instead like to maximize a function, just multiply it by -1 , or take its inverse, as appropriate, to convert it into a minimization problem. Note that although we can only optimize one variable f , it can depend on as many independent variables as we like, $\underline{\mathbf{x}}$. This is indicated by the variable $\underline{\mathbf{x}}$ that appears under the word “minimize”.

3.2 Gradient-based methods

Reference: B5:216

Mathematically, a necessary (but not sufficient) condition to state that we are at a local minimum is that the gradient is zero:

$$\nabla f = 0$$

This is not sufficient to say we are at a minimum in f —we could be at any stationary point³, which includes local maxima and other more complex features, such as saddle points. However, since most techniques we’ll discuss involve “walking downhill”, it’s likely that the stationary point we’ll hit will be a local minimum.⁴ Also, note the above equation applies equally at local minima or the global minimum, so the techniques we’ll discuss here only apply to finding nearby local minima.

This gradient is a system of equations:

$$\begin{aligned} \left(\frac{\partial f}{\partial x_1} \right) &= 0 \\ \left(\frac{\partial f}{\partial x_2} \right) &= 0 \\ &\vdots \end{aligned}$$

That is, this is just a system of nonlinear algebraic equations as we discussed earlier in Section §2, and we can use or adapt the methods from this section. To simplify the notation a bit, we often just refer to the gradient of f as $\underline{\gamma}$:

$$\underline{\gamma} \equiv \nabla f = \underline{\mathbf{0}}$$

²Other common names included “loss function” and “objective function”.

³Some mathematicians call these “critical points”, but we use that term in the physical sciences for something else.

⁴Local maxima can be ruled out; in some very complicated cost functions there is some non-zero chance of landing at a saddle point, but this is unusual.

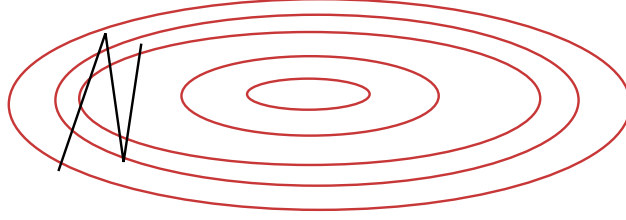


Figure 3.3: The trouble with taking steps straight downhill.

3.2.1 General step.

If we walk downhill we will be going towards a minimum.⁵ Gradient-based methods start from a guess vector $\underline{\mathbf{x}}^{[0]}$ and take downhill steps. We can define a general step like:

$$\underline{\mathbf{x}}^{[k+1]} = \underline{\mathbf{x}}^{[k]} + \alpha^{[k]} \underline{\mathbf{p}}^{[k]}$$

Here, the superscripts like $^{[k]}$ refer to the value at the k^{th} step. The direction of the step need not be the steepest way down the hill, and it can be better in general to choose something that is not the steepest descent. Here, $\underline{\mathbf{p}}^{[k]}$ is a vector pointing downhill (mathematically, $\underline{\mathbf{p}}^T \underline{\gamma} < 0$, where $\underline{\gamma}$ is the true gradient of f), and α is the step size.

That is, we need to choose the step size and step direction, and we have an algorithm.

3.2.2 Search direction.

Let's discuss how to choose which direction we should move in.

Steepest descent. The most obvious choice is to just go straight downhill; this is called the “steepest descent” algorithm. That is,

$$\underline{\mathbf{p}}^{[k]} = -\underline{\gamma}^{[k]}$$

(and $\alpha^{[k]}$ may be some user-defined parameter.)

Figure 3.3 shows the issue with this—we can often bounce around on the cost function surface, and the convergence can be rather slow and erratic.

Conjugate gradient. One solution to the bouncing-around problem is to “mix” in some of the last step with the next step, as in

$$\underline{\mathbf{p}}^{[k]} = -\underline{\gamma}^{[k]} + \beta \underline{\mathbf{p}}^{[k-1]}$$

If you work out what this would do to the second step in Figure 3.3, you can see that we would cancel out some of the bouncing, while keeping most of the direction we would like to go in. β need not be defined by the user, but instead can be chosen automatically via a couple of different formulations. *E.g.*, Fletcher–Reeves:

$$\beta = \frac{\underline{\gamma}^{[k]} \cdot \underline{\gamma}^{[k]}}{\underline{\gamma}^{[k-1]} \cdot \underline{\gamma}^{[k-1]}}$$

or Polak–Ribiere,

$$\beta = \frac{\underline{\gamma}^{[k]} \cdot (\underline{\gamma}^{[k]} - \underline{\gamma}^{[k-1]})}{\underline{\gamma}^{[k-1]} \cdot \underline{\gamma}^{[k-1]}}$$

where the latter is generally considered better. If you choose ‘CG’ in `scipy.optimize.minimize`, you will get Polak–Ribiere.

⁵Or negative infinity; but then you probably didn’t formulate your problem well.

3.2.3 Step size

Reference: B5:216

We also discussed linesearch in NLAE, let's look at that in more depth here. Once we have a good search direction, perhaps we can now perform a one-dimensional optimization? Optimization in one dimension is easier, so this could be done to find the absolute minimum along this search direction. If we do this, it is called a *strong linesearch*. However, since it's not so likely that the true minimum lies along this line, then this is kind of a waste and we're better off switching directions once we know we've started going downhill.

So typically this is done by a *weak linesearch*, which we described earlier in NLAE. That is, start with $a^{[k]} = a_{\max}$, if this does not decrease F , then

$$a^{[k]} \leftarrow \frac{a^{[k]}}{2}$$

This, specifically, is called backtrack linesearch. There are also means for finding a good value of a_{\max} , and for making more stringent criteria than just a decrease in f . See Beers pp. 213–217 if you want to know all those details.

3.3 Hessian-based methods

What we'll discuss here is entirely analogous to the discussion of the Jacobian matrix in NLAE. However, here we use the term *Hessian* matrix, because in an optimization problem we have a second, not a first, derivative. But the mechanics are entirely the same.

First, let's remember that at our true solution, $\underline{\gamma} = 0$, so at a perfect step size and direction $\underline{\mathbf{p}}$ we have $\underline{\gamma}(\underline{\mathbf{x}}^{[k]} + \underline{\mathbf{p}}) = 0$. How can we estimate the perfect step $\underline{\mathbf{p}}$? When in doubt, use a Taylor series expansion:

$$\underline{\gamma}(\underline{\mathbf{x}}^{[k]} + \underline{\mathbf{p}}) = \underline{\gamma}(\underline{\mathbf{x}}^{[k]}) + \underline{\underline{\mathbf{H}}}(\underline{\mathbf{x}}^{[k]})\underline{\mathbf{p}} + \dots$$

where $\underline{\underline{\mathbf{H}}}$ is the Hessian matrix, which we can see must be defined as

$$\underline{\underline{\mathbf{H}}} = [H_{ij}]$$

$$H_{ij} = \left(\frac{\partial \gamma_i}{\partial x_j} \right) = \left(\frac{\partial^2 f}{\partial x_i \partial x_j} \right)$$

That is, it is the matrix that describes the curvature of the function f at a certain point $\underline{\mathbf{x}}$. So, since we said the left-hand side of the Taylor series was zero, we can re-write this as the linear system

$$\underline{\underline{\mathbf{H}}}^{[k]}\underline{\mathbf{p}}^{[k]} = -\underline{\gamma}^{[k]}$$

So if we know $\underline{\underline{\mathbf{H}}}$, we can make a really good guess of where we should take our next step. (We can then use 1 as our a_{\max} described earlier in a linesearch implementation.)

Hessian, the person. Otto Hesse was a German mathematician. His PhD advisor was Carl Jacobi, of Jacobian matrix fame. The Hessian is the matrix of second derivatives, and the Jacobian is the matrix of first derivatives.

3.3.1 Approximate Hessians (Quasi-Newton)

In practice, the Hessian is often not known or hard to calculate. (But if it is available, use it!), and so methods exist to generate an approximate form of $\underline{\underline{\mathbf{H}}}$ during the course of the optimization, building up an approximate Hessian which we'll call $\underline{\underline{\mathbf{B}}}$. For example, we might start with $\underline{\underline{\mathbf{B}}}^{[0]} = \underline{\underline{\mathbf{I}}}$, then update with

$$\underline{\underline{\mathbf{B}}}^{[k+1]} = \underline{\underline{\mathbf{B}}}^{[k]} + \frac{\underline{\Delta\gamma}\underline{\Delta\gamma}^T}{\underline{\Delta\gamma}^T\underline{\Delta\mathbf{x}}} - \frac{\underline{\underline{\mathbf{B}}}^{[k]}\underline{\Delta\mathbf{x}}\underline{\Delta\mathbf{x}}^T\underline{\underline{\mathbf{B}}}^{[k]}}{\underline{\Delta\mathbf{x}}^T\underline{\underline{\mathbf{B}}}^{[k]}\underline{\Delta\mathbf{x}}}$$

This is known as the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm, which is probably the most popular optimizer and is considered the best general-purpose optimizer. This is what `scipy.optimize.minimize` defaults to for normal problems. (Note the delta terms are the difference in those vectors over the last two steps.) There also limited memory versions of this, known as L-BFGS, which can be useful if you have a very large number of variables in $\underline{\mathbf{x}}$.

3.4 Non-linear regression: parameter estimation

One very common use of optimization—including the `minimize` functions of `scipy`—is in performing generalized regression. Most of us are very familiar with performing linear regression—which you can easily do in a spreadsheet, for instance—in which you have some set of x, y data for which you want the best-fit line. Linear regression gives you the slope and intercept that minimize the error in the predicted line.

Regression to a nonlinear model is not really any harder, especially if you are working in a programming language like python or matlab.

First, you write a model. You can use whatever physics you like (hopefully something relevant!) to create the model. The model can have as many independent and dependent variables as you like, and in general there is no limit on the number of (adjustable) parameters in the model.⁶ Let's say that our model is of the form:

$$y^{\text{pred}} = f(\underline{\mathbf{x}}; \underline{\mathbf{P}})$$

where y^{pred} is the model's output (dependent variable), $\underline{\mathbf{x}}$ is the model's input (independent variables), and $\underline{\mathbf{P}}$ are the parameters to be adjusted. We assume that we have a series of measurements of $\underline{\mathbf{x}}$ and corresponding measurements of y ; each pair will be denoted as $(\underline{\mathbf{x}}_i, y_i)$.

Next, you construct a cost function whose purpose is to reduce the errors on the model prediction; you can construct this however you like. The most common is to reduce the sum of square errors (SSR):

$$\text{SSR}(\underline{\mathbf{P}}) \equiv \sum_i^{N_{\text{measurements}}} (y^{\text{pred}}(\underline{\mathbf{x}}_i; \underline{\mathbf{P}}) - y_i)^2$$

where $\{\underline{\mathbf{x}}_i, y_i\}$ is the set of data to fit. The optimization task is then to minimize this loss function by varying $\underline{\mathbf{P}}$:

$$\underset{\underline{\mathbf{P}}}{\text{minimize}} \text{SSR}(\underline{\mathbf{P}})$$

Below, we show a small example of how this can be written in practice; this assumes your measured data is stored in `xs`, `ys`.

```
from scipy.optimize import minimize

def get_ypred(x, parameters):
    # Your model here!
    # E.g., for a line
    slope, intercept = parameters
    return slope * x + intercept

def get_loss(parameters):
    SSR = 0.
    for x, y in zip(xs, ys):
        SSR += (y - get_ypred(x, parameters))**2
    return SSR

answer = minimize(get_loss, p0)
```

⁶We will leave the *quality* of your model to your particular field of expertise. In most fields, having a lot of adjustable parameters indicates a poor understanding of the physics of the problem. A key exception is machine learning, where the philosophy is to instead throw your hands up in the air and embrace adjustable parameters...

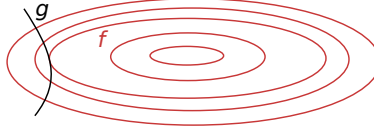


Figure 3.4: Constrained optimization. Think of f as a contour map (the lowest point is in the center); and we are hiking along g and looking for the lowest point we'll hike through.

Here, `p0` contains the initial guess of the parameters. Of course, if you can calculate the derivative of the loss function you should⁷ do that too, as it will make the optimization much more efficient.

3.5 Constrained optimization

Constrained optimization is inherently harder than unconstrained optimization, but it is very common. Constraints can result from many conditions specific to your problem: *e.g.*, mass cannot be negative, some emission level must be below a specified level, etc. Here, we'll briefly describe three ways to deal with constraints, which depend upon your particular problem.

3.5.1 Bounded variables

The simplest case is if your *dependent* variables have some bounds, which is conventionally written as:

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} && f(\mathbf{x}) \\ &\text{subject to} && \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max} \end{aligned}$$

This situation is relatively straightforward; the optimizer just algorithmically does not allow the constrained variables to step outside of the given ranges. (Actual algorithms are more sophisticated, of course, but we won't go through the details of how this is accomplished.) For example, you can supply the `bounds` keyword to `scipy` to do this. The syntax is similar in other codes such as Matlab.

3.5.2 Mathematically constrained

Alternatively, we can have mathematical constraints. These problems are generally written as

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} && f(\mathbf{x}) \\ &\text{subject to} && g_i(\mathbf{x}) = 0, i = 1, 2, \dots, n_i \\ & && h_j(\mathbf{x}) \geq 0, j = 1, 2, \dots, n_j \end{aligned}$$

where some functions, in the form of equalities or inequalities, need to be satisfied.

See Figure 3.4, which illustrates the case for a constraint where we are minimizing a function f along a constraint g . I like to think of this as if you are hiking a path (g) which is along the edge of a valley, and you are wondering what the lowest altitude is that you will encounter on this path. So f gives the height of the land, but g is the constraint as to where you can walk.

Adding terms to the loss function. Perhaps the easiest approach is to just add *penalties* to your loss function that add to it when your constraint is violated. For example, say you have a problem where you are trying to fit parameters to a model, as in §3.4; that is, you are minimizing the SSR (sum of square residuals). But let's also say we have a constraint; perhaps all the x_i 's in \mathbf{x} have to sum to one. (For example, if the

⁷If your only concern is computational efficiency, you definitely should. However, you can argue that if your objective is getting to the *right* solution, and your problem is cheap, you may not want to because this introduces a second place where you could have a bug in your code.

