Chapter I

Linear Equations

I.1. Solving Linear Equations

Prerequisites and Learning Goals

From your work in previous courses, you should be able to

- Write a system of linear equations using matrix notation.
- Use Gaussian elimination to bring a system of linear equations into upper triangular form and reduced row echelon form (rref).
- Determine whether a system of equations has a unique solution, infinitely many solutions or no solutions, and compute all solutions if they exist; express the set of all solutions in parametric form.
- Recognize when matrix sums and matrix products are defined, and to compute them.
- Compute the inverse of a matrix when it exists, use the inverse to solve a system of equations, describe for what systems of equations this is possible.
- Find the transpose of a matrix.
- Interpret a matrix as a linear transformation acting on vectors.

After completing this section, you should be able to

- Write down the definition and explain the meaning of the standard Euclidean norm, the 1-norm and the infinity norm of a vector; compute norms by hand or using the MATLAB/Octave command `norm`.
- Calculate the Hilbert-Schmidt norm of a matrix.
- Define the matrix norm of a matrix; describe the connection between the matrix norm and how a matrix stretches the length of vectors and express it in a mathematical form; explain what range of values the matrix norm can take; compute the matrix norm of a diagonal matrix and, given sufficient information, of other types of matrices.
- Find solutions to linear equations of the form $A\mathbf{x} = \mathbf{b}$ by executing and interpreting the output of the MATLAB/Octave commands `rref` and `\`.
- Define the condition number of a matrix and its relation to the matrix norm; compute it by hand when possible or using the MATLAB/Octave command `cond`.
- Discuss the sensitivity of a linear equation $A\mathbf{x} = \mathbf{b}$ by relating changes in the solution $\mathbf{x}$ to small changes in $\mathbf{b}$; define the relative error, explain why the condition number is useful in making estimates of relative errors in the solution $\mathbf{x}$, and use it to make such estimates.

I.1.1. Review: Systems of linear equations

The first part of the course is about systems of linear equations. You will have studied such systems in a previous course, and should remember how to find solutions (when they exist) using Gaussian elimination.

Many practical problems can be solved by turning them into a system of linear equations. In this chapter we will study a few examples: the problem of finding a function that interpolates a collection of given points, and the approximate solutions of differential equations. In practical problems, the question of existence of solutions, although important, is not the end of the story. It turns out that some systems of equations, even though they may have a unique solution, are very sensitive to changes in the coefficients. This makes them very difficult to solve reliably. We will see some examples of such ill-conditioned systems, and learn how to recognize them using the condition number of a matrix.

Recall that a system of linear equations, like this system of 2 equations in 3 unknowns

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

can be written as a matrix equation

$$\begin{bmatrix} 1 & 2 & 1 \\ 1 & -5 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

A general system of m linear equations in n unknowns can be written as

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

where A is an given $m \times n$ (m rows, n columns) matrix, $\mathbf{b}$ is a given m-component vector, and $\mathbf{x}$ is the n-component vector of unknowns.

A system of linear equations may have no solutions, a unique solutions, or infinitely many solutions. This is easy to see when there is only a single variable x, so that the equation has the form

$$ax = b$$

where a and b are given numbers. The solution is easy to find if $a \neq 0$: $x = b/a$. If $a = 0$ then the equation reads $0x = b$. In this case, the equation either has no solutions (when $b \neq 0$) or infinitely many (when $b = 0$), since in this case every x is a solution.

To solve a general system $A\mathbf{x} = \mathbf{b}$, form the augmented matrix $[A|b]$ and use Gaussian elimination to reduce the matrix to reduced row echelon form. This reduced matrix (which represents a system of linear equations that has exactly the same solutions as the original system) can be used to decide whether solutions exist, and to find them. If you don't remember this procedure, you should review it.

In the example above, the augmented matrix is

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

The reduced row echelon form is

$$\left[\begin{array}{ccc|c} 1 & 0 & 1 & 2/7 \\ 0 & 1 & 0 & -1/7 \end{array}\right],$$

which leads to a family of solutions (one for each value of the parameter s)

$$\mathbf{x} = \begin{bmatrix} 2/7 \\ -1/7 \\ 0 \end{bmatrix} + s \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}.$$

I.1.2. Solving a non-singular system of n equations in n unknowns

Let's start with a system of equations where the number of equations is the same as the number of unknowns. Such a system can be written as a matrix equation

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

where A is a square matrix, $\mathbf{b}$ is a given vector, and $\mathbf{x}$ is the vector of unknowns we are trying to find. When A is non-singular (invertible) there is a unique solution. It is given by $\mathbf{x} = A^{-1}\mathbf{b}$, where A^{-1} is the inverse matrix of A. Of course, computing A^{-1} is not the most efficient way to solve a system of equations.

For our first introduction to MATLAB/Octave, let's consider an example:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix}.$$

First, we define the matrix A and the vector $\mathbf{b}$ in MATLAB/Octave. Here is the input (after the prompt symbol >) and the output (without a prompt symbol).

```
>A=[1 1 1;1 1 -1;1 -1 1]

A =

   1   1   1
   1   1  -1
   1  -1   1

>b=[3;1;1]
```

```
b =

   3
   1
   1
```

Notice that the entries on the same row are separated by spaces (or commas) while rows are separated by semicolons. In MATLAB/Octave, column vectors are n by 1 matrices and row vectors are 1 by n matrices. The semicolons in the definition of `b` make it a column vector. In MATLAB/Octave, `X'` denotes the transpose of `X`. Thus we get the same result if we define `b` as

```
>b=[3 1 1]'

b =

   3
   1
   1
```

The solution can be found by computing the inverse of A and multiplying

```
>x = A^(-1)*b

x =

   1
   1
   1
```

However if A is a large matrix we don't want to actually calculate the inverse. The syntax for solving a system of equations efficiently is

```
>x = A\b

x =

   1
   1
   1
```

If you try this with a singular matrix A, MATLAB/Octave will complain and print an warning message. If you see the warning, the answer is not reliable! You can always check to see that $\mathbf{x}$ really is a solution by computing $A\mathbf{x}$.

```
>A*x

ans =

   3
   1
   1
```

As expected, the result is $\mathbf{b}$.

By the way, you can check to see how much faster `A\b` is than `A^(-1)*b` by using the functions `tic()` and `toc()`. The function `tic()` starts the clock, and `toc()` stops the clock and prints the elapsed time. To try this out, let's make `A` and `b` really big with random entries.

```
A=rand(1000,1000);
b=rand(1000,1);
```

Here we are using the MATLAB/Octave command `rand(m,n)` that generates an $m \times n$ matrix with random entries chosen between 0 and 1. Each time `rand` is used it generates new numbers.

Notice the semicolon `;` at the end of the inputs. This suppresses the output. Without the semicolon, MATLAB/Octave would start writing the 1,000,000 random entries of A to our screen! Now we are ready to time our calculations.

```
tic();A^(-1)*b;toc();

Elapsed time is 44 seconds.

tic();A\b;toc();

Elapsed time is 13.55 seconds.
```

So we see that `A\b` quite a bit faster.

I.1.3. Reduced row echelon form

How can we solve $A\mathbf{x} = \mathbf{b}$ when A is singular, or not a square matrix (that is, the number of equations is different from the number of unknowns)? In your previous linear algebra course you learned how to use elementary row operations to transform the original system of equations to an upper triangular system. The upper triangular system obtained this way has exactly the same solutions as the original system. However, it is much easier to solve. In practice, the row operations are performed on the augmented matrix $[A|\mathbf{b}]$.

If efficiency is not an issue, then addition row operations can be used to bring the system into reduced row echelon form. In the this form, the pivot columns have a 1 in the pivot position and zeros elsewhere. For example, if A is a square non-singular matrix then the reduced row echelon form of $[A|\mathbf{b}]$ is $[I|\mathbf{x}]$, where I is the identity matrix and $\mathbf{x}$ is the solution.

In MATLAB/Octave you can compute the reduced row echelon form in one step using the function `rref()`. For the system we considered above we do this as follows. First define `A` and `b` as before. This time I'll suppress the output.

```
>A=[1 1 1;1 1 -1;1 -1 1];

>b=[3 1 1]';
```

In MATLAB/Octave, the square brackets `[ ... ]` can be used to construct larger matrices from smaller building blocks, provided the sizes match correctly. So we can define the augmented matrix C as

```
>C=[A b]

C =

   1   1   1   3
   1   1  -1   1
   1  -1   1   1
```

Now we compute the reduced row echelon form.

```
>rref(C)

ans =

   1   0   0   1
   0   1  -0   1
   0   0   1   1
```

The solution appears on the right.

Now let's try to solve $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

This time the matrix A is singular and doesn't have an inverse. Recall that the determinant of a singular matrix is zero, so we can check by computing it.

```
>A=[1 2 3; 4 5 6; 7 8 9];
>det(A)

ans = 0
```

However we can still try to solve the equation $A\mathbf{x} = \mathbf{b}$ using Gaussian elimination.

```
>b=[1 1 1]';
>rref([A b])

ans =

   1.00000   0.00000  -1.00000  -1.00000
   0.00000   1.00000   2.00000   1.00000
   0.00000   0.00000   0.00000   0.00000
```

Letting $x_3 = s$ be a parameter, and proceeding as you learned in previous courses, we arrive at the general solution

$$\mathbf{x} = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix} + s \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}.$$

On the other hand, if

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix},$$

then

```
>rref([1 2 3 1;4 5 6 1;7 8 9 0])

ans =

   1.00000   0.00000  -1.00000   0.00000
   0.00000   1.00000   2.00000   0.00000
   0.00000   0.00000   0.00000   1.00000
```

tells us that there is no solution.

I.1.4. Gaussian elimination steps using MATLAB/Octave

If `C` is a matrix in MATLAB/Octave, then `C(1,2)` is the entry in the 1st row and 2nd column. The whole first row can be extracted using `C(1,:)` while `C(:,2)` yields the second column. Finally we can pick out the submatrix of `C` consisting of rows 1-2 and columns 2-4 with the notation `C(1:2,2:4)`.

Let's illustrate this by performing a few steps of Gaussian elimination on the augmented matrix from our first example. Start with

```
C=[1 1 1 3; 1 1 -1 1; 1 -1 1 1];
```

The first step in Gaussian elimination is to subtract the first row from the second.

```
>C(2,:)=C(2,:)-C(1,:)

C =

   1   1   1   3
   0   0  -2  -2
   1  -1   1   1
```

Next, we subtract the first row from the third.

```
>C(3,:)=C(3,:)-C(1,:)

C =

   1   1   1   3
   0   0  -2  -2
   0  -2   0  -2
```

To bring the system into upper triangular form, we need to swap the second and third rows. Here is the MATLAB/Octave code.

```
>temp=C(3,:);C(3,:)=C(2,:);C(2,:)=temp

C =

   1   1   1   3
   0  -2   0  -2
   0   0  -2  -2
```

I.1.5. Norms for a vector

Norms are a way of measuring the size of a vector. They are important when we study how vectors change, or want to know how close one vector is to another. A vector may have many components and it might happen that some are big and some are small. A norm is a way of capturing information about the size of a vector in a single number. There is more than one way to define a norm.

In your previous linear algebra course, you probably have encountered the most common norm, called the Euclidean norm (or the 2-norm). The word norm without qualification usually refers to this norm. What is the Euclidean norm of the vector

$$\mathbf{a} = \begin{bmatrix} -4 \\ 3 \end{bmatrix}?$$

When you draw the vector as an arrow on the plane, this norm is the Euclidean distance between the tip and the tail. This leads to the formula

$$\|\mathbf{a}\| = \sqrt{(-4)^2 + 3^2} = 5.$$

This is the answer that MATLAB/Octave gives too:

```
> a=[-4 3]

a =
  -4   3

> norm(a)
ans =  5
```

The formula is easily generalized to n dimensions. If $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ then

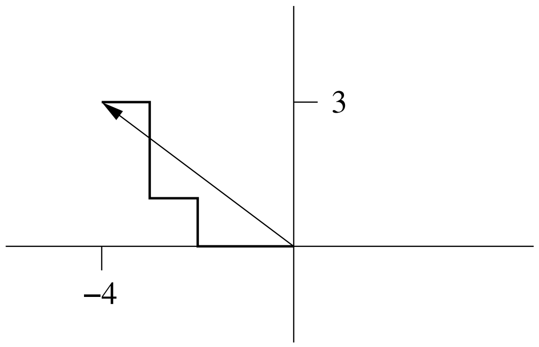
$$\|\mathbf{x}\| = \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2}.$$

The absolute value signs in this formula, which might seem superfluous, are put in to make the formula correct when the components are complex numbers. So, for example

$$\left\| \begin{bmatrix} i \\ 1 \end{bmatrix} \right\| = \sqrt{|i|^2 + |1|^2} = \sqrt{1+1} = \sqrt{2}.$$

Does MATLAB/Octave give this answer too?

