Chapter I

Linear Equations

I.1 Solving Linear Equations

I.1.1 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 $\mathbf{b}$. 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 that 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 **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);
```

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)*x;toc();

Elapsed time is 44 seconds.

tic();A\x;toc();

Elapsed time is 13.55 seconds.
```

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

I.1.2 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 last year, 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.1.3 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(12,2:4)`:.

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.1.4 Matrix norm and condition number

There are many ways to measure the size of a matrix A. For example, we could consider A to be a large vector whose entries happen to be written in a box rather than in a column or a row. From this point of view a natural norm to use is

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

This norm is called the Hilbert-Schmidt norm. It has the advantage of being easy to compute.

However, when A is considered as a linear transformation or operator, acting on vectors, there is another norm that is natural to use. It is defined by

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

This norm measures the maximum factor by which A can stretch the length of a vector. An equivalent definition is

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

The reason why these definitions give the same answer is that the quantity $\|A\mathbf{x}\|/\|\mathbf{x}\|$ does not change if we multiply $\mathbf{x}$ by a scalar. So, when calculating the maximum in the first expression for $\|A\|$, we we need only pick one vector in any given direction, and we might as well choose the unit vector.

In general, there is no easy formula that computes $\|A\|$ from its entries. However, if A is a diagonal matrix there is. As an example 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

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

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

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

and taking the maximum over all unit vectors $\mathbf{x}$ yields $\|A\| \leq 3$. 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

$$\|A\| \geq \|A\mathbf{e}_1\| = 3$$

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
```

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.

So to measure how well conditioned a system of equations is, we take the ratio of the largest stretching factor to the smallest shrinking factor. Another way of saying this is to let $\mathbf{x}$ range over all unit vectors, and then take the ratio of the largest possible value of $\|A\mathbf{x}\|$ to the smallest possible value. This leads to the following definition for the condition number of an invertible matrix:

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

To see that this is the ratio of the largest stretching factor to the smallest shrinking factor note that

$$\begin{aligned}\min_{\mathbf{x}\neq 0}\frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} &= \min_{\mathbf{x}\neq 0}\frac{\|A\mathbf{x}\|}{\|A^{-1}A\mathbf{x}\|}\\ &= \min_{\mathbf{y}\neq 0}\frac{\|\mathbf{y}\|}{\|A^{-1}\mathbf{y}\|}\\ &= \frac{1}{\max\limits_{\mathbf{y}\neq 0}\dfrac{\|A^{-1}\mathbf{y}\|}{\|\mathbf{y}\|}}\\ &= \frac{1}{\|A^{-1}\|}\end{aligned}$$

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.

In our applications we will use the condition number as a measure of how well we can solve the equations that come up accurately. To see why the condition number is a good measure of this let's start with a linear system of equations $A\mathbf{x} = \mathbf{b}$ and 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 since

$$\begin{aligned}\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} &= \frac{\|A^{-1}\Delta\mathbf{b}\|}{\|A^{-1}\mathbf{b}\|}\\ &= \frac{\|A^{-1}\Delta\mathbf{b}\|}{\|\Delta\mathbf{b}\|}\frac{\|\mathbf{b}\|}{\|A^{-1}\mathbf{b}\|}\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}\\ &= \frac{\|A^{-1}\Delta\mathbf{b}\|}{\|\Delta\mathbf{b}\|}\frac{\|AA^{-1}\mathbf{b}\|}{\|A^{-1}\mathbf{b}\|}\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}\\ &\leq \|A^{-1}\|\|A\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}\\ &= \mathrm{cond}(A)\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}\end{aligned}$$

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.

Summary: Math Concepts

Review:

- Matrix vector notation for linear equations.
- Solving systems of equations using Gaussian Elimination.
- Deciding whether there is a unique solution, many solutions or no solution.
- Matrix inverse.

New:

- Hilbert Schmidt norm
- Matrix (operator) norm
- Condition number

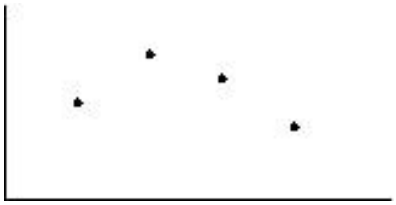
Summary: MATLAB/Octave Concepts

- Entering matrices (and vectors).
- Making larger matrices from smaller blocks.
- Multiplying matrices.
- Taking the inverse of a matrix.
- Taking the transpose of a matrix.
- Extracting elements, rows, columns, submatrices.
- Solving equations using `A\b`.
- Random matrices.
- Timing using `tic()` and `toc()`.
- Computing the norm and condition number.

I.2 Interpolation

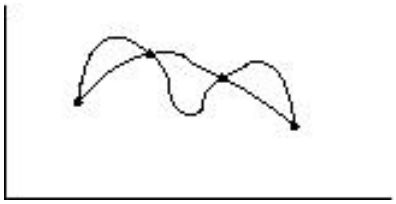
I.2.1 Introduction

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 its 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

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

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`.

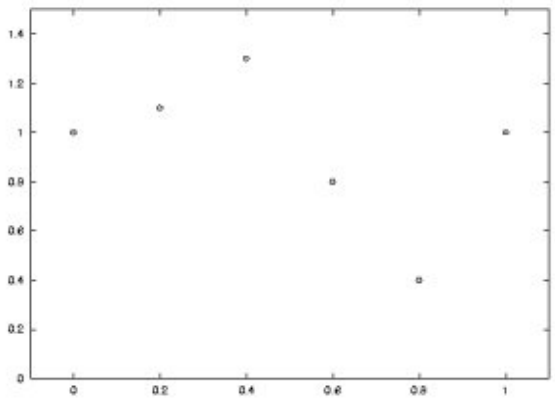
```
>X=[0 0.2 0.4 0.6 0.8 1.0]
>Y=[1 1.1 1.3 0.8 0.4 1.0]
```

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 determinant 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'
```