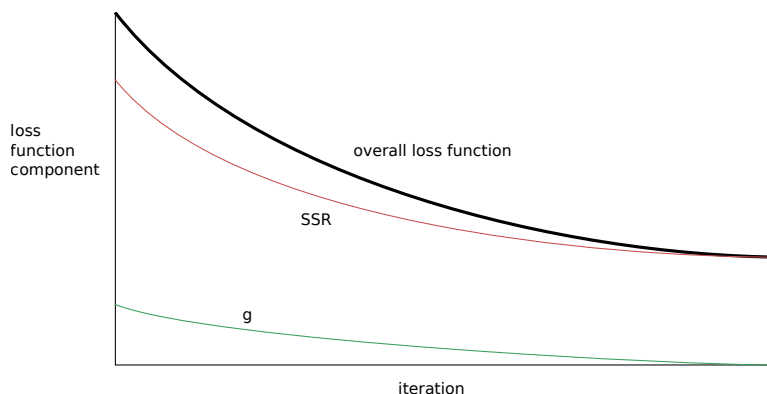


Figure 3.5: Example of a loss function that contains a built-in constraint.

x_i 's are chemical concentrations expressed in mole fractions, they need to sum to one.) In this case, we can write this problem as

$$\underset{\mathbf{x}}{\text{minimize}} \quad \text{SSR}(\mathbf{x})$$

$$\text{subject to} \quad g(\mathbf{x}) = \sum_i x_i - 1 = 0$$

Then, we can construct our loss function such that it includes this penalty. For example, in python:

```
def get_loss(parameters):
    SSR = 0.
    for x, y in zip(xs, ys):
        SSR += (y - get_ypred(x, parameters))**2
    penalty = (np.sum(xs) - 1.)**2
    loss = SSR + c1 * penalty
    return loss
```

You can use the constant `c1` to determine how much weight is put on reducing the residuals versus enforcing the constraint. Note that although you can only optimize with respect to one variable, you can monitor multiple variables during the course of the optimization—that is, you can check the value of $g(x)$ when your minimization routine finishes, and if it's not as close to zero as you'd like, you can adjust your strategy (such as increasing `c1`). An example of the combined decrease of g and SSR, making up the loss function is shown in Figure 3.5.

Lagrange multipliers. Lagrange multipliers are a well-known way to include constraints in optimization problems. Since it's covered in ENGN 2010, we'll only briefly mention this topic here.

You can work out intuitively (Figure 3.4), or mathematically, that the condition where you hit a local minimum is when the two functions are parallel, which is written by

$$\nabla f = \lambda \nabla g \quad (\text{at optimum})$$

$$\mathcal{L}(\mathbf{x}, \lambda) \equiv f(\mathbf{x}) - \lambda g(\mathbf{x})$$

Note that this is now a function of an extra variable, λ . If we take its gradient with respect to all the variables, we get

$$\nabla \mathcal{L} = 0$$

$$0 = \left(\frac{\partial f}{\partial x_i} \right) - \lambda \left(\frac{\partial g}{\partial x_i} \right)$$

(for all i) and

$$g(\mathbf{x}) = 0$$

So we recover our constraint, but we have a simple expression in \mathcal{L} .

This is an elegant technique, and is often useful in derivations. Interestingly, the best definition I know of for temperature is that it is a Lagrange multiplier! That is, if you maximize the entropy (*i.e.*, probability) of a large system, under the constraint that its total entropy is constant, you end up with an extra Lagrange multiplier β that cannot be removed. (We end up defining temperature as $\beta \equiv 1/k_B T$, where k_B is the Boltzmann constant.) You can find a derivation of this in many textbooks, for example Dill & Bromberg. [5]

3.6 Gradient-free methods

Often, the gradient of the function f to be minimized is unknown. This may be because f is a very complicated function (like the output of some other complicated algorithm), or perhaps because you don't want to take the effort (and risk) of deriving and coding the gradient of f . In this case, you have two choices: (1) use a numeric approximation of the gradient, or (2) use a gradient-free method.

In the former case (using a numeric approximation of ∇f), the gradient is typically approximated by perturbing each independent variable (that is, \mathbf{x}) and observing the response in f ; this is the same approach as in [\[link to come...\]](#). This is the approach used in `scipy`'s BFGS implementation, for example. The basic cost of this is an extra n function calls per iteration, where n is the number of elements in \mathbf{x} . This can lead to a large number of function calls in this process, so if the cost of the function call is high, it's typically worth examining whether this approximate gradient is worth it. (That is, put an variable inside your loss function that tracks the number of times it is called; this can be as simple as adding a print statement.⁸) Additionally, some “noisy” functions are challenging to estimate the derivative to analytically; noisy functions can arise, for example, if the function itself is a complicated algorithm that is solved iteratively, there will inherently be some noise.

Therefore, it can sometimes be useful to employ a gradient-free method, which optimizes only based on the value of the function f without trying to use the gradient in the process. The most famous is the Nelder–Mead algorithm, which we describe below. (This is the algorithm used in the `scipy.optimize.fmin` algorithm.)

3.6.1 Nelder–Mead (downhill simplex)

Reference: K22.3-22.4, B5:213

This algorithm relies on a “simplex”; think of a simplex as a generalized triangle. That is, a triangle is the smallest shape that fills space in two dimensions; it requires three points. (A line segment of 2 points would be the smallest shape that fills space in 1 dimension, while a tetrahedron of 4 points is the smallest shape that fills space in 3 dimensions. A simplex contains $n + 1$ points, where n is the number of dimensions.

This algorithm is pretty intuitive; I like to think of it as flipping a rock down a hill. Imagine you are in the mountains and you have a triangle-shaped rock sitting on the ground. In this algorithm, you go to the point of the rock that is farthest up the hill and you flip it over, so that it flops a little bit down the hill. This is the basic mental image I use to consider how this works; the real algorithm is described below and illustrated in Figure 3.6.

Algorithm. The algorithm is described very roughly here. In your homework, you are implementing the version on wikipedia which provides more detailed instructions.

1. Make a simplex; that is, a triangle of dimension $n + 1$ where n is the number of free variables. Calculate f at each of these points.

⁸In python, the easiest way to add an explicit counter to this is with a *global* variable; you could also write your loss function as a class so that it can store variables.

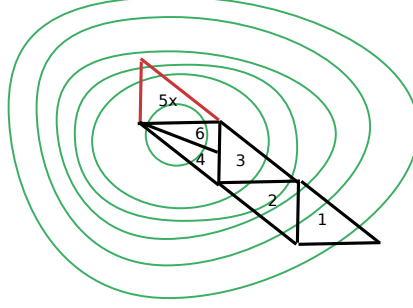


Figure 3.6: Nelder–Mead simplex descent algorithm.

2. Try “flipping it”: move the highest-valued (in f) point \underline{x}_{n+1} to

$$\underline{x}_r = \underline{x}_0 + \alpha(\underline{x}_0 - \underline{x}_{n+1})$$

where \underline{x}_0 is the centroid of the other points (the middle of the line, for a triangle) and α is a positive parameter, usually 1. Keep the flipped version if $f(\underline{x}_r)$ is less than any other $f(\underline{x}_i)$ (except $n + 1$, because then we’re just going to flip back and forth).

- Also let the simplex expand. If the reflected point is the best new point, then try

$$\underline{x}_e = \underline{x}_0 + \gamma(\underline{x}_r - \underline{x}_0)$$

3. If \underline{x}_r is still the highest energy point in the simplex after flipping, instead put it back where it was and start shrinking the simplex from this point. That is, when we get where the simplex surrounds the local minimum, we’re going to start moving in the highest energy points one by one as we hone in on the true answer. Specifically,

$$\underline{x}_c = \underline{x}_0 + \rho(\underline{x}_{n+1} - \underline{x}_0)$$

$$0 < \rho \leq 0.5.$$

4. If none of those improved it, shrink all points but the best with

$$\underline{x}_i = \underline{x}_1 + \sigma(\underline{x}_i - \underline{x}_1)$$

Topic 4

Graph theory

Reference: K23

Now we'll turn our focus to a different subject: graph theory. Graph theory is inherently the study of discrete objects, and how they are connected. The problems of graph theory are often with how to structure something (or how to interpret a structure). Some examples:

- GPS street directions: There are billions of ways to drive from here to St. Louis; how to pick something good?
- How to schedule logistics?
- How to lay out a manufacturing plant?
- What are the possible moves I can make in chess? (and the opponents' moves? and my follow-ups?)
This used to be how computerized chess worked; now they seem to be switching over to two neural networks playing each other.
- How to solve a Rubik's cube?
- How to optimally lay out a circuit?
- How to describe connectivity in a molecule?
- How to route traffic over the internet?
- How a computer can know when a piece of memory is no longer in use?
- How are bonds structured in a molecule or protein?
- How many degrees of separation are there between me and Nils Bohr?
- How do websites link to one another?
- How can phone calls be routed through switchboards?

The solution of these problems is inherently computational in nature; we generally can't write down neat mathematical forms for solutions to problems. Let's take an example; say you are playing the game tic-tac-toe; a game with just nine board places, and you want to determine the next outcome. See Figure 4.1; if we start from a certain configuration, we can draw the next configurations as shown in the figure. From each of these, we could branch out to the next possible configurations, that represent the next turn by the opposing player. In the end, we would construct a *graph* that describes all possible moves remaining in this game.

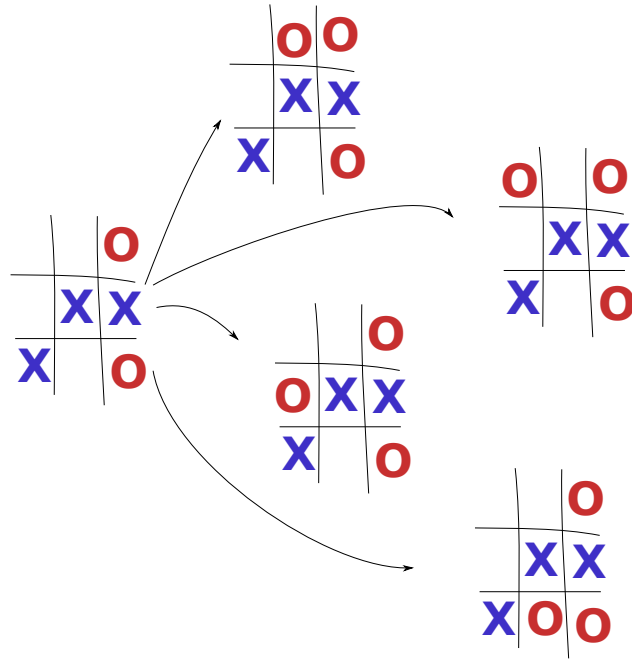


Figure 4.1: Caption.

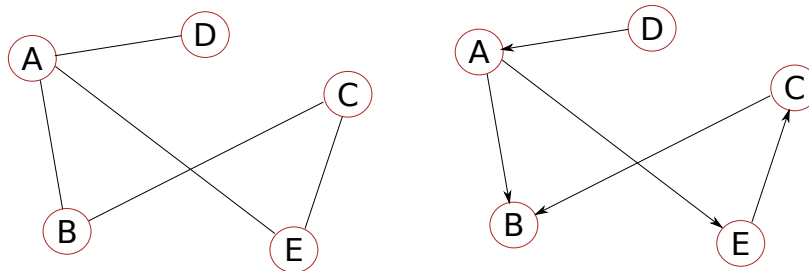


Figure 4.2: Example of graphs. The left example is an undirected graph, while the right is a directed graph.

4.1 Formalism

See the graphs drawn in Figure 4.2, which we'll use to define the terminology we'll use to describe a graph.

- *Vertex*. The points A, B, C, D, E are called vertexes; these are often also referred to as *nodes*. (In our tic-tac-toe example, each configuration of the game board is a vertex; in a molecule, each vertex corresponds to an atom (nucleus).)
- *Edge*. The connections between vertexes are called edges. That is, vertex D has one edge connecting it to vertex A. (In the tic-tac-toe example, an edge corresponds to a “move”; in a molecule, an edge corresponds to a bond.)

In Figure 4.2, we show two basic types of graphs.

- *Undirected graph*. The direction of the edges doesn't matter. A molecule would be an example of an undirected graph.
- *Directed graph*. Each edge has a direction; that is, the edges represent some transition or flow between vertexes. In our tic-tac-toe example, the edges are moves in a game, and since we can't “un-do” a move, only forward moves are allowed.

In compact nomenclature, we can represent the vertexes of a graph as a set:

$$V = \{A, B, C, D, E\}$$

Correspondingly, the edges are the set of connections in the vertexes:

$$E = \{(A, B), (A, D), (A, E), (B, C), (C, E)\}$$

If the graph is undirected, then the ordering within each pair is arbitrary (e.g., (A,B) and (B,A) represent the same thing), while if it is directed, then the ordering matters (e.g., (A,B), means there is an arrow in the direction A→B). The complete graph is then just a collection of vertexes and edges, which we can denote compactly as:

$$G = (V, E)$$

Adjacency. The book—unwisely, in my opinion—states that you should use an adjacency matrix to describe a graph on a computer. For example, the adjacency corresponding to the two graphs of Figure 4.2 are, respectively,

$$\underline{\underline{\mathbf{A}}} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad \underline{\underline{\mathbf{A}}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Note that the undirected graph has a symmetric adjacency matrix, while the directed graph does not.

However, in my opinion the use of a matrix for this purpose is a terrible idea, for the following reasons:

- Generally, most vertexes are connected to very few other vertexes. (*E.g.*, in a matrix describing how the world-wide web is connected, most individual websites do not link to one another.) Thus, this will be a huge matrix containing almost all zeros, with occasional 1's. This will take up a huge amount of memory, and computations on it will be expensive.

(Another way of saying this is that it's a sparse matrix. One solution would be to choose a sparse matrix type, as described in section §1.5.2.)

- To construct such a matrix, we need to pre-compute all entries in this matrix. That is, we need to find all the connections. This can be extraordinarily complicated.

Think of an example problem: we are going to mine openstreetmaps and use it to make our own driving directions app to compete with google maps, but we'll focus on walking directions. If we use an adjacency matrix, before we compute walking directions anywhere, we need to figure out how to identify every possible vertex and node and how they are connected in the world. This would take a long time to assemble, and the resulting matrix might be too big to put on a smartphone. But if instead, let's say we start at a vertex, it's easy to just check and see what roads come out of it to know what its connected to: that is, generate the connections on the fly.

As a second algorithm, consider building an algorithm to play chess. If we have a certain configuration on the board, I can calculate the next configuration of pieces—that is, the possible moves of my opponent—and my possible next moves, for example. However, why should I first have to worry about every other configuration of the board that I'm not likely to encounter? It's easier to compute on-the-fly.

In your homework, you compute distances on the Brown Engineering website. That is, if you start at a particular website, how many clicks does it take to get to another specified webpage? You will in this case generate it on-the-fly.

It's better to consider adjacency as a function. *E.g.*,

$$\text{Adj}(A) = B, D$$

Practically, in python, this could either be a dictionary or a function/class.

4.2 Exploring graphs / shortest-distance problem

The simplest, and perhaps most common problems in graph theory are the closely related problems of finding connections with a graph and finding the shortest distance between two vertices A and B. Here, we'll assume that all edges (connections between vertices) are of identical length; we'll relax this constraint in subsequent algorithms.

4.2.1 Breadth-first algorithm: Moore's algorithm.

Reference: K23.2

This is a brute-force algorithm, which it seems was first invented by Konrad Zuse. Zuse did other famous stuff like invent the programmable (Turing-complete) computer and the first high-level programming language. In his PhD thesis of 1945, he invented the algorithm we are about to discuss; however, he forgot to pay his enrollment fee and his PhD was rejected; thus, he didn't publish his algorithm in the open literature until decades later.

Meanwhile, Edward Moore—a Brown alum—invented the same thing about a decade later, and published it in the open literature. As the first to publish, our book names the algorithm after him. However, the algorithm is simple it has probably been invented independently thousands of times.

Here, we'll describe this while looking at an example; see Figure 4.3, focusing on the left-most drawing. Let's assume we start at the vertex labeled “start” and want to know the shortest distance to the vertex labeled “finish”, again assuming all connections have the same length. Here, we have a drawing of our graph where we can see all the connections, but we'll try to keep in mind that we want this algorithm to work where we don't know *a priori* what the connections are between any two nodes.