There are situations where other ways of measuring the norm of a vector are more natural. Suppose that the tip and tail of the vector $\mathbf{a} = [-4, 3]^T$ are locations in a city where you can only walk along the streets and avenues.



If you defined the norm to be the shortest distance that you can walk to get from the tail to the tip, the answer would be

$$\|\mathbf{a}\|_1 = |-4| + |3| = 7.$$

This norm is called the 1-norm and can be calculated in MATLAB/Octave by adding 1 as an extra argument in the norm function.

```
> norm(a,1)
ans =  7
```

The 1-norm is also easily generalized to n dimensions. If $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ then

$$\|\mathbf{x}\|_1 = |x_1| + |x_2| + \cdots + |x_n|.$$

Another norm that is often used measures the largest component in absolute value. This norm is called the infinity norm. For $\mathbf{a} = [-4, 3]^T$ we have

$$\|\mathbf{a}\|_\infty = \max\{|-4|, |3|\} = 4.$$

To compute this norm in MATLAB/Octave we use `inf` as the second argument in the norm function.

```
> norm(a,inf)
ans =  4
```

Here are three properties that the norms we have defined all have in common:

1. For every vector $\mathbf{x}$ and every number s, $\|s\mathbf{x}\| = |s|\|\mathbf{x}\|$.
2. The only vector with norm zero is the zero vector, that is, $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = \mathbf{0}$
3. For all vectors $\mathbf{x}$ and $\mathbf{y}$, $\|\mathbf{x}+\mathbf{y}\| \le \|\mathbf{x}\| + \|\mathbf{y}\|$. This inequality is called the triangle inequality. It says that the length of the longest side of a triangle is smaller than the sum of the lengths of the two shorter sides.

What is the point of introducing many ways of measuring the length of a vector? Sometimes one of the non-standard norms has natural meaning in the context of a given problem. For example, when we study stochastic matrices, we will see that multiplication of a vector by a stochastic matrix decreases the 1-norm of the vector. So in this situation it is natural to use 1-norms. However, in this course we will almost always use the standard Euclidean norm. If $\mathbf{v}$ a vector then $\|\mathbf{v}\|$ (without any subscripts) will always denote the standard Euclidean norm.

I.1.6. Matrix norms

Just as for vectors, there are many ways to measure the size of a matrix A.

For a start we could think of a matrix as a vector whose entries just happen to be written in a box, like

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

rather than in a row, like

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 2 \end{bmatrix}.$$

Taking this point of view, we would define the norm of A to be $\sqrt{1^2 + 2^2 + 0^2 + 2^2} = 3$. In fact, the norm computed in this way is sometimes used for matrices. It is called the Hilbert-Schmidt norm. For a general matrix $A = [a_{i,j}]$, the formula for the Hilbert-Schmidt norm is

$$\|A\|_{HS} = \sqrt{\sum_i \sum_j |a_{i,j}|^2}.$$

The Hilbert-Schmidt norm does measure the size of matrix in some sense. It has the advantage of being easy to compute from the entries $a_{i,j}$. But it is not closely tied to the action of A as a linear transformation.

When A is considered as a linear transformation or operator, acting on vectors, there is another norm that is more natural to use.

Starting with a vector $\mathbf{x}$ the matrix A transforms it to the vector $A\mathbf{x}$. We want to say that a matrix is big if increases the size of vectors, in other words, if $\|A\mathbf{x}\|$ is big compared to $\|\mathbf{x}\|$. So it is natural to consider the stretching ratio $\|A\mathbf{x}\|/\|\mathbf{x}\|$. Of course, this ratio depends on $\mathbf{x}$, since some vectors get stretched more than others by A. Also, the ratio is not defined if $\mathbf{x} = \mathbf{0}$. But in this case $A\mathbf{x} = \mathbf{0}$ too, so there is no stretching.

We now define the matrix norm of A to be the largest of these ratios,

$$\|A\| = \max_{\mathbf{x}:\|\mathbf{x}\|\neq 0} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}.$$

This norm measures the maximum factor by which A can stretch the length of a vector. It is sometimes called the operator norm.

Since $\|A\|$ is defined to be the maximum of a collection of stretching ratios, it must be bigger than or equal to any particular stretching ratio. In other words, for any non zero vector $\mathbf{x}$ we know $\|A\| \geq \|A\mathbf{x}\|/\|\mathbf{x}\|$, or

$$\|A\mathbf{x}\| \leq \|A\|\|\mathbf{x}\|.$$

This is how the matrix norm is often used in practice. If we know $\|\mathbf{x}\|$ and the matrix norm $\|A\|$, then we have an upper bound on the norm of $A\mathbf{x}$.

In fact, the maximum of a collection of numbers is the smallest number that is larger than or equal to every number in the collection (draw a picture on the number line to see this), the matrix norm $\|A\|$ is the smallest number that is bigger than $\|A\mathbf{x}\|/\|\mathbf{x}\|$ for every choice of non-zero $\mathbf{x}$. Thus $\|A\|$ is the smallest number C for which

$$\|A\mathbf{x}\| \le C\|\mathbf{x}\|$$

for every $\mathbf{x}$.

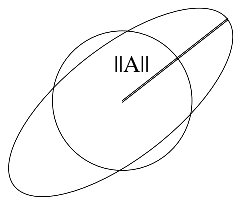
An equivalent definition for $\|A\|$ is

$$\|A\| = \max_{\mathbf{x}:\|\mathbf{x}\|=1} \|A\mathbf{x}\|.$$

Why do these definitions give the same answer? The reason is that the quantity $\|A\mathbf{x}\|/\|\mathbf{x}\|$ does not change if we multiply $\mathbf{x}$ by a non-zero scalar (convince yourself!). So, when calculating the maximum over all non-zero vectors in the first expression for $\|A\|$, all the vectors pointing in the same direction will give the same value for $\|A\mathbf{x}\|/\|\mathbf{x}\|$. This means that we need only pick one vector in any given direction, and might as well choose the unit vector. For this vector, the denominator is equal to one, so we can ignore it.

Here is another way of saying this. Consider the image of the unit sphere under A. This is the set of vectors $\{A\mathbf{x} : \|\mathbf{x}\| = 1\}$ The length of the longest vector in this set is $\|A\|$.

The picture below is a sketch of the unit sphere (circle) in two dimensions, and its image under $A = \begin{bmatrix} 1 & 2 \\ 0 & 2 \end{bmatrix}$. This image is an ellipse.



The norm of the matrix is the distance from the origin to the point on the ellipse farthest from the origin. In this case this turns out to be $\|A\| = \sqrt{9/2 + (1/2)\sqrt{65}}$.

It's hard to see how this expression can be obtained from the entries of the matrix. There is no easy formula. However, if A is a diagonal matrix the norm is easy to compute.

To see this, let's consider a diagonal matrix

$$A = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

If

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

then

$$A\mathbf{x} = \begin{bmatrix} 3x_1 \\ 2x_2 \\ x_3 \end{bmatrix}$$

so that

$$\begin{aligned} \|A\mathbf{x}\|^2 &= |3x_1|^2 + |2x_2|^2 + |x_3|^2 \\ &= 3^2|x_1|^2 + 2^2|x_2|^2 + |x_3|^2 \\ &\le 3^2|x_1|^2 + 3^2|x_2|^2 + 3^2|x_3|^2 \\ &= 3^2\|\mathbf{x}\|^2. \end{aligned}$$

This implies that for any unit vector $\mathbf{x}$

$$\|A\mathbf{x}\| \le 3$$

and taking the maximum over all unit vectors $\mathbf{x}$ yields $\|A\| \le 3$. On the other hand, the maximum of $\|A\mathbf{x}\|$ over all unit vectors $\mathbf{x}$ is larger than the value of $\|A\mathbf{x}\|$ for any particular unit vector. In particular, if

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

then

$$\|A\| \ge \|A\mathbf{e}_1\| = 3.$$

Thus we see that

$$\|A\| = 3.$$

In general, the matrix norm of a diagonal matrix with diagonal entries $\lambda_1, \lambda_2, \cdots, \lambda_n$ is the largest value of $|\lambda_k|$.

The MATLAB/Octave code for a diagonal matrix with diagonal entries 3, 2 and 1 is `diag([3 2 1])` and the expression for the norm of A is `norm(A)`. So for example

```
>norm(diag([3 2 1]))

ans =  3
```

I.1.7. Condition number

Let's return to the situation where A is a square matrix and we are trying to solve $A\mathbf{x} = \mathbf{b}$. If A is a matrix arising from a real world application (for example if A contains values measured in an experiment) then it will almost never happen that A is singular. After all, a tiny change in any of the entries of A can change a singular matrix to a non-singular one. What is much more likely to happen is that A is *close* to being singular. In this case A^{-1} will still exist, but will have some enormous entries. This means that the solution $\mathbf{x} = A^{-1}\mathbf{b}$ will be very sensitive to the tiniest changes in $\mathbf{b}$ so that it might happen that round-off error in the computer completely destroys the accuracy of the answer.

To check whether a system of linear equations is well-conditioned, we might therefore think of using $\|A^{-1}\|$ as a measure. But this isn't quite right, since we actually don't care if $\|A^{-1}\|$ is large, provided it stretches each vector about the same amount. For example, if we simply multiply each entry of A by 10^{-6} the size of A^{-1} will go way up, by a factor of 10^6, but our ability to solve the system accurately is unchanged. The new solution is simply 10^6 times the old solution, that is, we have simply shifted the position of the decimal point.

It turns out that for a square matrix A, the ratio of the largest stretching factor to the smallest stretching factor of A is a good measure of how well conditioned the system of equation $A\mathbf{x} = \mathbf{b}$ is. This ratio is called the condition number and is denoted $\mathrm{cond}(A)$.

Let's first compute an expression for $\mathrm{cond}(A)$ in terms of matrix norms. Then we will explain why it measures the conditioning of a system of equations.

We already know that the largest stretching factor for a matrix A is the matrix norm $\|A\|$. So let's look at the smallest streching factor. We might as well assume that A is invertible. Otherwise, there is a non-zero vector that A sends to zero, so that the smallest stretching factor is 0 and the condition number is infinite.

$$\begin{aligned}
\min_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} &= \min_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|}{\|A^{-1}A\mathbf{x}\|} \\
&= \min_{\mathbf{y}\neq 0} \frac{\|\mathbf{y}\|}{\|A^{-1}\mathbf{y}\|} \\
&= \frac{1}{\displaystyle\max_{\mathbf{y}\neq 0} \frac{\|A^{-1}\mathbf{y}\|}{\|\mathbf{y}\|}} \\
&= \frac{1}{\|A^{-1}\|}.
\end{aligned}$$

Here we used the fact that if $\mathbf{x}$ ranges over all non-zero vectors so does $\mathbf{y} = A\mathbf{x}$ and that the minimum of a collection of positive numbers is one divided by the maximum of their reciprocals. Thus the smallest stretching factor for A is $1/\|A^{-1}\|$. This leads to the following formula for the condition number of an invertible matrix:

$$\mathrm{cond}(A) = \|A\|\|A^{-1}\|.$$

In our applications we will use the condition number as a measure of how well we can solve the equations that come up accurately.

Now, let us try to see why the condition number of A is a good measure of how well we can solve the equations $A\mathbf{x} = \mathbf{b}$ accurately.

