Chapter 4 – Notes

2/23/07

Matrix Inverse

The inverse of a matrix A is defined as A^{-1} so that AA-1 = I, where I is the identity matrix. (In MATLAB, the function `eye` returns the identity matrix.) If we define $\vec{e}_i$ as the vector

$$\vec{e}_i \equiv \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

where the i-th row has the number 1 and the rest of the vector is zeros, then

$$I = \left[\vec{e}_1, \vec{e}_2, \vec{e}_3, \cdots \vec{e}_N\right]$$

If we solve the set of equations $A\vec{x}_i = \vec{e}_i$, then

$$A^{-1} = \left[\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_N\right]$$

for a matrix A of size N.

In MATLAB, one can simply find the inverse of a matrix A by using the `inv` command, so that A^{-1} = inv(A). Here we will discuss briefly some issues related to matrix inverse computation.

Singular and ill-conditioned matrices

Consider the simple matrix

$$A = \begin{pmatrix} 1+\varepsilon & 1 \\ 2 & 2 \end{pmatrix}$$

the determinant of A is 2ε. It ε is very small, the A is close to singular. One measure of how close a matrix is to singular is via the condition number which is a measure of the distance of this matrix to the nearest singular matrix. (This all depends on your choice of norm, see documentation on cond in MATLAB.) In MATLAB the function cond(A) returns the condition number of the matrix, where the larger the condition number the closer it is to singular. Here are a couple of things to note regarding poorly conditioned matrices:

1. As a rule of thumb, `log10(cond(A))` is the number of significant digits you can expect to lose in solving a matrix by Gaussian elimination.
2. Often a matrix that is close to singular is a clue that there is something missing (or even wrong) in your problem setup.
3. Another way to interpret ill-conditioned matrices is that small perturbations in the matrix can produce large changes in the solution.

Gaussian Elimination – Simple Example

Lets try doing some simple Gaussian elimination for the following matrix equation $A\vec{x} = \vec{b}$, the goal is to find the solutions for (x_1, x_2, x_3)

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 2 & 1 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

We will do this by doing a series of successive matrix multiplications, the goal is to get the matrix equation in the form $U\vec{x} = \vec{c}$:

$$\begin{pmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & f \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} g \\ h \\ i \end{pmatrix}$$

where the variables a, b…i are to be determined. The matrix U is known as a upper triangular matrix. Let start by making the 2 lower left parts of the matrix zero. One way to do this is to multiply both sides by the matrix

$$M_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}$$

so that we have

$$M_1 A\vec{x} = M_1\vec{b} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 2 & 1 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

and results in

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

we will leave the RHS untouched for now. To eliminate the bottom number we do the same thing and multiply by a matrix M_2

$$M_2 M_1 A\vec{x} = M_2 M_1\vec{b} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & -1 \\ 0 & -1 & 0 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

so that we have

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

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

by back substitution we can see that the solution is $x_3 = 1$, $x_2 = -1$ and $x_1 = 1$. Notice that once we have the matrix equation in the form $U\vec{x} = \vec{c}$, solving for the unknown $\vec{x}$ is easy. Similarly if one can reduce the equation to be in the form $L\vec{x} = \vec{c}$ where L is a lower triangular matrix of the form

$$L = \begin{pmatrix} a & 0 & 0 \\ b & c & 0 \\ d & e & f \end{pmatrix}$$

then it is equally easy to compute the solution. In general, if one can reduce a matrix problem to be of the above form (i.e, $U\vec{x} = \vec{c}$) then solution is easy. This is known as LY decomposition. One can show that any matrix can be expressed as A = LU. In the above example, it is easy to show that

$$L = M_1^{-1}M_2^{-1} \tag{1}$$

Since

$$U = M_2M_1A$$

and A = LU, then

$$U = M_2M_1LU$$

which implies $M_2M_1L = \mathrm{I}$ where I is the identity matrix, this automatically leads to (1). From above

$$M_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix} \text{ and } M_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

so that $M_1^{-1}M_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 1 & 1 \end{pmatrix}$

which is L.