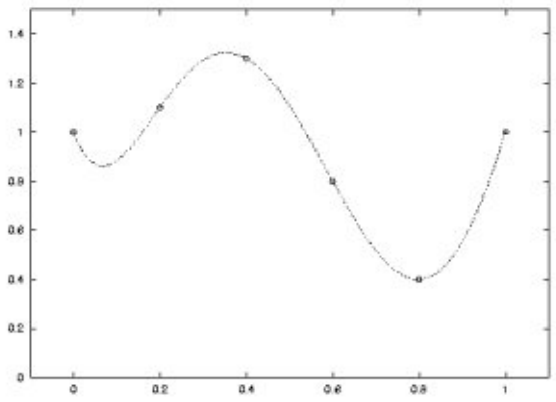
Let's have a look at the interpolating polynomial. The MATLAB/Octave function `polyval(a,X)` takes a vector `X` of x values, say $x_1, x_2, \ldots x_n$ and returns a vector containing the values $p(x_1), p(x_2), \ldots p(x_n)$, where p is the polynomial whose coefficients are in the vector `a`, that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

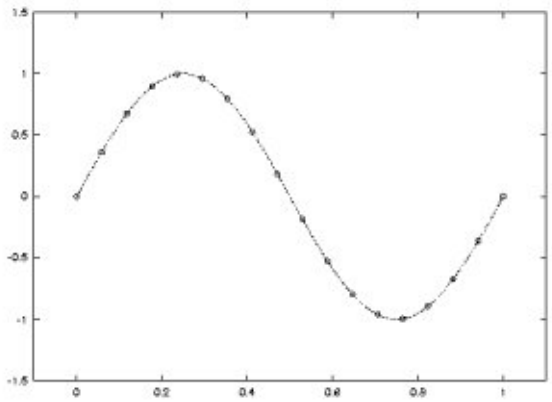


The MATLAB/Octave commands for this example are in `lagrange.m`.