Starting with $A\mathbf{x} = \mathbf{b}$ we change the right side to $\mathbf{b}' = \mathbf{b} + \Delta\mathbf{b}$. The new solution is

$$\mathbf{x}' = A^{-1}(\mathbf{b} + \Delta\mathbf{b}) = \mathbf{x} + \Delta\mathbf{x}$$

where $\mathbf{x} = A^{-1}\mathbf{b}$ is the original solution and the change in the solutions is $\Delta\mathbf{x} = A^{-1}\Delta\mathbf{b}$. Now the absolute errors $\|\Delta\mathbf{b}\|$ and $\|\Delta\mathbf{x}\|$ are not very meaningful, since an absolute error $\|\Delta\mathbf{b}\| = 100$ is not very large if $\|\mathbf{b}\| = 1,000,000$, but is large if $\|\mathbf{b}\| = 1$. What we really care about are the *relative* errors $\|\Delta\mathbf{b}\|/\|\mathbf{b}\|$ and $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$. Can we bound the relative error in the solution in terms of the relative error in the equation? The answer is yes. Beginning with

$$\begin{aligned}\|\Delta\mathbf{x}\|\|\mathbf{b}\| &= \|A^{-1}\Delta\mathbf{b}\|\|A\mathbf{x}\| \\ &\le \|A^{-1}\|\|\Delta\mathbf{b}\|\|A\|\|\mathbf{x}\|,\end{aligned}$$

we can divide by $\|\mathbf{b}\|\|\mathbf{x}\|$ to obtain

$$\begin{aligned}\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} &\le \|A^{-1}\|\|A\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} \\ &= \operatorname{cond}(A)\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}.\end{aligned}$$

This equation gives the real meaning of the condition number. If the condition number is near to 1 then the relative error of the solution is about the same as the relative error in the equation. However, a large condition number means that a small relative error in the equation can lead to a large relative error in the solution.

In MATLAB/Octave the condition number is computed using `cond(A)`.

```
>  A=[2 0; 0 0.5];
> cond(A)
ans =  4
```

I.1.8. Summary of MATLAB/Octave commands used in this section

How to create a row vector

`[ ]` square brackets are used to construct matrices and vectors. Create a row in the matrix by entering elements within brackets. Separate each element with a comma or space. For example, to create a row vector **a** with three columns (i.e. a 1-by-3 matrix), type

`a=[1 1 1]` or equivalently `a=[1,1,1]`

How to create a column vector or a matrix with more than one row

`;` when the semicolon is used inside square brackets, it terminates rows. For example,

`a=[1;1;1]` creates a column vector with three rows

`B=[1 2 3; 4 5 6]` creates a $2-\text{by}-3$ matrix

`'` when a matrix (or a vector) is followed by a single quote `'` (or apostrophe) MATLAB flips rows with columns, that is, it generates the transpose. When the original matrix is a simple row vector, the apostrophe operator turns the vector into a column vector. For example,

`a=[1 1 1]'` creates a column vector with three rows

`B=[1 2 3; 4 5 6]'` creates a $3-\text{by}-2$ matrix where the first row is `1 4`

How to use specialized matrix functions

`rand(n,m)` returns a n-by-m matrix with random numbers between 0 and 1.

How to extract elements or submatrices from a matrix

`A(i,j)` returns the entry of the matrix A in the i-th row and the j-th column

`A(i,:)` returns a row vector containing the i-th row of A

`A(:,j)` returns a column vector containing the j-th column of A

`A(i:j,k:m)` returns a matrix containing a specific submatrix of the matrix A. Specifically, it returns all rows between the i-th and the j-th rows of A, and all columns between the k-th and the m-th columns of A.

How to perform specific operations on a matrix

`det(A)` returns the determinant of the (square) matrix A

`rref(A)` returns the reduced row echelon form of the matrix A

`norm(V)` returns the 2-norm (Euclidean norm) of the vector V

`norm(V,1)` returns the 1-norm of the vector V

`norm(V,inf)` returns the infinity norm of the vector V

I.2. Interpolation

Prerequisites and Learning Goals

From your work in previous courses, you should be able to

- compute the determinant of a square matrix; apply the basic linearity properties of the determinant, and explain what its value means about existence and uniqueness of solutions.

After completing this section, you should be able to

- Give a definition of an interpolating function and show how a given interpolation problem (i.e. the problem of finding an interpolating function of given shape/form) translates into finding the solutions to a linear system; explain the idea of getting a unique interpolating function by restricting the class of functions under consideration.

- Define the problem of Lagrange interpolation and express it in terms of a system of equations where the unknowns are the coefficients of a polynomial of given degree; set up the system in matrix form using the Vandermonde matrix, derive the formula for the determinant of the Vandermonde matrix; explain why a solution to the Lagrange interpolation problem always exists.

- Define the mathematical problem of interpolation using splines, this includes listing the conditions that the splines must satisfy and translating them into mathematical equations; express the spline-interpolation problem in terms of a system of equations where the unknowns are related to the coefficients of the splines, derive the corresponding matrix equation.

- Compare and constrast interpolation with splines with Lagrange interpolation.

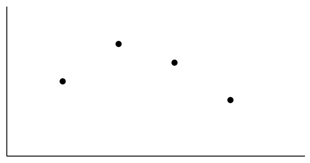
- Explain how minimizing the bending energy leads to a description of the shape of the spline as a piecewise polynomial function.

- Given a set of points, use MATLAB/Octave to calculate and plot the interpolating functions, including the interpolating polynomial in Lagrange interpolation and the piecewise cubic function for splines; this requires you be able to execute and interpret the output of specific commands such as `vander`, `polyval`, `plot`.

- Discuss how the condition numbers arising in the above interpolation problems vary; explain why Lagrange interpolation is not a practical method for large numbers of points.

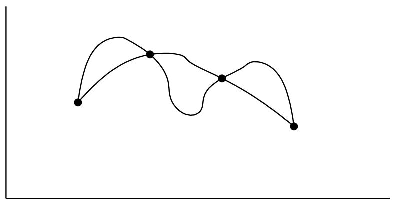
I.2.1. Introduction

Suppose we are given some points $(x_1, y_1), \ldots, (x_n, y_n)$ in the plane, where the points x_i are all distinct.



Our task is to find a function $f(x)$ that passes through all these points. In other words, we require that $f(x_i) = y_i$ for $i = 1, \ldots, n$. Such a function is called an interpolating function. Problems like this arise in practical applications in situations where a function is sampled at a finite number of points. For example, the function could be the shape of the model we have made for a car. We take a bunch of measurements $(x_1, y_1), \ldots, (x_n, y_n)$ and send them to the factory. What's the best way to reproduce the original shape?

Of course, it is impossible to reproduce the original shape with certainty. There are infinitely many functions going through the sampled points.



To make our problem of finding the interpolating function $f(x)$ have a unique solution, we must require something more of $f(x)$, either that $f(x)$ lies in some restricted class of functions, or that $f(x)$ is the function that minimizes some measure of "badness". We will look at both approaches.

I.2.2. Lagrange interpolation

For Lagrange interpolation, we try to find a polynomial $p(x)$ of lowest possible degree that passes through our points. Since we have n points, and therefore n equations $p(x_i) = y_i$ to solve, it makes sense that $p(x)$ should be a polynomial of degree $n - 1$

$$p(x) = a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_{n-1} x + a_n$$

with n unknown coefficients $a_1, a_2, \ldots, a_n$. (Don't blame me for the screwy way of numbering the coefficients. This is the MATLAB/Octave convention.)

The n equations $p(x_i) = y_i$ are n linear equations for these unknown coefficients which we may write as

$$\begin{bmatrix} x_1^{n-1} & x_1^{n-2} & \cdots & x_1^2 & x_1 & 1 \\ x_2^{n-1} & x_2^{n-2} & \cdots & x_2^2 & x_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ x_n^{n-1} & x_n^{n-2} & \cdots & x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n-2} \\ a_{n-1} \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Thus we see that the problem of Lagrange interpolation reduces to solving a system of linear equations. If this system has a unique solution, then there is exactly one polynomial $p(x)$ of degree $n-1$ running through our points. This matrix for this system of equations has a special form and is called a Vandermonde matrix.

To decide whether the system of equations has a unique solution we need to determine whether the Vandermonde matrix is invertible or not. One way to do this is to compute the determinant. It turns out that the determinant of a Vandermonde matrix has a particularly simple form, but it's a little tricky to see this. The 2×2 case is simple enough:

$$\det\left(\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix}\right) = x_1 - x_2.$$

To go on to the 3×3 case we won't simply expand the determinant, but recall that the determinant is unchanged under row (and column) operations of the type "add a multiple of one row (column) to another." Thus if we start with a 3×3 Vandermonde determinant, add $-x_1$ times the second column to the first, and then add $-x_1$ times the third column to the second, the determinant doesn't change and we find that

$$\det\left(\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix}\right) = \det\left(\begin{bmatrix} 0 & x_1 & 1 \\ x_2^2 - x_1x_2 & x_2 & 1 \\ x_3^2 - x_1x_3 & x_3 & 1 \end{bmatrix}\right) = \det\left(\begin{bmatrix} 0 & 0 & 1 \\ x_2^2 - x_1x_2 & x_2 - x_1 & 1 \\ x_3^2 - x_1x_3 & x_3 - x_1 & 1 \end{bmatrix}\right).$$

Now we can take advantage of the zeros in the first row, and calculate the determinant by expanding along the top row. This gives

$$\det\left(\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix}\right) = \det\left(\begin{bmatrix} x_2^2 - x_1x_2 & x_2 - x_1 \\ x_3^2 - x_1x_3 & x_3 - x_1 \end{bmatrix}\right) = \det\left(\begin{bmatrix} x_2(x_2 - x_1) & x_2 - x_1 \\ x_3(x_3 - x_1) & x_3 - x_1 \end{bmatrix}\right).$$

Now, we recall that the determinant is linear in each row separately. This implies that

$$\begin{aligned} \det\left(\begin{bmatrix} x_2(x_2 - x_1) & x_2 - x_1 \\ x_3(x_3 - x_1) & x_3 - x_1 \end{bmatrix}\right) &= (x_2 - x_1)\det\left(\begin{bmatrix} x_2 & 1 \\ x_3(x_3 - x_1) & x_3 - x_1 \end{bmatrix}\right) \\ &= (x_2 - x_1)(x_3 - x_1)\det\left(\begin{bmatrix} x_2 & 1 \\ x_3 & 1 \end{bmatrix}\right). \end{aligned}$$

But the determinant on the right is a 2×2 Vandermonde determinant that we have already computed. Thus we end up with the formula

$$\det\left(\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix}\right) = -(x_2 - x_1)(x_3 - x_1)(x_3 - x_2).$$

The general formula is

$$\det\begin{bmatrix} x_1^{n-1} & x_1^{n-2} & \cdots & x_1^2 & x_1 & 1 \\ x_2^{n-1} & x_2^{n-2} & \cdots & x_2^2 & x_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ x_n^{n-1} & x_n^{n-2} & \cdots & x_n^2 & x_n & 1 \end{bmatrix} = \pm \prod_{i>j}(x_i - x_j),$$

where $\pm = (-1)^{n(n-1)/2}$. It can be proved by induction using the same strategy as we used for the 3×3 case. The product on the right is the product of all differences $x_i - x_j$. This product is non-zero, since we are assuming that all the points x_i are distinct. Thus the Vandermonde matrix is invertible, and a solution to the Lagrange interpolation problem always exists.

Now let's use MATLAB/Octave to see how this interpolation works in practice.

We begin by putting some points x_i into a vector `X` and the corresponding points y_i into a vector `Y`.

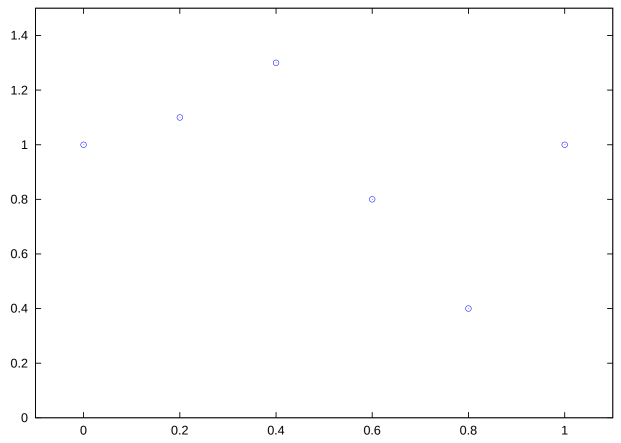
```
>X=[0 0.2 0.4 0.6 0.8 1.0]
>Y=[1 1.1 1.3 0.8 0.4 1.0]
```

We can use the plot command in MATLAB/Octave to view these points. The command `plot(X,Y)` will pop open a window and plot the points (x_i, y_i) joined by straight lines. In this case we are not interested in joining the points (at least not with straight lines) so we add a third argument: `'o'` plots the points as little circles. (For more information you can type `help plot` on the MATLAB/Octave command line.) Thus we type

```
>plot(X,Y,'o')
>axis([-0.1, 1.1, 0, 1.5])
>hold on
```

The axis command adjusts the axis. Normally when you issue a new plot command, the existing plot is erased. The `hold on` prevents this, so that subsequent plots are all drawn on the same graph. The original behaviour is restored with `hold off`.