Note on Norms in MATLAB

For a given vector $\vec{x}$ of length N the norm is defined as follows

p-norm
$$\|\vec{x}\|_p \equiv \left(\sum_{i=1}^{N}|x_i|^p\right)^{1/p}$$

1-norm
$$\|\vec{x}\|_1 \equiv \left(\sum_{i=1}^{N}|x_i|\right)$$

2-norm
$$\|\vec{x}\|_\infty \equiv \max_i|x_i|$$

∞-norm
$$\|\vec{x}\|_2 \equiv \left(\sum_{i=1}^{N}|x_i|^2\right)^{1/2}$$

For a matrix A of size NxN:

1-norm – absolute column sum

$$\|A\|_1 \equiv \max_j \left(\sum_{i=1}^{N} |A_{ij}| \right)$$

∞-norm – absolute row sum

$$\|A\|_\infty \equiv \max_j \left(\sum_{j=1}^{N} |A_{ij}| \right)$$

In MATLAB, the command is `norm(A,1)` for the 1-norm and `norm(A,inf)` for the infinity norm. For example, if

$$A = \begin{pmatrix} 2 & -1 & 1 \\ 1 & 0 & 1 \\ 3 & -1 & 4 \end{pmatrix}$$

then $\|A\|_1 = 6$ and $\|A\|_\infty = 8$.

Non Linear Equations

(Section 4.3 of the text)

A more interesting set of problems comes when one wants to solve for x* for a given equation f(x*) = 0 where f(x) is some function. A physical example of such an equation would be finding the depth of a floating sphere radius R and density ρ suspended at depth h in a liquid of density 1. See Figure 1. To solve this problem, we need to compute the mass M of water displaced by the sphere, this is given by

$$M = \int_0^h \pi(R^2 - (R-y)^2)dy$$

$$= \pi\left(Rh^2 - \frac{h^3}{3} \right)$$

According to Archimedes, the mass of the water displaced should equal the total mass of the ball, this leads to the equation

$$\frac{4}{3}\rho\pi R^3 = \pi\left(Rh^2 - \frac{h^3}{3} \right)$$

$$f(h) = 4\rho R^3 - \left(3Rh^2 - h^3 \right)$$

which is a nonlinear equation for h, for which we want to find h* so that f(h*)=0.

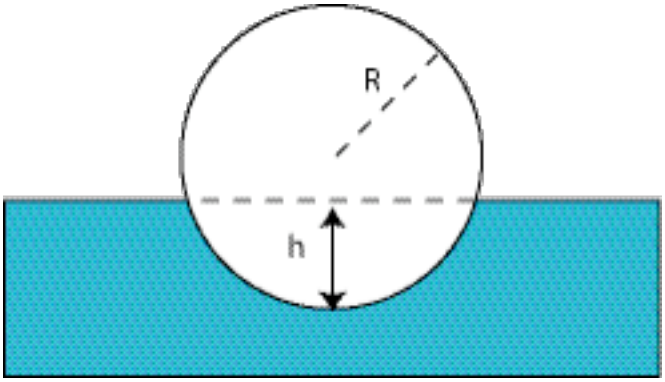

Figure 1: Floating Sphere problem

Newton's Method

To solve for h, we will resort to a numerical iterative method. The simplest and probably most commonly used approach is Newton's method, which is an iterative method. Figure 2 illustrates the basic technique. If at iteration n our guess for the root is given by x_n and we wish to improve our guess at the next iteration at x_{n+1}.

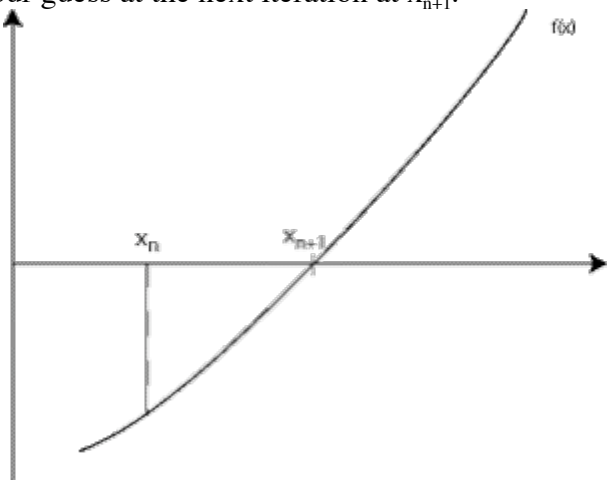

Figure 2: Newton's method

Given that: $x_{n+1} = x_n + \delta x$ and that we want $f(x_{n+1}) = 0$, we can expand about xn+1 using a Taylor series

$$f(x_{n+1}) = f(x_n + \delta x) = 0$$
$$= f(x_n) + \delta x f'(x_n) + O(\delta x)^2$$

solving for δx, we get

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

so that

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}}$$

This is Newton's method.