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);
```

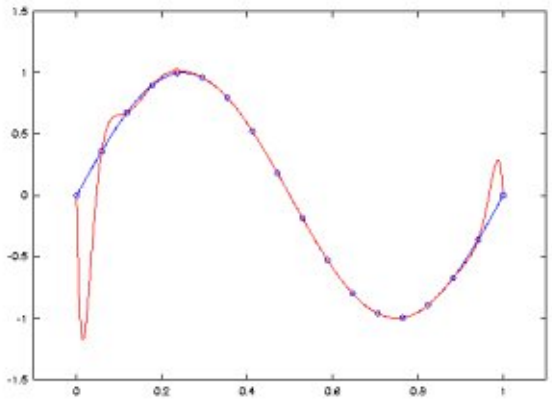
The resulting picture looks okay.



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 points between 0 and 1 the Vandermonde matrix is so ill conditioned that MATLAB/Octave considers it to be singular.

Summary: Math Concepts

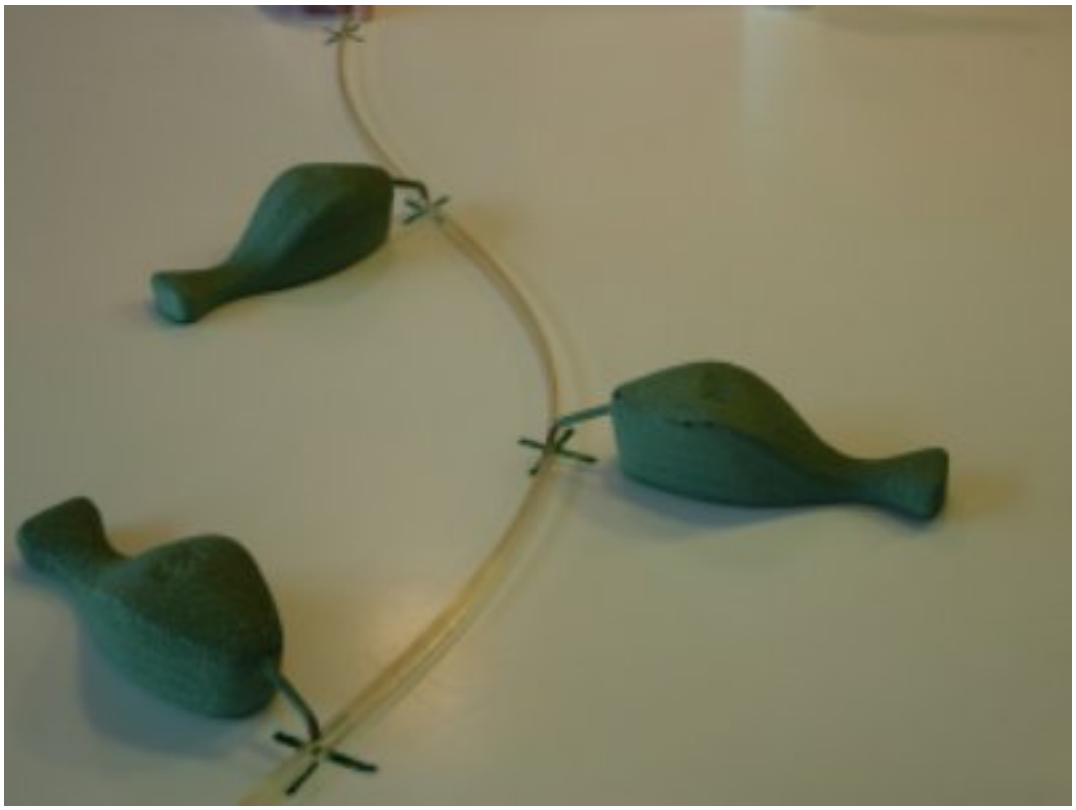
- Interpolation
- Lagrange interpolation by polynomials
- Finding the coefficients by solving a system of linear equations.
- Vandermonde matrix and how to compute its determinant.
- Why Lagrange interpolation is not a practical method for many points.

Summary: MATLAB/Octave Concepts

- Plotting basics, the function `plot`
- The functions `linspace`, `vander` and `polyval` and how to use them to plot the interpolating polynomial in Lagrange interpolation.

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.

Clearly, $f(x)$ should be a continuous function and it must pass through the given points. This results in the equations

$$f(x_i) = y_i$$

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. We will interpret the words "bendable but stiff" to mean that the first derivatives $f'(x)$ are continuous everywhere, including each interior duck location x_i.

For x in between the x_i we assume that f is perfectly smooth (that is, all derivatives exist) and that the higher derivatives have left and right limits at each x_i.

To proceed further 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

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

Now suppose that we have found the function f satisfying our constraints that minimizes $E[f]$. This means that among all functions satisfying the constraints, f is the one for which $E[f]$ is smallest. This means that $E[f + \epsilon h]$ can only be bigger, provided that $f(x) + \epsilon h(x)$ still satisfies our constraints. In other words there must be a local minimum for $E[f + \epsilon h]$ at $\epsilon = 0$. So

$$\left.\frac{dE[f + \epsilon h]}{d\epsilon}\right|_{\epsilon=0} = 0$$

provided f is our desired minimizer, and $f(x) + \epsilon h(x)$ satisfies our constraints for every ϵ.

We can make this second condition more explicit. The first thing we require is for every i that $f(x_i) + \epsilon h(x_i) = y_i$ for any ϵ. This only happens when

$$h(x_i) = 0$$

for every i. The second thing we require is that $f'(x) + \epsilon h'(x)$ be continuous across each interior x_i, again for any ϵ. Since we are already assuming that f' is continuous, h' must be continuous at each interior x_i too.

Now we compute the derivative with respect to epsilon and integrate by parts twice to obtain

$$\begin{aligned}
