Numerical Methods with Programming

Numerical Methods with Programming

Leonid Kovalev
Syracuse University

December 6, 2020

Acknowledgements

The following textbooks were helpful in preparation of these notes.

- Numerical Methods: Using MATLAB by George Lindfield and John Penny
- A First Course in Numerical Methods by Uri M. Ascher and Chen Greif
- Approximation Theory and Approximation Practice by Lloyd N. Trefethen
- Numerical Computing with MATLAB by Clive Moler

Preface

The topics covered in this course can be divided into 5 units.

1. Matlab: Matrix/vector manipulation, matrix/vector creation, element-wise operations, built-in functions, input/output, 2d graphics, scripting, functions.

2. Solving equations: linear systems, bisection method, fixed point method, Newton and secant methods, multivariable Newton's method, Broyden's method.

3. Numerical calculus: numerical differentiation, numerical integration: trapezoidal rule, Simpson's rule, orthogonal polynomials (Legendre, Laguerre), Gaussian integration, Gauss-Laguerre integration, adaptive integration, solving ODE with Euler's method and trapezoidal method, ODE systems.

4. Data fitting: polynomial interpolation, spline interpolation, discrete Fourier transform, linear least squares, model comparison, nonlinear least squares, transforming data.

5. Optimization: linear programming, single-variable minimization, steepest descent, conjugate gradient method, Nelder-Mead method, constrained optimization, applications.

Numerical linear algebra is not included above because it is the subject of a separate course (MAT 532).

Although the course is built around Matlab software, one can use Octave to run examples and do homework just as well.

Contents

Part I

Using Matlab

Chapter 1

Scalars, vectors and matrices

We will only need three types of Matlab objects: scalars, vectors, and matrices. A scalar is just a single number, like 2.75. Vectors and matrices deserve a more careful look. Being able to use them efficiently is a key to programming in Matlab. Note that "Mat" in "Matlab" stands for "matrix", not "mathematics".

1.1 Matlab interface

The first thing you see in Matlab is its **Command Window** where the prompt » invites you to enter any command that will be executed at once. This window is useful for quickly trying out things. You should use it to try out the one-line examples that appear in the text of these notes.

Most of our work, including homework assignments, will involve **Script Editor**. To open this editor, click "New Script" button on the left of the "Home" menu of Matlab. The chain of commands entered in the editor forms a script. The button "Run" of the "Editor" menu runs a script; pressing F5 does the same.

The result of script computations can be shown in the Command Window. There are several ways to make it appear. One of them is `disp` command: for example,

```
disp('Hello world')
```

displays "Hello world", and

```
disp(2/3)
```

displays 0.6667. Notice that Matlab displays just four digits after the decimal dot. This is intentional: showing fewer digits of each number makes is possible to fit more of them on the screen. There are ways to change the number format to display more digits, which we will consider later.

Example 1.1.1 Golden ratio. Write a script that displays an approximate decimal value of the **golden ratio** $\dfrac{\sqrt{5}+1}{2}$.

Answer.

```
disp((sqrt(5)+1)/2)
```

□

1.2 Assigning scalar values

The command

```
x = 3/4
```

assigns the value of 0.75 to variable `x`. Matlab will also display the result of this assignment (the number 0.75) to you. Most of the time, we do not need it to print the result of every computation. Ending a command with a semicolon `;` suppresses the output: try

```
x = 3/4;
```

Now that the variable `x` has a value, we can use it in computations. For example,

```
y = x^2 + 5*x - 1
```

will result in `y` being assigned the value 3.3125. It is important to realize that every variable must be assigned a value before it can be used on the right hand side of an assignment. If we try to execute `y = x^2` without assigning any value to `x`, the result will be an error: `x` is undefined.

It is important to understand the difference between equations and assignments. In mathematics, $x = x + 1$ is an equation which has no solutions. In Matlab,

```
x = x + 1;
```

is an assignment command which means "take the current value of x, add 1 to it, and assign the result to x". The effect is that the value of x is increase by 1.

Example 1.2.1 Arithmetic and geometric means. Write a script that does the following. Assign two unequal positive numbers to x and y (for example, two different digits of your SUID). Compute their arithmetic mean $a = \dfrac{x+y}{2}$ and geometric mean $g = \sqrt{xy}$. Then display the arithmetic mean of a and g. Finally, display the geometric mean of a and g.

Answer.

```
x = 4;
y = 7;
a = (x+y)/2;
g = sqrt(x*y);
disp((a+g)/2)
disp(sqrt(a*g))
```

Remark: Karl Gauss discovered that if the above process of taking arithmetic and geometric means is repeated, both these means *quickly* converge to the same number. This number is called the **arithmetic-geometric mean** of x and y. It is linked to certain integrals which cannot be evaluated with calculus methods, and therefore provides an efficient computational approach to such integrals. For more, see Wikipedia. □

1.3 Creating vectors and matrices

A **vector** in Matlab is an ordered list of numbers. It does not always have a geometric meaning. For example, recording the high temperature on every day of August, we get a vector with 31 components; we do not normally visualize it as an arrow of some length and direction. Matlab distinguishes between row vectors and column vectors, which are defined below.

A **matrix** is a rectangular array of numbers. For example, a 4×6 matrix has 4 rows and 6 columns. A matrix with one row (for example a 1×6 matrix) is a **row vector**. A matrix with one column (for example a 4×1 matrix) is a **column vector**. Thus, vectors in Matlab are just a special kind of matrices.

One can create a row vector by listing its components:

```
x = [8 6 7 5 3 0 9]
```

or

```
x = [8, 6, 7, 5, 3, 0, 9]
```

Either spaces or columns can be used to separate the entries. To create a column vector, one can either separate the entries by semicolons:

```
x = [8; 6; 7; 5; 3; 0; 9]
```

or create a row vector and **transpose** it by putting an apostrophe at the end:

```
x = [8 6 7 5 3 0 9]'
```

To create a matrix one needs both kinds of separators: spaces or colons within each row, and semicolons between the rows. For example,

```
A = [8 6; 7 5; 3 0]
```

creates a 3×2 matrix. Its transpose `A'` is a 2×3 matrix, same as

```
[8 7 3; 6 5 0]
```

Regularly-spaced vectors are described by three numbers: first entry, step size, and last entry. For example: `3:2:15` means the same as `[3 5 7 9 11 13 15]`: the first entry is 3, after that they increase by 2, until reaching 15. The step size can be omitted when it is equal to 1: that is, `-1:4` is the same as `[-1 0 1 2 3 4]`.

Zeros and ones are special kinds of matrices and vectors, filled with the same number: 0 or 1. The commands are called `zeros` and `ones`. They require two parameters: the number of rows and the number of columns. For example, `zeros(2, 2)` creates a 2×2 matrix filled with zeros, and `ones(1, 5)` is a row vector with 5 entries: `[1 1 1 1 1]`. These are useful as starting points for building matrices and vectors.

Example 1.3.1 Integers in descending order. Create a vector with numbers 20, 19, 18, ..., 3, 2, 1 in this order.

Answer.

```
v = 20:-1:1
```

The step size here is -1, so the term after 20 is 20 + (-1) = 19, after that we get 19 + (-1) = 18, and so on until reaching 1. □

1.4 Operations on vectors and matrices

Addition and *subtraction* works the same way as it does in Linear Algebra. If two vectors, or two matrices, have the same size, they can be added or subtracted. For example:

```
x = [3 7 2];
y = [8 -2 0];
z = x + y;
```

results in `z` being `[11 5 2]`. But we cannot add `[8 -2]` and `[2 5 8]` because these vectors have different sizes.

Scalar multiplication also works as expected: with `x` as above, `3*x` is the vector `[9 21 6]`.

One can also multiply two vectors, or two matrices, or a vector and a matrix. To understand how this works in Matlab, keep in mind that a vector is treated as a matrix with one row (or one column). Two matrices can be multiplied only when the inner dimensions agree: that is, two matrices of sizes $m \times n$ and $p \times q$ can be multiplied when $n = p$. The product has size $m \times q$. Some examples:

- `[3 7] * [4 2]` is ***an error***: the first argument has size 1×2, the second also has size 1×2, and the inner dimensions do not agree: $2 \neq 1$.
- `[3 7] * [4; 2]` is `26`. the first argument has size 1×2, the second has size 2×1, so the product is defined and has size 1×1 which makes it a scalar. This is how one computes the **scalar product** of two vectors: it comes from multiplying a row vector by a column vector.
- `[1 2 3; 4 5 6] * [-4; 0; 3]` is `[5; 2]`. The sizes 2×3 and 3×1 are compatible and the product has size 2×1. This is how **matrix-vector** products work in linear algebra too: the vector, placed to the right of a matrix, must be written as a column.
- `[5 -2] * [1 2 3; 4 5 6]` is `[-3 0 3]`. The sizes 1×2 and 2×3 are compatible and the product has size 1×3. So, we can have a vector to the left of a matrix when it is a row vector.
- `[0 1; -1 0] * [1 2 3; 4 5 6]` is `[4 5 6; -1 -2 -3]`. This is a **matrix product**. Two matrices have compatible sizes 2×2 and 2×3, and the product is of size 2×3.

Example 1.4.1 Matrix multiplication is not commutative. Write a script which illustrates that the product of two 2×2 matrices depends on their order.

Answer.
```
A = [1 2; 3 4];
B = [5 6; 7 8];
disp(A*B)
disp(B*A)
```

Remark: the particular numbers are not important: if you fill two square matrices with random nonzero numbers, you will probably find their product depends on the order of term. □

1.5 Accessing the entries of vectors and matrices

We often need to manipulate matrices and vectors by extracting, replacing or rearranging some elements. The elements are indexed *starting with 1*: so, after executing `v = [8 6 -4]` we find that `v(1)` is 8, `v(2)` is 6, and `v(3)` is -4. One can also use indexing from the end of the vector: `v(end)` is -4, `v(end-1)` is 6, and so on. For a matrix the indexes are ordered as (row, column). So, if `A = [5 6 7; 9 8 6]`, then `A(1, 3)` is 7 and `A(2, 1)` is 9.

A powerful tool for working with matrices is the **colon : selector**. When it replaces an index, it means "run through all values of that index". Given a matrix A, we can use `A(1, :)` to get its first row, `A(:, 2)` to get the second column, and `A(:, end)` for the last column.

We can use a **vector of indexes** to extract several elements of a vector or a matrix. For example, `v(3:end-1)` extracts all entries of `v` starting with the 3rd one and ending with the one before the last. If `v` was `[8 6 4 3 2 1]`, the result would be `[4 3 2]`. For another example, `v(2:2:end)` extracts all even-numbered elements of v. In the above example this would be `[6 3 1]`. For a matrix `A`, we can select both rows and columns at the same time: `A(2:3, 2:5)` means taking the elements of `A` that appear in rows 2-3 and in columns 2-5. The result is a submatrix of size 2×4.

The entries can also be assigned, for example `v(2:2:end) = -3` makes all even-numbered entries of vector `v` equal to -3.

Example 1.5.1 Extracting submatrices. Display the 2×2 submatrix in the middle of a given 4×4 matrix such as

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

Answer.

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16];
B = A(2:3, 2:3);
disp(B)
```

□

1.6 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 1.6.1 Scalar product of two vectors. Write a script that calculates and displays the scalar product of two vectors: [1 2 ... n] and [n n-1 ... 1]. Use the first two digits of SUID as your value of n.

Answer.

```
n = 28;
u = 1:n;
v = n:-1:1;
disp(u*v')
```

Explanation. Recall from Section 1.3 that `a:h:b` means a regularly spaced vector whose entries start with `a`, and then change by amount `h` until reaching `b`. When `h` is omitted, it is understood to be 1. So, this code makes `u` equal to [1 2 ... 28] and `v` equal to [28 27 ... 1]. Both are row vectors. Transposing the second vector into a column is necessary for their product to make sense, as noted in Section 1.4. Note that each of these lines ends with a semicolon, which prevents the intermediate results from being displayed on screen. Then `disp` displays the final result. □

Example 1.6.2 Create a vector for use in Simpson's method. Write a script that creates and displays a row vector with 25 elements of the form [1 4 2 4 2 ... 2 4 1]. This particular sequence of numbers is used in "Simpson's Method" of numerical integration which we will encounter later.

Answer.

```
v = ones(1, 25);
v(2:2:end) = 4;
v(3:2:end-2) = 2;
disp(v)
```

Explanation. The first line creates a vector of 25 ones, the second changes its even-numbered entries to 4, the third changes odd-numbered entries (except the first and last) to 2. Re-read Section 1.5 if this is unclear. □

Example 1.6.3 Checkerboard matrix. Create and display a "checkerboard" matrix of size 8×8: it should look like

```
1 0 1 0 ...
0 1 0 1 ...
1 0 1 0 ...
...........
```

Answer.

```
A = zeros(8, 8);
A(1:2:end, 1:2:end) = 1;
A(2:2:end, 2:2:end) = 1;
disp(A)
```

Explanation. The first line creates an 8×8 matrix of zeros. The second works with the entries where both the row number and column number are *odd*, changing them to 1. The third does the same when both row and column numbers are *even*. The result is a checkerboard. □

Question 1.6.4 Different order of multiplication. If in Example 1.6.1 we multiply the vectors in the different order, `v'*u`, there is no error message and Matlab produces a result. What does this result mean? □

Question 1.6.5 Even vs odd in Simpson's method. In Example 1.6.2, the pattern 14242....4241 requires an odd number of elements. We had 25 which is an odd number. If 25 is replaced by an even number, Matlab would not complain but the result would have a different pattern. What would it be? Try to answer without running Matlab. □

1.7 Homework

1. Suppose that `A` is a matrix. Describe, in words, the following objects:

(a) `A(3, :)`

(b) `A(:, end-2)`

(c) `A(1:2, end-1:end)`

(d) `A(1:2:end, 1)`

(e) `A(:, end:-1:1)`

2. Write a script to do the following:

```
x = (first 5 digits of your SUID);
y = (all entries of x except first);
z = (all entries of x except last);
u = (the average of vectors y and z);
v = (the scalar product of y and z);
```

Then display the vectors u and v using `disp`.

Note that `y` and `z` must be computed from the vector x, not entered number-by-number.

3. Write a script that does the following:

(a) Let `A` be some 3 by 3 matrix filled with numbers 1,2,. . . ,9

(b) Let `x` be the product of A and a row vector with all entries equal to 1 (think about the correct order of multiplication).

(c) Let `y` be the product of A and a column vector with all entries equal to 1.

When uploading your script to Blackboard, leave a comment with answers to the following questions:

(a) What is the meaning of the entries of x? In other words, what information about A do we get from the entries of x?

(b) What is the meaning of the entries of y?

Hint. Both x and y represent certain *sums*.

Chapter 2

Linear systems, array operations, plots

The goal of this class is to learn basic ways in which vectors and matrices are used: solving the systems of linear equations; performing computations with many numbers at once; and graphing functions.

2.1 Systems of linear equations

A system of linear equations, such as

$$\begin{aligned} 3x_1 - 2x_2 &= 6 \\ 2x_1 + 5x_2 &= -3 \end{aligned}$$

can be written in matrix form as $Ax = b$ where A is the matrix of coefficients,

$$A = \begin{pmatrix} 3 & -2 \\ 2 & 5 \end{pmatrix}$$

and b is the *column* vector on the right-hand side:

$$b = \begin{pmatrix} 6 \\ -3 \end{pmatrix}$$

The Matlab command for solving $Ax = b$ is very short:

```
x = A\b
```

The result is the column vector `[1.2632; -1.1053]`, meaning that $x_1 \approx 1.2632$ and $x_2 \approx -1.1053$.

Mathematically, x could be found as $x = A^{-1}b$ where A^{-1} is the inverse matrix. However, this is not a computationally efficient way to solve a linear system. Matlab does not actually compute the inverse matrix when solving a linear system. It chooses one of several algorithms based on the nature of the matrix; usually it is a form of "LU factorization" which is discussed in MAT 532. But even though the formula $x = A^{-1}b$ is not actually used, it suggests the notation used by Matlab. For two numbers a, b we can write ba^{-1} as `b/a`, meaning b divided by a. If the order of multiplication was important (as it is for matrices), then perhaps $a^{-1}b$ could be written as `a\b`, meaning b divided by *a from the left*. We do not actually "divide a vector by a matrix" but the notation `A\b` can remind us of $A^{-1}b$.

Example 2.1.1 Solve and verify. Write a Matlab script which solves the linear system

$$\begin{aligned} 5x_1 + 3x_2 &= 2 \\ -2x_1 + x_2 - 6x_3 &= 0 \\ 3x_2 + x_3 &= 3 \end{aligned}$$

and then plug the solution into the left hand side to check that the solution is correct.

Answer.

```
A = [5 3 0; -2 1 -6; 0 3 1];
b = [2; 0; 3];
x = A\b;
disp(x)
disp(A*x)
```

□

2.2 More tools for constructing matrices

The number of coefficients in a linear system grows rapidly with its size: a system with 20 equations and 20 variables has a matrix with 400 coefficients. Entering these by hand would be tedious. We already have two ways to construct large matrices: `zeros(m, n)` and `ones(m, n)` but these are not invertible matrices. This section presents two other useful constructions.

The **identity matrix** is usually denoted I in mathematics, or I_n if it is necessary to emphasize its size. For example,

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

In Matlab this matrix would be created as `eye(3)`. The name of the command was chosen because "eye" is pronounced the same as "I". We also get scalars multiples of the identity matrix, for example `5*eye(3)` is

$$\begin{pmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{pmatrix}$$

A **diagonal matrix** has zeros everywhere except on the main diagonal. The identity matrix I is a special case of a diagonal matrix. In Matlab, the command `diag(v)` creates a diagonal matrix which has the elements of vector `v` on its main diagonal. For example, `diag([6 -4 7])` creates the matrix

$$\begin{pmatrix} 6 & 0 & 0 \\ 0 & -4 & 0 \\ 0 & 0 & 7 \end{pmatrix}$$

Some problems in engineering and in differential equations lead to matrices where nonzero coefficients are *close* to the main diagonal but not exactly at it: for example,

$$\begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}$$

To construct such matrices we can use `diag(v, k)` which places the elements of vector `v` on the diagonal parallel to the main one but k positions above it. So, `diag(v, 1)` is just above the main diagonal and `diag(v, -1)` is just below. The matrix shown above could be formed as

```
diag([2 2 2 2 2]) - diag([1 1 1 1], 1) - diag([1 1 1 1], -1)
```

This is an example of a **tridiagonal matrix**: there are only three diagonals with nonzero entries. More generally, a matrix is called **sparse** if most of its elements are 0. Large sparse matrices frequently arise in computations.

Example 2.2.1 Product of sparse matrices. Let A be the 9×9 matrix with -1 on the main diagonal and 1 above it. Find and display the products of A with its transpose: $A^T A$ and AA^T.

Answer.
```
A = diag(ones(1, 8), 1) - eye(9);
disp(A'*A)
disp(A*A')
```

The two products are very similar but are not quite the same. □

2.3 Array operations

Matlab supports **array operations** in which the input data is placed into an array (usually a vector or a matrix) and some mathematical operation is carried out on all of the data at once. For example, to get the square roots of the first 25 positive integers we can execute

```
sqrt(1:25)
```

with the result

```
[1.0000 1.4142 1.7321 2.0000 2.2361 ... 4.8990 5.0000]
```

More generally, if `v` is a vector of nonnegative numbers, then `sqrt(v)` has the roots of those numbers. Other functions like `sin` and `exp` work the same way. The benefts of array operations are more compact code and faster execution.

Suppose `v = [5 2 -3]` and we want to get the squares of the elements of `v`. By analogy with the above, we try `v^2` but get an ***error***. The problem is that squaring `v` is understood as `v*v` which in Matlab means multiplication according to the rules of Section 1.4. The row vector `v` is treated as a matrix of size 1×3. And one cannot multiply a 1×3 matrix by another 1×3 matrix: the inner dimensions do not match. But we just want *elementwise* squaring, not any kind of matrix multiplication. To tell Matlab this, put a period `.` before the operation character: `v.^2` gives `[25 4 9]`. Similarly, `.*` and `./` are used for elementwise multiplication and division: `[3 4].*[2 -5]` is `[6 -20]` and `[3 4]./[2 -5]` is `[1.5 -0.8]`. We never need a period in front of `+` or `-` because these operations are always performed elementwise.

Summary: without a preceding period, `*`, `/` and `^` refer to what we do with matrices and vectors in linear algebra. Adding a period tells Matlab to forget about linear algebra and handle each term individually. Compare two ways to multiply the matrices

$$A = \begin{pmatrix} 3 & -2 \\ 2 & 5 \end{pmatrix}, \quad B = \begin{pmatrix} -5 & 4 \\ 1 & 3 \end{pmatrix}$$

The command

```
[3 -2; 2 5] * [-5 4; 1 3]
```

produces

$$\begin{pmatrix} -17 & 6 \\ -5 & 23 \end{pmatrix}$$

which is the matrix product according to linear algebra. The command

```
[3 -2; 2 5] .* [-5 4; 1 3]
```

produces

$$\begin{pmatrix} -15 & -8 \\ 2 & 15 \end{pmatrix}$$

which is the elementwise product: it would be considered a wrong way to multiply matrices in a linear algebra course.

Example 2.3.1 Roots of integers. Use array operations to display the numbers $n^{1/n}$ for $n = 1, \dots, 10$. What value of n has the largest such root?

Answer.

```
v = 1:10;
disp(v.^(1./v))
```

The root is largest for $n = 3$.

Note that both periods are necessary. Without the second one, Matlab would try to interpet `1/v` as a linear system with coefficients `v` and right hand side `1`, which is not what we want. With `1./v` we get the vector `[1/1 1/2 1/3 ...]`. Similarly, without the first period Matlab would try to interpret `v^(...)` as a matrix power, which would not make sense either. □

2.4 Plotting

The command `plot(x, y)` receives two vectors: the first has x-coordinates of the points to be shown, the second has y-coordinates. It also connects these points by lines. So, for example,

```
plot([1 2 5 6], [5 0.4 2 1])
```

produces the following plot:

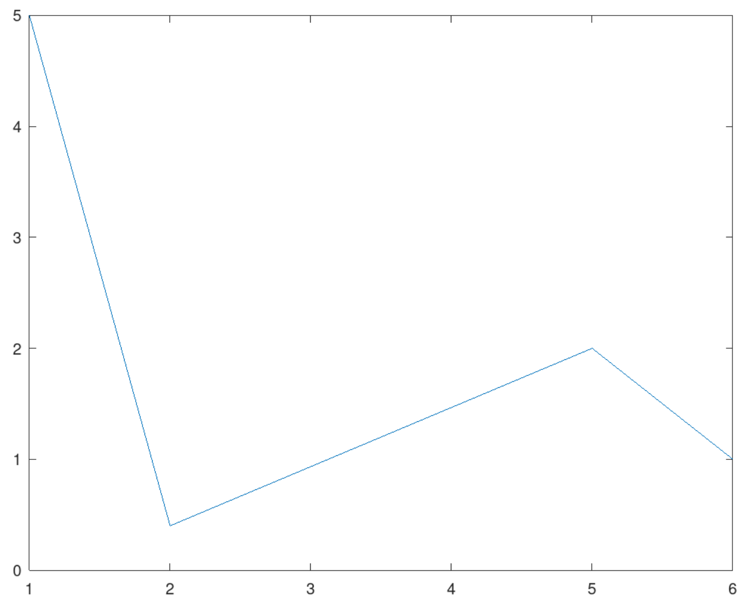

Figure 2.4.1 A plot based on four points

Sometimes we do not want to connect the points by a line: treating them as individual data points rather than representative of some function $y = f(x)$. If so,

```
plot([1 2 5 6], [5 0.4 2 1], '*')
```

simply puts an asterisk `*` for each point, without connecting them.

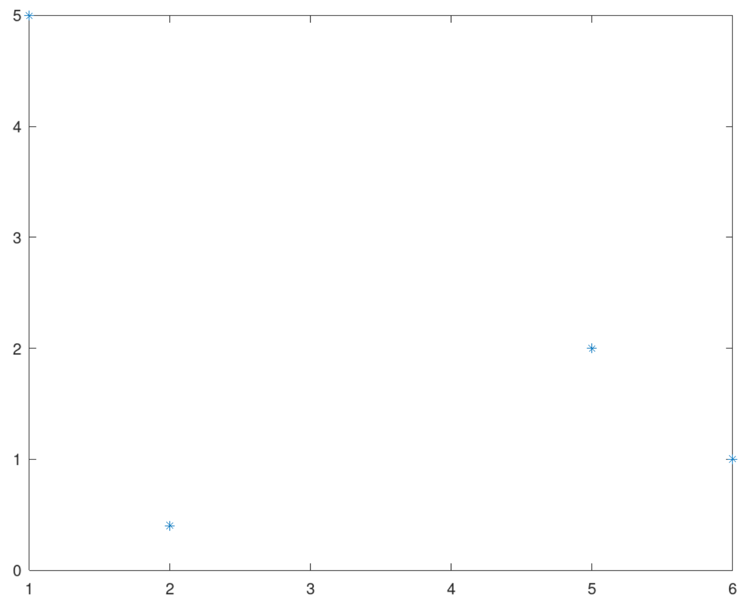

Figure 2.4.2 Four points (two are hard to see because of the axes)

Let us plot the function $y = x^3 e^{-2x}$ on the interval $[0, 3]$:

```
x = 0:0.1:3;
y = x.^3 .* exp(-2*x);
plot(x, y)
```

In the first line, I chose 0.1 as the step size for the x-interval. It would be a mistake to use `x = 0:3` because the step size of 1 unit would make a very rough (low-resolution) plot with only four points used. The step size could be 0.01 instead: we may get smoother graph at the expense of more computations. The second line compute the y-values. Consider the following:

- The period in `x.^3` is required to have elementwise power operation
- We do not need a period in `-2*x` because multiplying a vector by a scalar already works elementwise. There is no difference between `-2*x` and `-2.*x`
- We need `.*` to multiply two vectors elementwise.
- Whitespace around `.*` is not necessary but some people find `x.^3.*exp(-2*x)` harder to read because `3.` looks like a decimal dot after 3.

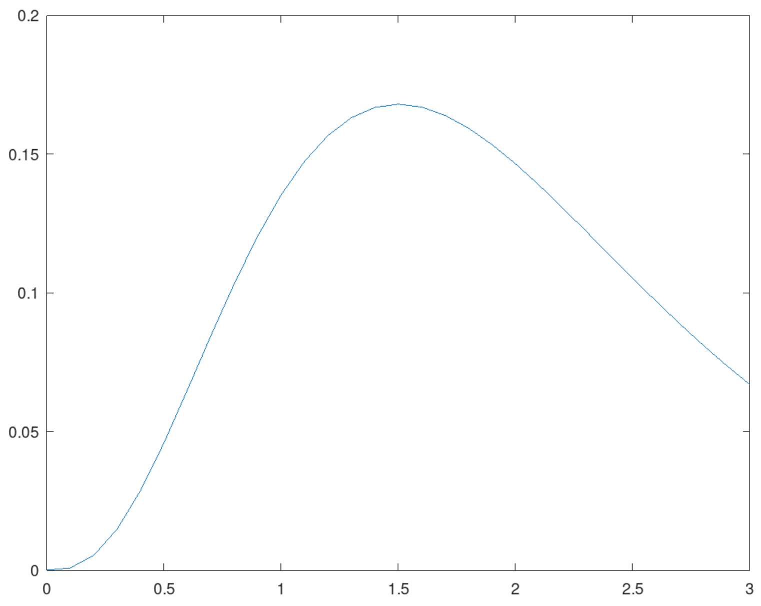

Figure 2.4.3 Plot of the function $y = x^3e^{-2x}$ on the interval $[0, 3]$

Specifying the x-values by choosing step size like in `x = 0:0.1:3` can be tedious, because the same step size would not work as well for the intervals $[2, 2.04]$ or for $[0, 50000]$. The command `linspace` can be more convenient: `linspace(a, b, n)` creates a vector with `n` uniformly spaced numbers from `a` to `b`. Usually, 500 points are enough for a smooth plot, so one can reasonably use `x = linspace(a, b, 500)` for plots no matter how small or large the interval $[a, b]$ is.

Example 2.4.4 Parametric plot. Draw the curve with parametric equations

$$x = t\cos t$$
$$y = t\sin t$$

where $0 \le t \le 20$.

Answer.
```
t = linspace(0, 20, 500);
plot(t.*cos(t), t.*sin(t))
```

This curve is called the **Archimedian spiral**. □

2.5 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 2.5.1 Tridiagonal matrix of arbitrary size. Given a positive integer n, construct a tridiagonal matrix which has entries -1, 2, -1 (as in Section 2.2) and size $n \times n$.

Answer.

```
diag(2*ones(1, n)) - diag(ones(1, n-1), 1) - diag(ones(1, n-1), -1)
```

or

```
2*diag(ones(1, n)) - diag(ones(1, n-1), 1) - diag(ones(1, n-1), -1)
```

or

```
2*eye(n) - diag(ones(1, n-1), 1) - diag(ones(1, n-1), -1)
```

It is important to recognize why `n-1` appears above. The main diagonal has length `n` but the diagonals next to it are shorter by 1 element, so their length is `n-1`. □

Example 2.5.2 Solving a tridiagonal system. Solve the system of linear equations

$$\begin{aligned}3x_1 - x_2 &= 1\\ -x_1 + 3x_2 - x_3 &= 0\\ -x_2 + 3x_3 - x_4 &= 0\\ &\cdots\\ -x_{n-2} + 3x_{n-1} - x_n &= 0\\ -x_{n-1} + 3x_n &= 2\end{aligned}$$

Use the first two digits of your SUID as the value of n.

Answer.

```
n = 28;
A = 3*eye(n) - diag(ones(1, n-1), 1) - diag(ones(1, n-1), -1);
b = zeros(n, 1);
b(1) = 1;
b(end) = 2;
x = A\b;
disp(x)
```

Explanation. The coefficient matrix has 3 on the main diagonal, and -1 above and below it. This matrix is created with the help of `diag` as explained in Section 2.2. The right-hand size column vector begins with all-zeros: note that `zeros(n, 1)` is a column of zeros, while `zeros(1, n)` is a row of zeros. The first and last entries of `b` are changed as described in the system. □

Example 2.5.3 Plot of a function. Plot the function $y = (x-1)/(x^2+1)$ on the interval $[-1, 5]$

Answer.

```
x = linspace(-1, 5, 500);
y = (x - 1)./(x.^2 + 1);
plot(x, y)
```

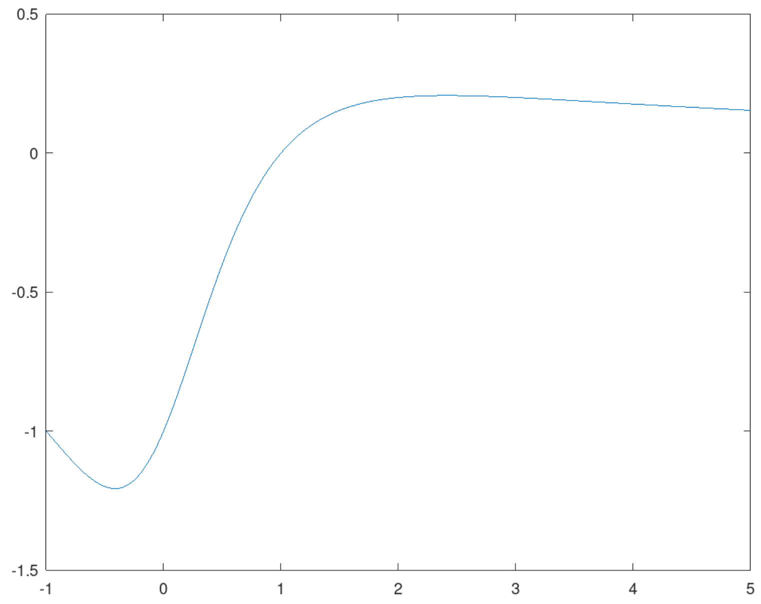

Figure 2.5.4 Plot of $y = (x-1)/(x^2+1)$ on $[-1, 5]$

Note the use of semicolons to prevent Matlab from displaying hundreds of numbers that we do not need to see. □

Question 2.5.5 x as a function of y. How would you plot a function where x is given as a function of y, for example $x = y + \cos y$ with $-5 \le y \le 5$? □

Question 2.5.6 A "wrong" way to multiply matrices. Can you imagine a situation where you would want to compute `A.*B` where both `A` and `B` are square matrices of the same size?

An idea: a matrix can be used to store a greyscale image, so that each entry is the brightness of the corresponding pixel. What would elementwise multiplication `A.*B` mean in this context? □

2.6 Homework

1. Consider the following relations: $F_0 = 1$, $F_1 = 1$, $F_2 = F_0 + F_1$, $F_3 = F_1 + F_2$, and so on until $F_6 = F_4 + F_5$. (These are the first 7 Fibonacci numbers).

(a) Write down a linear system that expresses these relations in matrix form $Af = b$. Here, A will be a 7×7 matrix and b a column vector.

(b) Write a Matlab script that creates this matrix A and vector b, then solves the linear system to find the Fibonacci numbers.

2. Write a script to do the following:

(a) Create a 10×10 matrix A with 2 on the main diagonal, and -1 next to it (both above and below).

(b) Solve the linear system $Ay = b$ where b is the vector

$$\begin{pmatrix} \sqrt{1} \\ \sqrt{2} \\ \vdots \\ \sqrt{10} \end{pmatrix}$$

Use a regularly-spaced vector, transpose, and `sqrt` to create b.

(c) Plot the solution y against the vector x of 10 equally spaced entries, beginning with 0 and ending with 1. The `linspace` command will be useful.

Chapter 3

Built-in functions, input, output

We take a look at some of mathematical and statistical functions available in Matlab, and consider the ways for user input and data output.

3.1 Built-in functions

Mathematical functions in Matlab have recognizable names: `sqrt`, `log` (meaning natural logarithm, base e), `exp`, `abs` (absolute value), `sin`, `cos` and so on. They can be applied to vectors or matrices elementwise: for example `exp([1 2 3])` returns a vector with components e^1, e^2, e^3.

There are also **aggregate functions** which are useful for summarizing data. The most important of these are `sum` (sum of values), `max` (maximum) and `min` (minimum). When applied to a vector, these functions return a single number. So, if `v = [3 1 4 1]` then `sum(v)` is 9, `max(v)` is 4, and `min(v)` is 1.

Example 3.1.1 Sum of cubes. Find the sum of cubes of the integers from 1 to 10.

Answer.

```
sum((1:10).^3)
```

Note that the parentheses are important here: `sum(1:10.^3)` would be the sum of all integers from 1 to 1000. □

Aggregate functions get more complicated (and powerful) when they are applied to matrices. In this case they *do not* return a single number. Given a matrix A, for example `A = [3 9; 5 2]`, we can find its:

- **Column sums** with `sum(A, 1)`: the result is a row vector `[8 11]` which contains the sum of each column.
- **Row sums** with `sum(A, 2)`: the result is a column vector `[12; 7]` which contains the sum of each row.

The second argument (1 or 2) means that we sum along the 1st index (row index) or 2nd index (column index). Summing along the row index means A(1, 1) + A(2, 1) + A(3, 1) + ... which produces the sum within each column. Entering `sum(A)` has the same effect as `sum(A, 1)`.

Stacking the columns of a matrix. When we want the sum (or another aggregate function) of all entries of matrix A, we can **stack** the columns of A using a single colon: `A(:)`. Then an aggregate function can be applied to this column. For example, if `A = [1 2; 3 4]` then `A(:)` is the column `[1; 3; 2; 4]` and `sum(A(:))` is 10.

Example 3.1.2 Maximum and minimum of row sums. Given a matrix, such as `A = [3 6; -8 4; 10 1]` find its "maximal absolute row sum", meaning add up the absolute values of the elements within each row, and take the maximum of the results.

Answer.

```
max(sum(abs(A), 2))
```

The result is 12, coming from the second row. The maximal absolute row sum is important in applied linear algebra. It is one of several **norms** that can be defined for a matrix. □

3.2 Random numbers and histograms

The command `rand` produces random numbers uniformly distributed in the interval from 0 to 1. Without providing parameters, we get a single number: `x = rand()` can make x equal to, for example, 0.8248. With parameters `rand(m, n)` we get a random m×n matrix, meaning a matrix in which each entry is chosen randomly.

The command `hist(v, k)` displays a histograph of the data collected in vector `v` using `k` bins. This means that the interval from `min(v)` to `max(v)` is divided into `k` equal subintervals, and the histogram counts how many data points fall in each subinterval. If the number of bins `k` is not provided, Matlab will try to choose it itself. Usually we want 10-50 bins. For example,

```
v = rand(1, 10000);
hist(v, 20);
```

produces a histogram from 10000 random numbers. This will not be an interesting histogram since the numbers are uniformly distributed between 0 and 1.

Technical remark: Matlab recommends a newer command `histogram` instead of `hist` but they both work for our purposes. I am using `hist` because it also works in Octave.

Example 3.2.1 Distribution of sums of random numbers. Suppose we take 10 random numbers, uniformly distributed between 0 and 1, and compute their sum. Repeating this experiment 10000 times, we get 10000 such sums. Plot the histogram of these random sums.

Answer.

```
A = rand(10, 10000);
S = sum(A, 1);
hist(S, 30);
```

This histogram resembles a "bell curve" of normal distribution. The similarly grows stronger if we take sums of more numbers, and plot more random sums. This is the content of **Central Limit Theorem** in probability. □

3.3 User input

When running numerical experiments like those in Section 3.2, it is inconvenient to edit the script every time we want to change the numbers in it, like the number of rows/columns of a matrix. One can use user input instead, making the script interactive:

```
m = input('Number of rows: ');
n = input('Number of columns: ');
```

The text in `input` command will be shown to the user as a prompt. When the user enters a number, that number gets assigned to the variable `m`. Same happens with `n`. After that the computation proceeds.

Example 3.3.1 Sums of random numbers with user input. Modify Example 3.2.1 so that instead of fixed numbers 10 and 10000, it takes user input for those numbers.

Answer.

```
m = input('How many numbers in a sum? ');
n = input('How many sums to compute? ');
A = rand(m, n);
S = sum(A, 1);
hist(S, 30)
```

□

3.4 Formatted output

We have been using `disp` command to display some number or text. But we may want to display a sentence with some numbers included in it. And we may want to show more (or fewer) digits than `disp` does. For this, the command `fprintf` is useful: it produces an **interpolated string** which combines text and numeric data according to some formatting codes. For example,

```
fprintf('The number pi is %.15f and its cube is %.3f\n', pi, pi^3)
```

displays “The number pi is 3.141592653589793 and its cube is 31.006”. Explanation:

- The first argument of `fprintf` is a string, the other arguments are numbers which will be inserted into the string. Here the numbers are pi and pi cubed.
- The formatting code `%.15f` says: insert a number with 15 digits after decimal point. This is the first formatting code, so it uses the first number provided after the string, which is pi.
- The formatting code `%.3f` says: insert a number with 3 digits after decimal point. Since this is the second formatting code in the string, it will use the second of the numbers after the string, which is pi cubed.
- The string ends with new line character `\n`, which is a good idea in general. Without it, subsequent output would appear on the same line, and would be hard to read.

Note that 15-16 decimal digits is about the limit of precision in Matlab computations. If we try `fprintf('The number pi is %.20f\n', pi)` the output is “The number pi is 3.14159265358979310000” which looks suspicious, and for a good reason: the zeros at the end are not correct digits of pi.

For very large or very small numbers we need exponential notation. For example, if you ask Matlab to compute `pi^100` it will display the answer as `5.1878e+49`. This means $5.1878 \cdot 10^{49}$. The number after the letter e is the power of 10 by which we should multiply the number before “e”. The letter “e” stands for “exponential” not for the mathematical constant e. To use exponential notation in formatted strings, replace letter `f` in formatting codes by `e`. To remember these, note that `f` means Fixed decimal point, and `e` means Exponential notation.

```
        fprintf('The 100th power of pi is %.6e\n', pi^100)
```

displays the result as “The 100th power of pi is 5.187848e+49”.

Often it is easiest to the formatting code `g` which lets Matlab decide between fixed point and exponential notation on its own, based on the size of the number. Here is an example.

```
x = 111.111
fprintf('The first four powers of %.6g are %.6g, %.6g, %.6g, %.6g\n', x, x, x^2, x^3, x^4)
```

The output is “The first four powers of 111.111 are 111.111, 12345.7, 1.37174e+06, 1.52415e+08”

There is also a formatting code `%d` to be used for integers, when you do not want a decimal dot at all.

3.5 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking “Answer”. There are also questions for reviewing the concepts of this section.

Example 3.5.1 Random matrix with entries between two given numbers. Given numbers a, b with $a < b$, construct a random 5×5 matrix in which the entries are uniformly distributed between a and b.

Answer.

```
A = (b-a)*rand(5, 5) + a;
```

Explanation: we take random numbers between 0 and 1, and then apply a linear function which sends 0 to a, and 1 to b. This function is $y = (b - a)x + a$. □

Example 3.5.2 Mean and median. The functions `mean` and `median` are aggregate functions which work similarly to `sum`. For example, `mean([1 9 8 3])` is 5.25 which is $(1 + 9 + 8 + 3)/4$. And `median([1 9 8 3])` is 5.5 because when these numbers are sorted as 1, 3, 8, 9 there are two in the middle, and their average is $(3 + 8)/2 = 5.5$. (When the number of data values is odd, the median is the number in the middle.)

Plot a histogram of the mean of 10 random numbers, with 10000 such means computed.

Answer.

```
A = rand(10, 10000);
m = mean(A, 1);
hist(m, 30)
```

Explanation. Having created a random matrix A, we take the mean of each column with `mean(A, 1)`. If we used `median(A, 1)` the result would be a similar histogram. □

Example 3.5.3 Difference between mean and median. For 10 randomly chosen numbers (with `rand`) we can compute both mean and median. How different can these be? That is, how large can the difference |mean-median| be? Try to approach this experimentally by computing this difference 10000 times and taking the maximum.

Answer.

```
A = rand(10, 10000);
m = mean(A, 1);
md = median(A, 1);
disp(max(abs(m-md)))
```

The result of computation depends on your luck. There is no guarantee that the maximum found after 10000 random tries is close to the actual maximum. □

Question 3.5.4 Mean and median for a 0-1 vector. Suppose `v` is a vector with 10 elements, in which 6 elements are 0 and the others are 1. What are the mean and the median of `v`? (You should not need a computer for this.) Is the difference between the mean and the median in this example greater than what you found experimentally in Example 3.5.3? □

Question 3.5.5 Squaring a random number. We know that `x = rand()` produces a random number between 0 and 1. The command `x = rand()^2` also produces a random number between 0 and 1. Is there a significant difference between these two ways of creating random numbers? □

3.6 Homework

1. Write a script that does the following:

(a) Asks the user to enter a number `n` that will be used for the size of a square matrix.

(b) Creates a random n×n matrix A with entries distributed between -1 and 1.

(c) Computes the matrix $B = A^2$.

(d) Shows a histogram of all entries of B, with 30 buckets.

2. Write a script that does the following:

(a) Asks the user for the number of rows and the number of columns, and creates a random matrix A of this size (with entries uniformly distributed between 0 and 1).

(b) Multiplies A by a random column vector x with entries uniformly distributed between 0 and 1.

(c) Shows the histogram of Ax, with 30 buckets.

(d) Displays formatted output: "The smallest entry of Ax is ... and the largest entry of Ax is ...", with 9 digits after decimal dot in each of them.

Chapter 4

Control flow: loops and conditional statements

Loops and conditional statements in Matlab are similar to those in other imperative programming languages. But in Matlab, array operations (Section 2.3) can often replace loops, and provide faster and shorter code. However, loops are sometimes unavoidable when the each computation depends on the result of the previous computation.

4.1 `for` loop

Syntax of `for` loop:

```
for (variable) = (some vector, usually regularly spaced a:h:b)
    (some computation)
end
```

This means that "some computation" will be repeated for each entry of the vector specified in the line with `for`. Do not forget to `end` the loop: Matlab needs it to know which part of the code should be repeated in the loop. Indenting the body of the loop is optional but it improves readability.

Example 4.1.1 Sum of cubes. Find the sum of cubes of the integers from 1 to 10 using a `for` loop.

Answer.

```
s = 0;
for k = 1:10
    s = s + k^3;
end
```

Explanation: before entering the loop, we initialize the variable `s` with 0, meaning that the value of the sum is 0 before we start adding things to it. Each time the command `s = s + k^3` is executed, the cube of `k` is added to `s`. This repeats for each value in the vector `1:10`. When the loop ends, the variable `s` will contain the sum of cubes.

This example is for illustration only: in practice the array operation `sum((1:10).^3)` would be a better choice because it is shorter and possibly more efficient than the loop.

One could use a more descriptive variable name instead of `s`, for example `total`. But do not use `sum` because this is the name of a built-in Matlab function. Same goes for `max` and `min`: since they are names of built-in functions, they should not be used as variable names. Also, although mathematicians often use i as index variable, it is too easy to confuse with digit 1 and so is best avoided. □

A loop is necessary for **iterative** computations, when every step of computation uses the result of the previous one. Example 4.1.1 is not like this, since the cubes of 1, 2, 3,... can be computed independently of each other: we do not need the cube of 5 in order to compute the cube of 6. In contrast, the computation of arithmetic-geometric mean in Example 1.2.1 is iterative: we compute the arithmetic and geometric means, and then use them as an input for next computation.

Example 4.1.2 Arithmetic-geometric mean using a `for` loop. Find the arithmetic-geometric mean (AGM) of 11 and 31 using a `for` loop.

Answer.

```
x = 11;
y = 31;
for k = 1:10
    a = (x+y)/2;
    g = sqrt(x*y);
    x = a;
    y = g;
end
disp(x)
```

It does not matter whether we display `x` or `y` at the end because they become nearly equal after 10 iterations. The output is the number 19.7127, which we can check for correctness using, for example, Wolfram Alpha: agm(11,31).

The logic of the script: compute arithmetic and geometric means of x and y, then use them as our new values of x and y. One may be tempted to shorten the loop as follows

```
for k = 1:10
    x = (x+y)/2;
    y = sqrt(x*y);
end
```

But this loop produces a wrong answer: 23.9912. The reason this script is wrong is that it changes x before the computation of geometric mean. The definition of AGM requires both arithmetic and geometric means to be computed from the same pair x, y. Only after both means are computed, we can replace x and y with new values. □

4.2 `while` loop

In Example 4.1.2 we used 10 iterations of arithmetic and geometric means computation. This was an arbitrary number. For some input values it may be enough or too many, for others not enough. A better approach is to stop the computation when the required precision is achieved. If we want the result to be accurate within 10^{-9}, which is `1e-9` in Matlab notation, then we can stop as soon as the difference between the two means has absolute value less than that number (called the **tolerance**). This is what `while` loop can be used for.

Syntax of `while` loop:

```
while (some condition)
    (some computation)
end
```

The loop will repeat while condition after `while` is true. It ends when this condition becomes false. So, if we want to stop when `abs(x-y) < 1e-9`, the loop should be written as `while abs(x-y) >= 1e-9`.

Example 4.2.1 Arithmetic-geometric mean using a `while` loop. Find the arithmetic-geometric mean (AGM) of two numbers given by a user, with `1e-9` accuracy.

Answer.

```
x = input('x = ');
y = input('y = ');
while abs(x-y) >= 1e-9
    a = (x+y)/2;
    g = sqrt(x*y);
    x = a;
    y = g;
end
fprintf('AGM(x, y) = %.9f\n', x)
```

□

There are two common issues with using `while` loops. One is that it can go on forever, if the algorithm fails to reach its goal (the difference never becomes smaller than the tolerance). It is better to admit that the algorithm failed than to be stuck in an infinite loop: we will deal with this in Section 4.3. For now, note that if your program gets stuck in an infinite loop, you can terminate it by clicking within the Command Window and pressing Ctrl+C.

Another issue is that the condition we want to test may not be defined at the beginning of the loop. This occurs when our condition for stopping is that the value of a variable essentially stopped changing. For example, one way to compute the **golden ratio** numerically is to start with any $x > 0$ and keep replacing x with $\sqrt{x+1}$. If we write this as:

```
x = 1;
while abs(newx - x) >= 1e-9
    newx = sqrt(x+1);
    x = newx;
end
fprintf('Golden ratio = %.9f\n', x)
```

the result will be an error: "Undefined function or variable 'newx'" because we attempt to compute an expression with `newx` before it is defined. Here is one way to correct this.

```
x = 1;
dx = 1;
while abs(dx) >= 1e-9
    newx = sqrt(x+1);
    dx = newx - x;
    x = newx;
end
fprintf('Golden ratio = %.9f\n', x)
```

Here we introduce a variable `dx` that represents the difference between old and new values of `x`. This variable can be initially assigned some arbitrary value that is greater than the tolerance, for example 1 or 100. This ensures that the condition `abs(dx) >= 1e-9` holds at the beginning of the loop. After that the loop changes the value of `dx` and the process repeats until it becomes small enough.

4.3 `if` statement

A `if` statement controls the flow of the program by executing one part of it if a certain condition is true (and possibly, executing another if the condition is false). The syntax is

```
if (condition)
    (what to do if the condition is true)
else
    (what to do if the condition is false)
end
```

The part with `else` is optional if you don't want to do anything special when the condition is false. The `end` is required, as with loops. Here is a simple script that tells the user if the number they entered is positive or negative.

```
x = input('x = ');
if x < 0
    disp('negative')
else
    disp('positive')
end
```

This code has a bug: if the user enters 0, the output is 'positive'. To correct this, we need to put another `if` in the `else` clause. This is done with `elseif`:

```
x = input('x = ');
if x < 0
    disp('negative')
elseif x > 0
    disp('positive')
else
    disp('zero')
end
```

Generally, conditions in `while` and `if` are of the form x (relation) y where the relation can be

- < less than
- > greater than
- <= less than or equal
- >= greater than or equal
- == equal
- ~= not equal

Warning: `if (x=y)` is incorrect, because a single = sign means *assignment* (make x equal to y), not equality (is x equal to y?). Also, we rarely need to test equality because it can fail because of tiny errors in computer arithmetics with non-integer numbers.

Example 4.3.1 Testing the equality of two numbers. Write an `if` statement that tests whether $0.1 + 0.2$ is equal to 0.3.

Answer.

```
if 0.1 + 0.2 == 0.3
    disp('equal')
else
    disp('not equal')
end
```

Running this code you will see a surprising result: "not equal". We will consider this later. The point to be taken here is that testing for equality is practical only when both sides of `==` are integers. □

4.4 Breaking out of a loop

In Section 4.2 we noted an issue with `while` loop: it can go on forever, or for such a long time that we simply have to give up on the computation and maybe try something else. The command `break` exits the loop (it can be either `for` or `while` loop). This command is always contained in an `if` statement because the idea is to break out of the loop when some condition is reached. Here is an example of such situation. Starting with `x=1`, the code repeatedly replaces x with $\frac{1}{2}(x + a/x)$ where the number `a` is provided by the user.

```
a = input('a = ');
tries = 1000;
x = 1;
for k = 1:tries
    newx = (x + a/x)/2;
    if abs(newx - x) < 1e-9
        break
    end
    x = newx;
end

if k < tries
    fprintf('Converged to x = %.9f\n', newx)
else
    disp('Failed to converge')
end
```

The loop has a limit of 1000 tries. It ends sooner if the required condition "new x is close to old one" is reached. After the loop there is another conditional statement which reports the result depending on whether the loop terminated early (success) or ran out of tries (failure). Try running this script with different inputs such as 2, -3, 4, -5, 6. What do you observe?

4.5 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking “Answer”. There are also questions for reviewing the concepts of this section.

Example 4.5.1 Partial sums of the harmonic series. The harmonic series $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots$ diverges, meaning that its partial sums grow without a bound. So, if we add enough terms of this series, we will get a number greater than 10. Use a `while` loop to determine how many terms are required to get a partial sum that is greater than 10.

Answer.

```
s = 0;
n = 1;
while s <= 10
    s = s + 1/n;
    n = n + 1;
end
fprintf('The partial sum has %d terms\n', n-1)
```

Explanation: The variable `s` represents partial sums. We add `1/n` to it, and then increase the value of `n`. When `s > 10` the process stops. The output uses formatting code `%d` for integers, since the answer is an integer. Why does the output have `n-1` and not `n`? Because of the way that the loop is written, the value of n is increased *after* each addition. So at the end of it, `n` is the number that is 1 more than the number of terms we added. □

Example 4.5.2 The Collatz conjecture. The Collatz conjecture, also known as the **Syracuse problem**, is a famous unsolved problem in mathematics. Its subject is the following process.

1. Start with a positive integer k.
2. If k is even, divide it by 2. Otherwise multiply it by 3 and add 1.
3. Repeat Step 2.

The conjecture is that this process always reaches the number 1 (after 1 it enters infinite loop 1, 4, 2, 1, 4, 2, ...) Write a script that verify this conjecture for initial values of k from 1 to 500, and for each of them record the number of steps that it takes to reach 1. Then plot the number of steps. Note: the condition that k is even can be tested as `if mod(k, 2) == 0` where `mod(k, 2)` computes the remained of dividing k by 2.

Answer.

```
N = 500;
steps = zeros(1, N);
for n = 1:N
    count = 0;
    k = n;
    while k > 1
        if mod(k, 2) == 0
            k = k/2;
        else
            k = 3*k+1;
        end
        count = count + 1;
    end
    steps(n) = count;
end
plot(1:N, steps, '.')
```

Explanation. The array `steps` is created as a bunch of zeros, to be replaced with actual counts of steps. The initial value of Collatz process is called `n` and it runs through the numbers from 1 to 500. So the variable `k` is initially assigned the value `n` and then modified according to the rules stated above. Every time it is modified, the `count` of steps is increased by 1. The plot at the end shows each data point as an individual dot, not connecting them by lines.

This example demonstrates the use of both kinds of loops. Note that the variable involved in `while` loop condition, `while k > 1`, should be modified within the loop, otherwise it will never end. In contrast, the variable involved in `for` loop condition `for n = 1:N` should *not* be modified since it is automatically incremented with each run of the loop. □

Question 4.5.3 Avoiding an intermediate variable. If in Example 4.1.2 we did not introduce the variable `g` and simply wrote `y = sqrt(x*y)`, would the result still be correct? Why or why not? □

Question 4.5.4 Changing the `for` loop variable. What exactly would go wrong in Example 4.5.2 if we did not introduce `k` and simply wrote the same `while` loop with `n`, as below?