Notes regarding Newton's method.

1. The key to finding a root successfully is a good first guess.
2. If a function has multiple roots, the method will converge to the 'nearest' root. The solution you get will depend on the starting point.
3. Sometimes the method diverges (eg if f'(x)=0 at some location)

Getting back to our sphere problem, then the function is

$$f(h) = 4\rho R^3 - \left(3Rh^2 - h^3\right) \text{ and } f'(h) = -\left(6Rh - 3h^2\right)$$

so that Newton's method is for guess h_{n+1} is

$$h_{n+1} = h_n + \frac{4\rho R^3 - (3Rh_n^2 - h_n^3)}{(6Rh_n - 3h_n^2)}$$

An example MATLAB program 'floatsphere.m' is given below:

```
% floating sphere problem
% finds and plots the floating point of a sphere radius 1
% and density rho (less than 1)
help floatsphere
clear all
format compact

rho = 0.4;
rho=input(' please input rho ( < 1): ')
h = 0.5; % first guess
hnew = 1;

eps = abs(hnew - h);
error = 1e-7; % error tolerance

iter = 0;
itermax = 1000; % max number of iterations

while (eps > error && iter < itermax)
    hnew = h - (4*rho - h^2*(3-h))/(3*h^2-6*h)
    eps = abs(hnew - h);
    h = hnew;
    iter = iter + 1;
end

fprintf('at iteration number %g the error is %g and the depth = %g
\n',iter,eps,h);

hplot = 0:0.1:2*h;

figure(1)
plot(hplot,(4*rho - (hplot.^2).*(3-hplot)),'-',h,(4*rho - h^2*(3-h)),'+')
xlabel('depth')
ylabel('force curve')
legend('force curve','solution')
grid

% generate a circle, radius 1 centered at  (1-h)
figure(2)
plot(sin(0:pi/20:2*pi),1-h+cos(0:pi/20:2*pi),'g-',-2:2,0*(-2:2),'b-')
axis equal
```

The program iterates until the solution stops changing within some specified error tolerance. Figure 3 shows an example output for a density of 0.3.

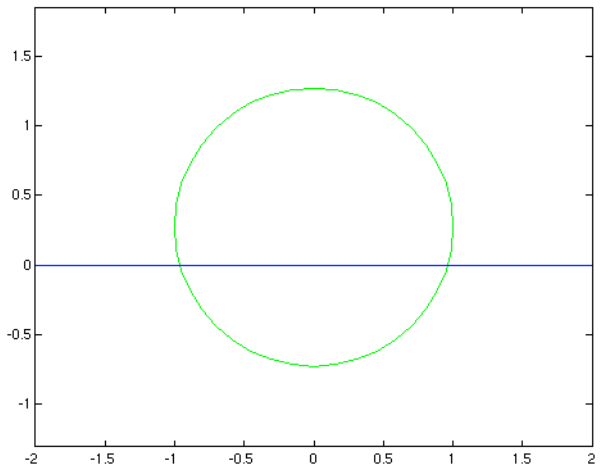

Figure 3: Output for the floatsphere.m program for ρ=0.3.

Multivariable Newton's Method

Newton's method can be generalized to an N-variable problem. If the unknowns are now a vector $\vec{x}$ and the function we are solving for is

$$\vec{f}(\vec{x}) = (f_1(\vec{x}) \quad f_2(\vec{x}) \quad \cdots \quad f_N(\vec{x}))$$

then the method becomes

$$\vec{f}(\vec{x}_{n+1}) = \vec{f}(\vec{x}_n) - \delta\vec{x}D(\vec{x}_n)$$

where

$$D_{ij}(\vec{x}_n) \equiv \frac{\partial f_j(\vec{x}_n)}{\partial x_i}$$

the next iteration is then

$$\vec{x}_{n+1} = \vec{x}_n - \vec{f}(\vec{x})D^{-1}(\vec{x}_n)$$

An example of this method is in the `newtn.m` program from the text.