0 = \left.\frac{dE[f+\epsilon h]}{d\epsilon}\right|_{\epsilon=0} &= \int_{x_1}^{x_n} 2\,(f''(x)'' + \epsilon h''(x))h''(x)\big|_{\epsilon=0}\,dx \\
&= 2\int_{x_1}^{x_n} f''(x)h''(x)dx \\
&= 2\sum_{i=1}^{n-1}\int_{x_i}^{x_{i+1}} f''(x)h''(x)dx \\
&= 2\sum_{i=1}^{n-1}\left(f''(x)h'(x)\big|_{x=x_i}^{x=x_{i+1}} - f'''(x)h(x)\big|_{x=x_i}^{x=x_{i+1}} + \int_{x_i}^{x_{i+1}} f''''(x)h(x)dx\right) \\
&= 2\sum_{i=1}^{n-1}\left(f''(x)h'(x)\big|_{x=x_i}^{x=x_{i+1}} + \int_{x_i}^{x_{i+1}} f''''(x)h(x)dx\right)
\end{aligned}$$

In the last line we used that $h(x_i) = 0$. Now we rearrange the terms, keeping in mind that h' is continuous at each x_i. This gives (dividing by 2)

$$0 = -h'(x_1)f''(x_1+) + \sum_{i=2}^{n-1} h'(x_i)(f''(x_i+) - f''(x_i-)) + h'(x_n)f''(x_n+) + \sum_{i=1}^{n-1}\int_{x_i}^{x_{i+1}} f''''(x)h(x)dx$$

For f to be the minimizer, this equation has to be true for every admissible choice of h. In particular, we could choose h that is zero everywhere except in the open interval (x_i, x_{i+1}). For all such h we then would obtain $0 = \int_{x_i}^{x_{i+1}} f''''(x)h(x)dx$. This can only happen if

$$f''''(x) = 0 \quad \text{for} \quad x_i < x < x_{i+1}$$

But once we know this, we need only choose h with one $h'(x_i)$ equal to 1 and all the others zero to conclude

$$\begin{aligned}
&f''(x_1+) = 0 \\
&f''(x_i+) = f''(x_i-) \quad \text{for} \quad i = 2, \ldots, n-1 \\
&f''(x_n-) = 0
\end{aligned}$$

In particular, f'' must be continuous across each interior duck position.

One final point about this derivation: By integrating four times, we see that the condition $f''''(x) = 0$ in each interval (x_i, x_{i+1}) is the same as saying that $f(x)$ is a cubic polynomial.

We can summarize the discussion as follows: The mathematical description of the shape of a spline is a function $f(x)$ that is a cubic polynomial in each interval (x_i, x_{i+1}). The polynomial can change from interval to interval, but the resulting function f must be continuous and pass through the specified points. The first *and second* derivatives $f'(x)$ and $f''(x)$ must be continuous as well. Finally we require $f'' = 0$ at both endpoints.

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

I.2.4 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

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

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

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

We make one more simplification. After multiplying some of the equations with suitable powers of L we can write these as equations for $\alpha_i = a_i L^3$, $\beta_i = b_i L^2$ and $\gamma_i = c_i L$. 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 coefficient 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;
```