```
while n > 1
    if mod(n, 2) == 0
        n = n/2;
    else
        n = 3*n+1;
    end
    count = count + 1;
end
```

□

4.6 Homework

1. Explain the output of the script in Section 4.4 mathematically. What does it converge to, and why does it sometimes fail to converge? (This problem does not involve Matlab.)
2. If in Example 4.5.1 we swap two lines `s = s + 1/n;` and `n = n + 1;` the answer becomes very different. Explain why this happens.
3. Write a script that computes the **arithmetic-harmonic mean** which means is similar to Example 4.2.1 except that instead of geometric mean `g = sqrt(x*y);` we use the harmonic mean `h = 2*x*y/(x+y)`. As a testing case, check that the arithmetic-harmonic mean of 5 and 20 is 10.

   Since there is no square root in this computation, we can use negative numbers too: the arithmetic-harmonic mean of -5 and -20 is -10. But if you start with two numbers of opposite signs, the loop never ends, so you will need to press Ctrl+C to end it. As a second part of this task, write another form of the loop, using the method from Section 4.4 to put a limit on the number of tries, for example 1000.

Chapter 5

User-defined functions

In addition to the functions built into Matlab, one can define new ones. There are two ways to introduce new functions: “anonymous functions” and “named functions”. The use of anonymous functions is similar to mathematical functions like `log`. The use of named functions has more to do with program structure than with mathematics.

5.1 Anonymous functions

The concept of an **anonymous function** in Matlab closely corresponds to mathematical notion of a function: it consists of a formula that transforms input into output. For example, to have mathematical function $f(x) = \exp(-x^2)$ in your code, you would write

```
f = @(x) exp(-x^2);
```

and then use `f` in expressions such as `f(0)` or `3*f(x-2)`.

When writing a function, one should consider whether it should accept input in the form of a vector or a matrix. The function in the previous paragraph works fine with scalar input, such as `f(2)` but throws an error if the input is a vector, for example `f([0 1 2 3])`. The reason is that with this input, `x^2` becomes `[0 1 2 3]^2` which is an error in Matlab where `^2` means matrix multiplication. We want an array operation, not a matrix product. Thus, the function should be written as

```
f = @(x) exp(-x.^2);
```

Then we can apply this function to a vector, for example to plot it.

```
x = linspace(-3, 3, 500);
plot(x, f(x))
```

An anonymous function can have multiple variables, for example the mathematical formula $f(x, y, z) = (xy+1)/z$ could be defined as `f = @(x,y,z) (x.*y + 1)./z`. The inclusion of periods allows this function to vectors for its inputs, provided the vectors are of the same size.

The names of the parameters (variables inside of anonymous function) do not matter outside of that function. For example, there is no problem with the following code

```
f = @(x) x + 4;
x = 1;
y = 3;
disp(f(y))
```

The result is 7, because this is the output that `f` produces when its input is 3. The variable `x` outside of the function (which was given the value of 1) has no relation to the variable `x` inside of the function.

A limitation of anonymous functions is that they consist of a single expression: there is no place for conditional statements or loops. But one can still construct piecewise defined functions using inequalities as a part of an arithmetic expression. When an inequality such as `(x < 3)` appears in an expression, it is treated as 1 when the inequality is true, and 0 when false. Therefore, the function defined by

```
f = @(x) (x>=0).*sqrt(x) + (x < 0).*exp(x)
```

evaluates to $\sqrt{x}$ when $x \geq 0$ and to e^x when $x < 0$.

Example 5.1.1 Sum of reciprocals. Write an anonymous function which, when given a number n, returns the sum of the reciprocals of all positive integers between 1 and n.

Answer.

```
H = @(n) sum((1:n).^(-1));
```

These sums are called **harmonic numbers** (hence the letter H for this function). What happens if it is given a negative or fractional input? The regular vector `1:n` contains all positive integers that are not greater than n. So, `1:3.7` is `[1 2 3]`, same as `1:3`. This means `H(3.7)` is `1/1 + 1/2 + 1/3`, same as `H(3)`. And if a number less than 1 is given, the regular vector `1:n` is empty, and the sum of an empty set of numbers is 0. For example, `H(-2)` is 0. □

5.2 Named functions

Named functions can do the same thing as anonymous functions. Here is $\exp(-x^2)$ again, this time as a named function.

```
function y = f(x)
    y = exp(-x.^2);
end
```

But there may be multiple lines within the body of a named function, including loops and conditional statements. In general, it looks like this.

```
function y = myFunction(x)
    (do something);
    (do more things);
    (y = result of computations);
end
```

A named functions can have multiple inputs: `function w = myFunction(x, y, z)`. It can also have multiple outputs: `function [u, v] = myFunction(x, y, z)` in which case both `u` and `v` must be assigned some values within the function.

Example 5.2.1 Arithmetic-geometric mean with given tolerance. Write a named function with three inputs: numbers `x, y` and `tol`, which finds the arithmetic-geometric mean of `a, b` with tolerance `tol` (meaning that the process stops when the difference of arithmetic and geometric mean is less than `tol`.)

Answer.

```
function z = agm(x, y, tol)
   while abs(x - y) >= tol
       a = (x + y)/2;
       g = sqrt(x*y);
       x = a;
       y = g;
   end
   z = x;
end
```

Note that the result has to be assigned to whichever function variable was designated as output on th first line of the function: in this case, `z`. □

Example 5.2.1 illustrates an important point: a function has limited communication with the code outside of it. It receives input and produces certain output; nothing else that it does has any influence on the outside world. Suppose that the following sequence of commands was executed.

```
x = 0.3;
y = 12;
disp(agm(x, y, 1e-6))
disp(x)
disp(a)
```

The third line will display the AGM which happens to be 3.7136. The fourth line will display the original value of x, which is `0.3`. Why is it still 0.3, even though the function `agm` changed the value of x in the loop? This is because a function operates like a separate program with its own variables. The variable x inside of the function is its own *local* variable x, and changing it does not change x in other places. For the same reason, the last command `disp(a)` produces an error: the variable `a` is undefined. A variable with that name was introduces inside of function `agm` but it does not exist outside of it.

One can have a function that does not return any values. Here is a function whose job is to plot the sine function on the interval $[a, b]$ using a given number of points.

```
function plotsin(a, b, num)
    x = linspace(a, b, num);
    plot(x, sin(x))
end
```

One can also have a function with no arguments (no input values). For example, this function computes an approximate value of the Golden Ratio by iterating $\sqrt{x+1}$.

```
function z = goldenratio()
   z = 1;
   for k = 1:100
       z = sqrt(z+1);
   end
end
```

One can use this function same as others: `x = goldenratio();` or `disp(goldenratio())`.

Back in the days when Matlab was designed, the convention was to put every function in a separate "M-file" named as that function: for example, `goldenratio.m`. Today, tying the name of a function to the name of a file in which it is contained seems strange but Matlab still insists on this. Fortunately, it allows us to have multiple functions in the same script file, with the following constraints:

- The first function is named as the file itself. This function gets executed when we click Run or press F5.
- If a script contains named functions, all of its code must be within some function.

This means that if a script contains named functions, and is saved in a file named `hw5.m`, it must begin with `function hw5()`, for example:

```
function hw5()
    x = 0.3;
    y = 12;
    disp(agm(x, y, 1e-6))
end

function z = agm(x, y, tol)
   while abs(x - y) >= tol
       a = (x + y)/2;
       g = sqrt(x*y);
       x = a;
       y = g;
   end
   z = x;
end
```

Here the first (main) function is executed: it assigns values to x and y, and then calls another function to compute their AGM.

Anonymous functions, described in Section 5.1, have no such restrictions.

Example 5.2.2 Smallest prime that is greater than a given number. Write a function which, given a value `n`, finds the smallest prime number that is greater than `n`.

To determine whether some number `k` is prime, use `isprime(k)`. For example, `while isprime(k)` is a loop that runs as long as `k` stays prime. To have a loop that runs while `k` is *not* prime, use `while ~sprime(k)`. The character `~` means logical NOT in Matlab (recall that `~=` means "not equal".)

Answer.

```
function k = smallestprime(n)
    k = n + 1;
    while ~isprime(k)
        k = k + 1;
    end
end
```

For example, `smallestprime(31)` is 37, because the numbers 32, 33, 34, 35, 36 are not prime. □

5.3 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer".

There are also questions for reviewing the concepts of this section.

Example 5.3.1 Mean value of sine function. Define an anonymous function `meansin = @(n) ...` which computes the mean value of sin(1), sin(2), ..., sin(n). Use it to display the values `meansin(n)` for n equal to 10, 50, 250, 1870, and 5000. What pattern do you observe?

Answer.

```
meansin = @(n) mean(sin(1:n));

for n = [10 50 250 1870 5000]
    disp(meansin(n))
end
```

Explanation: this is a reminder that the vector in `for` loop does not have to be of the form `a:b` even though this form is most common. Here the use of a loop avoids having to write

```
disp(meansin(10))
disp(meansin(50))
disp(meansin(250))
```

and so on.

It appears that the mean values approach 0 as `n` grows. This can be proved mathematically: see the homework section. □

Example 5.3.2 Evaluating a polynomial. Define a named function `function y = p(x, a, n)` which computes the polynomial

$$p(x, a, n) = 1 + \frac{x}{a} + \frac{x^2}{a+1} + \frac{x^3}{a+2} + \cdots + \frac{x^n}{a+n-1}$$

Assume that `x` may be a vector, `a` is a real number, and `n` is a positive integer (the degree of our polynomial).

Answer.

```
function y = p(x, a, n)
    y = 1;
    for k = 1:n
        y = y + x.^k/(a+k-1);
    end
end
```

One could also start with `y = 0` and run the loop `for k=0:n`; the result is the same. □

Example 5.3.3 Plotting a polynomial. Use the function from Example 5.3.2 to plot the polynomial with $a = -3/2$ and $n = 6$ on the interval $[-1, 1]$. Assume the script is in the file `example.m`.

Answer.

```
function example()
    x = linspace(-1, 1, 500);
    plot(x, p(x, -3/2, 6))
end

function y = p(x, a, n)
```

```
    y = 1;
    for k = 1:n
        y = y + x.^k/(a+k-1);
    end
end
```

□

Question 5.3.4 Function creating a vector. What does the function `f` defined by `f = @(x, y) [x y]` do? Can one give it vector inputs `x, y`. If so, does it matter if they are row vectors or column vectors? Do they need to be of the same size? □

5.4 Homework

1. (No programming involved.) Prove that the following inequality holds for every positive integer n:

$$|\sin(1) + \sin(2) + \cdots + \sin(n)| \le 2$$

Why does this explain the observation in Example 5.3.1?

Hint. Multiply the sum of sines by $\sin(1/2)$ and use the formula

$$\sin(x)\sin(y) = \frac{1}{2}(\cos(x-y) - \cos(x+y))$$

to make the sum "telescope".

2. Write an anonymous function `f = @(x, y) ...` which implements the mathematical function

$$f(x,y) = \frac{xy^2}{x^2+y^4}$$

in such a way that the input variables x, y are allowed to be vectors. Then plot the values of f along the parametric curve

$$x = t^2$$
$$y = \sin t$$

as follows.

```
t = linspace(-1, 1, 500);
plot(t, f(t.^2, sin(t)))
```

As t approaches 0, what value does f appear to approach?

3. Write a named function `function y = cositer(x, n)` that takes two arguments x and n and returns the n-th iteration of the mathematical function $x \mapsto 2\cos x$. For example, if `n = 4`, the result should be

$$2\cos(2\cos(2\cos(2\cos x)))$$

Use this function to plot several of these iterates on the interval $[0, 2]$ as shown below.

```
function hw5()
    x = linspace(0, 2, 2000);
    hold on
    for n = [4 5 30 31]
        plot(x, cositer(x, n))
    end
    hold off
end

function y = cositer(x, n)
   (your function goes here)
end
```

Remark: the command `hold on` refers to displaying several functions on the same plot. Normally, Matlab replaces each plot with next one. With `hold on`, it keeps the previous plot and draws next one over it, using a different color (unless you specify a color). At the end, `hold off` is used to restore the normal behavior.

Part II

Solving Equations

Chapter 6

Systems of linear equations

We study how Matlab treats various types of linear systems, including systems with multiple solutions and systems with no solutions. We are not going to study the algorithms used by Matlab to solve linear systems; this is done in MAT 532. Our goal is to interpret the results.

6.1 Classification of linear systems

A system of m linear equations with n unknowns can be written as $Ax = b$ where A is a $m \times n$ matrix, x is an unknown column vector with n coordinates, and b is a known column vector with m coordinates. Linear systems are classified by consistency and shape.

A linear system is called **consistent** if it has at least one solution. Otherwise it is **inconsistent**. So, in terms of solutions there are three possibilities:

1. consistent with a unique solution
2. consistent with infinitely many solutions
3. inconsistent

These possibilities are easy to visualize in terms of 2×2 systems:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}$$

Each of these equations determines a line in the plane. In Case 1 the two lines intersect; in Case 2 they are the same line; in Case 3 they are parallel.

In terms of shape, a linear system is **square** if it has as many equations as unknowns, $m = n$. It is called **underdetermined** there are fewer equations than variables: $m < n$, matrix has more columns than rows. A system is **overdetermined** if it has more equations than variables: $m > n$, matrix has more rows than columns.

The three shapes do not exactly correspond to the three classes based on consistency. But if you pick a matrix at random (like using `rand`), you will find that overdetermined systems are usually inconsistent, underdetermined systems are usually consistent with infinite solutions, and square systems are usually consistent with a unique solution.

The consistency can be analyzed in terms of the **rank** of a matrix. By definition, a matrix has rank r if it has r linearly independent columns, and all other columns are their linear combinations. One can replace "columns" by "rows" in the previous sentence. One can compute the rank with `rank` in Matlab. Also, the Matlab formula `[A b]` forms the **augmented matrix** of the system $Ax = b$, which has an extra column with the entries of b.

Theorem 6.1.1 Consistency and rank.

- *The system $Ax = b$ is consistent if and only if* `rank([A b])` *is equal to* `rank(A)`.
- *A consistent system has a unique solution if and only if* `rank(A)` *is equal to the number of columns of A.*

Example 6.1.2 Determine the consistency of a system. Use an if-elseif-else statement with `rank` to determine the consistency and the number of solutions of the system with matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

and with

$$b = \begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix}$$

Answer.

```
m = 3;
n = 3;
A = [1 2 3; 4 5 6; 7 8 9];
b = [10 11 12]';
if rank(A) < rank([A b])
    disp('Inconsistent')
elseif rank(A) == n
    disp('Consistent with a unique solution')
else
    disp('Consistent with infinitely many solutions')
end
```

□

6.2 Systems without free variables

What do we get when solving a system with `A\b` in Matlab? Let us first consider the case when `rank(A)` is `n`, so every column of A is a basic column, which means there are no free variables. When such a system is consistent, it has a unique solution and `A\b` finds this unique solution. When the system is inconsistent, we get the **least-squares solution**: the vector x for which the norm of the **residual vector** $Ax - b$ is as small as possible. Recall that the norm of a vector $v = (v_1, \dots, v_m)$ is

$$\|v\| = \sqrt{v_1^2 + \cdots + v_m^2}$$

In Matlab this norm can be computed in several ways:

- `sqrt(sum(v.^2))`

- `sqrt(v'*v)` when `v` is a column vector
- `sqrt(v*v')` when `v` is a row vector
- `norm(v)`

The last way is easiest.

Example 6.2.1 The least-squares solution and the norm of residual vector. Find the least-squares solution x and the norm of residual vector \$$Ax - b$ for the system with the matrix

$$A = \begin{pmatrix} 1 & 2 \\ 4 & 3 \\ 5 & 6 \end{pmatrix}$$

and with

$$b = \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix}$$

Answer.

```
A = [1 2; 4 3; 5 6];
b = [7 8 9]';
x = A\b;
disp(x)
disp(norm(Ax-b))
```

□

A practical use of least squares solutions is linear regression. Suppose we are given some data points (x_k, y_k) and want to find the line $y = cx + d$ that fits these points the best. The idea is to imagine we could have perfect fit:

$$\begin{aligned} cx_1 + d &= y_1 \\ cx_2 + d &= y_2 \\ \cdots &= \cdots \\ cx_m + d &= y_m \end{aligned}$$

This system can be written as $Az = b$ where

$$A = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{pmatrix}, \quad b = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}, \quad z = \begin{pmatrix} c \\ d \end{pmatrix}$$

This is an overdetermined system when we have more than 2 data points. Unless all x-values are the same (which should not happen), the rank of A is 2 and so the system has no free variables. This means that `A\b` will find its least squares solution: a vector with components (c, d) for which the sum of squares of the residuals $cx_k + d - y_k$ is as small as possible. This is the line of best least-squares fit, and this process is known as linear regression in statistics.

Example 6.2.2 Line of best fit. Find the line of best fit (in the sense of least squares) to the six points with x-values (-1, 0, 2, 3, 4, 7) and y-values (9, 8, 5, 5, 3, -2). Plot both the line and the points.

Answer.

```
x = [-1, 0, 2, 3, 4, 7]';
```

```
y = [9, 8, 5, 5, 3, 1]';
A = [x, x.^0];
z = A\y;
xx = linspace(min(x), max(x), 500)';
yy = [xx, xx.^0]*z;
hold on
plot(xx, yy)
plot(x, y, 'r*')
hold off
```

Explanation: think of $cx + d$ as $cx + dx^0$ because this naturally leads to the matrix of this linear system: its columns are `x` and `x.^0`. The plot of given data points created by `plot(x, y, 'k*')` consists of red asterisks, not joined by a line. But to plot the line of best fit (or a curve of best fit, in general) we use more points on the x-axis. These points are in the vector `xx`: they are evenly distributed over the same interval where the given data points lie. The way in which y-coordinates of this line are computed may look strange. The matrix product `[xx, xx.^0]*z` is logically the same as `z(1)*xx + z(2)`: multiply x-values by the coefficient c and add the constant term d. Writing it in matrix form allows for easier generalization later. □

6.3 Systems with free variables

When the rank of matrix A is less than the number of variables, the system has free variables. In a Linear Algebra course, we would move the free variables to the right, and solve the rest of the system. But Matlab sets free variables to 0 and then solves the system as described in Section 6.2.

Note for Octave users: this is one of rare cases when Octave behavior is different, as explained below. This will not make a difference for homework. (In my opinion, Octave has a better approach than Matlab.)

Example 6.3.1 System with a free variable. Use Matlab to solve the system

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} x = \begin{pmatrix} 10 \\ 11 \end{pmatrix}$$

Solution.

```
A = [1 2 3; 4 5 6];
b = [10; 11];
disp(A\b)
```

The displayed solution is `[-4.5; 0; 4.8333]`. □

In geometric terms, the system in Example 6.3.1 describes the intersection of two planes in 3-dimensional space. This intersection is a line. The solution that Matlab picks is the point where this line cross one of coordinate planes.

There is another logical choice of solution in a problem like this: one could pick the point on the line that is closest to the origin $(0, 0, 0)$, that is the solution of smallest norm. This is the approach that Octave uses: when given Example 6.3.1 it would produce `[-4.5556; 0.1111; 4.7778]`. One can get this solution with Matlab by using `pinv(A)*b`, which results in `[-4.5556; 0.1111; 4.7778]`. This approach takes longer because it involves computing the **pseudoinverse** of A which is a subject of MAT 532.

Example 6.3.2 Square rank-deficient system. Compare the system

$$\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{pmatrix} x = \begin{pmatrix}10\\ 11\\ 12\end{pmatrix}$$

to Example 6.3.1. Does it have the same solution?

Solution. This new system as one more equation, but it is redundant: if you multiply the second line by 2 and subtract the first line, the result is the 3rd line. So, this system is *mathematically* equivalent to the previous one. Yet, the Matlab solution is different.

```
A = [1 2 3; 4 5 6; 7 8 9];
b = [10; 11; 12];
disp(A\b)
```

The result is `[-25.3333; 41.6667; -16.0000]`, quite different from what we got in Example 6.3.1. Matlab did not even recognize this as a system with a free variable. □

What explains the result of Example 6.3.2? When solving it, we get a warning message: "Matrix is close to singular or badly scaled. Results may be inaccurate." This means the result of this computation are likely to be influenced by inevitable **round-off errors** in computer arithmetics, which is the subject of next section.

6.4 The effect of round-off-errors

The round-off errors made by a computer are usually very small: for example,

```
0.1 + 0.2 - 0.3
```

produces `5.5511e-17` instead of the expected `0`. The underlying reason is that, for example, the number 0.1 is internally represented not as 1/10 but as a fraction p/q where q must be a power of 2 (because the computer arithmetic is binary in nature. Both p and q are chosen very large which makes the approximation $p/q \approx 1/10$ very good for most purposes. But it cannot be perfect, because if $p/q = 1/10$, then $10p = q$ which is a contradiction: the left hand side is divisible by 5, so the right hand side cannot be a power of 2.

Example 6.4.1 Approximating numbers by fractions with a power of 2. For each of the numbers 0.1, 0.2, 0.3, find a fraction with denominator 32 which is the best approximation to it. Use these approximations, instead of the actual numbers, in the formula $0.1 + 0.2 - 0.3$. What is the result?

Solution. To find approximations, multiply each of the numbers by 32 and round the result to the nearest integer. Thus, $0.1 \approx 3/32$, $0.2 \approx 6/32$, and $0.3 \approx 10/32$ (because $0.3 \cdot 32 = 9.6$ rounds to 10). Hence

$$0.1 + 0.2 - 0.3 \approx \frac{3}{32} + \frac{6}{32} - \frac{10}{32} = -\frac{1}{32}$$

The result is not zero due to round-off errors. This is exactly what happens in Matlab computations, except that they involve a much larger power of 2. □

The tiny round-off errors have a major impact on the solution of linear systems when, as a result of some row operations, some entry should theoretically be zero but in practice it is not (like $0.1 + 0.2 - 0.3$ above). So, the process of elimination that was supposed to eliminate a redundant equation, making it

$0 = 0$, instead turns it into some “numerical garbage”. Matlab tries to detect such situations and warn the user about them, indicating that the solution it finds may be useless.

6.5 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking “Answer”. There are also questions for reviewing the concepts of this section.

Example 6.5.1 Parabola of best fit. Using the data of Example 6.2.2, find and plot the parabola $y = cx^2 + dx + e$ that fits the data best in the sense of least squares.

Answer.

```
x = [-1, 0, 2, 3, 4, 7]';
y = [9, 8, 5, 5, 3, 1]';
A = [x.^2, x, x.^0];
z = A\y;
xx = linspace(min(x), max(x), 500)';
yy = [xx.^2, xx, xx.^0]*z;
plot(xx, yy, 'b', x, y, 'r*')
```

Explanation: the only modification that we need is to add a column with the squares of x-values. In general, this process allows us to fit a function of the form $c_1 f_1(x) + c_2 f_2(x) + \cdots + c_n f_n(x)$ where $f_1, f_2, \ldots, f_n$ are given functions and $c_1, c_2, \ldots, c_n$ are coefficients (parameters) to be determined. □

Example 6.5.2 Slight perturbation of a singular system. Modify Example 6.3.2 by replacing 1 with 1.000001. How does the solution change? Repeat, this time replacing 9 by 9.000001. How did the solution change now?

Solution.

```
A = [1.000001 2 3; 4 5 6; 7 8 9];
b = [10; 11; 12];
```

leads to `A\b` being `[0.0000; -9.0000; 9.3333]`. In contrast, with

```
A = [1 2 3; 4 5 6; 7 8 9.000001];
```

we get the solution `[-9.3333; 9.6667; 0.0000]` which is quite different. Note that Matlab shows no warnings in either case. These systems do not have a free variable: the rank of A is 3. Yet, the solution is very sensitive to small changes of the system, because the matrix is close to being singular (or degenerate). □

Question 6.5.3 Inconsistent system. The simplest example of an inconsistent linear system is a system of two equations with one variable: $x = b_1$ and $x = b_2$ where b_1, b_2 are two unequal real numbers. What is the least-squares solution of this system, and why? (Try to answer without Matlab, and then check the conclusion with Matlab.) □

Question 6.5.4 Testing for invertibility with determinant or rank. Linear algebra says that an $n \times n$ matrix A is singular (non-invertible) if and only if $\det A = 0$, if and only if $\operatorname{rank} A < n$. But in computational practice these two tests for invertibility can give different results. Consider the diagonal matrix with entries 10^{16} and 1.

```
A = [1e16 0; 0 1];
disp(det(A))
disp(rank(A))
```

Matlab says the determinant is 10^{16} which makes sense: it is the product of diagonal entries. Yet, it also says the rank is 1. Why? □

Question 6.5.5 Model selection. How different is the best-fit parabola in Example 6.5.1 from the line of best fit? If you have to choose between fitting a line or a parabola to this data set, what would you use?

Later we will consider the issue of *model selection* systematically: how to decide which model is more appropriate (for example, linear or quadratic). □

6.6 Homework

1. For each of the following systems, decide if it is: (i) overdetermined, underdetermined, or square; (ii) has a unique solution, has infinitely many solutions, or is inconsistent. Briefly explain how you reached these conclusions. To save time, you may use Matlab's `rank` command to compute the ranks of matrices.

$$\begin{cases} 3x_1 - x_3 &= 0 \\ x_1 - 2x_2 &= 3 \\ 2x_2 + 5x_3 &= -1 \end{cases}$$

$$\begin{cases} 3x_1 + 4x_2 - x_3 &= 4 \\ x_1 - 2x_2 + 3x_3 &= 4 \end{cases}$$

$$\begin{cases} 3x_1 + 4x_2 - x_3 + x_4 &= 2 \\ x_1 - 2x_2 + 3x_3 - x_4 &= -3 \\ 7x_1 + 6x_2 + x_3 + x_4 &= 1 \end{cases}$$

2. Write a function `linearfit` that takes arguments `x` and `y` (vectors of the same size) and returns two numbers `c` and `d`, coefficients of the best-fitting line $y = cx + d$. Then apply it to the population of your hometown during years 1980, 1985, ..., 2010, 2015. (The data should be easy to find online; if not, use a larger nearby town). The code could look like this.

```
function hw6()
  x =  1980:5:2015;  % years
  y =                % population of your hometown in these years
  [c, d] = linearfit(x, y);            % find the coefficient of linear fit
  xx = linspace(min(x), max(x), 500);  % to be used for plotting the line
  plot(x, y, 'r*', xx, c*xx + d, 'b')  % plot red points and blue line
end

function [k, c] = linearfit(x,y)
  ... set up and solve an overdetermined system,
  ... then assign c and d based on its solution
end
```

The character `%` is used for inserting comments in Matlab code: everything after it to the end of the line is ignored.

Chapter 7

Root finding: bisection

A fundamental numerical problem is to solve an equation (in general, not linear). What does it mean to find a numeric solution of $f(x) = 0$? We consider the simplest of several methods of root-finding: the bisection method.

7.1 Motivation for solving equations numerically

We often want to solve an equation of the form $f(x) = 0$, given a real-valued function f of one real variable x. Some such equations admit a **symbolic** solution, obtained by some algebraic manipulations. For example, the equation $2e^x - 3 = 0$ has the solution $x = \log(3/2)$. But if we put together two or more unrelated functions, for example $2e^x + x^3 - 1 = 0$, algebra does not help.

For this reason, Matlab has a built-in root-finding function `fzero`. Its first argument is the function which should be equated to zero, the second argument is our initial guess at where the solution may be (put 0 if you have no idea). For example,

```
fzero(@(x) 2*exp(x) + x^3 - 1, 0)
```

finds the solution `-0.5439`. But even though we have `fzero`, we still have to understand the methods used for finding roots, because

(a) `fzero` only gives one solution, there may be others;

(b) the output of `fzero` may be wrong.

An example of wrong answer:

```
fzero(@(x) exp(-x^2), 0)
```

returns `x=-28.9631`. In reality, $e^{-x^2} = 0$ has no solutions, and there is nothing special about the number produced by `fzero`: this is just where its algorithm decided to stop, thinking "close enough".

7.2 The meaning of a numeric solution

In what sense do we expect the equation $f(x) = 0$ to hold? Because of the round-off errors (Section 6.4), we wlll not necessarily get 0 when plugging in some number x. For example, if `f = @(x) x^2-2;` then we would expect `f(sqrt(2))` to be 0, but actually `f(sqrt(2))` produces `4.4409e-16`, a small but nonzero number. We could ask for `abs(f(x))` to be small, for example `abs(f(x)) < eps` where the Matlab command `eps` returns `2^{-52}`, the amount of round-off error one can expect for numbers of size about 1. But there are two issues with this approach:

1. The error in computing `f(x)` may be much larger than `eps`, as round-off errors may accumulate.

2. Very small value of `f(x)` do not mean that `x` is near a root of function `f`: for example, the function `f = @(x) exp(-x^2);` has `abs(f(7)) < eps`, but the equation $f(x) = 0$ has no solutions at all.

A more robust approach is to look not for a single number but for an *interval* $[a, b]$ that is known to contain a root of function f. If the interval is small enough, it does not really matter where in the interval the solution is. How small is small enough? There is no universal answer. For many practical purposes, $b - a < 10^{-6}$ is good enough. Or we may want $b - a < 10^{-12}$ to get a more accurate answer. This makes sense when the root itself is not extremely large or extremely small. But when the root is of size 10^{20} we cannot possibly hope to get an interval of length 10^{-6} around it, or even of length 1. Indeed, the comparison `10^20 + 1 == 10^(20)` evaluates as True in Matlab, meaning that it cannot tell these numbers apart because of the limitation of computer arithmetic.

For the reason mentioned in the previous paragraph, the Matlab command `eps(x)` gives the smallest possible size of an interval including `x` that can be expressed in its arithmetic. For example, `eps(10^20)` returns 16384, indicating we cannot expect an interval around 10^{20} to be smaller than 16384. When looking for a root, a reasonable approach is to consider an interval $[a, b]$ "small enough" when `b - a < 100*eps(a)` where the factor 100 allows for accumulation of round-off errors during the computation.

7.3 Bisection method

The bisection method is based on the **Intermediate Value Theorem**. Suppose that the function f is continuous on the interval $[a, b]$ and $f(a)f(b) < 0$, meaning that f takes on values of opposite sign at the endpoints. The Intermediate Value Theorem says that there exists some root of f between a and b: that is, there exists x in (a, b) such that $f(x) = 0$.

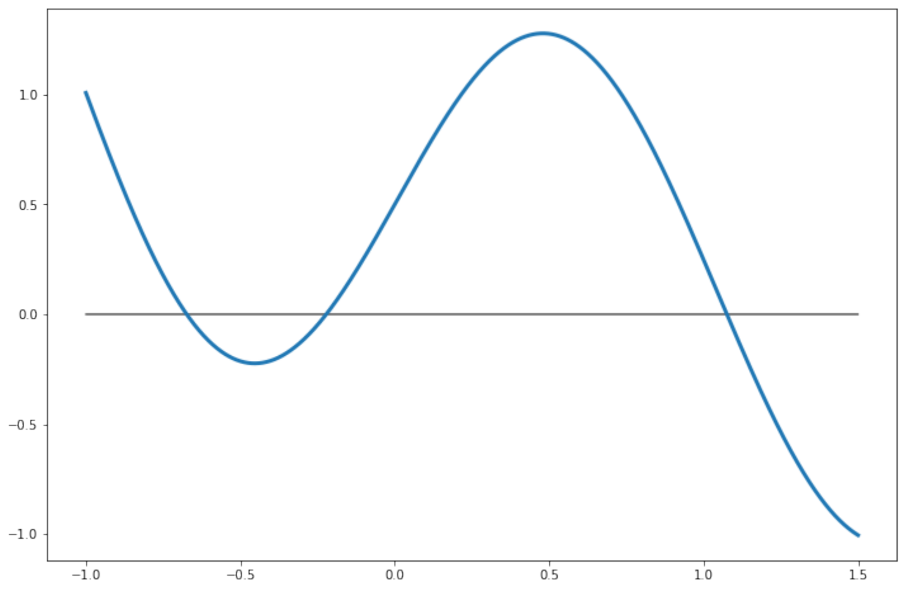

Figure 7.3.1 To go from positive to negative, a function function must turn to 0

An interval $[a, b]$ such that $f(a)f(b) < 0$ is called a **bracketing interval**. As Figure 7.3.1 shows, there could be multiple roots inside of the bracketing interva. The bisection method will find *one* of them.

Bisection method: Divide the interval in two halves by the midpoint $c = (a + b)/2$. Since $f(a)$ and $f(b)$ have opposite signs, one of the following holds:

- If $f(c)$ has the same sign as $f(a)$, then we replace a with c, making $[c, b]$ our new bracketing interval.
- If $f(c)$ has the same sign as $f(b)$, then we replace b with c, making $[a, c]$ our new bracketing interval.
- If $f(c) = 0$, we already found a root so the process stops.

The above process repeats until the bracketing interval is small enough, as discussed in Section 7.2.

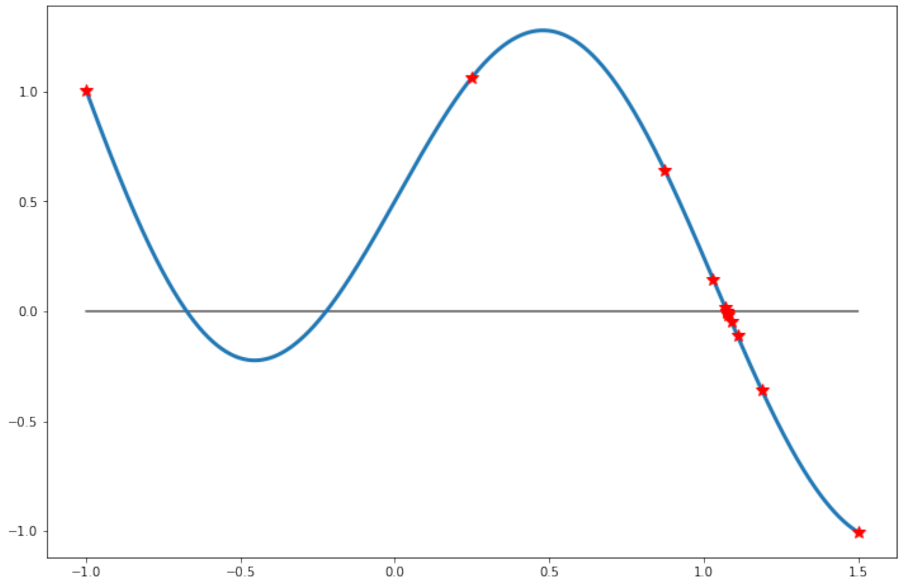

Figure 7.3.2 Bisection method illustrated by marking the bracketing intervals

Example 7.3.3 Finding a root by bisection. Find a root of equation $e^{-x/2} + \sin(3x) = 1/2$ on the interval $[-1, 1.5]$ using bisection. (This is the function shown on Figure 7.3.2). The Matlab function `sign` can be used to compare the signs of values.

Solution.

```
f = @(x) exp(-x/2) + sin(3*x) - 1/2;
a = -1;
b = 1.5;
if f(a)*f(b) >= 0
    error('Not a bracketing interval');
end

while abs(b-a) >= 100*eps(a)
    c = (a+b)/2;
    if sign(f(c)) == sign(f(a))
        a = c;
    elseif sign(f(c)) == sign(f(b))
        b = c;
    else
        break
    end
end

fprintf('Found a root x = %.12g\n', (a+b)/2);
```

If the function has the same sign at both given endpoints, the bisection method cannot run, so the command `error` is used to display a message and exit the program. This is a useful command to use when the computation has to be interrupted. The `while` loop runs until either the interval becomes small enough (length less than `100*eps(a)`) or we accidentally reach $f(c) = 0$ so the loop stops. □

The bisection method is not very fast, but is very reliable. If the initial bracketing interval had length L, after n iterations we get an interval of length

$2^{-n}L$. So, it takes about 50 iterations to reduce the bracketing interval from length 100 to length 10^{-13}. The speed of convergence of a numerical method is often visualized by plotting the logarithm of the error as a function of the number of steps. For the bisection method, the error after n steps is $2^{-n}|a-b|$ so the logarithm is $\log|a-b| - n\log 2$.

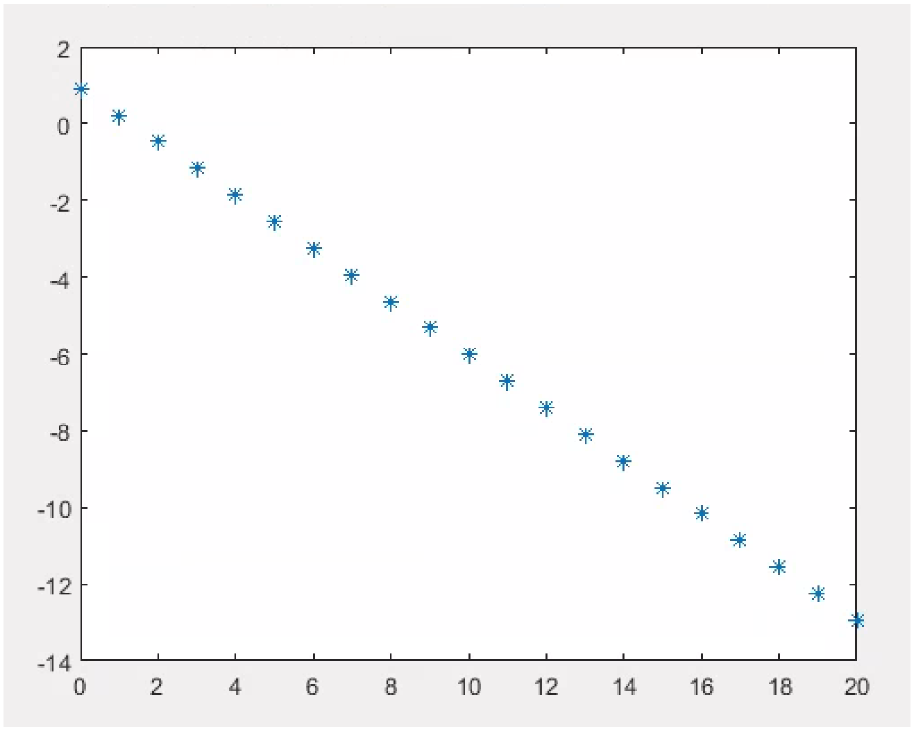

Figure 7.3.4 Logarithm of error as a function of the number of steps

Generally, if some numerical method has error ϵ_n after n steps, we say that the method is

- **linearly convergent** if $\epsilon_{n+1} \leq c\epsilon_b$ with some constant $c < 1$
- **quadratically convergent** if $\epsilon_{n+1} \leq M\epsilon_n^2$ with some constant M
- **convergent with order** p if $\epsilon_{n+1} \leq M\epsilon_n^p$ with some constants M, $p > 0$.

Since for the bisection method $\epsilon_{n+1} = \frac{1}{2}\epsilon_n$m the bisection method is linearly convergent.

7.4 Limitations of the bisection method

The main limitation of the bisection method are:

- It does not apply to systems of more than one equation
- It requires the knowledge of a bracketing interval
- It requires a continuous function
- Its speed of convergence is slow (linear)

To illustrate the second limitation, consider the equation $x^2 - 2x + 0.9999 = 0$. It has two roots, both of which are very close to 1. Outside of a small neighborhood of 1, the function is positive. So, in order to construct a bracketing interval one needs to place one of its endpoints very close to 1. This means one needs to have a good idea of where the roots are in order to start with the method.

Calculus methods involving the derivative f' can help us understand in what direction the function changes, improving our chance of finding a bracketing interval.

Example 7.4.1 Find the number of roots and a bracketing interval for each of them. For the function $f(x) = 2e^x + x^3 - 1$, determine the number of roots and find a bracketing interval for each of them.

Solution. The derivative $f'(x) = 2e^x + 3x^2$ is always positive. Therefore, the function f increases on the real line. Such a function either has no roots (if its graph never crosses the y-axis), or has one root. Since $f(x) \to \infty$ as $x \to \infty$ and $f(x) \to -\infty$ as $x \to -\infty$, it follows that the graph crosses the y-axis. We need a finite bracketing interval, since for the bisection method to work, both a and b must be finite. Since $f(0) = 2 + 0 - 1 = 1 > 0$ it remains to find a negative value. For example, $f(-1) = 2e^{-1} - 1 - 1 = 2(1 - e)/e < 0$.

Answer: one root, with a bracketing interval $[-1, 0]$. □

7.5 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 7.5.1 Number of roots and a bracketing interval for each of them. For the function $f(x) = 30xe^{10x} + 1$, use the derivative f' determine the number of roots and find a bracketing interval for each of them. (No programming is needed.)

Solution. The derivative $f'(x) = 30(10x+1)e^{10x}$ has the same sign as $10x+1$. Therefore, the function has a minimum at $x = -1/10$. Its value there is $f(-1/10) = -3e^{-1} + 1 = 1 - 3/e < 0$.

On the interval $(-\infty, -1/10)$ the function is decreasing, so there is at most one root here. Since $f(-1) = -30e^{-10} + 1 > 0$, there is a root with bracketing interval $[-1, -1/10]$.

On the interval $(-1/10, \infty)$ the function is increasing, so there is at most one root here. Since $f(0) = 1 > 0$, there is a root with bracketing interval $[-1/10, 0]$.

Answer: two roots, with bracketing intervals $[-1, -1/10]$ and $[-1/10, 0]$. □

Example 7.5.2 Incorrect answer obtained by bisection. Run the code in Example 7.3.3 with the following modification:

```
f = @(x) tan(x);
a = 1;
b = 2;
```

Note that $\tan(1) \approx 1.5574 > 0$ and $\tan(2) \approx -2.1850 < 0$, so the bisection algorithm can run. What is its output, and what is wrong with it? How could this error be avoided?

Solution. The program output is "Found a root x = 1.57079632679". But this value is not a solution of equation $\tan x = 0$. It is a point of discontinuity, where the tangent has vertical asymptote $x = \pi/2$. Because of discontinuity, the tangent function changes sign without passing through 0.

One way to avoid this error is to check whether the function has a "small" value at the root that we found, for example as follows.

```
x = (a+b)/2;
if abs(f(x)) < 1e-9
    fprintf('Found a root x = %.12g\n', x);
else
    fprintf('Suspected discontinuity at x = %.12g\n', x);
end
```

Then the output is "Suspected discontinuity at x = 1.57079632679" □

Question 7.5.3 "Trisection" method. The word "bisection" means dividing something into two parts, usually equal ones. In the bisection method, an interval is divided into two equal parts, of which we keep one. This means that the length of the interval is divided by 2 at each step.

One can imagine a similar "trisection method" where an interval is divided into three equal parts, and one of them is kept. With this method, the length of the interval is divided by 3 at each step, so the interval shrinks faster. Does this make the trisection method faster than the bisection method? Why or why not? □

Question 7.5.4 Better detection of discontinuity. In Example 7.5.2 we avoid mistaking discontinuity for a root by adding the check `abs(f(x)) < 1e-9`. What could go wrong with this approach? Think of some situation where this additional check will reject a valid solution. Could it be replaced by a better one? □

7.6 Homework

1. For the function $f(x) = \log(x^2+1/2)+9x^4$, use the derivative f' determine the number of roots and find a bracketing interval for each of them. (No programming is needed.)

2. Write a Matlab script which finds and prints *both* roots of the equation $30xe^{10x} = -1$, using the bisection of the bracketing interval found in Example 7.5.1. To avoid duplicating code, write a named function `bisection(f, a, b)` which takes the function and a bracketing interval as arguments and returns a root by using the algorithm in Section 7.3. The script could look as follows.

```
function hw7()
    f = @(x) 30*x*exp(10*x) + 1;
    root1 = bisection(f, -1, -0.1);
    root2 = ...
    fprintf('Found roots at %.12g and %.12g\n', root1, root2)
end

function x = bisection(f, a, b)
    ...
end
```

Chapter 8

Root finding: fixed point iteration

We look for solutions of the equation $f(x) = x$, which are called the fixed points of function f. The simplest approach to finding them is to iterate the function: that is, plug the value of f back into f, repeatedly.

8.1 Classification of fixed points

A **fixed point** of a function g is a number x^* such that $g(x^*) = x^*$. This is the same as a root of $g(x) - x = 0$ but we will see that writing an equation with x on the right provides a new approach to solving it.

Example 8.1.1 Find all fixed points of a function. Find all fixed points of the function $g(x) = \sqrt{1+x}$

Solution. Squaring both sides of $\sqrt{1+x} = x$ yields $x^2 - x - 1$ which is a quadratic equation with roots $(1 \pm \sqrt{5})/2$. However, the negative root does not satisfy the original equation $\sqrt{1+x} = x$.

Answer: one fixed point $x^* = (1 + \sqrt{5})/2$. □

The process of **iterating** some function g consists of the following: start from some initial value x_0, let $x_1 = g(x_0)$, $x_2 = g(x_1)$, and generally $x_{n+1} = g(x_n)$. If the sequence x_n has a limit x^*, and g is a continuous function, it follows that $g(x^*) = x^*$.

However, can we expect the iteration to converge to a fixed point x^*? The answer depends on the value of $g'(x^*)$. Indeed, when $x \approx x^*$, the difference $g(x) - x^*$ is approximately $g'(x^*)(x - x^*)$ because the graph of a differential function is close to its tangent line. When $|g'(x^*)| < 1$, the distance to fixed point is decreasing but if $|g'(x^*)| > 1$ it is increasing. This leads us to introduce the following concept.

Classification of fixed points. A fixed point x^* is called

- **repelling** if $|g'(x^*)| > 1$
- **attracting** if $|g'(x^*)| < 1$
- **super-attracting** if $g'(x^*) = 0$ (a special case of attracting fixed points)
- **neutral** if $|g'(x^*)| = 1$

According to the above, iteration of g may converge to an attracting point but will not converge to a repelling one. The neutral points are a borderline case: some of them are limits of iterative process, some are not.

Example 8.1.2 Find and classify the fixed points. Find and classify all fixed points of the function $g(x) = x^3 + x^2 - x$.

Solution. The equation $x^3 + x^2 - x = x$ simplifies to $x(x^2 + x - 2) = 0$ which has roots $0, 1, -2$. These are the three fixed points. Plug each into the derivative $g'(x) = 3x^2 + 2x - 1$ to find that

- $|g'(0)| = |-1| = 1$, a neutral fixed point
- $|g'(1)| = |4| = 4$, a repelling fixed point
- $|g'(-2)| = |7| = 7$, a repelling fixed point

□

8.2 Rewriting equations in the fixed-point form

We now have a new method of solving the equations $f(x) = 0$: rewrite it as $g(x) = x$, and build a sequence $x_{n+1} = g(x_n)$. If this sequence has a limit, which we recognize by $|x_{n+1} - x_n|$ being small, then we have a fixed point of g, hence a root of f. There are multiple ways to rewrite $f(x) = 0$ in the fixed-point form:

- $f(x) + x = x$
- $x - f(x) = x$
- $x - 3f(x)^3 = x$

and so on. Although the fixed points are the same each case (they are the roots of f), their classification may be different. For example, consider $f(x) = x^3 + 4x - 5$. We know it has a root at $x = 1$. If we try to rewrite the equation $f(x) = 0$ as $x^3 + 5x - 5 = x$ the fixed point at 1 is repelling because $(x^3 + 5x - 5)' = 3x^2 + 5$ has modulus $8 > 1$. Another rewrite is $(5 - x^3)/4 = x$. This one works because the derivative of $(5 - x^3)/4$ is $-3x^2/4$ with absolute value of $3/4$ at $x = 1$. The denominator 4 helped reduced the derivative here.

Example 8.2.1 Rewrite and solve by fixed-point iteration. Solve the equation $x^3 + 5x = 2$ by the fixed-point iteration method.

Solution. The function $x^3 + 5x$ is increasing from 0 to 6 on the interval $[0, 1]$. Therefore, the equation has a solution in this interval. Following the idea of the previous paragraph, rewrite it as $(2 - x^3)/5 = x$. This means $g(x) = (2 - x^3)/5$ and $g'(x) = -3x^2/5$. Note this $|g'| < 1$ on $[0, 1]$ which guarantees that the fixed point is attracting. Here is the code to find it.

```
g = @(x) (2-x^3)/5;
x0 = 0;
max_tries = 1000;
for k = 1:max_tries
    x1 = g(x0);
    if abs(x1-x0) < 100*eps(x0)
        break
    end
    x0 = x1;
end
if k < max_tries
```

```
    fprintf('Found x = %.12g after %d steps\n', x1, k);
else
    disp('Failed to converge')
end
```

The reason for using a `for` loop is to set a limit for the number of attempts (1000). The loop ends sooner if the values of x essentially stop changing. We do not need separate variables for every element of the sequence: x0 and x1 keep being reused for “old” and “new” x-values. The output is “Found x = 0.388291441005”. □

8.3 The speed of convergence of fixed-point iteration

Near an attracting fixed point with $0 < |g'(x^*)| < 1$, the convergence is linear in the sense that the error at the next step is about $|g'(x^*)|$ times the error of the previous step. But if $g'(x^*) = 0$ (meaning x_0 is a superattractive fixed point), then the rate of convergence is extremely fast. Indeed, Taylor's formula says that $|g(x) - g(x^*)|$ is bounded by a constant multiple of $|x - x^*|^2$, where the constant depends on the value of the second derivative $f''(x^*)$. This rate of convergence is called **quadratic** because at each step, the error size gets squared (and multiplied by a constant).

Compare two functions: $g_1(x) = \sqrt{x}$ and $g_2(x) = (x+1/x)/2$. Both have a fixed point $x^* = 1$. Since $|g_1'(1)| = 1/2$ and $|g_2'(1)| = 0$, the point 1 is attracting for g_1 and super-attracting for g_1. Let's see how quickly the iteration converges to the fixed point, starting from 5 for example.

```
g = @(x) sqrt(x);
x = 5;
for k = 1:10
    x = g(x);
    fprintf('After %d steps got %.15g\n', k, x)
end
```

The numbers approach 1 but not very quickly.

```
After 1 steps got 2.23606797749979
After 2 steps got 1.49534878122122
After 3 steps got 1.22284454499385
After 4 steps got 1.10582301703024
After 5 steps got 1.05158119849598
After 6 steps got 1.02546633220988
After 7 steps got 1.01265311543977
```

And so on: each error is about half of the previous, indicating linear speed of convergence. After 10 steps we are still more than 0.001 away from the fixed point. In contrast, running the above code with `g = @(x) (x+1/x)/2;` results in much faster convergence.

```
After 1 steps got 2.6
After 2 steps got 1.49230769230769
After 3 steps got 1.0812053925456
After 4 steps got 1.00304952038898
After 5 steps got 1.00000463565079
After 6 steps got 1.00000000001074
After 7 steps got 1
```

After just 7 steps of iteration, the value is so close to 1 that it becomes equal to 1, due to round-off involved.

8.4 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 8.4.1 Rewriting an equation to make a specific solution an attracting fixed point. The equation $x^5 - 2x = 1$ has a solution in the interval $(1, 2)$, as one can see from the Intermediate Value Theorem. Rewrite the equation as a fixed-point problem so that this root is an attracting fixed point.**Solution**. The form $(x^5 - 1)/2 = x$ does not work here because the function $g(x) = (x^5 - 1)/2$ has $|g'(x)| = 5x^4/2$ which is greater than 1 on the interval $(1, 2)$. But $(2x + 1)^{1/5} = x$ works because the function $g(x) = (2x + 1)^{1/5}$ has $|g'(x)| = 2|2x + 1|^{-4/5}/5$ which is less than $2/5$. □

Example 8.4.2 Classify an infinite set of fixed points. The function $g(x) = \tan x$ has infinitely many fixed points, because the graph of tangent intersects the line $y = x$ infinitely many times. Show that one of the fixed points is neutral and the rest are repelling.**Solution**. The neutral point is at 0, because $\tan 0 = 0$ and $g'(0) = 1$. How to find g' at other fixed points, if we do not know them? Recall that $g'(x) = \sec^2 x = \tan^2 x + 1 = g(x)^2 + 1$. Therefore, if x^* is a fixed point, then $g'(x^*) = (x^*)^2 + 1$. This expression is strictly greater than one when $x^* \neq 0$. So, all other fixed points are repelling. □

Question 8.4.3 Convergence to a neutral fixed point. Both functions $\sin x$ and $\tan x$ have 0 as a neutral fixed point. For one of these functions, the iteration method converges to 0, although *very* slowly. For another it does not. Why? (Hint: consider the plots of these functions.) □

Question 8.4.4 Logarithmic plot of error size. Recall that a sequence x_n converges to its limit x^* at a linear rate, the graph of $\log |x - x_n|$ as a function of n looks like a line. What will it look like when the rate of convergence is quadratic? □

8.5 Homework

1. For each of the following functions, find all of its fixed points. Classify each fixed point as attracting, repelling, or neutral.

(a) $f(x) = x^5$

(b) $g(x) = 16/x^3$

(c) $h(x) = x^3 - x/2$

(No programming is needed.)

2. Write a script that solves the equation $e^x + \log x = n$ using the fixed point iteration. Here n is the number formed by the first two digits of your SUID.

In order to use the fixed point method, you need to rewrite the given equation as $g(x) = x$. There are several ways to do this, try at least two different ones, or as many as are needed until a root is found. Starting from some value of x, such as $x = 10$, below, run fixed point iteration until either it converges to a root, or the number of iterations becomes extremely large. The script should display either “Root found at x = ... after ... steps” or “Failed to converge”. When submitting homework on Blackboard, add a comment with the different functions g you tried, and what the outcome was.

Chapter 9

Newton's method and secant method

Newton's method of solving an equation $f(x) = 0$ is based on iterating a specific function g such that every root of f is an attracting (usually super-attracting) point for g. The formula for g is geometrically motivated: it comes from tangent line approximation to the graph of f and therefore involves its derivative. Since the derivative is not always known, we consider an alternative **secant method** which replaces the tangent line by a secant line.

9.1 Newton's method

The idea of Newton's method is geometrically natural. We want to solve $f(x) = 0$ but since f may be complicated, we replace it by its linear approximation (tangent line at a certain point). The resulting linear equation is easy to solve. Of course, its solution is probably not a solution of the original equation, but perhaps it is close to it. The process repeats, using the previously found solution as a new point at which to draw a tangent line. A static image of this process is below. Interactive visualization is also available.

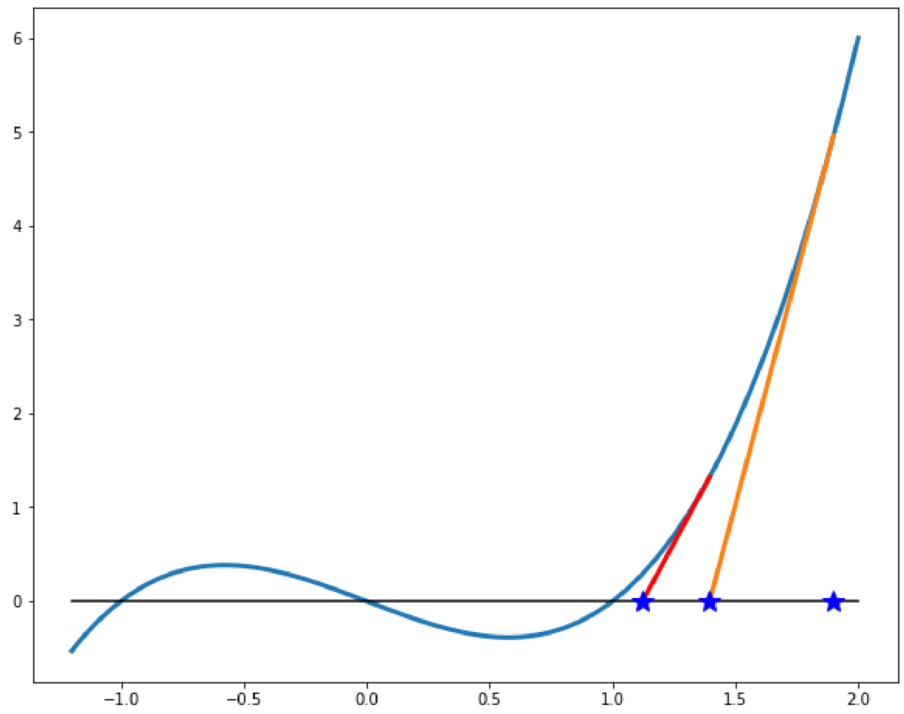

Figure 9.1.1 Two steps of Newton's method

Formally, the linear approximation to f at a point x is $f(x+h) \approx f(x) + f'(x)h$. Equating the right hand side to 0 we find $h = -f(x)/f'(x)$ which means that the point where the tangent line at x meets the line $y = 0$ is $x+h = x-f(x)/f'(x)$. Therefore, we can express Newton's method as iteration of the function

$$g(x) = x - \frac{f(x)}{f'(x)} = \frac{xf'(x) - f(x)}{f'(x)}$$

Note that any point where $f(x) = 0$ and $f'(x) \neq 0$ is a fixed point of g. In fact, it is a super-attracting point of g because

$$g'(x) = 1 - \frac{f'(x)f'(x) - f(x)f''(x)}{(f'(x))^2} = \frac{f(x)f''(x)}{(f'(x))^2}$$

which is zero when $f(x) = 0$.

Example 9.1.2 Find and simplify the function g of Newton's method. Given $f(x) = x^3 - x$, find and simplify the function $g(x) = x - f(x)/f'(x)$.

Solution. Bringing g to common denominator often helps:

$$g(x) = \frac{xf'(x) - f(x)}{f'(x)} = \frac{x(3x^2-1) - (x^3 - x)}{3x^2 - 1} = \frac{2x^3}{3x^2 - 1}$$

□

Once we have the function g, the rest of the process is the same as in Chapter 8, in particular in Example 8.2.1. Changing the first two lines of that program to

```
g = @(x) 2*x^3/(3*x^2-1);
x0 = 0.6;
```

we get "Found x = 1 after 11 steps". If the starting point is different, the process may converge to a different root of the equation $x^3 - x = 0$. For example with `x0 = 0.57` we get "Found x = -1 after 14 steps". A small change of initial condition produced a rather different solution; we went from finding x = 1 to finding x = -1. The pattern of what starting points produce which solution can be very complicated: see the illustrations on Wikipedia page Newton fractal.

9.2 Potential issues with Newton's method

First potential issue: we might not have a formula for the derivative of function f. This can be overcome by using numeric differentiation, which is covered later in the course.

The points where $f'(x) = 0$ are problematic for Newton's method, for two reasons. One is that g is not defined at such a point because of division by zero. So, if we happen to arrive at such a point in the process of iteration, the process has to be stopped and restarted again with a different initial value. Even if the derivative is not exactly zero but is very close to it, the value of g can be huge, sending the search to a location far from the actual solutions.

The second reason the zeros of derivative are problematic is that a point where $f(x) = f'(x) = 0$ is not a superattracting fixed point of g, even if we manage to extend the definition of g to such a point by canceling out the zero in the denominator. For example, let $f(x) = x^p$ where $p > 1$. Clearly, $f(0) = f'(0) = 0$. Computing

$$g(x) = \frac{xf'(x) - f(x)}{f'(x)} = \frac{px^p - x^p}{px^{p-1}} = \frac{p-1}{p}x$$

we see that g can be extended to $g(0) = 0$. Since $|g'(0)| = (p-1)/p < 1$, this is an attracting point. But it is not a super-attracting one, which means the speed of convergence of Newton's method to such **multiple roots** is slow: linear instead of quadratic. Here is an example of what happens with $f(x) = x^3$: the points approach the root $x = 0$ quite slowly, with each step being 1/3 of the distance to the root. This is even slower than bisection.

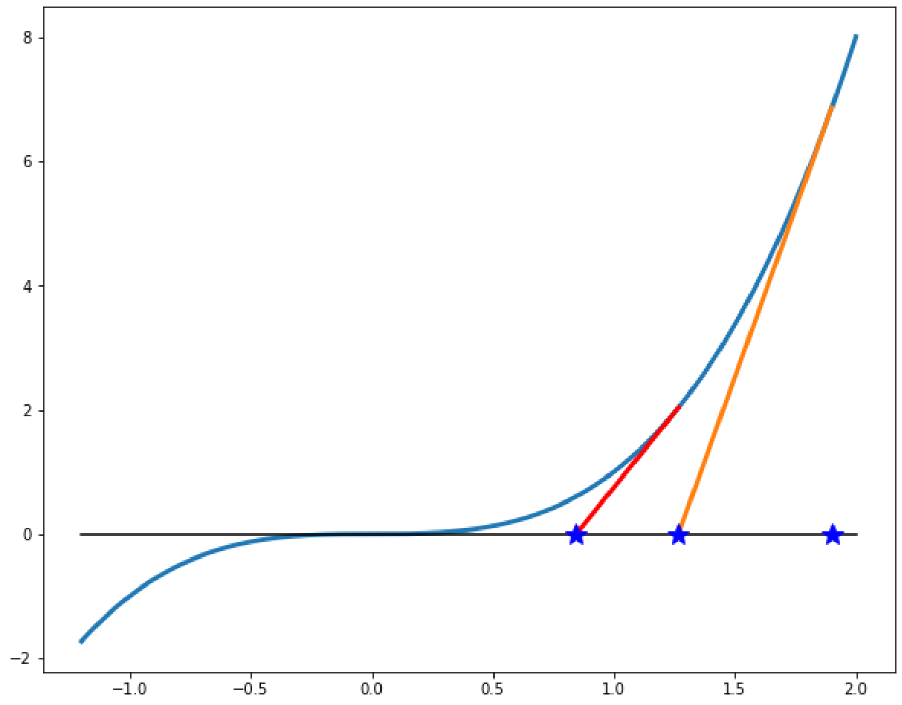

Figure 9.2.1 Newton's method converges slowly to a multiple root

A way to improve convergence in this case is presented in Section 9.4. Fortunately, a multiple root is a somewhat exceptional case and in practice we find Newton's method converging at quadratic rate.

But the most serious issue is that the method may fail to converge at all, either because the iteration of g gets stuck in a loop or because it tends to infinity. A typical example is $f(x) = \tan^{-1} x$ (note that Matlab notation for arctangent function is `atan`). Here $g(x) = x - (x^2 + 1)\tan^{-1}(x)$, which is `g = @(x) x - (x^2+1)*atan(x);` in Matlab notation. With the initial value `x0 = 1` the process quickly converges to 0. With the initial value `x0 = 2` (or even 1.5) it fails to converge. The problem can be seen numerically by computing a few iterations in Matlab console.