We start by labeling our first node with the number 0. We find its neighbors (that is we take the “adjacency” function of this vertex) and label each of them with the number 1. Then, from each node 1 we find their new neighbors, and label each of them as 2. And we continue until we find the node which we marked “finish” in our figure.

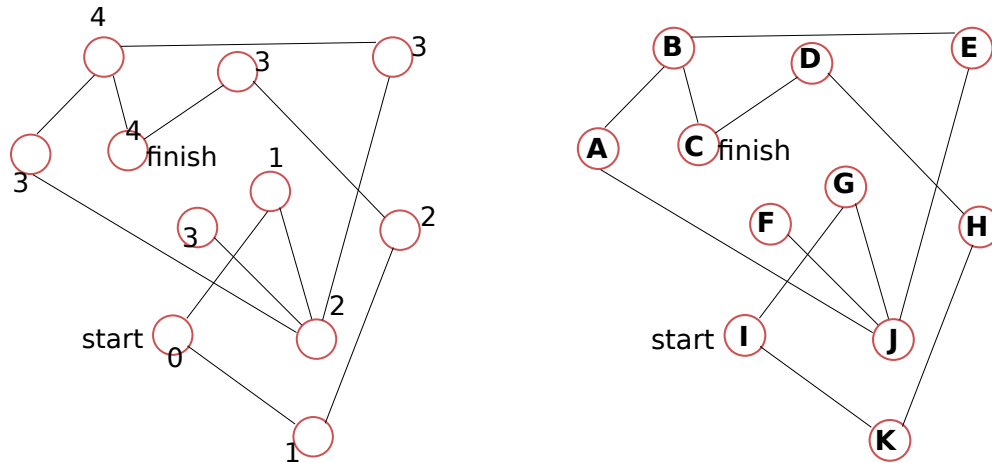


Figure 4.3: Illustration of Moore's ("breadth-first") algorithm.

Exploration vs. shortest distance. Note that this algorithm accomplishes two different things. First, it finds our shortest distance, as advertised above. But second, it also explores the graph and finds the connections; that is, if we did not know what the graph looked like we could build an adjacency table in this manner. For example, let's consider that we want to crawl through and find all the websites, so that we can build a search engine. We can start with an origin website and use all the links within this as its adjacency. The webpages these links point to are labeled 1; we do the same from each of them, labeling the new websites as 2. Thus we simultaneously explore the web (finding all connections) and build out a distance table from our original website. (We could then use the adjacency table or matrix with the PageRank algorithm of §1.4.8, for example, to rank search results.) In this incarnation, finding the shortest distance and finding the graph's connectivity are the same problem (or at least share a solution method).

How to do this on a computer? This seems easy if we can draw connections on a chalkboard. But how can we automate this for a large system where we don't have a drawing available?

Here, we'll use a combination of a queue and a step table; of course, you should modify this approach to your particular problem. In this approach, let's label the nodes as in the right-side of Figure 4.3, and in our problem we'd like to find the shortest distance from node I to node C.

We can assemble the queue and step table as follows; the results are shown in Table 4.1. We start with an empty queue and add (I,0) to it, which is our origin node; the 0 means we can reach it in zero steps from our origin node. Then we remove it from the queue and add it to our step table. When we process it, we use our adjacency function to find its neighbors, which are in this case G and K. We add each of these to the bottom of the queue, along with 1 indicating we can reach them from the origin in 1 step. We take the top element off of the queue (which now is "(G,1)") and add it to the step table, generating its new neighbor J. (J,2) is added to the bottom of the queue. We continue in this manner until we discover node C, which here we can see we can reach in 4 steps.

The key data structure here is the queue, which is just a list that we can append items to and take the top item off of.

4.2.2 Dijkstra's algorithm

Reference: K23.3

An alternative approach is Dijkstra's algorithm. This approach lends itself naturally to situations where the edges have different lengths, as in Figure 4.4. This is kind of a neat approach that relies on temporary labels which get converted one-by-one to permanent labels; it gives distances from an origin to everywhere, and the algorithm can be stopped at any point to give the maximum distance from the start to particular node.

We'll first work through the mechanics of the algorithm, then we'll come back to describe its logic. Let's work through the example in Figure 4.4 and Table 4.2, where we'll take A as the origin node. Algorithm:

Table 4.1: Moore's algorithm as we might do it on a computer, with a queue and a list of found objects. Note that items are crossed off the queue from the bottom down.

Queue	Node	Distance	New neighbors
(I, 0)	I	0	{ G, K }
(G, 1)	G	1	{ J }
(K, 1)	K	1	{ H }
(J, 2)	J	2	{ A, E, F }
(H, 2)	H	2	{ D }
(A, 3)	A	3	{ B }
(E, 3)	E	3	{ }
(F, 3)	F	3	{ }
(D, 3)	D	3	{ C }
(B, 4)	B	4	{ }
(C, 4)	C	4	{ }

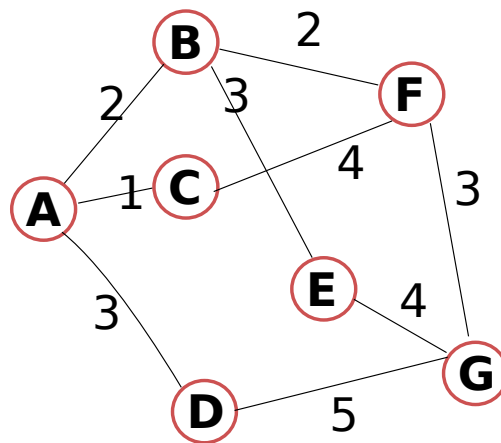


Figure 4.4: Graph for Dijkstra's algorithm.

Table 4.2: Dijkstra's algorithm.

Step	A	B	C	D	E	F	G
0	0	"2"	"1"	"3"	"∞"	"∞"	"∞"
1	0	"2"	1	"3"	"∞"	"5"	"∞"
2	0	2	1	"3"	"5"	"4"	"∞"
3	0	2	1	3	"5"	"4"	"8"
4	0	2	1	3	"5"	4	"7"
5	0	2	1	3	5	4	"7"
6	0	2	1	3	5	4	7

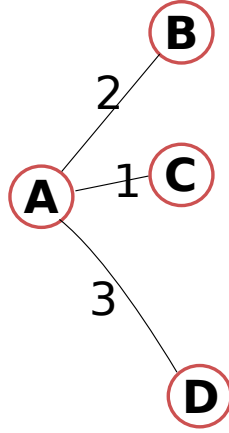


Figure 4.5: The first choices in Dijkstra’s algorithm.

Table 4.3: Caption

Step	A	B	C	D
0	0	“2”	“1”	“3”
0	0	“2”	1	“3”

1. Start with a permanent label of 0 on the starting position. In the table, we’ll denote the permanent labels with black text. Put temporary labels of l_{Ai} for all other i vertices, where l_{Ai} is the distance from node A. We’ll denote temporary labels in red text, and inside quotation marks.
2. Flip the lowest-valued temporary label into a permanent label.
3. For any neighbors of this “flipped” vertex, check the distance to the origin via the flipped vertex, and update the temporary label if the new proposed distance is lower than the old one.
4. Go to #2.

Dijkstra’s algorithm is used in driving directions algorithm; *e.g.*, we looked at the open-source code graphhopper, and found that under the hood it had a routine called `dijkstra`.

The logic of Dijkstra’s algorithm

Let’s look at this algorithm from a logical point of view, and why we can switch from temporary to permanent labels. Let’s start with just the stub of the graph shown in Figure 4.5.

I like to think of the vertices as cities, and the edges as bus fares between cities. Then we ask, what is the cheapest way to get from one city to another? Here, we’ll use the bus-fare analogy as we describe the logic. We start at city A, from where we can get to city B for \$2, C for \$1, and D for \$3; these are all the buses that leave from city A. Now, we’ll start to make a table of minimum bus fares from A, and we’ll put in placeholder values for the minimum fares to each city, as the first row in Table 4.3. That is, we know we shouldn’t pay *more* than \$3 to get from A to D.

But could we pay less? It’s possible—for example, when we examine the bus fares from city C, perhaps we’ll find there is a link from C to D for \$1, which would give us a route that costs (in total) less than the \$3 route we have found to date. This same logic applies to traveling to city B—we might find a \$0.50 fare from C to B, for example. However, for fares to city C, we can confidently say we’ll never find a route that costs less than \$1 by choosing a different route (as we would have already spent more than that to get to B or D). Therefore, we’re justified in converting the price to C into a permanent label; we can confidently say we will not get there for less than \$1, so this is the shortest distance (bus fare) from A to C. Then, we just continue this logic, and start exploring where we can go from C, adding in more temporary labels that are opened up from our knowledge of the cheapest (total) fare from A to C.

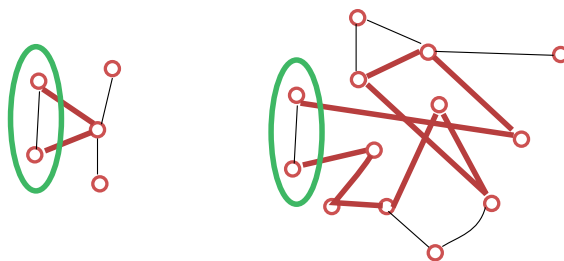


Figure 4.6: Some trees. (The heavy red lines indicate the chosen connections in the tree; the black lines are possible choices.) If the last connections (circled) are made then each tree will become a cycle.

4.3 Shortest spanning trees

Reference: K23.4–K23.5

The last prototypical graph theory problem we’ll examine is that of shortest spanning trees. This problem can be stated as: given a set of vertices and set of possible paths, what is the least-expensive way to all the vertices? *E.g.*, what is the cheapest way to get houses hooked to the sewer line? How do I route traffic over the internet? How are taxonomic trees built? The name for this problem is “shortest spanning trees”, and here we’ll define the terms in the name before describing how they can be solved.

Definition: tree. Some graphs are shown in Figure 4.6. The black lines are possible connections, while the red lines are the “chosen connections”. A tree is a connected set of edges that have no cycles. We can see that if we connect the points chosen in green, that we will no longer have a cycle.

Trees are useful for this sort of problem. In the graph shown on the left of Figure 4.6, it would be “wasteful” to add the circled edge, because we can already get from the top vertex to the bottom vertex; the cycle would be a redundant path.

Shortest spanning tree. With this definition of a tree, the shortest spanning tree is the shortest possible tree (in terms of total edge length) that can get us from any point to any other point. That is, the smallest tree that fully connects the graph.

Solution strategies. In class, we broke into groups and everyone spent 20 minutes trying to come up with strategies to solve this problem. All of the strategies presented were good basic ideas, and I was convinced that given a more reasonable amount of time (hours or days, not minutes), that we could re-invent good solution methods. Here, we’ll briefly present two solution strategies that are also contained in the textbook; it was nice to see that students came up with seeds of both approaches.

4.3.1 Kruskal’s greedy algorithm

Reference: K23.4

This approach is very simple (until we try to actually use it). It goes:

1. Order all the vertices from shortest length to longest length. (Note that the “length” is often a cost, like the cost of connecting two points with a sewer line.)
2. Starting from the cheapest (shortest length):
 - (a) Reject if it forms a cycle with the other vertices chosen.
 - (b) Otherwise accept.
3. Stop when we have $N_{\text{vertices}} - 1$ edges selected.

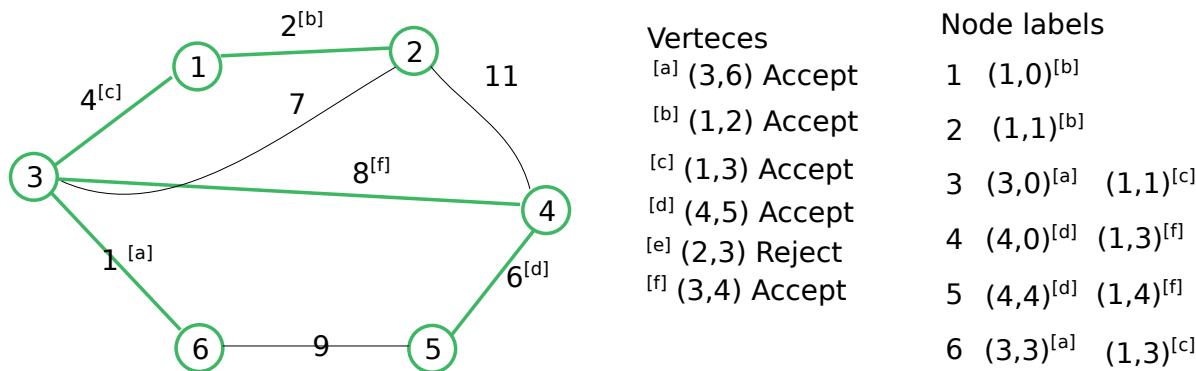


Figure 4.7: The double-label approach to Kruskal's greedy algorithm.

How do we know when we have a cycle? To use this algorithm, we need to be able to identify when we have formed a cycle. This is pretty obvious in Figure 4.6 on the left, when we only had three vertices involved. However, for the case on the right it is harder to see, and for much larger problems we certainly can't rely on a visual solution.

A scheme exists to track this, and it is the double-label scheme to attach labels to the vertices. We add to the algorithm:

- Every time we touch a vertex, add the label: (r_i, p_i) :, where the first is the root of the tree to which it belongs and the second is its predecessor.¹
- If an edge contains two vertices with the same root, reject it.

An example of this, that we went through in class is shown in Figure 4.7.

4.3.2 Prim's algorithm

Reference: K23.4

Another method is called Prim's algorithm; this is similar in spirit to Dijkstra's algorithm in which we use a table to keep track of the best connections we have found to date, and update them row-by-row. Much more elaborate descriptions are available in the text; here we give a brief summary and an example in Table 4.4 (using the same graph as the previous algorithm).

1. We choose an initial node, and make a table that contains the shortest distances we have found so far to every other node.
2. We choose the node that we can reach in the shortest distance from our current tree, and add it to our tree. We examine the new neighbors of this node, and update the table to contain the distance from the tree to these neighbors.
3. Go to #2.

4.4 Other types of graph theory problems

Most students have not dealt with graph theory before, and in this class we can only go over some basic introductory problems. Full classes are available on graph theory through mathematics and computer science departments which provide much more in-depth theory, along with rich theoretical descriptions. Here, we have only focused on the problem formulation as well as the basic logic employed. We have basically looked at three types of problems:

¹I actually think it can make more sense to instead list the children, but we'll just keep the textbook approach here.

Table 4.4: Example of Prim's method.

Step	1	2	3	4	5	6	Add vertex
0	-	2	4	∞	∞	∞	(1,2)
1	-	-	4	11	∞	∞	(1,4)
2	-	-	-	8	∞	1	(3,6)
3	-	-	-	8	9	-	(3,4)
4	-	-	-	-	6	-	(4,5)
5	-	-	-	-	-	-	

1. Exploration – finding all the connections (edges) with Moore's algorithm. If the edges are all of the same length, this also gives you the shortest distance from node A to any other node.
2. Shortest distance – if the graph is already known, the shortest distance from node A to any other point can be found with Dijkstra's algorithm. Note that this actually gives you a *tree* from the start point to all other points, where this is a shortest-distance tree from node A.
3. Shortest spanning trees – what is the minimum size graph I can create that connects all my nodes? E.g., getting all the houses in a town on the internet, but restricting the cables that I run to only run on existing streets. We looked at two different algorithms for this problem, one that focused on the vertices and one that focused on the edges.