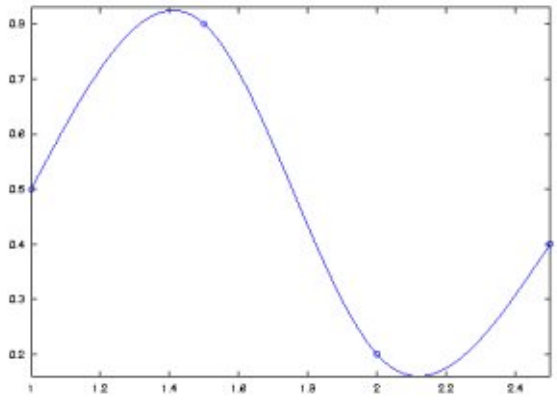
Now we evaluate the polynomial $p_1(x)$ and plot the resulting points.

```
>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



I have automated the procedure above and put the result in two files `splinemat.m` and `plotspline.m`. `splinemat(n)` returns the $(n-1)\times(n-1)$ matrix used to compute a spline through n points while `plotspline(X,Y)` plots the cubic spline going through the points in `Y` 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.

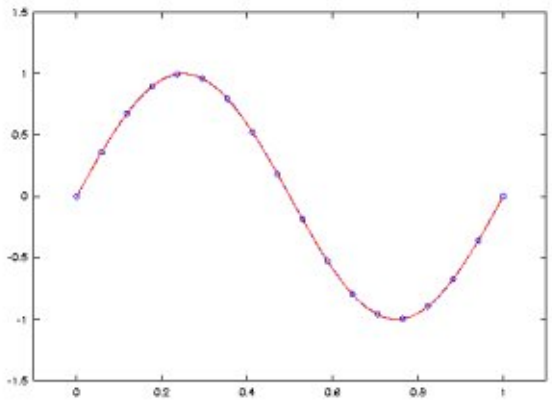
Let's use these functions to compare the cubic spline interpolation with the Lagrange interpolation by using the same points as we did before. Remember that we started with the points

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

Let's plot the spline interpolation of these points

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

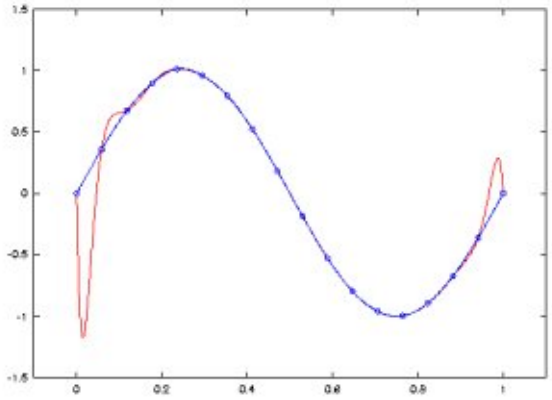
Here is the result with the Lagrange interpolation added (in red). The red (Lagrange) curve covers the blue one and its impossible to tell the curves apart.



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!!

Summary: Math Concepts

- Finding the shape with the smallest bending energy
- Cubic splines and how to compute them.

Summary: MATLAB/Octave Concepts

- The functions `zeros` and `ones`.
- Using m files.

I.3 Finite difference approximations

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

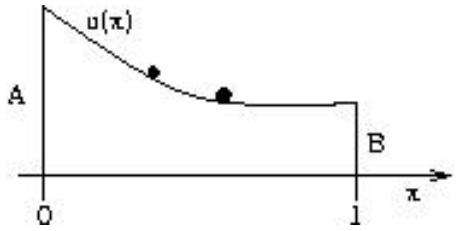
$$f''(x) + q(x)f(x) = r(x) \quad \text{for} \quad 0 \leq x \leq 1$$

subject to boundary conditions

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

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 $0 \leq x \leq 1$. 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 bending 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

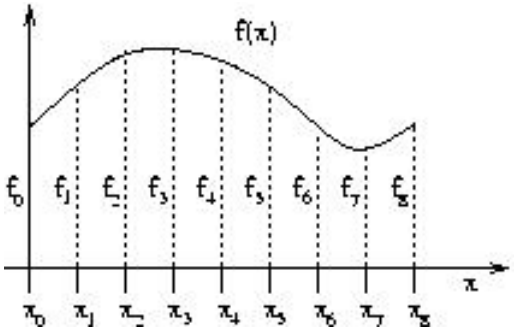
$$f''(x) = r(x)$$

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.

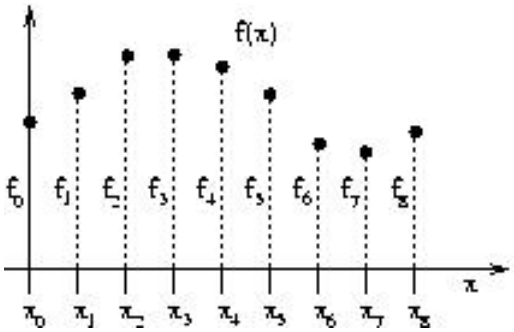
I.3.2 Discretization

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.

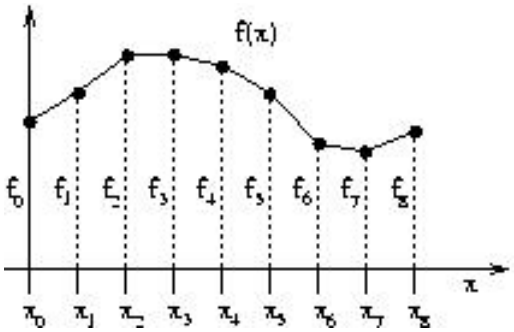
$$F = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_N \end{bmatrix}.$$