```
>> g(1.5)
ans = -1.6941
>> g(ans)
ans =  2.3211
>> g(ans)
ans = -5.1141
>> g(ans)
ans =  32.296
```

This also illustrates the use of `ans`, the variable that has the result of previous computation. Geometrically, Newton's method overshoots the target because the tangent lines have small slope.

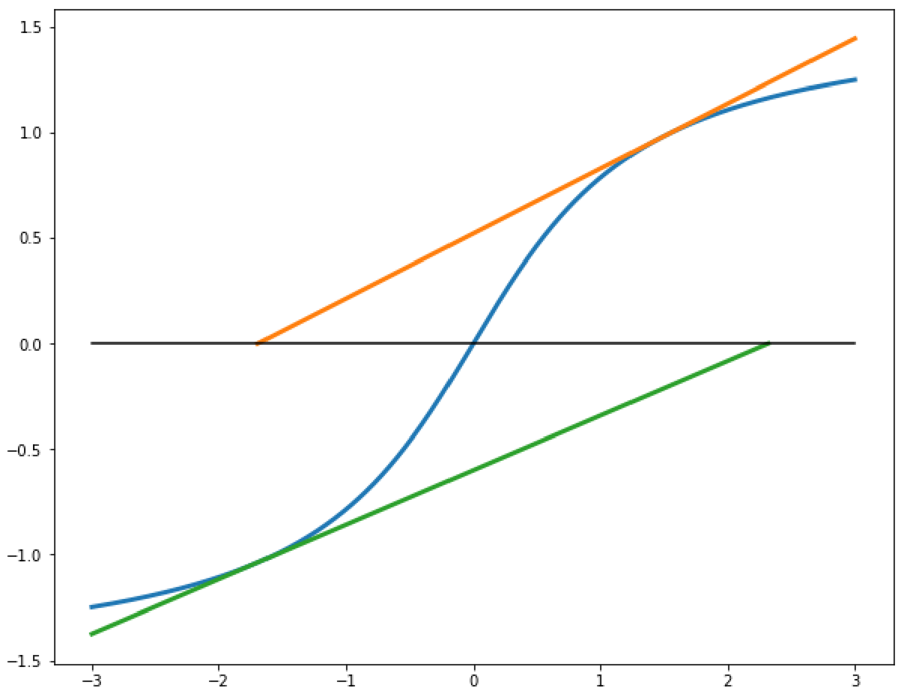

Figure 9.2.2 Newton's method overshoots and fails to converge

9.3 The secant method

The secant method is a close relative of Newton's method. The only difference is that instead of drawing a tangent line and finding where it crosses the line $y = 0$ we do the same for a secant line. The slope of a secant line is computed as $(f(x) - f(p))/(x - p)$ where x, p are two distinct points on the x-axis. The benefit is that we do not need the derivative of f, we only use the function itself. The drawback is that we have to keep track not of just one point, but of two. This makes the algorithm slightly more complicated: it is not just fixed-point iteration of some function g as in Newton's method. Also, the analysis of its speed of convergence is more complicated: generally, it is between linear and quadratic; faster than bisection but slower than Newton.

Example 9.3.1 Implement the secant method. Write a script that starts with the definition of a function f and two initial points, for example

```
f = @(x) x^3 - x;
x = 5;
p = 7;
```

and proceeds to find a root of f using the secant method: the next point to be computed will be

$$x - f(x)\frac{x - p}{f(x) - f(p)}$$

where the fraction represents division by the slope of secant line.

Solution.

```
f = @(x) x^3 - x;
x = 5;
```

```
p = 7;
max_tries = 1000

for k = 1:max_tries
    x1 = x - f(x)*(x-p)/(f(x)-f(p));
    if abs(x1-x) < 100*eps(x)
        break
    end
    p = x;
    x = x1;
end

if k < max_tries
    fprintf('Found x = %.12g after %d steps\n', x, k);
else
    disp('Failed to converge')
end
```

This script follows the structure of Example 8.2.1 except that we compute "new x", called x1, on the basis of two previous values (called `x` and `p`). After the computation, `x` and `p` are replaced by `x1` and `x`: the newly computed point becomes one of the two points through which we draw next secant line. □

9.4 The relaxation parameter

When faced with an iterative method which moves either too slowly, undershooting the target (Figure 9.2.1) or too quickly, overshooting it (Figure 9.2.2), we can try and fix the issue by introducing a **relaxation parameter**. This is a fixed positive number $\omega > 0$ by which we multiply the step size h obtained from Newton's method (or secant method, etc). That is, instead of iterating the function

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

we iterate the function

$$g(x) = x - \omega\frac{f(x)}{f'(x)}$$

If $\omega > 1$, this is **over-relaxation**, meant to make the process move faster. If $\omega < 1$, this is **under-relaxation**, meant to slow down the process so it does not overshoot and run off in a wrong direction.

How to determine an appropriate value of ω? Consider the case of a multiple root: $f(x) = x^p$ with $p > 1$. Here

$$g(x) = x - \omega\frac{f(x)}{f'(x)} = x - \omega\frac{x^p}{px^{p-1}} = (1 - \omega/p)x$$

so the best value to use $\omega = p$, which makes $g(x) = 0$, the solution is found in one step. So, for a double root, when the graph of f looks like a parabola near the root, we should use over-relaxation with $\omega = 2$; for triple roots, $\omega = 3$, and so on. Over-relaxation needs to be done very carefully: in these examples, any value of ω other than the correct one will fail to achieve the desired speed up of convergence.

Similarly, a function shaped like $f(x) = x^{1/3}$ may need under-relaxation, in this case the previous paragraph suggests $\omega = 1/3$. In general, under-relaxation is a less delicate issue than over-relaxation: seeing the failure to converge, one can simply try $\omega = 0.1$, and if that does not help, maybe $\omega = 0.01$. The convergence, if it is achieved, will usually be slow (linear).

9.5 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 9.5.1 Newton's method with a discontinuous function. Recall from Example 7.5.2 that the bisection method had a difficulty with the equation $\tan x = 0$, mistaking a discontinuity of this function for its root. How does Newton's method behave here, and why?

Solution. Newton's method has no issues with this function: if the starting point is close to a discontinuity, it will move away from it, toward the nearest zero. It will never overshoot the zero because of the concavity of the function: it is concave away from the x-axis, which means the tangent lines are between the graph and the x-axis.

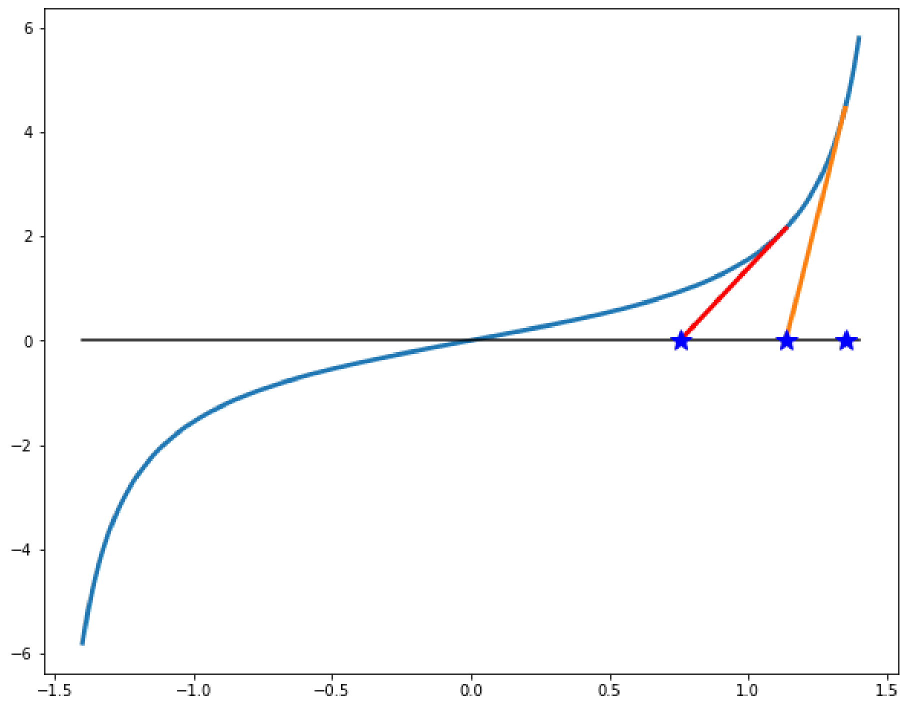

Figure 9.5.2 Newton's method works fine for the tangent function

Generally, vertical asymptotes are not a problem for Newton's method: horizontal asymptotes (and horizontal tangent lines) are. □

Example 9.5.3 Rate of convergence of the secant method. It is difficult to analyze the convergence of secant method in general. A representative example where such analysis can be done is $f(x) = x + x^2$. Find a simplified form of the function $x - \dfrac{x-p}{f(x)-f(p)} f(x)$ and try to justify the statement that the rate of convergence is faster than linear, assuming that x and p converge to 0.

Solution. Simplification shows that

$$\begin{aligned} x - \frac{x-p}{f(x)-f(p)} f(x) &= x - \frac{x-p}{(x-p)+(x^2-p^2)}(x+x^2) \\ &= x - \frac{1}{1+x+p}(x+x^2) \\ &= \frac{x+x^2+px-(x+x^2)}{1-x-p} \\ &= \frac{px}{1-x-p} \end{aligned}$$

Since $x, p \to 0$, the denominator is close to 1. Therefore, the method goes from x to approximately px where p is the point preceding x. For comparison, linear convergence would mean going from x to approximately cx where c is a fixed number. Since the factor p itself goes to zero, the secant method has faster than linear rate of convergence. □

Question 9.5.4 Equation with no solution. What happens if either Newton's method or secant method is applied to the equation $|x| + 1 = 0$ (which, of course, has no solutions)? □

Question 9.5.5 Another equation with no solution. What happens if Newton's method is applied to the equation $e^x = 0$ (which has no solutions)? □

9.6 Homework

1. (Theoretical problem.) Suppose that Newton's method is used to solve $f(x) = 0$ with $f(x) = e^x + e^{-x} - 2$. Show that the function $g(x) = x - f(x)/f'(x)$ (extended by $g(0) = 0$) has a nonzero derivative at 0 by computing the limit of $g(x)/x$ as $x \to 0$.

Hint. Hint: To simplify $f(x)/f'(x)$, you can multiply the numerator and denominator by e^x and recognize that they both can be factored.

2. (Theoretical problem.) Show that if we use $g(x) = x - 2f(x)/f'(x)$ in the previous problem, this function g has $g'(0) = 0$, which makes 0 a superattracting fixed point.

3. Write a script that attempts to solve the equation $\arctan x - 1 = 0$ using Newton's method (Note that Matlab's notation for arctangent is `atan`.) Similarly to Example 9.3.1 and Example 8.2.1, the script should show the number of steps it took to find a solution, or report that it failed to converge. Allow the algorithm up to 10000 steps before giving up. As the initial point `x0`, use the number formed by the first two digits of your SUID. (Note: it is expected that your script will fail to converge.)

Introduce a relaxation parameter ω in your script to help it converge. What value of ω achieves convergence? Be careful: ω should be small, but if it is too small, the algorithm will fail to converge because it takes too long.

When submitting homework on Blackboard, comment on what values of ω you found to be too large for convergence, what value was too small for convergence, and what value achieved convergence.

Chapter 10

Systems of several nonlinear equations: multivariate Newton's method

Having learned to solve a nonlinear equation with one unknown, we now consider systems of several such equations, with several unknowns. We assume the number of unknowns is equal to the number of equations.

10.1 Systems of nonlinear equations

Consider a non-linear system of more than one equation, such as

$$\begin{cases} x^2e^{3y} - 30 &= 0 \\ xy - \sin(x + y^2) &= 0 \end{cases} \tag{10.1.1}$$

What method could we use to solve it numerically?

- Bisection method does not apply. If we try to imitate it by taking a rectangle in the xy-plane and dividing it into 4 equal subrectangles, it is not clear which rectangle should be kept and why. The issue is that Intermediate Value Theorem, on which this method is based, does not generalize to systems.
- Fixed point iteration still makes sense: we can rewrite the system as
$$\begin{cases} x &= g(x, y) \\ y &= h(x, y) \end{cases}$$
and start iteration from some point (x_0, y_0): that is, $x_1 = g(x_0, y_0)$, $y_1 = h(x_0, y_0)$ and so on. If the process converges, we have a solution of the original system. But since there are more directions in which points can move under iteration, convergence is less likely than it was in one dimension.
- Newton's method, which is essentially a systematic approach to fixed-point iteration, still works. This is the method we will use in this section.
- It is not clear whether secant method makes sense, but its general idea - modifying Newton's method to avoid the use of derivatives - can be carried out. This will be considered later.

For notational convenience, we express a nonlinear system of n equations with n unknowns as a **vector equation** $\mathbf{F}(\mathbf{x}) = 0$ where $\mathbf{x}$ is an unknown vector with n components. For example, to express (10.1.1) in vector form we write

$$\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}, \quad \mathbf{F}(\mathbf{x}) = \begin{pmatrix} x_1^2 e^{3x_2} - 30 \\ x_1 x_2 - \sin(x_1 + x_2^2) \end{pmatrix} \tag{10.1.2}$$

How should we write a vector function $\mathbf{F}$ in Matlab? It can be an anonymous function.

```
F = @(x) [x(1)^2*exp(3*x(2)) - 30; x(1)*x(2) - sin(x(1) + x(2)^2)];
x = [3; -2];
disp(F(x))
```

In this example, the argument of F is a vector and the value it returns is also a vector. Note that `F(3, -2)` would result in an error "too many input arguments" because F takes only one argument (which is a vector). It needs to be `F([3; -2])`. Writing the input as a row vector, `F([3, -2])` would have the same effect but for consistency with the formulas below it is better to use column vectors.

When the notation `x(1), x(2)` becomes too cumbersome, one can consider writing F as a named function, where the input vector is first unpacked into separate variables such as `x, y`. This makes the formula more readable. The downside is the restriction on named functions in Matlab, noted in Section 5.2.

```
function v = F(u)
    x = u(1);
    y = u(2);
    v = [x^2*exp(3*y) - 30; x*y - sin(x + y^2)];
end
```

10.2 Multivariate Newton's method

Newton's method (Section 9.1) is based on the idea of replacing a nonlinear function with its linear approximation, and solving the resulting linear equation. The linear approximation comes from the derivative. In the setting of several variables we have **partial derivatives** which are arranged into the **Jacobian matrix**. The (i, j) entry of the Jacobian matrix is the derivative of the ith component of $\mathbf{F}$ with respect to the jth component of $\mathbf{x}$:

$$J = \begin{pmatrix} \partial F_1/\partial x_1 & \partial F_1/\partial x_2 \\ \partial F_2/\partial x_1 & \partial F_2/\partial x_2 \end{pmatrix}$$

(or more rows/columns if there are more variables).

Example 10.2.1 Find the Jacobian matrix. Find the Jacobian matrix of the function (10.1.2). Express it as an anonymous function in Matlab.

Solution. The Jacobian matrix is

$$J = \begin{pmatrix} 2x_1 e^{3x_2} & 3x_1^2 e^{3x_2} \\ x_2 - \cos(x_1 + x_2^2) & x_1 - 2x_2 \cos(x_1 + x_2^2) \end{pmatrix}$$

As a Matlab function, it can be written as follows.

```
J = @(x) [2*x(1)*exp(3*x(2)), 3*x(1)^2*exp(3*x(2));
          x(2) - cos(x(1) + x(2)^2), x(1) - 2*x(2)*cos(x(1) + x(2)^2)];
```

The linebreak between matrix rows is optional but improves readability. □

The linear approximation to $\mathbf{F}$ at a point $\mathbf{x}$ is $\mathbf{F}(\mathbf{x}+\mathbf{h}) \approx \mathbf{F}(\mathbf{x}) + J\mathbf{h}$ where $J\mathbf{h}$ is a matrix-vector product.

Example 10.2.2 Find the linear approximation. Find the linear approximation to function (10.1.2) using `x = [2; -1]` and `h = [0.3, 0.2]`. Compare the approximation to the actual value of `F(x+h)`.

Solution.

```
F = @(x) [x(1)^2*exp(3*x(2)) - 30; x(1)*x(2) - sin(x(1) + x(2)^2)];
J = @(x) [2*x(1)*exp(3*x(2)), 3*x(1)^2*exp(3*x(2));
          x(2) - cos(x(1) + x(2)^2), x(1) - 2*x(2)*cos(x(1) + x(2)^2)];

x = [2; -1];
h = [0.3; 0.2];
fprintf("Linear approximation: (%g, %g)\n", F(x) + J(x)*h)
fprintf("Actual value: (%g, %g)\n", F(x + h))
```

The output:

```
Linear approximation: (-29.6216, -2.14012)
Actual value: (-29.5201, -2.04023)
```

These are close, so the approximation is good. Note the use of formatted strings with vectors: Matlab automatically "unpacks" vectors when they are used in `fprintf`. This means one should have a formatting code for every entry of the vector, like `(%g, %g)` above. The formatting code `%g` without any specification of precision means the numbers are shown as Matlab would normally shows them. □

As with the single-variable Newton method, we equate the linear approximation to zero and solve the linear equation. The solution of $\mathbf{F}(\mathbf{x}) + J\mathbf{h} = 0$ is $\mathbf{h} = -J^{-1}\mathbf{F}(\mathbf{x})$. This formula is only for writing down the theoretical approach, in practice we do not invert the matrix J and simply ask Matlab to solve the linear system (this is more efficient than computing the inverse matrix). So, in Matlab terms the formula is `h = -J\F(x)`. Having found the vector $\mathbf{h}$ we replace $\mathbf{x}$ with $\mathbf{x}+\mathbf{h}$ and repeat. The process stops when the steps becomes sufficiently small, indicating we approached a solution. The size of vectors is measured by their norm: `norm(h)`.

The typical behavior of Newton's method is that it jumps around for several steps, but once it gets in a neighborhood of a solution, it converges to it very quickly.

Example 10.2.3 Solve a system using Newton's method. Solve the system (10.1.1) using Newton's method. Try different initial points. Does the method always converge? Does it converge to the same solution?

Solution. Following the structure of single-variable Newton method with notational adjustments:

```
F = @(x) [x(1)^2*exp(3*x(2)) - 30; x(1)*x(2) - sin(x(1) + x(2)^2)];
J = @(x) [2*x(1)*exp(3*x(2)), 3*x(1)^2*exp(3*x(2));
          x(2) - cos(x(1) + x(2)^2), x(1) - 2*x(2)*cos(x(1) + x(2)^2)];

x = [2; 1];
max_tries = 10000;

for k = 1:max_tries
    h = -J(x)\F(x);
    x = x + h;
    if norm(h) < 100*eps(norm(x))
        break
    end
end

if k < max_tries
    fprintf('Found a solution after %d steps:\n', k);
    disp(x)
    disp('The value F(x)')
    disp(F(x))
else
    disp('Failed to converge')
end
```

It is reasonable to increase the "max tries" number in Newton's method when several variables are involved, as convergence may take longer. In the above example, with initial point `[2; 1]`, a solution `[-0.0019; 5.3185]` is found in 15 steps. With initial point `[1; 2]` the method fails to converge. With initial point `[2; 2]` it converges in 10 steps to a different solution, `[-0.2734; 1.9982]`. □

10.3 Potential issues

All of the issues of single-variable method in Section 9.2 are present here as well. In particular, the problem with derivative $f'(x)$ being zero (or close to it) now becomes the problem with the Jacobian matrix J being singular (not invertible) or close to being such. This is when Matlab displays the warning "Matrix is close to singular or badly scaled. Results may be inaccurate." Note that despite this warning, the final outcome may be an accurate solution. Indeed, the issue with solving the linear system for $\mathbf{h}$ may result in a wrong value of $\mathbf{h}$ at some step. But this only means that at that particular step, the algorithm jumps where it should not have. It may still converge to a solution afterwards if at the new value of $\mathbf{x}$ the Jacobian matrix behaves better. The script in Example 10.2.3 displays the value $\mathbf{F}(\mathbf{x})$ at the end as a reassurance that the method indeed found a solution, despite the possible problems at intermediate steps.

A new issue is the **scaling** of variables. When the components of unknown vector have very different orders of magnitude, it is hard to understand what changes are significant or not. Suppose we are solving a system involving the distance and speed of New Horizons probe which is traveling at the speed of about 14 km/s and has distance about 6.43×10^9 km from the Sun (as of 2020). So the vector we work with will look like `[14; 6.43e9]`. The difference between this vector and, for example, `[20; 6.43e9]` looks relatively small: the difference has norm about 10^{-9} times the norm of the original vector, one billionth. But of course the difference between 14 and 20 is quite significant. To avoid this, one should try to use units in which the quantities involved have about the same order of magnitude. For New Horizons, one could measure distance in astronomical units (AU) which makes it about 43. So the vector becomes `[14; 43]` which is much better numerically.

10.4 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 10.4.1 Rewriting a polynomial equation in multivariate form. Consider a polynomial equation, for example $x^3 + 2x^2 - 7x + 1 = 0$. This is a scalar equation with one scalar variable, so it could be solved by the methods of previous chapters, including Newton's method in one variable. But then we would get only one root (depending on initial position), not all three. Rewrite this equation as a system of three equations for three roots x_1, x_2, x_3, using the fact that the polynomial factors as $(x - x_1)(x - x_2)(x - x_3)$.

Solution. Expanding the product, we find

$$x^3 + 2x^2 - 7x + 1 = (x - x_1)(x - x_2)(x - x_3) = x^3 + Ax^2 + Bx + C$$

where $A = -(x_1 + x_2 + x_3)$, $B = x_1x_2 + x_1x_3 + x_2x_2$, and $C = -x_1x_2x_3$. Equating the coefficients, we get the system

$$\begin{cases} x_1 + x_2 + x_3 & = -2 \\ x_1x_2 + x_1x_3 + x_2x_3 & = -7 \\ x_1x_2x_3 & = -1 \end{cases}$$

If we can solve this system, the solution will consist of all three roots of the original equation. □

Example 10.4.2 Find all roots of a polynomial simultaneously. Use the multivariate Newton method to solve the system from Example 10.4.1.

Solution. We have the vector function

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} x_1 + x_2 + x_3 + 2 \\ x_1x_2 + x_1x_3 + x_2x_3 + 7 \\ x_1x_2x_3 + 1 \end{pmatrix}$$

and its Jacobian matrix is

$$J(\mathbf{x}) = \begin{pmatrix} 1 & 1 & 1 \\ x_2 + x_3 & x_1 + x_3 & x_1 + x_2 \\ x_2x_3 & x_1x_3 & x_1x_2 \end{pmatrix}$$

Use the code from Example 10.2.3 with these functions:

```
F = @(x) [x(1) + x(2) + x(3) + 2;
          x(1)*x(2) + x(1)*x(3) + x(2)*x(3) + 7;
          x(1)*x(2)*x(3) + 1];
J = @(x) [1, 1, 1;
          x(2) + x(3), x(1) + x(3), x(1) + x(2);
          x(2)*x(3), x(1)*x(3), x(1)*x(2)];
```

The initial point should be a point with distinct coordinates, because at points with equal coordinates like `[0; 0; 0]` or `[1; 1; 1]` the Jacobian matrix is singular (it has equal columns). For example, `[1; 2; 3]` works just fine.

```
Found a solution after 9 steps:
    1.7240    0.1497   -3.8737
```

Although an algebraic formula for the roots of a cubic (degree 3) equation exists, the symbolic form of the roots is often too complicated to be of any use. See what Wolfram Alpha shows as a solution of this equation.

The above process, in a polished and simplified form, is known as the Durand-Kerner method for finding all polynomial roots at once. □

Question 10.4.3 Another cubic equation. If we apply the logic of Example 10.4.2 to the equation $x^3 + 2x^2 - 7x + 5 = 0$ (so, just replace 1 by 5 in the formula for `F`), the method fails to converge. Why? □

Question 10.4.4 A trigonometric system. What goes wrong if we try to use Newton's method to solve the system

$$\begin{cases} \sin x &= a \\ \cos y &= b \\ \tan z &= c \end{cases}$$

for some given values of a, b, c? □

Question 10.4.5 Number of variables different from the number of equations. What prevents us from using Newton's method for systems where the number of equations is not equal to the number of unknowns? For example, the equation $(x-2)^2 + (y+1)^2 = 0$ has a unique solution: can we find it with Newton's method? □

10.5 Homework

1. (Theoretical problem.) Let

$$\mathbf{F}(x,y,z) = \begin{pmatrix} xyz \\ 2(xy+yz+xz) \\ x^2+y^2+z^2 \end{pmatrix}$$

Find the Jacobian matrix of $\mathbf{F}$. (The matrix entries will involve the variables x, y, z).

2. At which of the following points (x, y, z) is the Jacobian matrix from the previous exercise invertible?

$$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 5 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix}$$

To save time, use Matlab: define an anonymous function like `J = @(x,y,z) [... ; ... ; ...]` so that you can then do `rank(J(1,1,1))` and so on, without computing the matrices yourself. Note the difference between this approach to J and the approach of Section 10.2: here the function `J` takes three scalar arguments, instead of one vector argument.

3. Let $V = 36+2a$, $A = 72+3a$, and $D = 7+a/5$ where a is the first digit of your SUID. Write a Matlab script which uses Newton's method to find the dimensions of a rectangular box with volume V, surface area A, and space diagonal D. The computations in Problem 1 will be helpful, because a box with dimensions x, y, z has volume xyz, surface area $2(xy+yz+xz)$, and space diagonal $\sqrt{x^2+y^2+z^2}$.

Chapter 11

Broyden's method

A major difficulty in using multivariate Newton's method is having to specify the Jacobian matrix. Broyden's method offers a workaround by building an approximation to the Jacobian matrix based on the values of the function. It can be viewed as a multivariate form of the secant method.

11.1 Idea of Broyden's method

The basic idea is: since we do not know the derivative, we are going to guess its value and then improve our guess at every step of the algorithm.

Let us return to one-variable situation first. Newton's method proceeds in steps computed as $h = -f(x_0)/f'(x_0)$ which requires the knowledge of $f'(x_0)$. Suppose we do not know the derivative but are willing to guess. For notational convenience, let us say we are guessing the value of $1/f'(x_0)$. We want some non-zero number. Knowing nothing whatsoever, we can guess "1". Let us call this guess $b_0 = 1$. Armed with this number, we proceed as usual in Newton's method: make the step $h = -b_0 f(x_0)$ arriving at the point $x_1 = x_0$.

Our next guess, for the value of $1/f'(x_1)$ can be better educated: we can think of the secant line through the two points $(x_0, f(x_0))$ and $(x_1, f(x_1))$, and decide it would be logical for b_1 to be the reciprocal of the slope of this secant line. This means we pick b_1 so that

$$b_1(f(x_1) - f(x_0)) = x_1 - x_0 \tag{11.1.1}$$

Then the process continues: we make the step $h = -b_1 f(x_1)$, arrive at $x_2 = x_1 + h$ and use the secant slope through the two newest points,

$$b_2(f(x_2) - f(x_1)) = x_2 - x_1$$

Apart from the initial uneducated guess for b_0 this is exactly the secant method, expressed in different notation.

In higher dimensions, we need a guess for the inverse of the Jacobian matrix, which is used in Newton's method $\mathbf{h} = -J^{-1}\mathbf{F}(\mathbf{x})$. At first we take an uneducated guess for the value of $J(x_0)^{-1}$, namely $B_0 = I$, the identity matrix. This enables us to make a step $\mathbf{h} = -B_0 F(\mathbf{x}_0)$. Now comes the important part we want our next guess to be better informed than the previous one. Looking back at (11.1.1), it seems we should choose a matrix B_1 so that

$$B_1(\mathbf{F}(\mathbf{x}_1) - \mathbf{F}(\mathbf{x}_0)) = \mathbf{x}_1 - \mathbf{x}_0$$

How to do that? We cannot "divide a vector by another", the formula

$$B_1 \overset{?}{=} \frac{\mathbf{x}_1 - \mathbf{x}_0}{\mathbf{F}(\mathbf{x}_1) - \mathbf{F}(\mathbf{x}_0)}$$

makes no sense (this is the difference between one variable and several variables.) We need to use linear algebra.

11.2 Outer products

If `a` and `b` are column vectors with n entries, then `a'*b` in Matlab is their scalar product (also called the inner product or the dot product):

$$\begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{pmatrix} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

where multiplication is carried out by the rules of matrix products: 1×n matrix times n×1 matrix gives a 1×1 matrix, a single number. The rules of matrix multiplication also allow us to compute `a*b'`:

$$\begin{pmatrix} a_1 \\ a_2 \\ \cdots \\ a_n \end{pmatrix} \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix} = \begin{pmatrix} a_1 b_1 & a_1 b_2 & \dots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \dots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n b_1 & a_n b_2 & \dots & a_n b_n \end{pmatrix}$$

This matrix is the **outer product** of vectors **a** and **b**. Since all of its rows are proportional to one another, its rank is at most 1. The rank is 0 if one of two vectors is the zero vector; otherwise it is equal to 1.

Example 11.2.1 Random outer product. Let M be the outer product of two random vectors with 5 entries, generated with `rand`. Display the rank and determinant of M.

Solution.

```
a = rand(5, 1);
b = rand(5, 1);
M = a*b';
disp(rank(M))
disp(det(M))
```

The result should be: rank is 1, and the determinant is some extremely small (but nonzero) number, for example `3.2391e-69`. Mathematically this is impossible: a 5×5 matrix of rank less than 5 must have determinant equal to 0. But the reality of computer arithmetic is that floating point numbers rarely add up exactly to zero, as noted in Section 6.4. Matlab's `rank` command takes this into account and reports the rank as 1 when the matrix is "close enough" to actually having rank 1. □

In mathematical notation, the outer product would be written as $\mathbf{a}\mathbf{b}^T$, with T indicating the transposed vector (column turned into row). Since $\mathbf{a}\mathbf{b}^T$ is a matrix, we can apply it to some vector $\mathbf{c}$. The associative property of matrix multiplication shows that

$$(\mathbf{a}\mathbf{b}^T)\mathbf{c} = \mathbf{a}(\mathbf{b}^T\mathbf{c}) = \mathbf{a}(\mathbf{b} \cdot \mathbf{c}) \tag{11.2.1}$$

that is, we get the vector $\mathbf{a}$ multiplied by the dot product $\mathbf{b}\cdot\mathbf{c}$. Formula (11.2.1) gives us an easy way to find a matrix M which satisfies the equation $M\mathbf{c} = \mathbf{d}$ for given vectors $\mathbf{c}, \mathbf{d}$. Namely, we can let

$$M = \frac{1}{\mathbf{b}^T\mathbf{c}}\mathbf{d}\mathbf{b}^T \tag{11.2.2}$$

which is an outer product of two vectors with a scalar factor in front. The associative property shows that

$$M\mathbf{c} = \frac{1}{\mathbf{b}^T\mathbf{c}}\mathbf{d}(\mathbf{b}^T\mathbf{c}) = \mathbf{d}$$

Note that the choice of $\mathbf{b}$ is up to us: any vector will work as $\mathbf{b}$ as long as $\mathbf{b}^T\mathbf{c} \neq 0$.

Once more, to make this point clear: one cannot "solve" $M\mathbf{c} = \mathbf{d}$ for M by "dividing" vector $\mathbf{d}$ by $\mathbf{c}$. But the process outlined above produces some matrix M that satisfies this equation.

11.3 Details of Broyden's method

Recall from Section 11.1 that we seek to improve our guess for the inverse of Jacobian by finding a matrix B_1 such that

$$B_1(\mathbf{F}(\mathbf{x}_1) - \mathbf{F}(\mathbf{x}_0)) = \mathbf{x}_1 - \mathbf{x}_0$$

To simplify notation, let $\mathbf{h} = \mathbf{x}_1 - \mathbf{x}_0$ and $\mathbf{w} = \mathbf{F}(\mathbf{x}_1) - \mathbf{F}(\mathbf{x}_0)$; both these vectors are known. We need

$$B_1\mathbf{w} = \mathbf{h} \tag{11.3.1}$$

Question 11.3.1 Why would not an outer product work here? We already know how to find a matrix that satisfies (11.3.1), by taking an outer product multiplied by a scalar (11.2.2). But this would be a bad, illogical choice for B_1. Why? □

We want B_1 to have some similarity to B_0 in the hope that the process of improving guesses $B_0, B_1, \dots$ will converge to something, rather than just jump around. We want to improve the previous guess, not replace it entirely. So, let $B_1 = (I + M)B_0 = B_0 + MB_0$ where M can be constructed from an outer product as in (11.2.2). Namely, we want

$$B_0\mathbf{w} + MB_0\mathbf{w} = \mathbf{h}$$

so

$$MB_0\mathbf{w} = \mathbf{h} - B_0\mathbf{w}$$

According to (11.2.2) we can achieve this by letting

$$M = \frac{1}{\mathbf{h}^T\mathbf{B}_0\mathbf{w}}(\mathbf{h} - B_0\mathbf{w})\mathbf{h}^T$$

where the chose the term $\mathbf{b}$ to be $\mathbf{h}$. To summarize, the formula for updating our guess for inverse Jacobian is

$$B_1 = B_0 + \frac{1}{\mathbf{h}^T\mathbf{B}_0\mathbf{w}}(\mathbf{h} - B_0\mathbf{w})\mathbf{h}^T B_0$$

It looks messy, but at least the derivation was logical.

The matrix B_1 is still a guess, it may be very different from the actual inverse of Jacobian matrix. But it is a more educated guess, and as this process repeats, these repeated corrections will produce more accurate guesses.

11.4 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 11.4.1 Use Broyden's method to find all roots of a cubic polynomial. Redo Example 10.4.2 using Broyden's method instead of Newton's method.

Solution. We have the same vector functionto be equated to zero

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} x_1 + x_2 + x_3 + 2 \\ x_1x_2 + x_1x_3 + x_2x_3 + 7 \\ x_1x_2x_3 + 1 \end{pmatrix}$$

but no longer use its Jacobian matrix. The code is modified by computing the steps as `h = -B*F(x)` and updating the matrix `B` according to Broyden's method.

```
F = @(x) [x(1) + x(2) + x(3) + 2;
          x(1)*x(2) + x(1)*x(3) + x(2)*x(3) + 7;
          x(1)*x(2)*x(3) + 1];

x = [1; 2; 3];
B = eye(3);
max_tries = 10000;

for k = 1:max_tries
    h = -B*F(x);
    w = F(x+h) - F(x);
    B = B + (h - B*w)*h'*B / (h'*B*w);
    x = x + h;
    if norm(h) < 100*eps(norm(x))
        break
    end
end

if k < max_tries
    fprintf('Found a solution after %d steps:\n', k);
    disp(x)
    disp('The value F(x)')
    disp(F(x))
else
    disp('Failed to converge')
end
```

Starting from the same point `[1; 2; 3]` as we used for Newton's method, we get:

```
Found a solution after 1359 steps:
    1.7240
   -3.8737
    0.1497
```

By contrast, Newton's method converged in 9 steps. There is a price to pay for the lack of Jacobian matrix. □

Question 11.4.2 The role of the starting point. The starting point `[1; 2; 3]` seems unfortunately chosen in Example 11.4.1, because if we start with `[1; 1; 1]` instead, the method converges much faster: in 37 steps. Recall that for Newton's method this starting point did not work at all. What makes the difference? □

Question 11.4.3 The role of the starting matrix. If we keep the starting point `[1; 2; 3]` in Example 11.4.1 but change the initial guess for inverse Jacobian to the matrix

```
B = 0.1*eye(3);
```

the algorithm converges much faster: in 53 steps. Why could this be? In what way does the "smaller" matrix help? Is the factor of 0.1 an under-relaxation parameter here? □

11.5 Homework

1. (Theoretical problem.) Find a matrix A of rank 1 such that

$$A\begin{pmatrix}1\\2\\3\end{pmatrix} = \begin{pmatrix}3\\1\\5\end{pmatrix}$$

2. Use Broyden's method to solve the nonlinear system

$$\begin{cases} x^2+y^2+\sin(x+y) & = a \\ x+y+\cos(xy) & = 5 \end{cases}$$

where a is the number formed by the first two digits of your SUID.

Try at least two different starting points (or more, if needed to find a solution). Report the outcome of running the method for each starting point. Was the root you found always the same?

Part III

Numerical Calculus

Chapter 12

Numerical Differentiation

We already encountered a situation (Newton's method) where it would be useful to be able to find the derivative of a given function numerically. This problem also arises in engineering applications: for example, having measured the position of some object over time, we may want to know its velocity, which is the derivative of position. In this section we study this process, with emphasis on the accuracy of formulas that give an approximate value of various derivatives.

12.1 Definition of derivative and the order of error

The definition of derivative says:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

This suggests a way to approximate $f'(x)$: take two consecutive values of f, subtract, divide by difference of x-values:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{12.1.1}$$

Here the value of h is some concrete number, so the two sides are only approximately equal. One can express this more precisely as

$$\frac{f(x+h) - f(x)}{h} = f'(x) + e(h)$$

where the error term e is a function of h. The accuracy of approximation is measured by the **order** of the error term, which is a number d such that $|e(h)|$ is bounded by some multiple of $|h|^d$.

There is a method to determine the order of error of an approximation to some derivative of f:

1. Let $n = 0$
2. Plug $f(x) = x^n$ into the formula and find the error term.
3. If the error term is some nonzero multiple of a power of h, the exponent of h is the order of the error term.
4. If the error term is zero, increase n by 1 and return to step 2.

Example 12.1.1 Find the order of error of approximation to f'. Find the order of error of the formuls (12.1.1).

Solution.

- With $f(x) = x^0$ we get $f'(x) = 0$ and $\frac{f(x+h)-f(x)}{h} = 0$, so the error term is zero for this function.
- With $f(x) = x^1$ we get $f'(x) = 1$ and $\frac{f(x+h)-f(x)}{h} = \frac{h}{h} = 1$, so the error term is zero for this function.
- With $f(x) = x^2$ we get $f'(x) = 2x$ and $\frac{f(x+h)-f(x)}{h} = \frac{x^2+2xh+h^2}{h} = 2x+h$, so the error term for this function is h^1.

In conclusion, the formula has order of error 1. □

The importance of the order of error can be illustrated by an example. Suppose we want to find $f'(x)$ with absolute error at most 10^{-12}.

- If the error term of our formula is h^1, we need to use extremely small h; that is $|h| \le 10^{-12}$.
- If the error term is h^2, we need $|h| \le 10^{-6}$ which is still small but not as extreme.
- If the error term is h^3, we need $|h| \le 10^{-4}$.
- If the error term is h^4, we need $|h| \le 10^{-3}$.

Thus, a higher-order formula should allow us to obtain an accurate result while avoidung extremely small values of h. But why do we want to avoid extremely small h? This is explained in next section.

12.2 Loss of significance

Loss of significance may occur when we subtract two numbers that are very close to each other. This can occur in any context, not just numeric differentiation. But since differentiation involves expressions like $f(x+h)-f(x)$ with small h, it is particularly vulnerable to the loss of significance.

A simplified illustration of the loss of significance: suppose we can only have three decimal digits, any others have to be rounded. This can still be useful for approximate computations like $9.23+3.45 = 12.7$ or $3.45+100 = 103$ (note the rounding here). These approximate results are relatively close to what we would get with exact arithmetic. But when *subtraction* of nearly equal quantities are involved, the results can be less satisfactory:

$$(3.45 + 100) - 102 = 103 - 102 = 1 \tag{12.2.1}$$

while the exact arithmetic gives 1.45. The error is nearly 50%.

An actual illustration is the following attempt to compute the derivative of $f(x) = \sqrt{x}$ at the point $x = 10^6$. Note that $f(10^6) = 1000$ and $f'(10^6) = 1/2000 = 5 \cdot 10^{-4}$. Both looks like reasonable numbers. Let us try to use the approximation (12.1.1) with the values $h = 10^{-9}, 10^{-10}, 10^{-11}$, in Matlab console:

```
>> x = 1e6;
>> h = 1e-9; (sqrt(x+h) - sqrt(x))/h
ans = 5.6843e-04

>> h = 1e-10; (sqrt(x+h) - sqrt(x))/h
```

```
ans = 0.0011

>> h = 1e-11; (sqrt(x+h) - sqrt(x))/h
ans = 0
```

The first result is off by about 10%, the second is more than twice the real answer, the third is even worse. All this is due to the loss of significant digits similar to what we see in (12.2.1).

Sometimes one can avoid the loss of significance by rearranging the expression algebraically. For example,

$$\frac{\sqrt{x+h}-\sqrt{x}}{h} = \frac{(\sqrt{x+h}-\sqrt{x})(\sqrt{x+h}+\sqrt{x})}{h(\sqrt{x+h}+\sqrt{x})} = \frac{x+h-x}{h(\sqrt{x+h}+\sqrt{x})} = \frac{1}{\sqrt{x+h}+\sqrt{x}}$$

Algebraically these expressions are identical, but in Matlab, `1/(sqrt(x+h) + sqrt(x))` produces accurate answer `5.0000e-04` for any of the above values of h. Matlab also has some built-in logic for avoiding loss of significance in two frequently encountered situations: $e^x - 1$ and $\log(1+x)$ when $x \approx 0$. These are the functions `expm1` and `log1p`. To see the difference, compare the results of `(exp(x)-1)/x` and `expm1(x)/x` when `x=1e-20`.

12.3 Symmetric difference formulas

Although the formula (12.1.1) looks natural, it is not the best way to find the approximate value of $f'(x)$. The following **symmetric difference** formula works better:

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} \tag{12.3.1}$$

The following computation explains why.

Example 12.3.1 Order of error of symmmetric difference approximation to f'. Find the order of error of the formuls (12.3.1).

Solution.

- With $f(x) = x^0$ we get $f'(x) = 0$ and $\frac{f(x+h)-f(x-h)}{2h} = 0$, so the error term is zero for this function.
- With $f(x) = x^1$ we get $f'(x) = 1$ and $\frac{f(x+h)-f(x-h)}{2h} = \frac{2h}{2h} = 1$, so the error term is zero for this function.
- With $f(x) = x^2$ we get $f'(x) = 2x$ and $\frac{f(x+h)-f(x-h)}{2h} = \frac{4xh}{2h} = 2x$, so the error term is zero for this function.
- With $f(x) = x^3$ we get $f'(x) = 3x^2$ and $\frac{f(x+h)-f(x-h)}{2h} = \frac{6x^2h+2h^3}{2h} = 3x^2 + h^2$, so the error term for this function is h^2.

In conclusion, the formula has order of error equal to 2. □

Here is a quick computation in Matlab console which compares the performance of both formulas on the function $f(x) = e^x$ at $x = 0$ with $h = 0.1$. The error of symmetric difference formula is about 30 times smaller in this example, despite the same value of h and the same amount of computations involved.

```
>> x = 0; h = 0.1;
>> (exp(x+h)-exp(x))/h
ans = 1.0517
>> (exp(x+h)-exp(x-h))/(2*h)
ans = 1.0017
```

The *second order derivative* also has a convenient symmetric formula:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{12.3.2}$$

One can derive many other formulas for derivative of any order k, starting with Taylor series for f:

$$f(x+\delta) = f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \frac{f'''(x)}{6}\delta^3 + \cdots \tag{12.3.3}$$

One can use (12.3.3) with values like $\delta = h$, $\delta = -h$, $\delta = 0$, or maybe $\delta = 2h$, and so on; and then take a linear combination of these expressions so that the derivatives of orders less than k cancel out, while the derivative of order k remains. The order of error of this formula will depends on what derivative of order greater than k remain in the linear combination. To have order of error n, one needs to make sure that the derivatives of orders between k and $k+n$ also cancel out.

12.4 Richardson extrapolation

Richardson extrapolation is a method that allows one to reduce the error of an approximate formula, provided that the order of this error is known. The idea is to use two different step sizes, for example h and $2h$. If the order of error is n, then the second approximation will have an error that is about 2^n times greater. How can we use this information?

Let Q be the quantity we want to find. Using step size h we get an approximation A_1 with error term E_1, meaning that

$$Q = A_1 + E_1 \tag{12.4.1}$$

Using step size $2h$ we get an approximation A_2 with error term E_2, meaning that

$$Q = A_2 + E_2 \tag{12.4.2}$$

Since we know that E_2 is close to $2^n E_1$, we can try to cancel out the error terms: multiply (12.4.1) by 2^n and subtract (12.4.2). The result is

$$(2^n - 1)Q = 2^n A_1 - A_2 + [2^n E_1 - E_2]$$

Since the error terms should almost cancel each other, the new approximation

$$Q \approx \frac{2^n A_1 - A_2}{2^n - 1} \tag{12.4.3}$$

should be much more accurate than the original approximation (12.4.1). The formula (12.4.3) expresses the method of Richardson extrapolation. This idea is not specific to numerical differentiation; it can be used any time a computed quantity has an error proportional to a known power of step size.

Example 12.4.1 Apply Richardson extrapolation to the symmetric formula for f'. Use Richardson extrapolation to improve the accuracy of the symmetric difference formula (12.3.1).

Solution. Recall from Example 12.3.1 that the symmetric difference formula has error of order 2. Therefore, in the equation (12.4.3) we have 2. The approximation A_1 is the original formula

$$A_1 = \frac{f(x+h) - f(x-h)}{2h}$$

To get A_2 we replace h by $2h$:

$$A_2 = \frac{f(x+2h) - f(x-2h)}{4h}$$

The formula (12.4.3) now tells us that

$$f'(x) \approx \frac{4A_1 - A_2}{3} = 2\frac{f(x+h) - f(x-h)}{3h} - \frac{f(x+2h) - f(x-2h)}{12h} \qquad (12.4.4)$$

which is the extrapolated approximation to $f'(x)$. □

12.5 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 12.5.1 Rewrite a formula to avoid the loss of significance. The formula $(1 - \cos(x))/x^2$ is prone to loss of significance when x is close to 0. Rewrite it in a mathematically equivalent way which avoids this issue. Compare the performance of both formulas in Matlab with $x = 10^{-9}$.

Solution. The issue is that $\cos x$ is close to 1 and we subtract it from 1. The identity

$$(1 - \cos x)(1 + \cos x) = 1 - \cos^2 x = \sin^2 x$$

can be used to avoid this issue. According to it,

$$\frac{1 - \cos(x)}{x^2} = \frac{\sin^2(x)}{x^2(1 + \cos x)}$$

There is no loss of significance in the formula on the right. Compare in Matlab:

```
f = @(x) (1-cos(x))/x^2;
g = @(x) sin(x)^2/(x^2*(1+cos(x)));
disp(f(1e-9))
disp(g(1e-9))
```

The first formula produces 0, the second produces 0.5. The second value is correct. Indeed,

$$\frac{1 - \cos(x)}{x^2} \to \frac{1}{2} \quad \text{as } x \to 0$$

□

Example 12.5.2 Compare the accuracy of three approximations to f'. Consider the function $f(x) = \sin x$ on the interval $[0, 5]$. Compare the accuracy of the three approximations to its derivative ((12.1.1), (12.3.1), (12.4.4)) by plotting the *base 10 logarithm* of the error for each of them. Use $h = 0.1$.

Solution.