We abstractly discussed a couple of other applications of graph theory; we only focus on the problem statement, so you can think about other applications of graph theory. Developing solution methods for all of these sorts of problems is well beyond the scope of the course.

Garbage trucks (complicated postman problem)

As I was watching garbage trucks drive around the city, it occurred to me that they must be using some fairly sophisticated graph theory algorithms in order to optimize their routes. They have a problem where they need to drive their trucks down every segment of every street in town. At it's simplest, this is called the "Chinese postman" problem² which you are doing in a be doing a couple of these on your homework. However, there are many complications, on top of the base (postman) algorithm, that they need to make sure they account for:

- They need to travel each (2-way) street twice, to empty the cans on each side. (Using a directional graph would be appropriate.)
- The trucks get full and have to go back to the transfer station to empty it out. Then they go back in to the city. Do they pick up where they left off or start somewhere else? Do they adjust their routes "on the fly" as trucks fill up, or do they have points when the trucks go to the transfer station, whether they are completely full or not?
- How many garbage trucks should be out running at the same time? How should the work be divided up?
- What about traffic? What time should (each) truck start its route? When should they avoid certain streets?
- Right turns are faster and safer than left turns. Should we penalize left turns in our algorithm?

So we can see that it will be pretty involved to come up with an approach that optimizes the route—but whatever trash company can come up with the best algorithm has the best chance of winning the contract.

²In the "Chinese postman" problem, the goal is to traverse every edge, whereas, in the "traveling salesman" problem the goal is to visit every node.

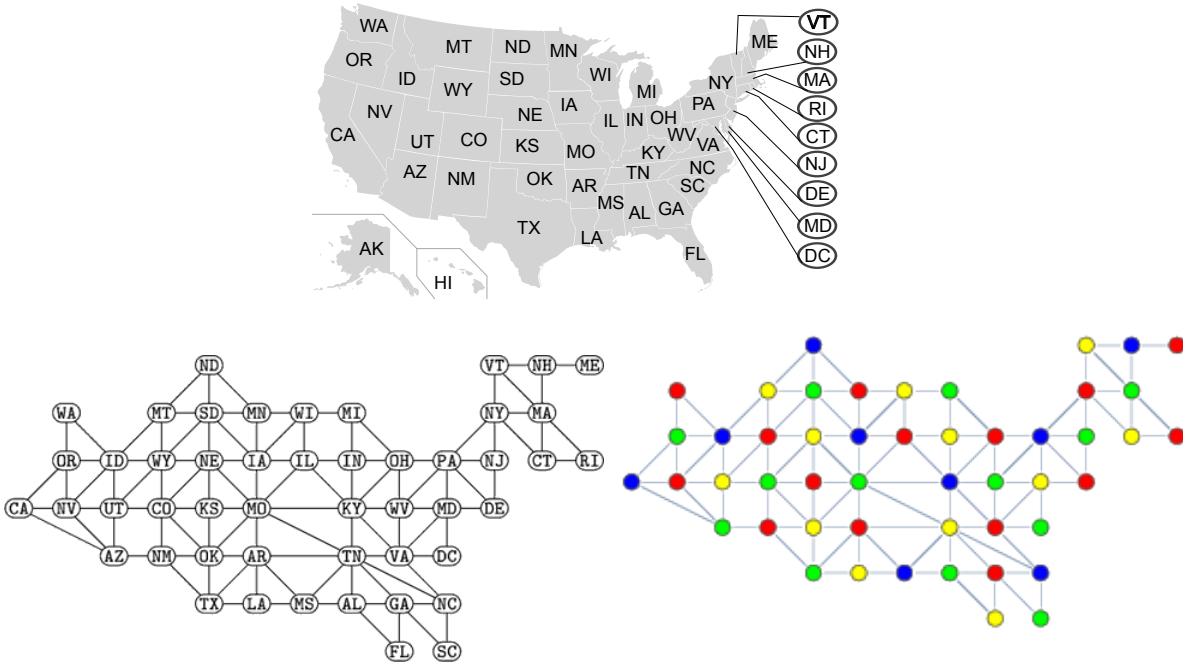


Figure 4.8: Map coloring of the United States. Map from https://upload.wikimedia.org/wikipedia/commons/6/6b/Labelled_US_map.svg. Graphs from <http://mathworld.wolfram.com/ContiguousUSAGraph.html>.

One of the complications of this problem is that there is a continuous variable—the time of day that they start—added in, so it's not a purely graph theory problem but is a mixture of types of problems. As we'll briefly discuss in [\[link to come...\]](#), this makes this a very challenging problem.

Map coloring (graph coloring)

Another famous problem of graph theory is motivated by drawing maps: if I want to color each state / country on my map with a color distinct from its neighbors, how many colors do I need and how do I arrange them? For a two-dimensional map, this has been proven to be capable of being solved with a palette of four colors.

An illustration of this for the states of the United States is shown in Figure 4.8. This type of problem is more challenging for more dimensions, and this type of algorithm is useful in other types of applications such as scheduling.

Topic 5

Mixed optimization

The previous two chapters covered simple optimization of continuous functions (§3) and graph-theory optimization (§4). Here, we'll briefly go over some last topics on optimization that apply to both topics.

5.1 Pareto optimality

Recall that the most important qualitative point of optimization is that you have to pick which *one* attribute of your system you would like to optimize (§3.1). However, if you are trying to understand the *trade-off* between various attributes, then people speak of Pareto optimal solutions.

A common trade-off in engineering design is capital cost versus operating cost—for example, how much it costs to build a power plant versus the cost of electricity produced by that power plant. See Figure 5.1. You come up with “generation #1” of your power plant, which has a certain level of capital and operating costs. Then, you find great ways to improve on it that lower *both* the operating cost and the capital cost. Great. You keep iterating and coming up with better and better solutions—but eventually you find that you can't lower the capital cost without increasing the operating cost, and vice versa. This is the condition of Pareto optimality.

This is shown as the Pareto-optimal solution in Figure 5.1. You can interpret any point along this curve in either of two ways: (1) for a given capital cost, if I am on the Pareto optimal curve I cannot further lower my operating cost, or (2) for a given operating cost, on the Pareto optimal curve I cannot further lower my capital cost.

This is an important point in real optimization schemes. When you care about multiple attributes, your ultimate design should be on the Pareto optimal line; that is, if you are any where else, you could be doing

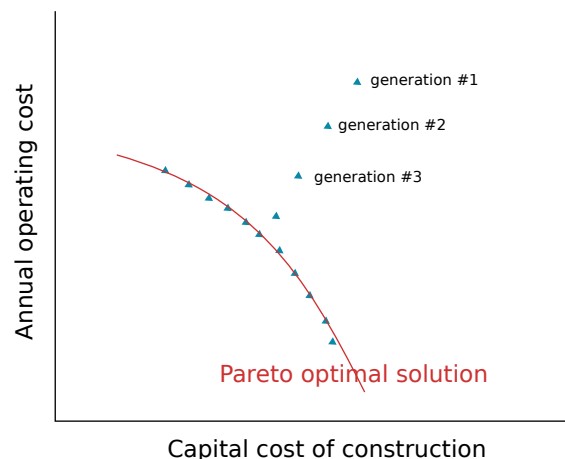


Figure 5.1: Understanding trade-offs in the design of an engineering facility.

better, even taking into account trade-offs. Once you are on the Pareto optimal curve, then it's a matter of judgement: how much do you prize one thing versus another?

You can imagine this is very useful in a business setting—if you can clearly lay out the trade-offs to your management, who will ultimately need to make the difficult decisions, then they have all the information they need to make informed decisions.

At its most basic, you can construct a Pareto optimality curve by just optimizing under constraints (§3.5). That is, if you are analyzing the trade-off between attributes A and B, optimize B under the constraint that A is fixed, and repeat this for various (fixed) values of A. This will give you the Pareto-optimal curve, which you can interpret in either way. Of course, the above approach can work only for the optimization of simple functions; we leave the optimization of more complex functions to their respective specialized fields. Of course, there are many advanced and sophisticated algorithms for understanding Pareto optimality.

5.2 Mixed problems

We have kept our earlier discussion to two discrete topics of optimization: continuous function optimization *vs.* graph theory. Of course, most challenging optimization problems involve some mixture of continuous and discrete variables. Solving these problems, in general, is hard; in this section, we'll give some basic concepts but will note that there are fields of specialty that deal with just these sorts of applications.

At its simplest, let's ignore the complexity of graph theory and just assume that we can pose our optimization problems involving continuous variables (*e.g.*, temperature, flow rate, ...) and integers (*e.g.*, in graph theory there is a 1 if the connection is used and a 0 otherwise). For example, if you are optimizing a process you might have to choose between two pieces of equipment, and the process must be optimized with each choice. And there may be thousands of pieces of equipment to choose from.

In general, this is mixed-integer non-linear programming. This is the hardest of optimization problems, but actually is real life. There typically is not a easy way to solve these, and this is generally the reason that engineers are hired. There exist solution methods if the problem can be well-posed, but they are computationally challenging and can fall into local minima. We won't cover that in this course; there are courses on this topic, typically in systems engineering curricula. Here, we'll just qualitatively cover some basic approaches.

5.2.1 Simple problems: combinatorics

Let's imagine that we have a system that has a mixture of continuous variables (that can take on any floating-point value) and binary variables (that can only be 0 or 1). For example, if we are optimizing a process, we may have a binary value that is 1 if there is a pump, and 0 if there is not a pump in our process.

If we only had one binary variable, the simplest solution would be to run the optimization twice: once with the variable set at 0, and once with it set at 1. Then we simply choose the better of the two answers; we use non-linear optimization twice.

This is straightforward if we only have a few binary variables. For example, if we had two binary variables, then we need to run our optimization under four scenarios: [(0,0), (0,1), (1,0), (1,1)].

However, we can see that this is a combinatoric problem, and the number of scenarios we need to run is 2^n if there are n binary variables. You might be able to pull this off with 10 binary variables,¹ but I wouldn't try it with 100.²

5.2.2 Treating as continuous; adding penalties

Another strategy is to let our integer variables be treated as continuous variables³ during the optimization. *I.e.*, we might allow a binary variable to take on any continuous value in the closed interval [0,1]. We check our answer at the end of the routine, and if the optimum solution has chosen exactly 0 or 1, we are in good

¹ $2^{10} = 1024$.

² $2^{100} = 1267650600228229401496703205376$.

³Of course, this isn't always possible. For example, if your binary variable indicates if there is or is not a pump, we really don't have a way to write a model with half a pump. But you'd be surprised what people can pull off: I have seen people define fake chemical elements with atomic numbers between those of the real elements on the periodic table!

shape. If not, we impose a penalty in our loss function for non-integer values using the techniques of §3.5.2; if the constraint is large enough it will force our solution to choose an integer value, at least to some arbitrary precision. (This is similar to what you did in your homework where you had a penalty to force the points to lie on a parabola.) Of course, how and when you implement this penalty will certainly make a difference in how you get to the ultimate solution.

5.2.3 Branch and bound

The branch-and-bound technique is systematic approach for dealing with integer variables in an otherwise continuous landscape; it is particularly useful if the integer variables can take on a wide range of values (*i.e.*, they are not just binary), and if the problem can be expressed as a purely linear problem, so that you will find the global minimum easily (when the variables are continuous). In this case, it will find the global optimum solution in a finite amount of time.

You can readily find examples of the implementation of this method online, which illustrate it nicely. Briefly, the discrete variables are “relaxed” into continuous variables and the problem is optimized. This provides an initial lower bound⁴ for the optimal solution. For discrete variables whose optimal solution comes out to be a non-integer, the solution tree is branched. For example, if the variable x_1 is only allowed to take on integer values and upon optimization, $x_1 = 7.4$, then we “branch” the solution into two problems: one with $x_1 \leq 7$ and one with $x_1 \geq 8$. We then perform constrained optimization on each of these branches; and we now have bounds on each of these branches. Ultimately, we are able to compare and the best-case scenario solutions to each of these branches, and eliminate solution branches until we find the optimal case.

5.2.4 Mixed-integer non-linear programming

If we are lucky enough that our optimization function can be written in a linear form, even if we have a mixture of linear and continuous variables, then we have ready solution approaches available, such as branch and bound. However, for the general case where we have a mixture of both types of variables and our functional form is the general non-linear case—most cases in engineering design!—then we have mixed-integer non-linear programming, which is generally considered the hardest class of optimization problems. There are typically not easy “off-the-shelf” methods to deal with these, and the optimal method depends on the physical problem being solved. There are many specialized research areas that examine problems in various disciplines. (Perhaps we wouldn’t have engineers if there was a simple, general solution to this problem.)

5.3 Global optimization

A second key point raised when we first introduced optimization in §3.1 is: finding a local optimum is usually rather straightforward, but finding the global optimum solution is in general one of the hardest problems in engineering. In general, there are not easy (that is, computationally feasible) solutions to this problem, even for the case when all of your free variables are continuous.

The approach you take will depend in large part on what your problem is. For example, I showed a paper that we published [6] that uses a technique called minima hopping. In this approach, atomic structures are launched out of local minima (by numerically integrating their equations of motion), and relaxed from new, random configurations; this also includes self-adjust parameters that adjust the momentum at which they are launched as well as the probability of accepting newfound minima. We also included custom constraints that act to preserve molecular identity throughout this process.

Perhaps the most famous global optimization approach is simulated annealing. This is based on the physical process of annealing—where a material (*e.g.*, steel) is very slowly cooled from a high temperature. In materials, this helps the structure form a “perfect” crystal, as it is slowly lowered in energy. In simulated annealing, we think of the loss function as the energy and we “cool” the system. That is, we initially assume it has some temperature so that it can sample states, then we slowly lower that temperature so that it just samples the lowest-energy state.

⁴Assuming we are performing a minimization; that is, a best-case scenario.

Practically, we assume we are at some state s , and we propose a move to s' . We accept the move with a probability like

$$P \sim \exp \left\{ \frac{-(E(s') - E(s))}{T} \right\}$$

For those of you with a statistical mechanics background, this should be very familiar. This is the probability given to us by Boltzmann,⁵ which is the basis of much behavior predicted by statistical mechanics.

⁵In the true Boltzmann factor, the denominator is $k_{\text{B}}T$, where k_{B} is the Boltzmann constant.

Topic 6

Integration of differential equations

Reference: K21, B4, B6

6.1 Models

Most of the models we encounter in engineering and physics are written in terms of differential equations; for example:

- Newton’s second law [mechanics]

$$F = m \frac{d^2 x}{dt^2}$$

- Navier–Stokes (simplified) [fluids]

$$\rho \frac{D\mathbf{v}}{Dt} = \rho \mathbf{g} - \nabla P + \mu \nabla^2 \mathbf{v}$$

- Fourier’s law [heat transfer]

$$q_x = -k \frac{dT}{dx}$$

- Fick’s law [mass transfer]

$$j_x = -D \frac{dC}{dx}$$

- Faradays’s law [electromagnetism]

$$\mathcal{E} = - \frac{d\Phi_B}{dt}$$

- Poisson equation [electrostatics]

$$\nabla^2 \phi = - \frac{\rho}{\epsilon}$$

- Fundamental equation [thermodynamics]

$$dU = T dS - p dV$$

Often, when we solve these problems in our undergraduate courses we do so for simple, idealized systems where we can find analytical, closed-form solutions. However, most real-world applications, including those encountered in research and in industry, are much more complicated, and solution “by hand” is not possible or practical. In this section, we’ll discuss the common approaches to solving such differential equations numerically. We’ll typically assume that you have learned to write the governing equations of your system from your undergraduate classes, and we’ll focus mostly on solution methods, not derivations, in this class. However, to establish good habits, we will derive the equations for a few model systems that we discuss.