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 be 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 F. The formula for the slope in the interval (x_i, x_{i+1}) is $(f_{i+1} - f_i)/h = N(f_{i+1} - f_i)$ where $h = 1/N$ is the distance $x_{i+1} - x_i$. Thus the vector containing the slopes is

$$F' = N \begin{bmatrix} f_1 - f_0 \\ f_2 - f_1 \\ f_3 - f_2 \\ \vdots \\ f_N - f_{N-1} \end{bmatrix} = N \begin{bmatrix} -1 & 1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 1 & 0 & \cdots & 0 \\ 0 & 0 & -1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_N \end{bmatrix} = ND_N F$$

where D_N is the $N \times (N+1)$ finite difference matrix in the formula above. The vector 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 F''. There will be one entry in this vector for each adjacent pair of slopes, that is, each adjacent pair of entries of F'. These are naturally labelled by the interior points $x_1, x_2, \ldots, x_{n-1}$. Thus we obtain

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

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 F'' is defined. Now we can write down the finite difference approximation to $f''(x) = r(x)$ as

$$N^2 D_{N-1} D_N F = \mathbf{r}$$

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! Instead of writing two extra equations, we will simply set $f_0 = A$ and $f_N = B$ in the vector F. Let's denote the vector of the remaining (unknown) interior values by $\mathbf{f}$. Then, moving the terms involving A and B to the right side of the equation and dividing by N^2, we may rewrite the equation as

$$\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -2 & 1 & \cdots & 0 & 0 \\ 0 & 1 & -2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & -2 \end{bmatrix} \mathbf{f} = N^{-2}\mathbf{r} - \begin{bmatrix} A \\ 0 \\ 0 \\ \vdots \\ B \end{bmatrix}$$

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 $N = 50$. The first thing is to define the $(N-1) \times (N-1)$ matrix on the left, which we will call L Notice that 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 `diag(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-2),1) + diag(ones(1,N-2),-1);
```