```
f = @(x) sin(x);
fp = @(x) cos(x);
```

```
x = linspace(0, 5, 500);
h = 0.1;

original = (f(x+h) - f(x))/h;
symmetric = (f(x+h) - f(x-h))/(2*h);
extrapolated = 2*(f(x+h) - f(x-h))/(3*h) - (f(x+2*h) - f(x-2*h))/(12*h);
exact = fp(x);

hold on
plot(x, log10(abs(original - exact)))
plot(x, log10(abs(symmetric - exact)))
plot(x, log10(abs(extrapolated - exact)))
legend('original', 'symmetric', 'extrapolated')
hold off
```

The result, shown below, indicates that the error is about 10^{-1} for the original formula (from the definition of derivative), about 10^{-3} for the symmetric formula, and about 10^{-6} for the extrapolated formula.

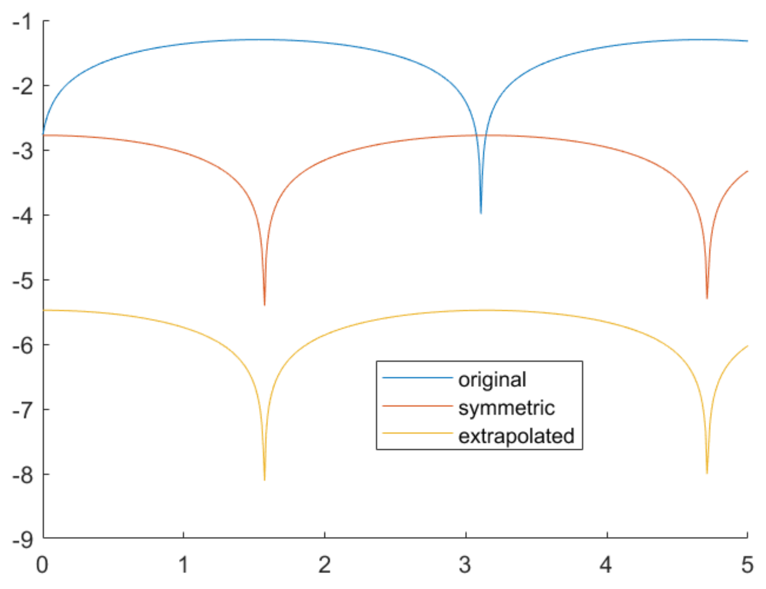

Figure 12.5.3 Comparison of errors

Note that Matlab's command `semilogy` can be used to plot on logarithmic scale directly, for example

```
semilogy(x, abs(original - exact))
```

has the same result as

```
plot(x, log10(abs(original - exact)))
```

□

Question 12.5.4 What makes the error smaller in some places? As we see in Figure 12.5.3, the size of error suddently becomes much smaller in certain parts of the interval. What is special about those places that makes the approximation more accurate?**Hint**. Recall the connection with Taylor series: the error term involves derivatives of higher order. □

12.6 Homework

1. Find the order of error of the formula (12.3.2) (theoretical exercise).

2. Rewrite the formula

$$\frac{\sqrt{9+x^4}-3}{\sin(x^4)}$$

to avoid the loss of significance when x is close to 0. Evaluate both the original and rewritten formulas when $x = 10^{-4}$. How different are the results?

3. A function f has been evaluated at the points $0, 0.1, 0.2, \dots, 1$ (in Matlab notation, `x = 0:0.1:1`). Its values are

```
y = [1, 0.99, 0.96, 0.91, 0.85, 0.78, 0.7, 0.61, 0.53, 0.44, 0.37]
```

Write a script which plots this function together with its first derivative f' and its second derivative f''. Use symmetric difference formulas with $h = 0.1$.

Hint. Expressions like `y(3:end) - y(1:end-2)` will be useful. Note that while the values of f can be plotted against the given vector `x`, the derivatives need to be plotted against `x(2:end-1)` because the symmetric formulas do not apply at the endpoints. Recall we can combine several functions in one plot command like `plot(x, y, xx, yy)`.

Chapter 13

Numerical integration: basic rules

We begin the study of numerical integration: the problem of finding, approximately, the definite integral of a given function on a given interval. This chapter covers basic rules of numerical integration that are covered in Calculus books.

13.1 Riemann sums

Numerical integration problem amounts to the following: we have a function f on some interval $[a, b]$, and we try to compute $\int_a^b f(x)\,dx$ based on evaluating f at some points of this interval. To begin with, recall that by definition, $\int_a^b f(x)\,dx$ is the limit of **Riemann sums** of f. The process of computing a Riemann sum is as follows.

1. Divide $[a, b]$ into subintervals. It can be divided into n subintervals by choosing some points $x_0 < x_1 < x_2 < \cdots < x_n$ so that $x_0 = a$ and $x_n = b$. The subintervals are $I_k = [x_{k-1}, x_k]$ where $k = 1, 2, \dots, n$
2. Evaluate f at some point s_k (a sample point) of each subinterval I_k.
3. Compute the sum of the products $f(s_k)(x_k - x_{k-1})$ for $k = 1, \dots, n$. This is the Riemann sum:

$$S = \sum_{k=1}^{n} f(s_k)(x_k - x_{k-1})$$

Therefore, a basic strategy of numeric integration consists of choosing subintervals and then choosing a point in each.

The simplest choice of subintervals is to have subintervals of equal length, meaning each has length $h = (b - a)/n$, which makes $x_k = a + kh$. We will eventually see that this choice is not as good as it seems, but it will do for now.

13.2 Left, right, midpoint rules

The three rules considered here use equal subintervals, but different choices of same points. In a subinterval $[x_{k-1}, x_k]$ we have three natural choices for a sample point: left endpoint x_{k-1}, right endpoint x_k, and midpoint $\bar{x}_k = (x_{k-1}+x_k)/2$. They give us three ways of approximating the integral:

- Left endpoint approximation $L_n = h(f(x_0) + f(x_1) + \cdots + f(x_{n-1}))$
- Right endpoint approximation $R_n = h(f(x_1) + f(x_2) + \cdots + f(x_n))$
- Midpoint Rule: $M_n = h(f(\bar{x}_1) + f(\bar{x}_2) + \cdots + f(\bar{x}_n))$

Example 13.2.1 Computing with left- right- and midpoint rules. For the function $f(x) = \sin(x^2)$ on the interval $[1,3]$, use Matlab to compute the left-, right, and midpoint approximations with $n = 10$

Solution.
```
f = @(x) sin(x.^2);
a = 1;
b = 3;
n = 10;
h = (b-a)/n;
x = a:h:b;

L = sum(f(x(1:end-1)))*h;
R = sum(f(x(2:end)))*h;
midpoints = (x(1:end-1) + x(2:end))/2;
M = sum(f(midpoints))*h;
fprintf('Leftpoint %g, Rightpoint %g, Midpoint %g\n', L, R, M);
```

The output is "Leftpoint 0.483958, Rightpoint 0.398087, Midpoint 0.474599". (One can check with WolframAlpha that the actual value of the integral is about 0.4633.) □

13.3 Trapezoidal rule

When a function is increasing, the leftpoint rule underestimates its integral, and rightpoint rule overestimates it. So, their average $T_n = (L_n+R_n)/2$ should be a better approximation. This is the **Trapezoidal Rule**:

$$T_n = \frac{h}{2}(f(a) + f(b)) + h(f(x_1) + \cdots + f(x_{n-1}))$$

In Matlab, the computation is similar to Example 13.2.1

```
  T = (f(x(1)) + f(x(end))) * h/2 + sum(f(x(2:end-1)))*h;
```

It may be simpler to compute the vector `y = f(x)` first, which reduces the number of calls to function `f` and the number of parentheses:

```
      T = (y(1) + y(end)) * h/2 + sum(y(2:end-1))*h;
```

Example 13.3.1 Comparing the accuracy of four rules. For the function $f(x) = e^x$ on the interval $[-1,1]$, use Matlab to compute the left-, right, and midpoint approximations with $n = 10$ and find the error of each approximation (that is, its difference with the actual integral).

Solution.
```
f = @(x) exp(x);
a = -1;
```

```
b = 1;
n = 10;
h = (b-a)/n;
x = a:h:b;
y = f(x);

L = sum(y(1:end-1))*h;
R = sum(y(2:end))*h;
T = (y(1) + y(end)) * h/2 + sum(y(2:end-1))*h;
midpoints = (x(1:end-1) + x(2:end))/2;
M = sum(f(midpoints))*h;

exact = exp(1)-exp(-1);
er = abs([L R T M] - exact);
fprintf('Errors: Leftpoint %g, Rightpoint %g, Trapezoidal %g, Midpoint %g\n', er);
```

Here `fprintf` "unpacks" the vector `er`, inserting its entries in the right places.

□

13.4 Error estimates

How to estimate the accuracy of integration rules? Here is an empirical method which is similar to how we estimated the accuracy of differentiation rules in Section 12.1.

There is a method to determine the order of error of an approximation to some derivative of f:

1. Let $d = 0$
2. Apply the integration rule to $f(x) = x^d$ on the interval $[0, h]$, with the smallest possible number of evaluation points. Find the error term.
3. If the error term is zero, increase d by 1 and return to step 2.
4. Express the error as $C\, d!\, h^{d+1}$ with an explicit constant C. Note that the factorial $d!$ is here because it is the derivative of f of order d, which is the source of the error.
5. The error bound for a single subinterval is $C \max |f^{(d)}| h^{d+1}$.
6. When the estimate is applied on an interval $[a, b]$ with step size $h = (b - a)/n$, the error estimate is multiplied by n; thus the final result is $C \max |f^{(d)}|(b - a)h^d$.

Example 13.4.1 The error of midpoint rule. Using the process describe above, estimate the error of the midpoint rule.

Solution. The midpoint rule is exact for x^0 and x^1. But for $\int_0^h x^2\,dx$ it predicts $\int_0^h x^2\,dx = h^3/4$, while the true value is $h^3/3$. So we stop at $d = 2$ with the error of $h^3/12$, which can be written as $\dfrac{1}{24}\ 2!\ h^3$. Therefore, the constant factor in the error formula: is $C = 1/24$. In conclusion, the error of the midpoint rule is at most

$$|\text{error}| \le \frac{1}{24} \max_{[a,b]} |f''|(b - a)h^2$$

□

13.5 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking “Answer”. There are also questions for reviewing the concepts of this section.

Example 13.5.1 The error of rightpoint rule. Estimate the error of the rightpoint rule.

Solution. The rightpoint rule is exact for x^0. But for $\int_0^h x^1\,dx$ it predicts $\int_0^h x^1\,dx = h^2$, while the true value is $h^2/2$. So we stop at $d = 1$ with the error of $h^2/2$, which cam be written as $\frac{1}{2}\;1!\;h^2$. Therefore, the constant factor in the error formula: is $C = 1/2$. In conclusion, the error of the rightpoint rule is at most

$$|\text{error}| \le \frac{1}{2}\max_{[a,b]}|f'|(b-a)h$$

□

Example 13.5.2 Using substitution to evaluate an improper integral. The integral

$$\int_1^\infty \frac{e^{-x}}{x}\,dx$$

is improper (of first kind, meaning the interval of integration is infinite). Having an integration limit $b = \infty$ is not good for any of the integration rules in this chapter. Make a substitution to transform this integral into a proper one.

Solution. The substitution $u = e^{-x}$ works well here. Since $du = -e^{-x}\,dx$ and $x = -\log u$, it follows that

$$\int_1^\infty \frac{e^{-x}}{x}\,dx = -\int_0^{1/e} \frac{1}{\log u}\,du$$

The integral on the right is proper. As $u \to 0+$, the logarithm tends to negative infinity, but this only makes the integrand zero; so it is a continuous function on the interval of integration.

Note that the substitution does not bring us any closer to the symbolic solution, since $1/\log u$ cannot be integrated with calculus methods either. But it makes a more convenient integral for numerical methods. □

Question 13.5.3 Failure of error estimates. The integral

$$\int_0^{1/e} \frac{1}{\log u}\,du$$

could be approximated using any of the rules in this chapter. The endpoint 0 is not a problem, because Matlab is okay with `1/log(0)`: it simply evaluates it as 0. But it is still a good idea to avoid this endpoint, so the midpoint rule is preferable. Try it with $n = 5$, for example. Then try to estimate the error of this rule using the formula in Example 13.4.1. What goes wrong, and what does this mean for the approximation? □

Question 13.5.4 An improper integral of second kind. The integral $\int_0^1 x^{-1/2}\,dx$ is improper (of second kind) but it converges. Which of the above integration rules could be used to evaluate this integral? Which of those rules would you use? □

13.6 Homework

1. Estimate the error of the trapezoidal rule following the procedure in Section 13.4 (theoretical exercise).

2. Calculate an approximate value of π using from the formula

$$\int_0^\infty \sin(x^2)\,dx = \sqrt{\frac{\pi}{8}}$$

Two approximations must be made here: replacing ∞ in the integral with a large number b, and using the *trapezoidal* rule with a large number of subintervals n. While avoiding extremely large b and n that might freeze your computer, calculate the value of integral I with enough precision so that `round(8*I^2, 2)` (rounding to 2 places) produces 3.14, a reasonable approximation to π.

Questions to answer: (a) What values of b and n did you use to achieve 3.14? (b) If you increase b while keeping n fixed, does the answer become more precise or less precise? Why? (Hint: think of the error estimate formula)

Chapter 14

Simpson's rule and other Newton-Cotes rules

This chapter introduces more sophisticated integration rules which attach different weights to different points of evaluation. However, the evaluation points are still chosen in a simple way, by putting them at equal distance from one another. This will eventually become a problem.

14.1 Simpson's rule

Consider the trapezoidal rule on the interval $[-1, 1]$ with $n = 2$. It has $h = 1$, so

$$\int_{-1}^{1} f(x)\,dx \approx f(0) + \frac{1}{2}(f(-1) + f(1)) \tag{14.1.1}$$

Surprisingly, the approximation (14.1.1) can be made much more precise without doing more computations. As in Section 12.4, we use the Richardson extrapolation to improve the accuracy. The same trapezoidal rule with doubled step size $h = 2$ yields

$$\int_{-1}^{1} f(x)\,dx \approx \frac{2}{2}(f(-1) + f(1)) \tag{14.1.2}$$

Since the error of trapezoidal rule is proportional to h^2 (the rule has second order of accuracy), we expect the second approximation (14.1.2) to have error 4 times greater than the first approximation (14.1.1). To cancel out most of the error, we divide (14.1.2) by 4 and subtract it from (14.1.1). The result is

$$\frac{3}{4}\int_{-1}^{1} f(x)\,dx \approx f(0) + \frac{1}{2}(f(-1) + f(1)) - \frac{1}{4}(f(-1) + f(1))$$

Hence

$$\int_{-1}^{1} f(x)\,dx \approx \frac{4}{3}f(0) + \frac{1}{3}(f(-1) + f(1)) \tag{14.1.3}$$

The approximation (14.1.3) uses the same information about function f as the trapezoidal rule (14.1.1): the values at -1, 0, 1. But the result is much more accurate. Consider, for example, the integral $\int_{-1}^{1} e^x\,dx = e - 1/e \approx 2.3504$. The trapezoidal rule (14.1.1) approximates it by

$$e^0 + \frac{1}{2}(e^{-1} + e^1) \approx 2.5431$$

with the error of 0.1927. The formula (14.1.3) produces

$$\frac{4}{3}e^0 + \frac{1}{3}(e^{-1} + e^1) \approx 2.3621$$

with the error of 0.0117. This is 16 times more accurate. This improvement was achieved by changing the **weights** (coefficients) attached to the values $f(-1), f(0), f(1)$: for trapezoidal rule they are $1/2, 1, 1/2$, but the formula (14.1.3) uses the weights $1/3, 4/3, 1/3$. The formula (14.1.3) is Simpson's rule (with $n = 2$ subintervals) on the interval $[-1, 1]$. On a general interval $[a, b]$ it becomes, via a change of variable,

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}(f(a) + 4f((a+b)/2) + f(b)) \tag{14.1.4}$$

Check that (14.1.4) is consistent with (14.1.3): the midpoint gets greater weight than the endpoints by the factor of 4, and the sum of weights is $b - a$.

Simpson's rule can be used with more subintervals, but their number n has to be even, since we are combining trapezoidal rule with n and $n/2$ subintervals. With $h = (b - a)/n$ we get

$$\int_a^b f(x)\,dx \approx \frac{h}{3}(1, 4, 2, 4, \dots, 2, 4, 1) \cdot (\text{vector of y-values}) \tag{14.1.5}$$

A better way to think of this formula is that we first divide the interval into some number of equal parts and then use Simpson's rule on each part. The result is called "composite Simpson's rule" in contrast to the "simple Simpson's rule" that we started with.

To summarize: Simpson's rule is the result of Richardson extrapolation of the Trapezoidal rule. Its order of accuracy is 4, meaning the error is proportional to h^4.

14.2 Newton-Cotes rules

One can create rules of any given order of accuracy, using enough sample points. Usually such rules are derived for the convenient interval $[-1, 1]$, and then applied to other intervals by change of variable.

Let $(x_1, \dots, x_n)$ be some points in the interval $[-1, 1]$. We want to come up with an integration rule

$$\int_{-1}^1 f(x)\,dx \approx w_1 f(x_1) + \cdots + w_n f(x_n) \tag{14.2.1}$$

where w_k are coefficients (weights). For example, Simpson's rule (in its simple form) has evaluation points $-1, 0, 1$ and weights $1/3, 4/3, 1/3$.

How to find the weights that make the rule (14.2.1) as accurate as possible? We can require both sides to be equal when f is each of the following: $x^0, x^1, \dots, x^{n-1}$. This gives n linear equations with n unknowns w_j. Then solve the system, perhaps using Matlab. The resulting rules are called Newton-Cotes rules when the points are distributed at equal intervals. They include trapezoidal rule (2 points) and Simpson's rule (3 points) as special cases.

Example 14.2.1 Find the weights for a 5-point Newton-Cotes formula. Let evaluation points on $[-1, 1]$ be $-1, -1/2, 0, 1/2, 1$. What will their weights be?

Solution. We need 5 linear equations involving integrals of $x^0, x^1, \dots, x^4$.

If we number them by index i running from 1 to 5, then the equation has $\int_{-1}^{1} x^{i-1}\,dx$ on the right hand side, which is $(1-(-1)^i)/i$. On the left we have the sum $\sum_j w_j x_j^{i-1}$ which means the coefficients of the unknown w_j is x_j^{i-1}. We can create a matrix using outer exponentiation of a row vector by a column vector.

```
n = 5;
x = linspace(-1, 1, n);
i = (1:n)';
A = x.^(i-1);
b = (1-(-1).^i)./i;
w = A\b;
disp(rats(w'))
```

Here `rats` is used to display the solution as rational numbers (which they are) rather than the usual decimal approximation.

Having computed the weights as above, we can integrate any function f on [-1, 1] simply by executing `f(x)*w`, which is the dot product of a vector with function values with the vector of weights. Try this for the exponential function. □

Although one can get rules of arbitrarily high order in this way, the gain in practice does not justify the increasing complexity. (Try computing the 9-point rule, for example). It turns out that using many equally spaced points on an interval is generally a bad idea. We need a smarter way of choosing the points, which we will start working on next time.

14.3 Homework

1. Sometimes one needs an "open-ended" version of Newton-Cotes rules, which does not use the values of the function at the endpoints (because the function may not be defined at the endpoints). Find the coefficients in the integration rule

$$\int_{-1}^{1} f(x)\,dx \approx w_1 f(-1/2) + w_2 f(0) + w_3 f(1/2)$$

so that the rule is precise for x^0, x^1, and x^2. (You can use Matlab for solving the system of equations for the weights.)

2. Apply the integration rule derived in the previous exercise to $\int_{-1}^{1}(1-x^2)^{-1/4}\,dx$. How close is the result to the actual value of the integral (you can find it, for example, with WolframAlpha)?

3. In a loop over `n=2:10`, do the following. First, compute the weights for n-point Newton-Cotes rule (of the standard type, as in Example 14.2.1). Then use this rule to approximate $\int_{-1}^{1} e^x\,dx$. After the loop ends you should have 9 approximations to this integral. Show how their accuracy changes with n by plotting the logarithm of absolute value of the difference between the integral and approximation, as a function of n.

Chapter 15

Legendre polynomials and Laguerre polynomials

In order to design better integration rules than Newton-Cotes rules found so far, we need more mathematical tools. This chapter introduces two families of **orthogonal polynomials**, which appear in many areas of mathematics. The roots of these polynomials will later be shown to be optimal points of evaluation for integration rules.

15.1 Motivation: search for better evaluation points

What is the most accurate approximation to $\int_{-1}^{1} f(x)\,dx$ using two points of evaluations? So far we saw the trapezoidal rule, which is $\int_{-1}^{1} f(x)\,dx \approx f(-1)+f(1)$. For example, if $f(x) = e^x$ we have $\int_{-1}^{1} f(x)\,dx = 2.3504$ while $f(-1)+f(1) = 3.0862$, so the rule makes a significant error, overestimating the integral by 0.7358. The concavity of the function is responsible for this error.

It turns out there is a much more accurate two-point integration rule:

$$\int_{-1}^{1} f(x)\,dx \approx f(-1/\sqrt{3}) + f(1/\sqrt{3})$$

For the exponential function it gives 2.3427, the error of only 0.0077. This is about 100 times more accurate than the trapezoidal rule, with the same number of function evaluations.

Where does $1/\sqrt{3}$ come from? It will take some time to develop a general theory leading to this choice, but a quick way to justify it is to consider small powers of x:

- If $f(x) = x^0$, then $\int_{-1}^{1} x^0\,dx = 2 = f(x_1) + f(x_2)$ holds for any choice of points x_1, x_2
- If $f(x) = x^1$, then $\int_{-1}^{1} x^1\,dx = 0 = f(x_1) + f(x_2)$ holds whenever the points are symmetric about 0, meaning that $x_2 = -x_1$.
- If $f(x) = x^2$, then $\int_{-1}^{1} x^2\,dx = 2/3$ and $f(x_1) + f(x_2) = 2x_1^2$. The two things are equal when $x_1 = 1/\sqrt{3}$.

As a bonus, the rule is also accurate for $f(x) = x^3$, by virtue of symmetry, so its order of accuracy is 4.

There is a general method of choosing evaluation points $x_1, \dots, x_n$ in an optimal way: they will be the roots of some special n-th degree polynomial. This method is called Gaussian integration and it is built into TI-83 calculators and many software packages, including Matlab (`quadgk` command). But before we get to actual integration, we have to study the orthogonal polynomials themselves. The process of using them for integration will be covered in next chapter.

15.2 Legendre polynomials

The idea of orthogonality can be taken from linear algebra and applied to functions. Two functions f and g are orthogonal on an interval $[a, b]$ if their *inner product* $\int_a^b f(x)g(x)\,dx$ is zero: $\int_a^b f(x)g(x)\,dx = 0$. For example, 1 and x are orthogonal on $[-1, 1]$. And x^2 is orthogonal to x on this interval, but it is not orthogonal to 1. Is there a second-degree polynomial that is orthogonal to *both* 1 and x? Yes, there is: $p(x) = x^2 - 1/3$ (check this).

This process continues indefinitely: for every positive integer n we can find a polynomial of degree n, say P_n, which is orthogonal to all polynomials of degrees up to $n-1$, by the Gram-Schmidt algorithm. The roots of this polynomial P_n turn out to be contained in the interval $[a, b]$. These roots will shown to be useful as the evaluation/sample points.

In the above process, the polynomials P_n are determined up to a multiplicative constant. It is convenient to normalize them by requiring $P_n(1) = 1$. This makes P_n the unique n-th degree polynomial such that $\int_{-1}^1 x^k P_n(x)\,dx = 0$ for $k = 0, 1, \dots, n-1$, and $P_n(1) = 1$. Note that $\int_{-1}^1 Q(x)P_n(x)\,dx = 0$ for every polynomial Q with degree $\deg Q < n$.

The polynomials P_n are called **Legendre polynomials**, and they come up in a number of contexts. They originated in physics: in a 1782 paper by Legendre, he used them to expand the electrostatic potential into a power series. A practical way to compute Legendre polynomials is the recursive formula

$$P_{n+1}(x) = \frac{(2n+1)xP_n(x) - nP_{n-1}(x)}{n+1} \tag{15.2.1}$$

which can be used to compute all these polynomials starting with $P_0(x) = 1$ and $P_1(x) = x$.

Example 15.2.1 Using the recursive formula for Legendre polynomials. Using the formula (15.2.1), find the polynomials P_n for $n = 2, 3, 4$. What are their roots? □

Question 15.2.2 Properties of Legendre polynomials. What patterns in the polynomials P_n do you observe on the basis of the examples computed above? □

15.3 Laguerre polynomials

In an earlier homework we found that finding integrals of the form $\int_0^\infty f(x)\,dx$ is hard: we had to pick some large upper limit b (mostly by guessing) to replace the infinite upper limit. Otherwise, there was no way to choose evaluation points by placing them at equal distances throughout the interval.

But now we have a way of choosing evaluation points differently, as roots of orthogonal polynomials. So we can try to integrate on an infinite interval by using orthogonal polynomials on such an interval.

However, on an infinite interval we cannot define the inner product of polynomials in the sense of the integral $\int_0^\infty fg$, the integral will diverge. Instead we define dot product as $\int_0^\infty f(x)g(x)e^{-x}\,dx$, introducing a **weight** e^{-x} which allows the integral to converge. This way we can build orthogonal polynomials: for example, 1 and $1-x$ are orthogonal (check).

The **Laguerre polynomial** of degree n, denoted L_n, is the unique n-th degree polynomial such that $\int_0^\infty x^k L_n(x)e^{-x}\,dx = 0$ for $k = 0, 1, \dots, n-1$, and $L_n(0) = 1$. Note that $\int_0^\infty Q(x)P_n(x)e^{-x}\,dx = 0$ for every polynomial Q with degree $\deg Q < n$.

Similarly to Legendre polynomials, the Laguerre polynomials have a recursive formula:

$$L_{n+1}(x) = \frac{(2n+1-x)L_n(x) - nL_{n-1}(x)}{n+1} \tag{15.3.1}$$

which can be used to compute them starting with $L_0(x) = 1$ and $L_1(x) = 1-x$.

Example 15.3.1 Using the recursive formula for Laguerre polynomials. Using the formula (15.3.1), find the polynomials L_n for $n = 2, 3$. What are their roots? □

Question 15.3.2 Properties of Laguerre polynomials. What patterns in the polynomials L_n do you observe on the basis of the examples computed above? □

15.4 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 15.4.1 Computing and plotting Legendre polynomials. Recursively find the coefficients and roots of Legendre polynomials of degrees up to 10. Plot all of them on the interval $[-1, 1]$. The polynomials should be represented as vectors of coefficients, for example `q = [1 0]` represents $q(x) = 1x + 0 = x$. Once the coefficients are computed, one can use `polyval(q, x)` to evaluate the polynomial at every point of vector `x`, so that, for example, `plot(x, polyval(q, x))` can be used to plot it.

Answer. We have to start with `[1]` and `[1 0]` which represent 1 and x. Once a vector of coefficients `q` exists, one can use concatenation `[q 0]` to shift the coefficients to the left, representing multiplication by x.

```
p = [1];
q = [1 0];
x = linspace(-1, 1, 1000);
hold on
plot(x, polyval(p, x), x, polyval(q, x));

for n = 1:9
    r = ((2*n+1)*[q 0] - n*[0 0 p])/(n+1);
    p = q;
    q = r;
    disp(r);
    plot(x, polyval(r, x))
end
hold off
```

The loop runs up to $n = 9$ because the polynomial being computed has degree

$m+1$. In the loop, `q` is a polynomial of degree n and `p` is a polynomial of degree $n-1$. Concatenation `[0 0 p]` adds zero terms to the latter polynomial; this does not change it mathematically but makes it possible to do vector addition of coefficients. □

Example 15.4.2 The roots of Legendre polynomials. Adapt the code in Example 15.4.1 to find and plot the roots of Legendre polynomials of degrees 2 through 10.

Answer. The command `roots` can be used to find the roots. A convenient way to plot them so that one can tell the difference between different degrees is to use the y-coordinate for the degree.

```
p = [1];
q = [1 0];
hold on
for n = 1:9
    r = ((2*n+1)*[q 0] - n*[0 0 p])/(n+1);
    p = q;
    q = r;
    plot(roots(r), n+1, 'b*');
end
hold off
```

□

Question 15.4.3 Patterns in the roots of Legendre polynomials. What patterns do you observe in the roots found in Example 15.4.2? □

15.5 Homework

1. Adapt Example 15.4.1 to Laguerre polynomials. That is, compute and plot Laguerre polynomials up to degree 10. Use $[0,5]$ as an interval for plotting since we cannot plot on $[0,\infty)$. Also, comment on some pattern you observe in their behavior.

Hint. Note that the degree 1 polynomial is no longer `q = [1 0]`. Also, the recursive formula needs to be changed. When changing it, think of $(2n+1-x)L_n(x)-nL_{n-1}(x)$ as $(2n+1)L_n(x)-xL_n(x)-nL_{n-1}(x)$ and find the vector of coefficients for each of these three terms.

2. Adapt Example 15.4.2 to Laguerre polynomials. That is, compute and plot the roots of Laguerre polynomials of degrees 2 to 10. Also, comment on some pattern you observe in their behavior.

Chapter 16

Gauss-Legendre and Gauss-Laguerre integration

We use the orthogonal polynomials developed in the previous section to formulate sophisticated integration rules which are extremely accurate for smooth functions.

16.1 Brief recap of orthogonal polynomials

Last time we studied Legendre polynomials P_n and Laguerre polynomials L_n. They have degree n, $n = 0, 1, 2 \dots$ and their main property is orthogonality, which means different things for different families of orthogonal polynomials.

- $\int_{-1}^{1} Q(x)P_n(x)\,dx = 0$ where P_n is the Legendre polynomial of degree n and Q is any polynomial of degree $< n$.
- $\int_0^\infty Q(x)L_n(x)e^{-x}\,dx = 0$ where L_n is the Laguerre polynomial of degree n and Q is any polynomial of degree $< n$.

A general fact about orthogonal polynomials is that all their roots are real and are contained in the interval over which they are orthogonal: $[-1, 1]$ for Legendre polynomials and $[0, \infty)$ for Laguerre polynomials.

As an aside, there are many other important families of orthogonal polynomials which we do not study, for example Hermite polynomials H_n which are orthogonal on the entire line $(-\infty, \infty)$ with the weight $\exp(-x^2)$. The theory of orthogonal polynomials is a vast subject.

16.2 Gauss-Legendre integration

The **Gauss-Legendre integration rule** is to let the evaluation points $x_1, \dots, x_n$ be the roots of P_n and then choose their weights $w_1, \dots, w_n$ so that the rule

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{k=1}^{n} w_k f(x_k)$$

is exact for $f(x) = x^p$, $p = 0, \dots, n-1$. The second part is not new; this is exactly what we did for Newton-Cotes rules. But with this specific choice of evaluation points, the integration rule is exact for all polynomials of degrees

$\leq 2n-1$, not just $\leq n-1$. This means the Gauss-Legendre integration has order of accuracy $2n$: for example, it has order 14 if we use 7 points.

Proof. Given a polynomial F of degree $\leq 2n-1$, use long division of polynomials to write it as $F = QP_n + R$ with Q and R having degree $\leq n-1$ (why is this possible?). Plug this into the integral:

$$\int_{-1}^{1} F(x)\,dx = \int_{-1}^{1} (Q(x)P_n(x) + R(x))\,dx = \int_{-1}^{1} R(x)\,dx$$

where the term QP_n integrates to zero because of orthogonality. Since $\deg R \leq n-1$, the Gauss-Legendre integration rule is exact for R. Therefore,

$$\int_{-1}^{1} R(x)\,dx = \sum_{k=1}^{n} w_k R(x_k) = \sum_{k=1}^{n} w_k F(x_k)$$

where the last step uses the fact that $F(x_k) = R(x_k)$ by virtue of $P_n(x_k) = 0$. This completes the proof that

$$\int_{-1}^{1} F(x)\,dx = \sum_{k=1}^{n} w_k F(x_k)$$

It turns out the Gauss-Legendre integration rule has the best possible order of accuracy. That is, there is no integration rule with n evaluation points that is exact for all polynomials of degrees $\leq 2n$. Indeed, suppose such a rule exists, with some evaluation points $x_1, \dots, x_n$. Let $F(x) = (x-x_1)^2 \cdots (x-x_n)^2$ which is a polynomial of degree $2n$. Since $F(x_k) = 0$ for every k, the evaluation rule will produce 0, no matter what the weights. But $\int_{-1}^{1} F(x)\,dx > 0$ because $F > 0$ in between the evaluation points.

Another nice feature of the integration rules based on orthogonal polynomials is that all weights w_k are positive (unlike in the Newton-Cotes rules, which may give negative weights to some points). To see why, consider the polynomial $G(x) = F(x)/(x-x_k)^2$ where F is as in the previous paragraph. Since $(x-x_k)^2$ cancels out, G is a polynomial of degree $2n-2$. Therefore, the integration rule is exact for it:

$$\int_{-1}^{1} G(x)\,dx = w_k G(x_k)$$

where the right hand side has just one term because $G(x_j) = 0$ for all indices $j \neq k$. Since G is a square, the left hand side is positive and $G(x_k)$ is also positive. Hence $w_k > 0$.

Since these integration rules are exact for the constant function $f(x) = 1$ (a polynomial of degree 0), we have $\sum_{k=1}^{n} w_k = \int_{-1}^{1} 1\,dx = 2$. As a consequence, all weights are between 0 and 2.

16.3 Gauss-Laguerre integration

The Gauss-Laguerre integration rule is stated for integrals of the form $\int_0^\infty f(x)e^{-x}\,dx$. This means that if we want to calculate, for example, $\int_0^\infty 1/(x^3+4)\,dx$, then the function to use is $f(x) = e^x/(x^3+4)$.

Apart from the presence of the weight function e^{-x}, the rest goes as in Section 16.2. Let $x_1, \dots, x_n$ be the roots of Laguerre polynomial L_n. Then solve a linear system for the weights $w_1, \dots, w_n$ so that the rule

$$\int_0^\infty f(x)e^{-x}\,dx \approx \sum_{k=1}^{n} w_k f(x_k) \tag{16.3.1}$$

is exact for $f(x) = x^p$ with $p = 0, \dots, n-1$. This choice and the orthogonality property make the rule exact when f is a polynomial of degree less than $2n$.

It is important to recognize that although there is an exponential term e^{-x} on the left hand side of (16.3.1), it does not appear on the right hand side. Think of it as being subsumed by the weights $w_1, \dots, w_n$. Aside: in terms of measure theory, we are approximating the continuous measure $e^{-x}\,dx$ by the discrete measure which puts mass w_k at the point x_k for each k.

One difference is the right hand side of the system which we use to solve for the weights w_k. For Gauss-Legendre integration we have $\int_{-1}^1 x^p\,dx = (1-(-1)^{p+1})/(p+1)$ there. For Gauss-Laguerre integration we have $\int_0^\infty x^p e^{-x}\,dx$ which takes a but more work. Integration by parts helps:

$$\int_0^\infty x^p e^{-x}\,dx - x^p e^{-x}\Big|_{x=0}^{x\to\infty} - \int_0^\infty px^{p-1}(-e^{-x})\,dx = p\int_0^\infty x^{p-1}e^{-x}\,dx$$

which shows, by induction, that $\int_0^\infty x^p e^{-x}\,dx = p!$.

For example, $L_2(x) = (x^2-4x+2)/2$ has roots $x_1 = 2-\sqrt{2}$ and $x_2 = 2+\sqrt{2}$. The system for weights consists of $w_1x_1^p + w_2x_2^p = p!$ for $p = 0, 1$. This simplifies to $w_1 + w_2 = 1$ and $w_1x_1 + w_2x_2 = 1$. The solution can be found by hand: $w_1 = (2+\sqrt{2})/4$ and $w_2 = (2-\sqrt{2})/4$.

Note that while for Gauss-Legendre integration the sum of weights is $\int_{-1}^1 1\,dx = 2$, for the Gauss-Laguerre integration the sum is $\int_0^\infty 1\,e^{-x}\,dx = 1$.

Let's apply two-point Gauss-Laguerre integration to $\int_0^\infty 1/(x^3+4)\,dx$.

```
x = [2-sqrt(2), 2+sqrt(2)];
w = [(2+sqrt(2))/4, (2-sqrt(2))/4]';
f = @(x) exp(x)./(x.^3+4);
disp(f(x)*w);
```

The result is 0.4666 while the true value of the integral is $2^{-1/3}3^{-3/2}\pi \approx 0.4799$. Not too bad a result after using just two evaluation points on an infinite interval. Even though this integral can be computed by hand, this is not an enjoyable task.

16.4 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 16.4.1 Compute the Gauss-Legendre points and weights. For a given integer $n \ge 2$, find the Gauss-Legendre evaluation points and weights.

Answer. Combine the code from Example 15.4.2 (computation of Legendre roots) and Example 14.2.1 (computing the weights). This only requires some changes in variable names and in the orientation of vectors (row/column).

```
n = input('n = ');
p = [1];
q = [1 0];
for m = 1:n-1
    r = ((2*m+1)*[q 0] - m*[0 0 p])/(m+1);
    p = q;
    q = r;
end
```

```
x = roots(r)';
disp(x);    %  the evaluation points
i = (1:n)';
A = x.^(i-1);
b = (1-(-1).^i)./i;
w = A\b;
disp(w');   %  the weights
```

□

Example 16.4.2 Apply the Gauss-Legendre integration rule. For a given integer $n \geq 2$, apply the Gauss-Legendre rule to the integrals $\int_{-1}^{1} e^x\,dx$ and $\int_{-1}^{1} (9x^2+1)^{-1}\,dx$. In each case, find the difference between the approximate and exact values.

Answer. Not repeating the code from Example 16.4.1, assume it already ran and computed `x, w`. With `exp(x)*w` we get an approximation to the integral of e^x. And `f(x)*w` does the same for the second function, if we define it as `f = @(x) (9*x.^2+1).^(-1)`.

```
approx = exp(x)*w;
exact = exp(1)-exp(-1);
disp(abs(approx-exact));

f = @(x) (9*x.^2+1).^(-1);
approx = f(x)*w;
exact = (2/3)*atan(3);
disp(abs(approx-exact));
```

The error is about $8.2 \cdot 10^{-10}$ for the first integral and about 0.058 for the second. Somehow, the rule is 100 million times more accurate for the first function than for the second, even though they are both perfectly smooth functions on the interval $[-1,1]$. The reason for such a different behavior is that e^x is easy to approximate by polynomials while $(9x^2+1)^{-1}$ is not so easy; it has to do with the Taylor series of these functions. This will come up again when we study approximation of functions. □

One could theoretically derive an estimate for the error of Gauss-Legendre integration rule in terms of the derivative of f of order $2n$. But this is impractical, because useful estimates of, for example, 10th derivative, are rarely available. The following example approaches the error estimation from another point of view.

Example 16.4.3 Estimating the accuracy of the Gauss-Legendre integration. Suppose that f is a function on $[-1,1]$ and there exists a polynomial p of degree 9 such that $|f(x)-p(x)| \leq 10^{-7}$ for all $x \in [-1,1]$. Estimate the error of the 5-point Gauss-Legendre rule applied to f.

Answer. The integral triangle inequality yields

$$\left| \int_{-1}^{1} f - \int_{-1}^{1} p \right| \leq \int_{-1}^{1} |f-p| \leq 2 \cdot 10^{-7}$$

Also by the triangle inequality,

$$\left| \sum_k w_k f(x_k) - \sum_k w_k p(x_k) \right| \leq \sum_k w_k |f(x_k) - p(x_k)| \leq 10^{-7} \sum_k w_k = 2 \cdot 10^{-7}$$

where the last step uses a remark at the end of Section 16.2. Finally, since the degree of p is $9 < 2 \cdot 5$, we have $\int_{-1}^{1} p = \sum_k w_k p(x_k)$. Combining the above, we conclude that

$$\left| \int_{-1}^{1} f - \sum_k w_k f(x_k) \right| \le 4 \cdot 10^{-7}$$

In general, the estimates above show that if f can be approximated within ε by some polynomial of degree $2n-1$, then the error of n-point Gauss-Legendre rule is at most 4ε. □

16.5 Homework

1. Write a script which, when given an integer $n \ge 2$, finds and displays the Gauss-Laguerre evaluation points and weights. This can be done by adapting the code in Example 16.4.1, similarly to exercises in Exercises 15.5. Note that the right-hand side of a system for weights will involve factorials; in Matlab they can be computed with `factorial` function, for example `factorial(i-1)`.

2. Apply the Gauss-Laguerre rule with $n = 5$ to the integral $\int_0^\infty \exp(-x^2)\,dx$ whose exact value is $\sqrt{\pi}/2$. The script should display the approximation obtained, and the error (the absolute value of the difference between the approximate and exact values).

Chapter 17

Adaptive integration

It is important for a numerical integration algorithm to be able to adapt to the behavior of the function being integrated, by increasing the number of evaluation points as needed to achieve desired precision. This adaptive approach can be combined with any of the numerical integration methods studied so far: one can have adaptive forms of Simpson's rule, Gauss-Legendre rule, etc.

17.1 Why we need adaptive methods

We have developed several methods of numerical integration: a family of Newton-Cotes methods, including Simpson's rule, and Gaussian integration methods. But the theory built so far has some weak points.

One weak point is the error estimation. When an integration rule has order d of accuracy, its error can be estimated by some multiple of the derivative of order d. For example, Simpson's rule is of order 4 and has error estimate involving the fourth derivative of f:

$$|\text{error}| \le \frac{1}{180} \max |f^{(4)}|(b-a)h^4 \tag{17.1.1}$$

where, as with all such estimates, the maximum of the absolute value of the derivative is taken over the interval of integration. But a high-order derivative may be difficult to find, and if found, difficult to estimate. And even if it is easy to find and estimate, the output of (17.1.1) may be too conservative or completely useless.

Example 17.1.1 Error of Simpson's method. Apply Simpson's method with $n = 2$ to the integral $\int_1^9 x^{3/2}\,dx$. Find the actual error of the method and compare it to the estimate (17.1.1).

Answer. Here $f(x) = x^{3/2}$ and $h = (9-1)/2 = 4$, so according to the formula (14.1.5)

$$\int_1^9 x^{3/2}\,dx \approx \frac{h}{3}(f(1) + 4f(5) + f(9)) \approx 96.9618$$

The exact value of this integral is $\frac{2}{5}(9^{5/2} - 1) = 484/5 = 96.8$. So, the error is 0.1618, relatively small.

To use the estimate (17.1.1) we need the fourth derivative of f, which is $(9/16)x^{-5/2}$. The absolute value of this function is maximal at $x = 1$, so $\max |f^{(4)}| = 9/16$. Also, $h = (b-a)/2 = 4$. So (17.1.1) says

$$|\text{error}| \le \frac{1}{180}\frac{9}{16}(9-1)4^4 = 6.4$$

Indeed, $0.1618 \leq 6.4$ but we see that the error estimate does not give the right idea of how large the error actually is. $\square$

It gets worse for the integral $\int_0^9 x^{3/2}\,dx$. The exact value is 97.2 and Simpson's rule gives 97.7756 so the error is 0.5756. But since the fourth derivative is $(9/16)x^{-5/2}$, it is unbounded on the interval $[0,9]$. So the right hand side of (17.1.1) is infinite. It is a true statement that $0.5756 \leq \infty$ but it is not a useful statement.

The above situation is not uncommon. When one derives an error bound for some numerical method, one has to consider "the worst-case scenario", with all possible errors accumulating in the worse possible way. In practice this rarely happens.

Moreover, formulas like (17.1.1) are difficult to implement in an algorithm. We know how to differentiate numerically (Chapter 12) but finding the absolute maximum of some function on an interval is not easy at all, as we will see later in the course.

The second weak point is that the function being integrated may have discontinuities or other special points, such as corners like $f(x) = |x|$. Numerical integration performs worse where the function is not smooth, and this requires using smaller step size around such features (but not elsewhere).

So we need an algorithm that estimates the error of numerical integration and adapts to any unusual features of the function.

17.2 Estimating error by using two step sizes

A practical way to estimate the error of some method of integration is to use it twice with two different step sizes, and compare the results. Typically, one compares the computations with steps h and $h/2$. Say, the first computation resulted in A_1 and the second, more accurate, in A_2. If the exact value is E, then we expect $|A_1 - E| \approx 2^d |A_2 - E|$ where d is the order of accuracy of the method. By the triangle inequality,

$$|A_1 - A_2| \geq |A_1 - E| - |A_2 - E| \approx (2^d - 1)|A_2 - E|$$

which gives us an idea of the error involved in A_2 (the more accurate approximation):

$$|A_2 - E| \lessapprox \frac{|A_1 - A_2|}{2^d - 1} \tag{17.2.1}$$

where the symbol $\lessapprox$ means the error is bounded by something like the right hand side. We do not assert this is an exact inequality, because the relation involving 2^d is only approximate.

Let us return to Example 17.1.1 where Simpson's method with $h = 4$ was used to get an approximation $A_1 = 96.9618$. Reducing the step size to $h = 2$ we get a second approximation:

$$A_2 = \frac{2}{3}(f(1) + 4f(3) + 2f(5) + 4f(7) + f(9)) \approx 96.8176$$

Since Simpson's rule has order $d = 4$, the error of second approximation can be estimated by (17.2.1) as follows:

$$|A_2 - E| \lessapprox \frac{|A_1 - A_2|}{15} = 0.0096$$

Recalling that the exact value is $E = 96.8$, we see that $|A_2 - E| = 0.0176$. So our error estimate, while not rigorous, is much closer to reality than the theoretical estimate (17.1.1).

17.3 Recursive subdivision algorithm

The previous section showed how the accuracy of integration can be automatically estimated. Next question is, what can be done when it is insufficient? Using more evaluation points is logical. But if the algorithm keeps the interval of integration $[a, b]$ the same and just increases the number of points on that interval (e.g., in Simpson's rule), this is not the best way to adapt. Typically, the accuracy is poor because of insufficient smoothness or large derivative at some parts of the interval, while other parts are fine.

A better way to adapt is **recursive subdivision**.

1. Given an interval $[a, b]$, compute the integral over it and estimate the error.
2. If the error is small enough, return the result of computation.
3. If the error is too large, divide the interval into two halves $[a, c]$ and $[c, b]$ where $c = (a + b)/2$, and restart the process from step 1, for each half separately.

This process involves **recursion**, because at step 3 the function evaluating the integral has to call itself. Schematically it looks like this.

```
function I = adaptive(f, a, b)
    I1 = % first computation
    I2 = % second computation, more accurate
    if abs(I1 - I2) % "small enough"
        I = I2;
    else
        c = (a+b)/2;
        I = adaptive(f, a, c) + adaptive(f, c, b);
    end
end
```

One has to carefully consider when the difference `abs(I1 - I2)` is "small enough". One approach is to ask for small relative error, like `abs(I1 - I2) < 1e-6 * abs(I2)`. However this risks infinite recursion for functions like $f(x) = \sqrt{x}$ near 0, which is both small and poorly behaved. No matter how small an interval $[0, \epsilon]$ we use, dividing it further will result in a change that is not very small in relative terms. The process terminates with Matlab reporting: "Out of memory. The likely cause is an infinite recursion within the program."

If one uses an absolute error bound like `abs(I1 - I2) < 1e-6`, there is another issue. We do not know how many subintervals will be used in the process, so even if each of them contributes an error of less than 10^{-6}, the total may be significantly higher.

A third approach is to enforce enforce error bound *relative to the size of interval*, for example `abs(I1 - I2) < 1e-6 * (b-a)`. This avoids the issue with functions that are problematic and small, such as $f(x) = \sqrt{x}$ near 0. But it runs into the same issue with functions that are problematic and large, such as $f(x) = 1/\sqrt{x}$ near 0. Here the integral is

A combination approach, such as `abs(I1 - I2) < 1e-6 * (b-a) + 1e-9`, allows the process to stop if the difference became small relative to interval size or just very small in absolute terms. This allows the subdivision to end even for functions like $f(x) = 1/\sqrt{x}$.

17.4 Combining two rules

What combination of two rules should we use for approximations I_1 and I_2? These rules should be of different accuracy; for example, it would be a bad idea to use the left endpoint and right endpoint rules. Both of these are about equally imprecise, so they may give the same answer which is still wrong by some large amount.

One can use, for example, the trapezoidal rule (order 2) and Simpson's rule (order 4). This has an advantage in that of of three evaluation points in Simpson's rule, two are also in trapezoidal rule. So one can compute function values once, for example

```
y = f([a, (a+b)/2, b]);
```

and obtain both trapezoidal and Simpson's rules from the vector `y`. This property of two rules being *nested* (evaluation points of one rule are contained in the other) is valuable for speeding up the computation.

Since the Gauss integration rule is the most accurate of those we discussed, one may decide to use it for both I_1 and I_2, just with different number of evaluation points, perhaps n and $2n$. A downside is that Gauss evaluation points are not nested. In practice, the Gauss rule is used for I_1 and the Kronrod rule (also based on orthogonal polynomials, but beyond our scope) is used for I_2. The Kronrod rule is designed to include all of the points used by the Gauss rule, and some additional ones. A popular choice, e.g., in TI graphing calculators, is (G7, K15), meaning 7-point Gauss rule combined with 15-point Kronrod rule. For an illustration of adaptive integration in Python module SciPy, see this blog post: it shows that Scipy uses (G10, K21) combination.

17.5 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 17.5.1 Adaptive Gauss integration. Implement adaptive integration as in Section 17.3 using the midpoint rule for I_1 and Gauss rule with $n = 2$ for I_2. Test it on the integral $\int_0^9 x^{-1/2}\,dx = 6$.

Answer. When implementing Gauss rules on a general interval $[a, b]$, it helps to introduce the midpoint $c = (a + b)/2$ and half-length $k = (b - a)/2$. This is because the rule was originally designed for $[-1, 1]$ and the linear function from $[-1, 1]$ to $[a, b]$ is $kx + c$. The coefficient k will appear in the formulas due to the change of variable formula.

```
function adaptive_example()
    f = @ (x) x.^(-1/2);
    fprintf('The integral is %.6f\n', adaptive(f, 0, 9));
end

function I = adaptive(f, a, b)
    c = (a+b)/2;
    k = (b-a)/2;
    I1 = k*2*f(c);    % midpoint, same as Gauss n=1
    I2 = k*(f(c - k/sqrt(3)) + f(c + k/sqrt(3)));  % Gauss n=2
    if abs(I1 - I2) < 1e-6 * (b-a) + 1e-9
        I = I2;
    else
        I = adaptive(f, a, c) + adaptive(f, c, b);
    end
end
```

Note that the function is not defined at 0 which is an endpoints of the interval of integration. This would be a problem if we used, for example, trapezoidal or Simpson's rules. Not a problem for midpoint or Gauss rules. By the way, the midpoint rule can be viewed as the Gauss rule with 1 point, because the degree 1 Legendre polynomial is $P_1(x) = x$, with the zero at the middle of interval $[-1, 1]$ □

Question 17.5.2 Gauss-Kronrod error estimate. The current version of Wikipedia article Gauss–Kronrod quadrature formula says: "The integral is then estimated by the Kronrod rule K15 and the error can be estimated as |G7-K15|". This estimate, while reasonable, is quite conservative: the actual error will be much less. Can you explain why? □

17.6 Homework

1. Estimate the error of computing $\int_{0.1}^{1.1} \sqrt{x}\,dx$ using Simpson's rule with $h = 1/2$, according to the error bound (17.1.1).
2. Calculate the actual error in Exercise 17.6.1, that is, the difference between what Simpson's rule gives and the exact value of the integral. Is the actual error similar in size to the error estimate found in Exercise 17.6.1?
3. Rewrite the adaptive integration script in Example 17.5.1 so that it uses trapezoidal and Simpson's rules for I_1, I_2, and test it on the integral $\int_0^4 x^{3/2}\,dx$ to make sure it returns a correct result.

Chapter 18

Multivariable integration

Numerical computation of multiple integrals presents new aspects: the shape of boundary, the exponential growth of evaluation points with dimension, and somewhat unexpected benefits of randomness in high-dimensional spaces.

18.1 Double integrals over rectangles

A rectangle $R = [a,b] \times [c,d]$ is the easiest two-dimensional domain to integrate over: in Calculus we would compute

$$\iint_R f(x,y)\,dA = \int_a^b \int_c^d f(x,y)\,dy\,dx$$

Suppose we pick two integration rules: one for the interval $[a,b]$, say $\int_a^b g(x)\,dx \approx \sum_i w_i g(x_i)$, the other for interval $[c,d]$, say $\int_c^d g(y)\,dy \approx \sum_j v_j g(y_j)$. The function g here is a placeholder; the rule consists of the choice of evaluation points and their weights. Note that the weights and points depend on the interval, so they will probably be different even if we use the same integration method for both variables. With this setup, the two-dimensional integration rule for this rectangle can be expressed as a double sum:

$$\iint_R f(x,y)\,dA = \sum_i \sum_j w_i v_j f(x_i, y_j) \tag{18.1.1}$$

Example 18.1.1 Set up Simpson's rule over a rectangle. Write out explicitly the sum (18.1.1) for the integral of function f over the rectangle $R = [-2,2] \times [-1,1]$ using the simple Simpson's rule in both variables.

Answer. In the y direction, the evaluation points are $-1, 0, 1$ with the weights $1/3, 4/3, 1/3$. In the x direction, the evaluation points are $-2, 0, 2$ with the weights $2/3, 8/3, 2/3$. So the sum has 9 terms:

$$\iint_R f(x,y)\,dA \approx \frac{2}{9}(f(-2,-1)+4f(0,-1)+f(2,-1)+4f(-2,0)+16f(0,0)+4f(2,0)+f(-2,1)+4f(0,1)+f(2,1$$

□

Given the large number of terms in double sum (18.1.1), we should try to organize such computations efficiently. The first thing to do is to evaluate the function on a rectangular grid formed by all (x_i, y_j) points. A useful Matlab command is `meshgrid`, which is used as

```
[X, Y] = meshgrid(x, y)
```

where `x,y` are vectors with evaluation points. The output consists of two matrices `X, Y` which represent the rectangular grid: `X` has the x-coordinates of all the grid points, and `Y` has their y-coordinates. The reason this is useful is that `f(X, Y)` can now be used to evaluate the function on the entire grid at once (producing a matrix of its values), provided f is written so that it can handle matrix input.

Having computed `V = f(X, Y)`, we need to combine it with weights and to get a single number at the end. Recall in one dimension we would do this as `f(x)*w` where `w` is a column vector of x-weights. We should still do this, but to also multiply and sum over the y-weights, we need `v*f(X, Y)*w` where `v` is a row of vector of y-weights.

Example 18.1.2 Implement Simpson's rule over a rectangle. Write a Matlab script to integrate the function $f(x) = \sqrt{4 + xy + y^3}$ over the rectangle in Example 18.1.1, using the integration points and weights from that example.

Answer.

```
x = [-2 0 2];
y = [-1 0 1];
w = (2/3)*[1 4 1];
v = (1/3)*[1 4 1];

f = @(x, y) sqrt(x.*y + y.^3 + 4);
[X, Y] = meshgrid(x, y);
disp(v*f(X, Y)*w')
```

The first four lines do not involve the function; they are setting the stage. The function is evaluated on the rectangular grid and its values are conveniently aggregated using matrix-vector multiplication. The result is 15.8859. WolframAlpha gives 15.9219. □

The rectangular grid computation can also be used to quickly plot the function: `surf(X, Y, f(X, Y))` does this.

18.2 Double integrals over general regions

One way to evaluate $\iint_D f(x,y)\,dA$ for a general region D is to define f to be 0 outside of D, and integrate this redefined function over some rectangle containing D. This approach is problematic because the redefined function will probably be discontinuous on the boundary of D, and discontinuities cause large errors in numerical integration rules.

Instead, one can literally adopt the iterated integration approach from calculus courses: do integration in one variable at a time. Suppose the integration is done in y variable first ("vertical slicing"). Then the limits of the inner integral will be functions of x. These functions will need to be implemented in Matlab, and will be called in a loop.

Example 18.2.1 Implement Simpson's rule over a non-rectangular region. Rewrite Example 18.1.2 to integrate the same function over the interior of the ellipse inscribed in the rectangle $R = [-2, 2] \times [-1, 1]$. (The ellipse with the equation $x^2/4 + y^2 = 1$.) Use composite Simpson's rule with $n = 4$ subintervals in both variables.

Answer. We need functions for top and bottom boundaries, according to the integral structure

$$\int_a^b \int_{\text{bottom}}^{\text{top}} f(x,y)\,dy\,dx$$

In the x-variable, the points and weights are set once. But in the y-direction they are calculated within a loop, because different x-values produce different limits of integration for y. The vector `innerint` is used below to store the values of the inner integral.