When you do this you should see a graph appear that looks something like this.



Now let's compute the interpolation polynomial. Luckily there are build in functions in MATLAB/Octave that make this very easy. To start with, the function `vander(X)` returns the Vandermonde matrix corresponding to the points in `X`. So we define

```
>V=vander(X)

V =

   0.00000   0.00000   0.00000   0.00000   0.00000   1.00000
   0.00032   0.00160   0.00800   0.04000   0.20000   1.00000
   0.01024   0.02560   0.06400   0.16000   0.40000   1.00000
   0.07776   0.12960   0.21600   0.36000   0.60000   1.00000
   0.32768   0.40960   0.51200   0.64000   0.80000   1.00000
   1.00000   1.00000   1.00000   1.00000   1.00000   1.00000
```

We saw above that the coefficients of the interpolation polynomial are given by the solution $\mathbf{a}$ to the equation $V\mathbf{a} = \mathbf{y}$. We find those coefficients using

```
>a=V\Y'
```

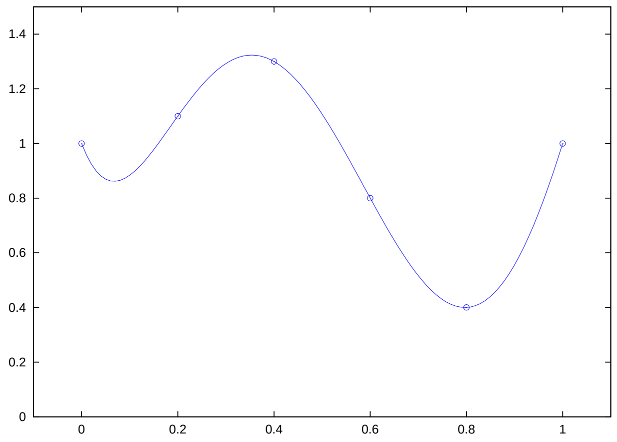
Let's have a look at the interpolating polynomial. The MATLAB/Octave function `polyval(a,X)` takes a vector `X` of x values, say $x_1, x_2, \ldots x_k$ and returns a vector containing the values $p(x_1), p(x_2), \ldots p(x_k)$, where p is the polynomial whose coefficients are in the vector `a`, that is,

$$p(x) = a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_{n-1} x + a_n$$

So `plot(X,polyval(a,X))` would be the command we want, except that with the present definition of `X` this would only plot the polynomial at the interpolation points. What we want is to plot the polynomial for all points, or at least for a large number. The command `linspace(0,1,100)` produces a vector of 100 linearly spaced points between 0 and 1, so the following commands do the job.

```
>XL=linspace(0,1,100);
>YL=polyval(a,XL);
>plot(XL,YL);
>hold off
```

The result looks pretty good

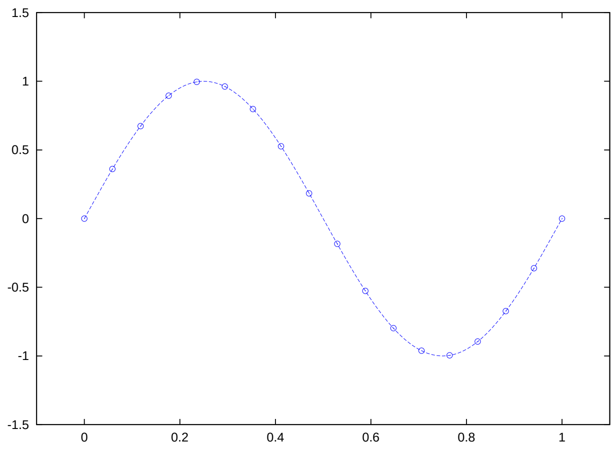


The MATLAB/Octave commands for this example are in `lagrange.m`.

Unfortunately, things get worse when we increase the number of interpolation points. One clue that there might be trouble ahead is that even for only six points the condition number of `V` is quite high (try it!). Let's see what happens with 18 points. We will take the x values to be equally spaced between 0 and 1. For the y values we will start off by taking $y_i = \sin(2\pi x_i)$. We repeat the steps above.

```
>X=linspace(0,1,18);
>Y=sin(2*pi*X);
>plot(X,Y,'o')
>axis([-0.1 1.1 -1.5 1.5])
>hold on
>V=vander(X);
>a=V\Y';
>XL=linspace(0,1,500);
>YL=polyval(a,XL);
>plot(XL,YL);
```

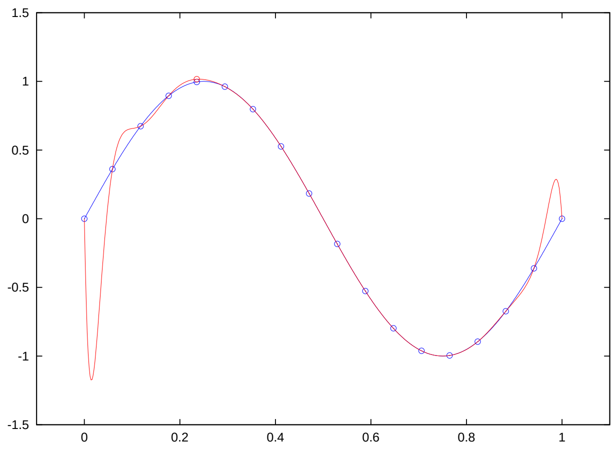
The resulting picture looks okay.



But look what happens if we change one of the y values just a little. We add 0.02 to the fifth y value, redo the Lagrange interpolation and plot the new values in red.

```
>Y(5) = Y(5)+0.02;
>plot(X(5),Y(5),'or')
>a=V\Y';
>YL=polyval(a,XL);
>plot(XL,YL,'r');
>hold off
```

The resulting graph makes a wild excursion and even though it goes through the given points, it would not be a satisfactory interpolating function in a practical situation.



A calculation reveals that the condition number is

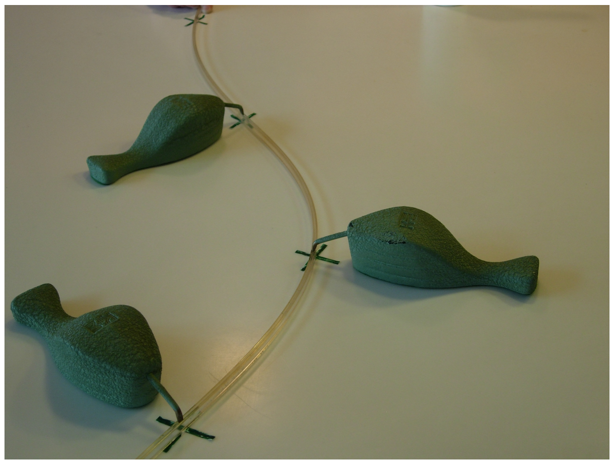
```
>cond(V)

ans =  1.8822e+14
```

If we try to go to 20 points equally spaced between 0 and 1, the Vandermonde matrix is so ill conditioned that MATLAB/Octave considers it to be singular.

I.2.3. Cubic splines

In the last section we saw that Lagrange interpolation becomes impossible to use in practice if the number of points becomes large. Of course, the constraint we imposed, namely that the interpolating function be a polynomial of low degree, does not have any practical basis. It is simply mathematically convenient. Let's start again and consider how ship and airplane designers actually drew complicated curves before the days of computers. Here is a picture of a draughtsman's spline (taken from `http://pages.cs.wisc.edu/~deboor/draftspline.html` where you can also find a nice photo of such a spline in use)



It consists of a bendable but stiff strip held in position by a series of weights called ducks. We will try to make a mathematical model of such a device.

We begin again with points $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$ in the plane. Again we are looking for a function $f(x)$ that goes through all these points. This time, we want to find the function that has the same shape as a real draughtsman's spline. We will imagine that the given points are the locations of the ducks.

Our first task is to identify a large class of functions that represent possible shapes for the spline. We will write down three conditions for a function $f(x)$ to be acceptable. Since the spline has no breaks in it the function $f(x)$ should be continuous. Moreover $f(x)$ should pass through the given points.

Condition 1: $f(x)$ is continuous and $f(x_i) = y_i$ for $i = 1, \ldots, n$.

The next condition reflects the assumption that the strip is stiff but bendable. If the strip were not stiff, say it were actually a rubber band that just is stretched between the ducks, then our resulting function would be a straight line between each duck location (x_i, y_i). At each duck location there would be a sharp bend in the function. In other words, even though the function itself would be continuous, the first derivative would be discontinuous at the duck locations. We will interpret the words "bendable but stiff" to mean that the first derivatives of $f(x)$ exist. This leads to our second condition.

Condition 2: The first derivative $f'(x)$ exists and is continuous everywhere, including each interior duck location x_i.

In between the duck locations we will assume that $f(x)$ is perfectly smooth and that higher derivatives behave nicely when we approach the duck locations from the right or the left. This leads to

Condition 3: For x in between the duck points x_i the higher order derivatives $f''(x), f'''(x), \ldots$ all exist and have left and right limits as x approaches each x_i.

In this condition we are allowing for the possibility that $f''(x)$ and higher order derivatives have a jump at the duck locations. This happens if the left and right limits are different.

The set of functions satisfying conditions 1, 2 and 3 are all the *possible* shapes of the spline. How do we decide which one of these shapes is the actual shape of the spline? To do this we need to invoke a bit of the physics of bendable strips. The bending energy $E[f]$ of a strip whose shape is described by the function f is given by the integral

$$E[f] = \int_{x_1}^{x_n} \left(f''(x)\right)^2 dx$$

The actual spline will relax into the shape that makes $E[f]$ as small as possible. Thus, among all the functions satisfying conditons 1, 2 and 3, we want to *choose the one that minimizes* $E[f]$.

This minimization problem is similiar to ones considered in calculus courses, except that instead of real numbers, the variables in this problem are *functions* f satisfying conditons 1, 2 and 3. In calculus, the minimum is calculated by "setting the derivative to zero." A similar procedure is described in the next section. Here is the result of that calculation: Let $F(x)$ be the function describing the shape that makes $E[f]$ as small as possible. In other words,

- $F(x)$ satisfies condtions 1, 2 and 3.
- If $f(x)$ also satisfies conditions 1, 2 and 3, then $E[F] \le E[f]$.

Then, in addition to conditions 1, 2 and 3, $F(x)$ satisfies

Condition a: In each interval (x_i, x_{i+1}), the function $F(x)$ is a cubic polynomial. In other words, for each interval there are coefficients A_i, B_i, C_i and D_i such that $F(x) = A_i x^3 + B_i x^2 + C_i x + D_i$ for all x between x_i and x_{i+1}. The coefficients can be different for different intervals.

Condition b: The second derivative $F''(x)$ is continuous.

Condition c: When x is an endpoint (either x_1 or x_n) then $F''(x) = 0$

As we will see, there is exactly one function satisfying conditions 1, 2, 3, a, b and c.

I.2.4. The minimization procedure

In this section we explain the minimization procedure leading to a mathematical description of the shape of a spline. In other words, we show that if among all functions $f(x)$ satisfying conditions 1, 2 and 3, the function $F(x)$ is the one with $E[f]$ the smallest, then $F(x)$ also satisfies conditions a, b and c.

The idea is to assume that we have found $F(x)$ and then try to deduce what properties it must satisfy. There is actually a is a hidden assumption here — we are assuming that the minimizer $F(x)$ exists. This is not true for every minimization problem (think of minimizing the function $(x^2+1)^{-1}$ for $-\infty < x < \infty$). However the spline problem does have a minimizer, and we will leave out the step of proving it exists.