We will denote the right side of the equation by `b`. To start, we will define `b` to be $N^{-2}\mathbf{r}$ 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. BR/¿

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

Now we solve the equation for $\mathbf{f}$ and attach the boundary values on either side to define the vector F

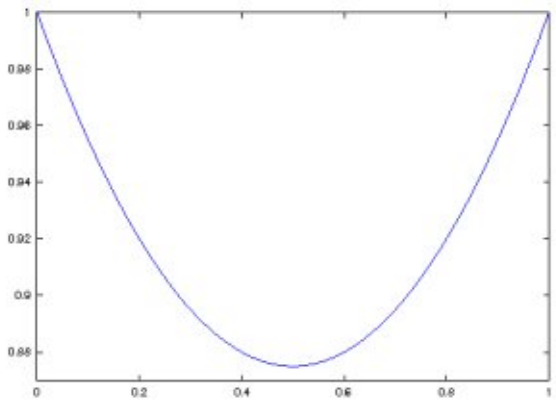
```
>f=L\b;
>F=[A;f;B];
```

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

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

Now we plot the result.

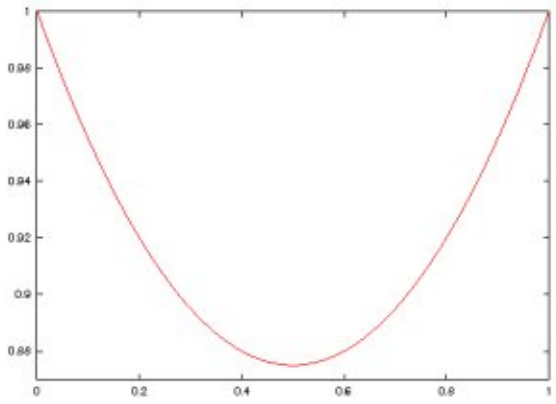
```
>plot(X,F)
```



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.

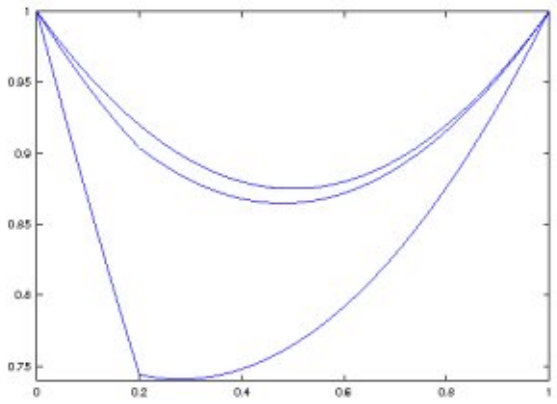
What happens if we increase the load at a single point? Recall that we have set the loading function $r(x)$ to be 1 everywhere. Let's increase it at just one point. Adding, say, 5 to one of the values of $\mathbf{r}$ is the same as adding $5/N^2$ to the right side `b`. So the following commands do the job. We are changing $\mathbf{r}_{10}$ which corresponds to changing $r(x)$ at $x = 0.2$.

```
>b(10) = b(10) + 5/N^2;
>f=L\b;
>F=[A;f;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(10) = b(10) + 50/N^2;
>f=L\b;
>F=[A;f;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 $q(x)(x)$ at the interior points and add the corresponding vector. Since we divided the whole equation by N^2 we must do the same to this term. Thus we must add the term

$$N^{-2}\begin{bmatrix} q_1 f_1 \\ q_2 f_2 \\ \vdots \\ q_{N-1} f_{N-1} \end{bmatrix} = \begin{bmatrix} q_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & q_2 & 0 & \cdots & 0 & 0 \\ 0 & 0 & q_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & q_{N-1} \end{bmatrix} \mathbf{f}$$

In other words, we replace the matrix L in our equation with $L + N^{-2}Q$ 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 $L + N^{-2}Q$ 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 $r(x)$.

Summary: Math Concepts

- Discretizing a differential equation $f''(x) + f(x)q(x) = r(x)$

Summary: MATLAB/Octave Concepts

- The `diag(Q,n)` function