Figure 6.1: A block, cooling.

6.2 Initial value problems

Reference: B4,K21.1–K21.3

6.2.1 Example: Heat loss

I think that engineers and physicists in all fields, at some point, have to worry about heat loss or gain, and we'll use this as a simple, motivating example.

Case I: A cooling block

Here, we'll start with a simple problem of a block of metal, that is initially hot at some uniform internal temperature T_0 and cools by losing heat to its environment, which is at T^∞ . We'll assume that at any time t , the temperature inside the block is uniform at T ; that is, it does not vary spatially within the metal.

We can derive the governing equations just for accounting for the energy flows:

$$(\text{Accum}) = (\text{In}) - (\text{Out}) + (\text{Gen}) - (\text{Cons})$$

The term on the left describes the decrease in energy within the block, where m is the mass of the block and C_P is its heat capacity. On the right, we are assuming that heat flows out and is governed by $hA, \Delta T$, where h is a heat-transfer coefficient and A is the heat-transfer area.

$$\frac{d}{dt}(mC_P T) = 0 - hA(T - T^\infty)$$

Note that we assumed that the block is cooling in this derivation ($T > T^\infty$), but if instead it is warming ($T < T^\infty$) then the term on the right-hand side will be positive, and this equation will correctly predict that the block warms. Thus, the equation is general to both cases.

If the heat capacity, C_P , is not a function of T over the temperature range of concern, then we can pull it and the mass out of the derivative and get:

$$mC_P \frac{dT}{dt} = -hA(T - T^\infty)$$

which is our governing equation, along with the initial condition

$$T|_{t=0} = T_0$$

Non-dimensionalize. I strongly recommend nondimensionalizing your equations as your first step, before attempting to solve them. This has a number of advantages:

1. *It can help you to recover more insight into the problem.* A common criticism of numerical methods is that you do not gain the “insight” into your system’s behavior that an analytical procedure provides; the non-dimensional behavior can help you recover some of that insight.
2. *It can alert you to when it’s valid to simplify your equations.* By examining which terms might be very small or very large, you can decide if you need to keep all aspects of your original equations.
3. *It reveals which parameters are truly independent, by naturally lumping together variables.* In this way, we reveal the “universal” behavior of related systems.

4. *It can make problems numerically better behaved.* Numerical integrators typically expect your variables to be of order 1.
5. *It can sometimes make the math easier.* The substitutions will sometimes simplify your final form, as we'll see here.

We'll demonstrate the non-dimensionalization procedure on this example. The general procedure is to scale all of your variables (not parameters), preferably so that they are of order 1. Here, let's start by scaling the temperature. I'll propose that introduce θ , which will be our non-dimensional temperature, and define it as

$$\theta \equiv \frac{T - T^\infty}{T_0 - T^\infty}$$

Here, we can see that $\theta = 1$ when the temperature is at the initial condition (T_0), and $\theta = 0$ when the temperature has reached the ambient condition (T^∞).

In our governing equation, we need to substitute not just T , but also dT/dt , so let's find that in terms of θ :

$$\frac{d\theta}{dt} = \frac{1}{T_0 - T^\infty} \frac{dT}{dt}$$

Now, we can just substitute these definitions into our governing equation, and we get:

$$mC_P(T_0 - T^\infty) \frac{d\theta}{dt} = -hA(T_0 - T^\infty)\theta$$

$$mC_P \frac{d\theta}{dt} = -hA\theta$$

Next, we need to figure out the proper way to scale t . Note that if we isolate θ (by dividing through by hA)

$$\frac{mC_P}{hA} \frac{d\theta}{dt} = -\theta$$

Now, since the term θ has no units, then the term on the left-hand side also has no units. This suggests a natural time scale for our problem; that is, we can define our non-dimensional time τ as

$$\tau \equiv \frac{t}{\frac{mC_P}{hA}} = \frac{hA}{mC_P} t$$

Now plug in $t = (mC_P/hA)\tau$ into the equation.

$$\boxed{\frac{d\theta}{d\tau} = -\theta}$$

We should also transform our initial condition, which just becomes

$$\boxed{\theta|_{\tau=0} = 1}$$

Notice how simple our equation became. Interestingly, we went from having many user-definable parameters (T_0 , T^∞ , m , C_P , h , A) to having none! Thus, we have revealed a “universal” behavior for this problem. We can further note the true independent effect of different parameters—*i.e.*, by examining which parameters are “lumped”.

This equation is of course trivial to integrate by hand—and it actually got slightly easier during our non-dimensionalization process. Here, we'll solve it by hand:

$$\int_1^\theta \frac{d\theta}{\theta} = - \int_0^\tau \tau$$

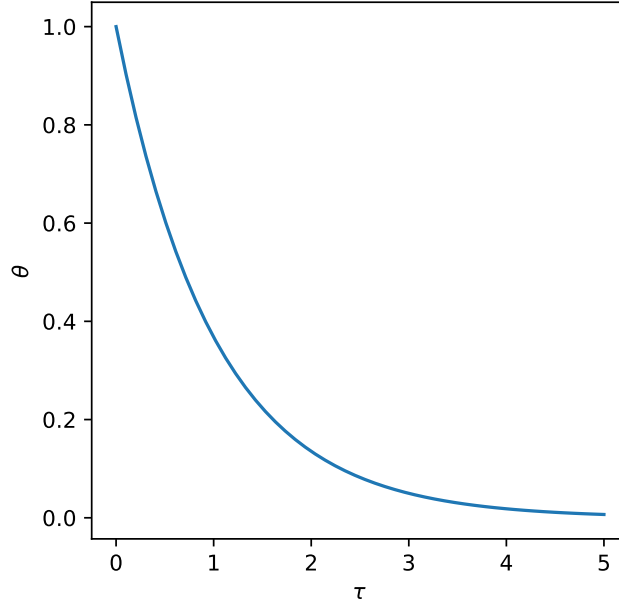


Figure 6.2: Cooling of a block.

$$\ln \theta = -\tau$$

$$\theta = e^{-\tau}$$

We've plotted this function in Figure 6.2, which is just an exponential decay. However, what is neat is this is now a universal plot—it is valid for any mass, heat capacity, ambient temperature, etc., as long as the original assumptions of our physical derivation are valid.

Case II: Add a source term

Now let's imagine that there's a source term within our black box. (Maybe our black box is a power adapter for a computer, and the computer is cycling on and off. Maybe our black box is a chemical reactor and the chemicals react differently at different times. Maybe our black box is a solar panel and the sun varies in intensity throughout the day and also goes behind the clouds now and then.) In this case, we would derive an equation that instead looks like:

$$\frac{d}{dt}(mC_P T) = 0 - hA(T - T^\infty) + g(t)$$

where $g(t)$ might be any of those terms above. For the sake of simple mathematics¹, let's say that the source term is sinusoidal

$$mC_P \frac{dT}{dt} = -hA(T - T^\infty) + a \cdot (1 + \sin(bt))$$

Non-dimensionalize. We'll again follow a similar non-dimensionalization procedure, starting with the same definition of θ and τ :

$$mC_P(T_0 - T^\infty) \frac{d\theta}{dt} = -hA(T_0 - T^\infty)\theta + a \cdot (1 + \sin(bt))$$

¹Note that when we solve things numerically, we can put very complex forms in here and it's not really any harder to solve.

$$\frac{mC_p}{hA} \frac{d\theta}{dt} = -\theta + \frac{a}{hA(T_0 - T^\infty)} (1 + \sin(bt))$$

$$\frac{d\theta}{d\tau} = -\theta + \frac{a}{hA(T_0 - T^\infty)} \left[1 + \sin\left(\frac{bmC_p}{hA}\tau\right) \right]$$

In this case, we can see that we have parameters left in our equation after we have non-dimensionalized both of our variables (T and t); therefore, these are the true (lumped) parameters that are left in our equation. If we define that first group of constants as α and the second as β , we get

$$\boxed{\frac{d\theta}{d\tau} = -\theta + \alpha \cdot (1 + \sin \beta \tau)}$$

We haven't solved our equation, but we have actually gotten it into a form where we can understand its behavior reasonably well.

- What if α is tiny? Specifically, since we've non-dimensionalized θ to be of order 1, we even know what is meant by tiny: $\alpha \ll 1$. Then we can pretty safely ignore that term, and we can integrate by hand; we'll just recover the same behavior we had from the previous derivation with no source term. Conversely, if α is giant, then it will dominate the response and we can ignore the first term, at least for small times.² When $\alpha \sim 1$, we can expect the heat addition and removal terms to be roughly balanced, and our temperature will hold somewhat steady (but oscillating, of course).
- Thus, the combination of variables:

$$\frac{a}{hA(T_0 - T^\infty)}$$

is a *dimensionless* number that tells us the relative importance of the source term to the heat transfer. Note that dimensionless numbers (*e.g.*, Reynolds number, Biot number, ...) are used extensively in engineering, and they are properly defined directly from problem statements in this manner.

- Similarly, β is a non-dimensional parameter that tells us about the frequency of the oscillation, relative to the materials ability to respond to that oscillation.

Plot. I've plotted the behavior of this system for various choices of α and β in Figure 6.3, using a numerical integrator.³ What's nice about our non-dimensionalization procedure is we can be confident that, just by examining the influence of α and β on our system's behavior, that we are capturing all classes of behavior. We know there is no benefit to adjusting all the individual parameters (T_0 , T^∞ , m , C_p , h , A , a , b) one-by-one, as their influence is combined. Thus, we can still gain a strong insight into the behavior of our system, even though we weren't able to get out an analytical function $y(t)$.

6.2.2 General form and numeric approach

An initial value problem—like the examples we just saw above—can be written generically as below.

$$\frac{dy}{dt} = f(y, t) \tag{6.1}$$

When we integrate, a constant of integration needs to be satisfied; this is taken care of by the initial condition.

$$y \Big|_{t=0} = y_0 \tag{6.2}$$

²This would actually be a strange situation, and there are two things to note here: (1) If we integrate an equation that has a source but no sink the temperature will grow and grow without bounds. *Eventually* the θ term will grow large enough that our block starts shedding heat (or it melts or explodes). (2) We could actually catch this fact from the numerics: the derivative $d\theta/d\tau$ was scaled to be of order 1, so if $\alpha \gg 1$, and it equals the derivative, then $d\theta/d\tau$ is a lot larger than we expected it to be, and perhaps we didn't scale one of our variables correctly.

³Specifically, `scipy.integrate.odeint`; you can see the source code in the repository for this book if you dig for it!

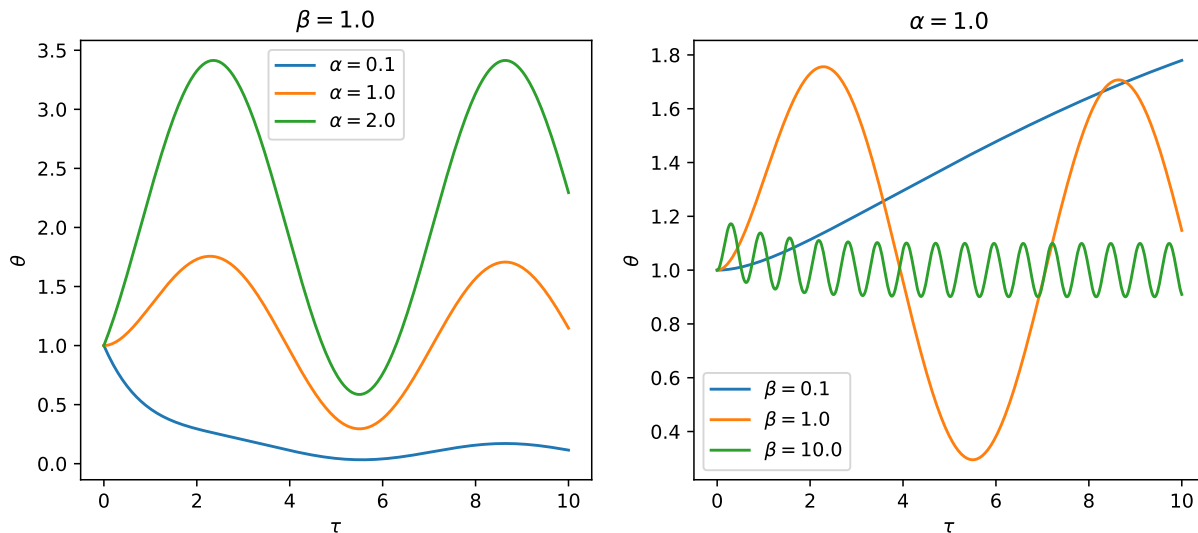


Figure 6.3: Cooling of a block with a sinusoidal source term.

The above two equations are the general form for a single, ODE, initial problem; we would like to solve for the value of y as a function of t . To be clear, $f(y, t)$ is a known algebraic expression, and y_0 is a known number. The independent variable is conventionally written as t because time is a very common independent variable in initial value problems, but any equation that can be written in this form can be integrated with the methods we will discuss.

Solution approach. The general solution approach is to start with the known value of y at $t = 0$, then to take a step forward in time to estimate y at $t = h$. We continue taking steps forward in time in this manner, with a general step being written as:

$$y|_{t+h} = y|_t + m h \quad (6.3)$$

where m is our estimate of the slope and h is our time step. In general, we re-estimate m at each time step and choose an optimal h ; however, for right now we'll assume our time step is constant across the entire integration.

Numerical integration is a very common, standardized method, and for most basic problems it makes sense to use pre-packaged integrator, such as `scipy.integrate.odeint`. In this section, we'll focus on how the basic integration approaches work, as well as how your problem should be formulated and how to properly choose the right integration technique. However, coding these methods up by hand is not challenging, and can even be done using cell operations in a spreadsheet, if you ever find yourself in an environment where you want to share a model with colleagues, this can be very useful.

In the following sections, we'll first focus on how m is estimated at each step; later, we'll briefly describe how the time steps h can be chosen to balance accuracy and speed.

6.2.3 Explicit Euler

Reference: K21.1, B4:177

The explicit Euler method is a simple, intuitive method which we'll use to estimate the slope, m , from equation (6.3)—it just extrapolates from the current value to predict the next value, and repeats this over consecutive time steps. Note that this method is really just useful for learning numerical integration; the Runge-Kutta method (§6.2.5) provides a much better estimate and is nearly as simple to implement. Here, we'll provide two equivalent derivations for this method:

Derivation #1. We start from the definition of a derivative:

$$\left. \frac{dy}{dt} \right|_t \equiv \lim_{h \rightarrow 0} \frac{y(t+h) - y(t)}{\Delta t}$$

For “small enough” values of h , we can approximate by dropping the \lim and re-arranging:

$$y(t+h) = y(t) + \underbrace{\left. \frac{dy}{dt} \right|_t}_m h$$

Derivation #2. We can use a Taylor Series expansion to approximate the value of y at $t+h$ about the value of $y(t)$, which we truncate after the first term.

$$y(t+h) = y(t) + \left. \frac{dy}{dt} \right|_t h + \dots$$

Thus, in both of the above we have that our slope estimate is just dy/dt evaluated at time t , which is the *start* of the interval that defines the step. This means we just plug the current values of y and t into $f(y, t)$ (equation (6.1)) to generate m . Also, from either derivation we note that the slope estimate is increasingly accurate as h gets smaller.