Given the minimizer $F(x)$ we want to wiggle it a little and consider functions of the form $F(x) + \epsilon h(x)$, where $h(x)$ is another function and ϵ be a number. We want to do this in such a way that for every ϵ, the function $F(x) + \epsilon h(x)$ still satisfies conditions 1, 2 and 3. Then we will be able to compare $E[F]$ with $E[F + \epsilon h]$. A little thought shows that functions of form $F(x) + \epsilon h(x)$ will satsify conditions 1, 2 and 3 for every value of ϵ if h satisfies

Condition 1': $h(x_i) = 0$ for $i = 1, \ldots, n$.

together with conditions 2 and 3 above.

Now, the minimization property of F says that each fixed function h satisfying 1', 2 and 3 the function of ϵ given by $E[F + \epsilon h]$ has a local minimum at $\epsilon = 0$. From Calculus we know that this implies that

$$\left.\frac{dE[F+\epsilon h]}{d\epsilon}\right|_{\epsilon=0} = 0. \tag{I.1}$$

Now we will actually compute this derivative with respect to ϵ and see what information we can get from the fact that it is zero for every choice of $h(x)$ satisfying conditions 1', 2 and 3. To simplify the presentation we will assume that there are only three points (x_1, y_1), (x_2, y_2) and (x_3, y_3). The goal of this computation is to establish that equation (I.1) can be rewritten as (I.2).

To begin, we compute

$$\begin{aligned} 0 = \left.\frac{dE[F+\epsilon h]}{d\epsilon}\right|_{\epsilon=0} &= \int_{x_1}^{x_3} \left.\frac{d(F''(x)+\epsilon h''(x))^2}{d\epsilon}\right|_{\epsilon=0} dx \\ &= \int_{x_1}^{x_3} 2\,(F''(x)+\epsilon h''(x))h''(x)\big|_{\epsilon=0}\, dx \\ &= 2\int_{x_1}^{x_3} F''(x)h''(x)dx \\ &= 2\int_{x_1}^{x_2} F''(x)h''(x)dx + 2\int_{x_2}^{x_3} F''(x)h''(x)dx \end{aligned}$$

We divide by 2 and integrate by parts in each integral. This gives

$$0 = F''(x)h'(x)\Big|_{x=x_1}^{x=x_2} - \int_{x_1}^{x_2} F'''(x)h'(x)dx + F''(x)h'(x)\Big|_{x=x_2}^{x=x_3} - \int_{x_2}^{x_3} F'''(x)h'(x)dx$$

In each boundary term we have to take into account the possibility that $F''(x)$ is not continuous across the points x_i. Thus we have to use the appropriate limit from the left or the right. So, for the first boundary term

$$F''(x)h'(x)\Big|_{x=x_1}^{x=x_2} = F''(x_2-)h'(x_2) - F''(x_1+)h'(x_1)$$

Notice that since $h'(x)$ is continuous across each x_i we need not distinguish the limits from the left and the right. Expanding and combining the boundary terms we get

$$\begin{aligned}0 &= -F''(x_1+)h'(x_1) + \big(F''(x_2-) - F''(x_2+)\big)h'(x_2) + F''(x_3-)h'(x_3)\\ &\quad - \int_{x_1}^{x_2} F'''(x)h'(x)dx - \int_{x_2}^{x_3} F'''(x)h'(x)dx\end{aligned}$$

Now we integrate by parts again. This time the boundary terms all vanish because $h(x_i) = 0$ for every i. Thus we end up with the equation

$$\begin{aligned}0 &= -F''(x_1+)h'(x_1) + \big(F''(x_2-) - F''(x_2+)\big)h'(x_2) + F''(x_3-)h'(x_3)\\ &\quad + \int_{x_1}^{x_2} F''''(x)h(x)dx - \int_{x_2}^{x_3} F''''(x)h(x)dx \end{aligned} \qquad \text{(I.2)}$$

as desired.

Recall that this equation has to be true for every choice of h satisfying conditions 1', 2 and 3. For different choices of $h(x)$ we can extract different pieces of information about the minimizer $F(x)$.

To start, we can choose h that is zero everywhere except in the open interval (x_1, x_2). For all such h we then obtain $0 = \int_{x_1}^{x_2} F''''(x)h(x)dx$. This can only happen if

$$F''''(x) = 0 \quad \text{for} \quad x_1 < x < x_2$$

Thus we conclude that the fourth derivative $F''''(x)$ is zero in the interval (x_1, x_2).

Once we know that $F''''(x) = 0$ in the interval (x_1, x_2), then by integrating both sides we can conclude that $F'''(x)$ is constant. Integrating again, we find $F''(x)$ is a linear polynomial. By integrating four times, we see that $F(x)$ is a cubic polynomial in that interval. When doing the integrals, we must not extend the domain of integration over the boundary point x_2 since $F''''(x)$ may not exist (let alone by zero) there.

Similarly $F''''(x)$ must also vanish in the interval (x_2, x_3), so $F(x)$ is a (possibly different) cubic polynomial in the interval (x_2, x_3).

(An aside: to understand better why the polynomials might be different in the intervals (x_1, x_2) and (x_3, x_4) consider the function $g(x)$ (unrelated to the spline problem) given by

$$g(x) = \begin{cases} 0 & \text{for } x_1 < x < x_2 \\ 1 & \text{for } x_2 < x < x_3 \end{cases}$$

Then $g'(x) = 0$ in each interval, and an integration tells us that g is constant in each interval. However, $g'(x_2)$ does not exist, and the constants are different.)

We have established that $F(x)$ satisfies condition a.

Now that we know that $F''''(x)$ vanishes in each interval, we can return to (I.2) and write it as

$$0 = -F''(x_1+)h'(x_1) + \big(F''(x_2-) - F''(x_2+)\big)h'(x_2) + F''(x_3-)h'(x_3)$$

Now choose $h(x)$ with $h'(x_1) = 1$ and $h'(x_2) = h'(x_3) = 0$. Then the equation reads

$$F''(x_1+) = 0$$

Similarly, choosing $h(x)$ with $h'(x_3) = 1$ and $h'(x_1) = h'(x_2) = 0$ we obtain

$$F''(x_3-) = 0$$

This establishes condition c.

Finally choosing $h(x)$ with $h'(x_2) = 1$ and $h'(x_1) = h'(x_3) = 0$ we obtain

$$F''(x_2-) - F''(x_2+) = 0$$

In other words, F'' must be continuous across the interior duck position. Thus shows that condition b holds, and the derivation is complete.

This calculation is easily generalized to the case where there are n duck positions $x_1, \ldots, x_n$.

A reference for this material is *Essentials of numerical analysis, with pocket calculator demonstrations,* by Henrici.

I.2.5. The linear equations for cubic splines

Let us now turn this description into a system of linear equations. In each interval (x_i, x_{i+1}), for $i = 1, \ldots n-1$, $f(x)$ is given by a cubic polynomial $p_i(x)$ which we can write in the form

$$p_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$$

for coefficients a_i, b_i, c_i and d_i to be determined. For each $i = 1, \ldots n-1$ we require that $p_i(x_i) = y_i$ and $p_i(x_{i+1}) = y_{i+1}$. Since $p_i(x_i) = d_i$, the first of these equations is satisfied if $d_i = y_i$. So let's simply make that substitution. This leaves the $n-1$ equations

$$p_i(x_{i+1}) = a_i(x_{i+1} - x_i)^3 + b_i(x_{i+1} - x_i)^2 + c_i(x_{i+1} - x_i) + y_i = y_{i+1}.$$

Secondly, we require continuity of the first derivative across interior x_i's. This translates to $p_i'(x_{i+1}) = p_{i+1}'(x_{i+1})$ or

$$3a_i(x_{i+1} - x_i)^2 + 2b_i(x_{i+1} - x_i) + c_i = c_{i+1}$$

for $i = 1, \ldots, n-2$, giving an additional $n-2$ equations. Next, we require continuity of the second derivative across interior x_i's. This translates to $p_i''(x_{i+1}) = p_{i+1}''(x_{i+1})$ or

$$6a_i(x_{i+1} - x_i) + 2b_i = 2b_{i+1}$$

for $i = 1, \ldots, n-2$, once more giving an additional $n-2$ equations. Finally, we require that $p_1''(x_1) = p_{n-1}''(x_n) = 0$. This yields two more equations

$$\begin{aligned} 2b_1 &= 0 \\ 6a_{n-1}(x_n - x_{n-1}) + 2b_{n-1} &= 0 \end{aligned}$$

for a total of $3(n-1)$ equations for the same number of variables.

We now specialize to the case where the distances between the points x_i are equal. Let $L = x_{i+1} - x_i$ be the common distance. Then the equations read

$$\begin{array}{lllll} a_iL^3 + b_iL^2 & +c_iL & & & = y_{i+1} - y_i \\ 3a_iL^2 + 2b_iL & +c_i & & -c_{i+1} & = 0 \\ 6a_iL + 2b_i & & -2b_{i+1} & & = 0 \end{array}$$

for $i = 1 \ldots n-2$ together with

$$\begin{array}{lll} a_{n-1}L^3 + b_{n-1}L^2 & +c_{n-1}L & = y_n - y_{n-1} \\ \quad + 2b_1 & & = 0 \\ 6a_{n-1}L + 2b_{n-1} & & = 0 \end{array}$$

We make one more simplification. After multiplying some of the equations with suitable powers of L we can write these as equations for $\alpha_i = a_iL^3$, $\beta_i = b_iL^2$ and $\gamma_i = c_iL$. They have a very simple block structure. For example, when $n = 4$ the matrix form of the equations is

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 6 & 2 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 2 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 6 & 2 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 & 2 & 0 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \beta_1 \\ \gamma_1 \\ \alpha_2 \\ \beta_2 \\ \gamma_2 \\ \alpha_3 \\ \beta_3 \\ \gamma_3 \end{bmatrix} = \begin{bmatrix} y_2 - y_1 \\ 0 \\ 0 \\ y_3 - y_2 \\ 0 \\ 0 \\ y_4 - y_3 \\ 0 \\ 0 \end{bmatrix}$$

Notice that the matrix in this equation does not depend on the points (x_i, y_i). It has a 3×3 block structure. If we define the 3×3 blocks

$$N = \begin{bmatrix} 1 & 1 & 1 \\ 3 & 2 & 1 \\ 6 & 2 & 0 \end{bmatrix}$$

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

$$\mathbf{0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$V = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 6 & 2 & 0 \end{bmatrix}$$

then the matrix in our equation has the form

$$S = \begin{bmatrix} N & M & \mathbf{0} \\ \mathbf{0} & N & M \\ T & \mathbf{0} & V \end{bmatrix}$$

Once we have solved the equation for the coefficients α_i, β_i and γ_i, the function $F(x)$ in the interval (x_i, x_{i+1}) is given by

$$F(x) = p_i(x) = \alpha_i \left(\frac{x - x_i}{L}\right)^3 + \beta_i \left(\frac{x - x_i}{L}\right)^2 + \gamma_i \left(\frac{x - x_i}{L}\right) + y_i$$

Now let us use MATLAB/Octave to plot a cubic spline. To start, we will do an example with four interpolation points. The matrix S in the equation is defined by

```
>N=[1 1 1;3 2 1;6 2 0];
>M=[0 0 0;0 0 -1; 0 -2 0];
>Z=zeros(3,3);
>T=[0 0 0;0 2 0; 0 0 0];
>V=[1 1 1;0 0 0;6 2 0];
>S=[N M Z; Z N M; T Z V]

S =

   1   1   1   0   0   0   0   0   0
   3   2   1   0   0  -1   0   0   0
   6   2   0   0  -2   0   0   0   0
   0   0   0   1   1   1   0   0   0
   0   0   0   3   2   1   0   0  -1
   0   0   0   6   2   0   0  -2   0
   0   0   0   0   0   0   1   1   1
   0   2   0   0   0   0   0   0   0
   0   0   0   0   0   0   6   2   0
```

Here we used the function `zeros(n,m)` which defines an $n \times m$ matrix filled with zeros.