```
f = @(x, y) sqrt(x.*y + y.^3 + 4);
top = @(x) sqrt(1-x.^2/4);
bottom = @(x) -sqrt(1-x.^2/4);
a = -2;
b = 2;

n = 4;
x = linspace(a, b, n+1);
w = [1 4 2 4 1]*(b-a)/(3*n);
innerint = zeros(size(x));

for k = 1:numel(x)
    c = bottom(x(k));
    d = top(x(k));
    y = linspace(c, d, n+1)';
    v = [1 4 2 4 1]*(d-c)/(3*n);
    innerint(k) = v*f(x(k), y);
end

disp(innerint*w');
```

The result is 11.8764, compared to the WolframAlpha output 12.5419. This is less accurate than what we got for rectangle. Note that all points with $x = \pm 2$ are wasted because there is no interval to integrate over with respect to y. For this reason, open-ended rules such as Gaussian integration is preferable. Replacing the choice of `x, w` above with the 3-point Gaussian rule

```
x = [-sqrt(3/5) 0 sqrt(3/5)]*(b-a)/2 + (a+b)/2;
w = [5/9 8/9 5/9]*(b-a)/2;
```

we get 12.7081, which is much more accurate. □

18.3 High-dimensional integration: Monte-Carlo method

Integration over a d-dimensional region with large d presents new challenges. One is setting up all the boundaries for multiple variables. Another is the number of evaluation points. If we use n evaluation points for each of d variables, the total number of evaluations is n^d which grows exponentially with dimension. It becomes crucial to not have large n, which is why Gaussian integration is often used.

There is an alternative: the Monte-Carlo integration method consists of picking evaluation points at random, from the uniform distribution over the region D. Averaging the computed values of f, we obtain an approximation to the average of f over D. To find the integral, multiply the average by the volume of D.

In terms of probability theory, the random choice of x makes $f(x)$ a **random variable**. This random variable has **mean value** μ equal to the average of f over D. Its **variance** σ^2 is the average of $(f-\mu)^2$ over D. The Central Limit Theorem tells us that the mean of N independent samples is approximately normally distributed with mean μ and variance σ^2/N. In particular, with probability about 99.7% the sample mean is within three standard deviations of the mean, which means the error of Monte-Carlo method is very likely to be bounded by $3\sigma/\sqrt{N}$. On one hand, this tends to zero quite slowly. On another hand, the rate at which it tends to zero is independent of dimension d.

Example 18.3.1 Implement the Monte-Carlo method over a high-dimensional cube. Integrate the function $f(\mathbf{x}) = \sqrt{1+|\mathbf{x}|^2}$ over the 10-dimensional unit cube $[0,1]^{10}$ using the Monte-Carlo method.

Answer.

```
f = @(x) sqrt(1 + x*x');
d = 10;
N = 1e6;

s = 0;
for k = 1:N
    s = s + f(rand(1, d));
end
disp(s/N)
```

The output is, of course, random, but usually it is about 2.069. If your computer handles a million samples (`N = 1e6`) easily, you may want to try `1e7`. It would be tedious to compute this integral with WolframAlpha; just entering the limits would take long.

The above code is not vectorized: it uses a loop to evaluate f at consecutive points. One could generate all these points at once with `rand(N, d)` and try to get all values of f at once. But this requires coding `f` in such a way that it can consume a matrix and produce a vector of values, and this may be difficult.
□

Remark 18.3.2 Averaging over a sphere. Monte-Carlo method can also be used to average a function over a *sphere* with the help of **normal distribution**. Here is a brief summary of random number generators in Matlab:

- `rand(m, n)` generates an $m \times n$ matrix of random real numbers taken from the uniform distribution on the interval $[0,1]$.
- `randi([A B], m, n)` generates an $m\times n$ matrix of random *integer* numbers taken from the discrete uniform distribution on the finite set $A, A+1, \dots, B$.
- `randn(m, n)` generates an $m \times n$ matrix of random real numbers taken from the standard normal distribution on the real line.

Try them out with `rand(1, 5)`, `randi([2 7], 1, 5)`, and `randn(1, 5)` to observe the differences in output.

One of the distinguishing features of the normal distribution is the following: if one generates a random d-dimensional vector $\mathbf{x}$ by taking each component $x_1, \dots, x_d$ independently from the normal distribution, the vector $\mathbf{x}$ is equally likely to point in any given direction. More precisely, the unit vectors $\mathbf{x}/|\mathbf{x}|$ are uniformly distributed over the unit sphere. With Matlab, this process amounts to the computations

```
x = randn(1, d);
x = x/norm(x);
```

Note that `x = randn(1, d)/norm(randn(1, d))` would not do the job, because each use of `randn` leads to a new random vector.

Using this random unit vector in a loop, like in Example 18.3.1, we can find an approximate value of the average of a function f over the unit sphere $\{\mathbf{x} \in \mathbb{R}^d : |\mathbf{x}| = 1\}$. This method works specifically for the normal distribution because of the following property of its probability density function p: the product $p(x_1)p(x_2)\cdots p(x_d)$ depends only on the norm of vector $\mathbf{x}$.

18.4 Homework

1. Rewrite Example 18.2.1 so that the number of subintervals n can be entered by the user. The input should be rejected with `error('n must be even')` if an odd number is entered. The command `mod(n, 2)` will be useful for checking this. The code producing the weights `w, v` will need to be generalized so that it works for general n.

Try choosing n large enough so that the displayed integral value is 12.5419 (or at least within 0.0001 of this value). In a comment, state the value of n which achieved the above. How many evaluation points were used with this n?

2. Use Remark 18.3.2 to find the averages of the following functions over the unit sphere in $\mathbb{R}^{10}$:

(a) $f(\mathbf{x}) = x_1 + \cdots + x_{10}$, the sum of coordinates.

(b) $f(\mathbf{x}) = x_1^2$, the first coordinate squared.

(c) $f(\mathbf{x}) = (x_1 + x_2 + \cdots + x_{10})^2$, the square of the sum of all coordinates.

The Matlab function `sum` will be useful.

The results should give a strong indication of what the exact values of these integrals are. Can you explain why these integrals have these values?

Chapter 19

Differential equations: Euler's method and its relatives

Ordinary differential equations (the subject of MAT 414 and MAT 485) involve at least one unknown function of one variable, as well as its derivatives. Differential equations model a process evolving in time. While there are symbolic solutions for some kinds of linear differential equations, the nonlinear equations usually require a numerical solution.

19.1 Ordinary differential equations

Suppose that y is an unknown function of variable t, with the property that $y'(t) = 6t$ for all t. This qualifies as a differential equation, although a very simple one: since the right hand side is explicit, integration with respect to t tells us $y(t) = 3t^2 + C$ where C is some undetermined constant. If we also know the value of y at some point (an **initial condition**), then C can be determined. So, indefinite integration (finding antiderivatives) is a special case of solving differential equations.

Suppose that y satisfies $y'(t) = 6y(t)$ for all t. This is a more typical example of a differential equation. We cannot find y by integrating the right hand side because it is unknown. The solutions of this equation are $y(t) = Ce^{6t}$ where C is again an undetermined constant.

Usually the argument of unknown function is omitted, so the above equation would be written as $y' = 6y$. Higher order derivatives may appear, for example $y'' + 6y' - t^2 y = \cos t$. Despite the appearance of terms like t^2, this differential equation is considered to be **linear**, because the unknown function and its derivative enter it linearly. The simplest equation of a **nonlinear** differential equation is $y' = y^2$.

A **system** of differential equations involves n equations with n unknown functions, for example

$$\begin{aligned} y_1' &= 3y_1 + y_2 + t \\ y_2' &= t\sin(y_1)\cos(y_2) \end{aligned}$$

This is usually considered as a single equation with an unknown vector-valued function $\mathbf{y}$:

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}) \quad \text{where } \mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} 3y_1 + y_2 + t \\ t\sin(y_1)\cos(y_2) \end{pmatrix}$$

An equation (or system of equations) involving higher derivatives can be rewritten as a system that involves only the first derivatives. The trick is to consider y' as another unknown function, say z, and add the relation $y' = z$ to the system. For example, $y'' + 6y' - t^2 y = \cos t$ becomes the system

$$\begin{aligned} y' &= z \\ z' &= -6z + t^2 y + \cos t \end{aligned}$$

Or in vector form, letting $\mathbf{y} = \begin{pmatrix} y \\ z \end{pmatrix}$:

$$\mathbf{y}' = \begin{pmatrix} y_2 \\ -6y_2 + t^2 y_1 + \cos t \end{pmatrix} \tag{19.1.1}$$

We will not need theoretical methods of solving differential equations, but you should be able to rewrite an equation/system of higher order as a first-order system such as (19.1.1).

A terminological remark: all of the above are called *ordinary* differential equations (ODE) because there is only one independent variable, t. With more than one independent variable such equations involve partial derivatives and are called *partial* differential equations (PDE). We will not consider PDE in this course.

19.2 Euler's method

To solve an ODE numerically, we "discretize" it: instead of variable t varying continuously over some interval, we choose step size h and work with t_0, $t_1 = t_0 + h$, $t_2 = t_1 + h$, and so on. (An adaptive method will change the step size h based on the outcome of computation; more on this later.) We try to find approximate values of y at the points t_k, starting with $y(t_0) = y_0$ which is known (initial condition). The idea of Euler's method is geometric: draw the line segment starting at (t_0, y_0) with the slope $f(t_0, y_0)$. When it reaches the abscissa t_1, we have new point (t_1, y_1). Then process is repeated. In a formula, it is expressed as

$$y_{k+1} = y_k + hf(t_k, y_k) \tag{19.2.1}$$

Example 19.2.1 Implement Euler's method. Implement Euler's method to solve the differential equation $y' = -ty$ with initial condition $y(0) = 1$. Plot the result on the interval $[0, 3]$ and compare it with the exact solution, which is $y(t) = \exp(-t^2/2)$. Try different step sizes: 0.3, 0.1, 0.01.

Repeat the above for $y' = ty$, where the exact solution is $y(t) = \exp(t^2)$.

Answer.

```
f = @(t, y) -t*y;   % or without -
h = 0.3;   %  or 0.1, 0.01
t = 0:h:3;
y0 = 1;
y = y0*ones(size(t));

for k = 1:numel(t)-1
    y(k+1) = y(k) + h*f(t(k), y(k));
end

exact = exp(-t.^2 /2);   % or without -
plot(t, y, t, exact)
```

```
legend('Euler', 'exact')
```

□

As Example 19.2.1 shows, a weakness of Euler's method is its tendency to fall behind the solution when the solution is increasing and concave up, or decreasing and concave down. The reason is that it uses the rate of change from point t_k for the entire interval $[t_k, t_{k+1}]$. When applied to an ODE of the form $y' = f(t)$, Euler's method becomes the left endpoint method of integration. So it is not very accurate.

19.3 Estimating the accuracy of numeric ODE methods

Evaluating the accuracy of ODE solution methods is a more complex issue because they involve approximate computations based on the results of previous other approximate computations. A convenient model problem to use is $y' = y$ with $y(0) = 1$. Its exact solution at $x = h$ is

$$e^h = \sum_{n=0}^{\infty} \frac{h^n}{n!} = 1 + h + h^2/2 + h^3/6 + \cdots$$

The approximate solution y_1 at $x = h$ will be some other expression. The difference $|e^h - y_1|$ gives us the **local error** of the method (for one step). The cumulative error can be estimated by multiplying the local error by the necessary number of steps to cover an interval $[a, b]$, which is $(b - a)/h$.

Euler's method has $y_1 = 1 + h$, hence $|e^h - y_1| = h^2/2 + \ldots$, the local error is of second order. The cumulative error is of order 1. This is consistent with what we know about left endpoint method of integration.

The above does not capture the entire picture. Although errors might accumulate, they do not always do that. When solving an equation like $y' = -ry$ (exponential decay) we get $y_k = (1 - rh)^k y_0$. As long as $h < 2/r$, we have $|1 - rh| < 1$ and so the approximate solution decays exponentially as it should. But if $h > 2/r$, the solution grows in magnitude instead of decreasing. This leads to the subject of *stability* of numerical ODE methods, which is beyond our scope.

19.4 Improving Euler's method

By analogy with Trapezoidal rule for numerical integration, one can try to improve the accuracy by using the average of both endpoints of an interval. That is, we would like to write

$$y_{k+1} = y_k + h\frac{f(t_k, y_k) + f(t_{k+1}, y_{k+1})}{2}$$

However, this means we need to know y_{k+1} in order to find y_{k+1}. So we first make a prediction for y_{k+1} and then use it to compute y_{k+1} again, more accurately. Namely, we use the original Euler's method to compute "predictor"

$$\tilde{y}_{k+1} = y_k + hf(t_k, y_k) \tag{19.4.1}$$

and then compute the "corrector"

$$y_{k+1} = y_k + h\frac{f(t_k, y_k) + f(t_{k+1}, \tilde{y}_{k+1})}{2} \tag{19.4.2}$$

This is one of many **predictor-corrector** methods for solving ODE. This particular method goes by various names (modified Euler method, improved Euler method); I prefer to call it the trapezoidal method, because this name describes the logic of (19.4.2).

Example 19.4.1 Implement trapezoidal method for ODE. Modify Example 19.2.1 to use the trapezoidal method.

Answer.

```
f = @(t, y) -t*y;   % or without -
h = 0.3;   %  or 0.1, 0.01
t = 0:h:3;
y0 = 1;
y = y0*ones(size(t));

for k = 1:numel(t)-1
    pred = y(k) + h*f(t(k), y(k));
    y(k+1) = y(k) + h/2*(f(t(k), y(k)) + f(t(k+1), pred));
end

exact = exp(-t.^2 /2);   % or without -
plot(t, y, t, exact)
legend('trapezoid', 'exact')
```

□

Theoretical analysis of accuracy confirms the above observation that trapezoidal method is much more accurate. For the model equation $y' = y$ it gives $y_1 = 1 + h(1 + 1 + h)/2 = 1 + h + h^2/2$, which matches exact solution $e^h = 1 + h + h^2/2 + h^3/3$ up to h^3 term. Therefore, the cumulative error is of order h^2.

A close relative is **midpoint method**. In it the trapezoidal corrector (19.4.2) is replaced by

$$y_{k+1} = y_k + hf((t_k + t_{k+1})/2, (y_k + \tilde{y}_{k+1})/2) \tag{19.4.3}$$

where $\tilde{y}_{k+1}$ is computed as in (19.4.1).

It is not so easy to improve accuracy further. There is a family of higher-order ODE solution methods (Runge-Kutta methods) of which the 4th order method is most popular; it involves several predictor-corrector steps. As with adaptive integration, one can compare the results of two methods of different orders in order to decide if the step size h should be reduced. This is what Matlab's ODE solvers do; for example, `ode45` is a solver that combines a 4th order method with a 5th order method. We will use it later.

19.5 Solving systems of differential equations

Conceptually, any of the above methods can be directly applied to systems of m differential equations, just by interpreting y as a vector function $\mathbf{y}$. The notation becomes awkward: now y_2 refer to the second component of $\mathbf{y}$, while the value $\mathbf{y}(t_2)$ might be denoted $\mathbf{y}_2$. There are also a few changes in Matlab notation regarding `y`: it will now be a matrix, where rows correspond to components and columns correspond to points of evaluation. The right-hand side function `f` should return a column vector of the same dimesion m.

Example 19.5.1 Implement trapezoidal method for ODE system.

Solve the system

$$\begin{aligned} y_1' &= t - 3y_2 \\ y_2' &= 2y_1 \end{aligned}$$

with initial condition $y(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, on the interval $[0, 10]$ using the trapezoidal method with the step $h = 0.01$. Plot the solution.

Answer.

```
f = @(t, y) [t-3*y(2); 2*y(1)];
h = 0.01;
t = 0:h:10;
y0 = [1; 0];
y = y0*ones(size(t));

for k = 1:numel(t)-1
    pred = y(:, k) + h*f(t(k), y(:, k));
    y(:, k+1) = y(:, k) + h/2*(f(t(k), y(:, k)) + f(t(k+1), pred));
end

plot(t, y)
```

The most visible difference is the notation `y(:, k)` which takes the entire kth column of the matrix `y`. Its mathematical meaning is the approximation to $\mathbf{y}(t_k)$. The initial condition `y0` is a column vector, which makes the computation `y = y0*ones(size(t));` an outer product: it creates a matrix in which every column is equal to `y0`. This is convenient because we need the first column to be equal, and the other columns will be rewritten anyway. □

The plot displayed by Example 19.5.1 is a **time series plot** in which one axis represents the independent variable (usually time) and the other axis shows the components of $\mathbf{y}$. The other option we have is to make a **phase plot** in which each component of $\mathbf{y}$ gets its own axis, and there is no time axis. This is achieved with

```
plot(y(1,:), y(2,:))
```

The phase plot may be preferable when the system does not involve time directly (an **autonomous system**). Try both kinds of plots for the autonomous system

```
f = @(t, y) [y(2)-y(2)^3; 2*y(1)];
```

19.6 Homework

1. (Theoretical) Show that the midpoint ODE method, described by (19.4.3), has the same order of accuracy as the trapezoidal ODE method: its cumulative error is proportional to h^2.
2. (Theoretical) Rewrite the second-order equation $y'' + yy' - ty^3 = e^{-t}$ as a system of two first-order equations.
3. Use the midpoint method with step size $h = 0.01$ to solve the autonomous

system

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -y_1 - y_2^3 \end{aligned}$$

with initial condition $y(0) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ on the interval $[0, 50]$. Display both time series plot and phase plot side by side, for example

```
subplot(1, 2, 1)
plot(t, y)
subplot(1, 2, 2)
plot(y(1,:), y(2,:))
```

How would you describe the long-term behaviour of this solution?

Chapter 20

Modeling with differential equations

Systems of differential equations are among the main tools of mathematical modeling, which can be informally described as the process of matching real-world phenomena with mathematical structures. Useful models allow us to make predictions and recommendations for future actions based on mathematical methods, including numerical methods.

20.1 Models of an epidemic

A basic model of epidemic divides the population into three groups: susceptible S (able to be infected), infected I, and removed R (not able to be infected). Often group R is called "recovered" but it also contains deceased and vaccinated. One can modify the model in various ways: include a separate group for deceased, allow for a possibility of reinfection, etc. The sum $S + I + R$ remains constant in time, because other demographic factors such as births do not substantially change the picture of an epidemic.

It is convenient to express S, I, R as a proportion of total population. This means these quantities are between 0 and 1 and their sum is 1. An important assumption of the model is that the rate of infection is proportional to the product SI. This is based on the simplifying idea that any two people in the population are equally likely to meet and transmit the disease, and since SI is the number of all susceptible-infected pairings, the number of transmissions will be proportional to it. This is not too far from reality when "population" means, for example, student population on some campus (pre-COVID). When modeling the population of a country, one needs to consider the effects of geography.

Thus, the rate of new infections is βSI where β is the coefficient of proportionality that depends on how contagious the disease is and what preventive measures are taken by the population. At the same time, the rate of new recoveries can be reasonably assumed to be γI, proportional to the number of currently infected. So we have the system

$$S' = -\beta SI \tag{20.1.1}$$

$$I' = \beta SI - \gamma I \tag{20.1.2}$$

$$R' = \gamma I \tag{20.1.3}$$

with some initial conditions (S_0, I_0, R_0) where typically $R_0 = 0$ and I_0 is small. The constants β, γ are the **parameters** of this system. Their values have a major impact on the long-term behavior of solutions; yet these values are very difficult to estimate in practice (especially β). One might try to estimate them by studying a small-scale outbreak closely and fitting a model to the observed counts of infected and recovered people.

We could solve the system (20.1.1)-(20.1.2)-(20.1.3) numerically using the methods of Chapter 19 but to keep the focus on the modeling aspect, we use Matlab's solver `ode45`. Its basic syntax is

```
[t, y] = ode45(rhs, [a, b], y0);
```

where `rhs` is the right hand side of the ODE system, `[a, b]` is the interval on which solution must be found, and `y0` is the initial condition at the left endpoint of that interval (`ode45` allows it to be either a column vector or a row vector.) Note that `rhs` must take two variables in this order: `rhs = @(t, y) ...` even if the system does not involve time (is autonomous). The solver returns a vectors of t-values and a vector or matrix of y-values. It is a matrix if we solve a system; in this case `y(:,1)` approximates the first unknown function, `y(:,2)` the second, and so on. The time series of the solution is obtained with `plot(t, y)`.

Example 20.1.1 Implement SIR model. Use `ode45` to solve the equations (20.1.1)-(20.1.2)-(20.1.3) on the interval $[0, 100]$ with $\beta = 0.3$, $\gamma = 0.2$, and $\mathbf{y}_0 = (0.99, 0.01, 0)$.

Answer.

```
beta = 0.3;
gamma = 0.2;
y0 = [0.99; 0.01; 0];
rhs = @(t, y) [-beta*y(1)*y(2); beta*y(1)*y(2) - gamma*y(2); gamma*y(2)];
[t, y] = ode45(rhs, [0, 100], y0);
plot(t, y, 'LineWidth', 3)
legend('Susceptible', 'Infected', 'Recovered')
```

In this scenario a part of the population (less then half) escapes the infection because the epidemic ends on its own after the transmission rate drops. Try varying the coefficients β, γ and observe the effect. □

In reality, β is not necessarily constant in time. Let us introduce seasonality into the model by making β a periodic function of t, for example

```
beta = @(t) 0.3 + 0.2*cos(0.2*t);
```

which is a periodic function that varies between 0.1 and 0.5. Try this out (note that `beta` should be replaced by `beta(t)` in the formula for `rhs`).

20.2 Predator-prey models

A traditional example of predator-prey relation is foxes and rabbits (in SU neighborhood, it could be hawks and squirrels instead). The Lotka-Volterra model of this relationship involves four positive parameters a, b, c, d:

$$R' = aR - bRF \tag{20.2.1}$$
$$F' = -cF + dRF \tag{20.2.2}$$

The first equation indicates that without foxes, rabbit population would grow at a constant relative rate. As in SIR model, the number of interactions is proportional to the product RF. Each interaction has a chance of reducing the number of rabbits. These interactions also contribute to the growth of foxes but not necessarily at the same rate (an eaten rabbit does not mean there is now an extra fox; we are not modeling a zombie apocalipse). Finally, the minus sign in $-cF$ ensures that without rabbits, foxes die out.

Example 20.2.1 Implement Lotka-Volterra model. Use `ode45` to solve the equations (20.2.1)-(20.2.2) on the interval $[0, 100]$ with

```
a = 0.1;
b = 0.004;
c = 0.2;
d = 0.001;
y0 = [100; 30];
```

Answer. To the code lines written above, we only have to add

```
rhs = @(t, y) [a*y(1) - b*y(1)*y(2); -c*y(2) + d*y(1)*y(2)];
[t, y] = ode45(rhs, [0, 100], y0);
plot(t, y, 'LineWidth', 3)
legend('Rabbits', 'Foxes')
```

Observing the periodicity of the solution, one may want to summarize it with a phase plot in addition to time-series plot:

```
figure()
plot(y(:,1), y(:,2), 'LineWidth', 3)
xlabel('Rabbits')
ylabel('Foxes')
```

□

The Lotka-Volterra model has two **equilibrium points**: $(0, 0)$ and $(c/d, a/b)$, meaning that a solution with either of these initial conditions stays constant. The former is mutual extinction, the latter is stable coexistence. But since the solution is periodic, it does not converge to either equilibrium.

20.3 Examples and questions

These are additional examples for reviewing the topic we have covered. When reading each example, try to find your own solution before clicking "Answer". There are also questions for reviewing the concepts of this section.

Example 20.3.1 A natural limit of the growth of prey. The original Lotka-Volterra model assumes that on their own, rabbit population would grow exponentially: $R' = aR$ leads to $R(t) = R_0 e^{at}$. But in reality, population is constrained by available resources (food and habitat). Modify the equation (20.2.1) by replacing aR with $aR(1-R/K)$ where K is the **carrying capacity** of the population. Adjust the parameters to

```
d = 0.003;
K = 200;
```

and solve the system on the interval $[0, 500]$. □

Question 20.3.2 Vaccination. How could one model the introduction of a vaccine during an epidemic? □

20.4 Homework

1. (Theoretical) Expand the Lotka-Volterra model (20.2.1)-(20.2.2) to include a second kind of predator, wolves W, which eats both rabbits and foxes. Note that the rate at which wolves eat rabbits would be different from the rate at which they eat foxes.
2. (Theoretical) The SIR model assumes that any two individuals are equally likely to "mix", i.e., to meet and potentially transmit the disease. Modify the model to account for different groups within a population (for example, on-campus and off-campus students). This means having two groups of susceptible individuals: S_1 and S_2, and two groups of infected ones: I_1 and I_2. The recovered/removed ones can be all in one group R since their interactions do not matter. Since there should be more interaction within each group than between the two groups, the products $S_1 I_1$ and $S_2 I_2$ should have larger coefficients in the system than $S_1 I_2$ and $S_2 I_1$.
3. Choose one of the two models you constructed in the previous exercises, and implement it in Matlab. Try to choose the coefficients and initial conditions so that the change of size of each group is visible on the plot (meaning the plot does not just show a horizontal line).

Part IV

Modeling Data

Chapter 21

Polynomial Interpolation

The main goal of this part of the course is to find some reasonable function that matches the given data points, perhaps approximately. Such a function may be used for making predictions about the future, or to fill the gaps between the data points (for example, estimate the population in a year when there was no census). It can also be used for numerical calculus: integration or differentiation.

Polynomials are the simplest nonlinear functions, so when we look for a simple function that could fit our data, it is natural to reach for polynomials. It turns out polynomials are not as simple as they appear at first.

21.1 Interpolation in the monomial basis

Given n data points (x_k, y_k) with $k = 1, \dots, n$, where all x_k are different, we are looking for a polynomial p of degree $< n$ such that $p(x_k) = y_k$ for $k = 1, \dots, n$. A straightforward way is to do this is to write $p(x) = \sum_{j=0}^{n-1} c_j x^j$ and solve a linear system for c_j; it has n equations with n unknowns. But this process is both slow and numerically fragile; the matrix of the linear system is likely to be ill-conditioned. And even if we could find the coefficients of monomials x_j easily, we do not necessarily want to, for the following reason.

We are used to seeing polynomials written in terms of the **monomial basis** $x^0, x^1, \dots, x^d$, that is, as a linear combination of the monomials. But this basis is ill-suited for numerical computations away from the point $x = 0$. For example, consider the simple polynomial $p(x) = (x - 7)^{15}$. If we expand it in the monomial basis and then try to compute its values on the interval $[6, 8]$, the result will be disappointing. In the code below, `c` is a vector of coefficients; the ellipsis notation allows it to by split between lines for readability.

```
c = [1, -105, 5145, -156065, 3277365, -50471421, 588833245, -5299499205, ...
    37096494435, -201969803035, 848273172747, -2699051004195, 6297785676455, ...
    -10173346092735, 10173346092735, -4747561509943];
d = 15;
x = linspace(6, 8, 1000);
p = zeros(size(x));
for k=1:d+1
    p = p + c(k)*x.^(d+1-k);
end
subplot(1, 2, 1)
plot(x, p)
```

```
subplot(1, 2, 2)
plot(x, (x-7).^15)
```

There is a catastrophic loss of significance here. If one uses `p = polyval(c, x)`, the result is not much better, even though this command tries to avoid the loss of significance. Expanding this polynomial in the monomial basis is numerically unwise.

21.2 Lagrange interpolating polynomial

Thinking of the set of all polynomials of degree less than n is a vector space of dimension n, we should realize this space may have a better basis for our task than $x^0, x^1, \dots, x^{n-1}$. Given interpolation data (x_k, y_k) with $k = 1, \dots, n$, we can construct one such basis as follows. The **Lagrange basis polynomials** are

$$\ell_k(x) = \prod_{j\neq k} \frac{x - x_j}{x_k - x_j} \tag{21.2.1}$$

They are constructed so that $\ell_k(x_j) = 0$ when $j \neq k$ and $\ell_k(x_k) = 1$. Consequently, in this basis the interpolation problem is solved with the Lagrange interpolating polynomial

$$p(x) = \sum_{k=1}^{n} y_k \ell_k(x) \tag{21.2.2}$$

which obviously satisfies $p(x_k) = y_k$. The coefficients of interpolating polynomial in this basis are just the given y-values.

For the sake of efficiency and numerical stability, the evaluation of p can be arranged as follows. Since the denominators in (21.2.1) do not involve x, they can be computed in advance. Specifically, we compute "barycentric weights"

$$w_k = \prod_{j\neq k} (x_k - x_j)^{-1} \tag{21.2.3}$$

and then the polynomial p can be computed as the quotient of two rational functions:

$$p(x) = \frac{\sum_{k=1}^{n} y_k w_k (x - x_k)^{-1}}{\sum_{k=1}^{n} w_k (x - x_k)^{-1}} \tag{21.2.4}$$

The advantage of (21.2.4) over the original form is that we avoid a large number of multiplications involving $(x - x_j)$ for every point x. A slight disadvantage is that one cannot plug in $x = x_k$ into (21.2.4), but we know that $p(x_k) = y_k$ anyway.

Example 21.2.1 Plot the Lagrange polynomial through 10 points. Plot the Lagrange polynomial each of the following data sets:

(a) the points $(k, \sin k)$, $k = 1, \dots, 10$;

(b) the points with x_k as above, and y_k chosen randomly from the standard normal distribution.

Answer.

```
n = 10;
x = 1:n;
y = sin(x);  % or y = randn(1, n);
w = ones(1, n);
for k = 1:n
```

```
    for j = 1:n
        if j ~= k
            w(k) = w(k)/(x(k)-x(j));
        end
    end
end

p = @(t) (y.*w*(t - x').^(-1)) ./ (w*(t - x').^(-1));
t = linspace(min(x)-0.01, max(x)+0.01, 1000);
plot(t, p(t), x, y, 'r*')
```

The double loop computes the weights `w` which do not involve the argument of the interpolating polynomial `p`. This argument is called `t` because `x` is already used for the data points. The interval for plotting is chosen to contain the given x-values with a small margin on both sides: this helps both visualization and evaluation, because we avoid plugging in $t = x_1, x_n$. The barycentric evaluation of the polynomial is vectorized. When `t` is a scalar, `t - x'` is a column vector. Taking reciprocals element-wise gives another column vector. Dot-product with `w` or `y.*w` implements the sums such as $\sum_{k=1}^n w_k/(t - x_k)$.

How does `p` accept a row vector as `t`? The expression `t - x'` is the difference of a row vector and a column vector; in Matlab this creates a matrix where (i, j) entry is $t_j - x_i$. Subsequent operations proceed as above, and the result is a row vector of values $p(t)$. □

21.3 Newton interpolating polynomial

Newton polynomial is a third way of constructing an interpolating polynomial of degree $< n$ through n given points. It is still the same polynomial, since there is only one such polynomial. But the method is different because, yet again, we use a different basis. The Newton basis is somewhere between monomial basis and Lagrange basis, in the following sense. Let $\mathbf{c}$ be the vector of coefficients of interpolating polynomial in some basis; these coefficients are obtained from a linear system $A\mathbf{c} = \mathbf{y}$ where the square matrix A depends on the basis.

- For the monomial basis, the matrix A is dense and often ill-conditioned (it has entries x_j^{i-1} and is known as the Vandermonde matrix.
- For the Lagrange basis, the matrix A is the identity matrix, so the coefficients c_k are simply the given values y_k.
- For the Newton basis, the matrix A is *triangular*. This means there is some work to be done to obtain $\mathbf{c}$ from $\mathbf{y}$ but it is straightforward.

The **Newton polynomial basis** consists of

$$\begin{aligned}
\phi_1(x) &= 1 \\
\phi_2(x) &= x - x_1 \\
\phi_3(x) &= (x - x_1)(x - x_2) \\
\phi_4(x) &= (x - x_1)(x - x_2)(x - x_3)
\end{aligned}$$

and so on. An advantage of Newton polynomial is that it is easier to evaluate than Lagrange polynomial. Also, we will see that adding a new data point is very easy to handle because the computation processes one step at a time. A key fact is that $\phi_j(x_k) = 0$ when $k < j$.

We are looking for $c_1, \dots, c_n$ such that the polynomial $p = c_1\phi_1 + \cdots + c_n\phi_n$ has $p(x_k) = y_k$.

- At $x = x_1$ only ϕ_1 is nonzero, so we must have $c_1 = y_1$
- At $x = x_2$ only ϕ_1, ϕ_2 are nonzero, so $c_1 + c_2(x - x_1) = y_2$. We can solve this for c_2.
- At $x = x_3$ only ϕ_1, ϕ_2, ϕ_3 are nonzero, so $c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) = y_3$. We can solve this for c_3. And so on.

The process of solving for c_k can be described algorithmically, in terms of getting the vector c from the vector y. Indeed, we are trying to match two sides of

$$y(x) = c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) + c_4(x - x_1)(x - x_2)(x - x_3) + \cdots \tag{21.3.1}$$

As noted above, $c_1 = y_1$. Next, subtract c_1 from both sides of (21.3.1)and divide them by $x - x_1$:

$$\frac{y(x) - c_1}{x - x_1} = c_2 + c_3(x - x_2) + c_4(x - x_2)(x - x_3) + \cdots \tag{21.3.2}$$

The coefficient c_2 is the value of the left hand side at $x = x_2$. In Matlab terms, it's the second entry of the array of y-values after we applied the subtract-and-divide process to it.

This repeats: we start with `c=y` and repeat the subtract-and-divide step to turn its term into the coefficients of the Newton polynomial.

```
c = y;
for k=2:n
    c(k:n) = (c(k:n)-c(k-1))./(x(k:n)-x(k-1))
end
```

At the end of the loop, the vector `c` contains the coefficients of Newton polynomial.

Evaluation of Newton's polynomial is done in a similar way, with repeated multiply-and-add process, using one coefficient at a time. Indeed, since

$$c_3(x - x_2)(x - x_1) + c_2(x - x_1) + c_1 = (c_3(x - x_2) + c_2)(x - x_1) + c_1$$

(and similarly for higher degrees), the evaluation of `p(t)` goes as is: Thus: begin with constant function c_n, then in the loop multiply by $(x - x_k)$ and add c_k, where k goes from $n - 1$ to 1.

```
p = c(n)*ones(size(t));
for k=n-1:-1:1
    p = p.*(t-x(k)) + c(k);
end
```

While being simpler than barycentric evaluation (21.2.4) of the Lagrange polynomial, the evaluation of Newton polynomial does not fit into an anonymous function because it uses a loop.

Example 21.3.1 Plot the Newton polynomial through 11 points. Plot the Newton polynomial for the points $(k, \arctan k)$, $k = -5, \ldots, 5$. Then repeat with 5 replaced by 8.

Answer.

```
x = -5:5;
```

```
n = numel(x);
y = atan(x);

c = y;
for k=2:n
    c(k:n) = (c(k:n)-c(k-1))./(x(k:n)-x(k-1));
end

t = linspace(min(x)-0.01, max(x)+0.01, 1000);
p = c(n)*ones(size(t));
for k=n-1:-1:1
    p = p.*(t-x(k)) + c(k);
end
plot(t, p, x, y, 'r*')
```

□

21.4 Homework

1. (Theoretical) Write down explicitly the Newton polynomial that interpolates the points $(-1, u)$, $(0, v)$, $(1, w)$. Its coefficients c_1, c_2, c_3 will depend on u, v, w.

2. (Theoretical) Integrate the polynomial found in Exercise 21.4.1 over the interval $[-1, 1]$. (It is easier to integrate its general form $c_1 + c_2(x+1) + c_3(x+1)x$ first and then plug in the formulas for coefficients.) Explain how this is related to Simpson's integration rule.

3. Use either Lagrange method or Newton method to interpolate the points $(k, \cos k)$, where $k = 1, \dots, 15$. Then do the same with the points $(k, \cos 2k)$, where $k = 1, \dots, 15$. Plot both side by side using `subplot`. Do you observe a difference in the quality of interpolating curves?

Chapter 22

Chebyshev Polynomials and Interpolation

We previously encountered Legendre and Laguerre polynomials. Chebyshev polynomials are another family of orthogonal polynomials which is useful in numerical methods, especially for interpolation. One of their attractive features is that there is a simple explicit formula for their roots and extrema, in contrast to the other orthogonal polynomials that we studied. These polynomials are important tools for interpolation and approximation of functions.

22.1 Estimating the error of polynomial interpolation

One of the uses of polynomial interpolation (Chapter 21) is to approximate a possible complicated function $y = f(x)$ by a polynomial, interpolating the values of the function at some points $x_1, \dots, x_n$. Indeed, this is what we tried in Exercise 21.4.3 for the functions $f(x) = \cos x$ and $f(x) = \cos 2x$, with mixed success. The following theorem is a first step toward understanding what is going on.

Suppose a polynomial p of degree less than n interpolates a smooth function f at some points $x_1, \dots, x_n$ of an interval $[a, b]$. Then for each $x \in [a, b]$ there exists a point $\xi \in [a, b]$ such that

$$f(x) - p(x) = \frac{f^{(n)}(\xi)}{n!} \prod_{k=1}^{n} (x - x_k) \tag{22.1.1}$$

where $f^{(n)}$ is the nth derivative of f.

The above theorem is easier to prove than one might expect. If x is one of interpolation points, then both sides of (22.1.1) are equal to 0. Suppose x is not equal to any interpolation point. Then we can let

$$A = \frac{f(x) - p(x)}{\prod_{k=1}^{n} (x - x_k)}$$

so that $f(x) - p(x) - A \prod_{k=1}^{n} (x - x_k) = 0$. Consider the function

$$g(t) = f(t) - p(t) - A \prod_{k=1}^{n} (t - x_k) \tag{22.1.2}$$

and notice that it has at least $n+1$ roots on the interval $[a,b]$, namely $x, x_1, x_2, \dots, x_n$. Between any two roots of g there is a root of g': this is Rolle's theorem. Therefore, g' has at least n roots on the interval $[a,b]$. Repeat this argument: g'' has at least $n-1$ roots, ... , $g^{(n)}$ has at least 1 root. Call this root ξ and notice that according to (22.1.2),

$$g^{(n)}(\xi) = f^{(n)}(\xi) - 0 - An!$$

hence $A = f^{(n)}(\xi)/n!$, completing the proof of (22.1.1).

What guidance can we extract from (22.1.1)? To minimize the interpolation error, we want the product

$$\prod_{k=1}^{n}(x - x_k) \tag{22.1.3}$$

to be small on the interval $[a,b]$. If the points $x_1, \dots, x_n$ are equally spaced, the product (22.1.3) can become large toward the endpoints, as the following Matlab example shows for 11 points on the interval $[0,7]$.

Example 22.1.1 Plot the product with equally spaced roots. Plot the product (22.1.3) for 11 equally spaced points on the interval $[0,7]$.

Answer.

```
a = 0;
b = 7;
n = 11;
x = linspace(a, b, n);     % interpolation points x1, ..., xn.
t = linspace(a, b, 1000);  % points for plotting
plot(t, prod(t - x'))
```

□

Using larger n in Example 22.1.1 would only make things worse. This indicates that polynomial interpolation at a large number of *equally spaced* points is unadvisable. As it was with numerical integration, a smarter choice of points $x_1, \dots, x_n$ is possible, and once again it is provided by the roots of certain orthogonal polynomials.

22.2 Chebyshev polynomials and interpolation

In Chapter 15 we studied Legendre polynomials P_n which are orthogonal on the interval $[-1,1]$ in the sense that $\int_{-1}^{1} P_n(x)P_k(x)\,dx = 0$ when $n \ne k$. We also use Laguerre polynomials L_n which are orthogonal on $[0,\infty)$ with the weight function e^{-x}, meaning that $\int_{-1}^{1} L_n(x)L_k(x)\,e^{-x}\,dx = 0$ when $n \ne k$. The **Chebyshev polynomials** T_n are orthogonal on the interval $[-1,1]$ with the weight function $1/\sqrt{1-x^2}$, meaning that

$$\int_{-1}^{1} T_n(x)T_k(x)\,\frac{dx}{\sqrt{1-x^2}} = 0 \quad \text{when } n \ne k$$

As for other families of orthogonal polynomials, we have a recursive formula for Chebyshev polynomials: starting with $T_0(x) = 1$ and $T_1(x) = x$ we can compute the rest as

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \tag{22.2.1}$$

The recursion is simpler than for P_n or L_n: there is no division, so all polynomials T_n have integer coefficients.

Example 22.2.1 Plot Chebyshev polynomials T_n. Recursively find the coefficients of Chebyshev polynomials T_n for $n \le 6$. Plot all of them on the interval $[-1, 1]$.

Answer. This is a straightforward modification of Example 15.4.1.

```
p = [1];
q = [1 0];
x = linspace(-1, 1, 1000);
hold on
plot(x, polyval(p, x), x, polyval(q, x));

for n = 1:5
    r = 2*[q 0] - [0 0 p];
    p = q;
    q = r;
    plot(x, polyval(r, x))
end
hold off
```

□

Example 22.2.1 indicates that Chebyshev polynomials oscillate between -1 and 1 on the interval $[-1, 1]$. The following remarkable identity confirms this observation:

$$T_n(\cos\theta) = \cos(n\theta) \tag{22.2.2}$$

for all $\theta \in \mathbb{R}$. The proof is by induction. Base of induction is $n = 0, 1$: for $T_0(x) = 1$ and $T_1(x) = x$ the validity of (22.2.2) is obvious. For the inductive step, assume (22.2.2) holds for $T_0, \dots, T_n$ and use (22.2.1):

$$T_{n+1}(\cos\theta) = 2\cos\theta\cos(n\theta) - \cos((n-1)\theta) = \cos((n+1)\theta)$$

where the second step involves the identity

$$2\cos\alpha\cos\beta = \cos(\alpha-\beta) + \cos(\alpha+\beta)$$

The right hand side of (22.2.2) is zero when $n\theta$ is an odd multiple of $\pi/2$, hence $\theta = (2k-1)\frac{\pi}{2n}$. Plugging $k = 1, \dots, n$ we find that

$$T_n(x) = 0 \quad \text{when } x = \cos\left((2k-1)\frac{\pi}{2n}\right), \ k = 1, \dots, n \tag{22.2.3}$$

and since the polynomial T_n can have at most n real roots, these are all of its roots. They are easy to visualize: draw a semi-circle with interval $[a, b]$ as diameter, divide it into n equal arcs, and project the *midpoint* of each arc back to the interval.

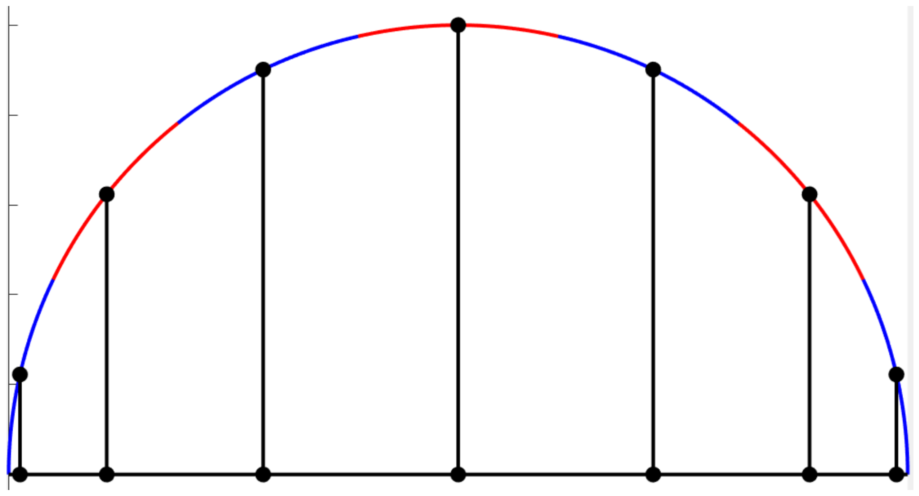

Figure 22.2.2 Chebyshev points of the first kind

The recursive formula shows that the leading coefficient of T_n is 2^{n-1}. Therefore,

$$T_n(x) = 2^{n-1} \prod_{k=1}^{n} (x - x_k)$$

where $x_1, \dots, x_n$ are the roots (22.2.3). It follows that the absolute value of the product $\prod_{k=1}^{n}(x - x_k)$ is bounded by 2^{1-n}. It can be proved that 2^{1-n} is as small as one can get for any choice of n points on the interval $[-1, 1]$. This suggests that Chebyshev points should work well for polynomial interpolation, and they do. Let us re-do Example 21.2.1 with random data to illustrate this.

Example 22.2.3 An interpolating polynomial using Chebyshev points. Let $x_1, \dots, x_{10}$ be the roots of T_{10}. Choose y_k randomly from the standard normal distribution, and draw the interpolating polynomial through the points (x_k, y_k).

Answer. The only change to Example 21.2.1 is replacing the x-coordinates.

```
n = 10;
x = cos((2*(1:n)-1)*pi/(2*n));
y = randn(1, n);
w = ones(1, n);
for k = 1:n
    for j = 1:n
        if j ~= k
            w(k) = w(k)/(x(k)-x(j));
        end
    end
end

p = @(t) (y.*w*(t - x').^(-1)) ./ (w*(t - x').^(-1));
t = linspace(min(x)-0.01, max(x)+0.01, 1000);
plot(x, y, 'r*', t, p(t))
```

□

Running the code Example 22.2.3 several times, we see that the interpolating polynomial behaves reasonably, without unnatural oscillations that come from interpolating at equally spaced points.

22.3 Chebyshev extreme points

Figure 22.2.2 involves the *midpoints* of equal arcs, which should remind us of the Midpoint Rule. There is a parallel "trapezoidal" story, in which we consider the *endpoints* of all these arcs and give half the weight to the smallest and largest.

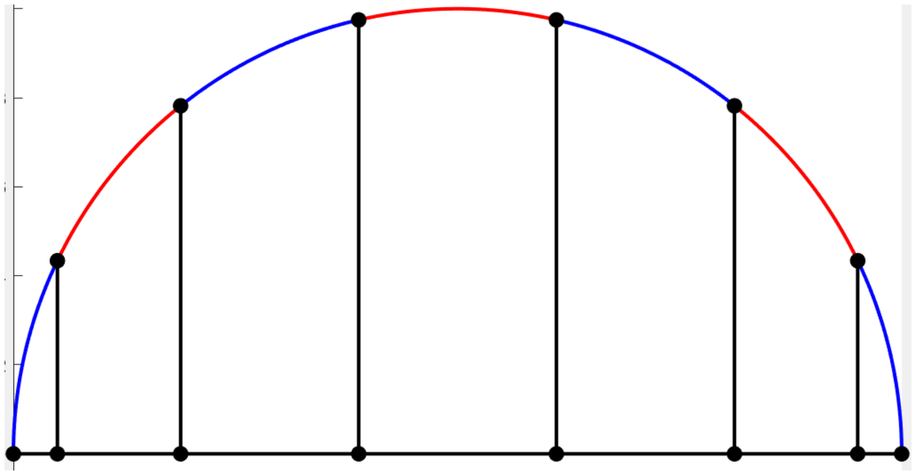

Figure 22.3.1 Chebyshev points of the second kind

These projections are called "Chebyshev extreme points" or "Chebyshev points of the second kind". Explicitly, they are

$$x_k = \cos(\pi k/n) \quad \text{where } k = 0, 1, \dots, n \tag{22.3.1}$$

In Matlab one could write

```
x = cos(linspace(0, pi, n+1));
```

While the midpoint projections (Chebyshev points of the first kind) are the *roots* of polynomials T_n, the points (22.3.1) are the *extreme points* of T_n. Indeed, the identity (22.2.2) says that T_n attains extreme values ± 1 when $\cos(n\theta) = \pm 1$, hence $\theta = \pi k/n$, hence $x = \cos(\pi k/n)$.

An advantage of using Chebyshev extreme points (22.3.1) as the points of interpolation is that one does not need to compute the weights like we did in Example 22.2.3. For this specific choice of points, the weights in the barycentric formula (21.2.4) turn out to be

$$w_k = (-1)^k \quad \text{except } w_0 = 1/2, w_n = (-1)^n/2 \tag{22.3.2}$$

This means that the barycentric formula for interpolation can be written as

$$p(x) = \frac{\sum'_{0\le k\le n} y_k (-1)^k (x - x_k)^{-1}}{\sum'_{0\le k\le n} (-1)^k (x - x_k)^{-1}} \tag{22.3.3}$$

where primes over summation signs mean that the terms with $k = 0, n$ should be divided by 2 (the "trapezoidal" adjustment). Formula (22.3.3) also works for an arbitrary interval $[a, b]$, where the Chebyshev extreme points are

$$x_k = \frac{b-a}{2}\cos(\pi k/n) + \frac{a+b}{2}$$

Note that the function p is in fact a polynomial of degree n, even if the formula does not look like it. The straightforward way of expressing a polynomial of high degree, as a sum of powers with constant coefficients, is likely to cause numerical issues, while the barycentric formula (22.3.3) is numerically stable.

Example 22.3.2 Interpolate through Chebyshev extreme points. Use the formula (22.3.3) to interpolate a few functions such as

- $f(x) = 1/(x^2+1)$ on $[-5, 5]$
- $f(x) = \tan^{-1} x$ on $[-7, 11]$
- $f(x) = \cos(2x)$ on $[1, 15]$
- $f(x) = \sin(x^2)$ on $[0, 5]$
- $f(x) = \operatorname{sign} x$ on $[-1, 1]$

by a polynomial of degree $n = 20$ (or another degree if you prefer).

Answer.