Example. Here, let’s integrate the following initial value problem, using a time step of $h = 0.1$:

$$\frac{dy}{dt} = ty + 1, \quad y|_{t=0} = 1$$

The slope m for the first time step can be found by plugging in $t = 0$ and $y|_{t=0} = 1$ to $ty + 1$, giving 1, and the first time step is

$$\begin{aligned} y(0.1) &= y(0) + m_0 h \\ &= 1 + (1)(0.1) = 1.1 \end{aligned}$$

This gives us our next point (0.1, 1.1); that is, $y_{t=0.1} = 1.1$. Our next slope is $ty + 1 = (0.1)(1.1) + 1 = 1.11$, so we can calculate the next step as:

$$y(0.2) = 1.1 + (1.11)(0.1) = 1.211$$

We continue on in this fashion indefinitely.⁴ For very small time steps this can work reasonably well, but we note that small errors in the slope estimate can propagate over time.

How to code? This approach is very easy to implement.⁵ We show an example implementation in python below.

```
def get_yprime(y, t):
    """This is our function to integrate."""
    return t * y + 1.

def step(last_y, t):
    """Takes a single Explicit Euler step."""
    yprime = get_yprime(last_y, t)
    return last_y + yprime * dt
```

⁴Typically we’ll integrate until some desired stopping time, t_{final} , or by observing observing the behavior of the function $y(t)$; for example, in the block cooling examples shown above, we might integrate until we see y level off.

⁵Again, don’t do this for real problems! If you need to code your own use something more accurate like Runge–Kutta. And if you don’t need to code your own, use a pre-packaged integrator like `scipy.integrate.odeint`.

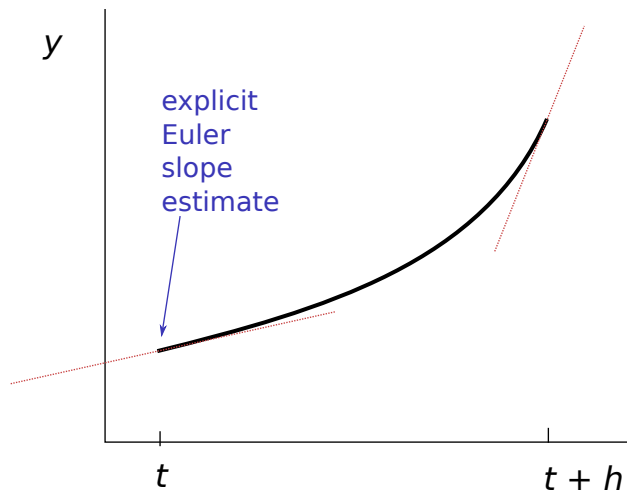


Figure 6.4: The slope estimate of the Explicit Euler method can carry systematic errors since it is always from the beginning of the interval $(t, t + h)$.

```
ys = [1.] # Results list, with initial value in first position
times = np.arange(0., 0.5, 0.01)

for time in times[1:]:
    y = step(ys[-1], time)
    ys.append(y)
```

Problems? Let's look at the plot of Figure 6.4. We would like a good estimate of the slope over the interval $(t, t + h)$, but the Explicit Euler always gives the slope right at the opening of the interval. In the example problem we just examined, we know that there will always be upward curvature, as shown in the figure. This means that (for upward curvature) we will always under-estimate the true slope, and thus under-estimate our true curve $y(t)$. This is a *systematic error*, and we can expect the error to compound over the course of an integration. (As opposed to a random error, where we might expect more error cancellation.) The only way to improve the accuracy of the Explicit Euler method is to decrease the step size h , which will improve the slope estimate. This is one way we can examine the accuracy of our implementation: try smaller and smaller time steps until we see that the answer “converges”. (As we've seen before, there is of course a limit to how low we can go, when we ultimately hit precision issues on the computer.) However, rather than decrease the step size, a better approach is to improve the slope estimate to include information from throughout the interval, rather than just from its beginning. This is the basis of the improved methods we will discuss in the following sections.

6.2.4 Predictor–corrector method (Improved Euler)

Reference: K21.1

A problem with the explicit Euler method is that the slope is only estimated at the beginning of the interval, but we know that (for curved functions) the slope changes continuously throughout the interval. It would be better if we could estimate an average slope over the interval, which we might be able to do by averaging the end values. However, to calculate $dy/dt = f(y, t)$ at any point, we need the value of both y and t . We only know y at the beginning of the interval—that is, to find dy/dt at $t + h$ we need to know $y(t + h)$, which was our objective in the first place.⁶

⁶This conundrum is resolved elegantly, with algebra, when we discuss implicit methods in §6.2.9.

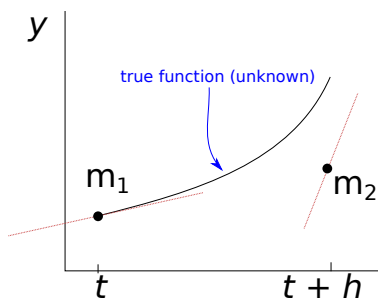


Figure 6.5: The Improved Euler (predictor–corrector) method.

The method we discuss here resolves this as follows. First, we get an estimate of the slope at the beginning of the interval; this is the same slope estimate as the one used in the Explicit Euler approach:

$$m_1 = \left. \frac{dy}{dt} \right|_{t, y_t}$$

Now, we’d like to get a second slope estimate at $t + h$, and we need a value of y for this. In the Improved Euler method, we just use the estimate provided by m_1 —that is, what would have been the Explicit Euler prediction—and use $y|_t + m_1 h$. This won’t be perfect, but it will be better.

$$m_2 = \left. \frac{dy}{dt} \right|_{t+h, y_t + m_1 h}$$

This is shown in Figure 6.5. Then the slope that we will use for the time step, equation (6.3), is

$$m = \frac{m_1 + m_2}{2}$$

Example. Using the same problem from the previous section, we show in the table how a couple of steps propagate using a time step of $h = 0.1$.

t	y_t	m_1	$y_t + m_1 h$	m_2	m	y_{t+h}
0	1	1	1.1	1.11	1.055	1.1055
0.1	1.1055	1.11055	1.216555	1.243311	1.1769305	1.22319305

6.2.5 Runge–Kutta

Reference: K21.1, B4:177

One of the traditional workhorses of solving differential equations, that gives a good blend of accuracy and speed, is the so-called Runge–Kutta algorithm. This is also a simple algorithm that can be easily coded by hand when a numerical solver is not available—for example, if you needed to write an excel macro to solve an ODE; you can even code this up within the cells of a spreadsheet. There are many varieties of this, but the classic method is known as “Fourth-order Runge–Kutta” (RK4) and blends four estimates of the slope in order to make its time step. This is in spirit very similar to the previous method (§6.2.4), but with a more refined estimate. The four estimates of the slope are listed below, and build on each other sequentially. These are also illustrated in Figure 6.6.

$$m_1 = \left. \frac{dy}{dt} \right|_{t, y_t}$$

$$m_2 = \left. \frac{dy}{dt} \right|_{t + \frac{1}{2}h, y_t + \frac{h}{2}m_1}$$

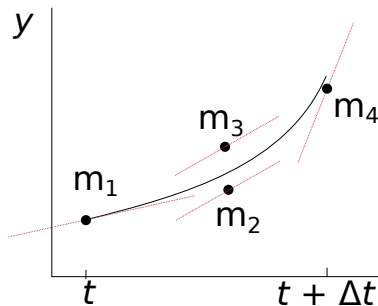


Figure 6.6: The Runge-Kutta slope estimates.

$$m_3 = \left. \frac{dy}{dt} \right|_{t+\frac{1}{2}h, y_t+\frac{h}{2}m_2}$$

$$m_4 = \left. \frac{dy}{dt} \right|_{t+h, y_t+hm_3}$$

The pooled estimate of the slope is a weighted average of the four slope estimates, with more weights given to the middle values:

$$m = \frac{m_1 + 2m_2 + 2m_3 + m_4}{6}$$

Finally, the step is taken as before:

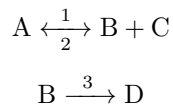
$$y|_{t+h} = y|_t + m h$$

6.2.6 Systems of ODEs

Next, we'll describe how to treat systems of ordinary differential equation problems, which if they can be set up correctly are treated nearly identically to single equations. We'll start by giving a couple of examples of derivations where systems arise.

Reactor

Consider the design of a chemical reactor to produce chemical “D” from chemical “A”, in which the following reactions take place:



A double arrow indicates the reaction can proceed either forwards or backwards, while a single arrow indicates only the forward reaction is allowed. The reaction rates r_i [mol L⁻¹ s⁻¹] are related to the concentrations C_j [mol L⁻¹] by rate constants k_i , whose units depend on the reaction order, and are given by:

$$\begin{aligned} r_1 &= k_1 C_A & k_{1f} &= 3 \text{ s}^{-1} \\ r_2 &= k_2 C_B C_C & k_{1r} &= 0.2 \text{ L mol}^{-1} \text{ s}^{-1} \\ r_3 &= k_3 C_B & k_2 &= 0.1 \text{ s}^{-1} \end{aligned} \tag{6.4}$$

We can write a “mole balance” on any species to determine its behavior; this gives:

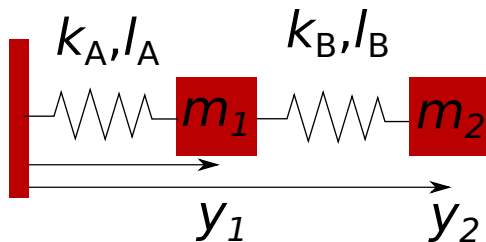


Figure 6.7: A combined system of springs.

$$\begin{aligned}\frac{dC_A}{dt} &= -k_1 C_A + k_2 C_B C_C \\ \frac{dC_B}{dt} &= k_1 C_A - k_2 C_B C_C - k_3 C_C \\ \frac{dC_C}{dt} &= k_1 C_A - k_2 C_B C_C \\ \frac{dC_D}{dt} &= k_3 C_B\end{aligned}$$

Now we have four first-order differential equations; when we integrate we will get four constants of integration, which we can satisfy by stating the initial concentration of each chemical. For example, assume that $C_{A0} = 1 \text{ mol L}^{-1}$ and the remaining inlet concentrations are zero.

We put this in standard form by defining the set of concentrations as the vector $\underline{\mathbf{y}}$:

$$\begin{aligned}\underline{\mathbf{y}} &\equiv [C_A \quad C_B \quad C_C \quad C_D]^T \\ \underline{\mathbf{y}}_0 &\equiv [C_{A0} \quad C_{B0} \quad C_{C0} \quad C_{D0}]^T\end{aligned}$$

which gives us the problem $d\underline{\mathbf{y}}/dt = \underline{\mathbf{f}}(\underline{\mathbf{y}}, t)$ with initial condition $\underline{\mathbf{y}}|_{t=0} = \underline{\mathbf{y}}_0$. This is set up for us to numerically integrate.

System of springs

Consider the system of masses and springs shown in Figure 6.7. Writing force balances (Newton's second law) on these two masses gives:

$$\begin{aligned}m_1 \frac{d^2 y_1}{dt^2} &= -k_A(y_1 - l_A) + k_B(y_2 - y_1 - l_B) \\ m_2 \frac{d^2 y_2}{dt^2} &= -k_B(y_2 - y_1 - l_B)\end{aligned}$$

These are 2nd-order differential equations. However, we can mathematically convert each into a system of first order equations by making the substitution⁷

$$\begin{aligned}y_3 &\equiv \frac{dy_1}{dt} \\ y_4 &\equiv \frac{dy_2}{dt}\end{aligned}$$

Then we can transform this into the system of first-order equations:

⁷Here, y_3 and y_4 have a physical meaning and are the velocities; however, we can consider this a purely mathematical procedure as it applies to any system of this sort.

$$\begin{aligned}\frac{dy_1}{dt} &= y_3 \\ \frac{dy_2}{dt} &= y_4 \\ \frac{dy_3}{dt} &= -\frac{k_A}{m_1}(y_1 - l_A) + \frac{k_B}{m_1}(y_2 - y_1 - l_B) \\ \frac{dy_4}{dt} &= -\frac{k_B}{m_2}(y_2 - y_1 - l_B)\end{aligned}$$

Now, we again have a system $\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t)$. We need to provide initial conditions for \mathbf{y} in order to cover the constants of integration; from the physics of this problem, it makes sense that we supply the initial positions of the masses (y_1, y_2) as well as their initial velocities (y_3, y_4). Thus, we again can write the initial condition as $\mathbf{y}|_{t=0} = \mathbf{y}_0$.

Solution procedure

The solution for a system of equations is really identical to that for a single equation, but we work with vectors instead of scalars. In both cases above, our problem could be stated as:

$$\begin{aligned}\frac{d\mathbf{y}}{dt} &= \mathbf{f}(\mathbf{y}, t) \\ \mathbf{y}|_{t=0} &= \mathbf{y}_0\end{aligned}$$

General step. Analogous to equation (6.3), we can write a vectorized version as

$$\mathbf{y}|_{t+h} = \mathbf{y}|_t + \mathbf{m}_t h$$

Then, each of the above methods (*e.g.*, Explicit Euler, Improved Euler, RK4) can be used in an analogous manner. For example, the slope estimate in Explicit Euler is just:

$$\mathbf{m}_t = \left. \frac{d\mathbf{y}}{dt} \right|_t$$

And the modifications are essentially identical in the other methods. Note that most pre-packaged solvers, like `scipy.integrate.odeint` expect that you will be dealing with a system, and expect function to take in a vector of \mathbf{y} and return a vector of $d\mathbf{y}/dt$. A scalar system is then just a special case of this, with vectors of length one. Thus, the procedure is really identical whether you are using a vectorized form or a scalar form.

6.2.7 Time steps

So far, we have been examining how to choose \mathbf{m} in the general equation:

$$\mathbf{y}|_{t+h} = \mathbf{y}|_t + \mathbf{m}_t h$$

which is the estimate of the slope for the current interval $(t, t + h)$. Yet, we still need to choose the interval width; that is, the time step h .

Generally, most solution methods become more accurate the smaller you make h (at least until you hit the numerical precision of the machine), so the choice of h controls the trade-off between speed and accuracy. If you are coding your own method, for example the RK4 approach, you can just try a systematic convergence test. *I.e.*, run your algorithm with a certain value of h . Then repeat with $h/2$ (or $h/10$, or whatever) and see if your results change. If the results do change, then your timestep was too large and you need to try

even smaller values. (And you should try moves in the opposite direction, as well.) This can give you an idea of the maximum time step that gives you an error that you can tolerate.