To proceed we have to know what points we are trying to interpolate. We pick four (x, y) values and put them in vectors. Remember that we are assuming that the x values are equally spaced.

```
>X=[1, 1.5, 2, 2.5];
>Y=[0.5, 0.8, 0.2, 0.4];
```

We plot these points on a graph.

```
>plot(X,Y,'o')
>hold on
```

Now let's define the right side of the equation

```
>b=[Y(2)-Y(1),0,0,Y(3)-Y(2),0,0,Y(4)-Y(3),0,0];
```

and solve the equation for the coefficients.

```
>a=S\b';
```

Now let's plot the interpolating function in the first interval. We will use 50 closely spaced points to get a smooth looking curve.

```
>XL = linspace(X(1),X(2),50);
```

Put the first set of coefficients (α_1, β_1, γ_1, y_1) into a vector

```
>p = [a(1) a(2) a(3) Y(1)];
```

Now we put the values $p_1(x)$ into the vector `YL`. First we define the values $(x - x_1)/L$ and put them in the vector `XLL`. To get the values $x - x_1$ we want to subtract the vector with `X(1)` in every position from `X`. The vector with `X(1)` in every position can be obtained by taking a vector with 1 in every position (in MATLAB/Octave this is obtained using the function `ones(n,m)`) and multiplying by the number `X(1)`. Then we divide by the (constant) spacing between the x_i values.

```
>L = X(2)-X(1);
>XLL = (XL - X(1)*ones(1,50))/L;
```

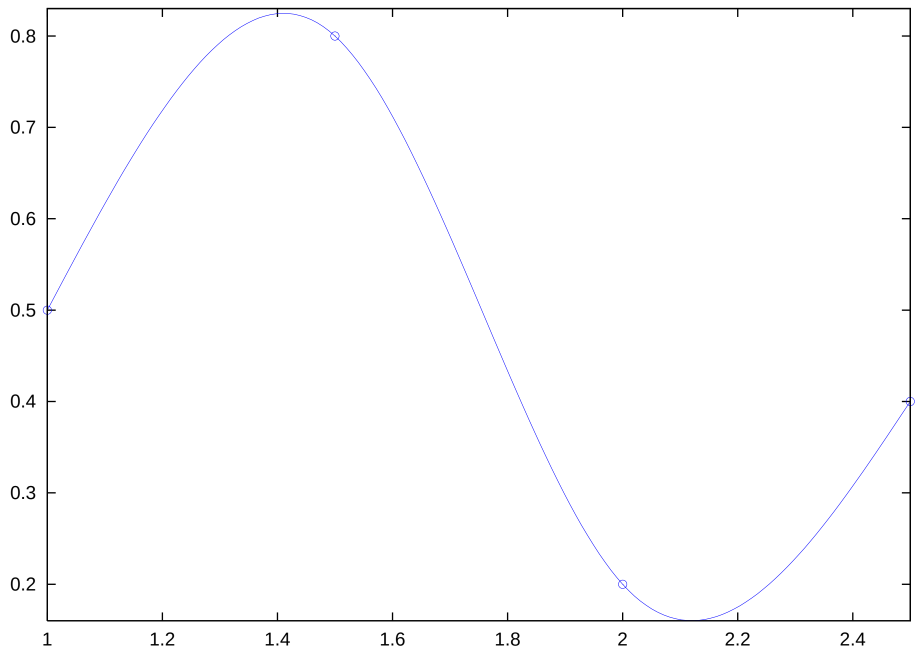
Now we evaluate the polynomial $p_1(x)$ and plot the resulting points.

```
>YL = polyval(p,XLL);
>plot(XL,YL);
```

To complete the plot, we repeat this steps for the intervals (x_2, x_3) and (x_3, x_4).

```
>XL = linspace(X(2),X(3),50);
>p = [a(4) a(5) a(6) Y(2)];
>XLL = (XL - X(2)*ones(1,50))/L;
>YL = polyval(p,XLL);
>plot(XL,YL);
>XL = linspace(X(3),X(4),50);
>p = [a(7) a(8) a(9) Y(3)];
>XLL = (XL - X(3)*ones(1,50))/L;
>YL = polyval(p,XLL);
>plot(XL,YL);
```

The result looks like this:



I have automated the procedure above and put the result in two files `splinemat.m` and `plotspline.m`. `splinemat(n)` returns the $3(n-1) \times 3(n-1)$ matrix used to compute a spline through n points while `plotspline(X,Y)` plots the cubic spline going through the points in `X` and `Y`. If you put these files in you MATLAB/Octave directory you can use them like this:

```
>splinemat(3)

ans =

   1   1   1   0   0   0
   3   2   1   0   0  -1
   6   2   0   0  -2   0
   0   0   0   1   1   1
   0   2   0   0   0   0
   0   0   0   6   2   0
```

and

```
>X=[1, 1.5, 2, 2.5];
>Y=[0.5, 0.8, 0.2, 0.4];
>plotspline(X,Y)
```

to produce the plot above.

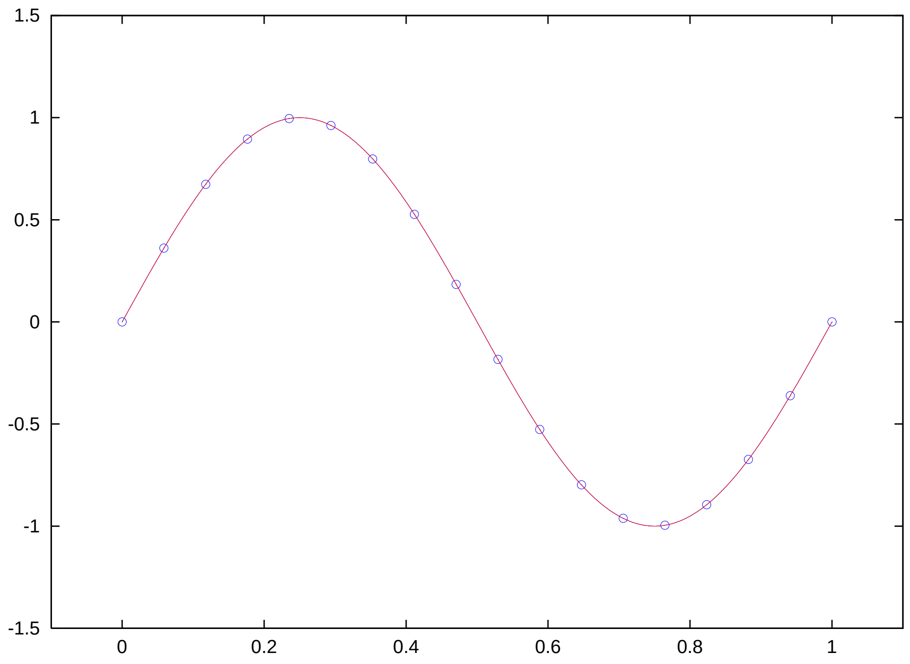
Let's use these functions to compare the cubic spline interpolation with the Lagrange interpolation by using the same points as we did before. Remember that we started with the points

```
>X=linspace(0,1,18);
>Y=sin(2*pi*X);
```

Let's plot the spline interpolation of these points

```
>plotspline(X,Y);
```

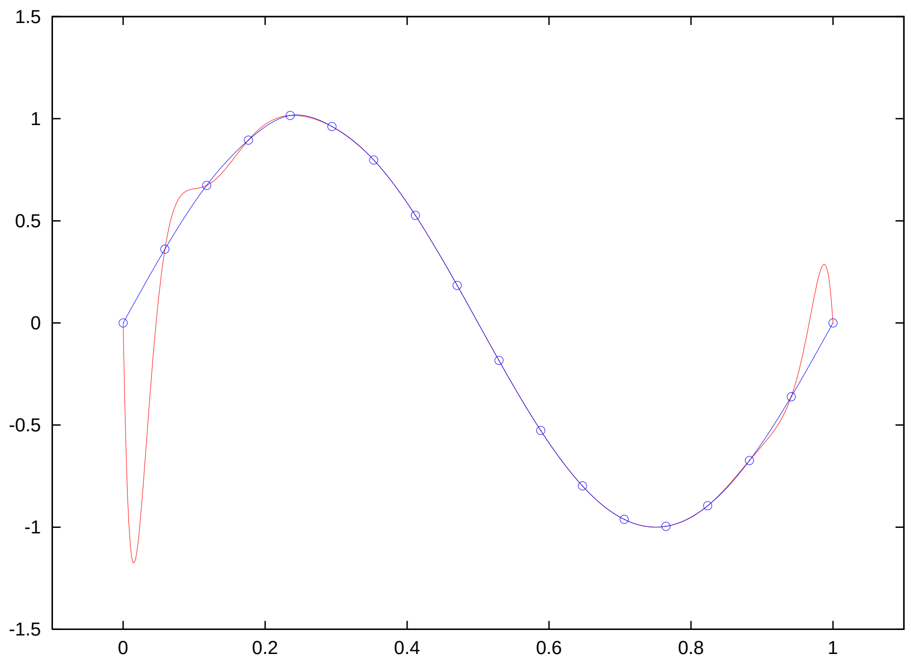
Here is the result with the Lagrange interpolation added (in red). The red (Lagrange) curve covers the blue one and its impossible to tell the curves apart.



Now we move one of the points slightly, as before.

```
>Y(5) = Y(5)+0.02;
```

Again, plotting the spline in blue and the Lagrange interpolation in red, here are the results.



This time the spline does a much better job! Let's check the condition number of the matrix for the splines. Recall that there are 18 points.

```
>cond(splinemat(18))

ans =  32.707
```

Recall the Vandermonde matrix had a condition number of `1.8822e+14`. This shows that the system of equations for the splines is very much better conditioned, by 13 orders of magnitude!!

Code for splinemat.m and plotspline.m

```
function S=splinemat(n)

L=[1 1 1;3 2 1;6 2 0];
M=[0 0 0;0 0 -1; 0 -2 0];
Z=zeros(3,3);
T=[0 0 0;0 2 0; 0 0 0];
V=[1 1 1;0 0 0;6 2 0];

S=zeros(3*(n-1),3*(n-1));
for k=[1:n-2]
for l=[1:k-1]
```

```
S(3*k-2:3*k,3*l-2:3*l) = Z;
end
S(3*k-2:3*k,3*k-2:3*k) = L;
S(3*k-2:3*k,3*k+1:3*k+3) = M;
for l=[k+2:n-1]
S(3*k-2:3*k,3*l-2:3*l) = Z;
end
end
S(3*(n-1)-2:3*(n-1),1:3)=T;
for l=[2:n-2]
S(3*(n-1)-2:3*(n-1),3*l-2:3*l) = Z;
end
S(3*(n-1)-2:3*(n-1),3*(n-1)-2:3*(n-1))=V;

end


function plotspline(X,Y)

n=length(X);
L=X(2)-X(1);

S=splinemat(n);

b=zeros(1,3*(n-1));
for k=[1:n-1]
b(3*k-2)=Y(k+1)-Y(k);
b(3*k-1)=0;
b(3*k)=0;
end

a=S\b';

npoints=50;
XL=[];
YL=[];
for k=[1:n-1]
XL = [XL linspace(X(k),X(k+1),npoints)];
p = [a(3*k-2),a(3*k-1),a(3*k),Y(k)];
XLL = (linspace(X(k),X(k+1),npoints) - X(k)*ones(1,npoints))/L;
YL = [YL polyval(p,XLL)];
end

plot(X,Y,'o')
hold on
```

```
plot(XL,YL)
hold off
```

I.2.6. Summary of MATLAB/Octave commands used in this section

How to access elements of a vector

`a(i)` returns the i-th element of the vector **a**

How to create a vector with linearly spaced elements

`linspace(x1,x2,n)` generates n points between the values $x1$ and $x2$.

How to create a matrix by concatenating other matrices

`C= [A B]` takes two matrices A and B and creates a new matrix C by concatenating A and B horizontally

Other specialized matrix functions

`zeros(n,m)` creates a n-by-m matrix filled with zeros

`ones(n,m)` creates a n-by-m matrix filled with ones

`vander(X)` creates the Vandermonde matrix corresponding to the points in the vector `X`. Note that the columns of the Vandermonde matrix are powers of the vector `X`.

Other useful functions and commands

`polyval(a,X)` takes a vector `X` of x values and returns a vector containing the values of a polynomial p evaluated at the x values. The coefficients of the polynomial p (in descending powers) are the values in the vector `a`.