```
f = @(x) 1./(x.^2+1);  % or another f
n = 20;   % or another n
a = -5;   % or another a
b = 5;    % or another b

x = cos(linspace(0, pi, n+1)) * (b-a)/2 + (a+b)/2;
y = f(x);
w = (-1).^(0:n);
w(1) = w(1)/2;
w(end) = w(end)/2;

p = @(t) (y.*w*(t - x').^(-1)) ./ (w*(t - x').^(-1));
t = linspace(a-0.01, b+0.01, 1000);
plot(x, y, 'r*', t, f(t), 'k', t, p(t), 'b', 'LineWidth', 2)
```

The plot includes the points of interpolation as red asterisks, the original function is a black curve, and the interpolating polynomial as a blue curve. Often the blue curve follows the black one so precisely that it covers it completely.

□

22.4 Homework

1. (Theoretical) The *Chebyshev polynomials of the second kind* U_n can be defined as follows: $U_0(x) = 1$, $U_1(x) = 2x$, and after that,

$$U_{n+1}(x) = 2xU_n(x) - U_{n-1}(x) \tag{22.4.1}$$

(This is the same recursive formula as for T_n, but it produces different polynomials because U_1 is different from T_1.) Prove that

$$U_n(\cos\theta)\sin\theta = \sin((n+1)\theta) \tag{22.4.2}$$

for all n and all θ.

Hint: follow the proof of (22.2.2).

2. (Theoretical) Use (22.4.2) to prove that the Chebyshev extreme points $x_k = \cos(\pi k/n)$ with $k = 1, \dots, n-1$ are the roots of polynomial U_{n-1}.

(This is where the name "Chebyshev points of the second kind" comes form.)

3. Suppose we want to draw a smooth *parametric curve* through 10 given points on xy-plane. The points are represented by vectors of their x- and y-coordinates:

```
X = [1 3 5 6 4 1 0 1 3 5];
Y = [1 0 0 2 4 4 6 9 9 7];
```

The command `plot(X, Y)` can join these points by line segments, but that does not make a smooth curve. To obtain a smooth curve, adapt Example 22.3.2 as follows:

- Your script should create two interpolating polynomials (`px` and `py`) on the interval $[-1, 1]$. One is computed from the values `X` and the other from the values `Y`.

- At the end of the script, the curve through the given points can be displayed with

```
t = linspace(-1, 1, 1000);
plot(X, Y, 'r*',  px(t), py(t), 'b')
```

The plot should show a smooth S-shaped curve passing through the points marked by asterisks.

Chapter 23

Spline Interpolation

Splines are piecewise defined functions where the pieces are polynomials, usually of low degree (3 is most popular degree). Although it is sometimes possible to efficiently use high-degree polynomial for interpolation (with Chebyshev points and barycentric evaluation), spline interpolation is often simpler and preferable, especially as it does not require special choice of interpolation points and has simple evaluation.

23.1 Piecewise linear interpolation

The current version of SU COVID dashboard uses *piecewise linear interpolation*. Geometrically this concept is clear: connect the points by straight lines, which is also what Matlab's `plot` does.

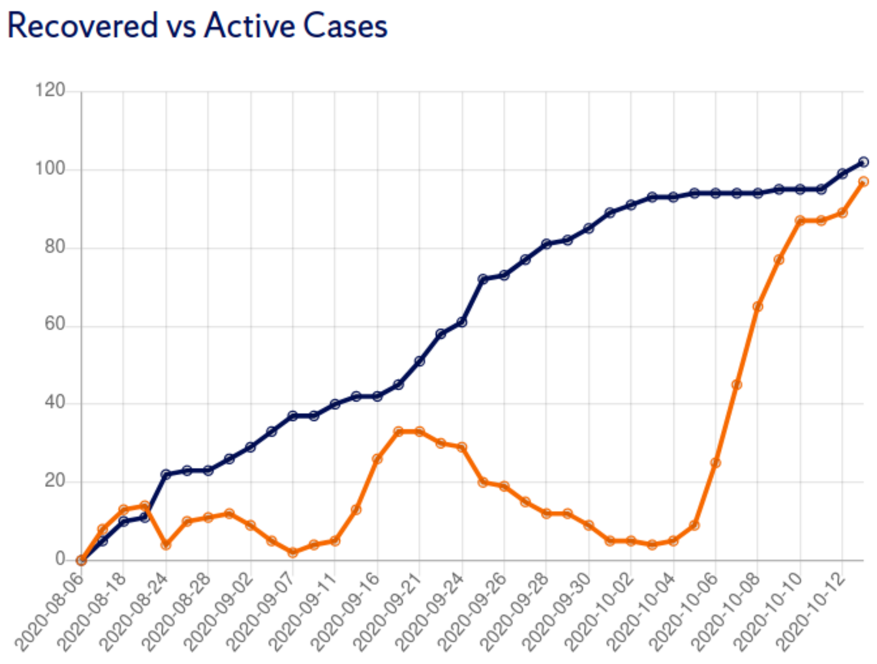

Figure 23.1.1 Piecewise linear interpolation

Formally, piecewise linear interpolation means that given some points $a = x_1 < x_2 < \cdots < x_n = b$ on an interval $[a, b]$ and corresponding y-values

$y_1, \dots, y_n$, we define a function $S\colon [a,b] \to \mathbb{R}$ by the piecewise formula

$$S(x) = y_k + \frac{y_{k+1} - y_k}{x_{k+1} - x_k}(x - x_k), \quad x_k \le x \le x_{k+1} \tag{23.1.1}$$

where the right hand side is a linear function through two points (x_k, y_k) and (x_{k+1}, y_{k+1}). So, instead of 1 interpolating polynomial of degree $n-1$ we use $n-1$ polynomials of degree 1. The function S is a spline of degree 1 meaning it is a piecewise function where the pieces are polynomials of degree 1.

A more abstract approach to the construction of degree 1 spline is as follows:

- We have $n-1$ subintervals between n data points.
- On each subinterval we choose a degree 1 polynomial; this means we have $2(n-1)$ coefficients to choose.
- But we want to spline to be continuous, so the values at each interior data point from the left and from the right must match. There are $n-2$ interior data points, so the continuity imposes $n-2$ equations.
- Finally, the spline must pass through each of n data points. This adds n equations.
- The total is $n-2+n = 2n-2$ equations with $2(n-1) = 2n-2$ unknowns, so we expect (and get) a unique solution.

There are infinitely many ways to connect given data points. The function (23.1.1) does so by a curve of minimal length, meaning that the graph of S is the shortest curve passing through the given data points. This is a reasonable way of connecting the points, but not necessarily the best or most natural way. The corners at the data points do not look natural. The piecewise linear graph does not give much insight into the rate of change of the function because it treats the rate as constant on each interval. For example, given the function values

```
x = 1:5;
y = [8 5.5 5.3333 5.75 6.4];
```

what is the derivative of the function at $x = 2$? Differentiating the piecewise linear interpolant gives some information but not much. Another question one might ask is: where is the minimum of this function? The minimum of piecewise linear interpolant is exactly at $x = 3$ but we understand the actual function probably has its minimum between 2 and 3.

23.2 Cubic splines

A cubic spline is a function S that is piecewise a degree 3 polynomial, and such that S, S', S'' are continuous (have the same value from both sides at the points where pieces meet). In our context of fitting to data points (x_k, y_k), $k = 1, \dots, n$, it is natural to have each cubic polynomial defined on $[x_k, x_{k+1}]$. The continuity of first and second derivatives means that the graph of a cubic spline has no corners of sudden changes of curvature, it flows quite naturally. Compare the plots below, for the same data set.

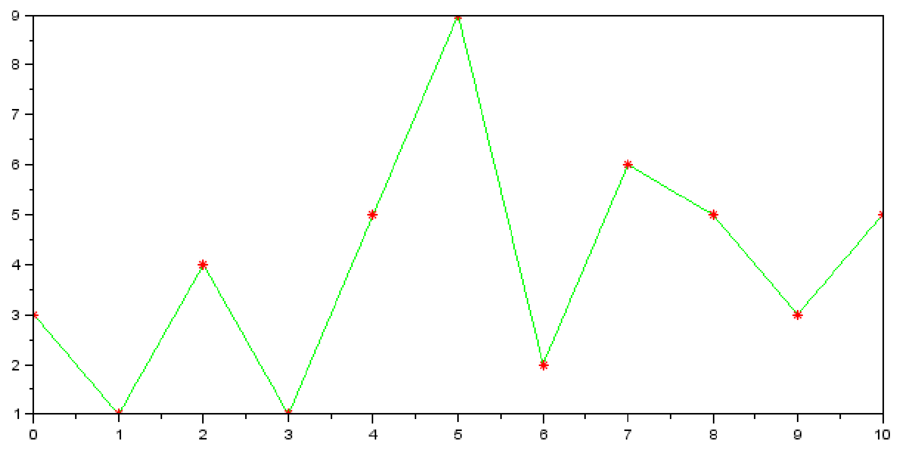

Figure 23.2.1 Piecewise linear interpolation

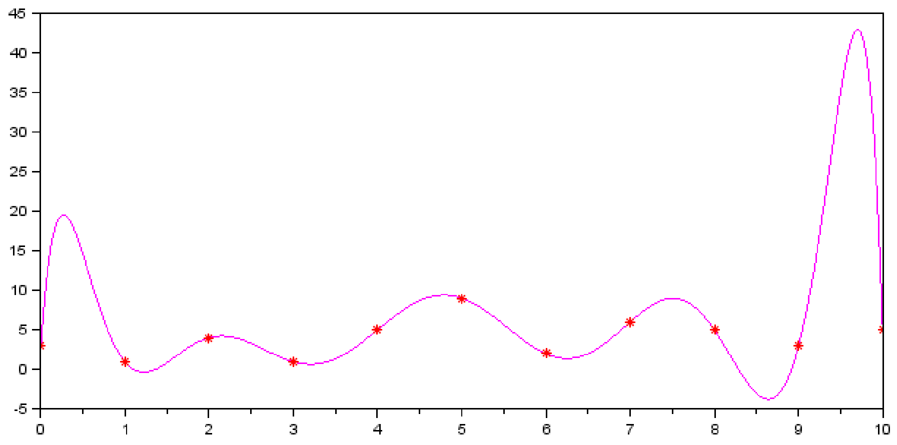

Figure 23.2.2 Interpolating polynomial

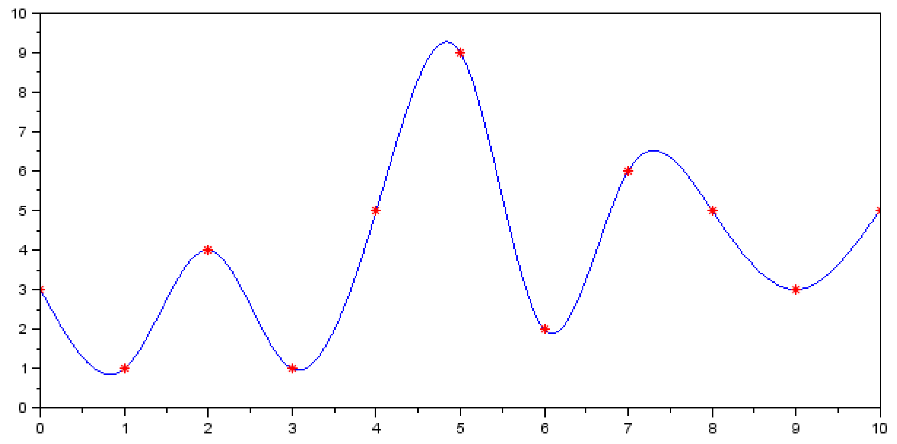

Figure 23.2.3 Cubic spline

Cubic splines were also used by the first version of SU COVID dashboard:

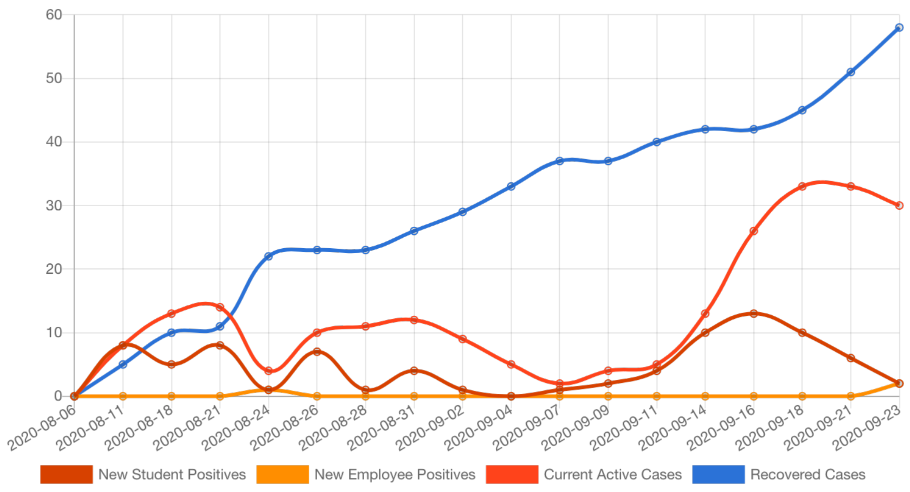

Figure 23.2.4 Dashboard with splines

Let us count unknowns and equations involved in the construction of a cubic spline.

- We have $n-1$ subintervals between n data points.
- On each subinterval we choose a degree 3 polynomial; this means we have $4(n-1)$ coefficients to choose.
- In order for S, S', S'' to be continuous at each of $n-2$ interior data points, there are $3(n-2)$ equations to satisfy.
- The spline must pass through each of n data points. This adds n equations.
- The total is $3(n-2)+n = 4n-6$ equations with $4(n-1) = 4n-4$ unknowns, so need two more equations to make the solution unique.

When discussing spline coefficients, it should be noted they are not necessarily the coefficients of powers of x. It is more natural and numerically safer to represent the spline on each subinterval $[x_k, x_{k+1}]$ in terms of the powers of $(x-x_k)$. That is,

$$S(x) = A_k(x-x_k)^3 + B_k(x-x_k)^2 + C_k(x-x_k) + D_k \tag{23.2.1}$$

where the coefficients a_k, b_k, c_k, d_k are to be determined. Of course, $D_k = y_k$ but to find the rest one needs to solve a linear system.

Here are three different ways to add two more equations to the linear system for coefficients; they result in three different kinds of cubic splines:

- Require $S''(a) = 0 = S''(b)$. This is a “natural spline”.
- Require $S'(a) = p$ and $S'(b) = q$ for some values p, q. This is a “clamped spline”.
- Require S''' to be continuous at x_2 and at x_{n-1}. This is a “not-a-knot spline”.

This terminology needs some explanation. The word spline used to refer to a long thin strip of flexible material (e.g., metal) which, if constrained at several points, assumes the shape of a smooth curve passing through these points. Unlike an elastic band which minimizes its length (creating a piecewise linear shape), the spline minimizes its *bending energy.* A mathematical idealization of the bending energy of a curve $y = f(x)$ is the integral $\int_a^b f''(x)^2\,dx$. It can be shown that among *all functions* (of whatever form) that satisfy the constraints $f(x_k) = y_k$, the "natural spline" has the smallest value of $\int_a^b f''(x)^2\,dx$. This attractive mathematical property earned it the "natural" name… but upon further reflection, the boundary conditions $S''(a) = 0 = S''(b)$ are actually not that natural. If we are interpolating some smooth function $y = f(x)$, there is no reason to assume that f'' happens to be 0 at the endpoints.

Clamped splines refer to holding the ends of the metal strip with a clamp, which restricts not only its position but also the angle. This spline is useful for for modeling a problem with boundary conditions on the derivative.

The *knots* of a spline are the transition points where it turns from one polynomial to another. These are often, but not always, the points x_k from the given data. Requiring S''' to be continuous at x_2 implies we have the same polynomial on both intervals $[x_1, x_2]$ and $[x_2, x_3]$ (why?), and therefore x_2 is *not a knot* for this spline. Same for x_{n-1}. The not-a-knot condition is the standard choice when we do not have any reason to impose other conditions.

Question 23.2.5 Is this a cubic spline? Is the following function a spline? Why or why not?

$$f(x) = \begin{cases} 1 + x - x^3, & 0 \le x \le 1, \\ x, & 1 \le x \le 2 \end{cases}$$

□

Example 23.2.6 Complete the formula for a natural spline. Complete the formula

$$S(x) = \begin{cases} 1 + x - x^3 & 0 \le x \le 1, \\ \dots & 1 \le x \le 2 \end{cases}$$

given that it describes a natural spline on $[0, 2]$.

Answer. The continuity of S, S', S'' implies that the difference of two polynomials used on both sides of $x = 1$ must be a constant multiple of $(x-1)^3$. Therefore, the second polynomial must be of the form $1 + x - x^3 + c(x-1)^3$. Using the natural spline condition $S''(2) = 0$ we can find $c = 2$. Thus,

$$S(x) = \begin{cases} 1 + x - x^3, & 0 \le x \le 1, \\ 1 + x - x^3 + 2(x-1)^3 & 1 \le x \le 2 \end{cases}$$

□

23.3 Using cubic splines

We do not go into the process of finding the coefficients of a cubic spline: it involves solving a linear system with only a few nonzero diagonals. In Matlab, the command `spline` computes a knot-a-knot spline from given data, for example

```
x = 1:7;
y = [1 3 2 4 4 1 5];
s = spline(x, y);
disp(s.coefs)
```

Each row of the output contains the coefficients A, B, C, D of cubic polynomial (23.2.1) which is used on the interval $[x_k, x_{k+1}]$. We could use these coefficients to evaluate the spline, but Matlab can do this automatically with `ppval` (meaning "Piecewise Polynomial VALues"). Specifically, `ppval(s, t)` evaluates the spline at a value (or vector of values) `t`. Thus,

```
t = linspace(min(x), max(x), 1000);
plot(t, ppval(s, t))
```

plots the spline computed above.

There is also an option to combine construction and evaluation of a spline into the same command `spline`, by feeding `t` as a third argument. That is, `spline(x, y, t)` constructs a spline from `x, y` and immediately evaluates it at `t`. For example:

```
t = linspace(min(x), max(x), 1000);
plot(t, spline(x, y, t))
```

We can also plot it with the original data points marked:

```
plot(t, spline(x, y, t), x, y, 'r*')
```

How else could we use the spline data? The formula (23.2.1) shows that $S'(x_k) = C_k$, which is the entry of the 3rd column of matrix `s.coefs`. Try `s.coefs(:, 3)'` and compare the values with the slope of the spline at the data points. This is another approach to *numerical differentiation*, different from what we did in Chapter 12: given discrete data, fit a spline to it and differentiate the spline.

Looking at the coefficients, we can also find the critical points of a spline, answering the question raised at the end of Section 23.1. Indeed, we have A, B, C, D coefficients of a cubic polynomial, so its derivative is a quadratic polynomial with coefficients $3A, 2B, C$. We can find its roots with `roots` but one must be careful to check if a root indeed falls into the subinterval on which this polynomial is used.

Example 23.3.1 Critical points of a cubic spline. Find and plot the critical points of the spline constructed from the data

```
x = 1:7;
y = [1 3 2 4 4 1 5];
```

Answer. Since the number of critical points is not known in advance, we can gather them by appending each new critical point to the vector `critpts`. To qualify for inclusion, a root of derivative of the cubic polynomial used on $[x_k, x_{k+1}]$ must be real and must lie within this interval.

```
s = spline(x, y);
critpts = [];
for k=1:numel(x)-1
    r = roots([3 2 1].*s.coefs(k, 1:3));
    for j = 1:numel(r)
        if isreal(r(j)) && r(j) >= 0 && r(j) <= x(k+1)-x(k)
            critpts = [critpts, x(k)+r(j)];
        end
    end
end
```

```
t = linspace(min(x), max(x), 1000);
plot(t, ppval(s, t), x, y, 'r+', critpts, ppval(s, critpts), 'k*')
```

The script uses `ppval` since the spline `s` is already constructed; calling `spline(x, y, ...)` repeatedly with the same `x, y` seems wasteful. □

23.4 Homework

1. (Theoretical) Complete the following formula for a clamped spline with boundary conditions $S'(0) = -1$ and $S'(3) = 6$:

$$S(x) = \begin{cases} \dots & 0 \le x \le 1, \\ 5 - 2x + x^3 & 1 \le x \le 2, \\ \dots & 2 \le x \le 3 \end{cases}$$

2. Adapt Example 23.3.1 to find and plot the *inflection points* of the same cubic spline (these are the points where the second derivative is zero.)

Chapter 24

Spline Approximation

One reason to interpolate the data is to obtain a function that approximates it. Since the data values are not necessarily exact, it is not always important to match them precisely. In this chapter we use a simple approximation device to build splines that approximate the data without interpolating it. The process has some attractive features which both spline interpolation and polynomial interpolation lack.

24.1 Weak points of spline interpolation

Looking closely at Figure 23.2.4 we can notice that the blue curve is not quite an increasing function: there is a small interval in which it decreases. This is not really logical since the curve follows the data (cumulative number of recovered cases) which cannot decrease. A smaller example is below.

```
x = 1:6;
y = [0 0 1 1 2 2];
t = linspace(1, 6, 1000);
plot(t, spline(x, y, t), x, y, 'r*')
```

So, cubic splines *do not preserve monotonicity* (while piecewise linear interpolation does). Note that SU COVID dashboard later replaced splines with piecewise linear functions, which are monotone when the data values are monotone.

Another issue is that spline interpolation may produce *negative values from positive data.*

```
x = 1:6;
y = [9 9 1 1 9 9];
disp(spline(x, y, 3.5))
```

The result is -0.4. This becomes a serious issue if the quantity being interpolated cannot in principle be negative (mass, density, number of people) and if the way in which the data is used relies on it not being negative (maybe one takes square root or plots the data on logarithmic scale).

Note again that this is something that piecewise linear interpolation would not do. A piecewise linear function through points with $y_k > 0$ stays positive.

24.2 Hat functions and B-splines

We look for a way to smoothly model the data while avoiding the issues in Section 24.1. To simplify the computations, assume the x-values are equally spaced, which means that after shifting and scaling we can assume $x_k = k$ for $k = 1, \dots, n$. Then the linear interpolation can be expressed as the sum

$$L(x) = \sum_{k=1}^{n} y_k H(x-k) \tag{24.2.1}$$

where $H(x) = \max(0, 1-|x|)$ is a function whose graph is a triangle with base $[-1, 1]$. Functions like H are sometimes called the "hat functions" based on the appearance of the graph. An important feature of the functions $H(x-k)$ is that they add up to 1 everywhere on the interval $[1, n]$ (why?). This ensures that interpolation of constant values $y_1 = \cdots = y_n$ produces a constant function, which is a necessary condition for the algorithm to preserve monotonicity of data.

As we know, piecewise linear interpolation lacks smoothness, and this is seen in the angular shape of the hat function (24.2.1). We would like to replace it with something of similar shape but smooth. Here is a piecewise cubic function which does the job.

$$B(t) = \frac{1}{6}\max(0, 2-|t|)^3 - \frac{2}{3}\max(0, 1-|t|)^3 \tag{24.2.2}$$

Its graph on $[-3, 3]$ is shown below, with the hat function H for comparison. In technical terms, both H and B are "cardinal B-splines" of degrees 1 and 3, respectively.

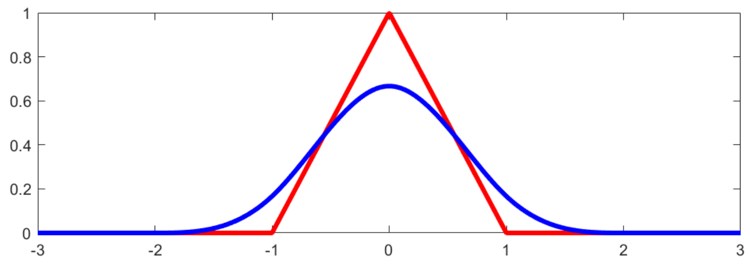

Figure 24.2.1 Cardinal B-splines of degrees 1 and 3

Question 24.2.2 Smoothness of cardinal B-spline. Despite the presence of absolute value and max, both potential sources of non-smoothness, the function (24.2.2) is differentiable and even twice differentiable, with continuous B''. (This qualifies it as a cubic spline.) Why is this so? ☐

Before considering how the smooth bump function (24.2.2) was constructed, let us try using it. One is the straightforward approach: replace H with B in the formula (24.2.1), obtaining a cubic spline

$$S(x) = \sum_{k=1}^{n} y_k B(x-k) \tag{24.2.3}$$

The first thing to notice is that the spline S *does not interpolate the data.* Indeed, from (24.2.2) we have $B(0) = 2/3$ and $B(\pm 1) = 1/6$, which implies that

$$S(x_k) = \frac{y_{k-1} + 4y_k + y_{k+1}}{6} \tag{24.2.4}$$

which is a Simpson-style averaging of neighboring y-values. It may be disappointing not to have $S(x_k) = y_k$ but on the other hand, averaging raw data values within some moving window is a standard approach to eliminating noise and short-term fluctuations.

It is also possible to derive an interpolating spline as a linear combination of bumps $B(x-k)$, by solving a simple linear system. Indeed, we can find the coefficients $c_1, \dots, c_n$ such that

$$\sum_{k=1}^{n} c_k B(x-k) = y_k, \quad k = 1, \dots, n$$

by solving the system $Ac = y$ where A is the *tridiagonal* matrix

$$A = \frac{1}{6}\begin{pmatrix} 4 & 1 & 0 & \cdots & 0 \\ 1 & 4 & 1 & \cdots & 0 \\ 0 & 1 & 4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

But this brings back the kind of splines in Chapter 23 with the deficiencies noted earlier, so we do not pursue this approach further.

As written, the formula (24.2.3) has an issue at the boundary, where instead of averages (24.2.4) we get

$$S(x_1) = \frac{4y_1 + y_2}{6}, \quad S(x_n) = \frac{y_{n-1} + 4y_n}{6} \tag{24.2.5}$$

because there is no y_0 or y_{n+1}. This is an issue because, for example, with constant values $y_k = 1$ we get $S(x_1) = 5/6 = S(x_n)$, failing monotonicity. To fix the boundary behavior, we should extend the sum (24.2.3) with two more terms:

$$S(x) = \sum_{k=0}^{n+1} y_k B(x-k), \quad y_0 = y_1,\ y_{n+1} = y_n \tag{24.2.6}$$

This is typical when computing with values on a grid: sometimes our formulas refer to out-of-grid points, and instead of just ignoring such terms we should somehow extrapolate the known values outside of the grid.

Example 24.2.3 Implement cubic spline approximation. Use the examples from Section 24.1 to demonstrate that the spline formula (24.2.6) preserves monotonicity and positivity.

Answer.

```
B = @(t) (1/6)*max(0, 2-abs(t)).^3 - (2/3)*max(0, 1-abs(t)).^3;
x = 1:6;
y = [0 0 1 1 2 2];
xe = [x(1)-1, x, x(end)+1];
ye = [y(1), y, y(end)];

t = linspace(min(x), max(x), 1000);
S = ye*B(t - xe');
plot(t, S, x, y, 'r*')
```

As discussed above, the x-y values are extended by two points to the left and right; this is what the vectors `xe` and `ye` represent. After that, the spline is evaluating in a single vectorized computation. The plot shows a smoothly increasing function. Trying this again with `y = [9 9 1 1 9 9]` we get a smooth positive function. □

24.3 Construction of cardinal B-splines

The formula (24.2.2) deserves some explanation. The idea is to start with the linear interpolant (degree 1 spline) $S_1 = L$ defined by (24.2.1) and repeatedly smoothen it using a **moving average** with a window of size 1:

$$S_2(x) = \int_{x-1/2}^{x+1/2} S_1(t)\, dt \tag{24.3.1}$$

$$S_3(x) = \int_{x-1/2}^{x+1/2} S_2(t)\, dt \tag{24.3.2}$$

and so on. The fundamental theorem of calculus shows that

$$S_2'(x) = S_1(x+1/2) - S_1(x-1/2)$$

in particular S_2 has continuous first derivative. Similarly,

$$S_3'(x) = S_2(x+1/2) - S_2(x-1/2)$$

so

$$S_3''(x) = S_2'(x+1/2) - S_2'(x-1/2)$$

is continuous as well.

The function S_3 defined by (24.3.2) is the approximating cubic spline discussed above, while S_2 is a quadratic spline. But these integral formulas should be simplified for practical use. By linearity of integrals,

$$S_2(x) = \sum_{k=1}^{n} y_k \int_{x-1/2}^{x+1/2} H(t)\, dt$$

where the integral can be evaluated explicitly: it is a cardinal B-spline of degree 2, which can be written out

$$\int_{x-1/2}^{x+1/2} H(t)\, dt = \begin{cases} 0 & x \le -3/2 \\ (x+3/2)^2/2 & -3/2 \le x \le -1/2 \\ 3/4 - x^2 & -1/2 \le x \le 1/2 \\ (x-3/2)^2/2 & 1/2 \le x \le 3/2 \\ 0 & x \ge 3/2 \end{cases}$$

Quadratic B-splines are awkward to use because of the fractional transition points above, which do not align with grid points. Averaging again by (24.3.2) improves the situation and leads, after tedious case-by-case computations, to the formula (24.2.2).

24.4 Homework

1. Following Example 24.2.3, plot an approximating cubic spline for the data $(k, \tan^{-1} k)$, $k = -5, -4, \dots, 5$. For comparison, also plot the actual function $y = \tan^{-1} x$ on the interval $[-5, 5]$. How accurate is the approximation? Plot the difference of the spline and arctangent to answer this question.

(Recall that arctangent is `atan` in Matlab.)

2. Let us revisit Example 3.2.1, changing it to the following.

```
k = 2;
A = rand(k, 1e7) - 1/2;
S = sum(A, 1);
histogram(S, 100, 'Normalization', 'pdf')
```

(If using Octave where `histogram` is not available, replace the last line with `hist(S, 100, 100/k)`.)

This produces a *normalized* histogram of 10^7 sample sums, with k numbers in each sample. The numbers are chosen from uniform distribution on $[-1/2, 1/2]$. Normalization means that the area under the histogram is 1 (in probability terms, this histogram approximates the *probability density function* of sample sums, hence the option "pdf".)

What should the value k be in order for the histogram to resemble the cardinal B-splines of degree 3 shown in Figure 24.2.1? Find the answer experimentally after trying a few values of k. Having found a suitable k, confirm your guess by adding the graph of the cardinal spline to the histogram plot (recall `hold on` / `hold off` are useful for combining graphs.)

Chapter 25

Discrete Fourier Transform

Some functions are naturally periodic, for example the parametric formulas for a closed curve. The interpolation technique considered so far involve (piecewise) polynomials which are not periodic. Their natural analogue for periodic functions is *trigonometric polynomials*. It turns out that the theory of interpolation or approximation of periodic functions by simpler periodic functions is computationally simpler than the corresponding theory for functions on an interval.

25.1 Periodic functions and trigonometric polynomials

A function f is **periodic** if there exists a constant T such that $f(x+T) = f(x)$ for all $x \in \mathbb{R}$. Examples include $\sin x$, $\cos x$, and their combinations called *trigonometric polynomials*:

$$p(x) = \sum_{k=0}^{n} a_k \cos(2\pi kx/T) + \sum_{k=1}^{n} b_k \sin(2\pi kx/T) \tag{25.1.1}$$

The function (25.1.1) has period T. Using Euler's identity

$$e^{ix} = \cos x + i \sin x \tag{25.1.2}$$

one can express trigonometric polynomials in *complex exponential form*:

$$p(x) = \sum_{k=-n}^{n} c_k \exp(2\pi ikx/T) \tag{25.1.3}$$

Indeed, the identity (25.1.2) implies $\cos x = (e^{ix} + e^{-ix})/2$ and $\sin x = (e^{ix} - e^{-ix})/(2i)$, so the expression (25.1.1) can be rewritten as a linear combination of complex exponentials as in (25.1.3). Conversely, expression (25.1.3) can be rewritten in terms of trigonometric functions using (25.1.2).

It is important to recognize that even if the coefficients a_k, b_k were real, the corresponding coefficients c_k will in general be complex. Complex numbers in Matlab can be entered directly like `3 + 4i` (note there is no `*` between 4 and i). If you have variables `a, b` and want to construct $a + ib$, then `a + 1i*b` works. Also, the letter `j` can be used in place of `i`.

The most common values of T are $T = 2\pi$ and $T = 1$. Each choice makes some formulas simpler at the cost of making other formulas more complicated.

This is why the theory of Fourier transform is presented in different sources with different notational conventions.

Example 25.1.1 Plot a random trigonometric polynomial. Plot a random trigonometric polynomial of degree 3, using complex exponential form with complex coefficients taken from the normal distribution. Note that Matlab command `plot(z)` when given a single vector `z` with complex values, interprets it as `plot(real(z), imag(z))`.

Answer. This example uses $T = 2\pi$ (the choice of period does not change these plots). Note the vectorized evaluation `c*exp(1i*k'*t)` and recognize the role of transpose `k'` here.

```
n = 3;
c = randn(1, 2*n+1)  + 1i*randn(1, 2*n+1);
t = linspace(0, 2*pi, 1000);
k = -n:n;
z = c*exp(1i*k'*t);
plot(z)
```

□

25.2 Fourier series

Using the period $T = 1$, we find that the functions $f_k(x) = \exp(2\pi ikx)$ are *orthonormal* with respect to the inner product

$$\langle f, g\rangle = \int_0^1 f(x)\overline{g(x)}\,dx \tag{25.2.1}$$

(For other periods they are orthogonal but not orthonormal.) This makes it easy to compute the **Fourier coefficients** of a given function f in the basis $\{f_k : k \in \mathbb{Z}\}$:

$$c_k = \langle f, f_k\rangle = \int_0^1 f(x)e^{-2\pi ikx}\,dx \tag{25.2.2}$$

For sufficiently nice periodic functions (having a continuous derivative is enough), the **Fourier series** $\sum c_k f_k$ converges to function f.

Example 25.2.1 Compute Fourier coefficients. Use the built-in command `integral` to compute the Fourier coefficients c_k for $k = -3, \dots, 3$ for the complex-valued function

$$f(x) = \sqrt{1 + \cos(2\pi x)} + i\log(2 + \sin(2\pi x))$$

Then plot both the function and its partial Fourier sum $\sum_{|k|\le 3} c_k f_k$ for comparison.

Answer. The syntax of `integral` is `integral(@(x) ..., a, b)` where a, b are the limits of integration and the function is expressed in a form that allows vectorized evaluation. Here is the computation of Fourier coefficient

```
f = @(x) sqrt(1 + cos(2*pi*x)) + 1i*log(2 + sin(2*pi*x));
n = 3;
c = zeros(1, 2*n+1);
for k = -n:n
    c(k+n+1) = integral(@(x) f(x).*exp(-2*pi*1i*k*x), 0, 1);
end
disp(c)
```

Then we combine both complex plots using different colors.

```
t = linspace(0, 1, 1000);
k = -n:n;
p = c*exp(2*pi*1i*k'*t);
hold on
plot(f(t))
plot(p, 'r')
hold off
```

Note that the lack of smoothness at one point slows down the convergence of Fourier series. If the function had `sqrt(1.1 + cos(2*pi*x))` instead, the convergence would be much better. □

25.3 Discrete Fourier Transform

There is a discrete version of the Fourier series theory outlined in Section 25.2. In it we fix an integer n and replace a function $f\colon [0,1] \to \mathbb{C}$ by its sampled values $z_k = f(k/n)$, $k = 0, 1, \dots, n-1$. The discrete version of inner product (25.2.1) is

$$\langle f, g\rangle_d = \sum_{\ell=0}^{n-1} f(\ell/n)\overline{g(\ell/n)} = \sum_{k=0} z_\ell \overline{w_\ell} \tag{25.3.1}$$

The functions $f_k = \exp(2\pi i k)$, $k = 0, \dots, n-1$, are orthogonal with respect to this inner product. (Not orthonormal, as $\langle f_k, f_k\rangle = n$.) Note we only need n of these functions (why?).

The **Fourier coefficients** of a given discretized function $(z_0, z_1, \dots, z_{n-1})$ are computed as:

$$c_k = \sum_{\ell=0}^{n-1} z_\ell \exp(-2\pi i k\ell/n) \tag{25.3.2}$$

and from there, the function is precisely reconstructed as

$$z_\ell = \frac{1}{n}\sum_{k=0}^{n-1} c_k \exp(2\pi i k\ell/n) \tag{25.3.3}$$

Some people put $1/n$ in (25.3.2) instead of (25.3.3), and some put $1/\sqrt{n}$ in both, for symmetry.

The computation of Fourier coefficients (25.3.2) amounts to matrix-vector multiplication $c = Wz$ where W is the **Fourier matrix** defined as follows. Let $\omega = \exp(-2\pi i/n)$. The matrix W has size $n \times n$, with entries given by $W_{k\ell} = \omega^{(k-1)(\ell-1)}$. Explicitly,

$$W = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(n-1)} \\ \dots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{pmatrix} \tag{25.3.4}$$

Normally, matrix-vector multiplication takes on the order of n^2 operations. The **Fast Fourier Transform** (FFT) is an algorithm that uses the specific structure of W to reduce the number of computations to something of the order $n \log n$. The reconstruction of function values from coefficients (25.3.3) is very similar. Besides the factor of $1/n$, the only other difference is replacing matrix W with matrix W^* of the same form (25.3.4) but with $\omega = \exp(2\pi i/n)$.

In Matlab, the computation (25.3.2) is implemented by command `fft` while (25.3.3) is implemented by `ifft` ("i" for "inverse"). Try applying them to standard basis vectors and compare the output to the columns of W or $\frac{1}{n}W^*$. (The commands also allow matrix input: `fft(eye(n))` and `ifft(eye(n))` return the matrices W and $\frac{1}{n}W^*$.)

25.4 Trigonometric Interpolation

Discrete Fourier Transform allows us to interpolate any periodic function by a trigonometric polynomial by first discretizing it as $z_k = f(k/n)$, $k = 0, \dots, n-1$, applying FFT to compute Fourier coefficients c_k, and then constructing a trigonometric polynomial with these coefficients. By literally following the formula (25.3.3) we arrive at the trigonometric polynomial

$$p(x) = \frac{1}{n}\sum_{k=0}^{n-1} c_k \exp(2\pi i k x) \tag{25.4.1}$$

This indeed interpolates the function but not in a good way, as the following code shows. It interpolates the function in Example 25.2.1 using `n=9` points, which are marked as black asterisks.

```
f = @(x) sqrt(1 + cos(2*pi*x)) + 1i*log(2 + sin(2*pi*x));
n = 9;
z = f((0:n-1)/n);
c = fft(z);
k = 0:n-1;
p = c/n*exp(2*pi*1i*k'*t);

t = linspace(0, 1, 1000);
hold on
plot(f(t))
plot(p, 'r')
plot(z, 'k*')
hold off
```

The problem is that our formula for `p` involves complex exponential functions with "frequencies" $k = 0, 1, \dots, 9$ which is not how trigonometric polynomials (25.1.3) normally look. The high frequencies create the oscillatory behavior which, while matching the data, does not create a reasonable interpolant.

A key term here is **aliasing**, which is the fact that different trigonometric (or complex exponential) functions may coincide when restricted to a discrete grid of points (illustration). Specifically,

$$\exp(2\pi i k t) = \exp(2\pi i (k-n) t), \quad t = 0, 1/n, 2/n, \dots, 1$$

In order to avoid extraneous oscillations we should subtract n from large elements of the vector `k = 0:n-1` above, so that instead of

```
k = [0 1 2 3 4 5 6 7 8];
```

we have

```
k = [0 1 2 3 4 -4 -3 -2 -1];
```

Replacing `k = 0:n-1` in the above code with `k = [0:(n-1)/2 (1-n)/2:-1];` corrects the issue.

The previous paragraph operates with an odd number n which is more natural in this context because trigonometric polynomials (25.1.3) normally contain an odd number of terms. But what if we are given an even number of points to interpolate through, such as 10? It is not clear whether to keep $k = n/2$ as positive: `k=[0:n/2 1-n/2:-1]` or to move it to negative: `k=[0:n/2-1 -n/2:-1]`. Neither version works well. Their average does work; essentially, the term with $k = n/2$ should be split into two. To avoid this complication, we stick to odd n.

25.5 Homework

1. (Theoretical) Suppose n is an odd positive integer and that the numbers $z_0, \dots, z_{n-1}$ are real and satisfy the symmetry condition $z_{n-\ell} = z_\ell$ for $\ell = 1, \dots, n-1$. Show that the Discrete Fourier Transform (25.3.2) also has real values.

2. Choose some complex numbers $z_0, z_1, \dots, z_{n-1}$ (with odd n) so that connecting them in this order (and connecting z_{n-1} to z_0) one gets a closed curve similar to letter B. Then use Discrete Fourier Transform to plot a trigonometric polynomial interpolating these numbers. Does its plot look like letter B?

Chapter 26

Applications of Discrete Fourier Transform

Having computed the Fourier coefficients of a function (which may represent, for example, a sound recording or an image) we often discover that most of the coefficients are relatively small. One can try to discard small Fourier coefficients and reconstruct the function/sound/image from what remains, thus achieving some amount of data compression.

26.1 Working with images in Matlab

An image is loaded into Matlab with a command like `im = imread('filename.jpg')`. Try this using, for example, the winter scene (first save the file in Matlab/Octave current directory). There are several things to learn about this `im` object:

- `size(im)` shows its size (dimensions) as $462\times692\times3$. This is a three-dimensional array, where the last index means the **color channel**: Red, Green, Blue. The first two indices are rows and columns. One can access individual channels with `im(:, :, 1)` (for red). Matlab can also work with **grayscale** images which have just one channel for brightness; such images are represented by matrices which makes them easier to work with.

- `class(im)` says "uint8". All other Matlab objects we worked have class `double` meaning they are floating-point numbers with double-precision, able to store about 16 decimal digits. But `uint8` is different: it is an **unsigned 8-bit integer**, with possible values $0, 1, \dots, 255$. This reflects the image format: in each channel, the brightness of each pixel ranges from 0 to 255. (How many possible colors does this create?) In other languages, one would say "type" instead of "class".

The command `imshow(im)` displays the image. An image (an array of numbers) can be transformed in multiple ways and then either saved as a file with `imwrite` or just displayed with `imshow`; we will do the latter, which is simpler. Try the following and observe the effects:

- Remove one of colors entirely, for example `im(:, :, 3) = 0;`

- Set all pixels darker than some level to black (0) with `im(im < 128) = 0;`

- Set the intensity of some color to some specific value, for example `im(:, :, 2) = 64;`
- ...

If you do something that causes the array to lose its class "uint8", that can usually be fixed with the command `uint8(...)`.

26.2 Two-dimensional DFT

Since images are two-dimensional (considering either grayscale images, or working with each color channel separately), they can be processed using the two-dimensional version of Discrete Fourier Transform, implemented in Matlab as `fft2` and (inverse) `ifft2`. The formulas for 2D transform follow the same logic as one-dimensional formulas (25.3.2) and (25.3.3). Namely, the **2D Fourier coefficients** of a given matrix $(z_{k\ell})$ with indices $k = 0, \dots, m-1$, $\ell = 0, \dots, n-1$ are computed as:

$$c_{pq} = \sum_{k=0}^{m-1} \sum_{\ell=0}^{n-1} z_{k\ell} \exp(-2\pi ikp/m) \exp(-2\pi i\ell q/n) \tag{26.2.1}$$

The matrix is reconstructed from its coefficients by inverse transform

$$z_{k\ell} = \frac{1}{mn} \sum_{p=0}^{m-1} \sum_{q=0}^{n-1} c_{pq} \exp(2\pi ikp/m) \exp(2\pi i\ell q/n) \tag{26.2.2}$$

The fact that Matlab indices begin with 1 means they have to be shifted by 1 in the above formulas, but since we will just use `fft2` and (inverse) `ifft2`, we will not worry about this.

Let us try this with the grayscale version of the winter image, saved here as a PNG file for lossless compression.

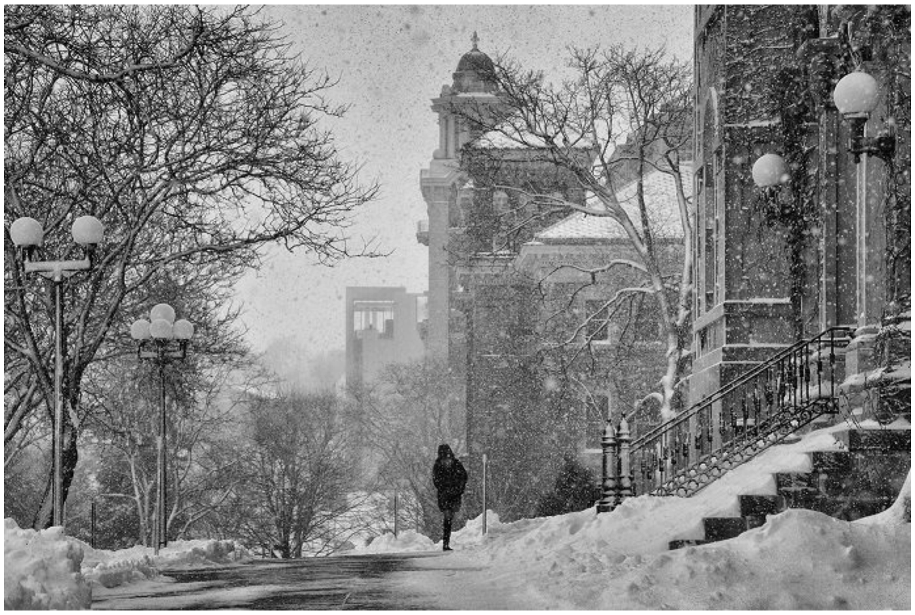

Figure 26.2.1 Grayscale PNG image

After loading the image with `im = imread('winter.png')`, check this is a matrix (no separate channels for colors). Try `fc = fft2(im);` which produces a matrix of the same size but with complex entries. It is not easy to understand all of it. Small pieces can be inspected with `fc(1:5, 1:5)` for example. A histogram can help visualize the distribution of their magnitudes: `hist(abs(fc(:)), 50)`. (Recall that `fc(:)` means flattening a matrix to a vector.) Looks like nearly all coefficients are much smaller than the largest ones.

We can set all small Fourier coefficients (less than some threshold `thr`) to zero using `fc(abs(fc) < thr) = 0;`. One way to determine the threshold reasonably is to use **percentiles**: for example,

```
thr = prctile(abs(fc(:)), 75);
```

returns the number `thr` such that 75 percent of Fourier coefficients are less than it, and the other are greater (in absolute value).

Example 26.2.2 Compress an image using Discrete Fourier Transform. Given a percentage `p`, such as 75, compress the "winter" image by setting `p` percent of its Fourier coefficients to zero.

Solution.

```
p = 75;
im = imread('winter.png');
fc = fft2(im);
thr = prctile(abs(fc(:)), p);
fc(abs(fc) < thr) = 0;
im2 = uint8(real(ifft2(fc)));
imshow(im2)
```

The inverse Fourier transform may have imaginary part due to the truncation of Fourier coefficients and rounding errors in computations. This imaginary part is removed with `real`. The result is converted to unsigned 8-bit integers with `uint8`. □

The JPEG compression algorithm is based on the Fourier Transform but differs from the above example in several ways:

- It divides the image into 8×8 pixel blocks and operates on each block separately.
- It uses Discrete Cosine Transform in which the role of complex exponentials $\exp(2\pi ik/m)$ is played by cosine functions $\cos(2\pi k/m)$. This improves the efficiency and eliminates the appearance of complex numbers.
- In addition to setting small Fourier coefficients to 0, it quantizes the other ones: rounds to the nearest integer, or nearest multiple of 5, etc.
- Since setting many elements to 0 does not by itself make a matrix smaller, the final step of the process is running a lossless compression algorithm which exploits the existence of many equal entries in the matrix.

26.3 Cosine interpolation of non-periodic functions

In Chapter 25 we worked with periodic functions, approximating them by trigonometric polynomials. Can we use trigonometric polynomials to approximate non-periodic functions? For example, suppose we solved the initial-value problem

$$y'(t) = \sin(y^2) + t, \quad y(0) = 1 \tag{26.3.1}$$

with

```
[t, y] = ode45(@(t, y) sin(y^2) + t, 0:0.1:3, 1);
```

Note that in contrast to Chapter 20 the second argument is given not as an interval (two-element vector) `[0 3]` but as a set of t-values at which the solution needs to be found. This ensures that we get y-values for these specific t-values. But now that we have them, how can we get a simple *approximate* solution?

- Polynomial interpolation? Hopeless, a degree 30 polynomial through 31 equally spaced points is not even worth trying.
- Cubic spline interpolation? It would make a nice plot but in terms of a formula, the spline would have 120 coefficients (4 for each of 30 subintervals). This is hardly a simple solution.
- Trigonometric polynomials? But they are periodic, and our function has different values at 0 and 3.

Compare the behavior of cosines $\cos kx$ and sines $\sin kx$ on the interval $[0, \pi]$. The cosines have different values at $0, \pi$ while the sines are always 0 at both ends. This makes cosines, specifically in the form $\cos(\pi kx/L)$, preferable for approximating a non-periodic function on $[0, L]$. The **Discrete Cosine Transform** is a tool used to find the coefficients of a given function in the cosine basis. It is somewhat technical (there are 4 different types of the transform), but the essence of the method can be stated simply: apply Fourier transform to the **even periodic extension** of the given function.

The discrete version of even periodic extension is as follows: given function values $y_1, \dots, y_n$, we extend them to $y_1, \dots, y_n, y_{n-1}, \dots, y_2$ (for the total of $2n-2$ values). In Matlab terms, the extended vector is

```
ye = [y y(end-1:-1:2)];
```

Why not simply `[y y(end:-1:1)]`? Repeating the endpoint values y_1, y_n would distort the picture of this function. They should appear once while the rest appear twice; this can be viewed as yet another form of "trapezoidal adjustment".

Having found the Fourier coefficients `c = fft(ye)/numel(ye)` we observe they are all real (any imaginary component is due to round-off errors). This makes it possible to "fold" the sum of complex exponentials into a cosine sum: for $k = 1, \dots, n-2$ the pair of terms

$$c_k \exp(ikx) + c_{2n-k} \exp(i(2n-k)x)$$

is aliased to

$$c_k \exp(ikx) + c_{2n-k} \exp(i(-k)x) = 2c_k \cos(kx)$$

(here x is a multiple of π/n.) The terms c_0 and c_{n-1} do not have a counterpart among the coefficients $c_0, \dots, c_{2n-2}$, so the cosine sum ends up being

$$c_0 + 2\sum_{k=1}^{n-2} c_k \cos(\pi kx/L) + c_{n-1}\cos(\pi(n-1)x/L) \tag{26.3.2}$$

And of course, Matlab indexing of the coefficients begins with 1, forcing another adjustment: see the following example.

Example 26.3.1 Interpolate with cosines. Apply the cosine interpolation to the numerical solution of ODE (26.3.1).

Solution. Since `ode45` returns `y` as a column, it is transposed in the code below. Then we extend this vector to `ye`, take its Fourier Transform `c`, and fold it into cosine coefficients `cf`. Then plot the resulting sum of cosines.

```
[t, y] = ode45(@(t, y) sin(y^2) + t, 0:0.1:3, 1);
y = y';

n = numel(y);
ye = [y y(end-1:-1:2)];
c = real(fft(ye))/numel(ye);
cf = [c(1) 2*c(2:n-1) c(n)];

L = 3;
x = linspace(0, L, 1000);
k = 0:n-1;
g = cf*cos(k'*x*pi/L);
plot(x, g, t, y, 'r*')
```

□

The cosine sum in Example 26.3.1 has 31 terms, as many as there are data points. But many of them are very close to zero, as `plot(cf)` shows. For example, we could keep the first `m=8` of them and still have a good approximation for this function.

```
m = 8;
k = 0:m-1;
g = cf(1:m)*cos(k'*x*pi/L);
```

This is essentially a one-dimensional version of image compression discussed in Section 26.2.

26.4 Homework

1. Instead of deciding in advance how many coefficients to discard, it is usually preferable to maintain some degree of quality of the image. This is often measured in terms of "energy", defined as the sum of squares of absolute values of Fourier coefficients. That, is to have 95% quality, we want to keep enough Fourier coefficients so that the sum of their squares is at least 95% of the total sum of squares. Equivalently, we discard the coefficients that contribute 5% of the energy.

The goal of this exercise is to modify Example 26.2.2 so that the input value is quality `q` such as 95 or 98, and the image in Figure 26.2.1 is compressed based on the energy consideration.

Suggestions: sort the Fourier coefficients from smallest to largest using

```
sorted = sort(abs(fc(:)));
```

The total energy is

```
total = sum(sorted.^2);
```

The key point is to write a while loop that sums the squares of the elements of `sorted` array until they reach the desired proportion of the total energy. When the loop exits, you will know what threshold value `thr` should be used for truncating Fourier coefficients.

Compare the quality of compressed image with `q = 98`, `q = 95`, and `q = 90`.

Chapter 27

Linear Least Squares

We briefly encountered least-squares fit in Chapter 6 but this method deserves a closer look in regard to how we choose a model to fit a given data set. It is one thing to find a function that matches the data (interpolation), another to have a model that learns from the data and has predictive power. Interpolation uses about as many parameters (coefficients) as the number of data points; curve-fitting uses relatively few parameters, hence does not hit the points precisely. Using too many parameters results in overfitting: a model begins to memorize training data rather than learning to generalize from trend.

27.1 Overview of least squares

Least squares fit is about more than just fitting a curve to some points like Example 6.5.1. In its general form we have a vector $\mathbf{y}$ of n observations and p vectors of **explanatory variables**, say $\mathbf{v}_1, \dots, \mathbf{v}_p$. Our goal is to approximate $\mathbf{y}$ by a linear combination of $\mathbf{v}_k$, say $\sum_{k=1}^p \beta_k \mathbf{v}_k$. What are these vectors in the polynomial model?

If $\mathbf{z} = \sum_{k=1}^p \beta_k \mathbf{v}_k$ is the best prediction, then the difference $\mathbf{z} - \mathbf{y}$ is orthogonal to each $\mathbf{v}_k$. That is,

$$\mathbf{z} \cdot \mathbf{v}_k = \mathbf{y} \cdot \mathbf{v}_k \tag{27.1.1}$$

for each k. These are p linear equations for p parameters. Let us express them in matrix form. Let X be the $n \times p$ matrix formed by vectors $\mathbf{v}_1, \dots, \mathbf{v}_p$ as columns. Then $X^T\mathbf{y}$ is the column vector with entries $\mathbf{y} \cdot \mathbf{v}_k$, matching the right hand side of (27.1.1). The left hand side of (27.1.1) involves $z = X\beta$ where β is the vector of coefficients that we are looking for. With this, we recognize that $X^TX\beta$ is the vector whose entries match the left hand side. In conclusion, the matrix form of (27.1.1) is

$$X^TX\beta = X^T\mathbf{y} \tag{27.1.2}$$

which is known as the **normal equations** associated to the overdetermined system $X\beta = \mathbf{y}$. Note that X^TX is invertible if and only if the explanatory vectors are linearly independent (which they should be; otherwise we have redundant ones).

The goodness of fit could be measured in several ways. One indicator is the norm of the residual vector $\mathbf{y} - \mathbf{z}$, or better yet, the squared norm

$$|\mathbf{y} - \mathbf{z}| = \sum_{k=1}^n |y_k - z_k|^2 \tag{27.1.3}$$

However this quantity has the units of y_k squared, and a unit-free quantity is preferable. A popular approach is to compare the residual sum fo squares (27.1.3) to the total sum of squares

$$\sum_{k=1}^{n} |y_k - \bar{y}|^2, \quad \bar{y} = \frac{1}{n}\sum_{k=1}^{n} y_k \tag{27.1.4}$$

We want our model to predict y_k better than the baseline guess $\bar{y}$ (which does not use any explanatory variables). So, if (27.1.3) is much smaller than (27.1.4), the fit is good. The goodness-of-fit is then measured by the **coefficient of determination**

$$R^2 = 1 - \frac{\sum_{k=1}^{n} |y_k - z_k|^2}{\sum_{k=1}^{n} |y_k - \bar{y}|^2} \tag{27.1.5}$$

which shows how much of the variability in observation is predicted by the model. The value does not exceed 1, with 1 being exact fit and close to 0 being a rather useless model (example). The quantity R^2 may even be negative if the model does a worse job than the constant predictor $\bar{y}$. The notation R^2 comes from the fact that for linear regression $y = ax + b$ (without testing-training split), it is the square of correlation coefficient. But in general it is not a square of anything.

27.2 Overfitting, training and testing

Recall Example 6.5.1 in which we find a best-fitting parabola to given data. The code is extended below to allow arbitrary degree d of the polynomial, and to add the computation of R^2 according to (27.1.5). Note the use of column vectors below.

Example 27.2.1 Fitting a polynomial of any degree. Improve Example 6.5.1 to work with polynomials of any degree d, and add a computation of R^2 to it.

Solution.

```
x = (0:14)';
y = [9 8 5 5 3 1 2 5 4 7 8 8 7 7 8]';  % data
d = 2;                      % degree of polynomial
X = x.^(0:d);               % matrix of linear system for parameters
beta = X\y;                 % optimal parameters
f = @(x) (x.^(0:d))*beta;   % best-fitting function f

t = linspace(min(x), max(x), 1000)';
plot(t, f(t), 'b', x, y, 'r*')

total = norm(y - mean(y))^2;
residual = norm(y - f(x))^2;
fprintf('R^2 for degree %d is %g\n', d, 1 - residual/total);
```

□

If we increase n toward 7, the curve gets closer to the points but it also becomes more wiggly. Most importantly, the prediction it makes for $x = 10$, that is $f(10)$, becomes less realistic. This is **overfitting**, which describes the situation in which our model f mostly memorizes the data instead of learning from it. (Food for thought from xkcd)

A simple but effective way to combad overfitting is to split the data into two parts: one (training set) is used to compute the coefficients β_k, and then another part (testing set) is used to evaluate goodness of fit. The split may be done randomly, perhaps 50:50 or 60:40 proportion. Which of the following 50:50 splits looks better?

```
x_train = x(1:8);
x_test = x(9:end);
```

(and similarly for y) or

```
x_train = x(1:2:end);
x_test = x(2:2:end);
```

Example 27.2.2 Training-testing split. Use the training-testing split in Example 27.2.1.

Solution.

```
x = (0:14)';
y = [9 8 5 5 3 1 2 5 4 7 8 8 7 7 8]';
x_train = x(1:2:end);
y_train = y(1:2:end);
x_test = x(2:2:end);
y_test = y(2:2:end);

d = 2;

X = x_train.^(0:d);
beta = X\y_train;
f = @(x) (x.^(0:d))*beta;

t = linspace(min(x), max(x), 1000)';
plot(t, f(t), 'b', x, y, 'r*')

total = norm(y_test - mean(y_test))^2;
residual = norm(y_test - f(x_test))^2;
fprintf('R^2 for degree %d is %g\n', d, 1 - residual/total);
```

□

An additional detail: when one reports the model performance (not in this class, but perhaps for a publication), the reported value of R^2 or other estimate of goodness-of-fit should be computed on a separate **validation set** which was not used for either training or testing. Using the testing set for reporting R^2 introduces a subtle source of bias: if *many* models are tested, and one of them wins, it is more likely than not that the winning model "had better luck" with the specific data set used for testing, and that would lead us to overestimate its goodness-of-fit. (Here is an example of testing too many models and not validating.)

27.3 The standard error of parameters

A more statistics-oriented approach to model selection is to examine the randomness of the parameters β_k we found: we study the hypothesis that the "true value" of a particular parameter β_k is zero, meaning that the value we found for it is nonzero only due to chance. This approach does not really say whether a model is good for describing the data, but whether a particular parameter actually helps in it.

With this approach, we think of $\mathbf{y} - \mathbf{z}$ as a vector of n observations of a random variables with $n - p$ degrees of freedom (where p is the number of parameters). The variance of such a random variable is estimated as

$$s^2 = \frac{1}{n-p} \sum_{k=1}^{n} (y_k - z_k)^2 \tag{27.3.1}$$

You may have seen this formula with $p = 1$ which corresponds to the constant model $z_k \equiv \bar{y}$.

Solving for optimal parameters (theoretically) involves inverting the matrix $X^T X$, seen in (27.1.2). The matrix $s^2(X^T X)^{-1}$ can be considered the **covariance matrix** of estimated parameters. Specifically, its diagonal element in position (k, k) is the variance of β_k. The square root of variance gives the **standard error** of β_k. As a rule of thumb, if $|\beta_k| < 2\,\mathrm{SE}(\beta_k)$ (a parameter is within two standard errors of 0), this parameter should be considered inessential and possibly removed from the model. The following computation implements this statistical analysis.

```
s2 = norm(y-z)^2/(n-p);      % variance of error
C = s2*inv(X'*X);            % covariance matrix for parameters
ts = beta./sqrt(diag(C));    % the t-statistics of the parameters
```

The parameters with $|t| < 2$ should be considered for removal. In the example of fitting polynomials of degree d, we have $p = d + 1$ parameters. The consideration should be focused on the leading coefficient, because if it appears to be inessential, we should remove it by decreasing the degree d by 1. Note that the removal of one parameter causes all others to be recalculated, so the rest may become essential now.

Example 27.3.1 Computing the t-statistics of parameters. Modify Example 27.2.1 to include the computation of t-statistics for each coefficient of the polynomial.

Solution. Note that the vector of t-statistics, which is called `ts` below, contains the statistics for the coefficients of $x^0, \dots, x^d$ in this order. This means that $ts(end)$ is really the value of most interest here.

```
x = (0:14)';
y = [9 8 5 5 3 1 2 5 4 7 8 8 7 7 8]';
d = 2;
X = x.^(0:d);
beta = X\y;
f = @(x) (x.^(0:d))*beta;

[n, p] = size(X);
s2 = norm(y-f(x))^2/(n-p);
C = s2*inv(X'*X);
ts = beta./sqrt(diag(C));
disp(ts)
```

□

27.4 Multiple regression

Least squares method, as outlined at the beginning of Section 27.1, includes **multiple linear regression**, in which we model a dependent variable by a linear combination of several independent variables. In this form, the matrix X consists not of various powers of one variable, but of several explanatory variables. In the following example the observations modeled are the scores on a final exam, while explanatory variables are Homework average, Quiz average, Exam 1 score, Exam 2, and Exam 3. After fitting a linear combination of these five variable to the final exam score, the code calculates R^2 and t-statistics.