In general, however, the step size h does not need to be identical at every step throughout the algorithm. Intuitively, we need small steps in regions where the slope is changing rapidly—that is, in regions of high curvature—and where the slope is roughly constant, we can take very large steps. The prepackaged integrators (like `scipy.integrate.odeint`) have built in methods to adaptively choose the time step throughout the course of the integration. (This is in spirit similar to what we discussed in the optimization section, §3.2.3.) To take advantage of such capabilities, it’s a good idea to non-dimensionalize your system before integrating it (§6.2.1)—particularly if your original time scales are far from order one—as the algorithm will tend to be better behaved. There are different approaches to adapting the time step. According to Beers (B4:178), here is how the adaptation works in Matlab:

In MATLAB, `ode45` implements the RungeKuttaFehlberg 4-5 (RKF45) method, based upon a fifth-order RungeKutta rule, from which a fourth-order update formula is extracted using the same function evaluations. If the two methods agree closely, the time step can be increased safely, but if they disagree, the time step is too large and should be reduced. Thus, the user can specify a desired level of accuracy, and let `ode45` adjust the step size as needed. Use of `ode45` is demonstrated below.

6.2.8 Stiff systems

The optimal time step h is dictated by the characteristic times of the system. (These characteristic times often become naturally apparent when we non-dimensionalize, which in general you should do.) However, there can sometimes be multiple time scales within the same problem, and if they severely mismatch, we say we have a *stiff system*.

Examples of stiff systems. Here, let’s paint a picture of where stiff systems can arise with some examples.

- Let’s examine our spring system of §6.2.6. For the sake of argument, let’s say that $m_1 = m_2$ and $l_A = l_B$ but $k_A \gg k_B$. (Coincidentally or not, we say that Spring A is much stiffer than Spring B.) Let’s examine just the y_3 and y_4 terms. First, we see that y_3 is dominated by k_A , which moves very fast. Thus, we should choose very small time steps based on this observation. However, y_4 is dominated by k_B , which moves very slow. So there is not a great choice to make for the natural time step of the problem—we need to take small time steps, but take an immense number of them in order to capture any movement in y_4 .
- In our system of chemical reactions, it is normal for individual reaction rates to vary from one another by many orders of magnitude. That is, some steps of a reaction go immensely fast, while others are relatively slow. (If you remember the concept of a “rate-limiting step” from chemistry, this is the basis behind it.) Again, to capture the fast steps we should have tiny time steps, but to capture the overall kinetics we need to integrate for a very long time.
- If we’d like to describe the motions of the planets and moons in the solar system, we have some very different time scales. There are moons that make it around Jupiter in only hours (!), while Neptune takes about 165 earth years to make it once around the sun. All of the bodies of the solar system make gravitational tugs on one another, so to capture the fast events we need small time steps, but to capture the long events we need many of them.

It’s important to recognize when you have a stiff system as your solution method should change. In Matlab, you need to explicitly choose a stiff solver (*e.g.*, `ode23s`) when this occurs, although `scipy`’s solvers should automatically detect stiff systems and change methods appropriately. In general, implicit solvers work better for stiff systems, which we briefly introduce in the next section.

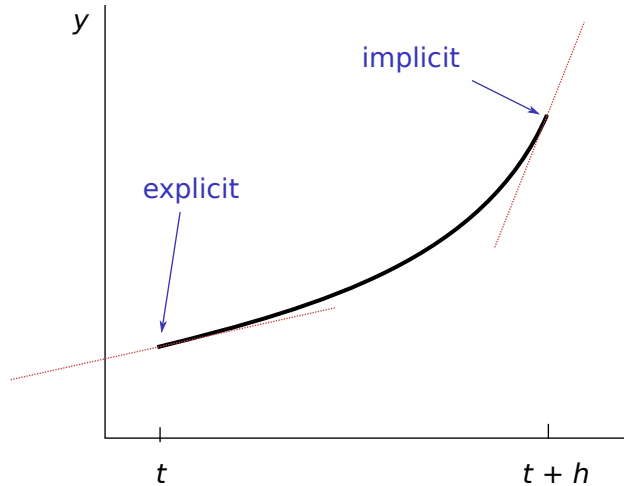


Figure 6.8: Comparison of the slope estimate in the two Euler methods.

6.2.9 Implicit methods

Implicit methods use the slope from the end (or midpoints) of an interval by solving for it algebraically; this makes them more expensive (due to the need to solve a nonlinear algebraic system at each time step), but the expense is typically worth it for stiff systems, as it allows for larger time steps. Here, we'll just introduce the Implicit Euler method, which introduces the basic concept used in implicit methods.

Implicit Euler

In the Explicit Euler method, we used the slope estimate from the beginning of the interval, as shown in Figure 6.4. A different option, that ends up being more robust for some systems such as stiff systems, is to use the slope at the end of the time step interval, shown in Figure 6.8. Mathematically,

$$m_t = \left. \frac{dy}{dt} \right|_{t+h, y(t+h)}$$

$$y|_{t+h} = y|_t + \left. \frac{dy}{dt} \right|_{t+h, y(t+h)} h$$

Using this method on our example from §6.2.3,

$$y(0.1) = y(0) + [(t+h) \cdot y(0.1) + 1] h$$

$$y(0.1) = 1 + [0.1 \cdot y(0.1) + 1] (0.1)$$

This equation does not give the value of $y(0.1)$ *explicitly*; instead, we have to do some algebra to solve for $y(0.1)$ in the above equation. If my algebra is correct, we get 1.111 in this case, close to, but not identical to, the value of 1.110 in the explicit method.

For a general case, the algebra will not be this easy. Thus, the algorithm would use a numerical method to solve for the value of $y(0.1)$ in the above case. So at each time step of our numerical integration, the algorithm also needs to numerically solve a system of nonlinear algebraic equations, as we did in §2. This results in a costlier algorithm, although in general we will have very good initial guesses for our algebraic solver, as it can just use the values from the previous step as the initial guess. (Or, even better, it could use an explicit Euler estimate as the initial guess.) However, some implicit methods allow larger time steps to be taken robustly, so although the time per step can increase, the number of steps may decrease to offset this.

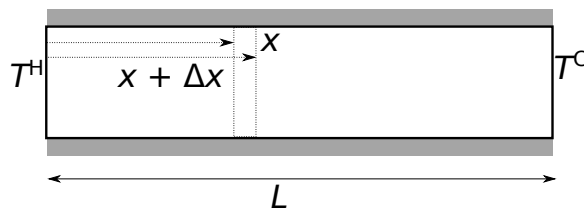


Figure 6.9: A simple example for the boundary value problem.

6.3 Boundary value problems

Reference: B6, K21.4–K21.5

We next move on to boundary value problems. Boundary value problems are second-order differential equations, whose constants of integration are satisfied at two opposing boundaries of the problem; they typically involve spatial derivatives, where the physics of the problem dictates what is occurring at the boundary. Whereas initial value problems looked like extrapolation, boundary value problems are more similar to interpolation, and thus can be a bit better behaved. Consequently, the numerical approximations are simpler, but we typically do a bit more work to code up the approach ourselves. We'll start with an example of a boundary value problem before getting into the numerical approach.

6.3.1 Example

In the initial value problem section, we used as an example a block of metal, where we assumed the inside of the block was at a uniform temperature and that it was cooling—but there were no spatial temperature profiles within the metal block. Here, we'll look at the opposite limiting case: where there is a temperature profile within the block, but it is constant in time.

Figure 6.9 shows a block of material where one end is at T^H and the other at T^C ; to keep the problem simple, we'll assume that heat conduction occurs only in the x direction, as the bar is insulated in the opposite directions.

We'll further assume that the system is at *steady state*. Steady state means that the properties of the system are not changing with time. Although heat is certainly flowing through our system, we say the system itself is unchanged if we take a snapshot of it now or a minute from now. This is the limiting case as $t \rightarrow \infty$, and let's us ignore time derivatives. In this case, in the real system the bar would start out with some initial temperature profile, but over time this profile would evolve into a profile that is unchanging with time; this latter profile is the one we are interested in here.

To analyze this system, we create a small “shell” between x and $x + \Delta x$ on our figure. This is a technique you likely learned in a thermodynamics or heat transfer course. Then we can write a balance equation to track the heat flows.

$$(\text{Accumulation within}) = (\text{Flow in}) - (\text{Flow out}) + (\text{Generation within}) - (\text{Consumption within})$$

$$0^{(\text{SS})} = qA \Big|_x - qA \Big|_{x+\Delta x} + 0 - 0$$

where q_j is the heat flux and A is the area. Note that a “flux” is defined as a flow per cross-sectional area, which is why we multiply it by the area A . The heat flux is defined as positive when flowing from left to right. (Here, we are picturing that the heat flows from left to right in our image, but this equation works fine if the temperatures are reversed.) Assuming A is constant with respect to x —that is, the bar is of uniform width—we can divide by $A\Delta x$ and take the limit as $\Delta x \rightarrow 0$, which changes our algebraic equation into a differential equation:

$$\lim_{\Delta x \rightarrow 0} \frac{q|_x - q|_{x+\Delta x}}{\Delta x} = 0$$

$$\frac{dq_x}{dx} = 0$$

Fourier's law of heat conduction says $q = -k \frac{dT}{dx}$ (for one-dimensional flux). Assuming k is a constant in this material, we get:

$$k \frac{d^2T}{dx^2} = 0$$

$$\boxed{\frac{d^2T}{dx^2} = 0}$$

The above is the governing equation for this system, valid in the interior range $0 < x < L$. When we integrate this second-order equation, we will get two constants of integration, which we can satisfy with the boundary conditions

$$T|_{x=0} = T^H$$

$$T|_{x=L} = T^C$$

Of course, we could and should solve this particular problem by hand, but we'll use it as an example for our numerical approaches.

6.3.2 Non-dimensionalization

Let's be good and nondimensionalize this equation. For temperature, we can be reasonably sure that the temperature in the block will be contained in the range of hot and cold limits, so let's define θ to be 1 when hot and 0 when cold. We can do this with:

$$\theta \equiv \frac{T - T^C}{T^H - T^C}$$

We also have a very natural length scale: the temperature varies over the length of the bar, L , and we can define a non-dimensional length as:

$$\tilde{x} \equiv \frac{x}{L}$$

Plugging this in we get:

$$\frac{d^2\theta}{d\tilde{x}^2} = 0$$

$$\theta|_{\tilde{x}=0} = 1$$

$$\theta|_{\tilde{x}=1} = 0$$

This didn't change our system much in this case, since we didn't have any parameters in our equation beyond the boundary value conditions, but it at least allows us to have a general solution.

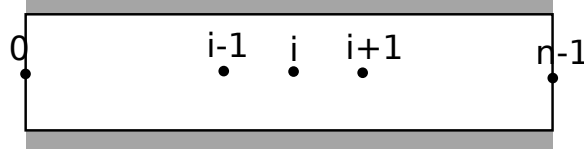


Figure 6.10: A grid for the finite-difference approximation. For convenience when we implement this, we have adopted python’s numbering convention (starting at 0).

6.3.3 Finite-difference approximation

We can use the same approach that we’ve seen several times by now: we write a finite difference approximation for our derivatives. Here, we need to do this for a second-order derivative. Let’s set up a discretized grid on which we can estimate our derivatives; this is shown in Figure 6.10. Our first-order finite difference approximation to estimate the derivative at point i can be written⁸ as:

$$\left. \frac{du}{dx} \right|_i \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x}$$

We can intuitively derive the second-order derivative as a series of two first derivatives; you could also derive this more formally with a Taylor series expansion.

$$\left. \frac{d^2u}{dx^2} \right|_i = \frac{d}{dx} \left(\left. \frac{du}{dx} \right|_i \right) \approx \frac{\frac{u_{i+1} - u_i}{\Delta x} - \frac{u_i - u_{i-1}}{\Delta x}}{\Delta x}$$

$$\left. \frac{d^2u}{dx^2} \right|_i \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2}$$

Applied to the example

If we apply the finite-difference approximation to our example, we will convert our differential equations into a system of algebraic equations. We will write one equation per point on our grid, shown in Figure 6.10.

Let’s start at a general interior point i , which are governed by $d^2\theta/d\tilde{x}^2 = 0$. We can discretize this as:

$$\frac{\theta[i+1] - 2\theta[i] + \theta[i-1]}{\Delta \tilde{x}^2} = 0$$

$$\theta[i+1] - 2\theta[i] + \theta[i-1] = 0$$

The above is a simple algebraic equation and is valid at $0 < i < n-1$. Node 0 is our left boundary condition:

$$\theta[0] = 1$$

Node $n-1$ is our right boundary condition

$$\theta[n-1] = 0$$

Now, we have a system of n algebraic equations, with n unknowns (in the vector $\underline{\theta}$).

⁸This is the “centered” finite-difference approximation, where we take the perturbations to be centered on our point of interest. We can also do “forward” FDAs and “backward” FDAs that only examine the downstream or upstream neighbor.

Solution approach

In general, we'll have a nonlinear system of n equations and n unknowns, and we can use a non-linear solver as discussed in §2 to find the values of θ across our grid. However, in this particular case, we actually get a simple linear system that we can write in the standard form $\underline{\mathbf{A}}\underline{\mathbf{x}} = \underline{\mathbf{b}}$, and use the methods from §1.3. Writing in this form, our system (for $n = 6$) is

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \\ \theta_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Now, we just solve this with linear algebra; we can do this in python as:

```
import numpy as np

n = 6
A = np.diag([-2] * n) + np.diag([1] * (n-1), 1) + np.diag([1] * (n-1), -1)
A[0,:] = 0.
A[0,0] = 1.
A[-1,:] = 0.
A[-1,-1] = 1

b = np.zeros((n, 1))
b[0] = 1.

x = np.linalg.solve(A, b)
print(x)
```

If we run this code, we get the result that $\underline{\theta} = [1, .8, .6, .4, .2, 0]$ That is, we just get out a linear profile (which we should expect from our original equations).

6.3.4 Multidimensional boundary value problems

The approach from the previous section also works if we have multiple dimensions, which results in partial differential equations. However, we'll have to do a little mapping in order to convert the resulting arrays into vectors, which is the form that is expected by non-linear algebraic equation solvers. Here, we'll again assume our independent variables are spatial variables, which is a common case in engineering and physics problems.