`sin(X)` takes a vector `X` of values x and returns a vector containing the values of the function $\sin x$

`plot(X,Y)` plots vector Y versus vector X. Points are joined by a solid line. To change line types (solid, dashed, dotted, etc.) or plot symbols (point, circle, star, etc.), include an additional argument. For example, `plot(X,Y,'o')` plots the points as little circle.

I.3. Finite difference approximations

Prerequisites and Learning Goals

From your work in previous courses, you should be able to

- explain what it is meant by a boundary value problem.

After completing this section, you should be able to

- compute an approximate solution to a second order linear boundary value problem by deriving the corresponding finite difference equation and computing its solution with MATLAB/Octave; model different types of boundary conditions, including conditions on the value of the solution at the boundary and on the value of the first derivative at the boundary.

I.3.1. Introduction and example

One of the most important applications of linear algebra is the approximate solution of differential equations. In a differential equation we are trying to solve for an unknown function. The basic idea is to turn a differential equation into a system of $N \times N$ linear equations. As N becomes large, the vector solving the system of linear equations becomes a better and better approximation to the function solving the differential equation.

In this section we will learn how to use linear algebra to find approximate solutions to a boundary value problem of the form

$$f''(x) + q(x)f(x) = r(x) \quad \text{for} \quad 0 \le x \le 1$$

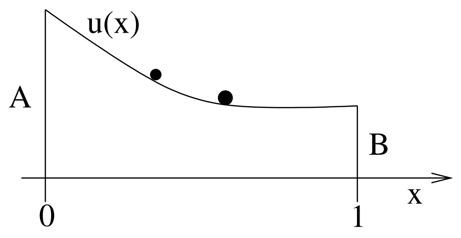
subject to boundary conditions

$$f(0) = A, \quad f(1) = B.$$

This is a differential equation where the unknown quantity to be found is a function $f(x)$. The functions $q(x)$ and $r(x)$ are given (known) functions.

As differential equations go, this is a very simple one. For one thing it is an ordinary differential equation (ODE), because it only involves one independent variable x. But the finite difference methods we will introduce can also be applied to partial differential equations (PDE).

It can be useful to have a picture in your head when thinking about an equation. Here is a situation where an equation like the one we are studying arises. Suppose we want to find the shape of a stretched hanging cable. The cable is suspended above the points $x = 0$ and $x = 1$ at heights of A and B respectively and hangs above the interval $0 \le x \le 1$. Our goal is to find the height $f(x)$ of the cable above the ground at every point x between 0 and 1.



The loading of the cable is described by a function $2r(x)$ that takes into account both the weight of the cable and any additional load. Assume that this is a known function. The height function $f(x)$ is the function that minimizes the sum of the stretching energy and the gravitational potential energy given by

$$E[f] = \int_0^1 [(f'(x))^2 + 2r(x)f(x)]dx$$

subject to the condition that $f(0) = A$ and $f(1) = B$. An argument similar (but easier) to the one we did for splines shows that the minimizer satisfies the differential equation

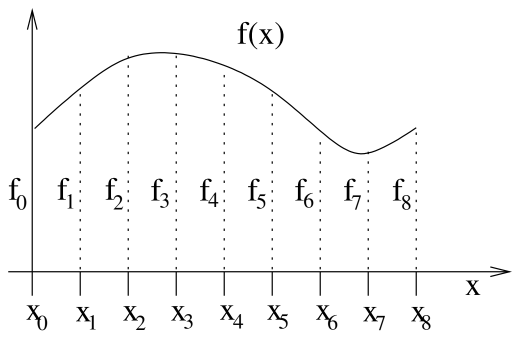
$$f''(x) = r(x).$$

So we end up with the special case of our original equation where $q(x) = 0$. Actually, this special case can be solved by simply integrating twice and adjusting the constants of integration to ensure $f(0) = A$ and $f(1) = B$. For example, when $r(x) = r$ is constant and $A = B = 1$, the solution is $f(x) = 1 - rx/2 + rx^2/2$. We can use this exact solution to compare against the approximate solution that we will compute.

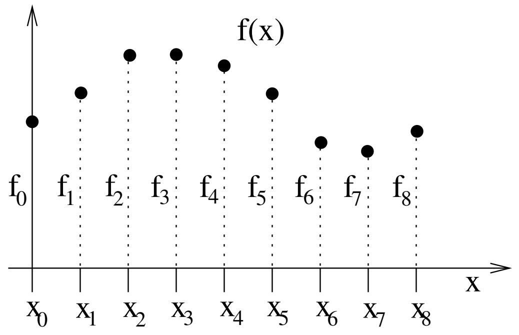
I.3.2. Discretization

In the finite difference approach to solving differential equations approximately, we want to approximate a function by a vector containing a finite number of sample values. Pick equally spaced points $x_k = k/N$, $k = 0, \ldots, N$ between 0 and 1. We will represent a function $f(x)$ by its values $f_k = f(x_k)$ at these points. Let

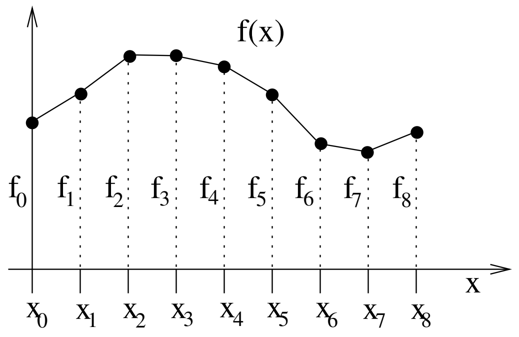
$$\mathbf{F} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_N \end{bmatrix}.$$



At this point we throw away all the other information about the function, keeping only the values at the sampled points.



If this is all we have to work with, what should we use as an approximation to $f'(x)$? It seems reasonable to use the slopes of the line segments joining our sampled points.



Notice, though, that there is one slope for every interval (x_i, x_{i+1}) so the vector containing the slopes has one fewer entry than the vector $\mathbf{F}$. The formula for the slope in the interval (x_i, x_{i+1}) is $(f_{i+1} - f_i)/\Delta x$ where the distance $\Delta x = x_{i+1} - x_i$ (in this case $\Delta x = 1/N$). Thus the vector containing the slopes is

$$\mathbf{F}' = (\Delta x)^{-1} \begin{bmatrix} f_1 - f_0 \\ f_2 - f_1 \\ f_3 - f_2 \\ \vdots \\ f_N - f_{N-1} \end{bmatrix} = (\Delta x)^{-1} \begin{bmatrix} -1 & 1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 1 & 0 & \cdots & 0 \\ 0 & 0 & -1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_N \end{bmatrix} = (\Delta x)^{-1} D_N \mathbf{F}$$

where D_N is the $N \times (N+1)$ finite difference matrix in the formula above. The vector $\mathbf{F}'$ is our approximation to the first derivative function $f'(x)$.

To approximate the second derivative $f''(x)$, we repeat this process to define the vector $\mathbf{F}''$. There will be one entry in this vector for each adjacent pair of slopes, that is, each adjacent pair of entries of $\mathbf{F}'$. These are naturally labelled by the interior points $x_1, x_2, \ldots, x_{n-1}$. Thus we obtain

$$\mathbf{F}'' = (\Delta x)^{-2} D_{N-1} D_N \mathbf{F} = (\Delta x)^{-2} \begin{bmatrix} 1 & -2 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_N \end{bmatrix}.$$

Let $r_k = r(x_k)$ be the sampled points for the load function $r(x)$ and define the vector approximation for r at the interior points

$$\mathbf{r} = \begin{bmatrix} r_1 \\ \vdots \\ r_{N-1} \end{bmatrix}.$$

The reason we only define this vector for interior points is that that is where $\mathbf{F}''$ is defined. Now we can write down the finite difference approximation to $f''(x) = r(x)$ as

$$(\Delta x)^{-2} D_{N-1} D_N \mathbf{F} = \mathbf{r} \qquad \text{or} \qquad D_{N-1} D_N \mathbf{F} = (\Delta x)^2 \mathbf{r}$$

This is a system of $N-1$ equations in $N+1$ unknowns. To get a unique solution, we need two more equations. That is where the boundary conditions come in! We have two boundary conditions, which in this case can simply be written as $f_0 = A$ and $f_N = B$. Combining these with the $N-1$ equations for the interior points, we may rewrite the system of equations as

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{bmatrix} \mathbf{F} = \begin{bmatrix} A \\ (\Delta x)^2 r_1 \\ (\Delta x)^2 r_2 \\ \vdots \\ (\Delta x)^2 r_{N-1} \\ B \end{bmatrix}.$$

Note that it is possible to incorporate other types of boundary conditions by simply changing the first and last equations.

Let's define L to be the $(N+1) \times (N+1)$ matrix of coefficients for this equation, so that the equation has the form

$$L\mathbf{F} = \mathbf{b}.$$

The first thing to do is to verify that L is invertible, so that we know that there is a unique solution to the equation. It is not too difficult to compute the determinant if you recall that the elementary row operations that add a multiple of one row to another do not change the value of the determinant. Using only this type of elementary row operation, we can reduce L to an upper triangular matrix whose diagonal entries are $1, -2, -3/2, -4/3, -5/4, \ldots, -N/(N-1), 1$. The determinant is the product of these entries, and this equals $\pm N$. Since this value is not zero, the matrix L is invertible.

It is worthwhile pointing out that a change in boundary conditions (for example, prescribing the values of the derivative $f'(0)$ and $f'(1)$ rather than $f(0)$ and $f(1)$) results in a different matrix L that may fail to be invertible.

We should also ask about the condition number of L to determine how large the relative error of the solution can be. We will compute this using MATLAB/Octave below.

Now let's use MATLAB/Octave to solve this equation. We will start with the test case where $r(x) = 1$ and $A = B = 1$. In this case we know that the exact solution is $f(x) = 1 - x/2 + x^2/2$.

We will work with $N = 50$. Notice that, except for the first and last rows, L has a constant value of -2 on the diagonal, and a constant value of 1 on the off-diagonals immediately above and below.

Before proceeding, we introduce the MATLAB/Octave command `diag`. For any vector `D`, `diag(D)` is a diagonal matrix with the entries of `D` on the diagonal. So for example

```
>D=[1 2 3 4 5];
>diag(D)

ans =

   1   0   0   0   0
   0   2   0   0   0
   0   0   3   0   0
   0   0   0   4   0
   0   0   0   0   5
```

An optional second argument offsets the diagonal. So, for example

```
>D=[1 2 3 4];
>diag(D,1)

ans =

   0   1   0   0   0
   0   0   2   0   0
   0   0   0   3   0
   0   0   0   0   4
   0   0   0   0   0
```

```
>diag(D,-1)

ans =

   0   0   0   0   0
   1   0   0   0   0
   0   2   0   0   0
   0   0   3   0   0
   0   0   0   4   0
```

Now returning to our matrix L we can define it as

```
>N=50;
>L=diag(-2*ones(1,N+1)) + diag(ones(1,N),1) + diag(ones(1,N),-1);
>L(1,1) = 1;
>L(1,2) = 0;
>L(N+1,N+1) = 1;
>L(N+1,N) = 0;
```

The condition number of L for $N = 50$ is

```
>cond(L)
ans =  1012.7
```

We will denote the right side of the equation by `b`. To start, we will define `b` to be $(\Delta x)^2 r(x_i)$ and then adjust the first and last entries to account for the boundary values. Recall that $r(x)$ is the constant function 1, so its sampled values are all 1 too.

```
>dx = 1/N;
>b=ones(N+1,1)*dx^2;
>A=1; B=1;
>b(1) = A;
>b(N+1) = B;
```

Now we solve the equation for $\mathbf{F}$.