```
hw = [97 99 90 76 96 80 100 55 69 85 89 100 73 89 95 98 98 73 100 84]';
quiz = [94 104 101 76 102 83 90 81 101 101 105 99 91 98 95 101 88 77 97 95]';
exam1 = [96 100 93 79 95 62 83 88 86 96 95 96 74 91 88 94 96 98 97 94]';
exam2 = [71 80 59 31 70 26 63 65 69 78 79 65 38 75 71 75 68 64 73 68]';
exam3 = [83 86 65 42 82 55 69 38 70 77 65 80 50 79 70 88 75 75 73 56]';
y = [82 92 74 57 77 56 51 42 79 92 89 79 49 84 88 84 71 88 83 89]';      % final exam

X = [hw quiz exam1 exam2 exam3];
beta = X\y;
z = X*beta;      % predicted final scores

total = norm(y - mean(y))^2;
residual = norm(y - z)^2;
disp('Parameters:')
disp(beta');
fprintf('R^2 is %g\n', 1 - residual/total);

[n, p] = size(X);   % X is a matrix of size n by p
s2 = norm(y-z)^2/(n-p);
C = s2*inv(X'*X);
ts = beta./sqrt(diag(C));
disp('t-statistic values');
disp(ts')
```

How to intepret these results? Can the model be improved by discarding some explanatory variables which fail to explain anything?

27.5 Homework

1. Modify Example 27.2.2 so that it runs a loop `for n=1:7` and within that loop, R^2 is computed for each n. The code should then print the optimal value of n, its R^2 value, and plot the graph for optimal model.

2. The number of active Covid cases at SU during October 1 - October 23 is given below.

```
y = [5 5 4 5 9 25 45 65 77 87 87 89 97 101 90 74 57 26 20 13 12 12 13]';
```

(The x-values can be `x = (1:23)'`.) Use the t-statistic for the leading coefficient as in Example 27.3.1, determing what degree of a polynomial should be used to model this data. Also plot both the optimal polynomial and the data points.

Chapter 28

Transforming Data for LLS

The method of previous chapter works only for models in which parameters appear in a linear way, as coefficients to predetermined functions. This is often insufficient: for example, exponential and logistic models contain some parameters nonlinearly. In this chapter we expand the reach of Linear Least Squares method by transforming the variables involved in the problem, or by applying the method to the coefficients of an implicit function.

28.1 Exponential fit

Exponential growth is modeled by a function of the form $f(x) = ae^{bt}$. Suppose we have some data (x_k, y_k), $k = 1, \dots, n$, and think that it describes exponential growth (which naturally occurs in ecology and epidemiology). A direct attempt to minimize the sum of differences squared:

$$\sum_{k=1}^{n} \left(y_k - ae^{bx_k}\right)^2 \to \min \tag{28.1.1}$$

is possible but requires optimization methods which we will consider later. There is an alternative approach: instead of fitting ae^{bx} to the values y_k, fit its logarithm, $bx + \log a$, to the values $\log y_k$. This means we will minimize the sum of squares of differences of (natural) logarithms:

$$\sum_{k=1}^{n} (\log(y_k) - bx_k - \log a)^2 \to \min \tag{28.1.2}$$

It is important to understand the two problems (28.1.1) and (28.1.2) are not equivalent. If exact fit is possible, either approach will find it. But in general, some deviations (nonzero residuals) are inevitable, and the two approaches penalize deviations in different ways.

For example, if the given y-values are `[1 2 10]` and we have two competing exponential functions: one predicts `[0.5 2 10]` while the other predicts `[1 2 11]`. Using the penalty function (28.1.1), the first model has the penalty 0.25 while the second has penalty 1, so the first looks better. But if we look at the differences of logarithms, then the first has penality $(\log 1 - \log 0.5)^2 = 0.48$ while the second has penalty $(\log 10 - \log 11)^2 = 0.01$, so the second looks much better.

Most importantly, the minimization problem (28.1.2) can be easily solved using the method of Chapter 27. For notational convenience let $\beta_1 = \log a$ and

$\beta_2 = b$. If the left hand side of (28.1.2) was zero, that would mean that β_1, β_2 is a solution of the overdetermined linear system

$$\beta_1 + \beta_2 x_k = \log y_k, \quad k = 1, \dots, n \tag{28.1.3}$$

The least squares solution of (28.1.3) is the vector $\beta = (\beta_1, \beta_2)$ that minimizes (28.1.2).

Example 28.1.1 Fit an exponential to Covid data. The following are new Covid infections in France as reported on Fridays in September-October (from Sep 4 to Oct 23).

```
y = [6011 7742 9335 12048 10946 14618 20399 29472]';
```

Fit an exponential function to this data, using week numbers `1:8` as x-values. Extend it two weeks into the future by plotting it up to $x = 10$. Include the original data on the plot.

Solution. The vector `y` is already defined above, but we need `yt = log(y)` to get transformed y-values.

```
x = (1:8)';
yt = log(y);
X = x.^(0:1);
beta = X\yt;
f = @(x) exp((x.^(0:1))*beta);

t = linspace(1, 10, 1000)';
plot(t, f(t), 'b', x, y, 'r*')
```

□

28.2 Logistic fit

If we simplify the SIR model (Section 20.1) by removing the recovering process, the proportion of infected people in the population grows according to the ODE $y' = cy(1-y)$ with some coefficient c. This ODE has an explicit solution

$$y = \frac{1}{1 + ae^{-kt}} \tag{28.2.1}$$

with $a, k > 0$ depending on coefficient c (which we usually don't know in practice) and the initial condition $y(0)$ (which we might not know either). But if we can collect some data, we can fit a logistic function (28.2.1) to it. As written, the model includes parameters a, k in a nonlinear way. But applying the "logit" transformation $\log(y/(1-y))$ we obtain

$$\log \frac{y}{1-y} = kt - \log a \tag{28.2.2}$$

Thus, we can fit a linear function to the values $\log(y_k/(1-y_k))$ and then apply the inverse of the transformation $z = \log(y/(1-y))$, which is $y = 1/(1+e^{-z})$.

The logistic model can reasonable apply to other processes which are slow initially, speed up, and then slow down again. The following example, where "population" consists of Covid-positive students, models the process of their recovery.

Example 28.2.1 Fit a logistic function to Covid data. There were at least 121 reported Covid infections among SU students between October 8 and October 26. The following are the cumulative counts of people recovered, day by day, starting with October 9.

```
y = [1 1 1 5 8 10 27 44 62 93 101 110 112 115 116 117 118 120]';
```

Consider these as proportions of the total "population" of 121 and fit a logistic function to these proportions.

Solution. The vector `y` is already defined above, but we need to compute proportions `p = y/121` and transform them by `yt = log(p./(1-p))`. Then the usual linear regression is applied.

```
x = (1:numel(y))';
p = y/121;
yt = log(p./(1-p));
X = x.^(0:1);
beta = X\yt;
f = @(x) 1 ./ (1 + exp(-(x.^(0:1))*beta));

t = linspace(min(x), max(x), 1000)';
plot(t, f(t), 'b', x, p, 'r*')
```

□

The outcome of Example 28.2.1 is not entirely satisfactory. The logit transformation emphasizes the values near 0 and 1, and de-emphasizes those in between. We will revisit logistic fit later.

28.3 Fitting an ellipse to data points

Not all data points represent a function of the form $y = f(x)$. They may happen to accumulate along some closed curve such as an ellipse. General implicit equation of an ellipse is

$$Ax^2 + Bxy + Cy^2 + Dx + Ey = 1 \tag{28.3.1}$$

There are five unknown coefficients here. So, given a set of points (x_k, y_k), $k = 1, \dots, n$, we get a linear system by plugging each point into (28.3.1). Its least squares solution describes an ellipse that fits the data best in the sense of having the smallest sum

$$\sum_{k=1}^{n} (Ax_k^2 + Bx_k y_k + Cy_k^2 + Dx_k + Ey_k - 1)^2$$

which should hopefully fit the ellipse.

Example 28.3.1 Fit an ellipse to given points. Plot an ellipse that fits the given 7 data points:

```
x = [1 2 4 5 4 3 1]';
y = [2 0 -1 1 2 3 4]';
```

Then plot it together with the points, using `fimplicit` command. Its syntax is

```
fimplicit(@(x, y) ..., [xmin xmax ymin ymax])
```

where the first argument is the implicit equation (something equated to 0), and the second is the plot window.

Solution. The vector `y` is already defined above, but we need to compute proportions `p = y/133` and transform them by `yt = log(p/(1-p))`. Then the usual linear regression is applied.

```
X = [x.^2 x.*y y.^2 x y];
b = X\ones(7, 1);
hold on
plot(x, y, 'r*')
fun = @(x, y) b(1)*x.^2 + b(2)*x.*y + b(3)*y.^2 + b(4)*x + b(5)*y - 1;
win = [min(x)-1 max(x)+1 min(y)-1 max(y)+1];
fimplicit(fun, win)
hold off
```

Here the vector of parameters is called `b` instead of `beta` to shorten the implicit formula. It is obtained by solving a system with 1 on the right hand side, according to (28.3.1). The plot window is calculated from the data to allow margins of 1 unit on all sides.

Note that while it's safe to leave spaces around + signs in the formula `fun` (some find it improves readability), one has to be extra careful with such spaces within a vector. Indeed,

```
[min(x) -1 max(x) +1 min(y) -1 max(y) +1]
```

would be a vector of 8 numbers, not 4. Matlab does understand

```
[min(x) - 1 max(x) + 1 min(y) - 1 max(y) + 1];
```

as a vector of 4 numbers, however. □

Question 28.3.2 Vectorizing a function for implicit plot. It is tempting to write the anonymous function as `fun = @(x, y) [x.^2 x.*y y.^2 x y]*b - 1` which is shorter and consistent with how we used parameters in the past. And this works but Matlab complains: "Warning: Function behaves unexpectedly on array inputs. To improve performance, properly vectorize your function to return an output with the same size and shape as the input arguments." What is it complaining about? □

28.4 Homework

1. A **power law** is a function of the form $y = ax^p$. By taking logarithm on both sides we get $\log y = p \log x + \log a$, which is a linear function of $\log x$. So, to fit a power law to given data (x_k, y_k), we apply the logarithm to both x_k and y_k, and follow the linear regression process. Try this with France Covid data Example 28.1.1. Does the power law fit better or worse than exponential? (Compare visually, on the basis of plots.)

2. The number of active Covid cases at SU during October 1 - October 23 is given below.

```
y = [5 5 4 5 9 25 45 65 77 87 87 89 97 101 90 74 57 26 20 13 12 12 13]';
```

The x-values can be `x = (1:23)'`. Try to fit a "bell-shaped" Gaussian function $y = \exp(Ax^2 + Bx + C)$ (where $A < 0$) to the data. This can be done by the logarithmic transform `yt = log(y)` followed by fitting a quadratic function to transformed data in the usual way, via the matrix `X = x.^(0:2)`. Having found the parameters `beta`, we get a suitable function `f = @(x) exp(x.^(0:2)*beta)` which should be plotted together with the data points.

3. As an alternative to Exercise 28.4.2, one may try to model the same data by a rational function of the form $y = 1/(Ax^2 + Bx + C)$. (Indeed, the graph of function $y = 1/(x^2 + 1)$ also looks bell-shaped.) Try following the process of Exercise 28.4.2 by transforming data `yt = 1./y`, fitting a quadratic function, and plotting the reciprocal of that function. What goes wrong?

Then try the following fix to the issue encountered in the previous paragraph. Following the logic of Example 28.3.1, rewrite the equation $y = 1/(Ax^2 + Bx + C)$ as $Ax^2y + Bxy + Cy = 1$ and find the optimal A, B, C using least squares. Unlike Example 28.3.1, you do not need `fimplicit` to plot this function since the function $y = 1/(Ax^2 + Bx + C)$ can be plotted directly (note that you will need `1./(...)` rather than `1/(...)` to compute the reciprocals in a vectorized way.) Plot the function together with the data points. Does it fit better or worse than the Gaussian function in Exercise 28.4.2?

Chapter 29

Nonlinear Least Squares

Transforming data to make the Linear Least Squares work has limited applicability. Most nonlinear models cannot be transformed in this way, and even if the transformation is possible, it sometimes results in poorly fitting curves. We should consider the possibility of working directly with parameters that lead to nonlinear problems.

29.1 Motivating examples for Nonlinear Least Squares

Suppose we observe population growth over time, and expect that it should be governed by a differential equation of the form

$$y' = ry(1 - y/M) \tag{29.1.1}$$

which means the growth, initially exponential, is eventually constrained by the **carrying capacity** K. The general solution of (29.1.1) is

$$y = \frac{M}{1 + Ae^{-rx}} \tag{29.1.2}$$

where A depends on the initial condition. In practice, we do not know the values of A, M, r but we can try to estimate them by fitting a function of type (29.1.2) to observations (x_k, y_k), $k = 1, \dots, n$. It seems impossible to transform the data in a way that will make all three parameters linear (logit transformation works only if M is known).

Recall that *least squares* method means minimizing the sum of squares of residuals, which is a function of parameters A, M, r:

$$S(A, M, r) = \sum_{k=1}^{n} \left(y_k - \frac{M}{1 + Ae^{-rx_k}} \right)^2 \tag{29.1.3}$$

Since S is a smooth function, we could theoretically try the multivariable calculus approach: finds its partial derivatives of first order, equate them to zero, and solve the resulting system of equations:

$$\frac{\partial S}{\partial A} = 0, \quad \frac{\partial S}{\partial M} = 0, \quad \frac{\partial S}{\partial r} = 0 \tag{29.1.4}$$

But this system will be *nonlinear* and is unlikely to have a symbolic solution. Should we try some multivariable root-finding method such as Newton's method?

In practice, trying to minimize a function by solving (29.1.4) is not the most productive approach: there are probably many critical points, some of them saddle points, and some of them may be points of local maximum rather than minimum. It is better to search for a minimum directly, which is a problem of **multivariable optimization**. We will study optimization in the last part of this course. For now, we'll use the built-in Matlab tool `fminsearch`, the underlying principle of which (the Nelder-Mead method) will be considered later.

29.2 Using fminsearch for curve-fitting

The syntax of `fminsearch` is similar to `fsolve` (which searchers for solutions $f = 0$): the first argument is the function to be minimized, the second is initial point from which to start the search. For example,

```
fminsearch(@(x) x^2 + x, 0)
```

returns `-0.5` which is where the function is minimal. The choice of initial point does not matter here, since the minimum is unique. But for the second example

```
fminsearch(@(x) x^4 - x^2, 0)
```

the initial point matters: the function may find the minimum 0.7071 or -0.7071. Note that `fminsearch` is not misled by $x = 0$ which is also a critical point: it can tell a difference between minimum and maximum.

As most numerical methods, `fminsearch` may fail at its task:

```
fminsearch(@(x) exp(x), 0)
```

returns "-786.4315" (there are no points of minimum, of course), while

```
fminsearch(@(x) x^3 - x, 4)
```

overlooks the local minimum at $x = 1/\sqrt{3}$ and diverges to negative infinity. The success is much more likely if the initial point is chosen with some care. In case of fitting a curve to data, one should try to guess the signs and approximate sizes of the parameters.

Since our models generally have more than one parameter, we need multi-variable `fminsearch`. For example,

```
fminsearch(@(b) b(1)^2 + b(2)^4 + 3*b(1) - b(2), [1; 2])
```

finds the minimum of $b_1^2 + b_2^4 + 3b_1 - b_2$ (using letter b to shorten the formula, as usually our parameters are called $\beta_1, \beta_2, \dots$.) Note that the function being minimized still has only one argument, but this argument is now a vector.

Example 29.2.1 Fit a logistic function to Covid data. Fit a general logistic curve to the data in Example 28.2.1 without variable transformations.

Solution. Once we have the data `x, y`, we can set up the sum-of-squares function `S` according to (29.1.3). The parameters are to be represented by the components `b(1), b(2), b(3)` in some order (I use the alphabetical order A, M, r). The parameters should be positive, so perhaps `[1; 1; 1]` can be our initial point. The orientation of data vectors `x, y` does not matter here since we no longer make a linear system out of them.

```
y = [1 1 1 5 8 10 27 44 62 93 101 110 112 115 116 117 118 120];
x = 1:numel(y);

f = @(x, b) b(2)./(1+b(1)*exp(-b(3)*x));                % the model equation
opt = fminsearch(@(b) sum((y - f(x, b)).^2), [1; 1; 1]);  % optimal b

t = linspace(min(x), max(x), 1000)';
plot(t, f(t, opt), 'b', x, y, 'r*')
```

The two middle lines describe the logic of the method: set up a model as a function of explanatory variable(s) `x` and parameter(s) `b`, then minimize the sum of squares of the deviations `y - f(x, b)`. Finally, the optimal (?) values of parameter `b` are used to plot the best-fitting curve.

The above code fails, as is often the case with multivariable optimization. We should think of a better way to choose the initial point than just `[1; 1; 1]`. For example, the second parameter is the carrying capacity, and this is clearly at least as large as 120. An initial point like `[1; 100; 1]` does the job. Note that the fit is much better than what we achieved in Example 28.2.1. □

29.3 Beyond curve-fitting: source location

Suppose someone took water samples at various points (x_k, y_k) and measured the concentration of some pollutant (microplastics, oil, etc) at each point. Let's use z_k for these amounts. A question of interest is the **source** of this pollution.

The theory of partial differential equations tells us that if a substance diffuses from a point source at (u, v) under somewhat idealized conditions (ignoring currents, decay, etc) then its concentration after some time will be expressed by the two-dimensional Gaussian function

$$z = A\exp\left(-C((x-u)^2 + (y-v)^2)\right) \tag{29.3.1}$$

Here A, C are the shape parameters of the Gaussian (describing how tall and wide it is) and (u, v) are its location parameters (where it is centered). We are really interested in the latter, but all four will need to be determined by the nonlinear least squares process.

One can visualize the above as surface-fitting: given the points (x_k, y_k, z_k) we seek to fit a surface of type (29.3.1) to them, with the ultimate goal of locating the center of that Gaussian. Suppose the data is as follows.

```
x = [1 3 5 1 2 4 6 1 2 5 7];
y = [1 2 1 3 2 4 3 4 4 5 3];
z = [3 9 8 4 6 9 8 3 6 6 5];
```

Example 29.3.1 Find the source of pollution. Fit a two-dimensional Gaussian to the above data and report the expected location of the source.

Solution. The problem is solved in three lines: model function, optimization, output. The components of vector `b` below represent the parameters A, C, u, v in this order.

```
f = @(x, y, b) b(1)*exp(-b(2)*((x-b(3)).^2 + (y-b(4)).^2));
opt = fminsearch(@(b) sum((z - f(x, y, b)).^2), [1; 1; 1; 1]);
fprintf('The source is located near (%g, %g) \n', opt(3:4))
```

□

29.4 A closer look at the logistic curve

This section takes another look at the logistic function (29.1.2) in preparation for the homework assignment. The meaning of parameter A is not very intuitive, which motivates the following *equivalent form* of the function:

$$y = \frac{M}{1+e^{-r(x-a)}} \tag{29.4.1}$$

(so, $A = e^{ra}$.) Note that this curve is obtained from the **standard logistic function**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

with

- vertical stretching by factor M;
- shift to the right by a units;
- horizontal compression by factor r.

(Reference and plot.)

Since the standard logistic function rises from 0 to 1 and has maximal slope when $x = 0$, we can relate parameters a, M, r directly to shape of the curve:

- a is the x-value at which the curve has the steepest slope;
- M is the amount by which the curve rises;
- r measures its steepness; the curve is essentially flat outside of the interval $[a - 5/r, a + 5/r]$.

29.5 Homework

1. Consider the count of world-wide Covid cases (source):

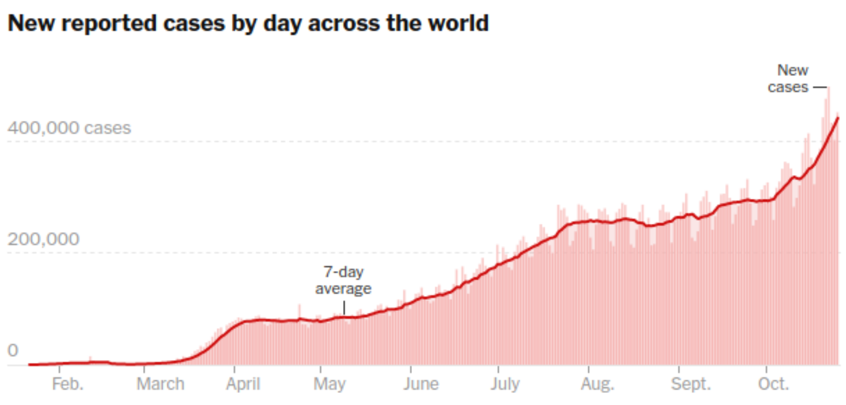

Figure 29.5.1 New reported cases

The graph shows three stages of accelerated growth: March-April, June-July, and October; the third one is ongoing. Since the total reflects the spread of disease in different regions, it is natural that it looks like the sum of different logistic curves.

Accordingly, we should model this by the sum of three logistic curves. To reduce the amount of data, we use the 7-day averages reported on the 1st, 11th, and 21th of every month, from March 1 to October 21. These 24 data values (measured in thousands) are below.

```
y = [1 4 21 67 80 79 77 85 94 110 126 148 180 205 230 257 259 254 265 267 292 295 337 389];
```

The corresponding x-values are simply `x = 1:numel(y)`. Run `plot(x, y, '*')` to visualize the data: this will help with choosing initial parameters, as described below.

High-dimensional optimization is difficult, and `fminsearch` will need your help in the form of a good initial point (`ones(9, 1)` is not good enough). The logistic equation (29.4.1) leads to the 9-parameter model

$$y = \frac{M_1}{1+e^{-r_1(x-a_1)}} + \frac{M_2}{1+e^{-r_2(x-a_2)}} + \frac{M_3}{1+e^{-r_3(x-a_3)}} \tag{29.5.1}$$

Reasonable initial guesses would be:

- a_1, a_2, a_3 are the x-values of places where the three "waves" are the steepest
- M_1, M_2, M_3 are the sizes of the three waves
- r_1, r_2, r_3 could be set to 1

Follow the examples such as Example 29.2.1 in plotting the best-fitting curve together with the data. Since the main interest in building a model like this is to predict future development, extend the curve by two months into the future (this means 6 units on x-axis). What does it predict for December 21?

Part V

Optimization

Chapter 30

Single Variable Minimization

The problem of optimization is to find, approximately, a minimum (preferably global) of a given function (on a certain set). There are some parallels with root-finding methods of Part II (related to the gradient being zero at the critical points) but the subjects turn out to be different. We begin the study of optimization with minimization of a given function on an interval.

30.1 Basic concepts of optimization

Optimization may be naturally stated as search for maximum (for example, maximum profit) or minimum (minimum cost). But since every point of maximum of a function f is also a point of minimum of $-f$, it is enough to study the minimization problem.

Suppose $f\colon E \to \mathbb{R}$ is a real-valued function on some set $E \subset \mathbb{R}^d$. The function attains its **global minimum** at $p \in E$ if $f(x) \geq f(p)$ for all $x \in E$. It is possible to have several points of global minimum, for example $f(x) = \sin x$ on the interval $[0, 4\pi]$ attains global minimum of -1 at the points $x = 3\pi/2$ and $x = 7\pi/2$.

The **Extreme Value Theorem** states that if f is a continuous function and the set E is closed (includes all its boundary points) and bounded (is contained in some cube), then there is a point $p \in E$ at which f attains its global minimum.

A function f attains its **local minimum** at $p \in E$ if there exists $r > 0$ such that $f(x) \geq f(p)$ for all $x \in E$ such that $|x - r| < r$. That is, there exists a neighborhood of point p such that the smallest value of f in that neighborhood is $f(p)$.

A major source of difficulty of optimization is searching for global minimum and finding only a local one. There is no universal recipe for avoiding this problem. Consider the function $f(x) = \sin x + \sin \sqrt{2}x$ on some long interval such as $[-100, 100]$. Where is its global minimum? There is no reliable and efficient way to find it.

A function that has only one minimum on some set is called **unimodal**. For example, $f(x) = x^2 - x$ is unimodal on the interval $[0, 2]$ but $f(x) = x - x^2$ is not (according to our definition; when people focus on maximum instead of minimum, the situation is reversed). Working with unimodal functions is much easier because our algorithm is much more likely to find minimum.

A function is **convex** (also called "concave up" in calculus) if its graph lies below its secant lines, meaning

$$f((1-t)a+tb) \le tf(a)+(1-t)f(b), \quad 0<t<1 \tag{30.1.1}$$

A function is **strictly convex** if the inequality (30.1.1) is strict. Such a function is unimodal on any interval. (Can you find a unimodal function that is not convex?)

Also, f is called **concave** (in calculus, "concave down") if the sign of inequality in (30.1.1) is reversed.

30.2 Brute force search

This is a very direct approach to minimization of $f\colon [a,b] \to \mathbb{R}$: choose some large number of points $x_1, \dots, x_n$ on the interval, equally spaced, evaluate f at each point, take the minimum of $f(x_k)$. How close will this be to the actual global minimum? If f is differentiable and $|f'| \le M$ on the interval, the mean value theorem implies

$$|f(x)-f(y)| \le M|x-y| \tag{30.2.1}$$

for all x, y. Placing n points on the interval, we have intervals of size $h = (b-a)/(n-1)$ between them. Hence, each point of the interval is within distance $h/2$ of some grid point x_k. Therefore,

$$\min_k f(x_k) - \frac{Mh}{2} \le \min_{a\le x\le b} f(x) \le \min_k f(x_k) \tag{30.2.2}$$

This may be good enough for a rough estimate of global minimum. One can also refine this estimate by applying a more advanced method on the interval $[x_{k-1}, x_{k+1}]$ around the point x_k where $\min_k f(x_k)$ is attained.

In Matlab, the code for brute force search may look like

```
x = linspace(a, b, n);
y = f(x);
[fm, ind] = min(y);
```

Note the two-outputs form of `min` command: when used in this way, the first output is the value of minimum, and the second is the index at which it was found. So, `y(ind)` is the same as `fm` but more importantly, `x(ind)` is the corresponding x-value.

Example 30.2.1 Using brute force search on an interval. Minimize $f(x) = \sin(x)+\sin(\sqrt{2}x)$ on the interval $[0, 100]$ using $n = 100000$ points. Then use (30.2.2) to estimate the true global minimum of f on this interval.

Solution.

```
f = @(x) sin(x) + sin(sqrt(2)*x);
a = 0;
b = 100;
n = 100000;   % number of points
x = linspace(a, b, n);
y = f(x);
[fm, ind] = min(y);
fprintf('Minimum %g attained near %g \n', fm, x(ind));
```

The output is "Minimum -1.99306 attained near 29.9413". To estimate the accurac, note that $|f'(x)| \le 1+\sqrt{2} \approx 2.4$ everywhere, and that $h = (100-0)/(10^6-1) \approx 10^{-4}$. So $Mh/2 \approx 0.00012$ and therefore, the true global minimum of f lies between $-1.99306 - Mh/2 = -1.99218$ and -1.99306. □

Brute force search requires many function evaluations, especially with several parameters. It is not practical as the primary search method, but may be useful in combination with other methods.

30.3 Newton method for minimization

If we can find the derivative f' (or gradient ∇f in higher dimensions), it may be possible to apply one of root-finding methods of Part II to it. For example, Newton's method was one of those. But since we are solving $f' = 0$, the method will involve the second derivative of f: it amounts to iteration of the function

$$g(x) = x - \frac{f'(x)}{f''(x)} \tag{30.3.1}$$

Let's try this out on a simple example.

Example 30.3.1 Using Newton method for minimization. Minimize $f(x) = x^4 - 3x^2 + x + 1$ using the Newton method with initial point $x_0 = 0$. How does the result depend on the initial point?

Solution. First compute the derivatives $f'(x) = 4x^3 - 6x + 1$ and $f''(x) = 12x^2 - 6$. Then iterate the function g from (30.3.1) as in Example 8.2.1. Most of the code is copied from there.

```
f = @(x) x.^4 - 3*x.^2 + x + 1;
fp = @(x) 4*x.^3 - 6*x + 1;
fpp = @(x) 12*x.^2 - 6;

g = @(x) x - fp(x)/fpp(x);
x0 = 0;
max_tries = 1000;
for k = 1:max_tries
    x1 = g(x0);
    if abs(x1-x0) < 100*eps(x0)
        break
    end
    x0 = x1;
end
if k < max_tries
    fprintf('Found x = %g with f(x) = %g after %d steps\n', x1, f(x1), k);
else
    disp('Failed to converge')
end
```

The result is obtained quickly: "Found x = 0.169938 with f(x) = 1.08414 after 5 steps". But is this a minimum? Try

```
t = linspace(-2, 2, 1000);
plot(t, f(t))
```

to check. Also, note that looking at a graph like this is essentially a human-assisted brute-force search, since the function was evaluated on a uniform grid through the interval $[-2, 2]$.

Do we get a better result with a different starting point? □

The issue demonstrated by Example 30.3.1 is that applying Newton's method (or another root-finding method) to f' is equally likely to lead to a maximum as to a minimum. However, this can be useful when we know that a function has just one critical point and that point is a minimum.

30.4 Golden section search

This is a bracket-based method for minimization which is similar to bisection method. However, if we applied bisection method to f' we might find a local maximum. The **golden section** search will not do that; it is designed to search for a minimum.

The bisection method was based on the idea that if $f(a)f(b) < 0$ (and f is continuous), then there is a root of f on the interval $[a, b]$. What do we need to be sure there is at least a local minimum on $[a, b]$? Some point $c \in [a, b]$ such that $f(c) < f(a)$ and $f(c) < f(b)$. However, if we divide the interval into $[a, c]$ and $[c, b]$, it is not clear what to do next: which part should we keep?

Instead, the bracket should include two interior points, say $a < c < d < b$. Then we can compare $f(c)$ and $f(d)$; the smaller one will point to us which interval to keep, either $[a, d]$ (if $f(c)$ is smaller) or $[c, b]$ (if $f(d)$ is smaller).

The choice of two points c, d is not obvious. We could place them 1/3 of the way from each end, namely $c = a + (b - a)/3$ and $d = b - (b - a)/3$. Then if, for example, $f(c) < f(d)$, we know there is a point of minimum in $[a, d]$. To keep the process going, need to pick two interior evaluation points on $[a, d]$. Frustratingly, the point c which is in $[a, d]$, does not fit our algorithm because it is in the middle of $[a, d]$, not 1/3 from an edge.

Golden section method improves on the above by choosing c, d as follows:

$$c = a + r(b - a), \quad d = b - r(b - a), \quad \text{where } r = \frac{3 - \sqrt{5}}{2}$$

The number r is related to golden ratio $\varphi = (\sqrt{5} + 1)/2 \approx 1.618$ by $r = 1/\varphi^2$. The following intervals are all in the golden ratio to each other:

$$\frac{b-a}{d-a} = \frac{b-a}{b-c} = \frac{d-a}{c-a} = \frac{b-c}{b-d} = \frac{c-a}{d-c} = \frac{b-d}{d-c} = \varphi$$

The result of this invariance of the ratio is that when the bracket is reduced, the remaining interior point again divides it in the golden ratio.

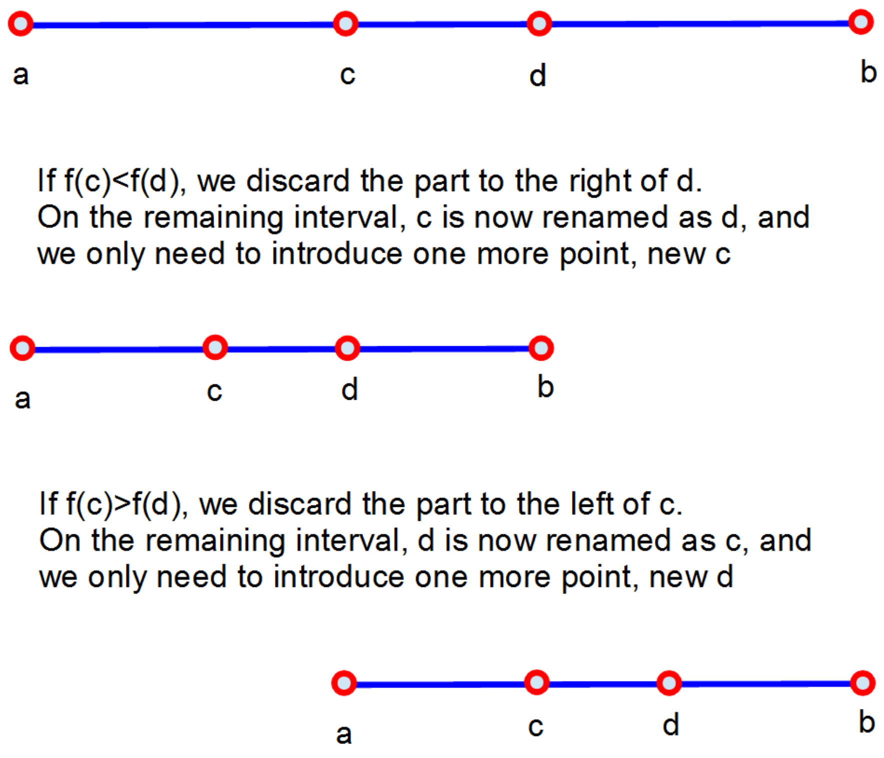
a
c
d
b
If f(c)<f(d), we discard the part to the right of d.
On the remaining interval, c is now renamed as d, and
we only need to introduce one more point, new c
a
c
d
b
If f(c)>f(d), we discard the part to the left of c.
On the remaining interval, d is now renamed as c, and
we only need to introduce one more point, new d
a
c
d
b

Figure 30.4.1 Golden section search

With each step of iteration, only one additional evaluation of f is needed. The bracket size is divided by φ at every step. This is linear rate of convergence, which is slow but reliable (if the function is unimodal), and we don't need the derivative of f to use this method.

What if the function is not unimodal? If we are lucky, the process may converge to the global minimum. But it's quite possible that the global minimum will be lost from the bracket at some step and then we'll converge to a local minimum instead.

Example 30.4.2 Minimize by golden section. Minimize $f(x) = x^4 - 3x^2 + x + 1$ using the golden section method on the interval $[-2, 2]$.

Solution.

```
f = @(x) x.^4 - 3*x.^2 + x + 1;
a = -2;
b = 2;
r = (3-sqrt(5))/2;

while b-a >= 100*eps(a)
    c = a + r*(b-a);
    d = b - r*(b-a);
    if f(c) < f(d)
        b = d;
    else
        a = c;
    end
end

fprintf('Found a minimum x = %g with f(x) = %g \n', c, f(c));
```

□

The code in Example 30.4.2 does not explicitly store the previously computed values to be re-used at next step, but, depending on the language used, such *caching* of previously computed function values will happen behind the scenes.

30.5 Homework

1. (Theoretical) For each of the following functions, use the derivative f' to determine if it is unimodal on the given interval. You do not need to actually find the minimum.

(a) $e^{-x}\sin x$ on $[1, 100]$

(b) $\ln(x) + 10/x$ on $[1, 100]$

(c) $x^2 e^x$ on $[-10, 10]$

(d) $x^3 e^x$ on $[-10, 10]$

2. One can combine the reliability of golden section method with the speed of Newton's method as follows: start with golden section, and when the bracket becomes small (say, less than 0.1), switch to Newton's method. This makes sense because once we get close to a root of f', Newton's method converges to that specific root very quickly.

Apply the idea of previous paragraph to the minimization problem in Example 30.4.2. This means exiting the while loop sooner, and following Example 30.3.1 after that.

Chapter 31

Parabolic Interpolation and Gradient Descent

Previously we considered the analogues of bisection and Newton method for minimization. There is also an analogue of secant method: parabolic interpolation. It does not require derivatives. On the other hand, if the first-order derivative is available, it can be used for minimization in a way different from Newton's method.

31.1 Successive parabolic interpolation

Recall that Newton's method for root-finding, namely $x = a - f(a)/f'(a)$, has a geometric interpretation: draw a tangent line to $y = f(x)$ at $x = a$, and use the intersection of that line with the horizontal axis as the new x-value. This geometric construction naturally led to the secant method: just replace a tangent line by a secant line. Is there a geometric interpretation for minimization using Newton's method, that is $x = a - f'(a)/f''(a)$?

Yes, there is. Let T_2 be the Taylor polynomial at $x = a$ of degree 2, namely

$$T_2(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2$$

The critical point of this polynomial is found by equating T_2' to zero: since

$$T_2'(x) = f'(a) + f''(a)(x-a)$$

we get $x = a - f'(a)/f''(a)$. Therefore, the logic of minimization by Newton's method is to draw a "tangent parabola" at $x = a$ and use its critical point (hopefully a minimum) as the new x-value.

Having made this geometric observation, we can eliminate derivatives from the picture by constructing a "secant parabola" instead, which is a parabola passing through three points on the graph of a function. Following the logic of the secant method, we should replace the "oldest" of the three points used to construct the parabola by the critical point of this parabola. The process repeats until it (hopefully) converges to a point of minimum. This is the method of **successive parabolic interpolation**.

How to find the critical point of parabola passing through points $(a, f(a))$, $(b, f(b))$, $(c, f(c))$? Newton interpolating polynomial is one way: it provides the interpolating polynomial in the form

$$g(x) = f(a) + \alpha(x-a) + \beta(x-a)(x-b) \tag{31.1.1}$$

from where we can find $g'(x) = \alpha + \beta(2x - a - b)$, hence

$$x = \frac{a+b}{2} - \frac{\alpha}{2\beta} \tag{31.1.2}$$

The coefficients α, β are determined from $g(b) = f(b)$ and $g(c) = f(c)$, namely

$$\alpha = \frac{f(b) - f(a)}{b - a}, \quad \beta = \frac{f(c) - f(a) - \alpha(c-a)}{(c-a)(c-b)} \tag{31.1.3}$$

Example 31.1.1 Minimize by successive parabolic interpolation. Minimize $f(x) = x^4 - 3x^2 + x + 1$ using successive parabolic interpolation with initial triple of points $a, b, c = -2, 0, 2$.

Solution.

```
f = @(x) x.^4 - 3*x.^2 + x + 1;
a = -2;
b = 0;
c = 2;

max_tries = 10000;

for k = 1:max_tries
    alpha = (f(b)-f(a))/(b-a);
    beta = (f(c)-f(a)-alpha*(c-a))/((c-a)*(c-b));
    x = (a+b)/2 - alpha/(2*beta);
    a = b;
    b = c;
    c = x;
    if max([a b c]) - min([a b c]) < 1e-6
        break
    end
end

if k < max_tries
    fprintf('Found x = %g with f(x) = %g after %d steps\n', c, f(c), k);
else
    disp('Failed to converge')
end
```

As the example shows, parabolic interpolation can converge to a local maximum. Indeed, a closer look shows it does not distinguish between f and $-f$, so it can minimize just as well as maximize. As with Newton's method, we can benefit from this method by using it in a small neighborhood of minimum. □

Would it help if instead of replacing the "oldest" of a, b, c we replaced the point with largest value of f? This would at least make the method prefer minimization to maximization. Try this out on the above example, using something like

```
old = [a b c];
[fm, ind] = max(f(old));
old(ind) = x;
```

and so on. (This sounds like a good idea but it can turn the process into an infinite loop.)

31.2 Gradient descent in one dimension

The method of gradient descent uses only the first derivative of function f to determine the direction in which to look for its minimum. As a motivating example, suppose we started Newton's method at $a = 1$ and found $f'(1) = 3$ and $f''(1) = -2$. According to Newton's method our next point should be $1 - \frac{3}{-2} = 2.5$. But since the function is increasing near 1, should not the search move to the left instead of to the right?

In its simplest, one-dimensional form, gradient descent amounts to repeatedly computing $x = a - \beta f'(a)$ where a parameter $\beta > 0$ may be a fixed number or be somehow adjusted in the process. The idea is to make a small step in the direction where the function f decreases.

Example 31.2.1 First attempt at gradient descent. Minimize the function $f(x) = x^4 - 3x^2 + x + 1$ from Example 30.3.1 using gradient descent with $\beta = 0.1$ and initial point 0.

Solution.

```
f = @(x) x.^4 - 3*x.^2 + x + 1;
fp = @(x) 4*x.^3 - 6*x + 1;

beta = 0.1;
a = 0;
max_tries = 10000;
for k = 1:max_tries
    x = a - beta*fp(a);
    if abs(x-a) < 100*eps(a)
        break
    end
    a = x;
end

if k < max_tries
    fprintf('Found x = %g with f(x) = %g after %d steps\n', x, f(x), k);
else
    disp('Failed to converge')
end
```

The code takes more steps than Newton's method in Example 30.3.1 but it actually minimizes the function. □

If no formula for f' is available, the methods of numerical differentiation (Chapter 12) can be used to implement gradient descent, for example with $f'(a) \approx (f(a+h) - f(a-h))/(2h)$ with some small h.

Unfortunately, gradient descent with fixed β is not a reliable method. Simply multiplying both `f` and `fp` by 2 in Example 31.2.1 results in failure to converge (why? add `disp(x)` to the loop to see). We could make β smaller and restore convergence, but this is manual fine-tuning. The underlying issue is *scaling*: multiplying f by a positive constant does not change the location of its minima, yet it affects the gradient descent if it uses the same β value. (Newton's method is invariant under scaling because f'/f'' is invariant.)

Example 31.2.2 Model case of gradient descent. Determine for what values of β the gradient descent will converge for the function $f(x) = Mx^2$ where $M > 0$.

Solution. Here $f'(x) = 2Mx$, hence $a - \beta f'(a) = (1 - 2M\beta)a$. This converges to 0 if and only if $|1 - 2M\beta| < 1$. This means we need $0 < \beta < 1/M$, and the optimal value of β is $1/(2M)$. Note that $f''(x) = 2M$. □

The message of Example 31.2.2 is that we should try to adjust β to be of the size $1/f''$, even if we do not have a formula for f''. One way to do this is to start with a guess like $\beta = 0.1$ and then *update* β after every step using $\beta = |x - a|/|f'(x) - f'(a)|$. This makes β approximately the reciprocal of $|f''|$. (Similar idea was used in Broyden's method in Chapter 11.) To implement this idea, we insert the line

```
beta = abs(x-a)/abs(fp(x)-fp(a));
```

into the loop in Example 31.2.1 (where?). Now if the function is multiplied by 2 or even 2000, the method continues to converge.

31.3 Homework

1. (Theoretical) Show that no fixed value $\beta > 0$ can make the gradient descent converge for the function $f(x) = |x|^{3/2}$.

(Note that f is differentiable, with $f'(x) = (3/2)\sqrt{x}$ for $x \ge 0$ and $f'(x) = -(3/2)\sqrt{-x}$ for $x \le 0$.)

2. Parabolic interpolation is also useful when applied just once, without iteration. For example, after evaluating f on a grid $x_1, \dots, x_n$ and choosing k with smallest $f(x_k)$ (i.e., brute force minimization) we can get a more precise location of the minimum with parabolic interpolation through the points $(x_j, f(x_j))$, $j = k-1, k, k+1$.

Write a script that applies the above idea to the function $f(x) = \sin(x)$ on the interval $[0, 7]$. Use $n = 100$ points for brute force minimization, locating a grid point x_k with the smallest function value $f(x_k)$. Then find a more precise point of minimum x^* using parabolic interpolation. Since the exact point of minimum is $3\pi/2$, the script should display the differences $|x_k - 3\pi/2|$ and $|x^* - 3\pi/2|$ to illustrate the benefit of parabolic interpolation.

Chapter 32

Gradient methods and Newton's method

Although the idea of following the direction of steepest descent is natural, it has important limitations in multivariate optimization. We consider why the steepest descent might not be the right direction, and some approaches to correct this.

32.1 Gradient descent in several variables

In order to minimize a function $f\colon \mathbb{R}^n \to \mathbb{R}$, we can start with initial vector $\mathbf{a}$ and compute $\mathbf{x} = \mathbf{a} - \beta \nabla f(\mathbf{a})$, then replace $\mathbf{a}$ with $\mathbf{x}$ and repeat until convergence is achieved (or the limit on steps is reached). This is the n-dimensional version of the gradient descent method in Section 31.2. We already saw that the choice of β is difficult even in one dimension. It gets worse in several dimensions.

Consider the function

$$f(\mathbf{x}) = x_1^2 + 10^6 x_2^2 \tag{32.1.1}$$

with the minimum at $(0,0)$. Its gradient is $\nabla f = \langle 2x_1, 2\cdot 10^6 x_2\rangle$. So, the process described above is

$$\mathbf{x} = \langle (1-2\beta)a_1, (1-2\beta\cdot 10^6)a_2\rangle$$

There is no good value of β to use here. If $\beta > 10^{-6}$, the second coordinate grows exponentially. If $\beta < 10^{-6}$, for example $\beta = 10^{-6}/2$ (which is optimal for the second coordinate), it will take a million steps just to reduce the first coordinate, say, from 1 to 0.37.

The underlying issue is that the direction $-\nabla f$ does not really point toward the minimum of f in this example. The steepest descent goes sideways due to very different rates of change in different variables. This is one of main reasons why optimization in several variables is hard, and why it is sensitive to *scaling* of the variables. We would not have such difficulties with $x_1^2 + 3x_2^2$.

One way to improve the situation is to run a one-variable minimization algorithm, **line search** at each step, to find the minimum of f on the line of steepest descent. This means minimizing the function $h(t) = f(\mathbf{a} + t\mathbf{g})$ where $\mathbf{g} = -\nabla f(\mathbf{a})$. If we find the exact minimum on the line each time, then the minimization process for function converges to the minimum quickly. But for slightly more complicated functions the difficulty emerges again. Consider Rosenbrock's function

$$f(\mathbf{x}) = (x_1 - 1)^2 + 100(x_1^2 - x_2)^2 \tag{32.1.2}$$

Its graph has a curved "valley" along the parabola $x_2 = x_1^2$. One-dimensional minimization along the line of steepest descent leads to each consecutive direction being perpendicular to the previous one (why?). So the algorithm ends up zig-zagging in tiny steps in the deep narrow valley, making very little progress toward minimum. And since each step is its own minimization problem (in one dimension), the entire process may take too long.

In order to try the method of previous paragraph in practice, we need a line-search subroutine we will use Matlab's command for one-dimensional minimization on an interval: `fminbnd`. Example:

```
fminbnd(@(t) t*cos(t), 0, 5)
```

returns 3.4256 which is the location of the minimum (not the minimum value). The command `fminbnd` is guaranteed to stay within the given interval but is not guaranteed to find the absolute minimum on that interval. Indeed, `fminbnd(@(t) t*cos(t), 0, 12)` returns the same 3.4256 even though the absolute minimum is at $t \approx 9.5293$. To see what is going on, use `optimset` to enable the display of iterative minimization process.

```
options = optimset('Display', 'iter');
fminbnd(@(t) t*cos(t), 0, 12, options)
```

As this expanded output shows, the command `fminbnd` is a straightforward combination of golden section and parabolic interpolation methods. As a result, it is fast but can easily miss the absolute minimum. For our purposes, using a line search at each step of gradient descent, "fast" is more important.

Example 32.1.1 Gradient descent with line search. Minimize Rosenbrock's function (32.1.2) using line search in the direction of steepest descent, $-\nabla f$, at each step. Use a random starting point `randn(2, 1)`. Plot the path of the search.

Solution. The gradient of Rosenbrock's function is

$$\begin{pmatrix} 2(x_1 - 1) + 400x_1(x_1^2 - x_2) \\ -200(x_1^2 - x_2) \end{pmatrix} \tag{32.1.3}$$

The code uses a matrix `path` to record the path taken by the search.

```
f = @(x) (x(1)-1).^2 + 100*(x(1).^2 - x(2)).^2;
fg = @(x) [2*(x(1)-1) + 400*x(1)*(x(1).^2 - x(2)); -200*(x(1).^2 - x(2))];
x = randn(2, 1);
max_tries = 10000;

path = zeros(2, max_tries);
for k = 1:max_tries
    path(:, k) = x;
    g = -fg(x);    % direction of descent
    t_min = fminbnd(@(t) f(x + t*g), 0, 1);
    dx = t_min*g;
    x = x + dx;
    if norm(dx) < 1e-6
```

```
        break
    end
end

plot(path(1, 1:k), path(2, 1:k), '-+')
if k < max_tries
    fprintf('Found x = (%g, %g) with f(x) = %g after %d steps\n', x, f(x), k);
else
    disp('Failed to converge')
end
```

Whether the code converges or fails depends on the initial point. Often it fails despite being on the right track, because of too many tiny zigzagging steps. □

Note there is a reason to have a less restrictive stopping conditions for step size `norm(dx)` when the goal is minimization (compared to root-finding). A smooth function changes slowly near a point of minimum (like a parabola near its vertex), so an error of size 10^{-6} in terms of location of the minimum could mean an error of about 10^{-12} for the minimal value.

There are several "weak line search" methods designed to determine a step size that makes the function noticeably smaller without finding its exact minimum. They can speed up the process but do not completely resolve the zigzagging issue which is the root of the difficulty in Example 32.1.1. The *directions* seem to be a problem, not just the step sizes.

32.2 Newton's method for multivariate minimization

Recall from Chapter 10 that the Newton method for solving a vector equation $\mathbf{F}(\mathbf{x}) = 0$ proceeds in iterative steps of the form $\mathbf{x} = \mathbf{a} - J(\mathbf{a})^{-1}\mathbf{F}(\mathbf{a})$ where J is the Jacobian matrix of $\mathbf{F}$. For the purpose of minimizing a scalar function f, we let $\mathbf{F} = \nabla f$. The Jacobian matrix J of $\mathbf{F}$ then becomes the **Hessian matrix** of f, which is the matrix of all second-order partial derivatives.

$$H = \begin{pmatrix} \partial^2 f/\partial x_1^2 & \partial^2 f/\partial x_1 \partial x_2 \\ \partial^2 f/\partial x_1 \partial x_2 & \partial^2 f/\partial x_2^2 \end{pmatrix}$$

Thus, the Newton method for minimization proceeds in steps $\mathbf{x} = \mathbf{a} - H^{-1}\nabla f(\mathbf{a})$. One can view the application of H^{-1} as the "course correction": with rare exceptions, the method does *not* choose the direction of steepest descent. For the badly scaled quadratic function (32.1.1) the Newton direction is right on target: since

$$H = \begin{pmatrix} 2 & 0 \\ 0 & 2\cdot 10^6 \end{pmatrix}$$

the step taken from any point $\mathbf{a}$ is

$$-H^{-1}\nabla f(\mathbf{a}) = -\begin{pmatrix} 1/2 & 0 \\ 0 & (1/2)\cdot 10^{-6} \end{pmatrix} \begin{pmatrix} 2a_1 \\ 2\cdot 10^6 a_2 \end{pmatrix} = -\mathbf{a}$$

so we arrive at the minimum $\mathbf{x} = 0$ after one step. This works for any quadratic function.

Since a smooth function locally looks like a quadratic polynomial (2nd order Taylor polynomial), Newton's method has a chance of improving the situation even for challenging functions like Rosenbrock's. Let us try it out.

Example 32.2.1 Multivariate minimization with Newton's method. Minimize Rosenbrock's function (32.1.2) using Newton's method with a random starting point `randn(2, 1)`. Plot the path of the search.

Solution. The gradient of Rosenbrock's function is (32.1.3) and its Hessian matrix is

$$\begin{pmatrix} 2+400(x_1^2-x_2)+800x_1^2 & -400x_1 \\ -400x_1 & 200 \end{pmatrix} \tag{32.2.1}$$

```
f = @(x) (x(1)-1).^2 + 100*(x(1).^2 - x(2)).^2;
fg = @(x) [2*(x(1)-1) + 400*(x(1).^2 - x(2))*x(1); -200*(x(1).^2 - x(2))];
fh = @(x) [2 + 400*(x(1)^2-x(2)) + 800*x(1)^2,  -400*x(1); -400*x(1), 200];
x = randn(2, 1);

max_tries = 10000;

path = zeros(2, max_tries);
for k=1:max_tries
    path(:, k) = x;
    dx = -fh(x)\fg(x);
    x = x + dx;
    if norm(dx) < 1e-6
        break
    end
end

plot(path(1, 1:k), path(2, 1:k), '-+')
if k < max_tries
    fprintf('Found x = (%g, %g) with f(x) = %g after %d steps\n', x, f(x), k);
else
    disp('Failed to converge')
end
```

The process usually makes a few seemingly random jumps but then quickly converges. □

The downsides of Newton's method were already noted in Section 30.3: it requires second-order derivatives, and it could just as well converge to a local maximum.

32.3 Conjugate gradient method

The methods considered so far in this chapter have one thing in common: lack of memory. Each step is taken as if it was the first. In contrast, **conjugate gradient method** uses its previous step to determine the direction of next one. The primary benefit is avoiding "sharp turns" which create a zigzagging pattern we saw in Example 32.1.1. The method described below is sometimes called *nonlinear* conjugate gradient method because "conjugate gradient method" often refers specifically to minimization of a quadratic function $f(\mathbf{x}) = |A\mathbf{x} - \mathbf{b}|^2$ as a way of approximately solving the linear system $A\mathbf{x} = \mathbf{b}$.

Suppose we arrived to point $\mathbf{a}$ from point $\mathbf{b}$, following the direction vector $\mathbf{v}$. With the conjugate gradient method, the direction of our next move will be$\mathbf{w} = -\nabla f(a) + \gamma\mathbf{v}$ where $\gamma > 0$. Large γ means we keep mostly the same direction as before, small γ means we go with the gradient. We should make γ smaller, giving more importance to the gradient, if $\mathbf{a}$ is much closer to the minimum than $\mathbf{b}$. This progress can be measured by the ratio $|\nabla f(\mathbf{a})|^2/|\nabla f(\mathbf{b})|^2$ (small if we arrived in the vicinity of minimum), so we use this quantity as γ.

Example 32.3.1 Conjugate gradient method. Modify Example 32.1.1 so that it minimizes the simplified Rosenbrock's function

$$f(\mathbf{x}) = (x_1 - 1)^2 + (x_1^2 - x_2)^2 \tag{32.3.1}$$

using conjugate gradient method. Use a random starting point `randn(2, 1)`. Plot the path of the search.

Solution.