Example: Two-dimensional block

Here, we'll look at heat conduction in a two-dimensional block, in which the edges are at four distinct but constant temperatures; Figure 6.11. Here we'll assume that the temperature profile is constant in the z direction. This is the exact same problem as we did in one dimension, but with a second dimension. To derive the governing equations, we'll use the same shell balance approach as we did in the previous example, but now we'll allow heat to flow in two orthogonal directions, x and y . (Thus, heat can flow in any (two-dimensional) direction, as a linear combination of the two flows.) We'll assume it's a uniform thickness L_z and perfectly insulated on the top and bottom, making this a purely two-dimensional problem, and solve for the solution at steady state (SS):

$$(\text{Accumulation within}) = (\text{Flow in}) - (\text{Flow out}) + (\text{Generation within}) - (\text{Consumption within})$$

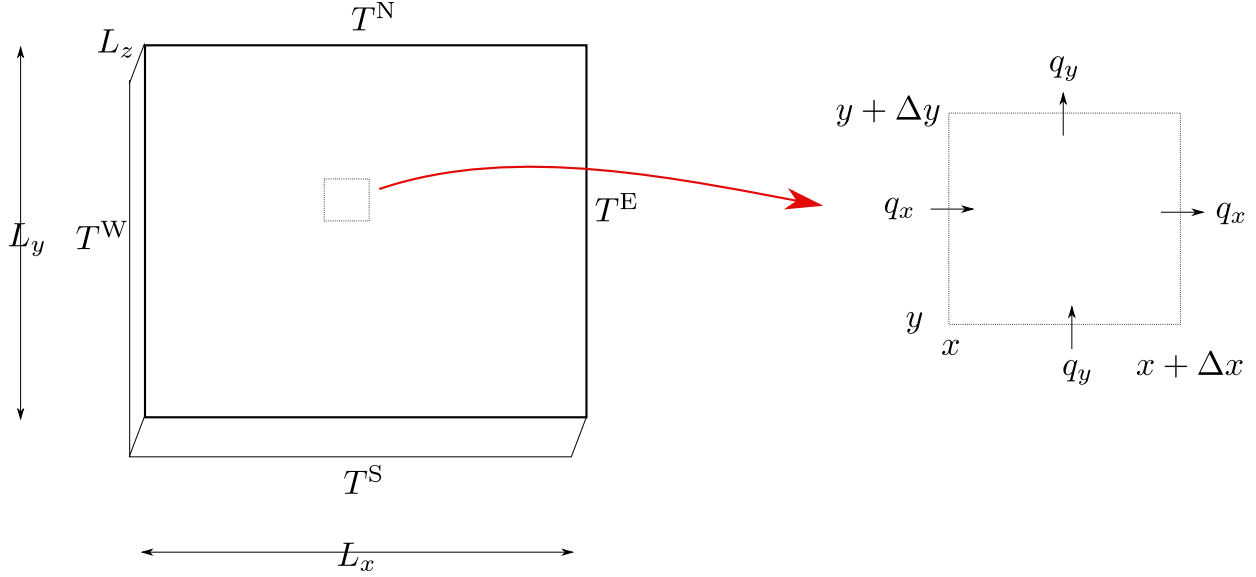


Figure 6.11: A two-dimensional block, at steady state, with four different edge temperatures.

$$0^{(\text{ss})} = q_x A_x \Big|_x + q_y A_y \Big|_y - q_x A_x \Big|_{x+\Delta x} - q_y A_y \Big|_{y+\Delta y}$$

The subscripts can be a little confusing, so we'll go through each individually. Here, q_x refers to the flux in the x direction, which is positive in the direction of increasing x ; q_y is analogous in the y direction. These are shown on the inset drawing. A_x and A_y are the cross sectional areas across which q_x and q_y flow; we'll define these next. Finally, the subscripts on the bar ($|_x$, $|_{x+\Delta x}$, ...) mean "evaluated at" that location. Putting it together in words, $q_x A_x|_{x+\Delta x}$ means "the flux in the x direction multiplied by the cross-sectional area in this direction evaluated at a coordinate of $x + \Delta x$ ". The appropriate cross sectional areas will be the area of the plane orthogonal to the flux:

$$A_x = \Delta y \cdot L_z$$

$$A_y = \Delta x \cdot L_z$$

We'll plug these in and divide by $L_z \cdot \Delta x \cdot \Delta y$:

$$0^{(\text{ss})} = q_x \Delta y L_z \Big|_x + q_y \Delta x L_z \Big|_y - q_x L_z \Delta y \Big|_{x+\Delta x} - q_y L_z \Delta x \Big|_{y+\Delta y}$$

$$0 = \frac{q_x|_x - q_x|_{x+\Delta x}}{\Delta x} + \frac{q_y|_y - q_y|_{y+\Delta y}}{\Delta y}$$

Now, we can see that we are set up to turn these fractions into derivatives. Taking the limit as both Δx and Δy go to zero gives

$$0 = -\frac{\partial q_x}{\partial x} - \frac{\partial q_y}{\partial y}$$

Note that these are now *partial derivatives*, since there are two independent variables x and y . Fourier's law can be applied in each of the orthogonal directions,⁹ $q_x = -k \partial T / \partial x$ and $q_y = -k \partial T / \partial y$, leading to

⁹Note that k —the heat conduction coefficient—could in principle be different in the x and y directions; this would be referred to as an anisotropic material. An example would be graphite, which conducts heat much more strongly across the graphene planes than it does orthogonal to them. If that were the case, we would just use k_x and k_y instead of k .

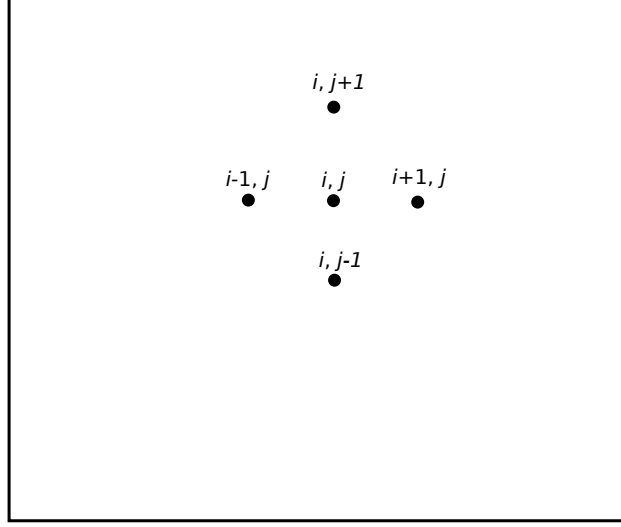


Figure 6.12: The discretization for the two-dimensional problem.

$$0 = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

This is the governing equation for the interior of the block. With two second-order derivatives we'll have four constants of integration to cover, which we do with our four boundary conditions. The boundary conditions are:

$$T \Big|_{x=0} = T^W$$

$$T \Big|_{x=L_x} = T^E$$

$$T \Big|_{y=0} = T^S$$

$$T \Big|_{y=L_y} = T^N$$

In the interest of brevity, we'll skip non-dimensionalizing this problem. (x and y would be scaled by L_x and L_y , respectively. We'd probably use some physical knowledge of the problem (about the maximum and minimum temperatures expected among T^N , T^S , T^W , T^E) to choose the scaling.

Numerics

Now we'll look at the numerical solution; discretized as in Figure 6.12. Note that we now have two indices to worry about: i indicates the x direction, while j indexes the y direction. The finite-difference approximation in two dimensions is just written as

$$\frac{\partial T}{\partial x} \approx \frac{T_{i+1,j} - T_{i-1,j}}{2\Delta x}$$

$$\frac{\partial T}{\partial y} \approx \frac{T_{i,j+1} - T_{i,j-1}}{2\Delta y}$$

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2}$$

Table 6.1: Mapping.

New	Old
A[0]	$T_{0,0}$
A[1]	$T_{0,1}$
A[2]	$T_{0,2}$
\vdots	\vdots
A[n-1]	$T_{0,n-1}$
A[n]	$T_{1,0}$
A[n+1]	$T_{1,1}$
\vdots	\vdots

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2}$$

Therefore, if we plug this into our differential equation to discretize the equation we get an array of equations:

$$f[i, j] = 0 = \frac{1}{(\Delta x)^2} (T[i-1, j] - 2T[i, j] + T[i+1, j]) + \frac{1}{(\Delta y)^2} (T[i, j-1] - 2T[i, j] + T[i, j+1])$$

There is one of these above equations for each of the interior points within our grid. At the edges, we employ our boundary conditions to get our remaining equations:

$$\begin{aligned} f[0, j] &= T[0, j] - T^W = 0 && \text{for all } j \\ f[n-1, j] &= T[n-1, j] - T^E = 0 && \text{for all } j \\ f[i, 0] &= T[i, 0] - T^S = 0 && \text{for all } i \\ f[i, n-1] &= T[i, n-1] - T^N = 0 && \text{for all } i \end{aligned}$$

Note that there are some “disputed” points at the corners. For example, at node [0,0] both T^S and T^W apply. We can decide what to do with the disputed points at the corners, like average them between the two limits, for example, or just assign them to one or the other limit. In practice, as the grid spacing gets smaller, all of these treatments will be roughly identical, as the influence of the single point will be insignificant.

Solving algebraically.

We now have $n_x \times n_y$ equations (the array $\underline{\mathbf{T}}$) and $n_x \times n_y$ unknowns (the array $\underline{\mathbf{f}}$), which we can solve algebraically. However, our system is set up to be represented as a 2-d array, but most algebraic solvers expect a vector. Thus, we need to map our equations from this array into vector form. This used to be a slightly tedious operation, where we would reassign indices as in Table 6.1; if you are coding something in Fortran or similar you will likely still need to take a similar approach.


Python / speed notes. However, the re-mapping task is actually quite simple to do using in python. We can do the mapping internally in python just using the `resize` (or `reshape` command). An example solution in python is shown below. Note that here we’ve erred on the side of simplicity of the code, but this will be somewhat slow. There are reasons that it is slow, which if they are addressed will speed it up significantly:

1. Nested for loops are slow in python and other “scripting languages”; if they are the core part of your calculation like this (and you can’t tolerate the wait), you should seek a different approach. There are a number of approaches that can fix this:

- Write your code in Fortran, which is efficient for such loops.

- Write just that function in Fortran, then link it to python using “f2py”.
- Use a python compiler that tries to optimize for speed, like “numba” or “cython”.
- Convert your equations into standard array operations, which are handled quickly by numpy. This will involve rewriting your functions as an operator: this could be an operator that transforms your flattened T array, for example.

2. We are not using the Jacobian for \mathbf{f} . As we discussed in §2, if you do not supply the Jacobian, it will calculate one for you analytically. Each analytical jacobian estimate involves $n_x \times n_y$ function calls. Since these expressions are simple, it would be straightforward to write a Jacobian for f , which would speed up the procedure immensely. Note that code that automatically returns analytical derivatives is becoming increasingly popular; autograd is an example package of this for python.

File attached here. 

```

1 import numpy as np
2 from scipy.optimize import fsolve
3 from matplotlib import pyplot
4
5 def get_f(x):
6     T = x.reshape((nx, ny)) # <-----Note!
7     f = np.empty((nx, ny))
8     # Interior points.
9     for i in range(1, nx - 1):
10         for j in range(1, ny - 1):
11             f[i,j] = (T[i-1,j] - 2. * T[i,j] + T[i+1,j]) / dx**2
12             f[i,j] += (T[i,j-1] - 2. * T[i,j] + T[i,j+1]) / dy**2
13     # Edges.
14     f[0,:] = T[0,:] - T_W
15     f[-1,:] = T[-1,:] - T_E
16     f[:,0] = T[:,0] - T_S
17     f[:, -1] = T[:, -1] - T_N
18     return f.reshape((nx * ny)) # <-----Note!
19
20 nx, ny = 50, 50
21 T_N, T_S, T_W, T_E = 400., 200., 320., 290. # K
22 Lx, Ly = 1., 1.
23 dx = Lx / nx
24 dy = Ly / ny
25 Tguess = np.ones((nx, ny)) * (T_N + T_S + T_E + T_W) / 4.
26 Tguess = Tguess.reshape(nx * ny) # <-----Note!
27 ans = fsolve(func=get_f, x0=Tguess)
28 ans.resize((nx, ny)) # <-----Note!
29 fig, ax = pyplot.subplots()
30 ax.contourf(ans.T)
31 ax.set_xlabel('$x$ dimension')
32 ax.set_ylabel('$y$ dimension')
33 fig.savefig('twod.pdf')

```

The result of this integration using the script provided (on a 50×50 grid) is shown in Figure 6.13.

Convergence notes. Sometimes, it can be challenging to find a good initial guess for your solver. (In this case, the initial guess is a temperature profile of the block.) It can sometimes be more reliable—and even faster—to instead solve the non-steady-state problem, as we’ll discuss in [\[link to come...\]](#). That is, we instead start from a physically reasonable initial temperature profile—such as the block at the mean of the four temperatures—and we integrate forward in time how the temperature profile develops. As $t \rightarrow \infty$, the profile eventually stops changing, and this is the steady state profile we were after above. (Since the time-based integration only asymptotically approaches the true steady state value, if you would like additional accuracy you could then use the resulting value as an initial guess in the non-linear solver approach above, to solve for arbitrary accuracy.)

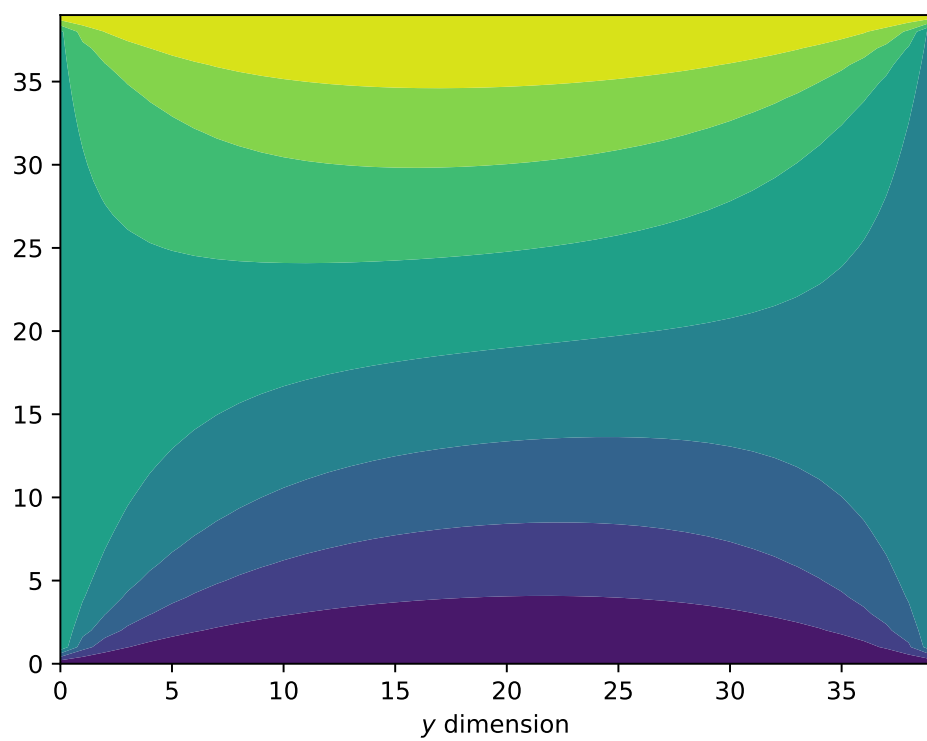


Figure 6.13: The two-dimensional integration.

Bibliography

- [1] Kreyszig, E. *Advanced Engineering Mathematics*. Wiley, 10th edition edition **2011**.
- [2] Beers, K.J. *Numerical Methods for Chemical Engineering: Applications in Matlab*. Cambridge University Press **2007**.
- [3] Bryan, K.; Leise, T. The \$25,000,000,000 Eigenvector: The Linear Algebra behind Google. *SIAM Review* **2006**; 48, 569–581.
- [4] Schmidt, L.D. *The Engineering of Chemical Reactions*. Oxford University Press **1998**.
- [5] Dill, K.; Bromberg, S. *Molecular Driving Forces: Statistical Thermodynamics in Biology, Chemistry, Physics, and Nanoscience*. Garland Science **2010**.
- [6] Peterson, A.A. Global Optimization of Adsorbate–Surface Structures While Preserving Molecular Identity. *Topics in Catalysis* **2014**; 57, 40–53.