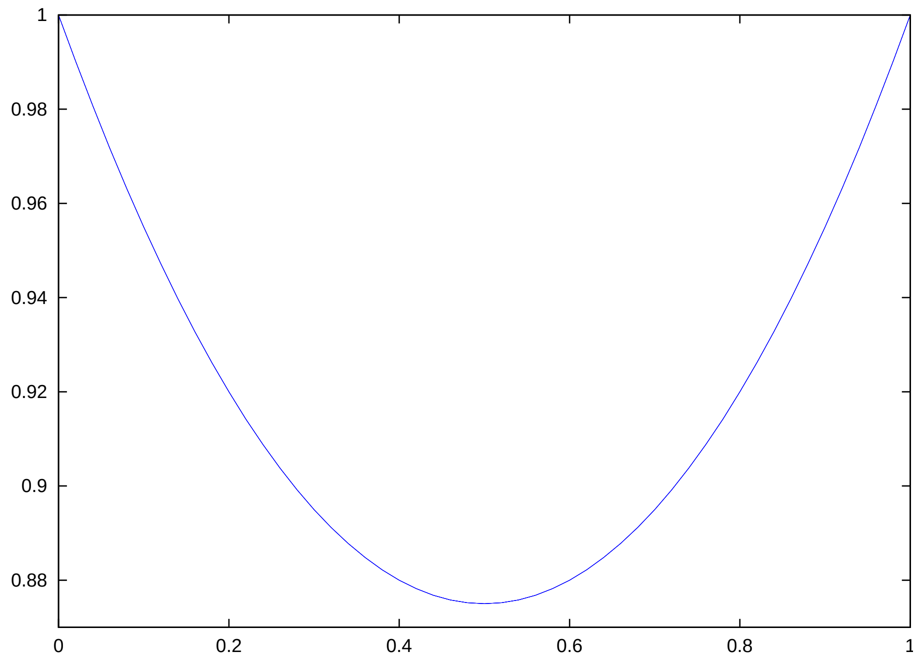
```
>F=L\b;
```

The x values are $N + 1$ equally spaced points between 0 and 1,

```
>X=linspace(0,1,N+1);
```

Now we plot the result.

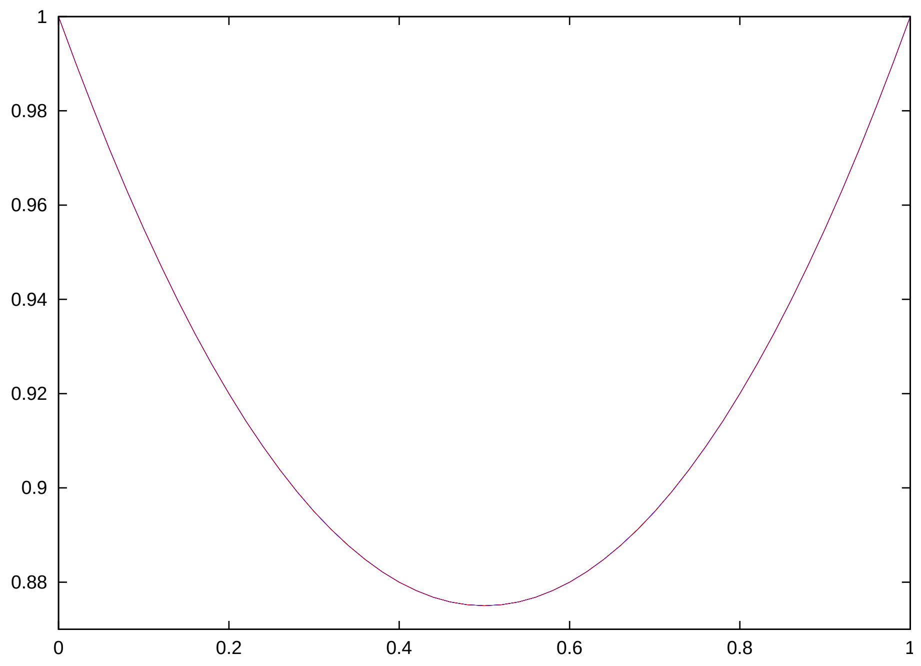
```
>plot(X,F)
```



Let’s superimpose the exact solution in red.

```
>hold on
>plot(X,ones(1,N+1)-X/2+X.^2/2,'r')
```

(The . before an operator tells MATLAB/Octave to apply that operator element by element, so X.^2 returns an array with each element the corresponding element of X squared.)



The two curves are indistinguishable.

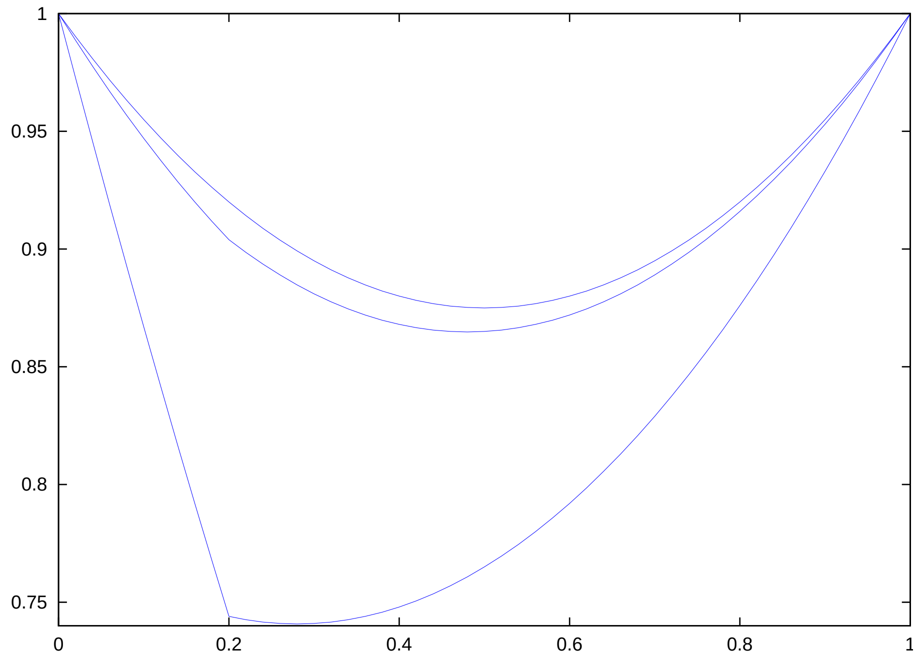
What happens if we increase the load at a single point? Recall that we have set the loading function $r(x)$ to be 1 everywhere. Let's increase it at just one point. Adding, say, 5 to one of the values of $\mathbf{r}$ is the same as adding $5(\Delta x)^2$ to the right side `b`. So the following commands do the job. We are changing $\mathbf{b}_{11}$ which corresponds to changing $r(x)$ at $x = 0.2$.

```
>b(11) = b(11) + 5*dx^2;
>F=L\b;
>hold on
>plot(X,F);
```

Before looking at the plot, let's do this one more time, this time making the cable really heavy at the same point.

```
>b(11) = b(11) + 50*dx^2;
>F=L\b;
>hold on
>plot(X,F);
```

Here is the resulting plot.



So far we have only considered the case of our equation $f''(x) + q(x)f(x) = r(x)$ where $q(x) = 0$. What happens when we add the term containing q? We must sample the function $q(x)$ at the interior points and add the corresponding vector. Since we multiplied the equations for the interior points by $(\Delta x)^2$ we must do the same to these terms. Thus we must add the term

$$(\Delta x)^2 \begin{bmatrix} 0 \\ q_1 f_1 \\ q_2 f_2 \\ \vdots \\ q_{N-1} f_{N-1} \\ 0 \end{bmatrix} = (\Delta x)^2 \begin{bmatrix} 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & q_1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & q_2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & q_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & q_{N-1} & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix} \mathbf{F}.$$

In other words, we replace the matrix L in our equation with $L + (\Delta x)^2 Q$ where Q is the $(N+1) \times (N+1)$ diagonal matrix with the interior sampled points of $q(x)$ on the diagonal.

I'll leave it to a homework problem to incorporate this change in a MATLAB/Octave calculation. One word of caution: the matrix L by itself is always invertible (with reasonable condition number). However $L + (\Delta x)^2 Q$ may fail to be invertible. This reflects the fact that the original differential equation may fail to have a solution for some choices of $q(x)$ and $r(x)$.

Let us briefly discuss what changes need to be made if we replace the the boundary condition $f(0) = A$ with a condition on the first derivative of the form $f'(0) = \alpha$.

Here is a summary of what we have done to model the condition $f(0) = A$. When we represent $f(x)$ by a vector $\mathbf{F} = [f_0, f_1, \ldots, f_N]^T$ the boundary condition is represented by the equation

$f_0 = A$. This equation corresponds to the first row in the matrix equation $L\mathbf{F} + (\Delta x)^2 Q\mathbf{F} = \mathbf{b}$ since the first row of L picks out f_0, the first row of Q is zero and the first entry of $\mathbf{b}$ is A.

A reasonable way to model the boundary condition $f'(0) = \alpha$ is to set the first entry in the vector $\mathbf{F}'$ equal to α. In other words we want $(f_1 - f_0)/(\Delta x) = \alpha$, or $(f_1 - f_0) = (\Delta x)\alpha$. So let's change the first row of our matrix equation $L\mathbf{F} + (\Delta x)^2 Q\mathbf{F} = \mathbf{b}$ so that it corresponds to this equation. To do this, change the first row of L to $[-1, 1, 0, \ldots, 0]$ and the first entry of $\mathbf{b}$ to $(\Delta x)\alpha$.

A similar change can be used to model a boundary condtion for $f'(1)$ on the other endpoint of our interval. You are asked to do this in a homework problem.

I.3.3. Another example: the heat equation

In the previous example involving the loaded cable there was only one independent variable, x, and as a result we ended up with an ordinary differential equation which determined the shape. In this example we will have two independent variables, time t, and one spatial dimension x. The quantities of interest can now vary in both space and time. Thus we will end up with a partial differential equation which will describe how the physical system behaves.

Imagine a long thin rod (a one-dimensional rod) where the only important spatial direction is the x direction. Given some initial temperature profile along the rod and boundary conditions at the ends of the rod, we would like to determine how the temperature, $T = T(x, t)$, along the rod varies over time.

Consider a small section of the rod between x and $x + \Delta x$. The rate of change of internal energy, $Q(x, t)$, in this section is proportional to the heat flux, $q(x, t)$, into and out of the section. That is

$$\frac{\partial Q}{\partial t}(x, t) = -q(x + \Delta x, t) + q(x, t).$$

Now the internal energy is related to the temperature by $Q(x, t) = \rho C_p \Delta x T(x, t)$, where ρ and C_p are the density and specific heat of the rod (assumed here to be constant). Also, from Fourier's law, the heat flux through a point in the rod is proportional to the (negative) temperature gradient at the point, i.e., $q(x, t) = -K_0 \partial T(x, t)/\partial x$, where K_0 is a constant (the thermal conductivity); this basically says that heat "flows" from hotter to colder regions. Substituting these two relations into the above energy equation we get

$$\begin{aligned} \rho C_p \Delta x \frac{\partial T}{\partial t}(x, t) &= K_0 \left(\frac{\partial T}{\partial x}(x + \Delta x, t) - \frac{\partial T}{\partial x}(x, t) \right) \\ \Rightarrow \frac{\partial T}{\partial t}(x, t) &= \frac{K_0}{\rho C_p} \frac{\frac{\partial T}{\partial x}(x + \Delta x, t) - \frac{\partial T}{\partial x}(x, t)}{\Delta x}. \end{aligned}$$

Taking the limit as Δx goes to zero we obtain

$$\frac{\partial T}{\partial t}(x, t) = k \frac{\partial^2 T}{\partial x^2}(x, t),$$

where $k = K_0/\rho C_p$ is a constant. This partial differential equation is known as the heat equation and describes how the temperature along a one-dimensional rod evolves.

We can also include other effects. If there was a temperature source or sink, $S(x,t)$, then this will contribute to the local change in temperature:

$$\frac{\partial T}{\partial t}(x,t) = k\frac{\partial^2 T}{\partial x^2}(x,t) + S(x,t).$$

And if we also allow the rod to cool down along its length (because, say, the surrounding air is a different temperature than the rod), then the differential equation becomes

$$\frac{\partial T}{\partial t}(x,t) = k\frac{\partial^2 T}{\partial x^2}(x,t) - HT(x,t) + S(x,t),$$

where H is a constant (here we have assumed that the surrounding air temperature is zero).

In certain cases we can think about what the *steady state* of the rod will be. That is after sufficiently long time (so that things have had plenty of time for the heat to "move around" and for things to heat up/cool down), the temperature will cease to change in time. Once this steady state is reached, things become independent of time, and the differential equation becomes

$$0 = k\frac{\partial^2 T}{\partial x^2}(x) - HT(x) + S(x),$$

which is of the same form as the ordinary differential equation that we considered at the start of this section.