```
f = @(x) (x(1)-1).^2 + (x(1).^2 - x(2)).^2;
fg = @(x) [2*(x(1)-1) + 4*x(1)*(x(1).^2 - x(2)); -2*(x(1).^2 - x(2))];

x = randn(2, 1);
v = zeros(2, 1);
gamma = 0;
max_tries = 10000;

path = zeros(2, max_tries);
for k = 1:max_tries
    path(:, k) = x;
    w = -fg(x) + gamma*v;  % with correction
    t_min = fminbnd(@(t) f(x + t*w), 0, 1);
    dx = t_min*w;
    if norm(dx) < 1e-6
        break
    end

    gamma = norm(fg(x+dx))^2/norm(fg(x))^2;  % update gamma
    x = x + dx;                              % update x
    v = w;                 % record the step for the future
end

plot(path(1, 1:k), path(2, 1:k), '-+')
if k < max_tries
    fprintf('Found x = (%g, %g) with f(x) = %g after %d steps\n', x, f(x), k);
else
    disp('Failed to converge')
end
```

Compare the search path to what we get without correction in `w`: it is less zigzagging. However, if we use the original Rosenbrock function (32.1.2) which is very far from quadratic or convex, the search diverges to infinity. Several other correction terms (different choices for γ coefficient) have been proposed for the nonlinear conjugate gradient method. But it appears that the "narrow valley" landscape of the Rosenbrock function is best navigated either by using both first and second order derivatives (the Newton method) or by using no derivatives at all (the Nelder-Mead method, to be considered later). □

32.4 Homework

1. Modify Example 32.2.1 to include the idea of Broyden's method from Example 11.4.1: the script should not use the Hessian matrix `fh`, instead relying on an approximation to its inverse obtained with Broyden's method. The following modification should help with convergence: start with small initial guess

    ```
    B = 0.001*eye(2);
    ```

    and make the stopping condition less restrictive:

    ```
    norm(h) < 1e-6
    ```

    Re-run the script several times (about five): since it has a random initial point, the results may differ. Does the minimization process converge to the minimum $(1,1)$ every time?

Remark: the combination of ideas of Broyden's method and line search, with some modifications, is known as the BFGS algorithm, which is widely used for gradient-based unconstrained optimization. It is implemented in Matlab (`fminunc` command), Python scientific library SciPy, etc.

Chapter 33

The Nelder-Mead method

Functions of one variable can be efficiently minimized by a combination of golden section and parabolic interpolation methods, which is implemented by `fminbnd` command in Matlab. These methods are derivative-free: we do not need to either know or estimate any derivatives of the objective function. In contrast, all methods for minimization in several variables that we studied so far require derivatives. The goal of this chapter is to understand a derivative-free minimization method introduced by Nelder and Mead in 1965, which powers the Matlab command `fminsearch`.

33.1 First attempt at derivative-free minimization

The intuitive picture of steepest descent (the gradient method) is a drop of water sliding down the graph of a function under the force of gravity: the water follows the steepest way down. The intuition of the Nelder-Mead method is a geometric shape like a triangle or a tetrahedron tumbling down a sloped surface. In two dimensions, the shape is a triangle, in three dimensions it is a tetrahedron; in n dimensions it is an n-dimensional **simplex**, meaning $n+1$ points that are not contained in any hyperplane. To keep the explanation simple, we focus on the case $n = 2$.

Given a triangle ABC in xy-plane, we evaluate the function f at its vertices. The vertex with the largest value of f (for example, C) *might* get replaced by its reflection. One could imagine reflecting C about the opposite side AB by dropping a perpendicular line from C onto AB and following it. However, this kind of reflection is not invariant under linear transformations of xy plane, such as the scaling of coordinates. The Nelder-Mead method uses reflection about the midpoint of AB which can be expressed in vector form as

$$R = A + B - C = (A + B + C) - 2C \tag{33.1.1}$$

The second version is slightly easier to code, see below.

However, it would be pointless to replace C by R if this did not make the function smaller. So the replacement only happens if $f(R) < f(C)$. Otherwise the program stops and reports the midpoint $(A + B + C)/3$ of the last computed triangle as an approximate point of minimum. This is a "reflection-only" version of the Nelder-Mead method.

Example 33.1.1 Reflection-only Nelder-Mead method. Minimize the function

$$f(\mathbf{x}) = x_1^4 + x_2^2 - \sin(x_1 + x_2) \tag{33.1.2}$$

using the reflection-only version of the Nelder-Mead method. Use a random starting point `randn(2, 1)` as one vertex A of the initial triangle, and let other vertices be $A + h\mathbf{e}_1$, $A + h\mathbf{e}_2$ with small $h = 0.01$ (this makes a small right isosceles triangle). Plot the path of the search.

Solution. It is convenient to arrange column vectors A, B, C into a matrix T representing a triangle. This makes it easy to replace the vertex with greatest value of f after identifying it with two-output `max` command. The command `mean(T, 2)` computes the mean $(A+B+C)/3$ while `sum(T, 2)` is $A+B+C$.

```
f = @(x) x(1)^4 + x(2)^2 - sin(x(1)+x(2));

A = randn(2, 1);
B = A + [0.01; 0];
C = A + [0; 0.01];
T = [A B C];
max_tries = 10000;
path = zeros(2, max_tries);

for k = 1:max_tries
    path(:, k) = mean(T, 2);
    values = [f(T(:,1)) f(T(:,2)) f(T(:,3))];
    [fmax, ind] = max(values);
    R = sum(T, 2) - 2*T(:, ind);
    if f(R) < fmax
        T(:, ind) = R;
    else
        break
    end
end

plot(path(1, 1:k), path(2, 1:k), '-+')
if k < max_tries
    x = mean(T, 2);
    fprintf('Found x = (%g, %g) with f(x) = %g after %d steps\n', x, f(x), k);
else
    disp('Failed to converge')
end
```

□

33.2 Reflection-contraction combination

One issue with the reflection-only method of Section 33.1 is the fixed size of the triangle. With a large triangle, we will only have a rough idea of where the minimum is when the search ends. Also, a large triangle will not detect any fine features of the landscape such as the narrow valley of Rosenbrock's function. On the other hand, a small triangle necessarily moves in small steps and is therefore slow.

There is a way to improve the situation: instead of stopping when reflection does not work, reduce the size of triangle in half, contracting it toward the "best" vertex (the one is the with lowest value). The contraction map is $f(x) =$

$(x + b)/2$ where b is the vertex toward which we contract. This reflection-contraction method needs a stopping criterion, so it does not run forever. We can stop when the triangle becomes very small, with vertices getting close to one another.

Example 33.2.1 Reflection-contraction Nelder-Mead method. Minimize the function (33.1.2) using the reflection-contraction version of the Nelder-Mead method. Use a random starting point `randn(2, 1)` as one vertex A of the initial triangle, and let other vertices be $A + \mathbf{e}_1$, $A + \mathbf{e}_2$, so that the initial triangle is not small anymore. Plot the path of the search.

Solution. The code is mostly the same as in Example 33.1.1, with additional lines noted in comments.

```
f = @(x) x(1)^4 + x(2)^2 - sin(x(1)+x(2));
% f = @(x) (x(1)-1)^2 + 100*(x(1)^2 - x(2))^2;

A = randn(2, 1);
B = A + [1; 0];
C = A + [0; 1];
T = [A B C];
max_tries = 10000;
path = zeros(2, max_tries);

for k = 1:max_tries
    path(:, k) = mean(T, 2);
    if max(abs(T - path(:, k))) < 1e-6
        break     % stop, the triangle is small enough
    end
    values = [f(T(:,1)) f(T(:,2)) f(T(:,3))];
    [fmax, ind] = max(values);
    R = sum(T, 2) - 2*T(:, ind);
    if f(R) < fmax
        T(:, ind) = R;
    else
        [fmin, ind] = min(values);   % find the best vertex
        T = (T + T(:, ind))/2;       % contract toward it
    end
end

plot(path(1, 1:k), path(2, 1:k), '-+')
if k < max_tries
    x = mean(T, 2);
    fprintf('Found x = (%g, %g) with f(x) = %g after %d steps\n', x, f(x), k);
else
    disp('Failed to converge')
end
```

□

Example 33.2.1 works fine in terms of speed and accuracy. However, replacing the relatively simple function f in Example 33.1.1 with Rosenbrock's function (32.1.2)

```
f = @(x) (x(1)-1)^2 + 100*(x(1)^2 - x(2))^2
```

we find the search usually fails. The moving triangle has to become very small to fit into the "narrow valley" of this function, which makes subsequent movement along the valley floor very slow. As as the valley is long, it fails to reach the minimum within the allowed number of steps.

33.3 Reflection-contraction-expansion Nelder-Mead method

The reason for the lack of success with Rosenbrock's function is that the version of Nelder-Mead method described in Section 33.2 does not allow the triangle to change its *shape* to fit the geometry of the graph. A long narrow valley calls for a long narrow triangle.

The process of changing the shape of the triangle is *expansion*, which is introduced as an alternative to reflection from Section 33.1. Suppose that the current triangle qualifies for reflection: for example, C is the point with largest value, and $f(R) < f(C)$. Consider the possibility of replacing C with a point E that lies beyong R, thus making the triangle longer. The point E is determined by the following equation, which shows that it is as far from R as R is from the center of reflection $(A+B)/2$:

$$E = \frac{3}{2}(A+B) - 2C = 1.5(A+B+C) - 3.5C$$

If $f(E) < f(R)$, then we replace C with E rather than R. In other words, the "highest" vertex of the triangle is replaced by whichever of E, R is lower.

Having both contraction and expansion in the process means the triangle can grow smaller and larger, moving faster when the path is clear, or becoming very narrow to fit into a difficult to reach valley.

Example 33.3.1 Reflection-contraction-expansion Nelder-Mead method. Minimize Rosenbrock's function (32.1.2) using the reflection-contraction-expansion version of the Nelder-Mead method. Use a random starting point `randn(2, 1)` as one vertex A of the initial triangle, and let other vertices be $A+\mathbf{e}_1$, $A+\mathbf{e}_2$. Plot the path of the search.

Solution. The code is mostly the same as in Example 33.2.1, with additional lines noted in comments.

```
f = @(x) (x(1)-1)^2 + 100*(x(1)^2 - x(2))^2;

A = randn(2, 1);
B = A + [1; 0];
C = A + [0; 1];
T = [A B C];
max_tries = 10000;
path = zeros(2, max_tries);

for k = 1:max_tries
    path(:, k) = mean(T, 2);
    if max(abs(T - path(:, k))) < 1e-6
        break
    end
    values = [f(T(:,1)) f(T(:,2)) f(T(:,3))];
    [fmax, ind] = max(values);
    R = sum(T, 2) - 2*T(:, ind);
    if f(R) < fmax
        E = 1.5*sum(T, 2) - 3.5*T(:, ind);  % consider expansion
        if f(E) < f(R)
```

```
            T(:, ind) = E;                  % choose to expand
        else
            T(:, ind) = R;                  % choose to reflect
        end
    else
        [fmin, ind] = min(values);
        T = (T + T(:, ind))/2;
    end
end

plot(path(1, 1:k), path(2, 1:k), '-+')
if k < max_tries
    x = mean(T, 2);
    fprintf('Found x = (%g, %g) with f(x) = %g after %d steps\n', x, f(x), k);
else
    disp('Failed to converge')
end
```

□

The algorithm described above works quite well, considering we are minimizing a challenging function without using any derivative information. The version of the Nelder-Mead algorithm implemented in Matlab is a little different in that it chooses between three ways of contracting a simplex (called “contract inside”, “contract outside”, “shrink”) but the main ideas are the same. To observe the process of `fminsearch` optimization, run

```
f = @(x) (x(1)-1)^2 + 100*(x(1)^2 - x(2))^2;
options = optimset('Display', 'iter');
fminsearch(f, randn(2, 1), options)
```

33.4 Homework

1. The stopping condition of the method in Example 33.3.1 is that

```
max(abs(T - path(:, k))) < 1e-6
```

meaning that every coordinate of every vertex of triangle T with within 10^{-6} of the corresponding coordinate of the center of T. This is just saying “the triangle is small”, and makes no use of the values of function f. Replacing this stopping condition with “the difference between the largest and smallest of the values of f at the vertices of T is less than 10^{-6}.” Does the method still reliably find the minimum $(1, 1)$ of Rosenbrock’s function? Does it take more or fewer steps with the new stopping condition, or about the same as before?

Chapter 34

Constrained Optimization

Often the variables in an optimization problem cannot be chosen arbitrarily: there are constraints as as being nonnegative, or having some upper bound, or satisfying some equation or inequality (which could be linear or nonlinear). We consider some approaches to constrained optimization problems, including the possibility of converting them to unconstrained optimization by adding a penalty term.

34.1 Penalty method

A typical problem of constrained optimization is to find the minimum of a function f on some region D of $\mathbb{R}^n$. One approach is to add a penalty for being outside of D, for example let

$$g(x) = \begin{cases} f(x) & x \in D \\ f(x) + M & x \notin D \end{cases} \tag{34.1.1}$$

where M is a large number, say 10^6. Then try to minimize g, with the expectation that the minimum will be in D. Since $g = f$ within D, such a minimum will also be a minimum of f.

The *constant penalty* (34.1.1) has several disadvantages. It makes g discontinuous, and it does not eliminate *local* minima outside of D. Since minimization methods often converge to a local minimum, we may end up outside of D even though there are much smaller values within D.

A way to correct both of the above issues is to add to f a continuous penalty function P that is equal to 0 inside D and is positive outside of D. Then minimize $f + P$. One way to construct P is to describe the region D by way of an inequality $c \le 0$, for example $x^2 + y^2 - 1 \le 0$ describes a disk of radius 1. There are multiple ways to use such an inequality to form a penalty function. For example, the linear penalty

$$P_1(x) = M \max(0, c) \tag{34.1.2}$$

(with a large constant M) and quadratic penalty

$$P_2(x) = M \max(0, c)^2 \tag{34.1.3}$$

The linear penalty is continuous but not differentiable on the boundary of D. The quadratic penalty is differentiable, which is preferable for the methods that rely on the gradient. Indeed, the gradient of (34.1.3) is

$$\nabla P_2(x) = 2M \max(0, c) \nabla c \tag{34.1.4}$$

On the other hand, the fact that P_2 starts off with zero slope makes it more likely that the minimum of $f + P_2$ will be slightly outside of the region D. So there is a tradeoff between reaching the minimum and enforcing the constraint.

Let us try both methods with a simple function: minimize $f(x, y) = (x + 7y)^3$ on the disk $x^2 + y^2 \le 1$. We could use our own implementation of Nelder-Mead method from Chapter 33 but `fminsearch` already does that, so it is used below.

Example 34.1.1 Comparing two penalty functions. Minimize $f(x, y) = (x + 7y)^3$ on the disk $x^2 + y^2 \le 1$ by using `fminsearch` with a penalty term.

Solution.

```
f = @(x) (x(1) + 7*x(2)).^3;
c = @(x) x(1).^2 + x(2).^2 - 1;
M = 1e6;

x0 = randn(2, 1);
xm1 = fminsearch(@(x) f(x) + M*max(c(x), 0), x0);
xm2 = fminsearch(@(x) f(x) + M*max(c(x), 0).^2, x0);

fprintf('Linear penalty: found x = (%g, %g), |x| = %g, f(x) = %g \n', xm1, norm(xm1), f(xm1));
fprintf('Quadratic penalty: found x = (%g, %g), |x| = %g, f(x) = %g \n', xm2, norm(xm2), f(xm2)
```

With the linear penalty, the minimization process sometimes fails to converge within the allowed attempts; but if it converges, it stays in the unit disk. With the quadratic penalty, the convergence is more reliable but the norm of the point of minimum is usually greater than 1. □

In practice, quadratic penalty is usually preferable and the issue of the minimum being slightly out of bounds is dealt with by increasing M and using the previously found point of minimum as a new starting point. This is done below.

Example 34.1.2 Iterative minimization on the unit disk. Minimize $f(x, y) = (x + 7y)^3$ on the disk $x^2 + y^2 \le 1$ by iteratively using `fminsearch` with a quadratic penalty term.

Solution.

```
f = @(x) (x(1) + 7*x(2)).^3;
c = @(x) x(1).^2 + x(2).^2 - 1;
x0 = randn(2, 1);

for M = [1e3, 1e6, 1e9, 1e12]
    x0 = fminsearch(@(x) f(x) + M*max(c(x), 0).^2, x0);
end

fprintf('Found x = (%g, %g), |x| = %g, f(x) = %g \n', x0, norm(x0), f(x0));
```

With the linear penalty, the minimization process sometimes fails to converge within the allowed attempts; but if it converges, it stays in the unit disk. With the quadratic penalty, the convergence is more reliable but the norm of the point of minimum is usually greater than 1. □

Note that the penalty method works just as well with **equality constraints** of the form $c = 0$. In this case one adds Mc^2 instead of $M \max(c, 0)^2$ to the objective function. And if there are multiple constraints, they are all added.

34.2 Lagrange multiplier method

Suppose we want to minimize f subject to equality constraint $c = 0$ in n-dimensional space. A point x^* of such constrained minimum must have ∇f perpendicular to the surface $c = 0$. Therefore, ∇f is parallel to ∇c. We can express this as $\nabla f = \lambda \nabla c$ where λ is the **Lagrange multiplier** (so far unknown). We have the system

$$\nabla f(\mathbf{x}) - \lambda \nabla c(\mathbf{x}) = 0, \quad c(\mathbf{x}) = 0$$

of $n + 1$ equations with $n + 1$ unknowns. The Jacobian of this system is the block matrix

$$\begin{pmatrix} Hf & -\nabla c \\ (\nabla c)^T & 0 \end{pmatrix}$$

where Hf is the Hessian of f. It is feasible to solve the system using multi-varible Newton's method, for example.

Example 34.2.1 Using the Lagrange multiplier method. Minimize $f(\mathbf{x}) = x_1^4 + \exp(3x_1 + x_2)$ on the circle $x_1^2 + x_2^2 = 1$ using the Lagrange multiplier method.

Solution. The code is similar to Example 32.2.1 but with more complicated gradient and Hessian. To avoid notational confusion, it uses `x` for the vector (x_1, x_2) and `v` for the vector (x_1, x_2, λ). The functions `Fg` and `Fh` below are the gradient and Hessian of the function

$$F(x_1, x_2, \lambda) = x_1^4 + \exp(3x_1 + x_2) - \lambda(x_1^2 + x_2^2 - 1)$$

We equate `Fg` to zero vector, and use `Fh` according to Newton's method.

```
f = @(x) x(1)^4 + exp(3*x(1)+x(2));

Fg = @(v) [4*v(1)^3 + 3*exp(3*v(1) + v(2)) - v(3)*(2*v(1)); ...
    exp(3*v(1) + v(2)) - v(3)*(2*v(2)); ...
    -(v(1)^2 + v(2)^2 - 1)];
Fh = @(v) [12*v(1)^2 + 9*exp(3*v(1) + v(2)) - v(3)*2, 3*exp(3*v(1) + v(2)), -(2*v(1)); ...
    3*exp(3*v(1) + v(2)), exp(3*v(1) + v(2)) - v(3)*2, -(2*v(2)); ...
    -2*v(1),  -2*v(2), 0];

v = randn(3, 1);

max_tries = 10000;

for k=1:max_tries
    dv = -Fh(v)\Fg(v);
    v = v + dv;
    if norm(dv) < 1e-6
        break
    end
end

x = v(1:2);
if k < max_tries
    fprintf('Found x = (%g, %g) with f(x) = %g after %d steps\n', x, f(x), k);
else
    disp('Failed to converge')
end
```

The method converges quickly and the point it finds satisfies to constraint. But this point is often a maximum rather than a minimum of the function. As usual with Newton's method, the situation can be improved by running a rough search algorithm first, just to locate a good starting point. □

Matlab provides a command `fmincon` which implements constrained minimization with various constraints, both equality and inequality types. However, its availability may vary depending on version, and Octave does not have it built-in.

34.3 Homework

1. (Theoretical) Use the Lagrange multiplier method to find the critical points of the function $f(x, y) = (x + 7y)^3$ on the circle $x^2 + y^2 = 1$. Compare with the result of Example 34.1.2.

2. Using the penalty method of Example 34.1.2, find four numbers x_1, x_2, x_3, x_4 on the interval $[-1, 1]$ for which the product of all pairwise distances

$$\prod_{1\le k<\ell\le 4} |x_k - x_\ell|$$

is as large as possible. What pattern do you observe in the points that achieve this maximum?

Remark: such maximization problem makes sense for any number of points, and on any bounded set. The points that maximize the product of distances are known as **Fekete points** and are used in approximation theory, specifically for polynomial interpolation.

Hint: to improve the performance of `fminsearch`, do not use absolute value: instead, minimize the product

$$\prod_{1\le k<\ell\le 4} (x_k - x_\ell)$$

This product can be concisely expressed in Matlab as

```
prod(x(1:3)-x(4))*prod(x(1:2)-x(3))*(x(1)-x(2))
```

The constraint function `c` could be simply `max(abs(x)) - 1`; this constrains all four points to the interval at once.

Chapter 35

Linear Programming

A special but very important case of constrained optimization is minimizing a linear function on a set that is described by a set of linear inequalities or equations (a **convex polyhedron**). A large number of applied problems, for example in economics, can be modeled by a linear programming problem.

35.1 Introduction to linear programming

Suppose we want to maximize the function $f(\mathbf{x}) = x_1 + 2x_2 - 5x_3$ subject to constraints $0 \le x_1 \le x_2 \le x_3 \le 10$ and $2x_3 - x_1 \ge 3$. This is a linear programming **(LP)** problem, since each constraint is a linear inequality and the objective function is linear.

There are various **standard forms** to which linear programming problems can be transformed by manipulations such as the following:

- An equality constraint can be expressed as the combination of $\le$ and $\ge$ constraints.
- Any $\ge$ constraint becomes $\le$ after multiplication by -1.
- Maximization of f is equivalent to minimization of $-f$.

Using the transformations described above, any LP can be expressed as:

$$\mathbf{c}^T\mathbf{x} \to \min \quad \text{where } A\mathbf{x} \le \mathbf{b} \tag{35.1.1}$$

where the inequality between vectors $A\mathbf{x} \le \mathbf{b}$ is understood as a system of linear inequalities: each component on the left is less than or equal to the corresponding component on the right. Compare this to how $A\mathbf{x} = \mathbf{b}$ means a system of linear equations.

Sometimes the nature of the problem requires the variables x_k to be non-negative, which leads to

$$\mathbf{c}^T\mathbf{x} \to \min \quad \text{where } A\mathbf{x} \le \mathbf{b}, \quad \mathbf{x} \ge 0 \tag{35.1.2}$$

Or, one may have linear equations instead of inequalities:

$$\mathbf{c}^T\mathbf{x} \to \min \quad \text{where } A\mathbf{x} = \mathbf{b}, \quad \mathbf{x} \ge 0 \tag{35.1.3}$$

One can convert between these three "standard" forms as follows.

From (35.1.1) to (35.1.2): replace each variable x_k with the difference $x_k = x_k' - x_k''$, and require x_k', x_k'' to be nonnegative. Since every real number is the difference of two nonnegative numbers, this does not change the essence of the problem. In terms of A, b, this transformation keeps $\mathbf{b}$ the same but doubles the size of A by appending $-A$ to the right. The vector $\mathbf{c}$ is similarly doubled.

From (35.1.2) to (35.1.1): add linear inequalities of the form $-x_k \le 0$. This means appending $-I$ to the bottom of matrix A, with zeros added to the bottom of vector $\mathbf{b}$.

From (35.1.2) to (35.1.3): introduce **slack variables** $z_k = b_k - (A\mathbf{x})_k$. Then replace all linear inequalities $A\mathbf{x} \le \mathbf{b}$ with nonnegativity constraints $\mathbf{z} \ge 0$. What remains is the definition of slack variables, which is a system of linear *equations*.

From (35.1.3) to (35.1.2): rewrite each linear equation as two linear inequalities. That is, a system of linear equations $A\mathbf{x} = \mathbf{b}$ is equivalent to having two systems of linear inequalities: $A\mathbf{x} \le \mathbf{b}$ and $A\mathbf{x} \ge \mathbf{b}$. The latter system is rewritten as $(-A)\mathbf{x} \le -\mathbf{b}$ and appended to the former.

In Matlab, the problem (35.1.1) is solved with

```
x = linprog(c, A, b)
```

which returns the optimal $\mathbf{x}$. One can get the optimal value of objective function with `c'*x`.

Example 35.1.1 A simple linear programming problem. Maximize $x_1 + x_2$ subject to constraints $x_1 + 2x_2 \le 4$ and $3x_1 + x_2 \le 5$.

Solution. Rephrasing this as the minimization of $-x_1 - x_2$, we see that the coefficient vector should be `[-1 -1]`. Executing

```
linprog([-1; -1], [1 2; 3 1], [4; 5])
```

yields `[1.2; 1.4]`. (Try to visualize this problem and its solution.) □

Example 35.1.1 illustrates basic use of `linprog`. This command allows more parameters which express equality constraints and additional lower and upper bounds on the variables: see `help linprog`.

Octave-specific remark. The current version of Octave has a similar function `glpk` instead of `linprog`. The main difference is that `glpk(c, A, b)` solves (35.1.3) instead of (35.1.1). However, the fourth argument of `glpk` can be used to set lower bounds to $-\infty$ instead of 0, the fifth argument (for upper bounds) can be left empty `[]`, and the sixth argument is a string indicating the form of constraints, with letters 'U' for $\le$. So, Example 35.1.1 with Octave would be

```
glpk([-1; -1], [1 2; 3 1], [4; 5], [-Inf; -Inf], [], 'UU')
```

See `help glpk` for more.

As an additional example, let us solve the problem at the beginning of this section. The objective function $f(\mathbf{x}) = x_1 + 2x_2 - 5x_3$ was to be maximized, so we minimize $-f$ which means `c = [-1; -2; 5]`. The constraints $0 \le x_1 \le x_2 \le x_3 \le 10$ become $-x_1 \le 0$, $x_1 - x_2 \le 0$, $x_2 - x_3 \le 0$, and $x_3 \le 10$. Finally, $2x_3 - x_1 \ge 3$ becomes $x_1 - 2x_3 \le -3$. We arrive at a system of five linear constraints, summarized in `A, b` below:

```
c = [-1; -2; 5];
A = [-1 0 0; 1 -1 0; 0 1 -1; 0 0 1; 1 0 -2];
b = [0; 0; 0; 10; -3];
x = linprog(c, A, b);
fprintf('Found x = (%g, %g, %g) with c*x = %g \n', x, c'*x) ;
```

Note that `A*x` is the vector `[0; -1.5; 0; 1.5; -3]` where three of components are equal to the components of `b` and the others are strictly less. The equality means that the corresponding constraints, such as $-x_1 \le 0$, are **active** at the optimal point: the point pushes against them. The constraints $x_1 \le x_2$ and $x_3 \le 10$ are **inactive** at this point: they hold as strict inequalities. The solution would stay the same if the inactive constraints were removed.

The following terms may appear in error messages or warnings issued by `linprog` or its analogues in other software.

- A **feasible point** is a vector x that satisfies the constraints. The set of all such points is the feasible set.
- A problem is **infeasible** if there are no feasible points. (Example: $x_1 \ge 2$, $x_2 \ge 2$, $x_1 + x_2 \le 3$.) Such a problem has no solution.
- A problem is **unbounded** if the objective function takes arbitrarily small values on the feasible set (in case of minimization) or arbitrarily large values (in case of maximization). Such a problem has no solution.
- If the feasible set is nonempty and bounded, the problem surely has a solution. But it is possible to have an unbounded feasible set on which the objective function is bounded: recall Example 35.1.1.

Applied LP problems tend to have many variables and constraints, and a lot of work went into developing refining its solution methods such as the simplex method (walking along the edges of the feasible set) and interior point methods (moving through the interior of the feasible set toward its boundary).

35.2 Duality in linear programming

The main differences between the standard forms of LP problems can be summarized as:

- Constraints: Inequalities or Equations
- Variables: Real or Nonnegative

Other differences: minimization vs maximization, inequalities $\le$ vs $\ge$, are relatively minor: changing the signs of $A, \mathbf{b}$, or $\mathbf{c}$ takes care of those variations. So, one can imagine four major types of LP problems: IR, IN, ER, and EN. However, "ER" type problems are not really interesting, because a system of linear equations with real variables determines a hyperplane, and the restriction of a linear function to a hyperplane is either unbounded or constant. This leaves us with three types: IR, IN, EN, the examples of which are given by (35.1.1), (35.1.2), and (35.1.3), respectively.

This section will describe how each LP problem has a **dual** LP problem. Specifically:

- A problem of type IR has a dual of type EN
- A problem of type EN has a dual of type IR
- A problem of type IN has a dual of type IN

In general, the dual of the dual problem is the original (**primal**) problem. The IN-IN duality is symmetric while IR-EN duality is asymmetric.

Specifically, symmetric duality states that the dual of

$$\mathbf{c}^T\mathbf{x} \to \max \quad \text{where } A\mathbf{x} \le \mathbf{b}, \quad \mathbf{x} \ge 0 \tag{35.2.1}$$

is

$$\mathbf{b}^T\mathbf{y} \to \min \quad \text{where } A^T\mathbf{y} \ge \mathbf{c}, \quad \mathbf{y} \ge 0 \tag{35.2.2}$$

Asymmetric duality states that the dual of

$$\mathbf{c}^T\mathbf{x} \to \max \quad \text{where } A\mathbf{x} \le \mathbf{b} \tag{35.2.3}$$

is

$$\mathbf{b}^T\mathbf{y} \to \min \quad \text{where } A^T\mathbf{y} = \mathbf{c}, \quad \mathbf{y} \ge 0 \tag{35.2.4}$$

Note that in both cases, the variables and constraints trade places; the dual has as many constraints as the original problem had variables (not counting the nonnegativity requirements). Also, the dual of a maximization problem is a minimization problem (and the other way around).

Strong Duality Theorem: Suppose that the primal LP problem has an optimal solution $\mathbf{x}^*$. Then the dual problem has an optimal solution $\mathbf{y}^*$ where $\mathbf{c}^T\mathbf{x}^* = \mathbf{b}^T\mathbf{y}^*$. Thus, although these problems have different objective functions, both objective functions have the same extreme (optimal) value.

Every feasible point $\mathbf{x}$ of (35.2.1) gives a lower bound on its optimal value: that is, $\mathbf{c}^T\mathbf{x}^* \ge \mathbf{c}^T\mathbf{x}$. On the other hand, every feasible point $\mathbf{y}$ of (35.2.2) gives an *upper* bound on its optimal value: that is, $\mathbf{b}^T\mathbf{y}^* \le \mathbf{b}^T\mathbf{y}$. Since the optimal value (so far unknown) is the same for both problems, it is contained somewhere between known quantities $\mathbf{c}^T\mathbf{x}$ and $\mathbf{b}^T\mathbf{y}$. This leads to the idea of moving the points $\mathbf{x}, \mathbf{y}$ through the feasible sets so that the *duality gap* $\mathbf{b}^T\mathbf{y} - \mathbf{c}^T\mathbf{x}$ is decreased. Once the gap reaches zero, we have the solution. Or, if the gap becomes extremely small, we have a point $\mathbf{x}$ that is almost as good as the optimal one.

Without going into the detailed proof of the Strong Duality Theorem, here is a sketch for asymmetric duality. Suppose $\mathbf{x}^*$ is an optimal point for (35.2.3). If $\mathbf{y}$ is any feasible point of (35.2.4), then

$$\mathbf{c}^T\mathbf{x}^* = (A^T\mathbf{y})^T\mathbf{x}^* = \mathbf{y}^T(A\mathbf{x}^*) \le \mathbf{y}^T\mathbf{b} = \mathbf{b}^T\mathbf{y} \tag{35.2.5}$$

which shows that each value of the objective function of the dual problem gives an upper bound for the primal problem (this part is the "weak duality theorem"). It remains to show that there exists $\mathbf{y}$ for which equality holds in (35.2.5).

Imagine applying force $\mathbf{c}$ to the point $\mathbf{x}^*$: if it can be moved in this direction, the objective function will increase. Since such an increase is impossible, the force $\mathbf{c}$ must be balanced by reaction forces of the "walls", i.e., hyperplanes $(A\mathbf{x})_k \le b_k$, which represent constraints. Only the *active* constraints, those for which $(A\mathbf{x}^*)_k = b_k$, are relevant here, since $\mathbf{x}^*$ does not touch the hyperplanes of inactive constraints. The normal vectors of the constraint hyperplanes are given by the rows of A. Transposition makes $\mathbf{c}$ a linear combination of the columns of A^T with nonnegative coefficients; moreover, the coefficients of inactive constraints are zero. This means that $= A^T\mathbf{y}^*$ for some vector $\mathbf{y}^* \ge 0$ such that if $(A\mathbf{x}^*)_k < b_k$ (inactive constraint) then $y_k^* = 0$. Returning to (35.2.5) we see that with $\mathbf{y} = \mathbf{y}^*$ it becomes an equality.

35.3 Interpretation of duality in microeconomics

The following example is adapted from Wikipedia article Dual linear program, available under the Creative Commons Attribution-ShareAlike License.

Consider a farmer who may grow wheat and barley with the set provision of some L land, F fertilizer and P pesticide. To grow one unit of wheat, one unit of land, F_1 units of fertilizer and P_1 units of pesticide must be used. Similarly, to grow one unit of barley, one unit of land, F_2 units of fertilizer and P_2 units of pesticide must be used.

The primal problem would be the farmer deciding how much wheat (x_1) and barley (x_2) to grow if their sell prices are S_1 and S_2 per unit. The goal is to maximize the total revenue:

$$S_1x_1 + S_2x_2 \to \max$$

subject to constraints:

- $x_1 + x_2 \leq L$ (cannot use more land than available)
- $F_1x_1 + F_2x_2 \leq F$ (cannot use more fertilizer than available)
- $P_1x_1 + P_2x_2 \leq P$ (cannot use more pesticide than available)
- $x_1, x_2 \geq 0$ (cannot grow negative amounts)

For the dual problem assume that y unit prices for each of these means of production (inputs) are set by a planning board. The planning board's job is to minimize the total cost of procuring the set amounts of inputs while providing the farmer with a floor on the unit price of each of his crops (outputs), S_1 for wheat and S_2 for barley. This corresponds to the following problem: minimize the total cost

$$Ly_L + Fy_F + Py_P \to \min$$

subject to constraints:

- $y_L + F_1y_F + P_1y_P \geq S_1$ (the farmer must receive at least S_1 for each unit of wheat)
- $y_L + F_2y_F + P_2y_P \geq S_2$ (the farmer must receive at least S_2 for each unit of barley)
- $y_L, y_F, y_P \geq 0$ (prices cannot be negative)

In matrix form this becomes: minimize

$$\begin{pmatrix} L & F & P \end{pmatrix} \begin{pmatrix} y_L \\ y_F \\ y_P \end{pmatrix}$$

subject to:

$$\begin{pmatrix} 1 & F_1 & P_1 \\ 1 & F_2 & P_2 \end{pmatrix} \begin{pmatrix} y_L \\ y_F \\ y_P \end{pmatrix} \geq \begin{pmatrix} S_1 \\ S_2 \end{pmatrix}$$

and

$$\begin{pmatrix} y_L \\ y_F \\ y_P \end{pmatrix} \geq 0.$$

The primal problem deals with physical quantities. With all inputs available in limited quantities, and assuming the unit prices of all outputs is known, what quantities of outputs to produce so as to maximize total revenue? The dual problem deals with economic values. With floor guarantees on all output unit prices, and assuming the available quantity of all inputs is known, what input unit pricing scheme to set so as to minimize total expenditure?

To each variable in the primal space corresponds an inequality to satisfy in the dual space, both indexed by output type. To each inequality to satisfy in the primal space corresponds a variable in the dual space, both indexed by input type.

The coefficients that bound the inequalities in the primal space are used to compute the objective in the dual space, input quantities in this example. The coefficients used to compute the objective in the primal space bound the inequalities in the dual space, output unit prices in this example.

Both the primal and the dual problems make use of the same matrix. In the primal space, this matrix expresses the consumption of physical quantities of inputs necessary to produce set quantities of outputs. In the dual space, it expresses the creation of the economic values associated with the outputs from set input unit prices.

35.4 Duality in optimal transportation

The book *Topics in Optimal Transportation* by Cédric Villani presents the following illustration of duality, credited by the author to Luis Caffarelli. In this book the duality theorem is named after Leonid Kantorovich.

> Suppose for instance that you are both a mathematician and an industrialist, and want to transfer a huge amount of coal from your mines to your factories. You can hire trucks to do this transportation problem, but you have to pay them $c(X, Y)$ for each ton of coal which is transported from place X to place Y. Both the amount of coal which you can extract from each mine, and the amount which each factory should receive, are fixed. As you are trying to [...] minimize the price you have to pay, another mathematician comes to you and tells you:
>
> "My friend, let me handle this for you: I will ship all your coal with my own trucks and you won't have to worry about what goes where. I will just set a price $\varphi(X)$ for loading one ton of coal at place X, and a price $\psi(Y)$ for unloading it at destination Y. I will set the prices in such a way that your financial interest will be to let me handle all your transportation! Indeed, you can check very easily that for any X and Y, the sum $\varphi(X) + \psi(Y)$ will always be $\leq c(X, Y)$ (in order to achieve this goal, I am even ready to give financial compensations for some places, in the form of negative prices!)".
>
> Of course you accept the deal. Now, what Kantorovich's duality tells you is that if this shipper is clever enough, then he can arrange the prices in such a way that you will pay [...] as much as you would have been ready to spend by the other method.

What form of duality is this, and where do negative prices come from? This is an example of asymmetric duality, where the primal problem is of type EN (moving nonnegative amounts of coal, which must add up exactly to what is available or required), and the dual problem is of type IR (setting possibly negative prices, allowing for inequalities $\varphi(X) + \psi(Y) \leq c(X, Y)$). Here is a concrete example with two coal mines and three factories: the amounts of production, consumption, and transportation costs per unit are stated on the diagram.

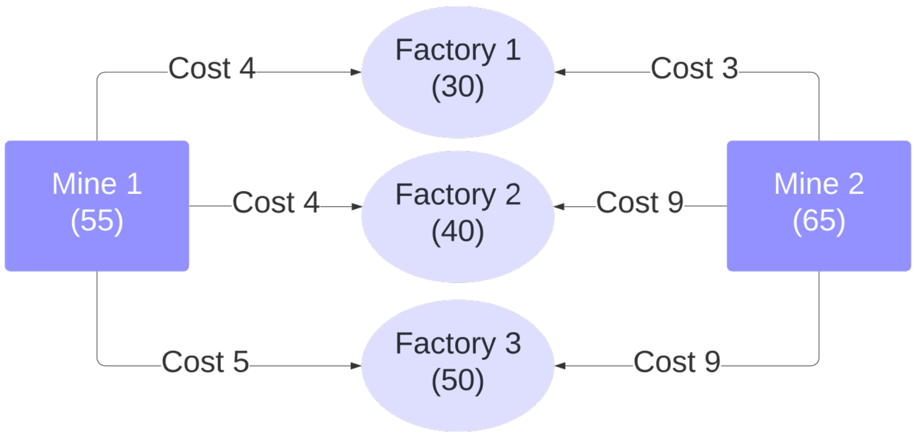

Figure 35.4.1 Optimal transportation problem with amounts and unit costs

Let x_1, x_2, x_3 be the amounts sent from Mine 1 to Factories 1, 2, 3. Also let x_4, x_5, x_6 be the amounts sent from Mine 2 to Factories 1, 2, 3. The constraints can be expressed in matrix form as $A\mathbf{x} = \mathbf{b}$ where

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 55 \\ 65 \\ 30 \\ 40 \\ 50 \end{pmatrix}$$

The coefficients of objective functions are the unit cost, $\mathbf{c}^T = (4, 4, 5, 3, 9, 9)$. And of course, $\mathbf{x} \geq 0$: even if we wanted to allow factories to send excess coal back to a mine, this would not fit the LP model since the cost of such transportation would not be negative.

To solve the linear program in Matlab, we use expanded syntax of `linprog`:

```
x = linprog(c, A, b, Aeq, beq, lb, ub);
```

where the additional parameters `Aeq`, `beq` express a linear system of *equality* constraints, and the vectors `lb`, `ub` provide lower and upper bounds on the variables. In our case, we do not have any inequality constraints so the second and third arguments are empty. The last one is omitted since we do not have upper bounds.

```
A = [1 1 1 0 0 0; 0 0 0 1 1 1; 1 0 0 1 0 0; 0 1 0 0 1 0; 0 0 1 0 0 1];
b = [55; 65; 30; 40; 50];
c = [4; 4; 5; 3; 9; 9];
x = linprog(c, [], [], A, b, zeros(6, 1));
fprintf('Found x = (%g, %g, %g, %g, %g, %g) with total cost %g \n', x, c'*x);
```

The solution is pictured below.

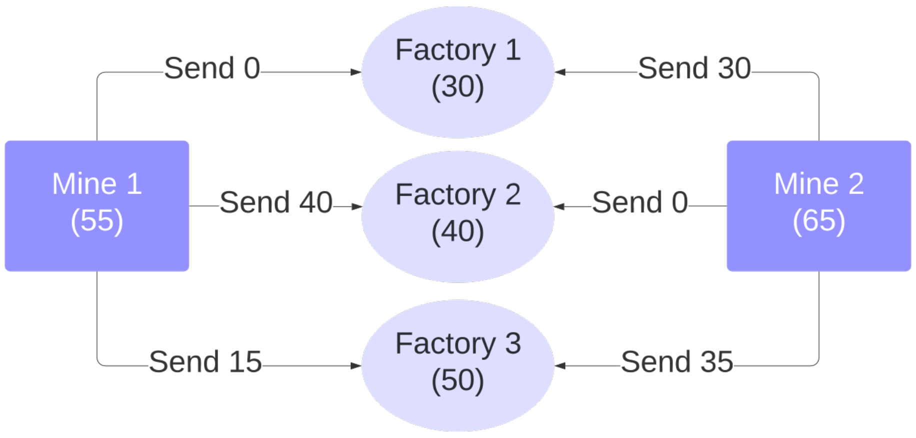

Figure 35.4.2 Solution of the transportation problem

According to Section 35.2, the dual problem is to maximize $\mathbf{b}^T\mathbf{y}$ subject to the constraint $A^T\mathbf{y} \le \mathbf{c}$. The vector $\mathbf{y}$ expresses the pricing policy of the second mathematician (or a shipping company): they charge y_1, y_2 to load a unit of coal at Mines 1, 2, and also charge y_3, y_4, y_5 to unload it at Factories 1, 2, 3. The constraint $A^T\mathbf{y} \le \mathbf{c}$ is what the company needs to win your business, and the amount maximized, $\mathbf{b}^T\mathbf{y}$, is their total revenue.

```
y = linprog(-b, A', c);
fprintf('Found y = (%g, %g, %g, %g, %g) with total revenue %g \n', y, b'*y);
```

The solution is pictured below. The costs $\mathbf{c}$ are included to demonstrate that the constraints are satisfied. The message "Found y = (4, 8, -5, 0, 1) with total revenue 640" shows that the company will subsidize unloading at Factory 1 in order to maximize its revenue while undercutting any competition.

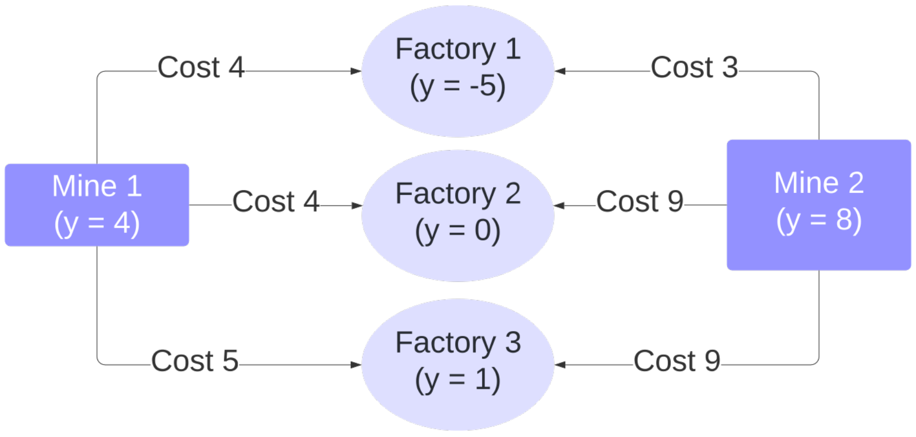

Figure 35.4.3 Solution of the dual problem

If we required the loading/unloading prices to be nonnegative, the transportation company would not be able to realize the same revenue.

```
y = linprog(-b, A', c, [], [], zeros(5, 1));
fprintf('Found y = (%g, %g, %g, %g, %g) with total revenue %g \n', y, b'*y);
```

"Found y = (0, 3, 0, 4, 5) with total revenue 605"

35.5 Homework

1. Use Matlab to solve the primal problem in Section 35.3 with the following data:

- Available amounts are $L = 900$, $F = 600$, $P = 250$
- Each unit of wheat requires $F_1 = 0.7$ units of fertilizer and $P_1 = 0.2$ units of pesticide
- Each unit of barley requires $F_2 = 0.5$ units of fertilizer and $P_2 = 0.3$ units of pesticide
- The sell prices are $S_1 = 60$ for wheat and $S_2 = 70$ for barley.

One way is to convert this LP problem to the form (35.1.1) (including the nonnegativity requirements in the system of linear inequalities) and apply `linprog(c, A, b)`. Another is to follow the examples in Section 35.4 (or read `help linprog`) and directly impose the lower bound of 0 on the variables.

In a comment, answer the questions: how should the land be used? What will the total revenue be? Will there be any leftover land, fertilizer, or pesticide, and if so, how much?

2. Use Matlab to solve the dual problem in Section 35.3 with the same data as in Exercise 35.5.1.

In a comment, answer the questions: What prices should the planning board set? What will the total cost be?

Chapter 36

Classification Problems

So far, our models for data were quantitative, trying to predict the amount of some quantity y. The problem changes when one tries to predict a categorical variable, which describes some feature in non-quantitative terms (an email may be spam or not spam, for example).

36.1 Classification and logistic regression

We focus on the case of two categories, represented by the binary outcome variable z attaining two possible values, 0 and 1. The model may involve one or more explanatory variables x_k, and the goal is to predict the outcome based on these variables. Some examples:

- Explanatory variables: temperature, humidity, atmospheric pressure. Outcome: rain (1) or no rain (0).
- Explanatory variables: patient's weight, height, age, activity level. Outcome: has diabetes (1) or not (0).

In the examples such as these, it is understood that the prediction may be far from certain. A way to think of such a model as assigning a probability of outcome (1), based on the data available. This leads to the idea of a model being a function that takes values between 0 and 1. The standard logistic function (Section 29.4) has this property:

$$\sigma(x) = \frac{1}{1+\exp(-x)}$$

In order to include several explanatory variables $x_1, \dots, x_m$ in this model, we use the composite function

$$h_{\mathbf{c}}(\mathbf{x}) = \sigma\left(\sum_{k=1}^{m} c_k x_k\right) = \frac{1}{1+\exp\left(-\sum_{k=1}^{m} c_k x_k\right)} \tag{36.1.1}$$

where the coefficients $c_1, \dots, c_m$ are to be determined by training the model on some data set with known outcomes.

In Chapter 29 our training method was to minimize the sum of squares

$$S(\mathbf{c}) = \sum_{j=1}^{n} (z_j - h_{\mathbf{c}}(\mathbf{x}_j))^2$$

However, in the context of probability there is a more natural method: maximizing the **likelihood** function. Think of $h_{\mathbf{c}}(\mathbf{x}_j)$ as the probability that $z_j = 1$, and therefore $1 - h_{\mathbf{c}}(\mathbf{x}_j)$ is the probability that $z_j = 0$. Assuming the independence of outcomes, the probability of observing the data that was actually observed is

$$L(\mathbf{c}) = \prod_{z_j=1} h_{\mathbf{c}}(\mathbf{x}_j) \cdot \prod_{z_j=0} (1 - h_{\mathbf{c}}(\mathbf{x}_j)) \tag{36.1.2}$$

Since the outcomes z_j already occurred, this expression is not really the probability of a random event; it is the **likelihood** of the parameters being $\mathbf{c}$. We want to maximize this likelihood. Since the product may be very small when there are many observations, it is easier to maximize the **log-likelihood**, the logarithm of (36.1.2):

$$\log L(\mathbf{c}) = \sum_{z_j=1} \log h_{\mathbf{c}}(\mathbf{x}_j) + \sum_{z_j=0} \log(1 - h_{\mathbf{c}}(\mathbf{x}_j)) \tag{36.1.3}$$

Once we find $\mathbf{c}$ which maximizes $\log L$, the model can then be used to make predictions about unobserved outcomes on the basis of explanatory variables $\mathbf{x}$. If $h_{\mathbf{c}}(\mathbf{x})$ is close to 1, we expect the outcome $z = 1$; it is close to 0, we expect the outcome $z = 0$. The quantity $h_{\mathbf{c}}(\mathbf{x})$ may be interpreted as the probability of $z = 1$, predicted by the model.

Unless the explanatory variables $\mathbf{x}$ are **centered** (have zero mean by design), the exponential should include a constant term, as shown in the following example.

Example 36.1.1 Smoking, age and blood pressure. The following data, taken from Matlab's built-in `patient` data sample (reference), records the ages and systolic pressure of 10 smokers ($z = 1$) and 12 nonsmokers ($z = 0$). Train a logistic regression model on this data.

```
age1 = [38 33 39 48 32 27 44 28 30 45];
sys1 = [124 130 130 130 124 123 128 129 127 134];
age0 = [43 38 40 49 46 40 28 31 45 42 25 36];
sys0 = [109 125 117 122 121 115 115 118 114 115 127 114];
```

Then test the model on the data set, generating predictions for the people based on their age and systolic pressure.

```
age = [38 45 30 48 48 25 44 49 45 48];
sys = [138 124 130 123 129 128 124 119 136 114];
```

Finally, compare the model's prediction with actual smoker status.

```
actual = [1 0 0 0 0 1 1 0 1 0];
```

Solution. We set up the log-likelihood function `LogL`, maximize it, and use the optimal parameters `cc` to predict the status of the 10 people who were not a part of the training set.

```
LogL = @(c) sum(log(1./(1+exp(-c(1)*age1-c(2)*sys1-c(3))))) + sum(log(1 - 1./(1+exp(-c(1)*age0-c
cc = fminsearch(@(c) -LogL(c), [0; 0; 0]);

age = [38 45 30 48 48 25 44 49 45 48];
sys = [138 124 130 123 129 128 124 119 136 114];
prediction = 1./(1+exp(-cc(1)*age-cc(2)*sys-cc(3)));
actual = [1 0 0 0 0 1 1 0 1 0];
disp([prediction' actual']);
```

Excluding the cases where prediction is a number close to 0.5 (which should be considered a "don't know" answer), the model got 6 out of 8 right. □

36.2 Classification and linear programming

Linear programming offers a very different, geometric approach to classification. Think of observations with m explanatory variables as points in $\mathbb{R}^m$, which are colored (say, red or blue) according to their category. Our goal is to find a plane (or surface) that separates the points of different colors as cleanly as possible.

To help with visualization, consider the case $m = 2$ when the explanatory variables (x_1, x_2) are more conveniently labeled (x, y). We are looking for a line $y = ax + b$ such that the points of one category are above it, and the points of other category are below it. If such a line exists, it is not unique, so we want to choose it to maximize the width of separation. That is, we look for maximal $w \geq 0$ such that $y_j \geq ax_j + b + w$ in the first set (above the line) and $y_j \leq ax_j + b - w$ in the second set (below the line). This is a linear programming problem, with three variables a, b, w, objective function w, and the linear constraints stated above.

The same idea applies to separation by a curve (for example, a polynomial).

Example 36.2.1 Complete nonlinear separation. Find and plot the best parabola separating the points $(2, 3)$, $(4, 6)$, and $(7, 4)$ (the "top" group) from $(3, 4)$ and $(5, 5)$ (the "bottom" group).

Solution. We maximize w subject to constraints $y_j \geq ax_j^2 + bx_j + c + w$ in the first group and $y_j \leq ax_j^2 + bx_j + c - w$ in the second. Rewrite the constraints as

$$ax_j^2 + bx_j + c + w \leq y_j$$

in the first group and

$$-ax_j^2 - bx_j - c + w \leq -y_j$$

in the second. Also, $w \geq 0$ is required by the geometric meaning of w as the "width" of separation. Given the vectors `xt, yt` describing the first group and `xb, yb` for the second, we can form a system of linear inequalities $A\mathbf{x} \leq \mathbf{b}$ as below

```
xt = [2; 4; 7];
yt = [3; 6; 4];
xb = [3; 5];
yb = [4; 5];
A = [xt.^2 xt xt.^0 xt.^0; -xb.^2 -xb -xb.^0 xb.^0; 0 0 0 -1];
b = [yt; -yb; 0];
opt = linprog([0; 0; 0; -1], A, b);

t = linspace(min([xt; xb]), max([xt; xb]), 1000);
y = opt(1)*t.^2 + opt(2)*t + opt(3);
plot(xt, yt, 'b*', xb, yb, 'r*', t, y, 'k')
```

Instead of including $-w \leq 0$ as one of the inequalities in the system $A\mathbf{x} \leq b$, we can impose this constraint separately as a lower bound in `linprog`.

```
A = [xt.^2 xt xt.^0 xt.^0; -xb.^2 -xb -xb.^0 xb.^0];
b = [yt; -yb];
opt = linprog([0; 0; 0; -1], A, b, [], [], [-Inf; -Inf; -Inf; 0]);
```

The result is the same. With either version of the code, if the data points are changed so that there is no parabola that separates them completely, we get the message "No feasible solution found. Linprog stopped because no point satisfies the constraints." □

Complete separation of the categories may be impossible (which makes the linear programming problem infeasible, as Matlab would report). Perhaps some data points have incorrect measurement values, or are mislabeled by category, or perhaps (quite likely) one simply cannot predict the outcome with certainty, based on the explanatory variables we have. In such a case we can still look for a separating line or curve, but the separation will be *incomplete*, with some points left on the “wrong” side.

We look for a line that minimizes the sum of penalties, the penalty being the amount by which a point falls into the wrong group. Formally, we minimize $\sum p_j$ where $p_j \ge 0$ are the penalties, subject to constraints $y_j \ge ax_j + b - p_j$ for points in the “top” category and $y_j \le ax_j + b + p_j$ for points in the “bottom” category. This is still a linear programming problem, but there are more variables now: $a, b, p_1, \dots, p_n$, and the objective function is $\sum p_j$.

Example 36.2.2 Incomplete linear separation. Find and plot the best line that separates (probably not completely) 8 random points from 7 other random points, which are generated as follows:

```
xt = randn(8, 1);
yt = randn(8, 1) + 1;
xb = randn(7, 1);
yb = randn(7, 1);
```

Solution. We minimize $\sum p_j$ subject to constraints $y_j \ge ax_j + b - p_j$ in the first group and $y_j \le ax_j + b + p_j$ in the second. Rewrite the constraints as

$$ax_j + b - p_j \le y_j$$

in the first group and

$$-ax_j - b - p_j \le -y_j$$

in the second. We also require $p_j \ge 0$. Given the vectors `xt, yt` describing the first group and `xb, yb` for the second, we can form a system of linear inequalities $A\mathbf{x} \le \mathbf{b}$ and separately impose the nonnegativity requirement on p_j (though not on the coefficients a, b of the line).

```
A = [xt xt.^0; -xb -xb.^0];
n = numel(xt) + numel(xb);
A = [A -eye(n)];
b = [yt; -yb];
opt = linprog([0; 0; ones(n, 1)], A, b, [], [], [-Inf; -Inf; zeros(n, 1)]);

t = linspace(min([xt; xb]), max([xt; xb]), 1000);
y = opt(1)*t + opt(2);
plot(xt, yt, 'b*', xb, yb, 'r*', t, y, 'k')
```

The reason that the presence of penalties changes the matrix as `A = [A -eye(n)];` is that we subtract p_1 from the first inequality, p_2 from the second, and so on. The penalties make it possible for all the constraints to be satisfied, but their sum $p_1 + \cdots + p_n$ needs to be minimized. In this example, the variables are $a, b, p_1, \dots, p_n$ and accordingly, the coefficients of the objective function are $0, 0, 1, \dots, 1$. □

36.3 Homework

1. Modify Example 36.2.2 to separate the points (probably incompletely) by a parabola rather than by a line.