Chapter 4 – Notes

3/3/20

Coupled Harmonic Oscillators

[pp. 119 of Garcia]

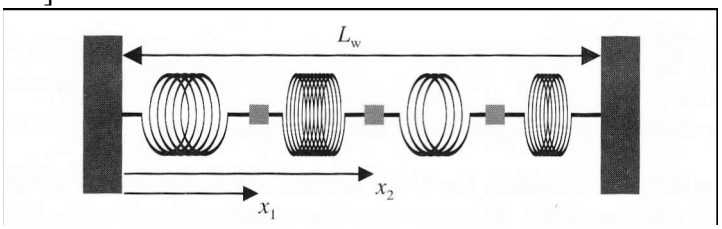

Figure 1: From figure 4.3 of Garcia

From the above figure we have 3 masses coupled by 4 springs attached at both ends to a wall. Each mass m_i at location x_i has a spring of constant k_i and rest length L_i attached to its left and another spring of constant k_{i+1} and rest length L_{i+1} to its right. If we compute the forces on each mass as

$$\begin{aligned} F_1 &= -k_1(x_1 - L_1) + k_2(x_2 - x_1 - L_2) \\ F_2 &= -k_2(x_2 - x - L_2) + k_3(x_3 - x_2 - L_3) \\ F_3 &= -k_3(x_3 - x_2 - L_3) + k_4(L_W - x_3 - L_4) \end{aligned} \qquad (1)$$

In matrix form, it takes

$$\begin{pmatrix} F_1 \\ F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} -k_1 - k_2 & k_2 & 0 \\ k_2 & -k_2 - k_3 & k_3 \\ 0 & k_3 & -k_3 - k_4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} - \begin{pmatrix} -k_1 L_1 - k_2 L_2 \\ -k_2 L_2 + k_3 L_3 \\ -k_3 L_3 + k_4(L_4 - L_W) \end{pmatrix}$$

which has the generic form $\boldsymbol{F} = \boldsymbol{K}\vec{x} - \vec{b}$. In static equilibrium, **F=0,** which then leads to the matrix problem $\boldsymbol{K}\,\vec{x} = \vec{b}$, the solution to which is $\vec{x} = K^{-1}\,\vec{b}$. One thing to note about the matrix **K** - it is symmetric.

An alternative approach, which is somewhat easier in the sense that it can take care of the sign for you (provided you are consistent), is to construct a Lagrangian. The kinetic energy, T simply becomes

$$T = \frac{1}{2}\sum_{k=1}^{3} m\dot{x}_i^2$$

while the potential energy, V, is

$$V = \frac{1}{2}k_1(x_1 - L_1)^2 + \frac{1}{2}k_2(x_2 - x_1 - L_2)^2 + \frac{1}{2}k_3(x_2 - x_1 - L_3)^2 + \frac{1}{2}k_4(L_W - x_3 - L_4)^2$$

In equilibrium, the acceleration is zero, and Lagrange's equation becomes

$$\begin{aligned} \frac{\partial V}{\partial x_1} &= k_1(x_1 - L_1) - k_2(x_2 - x_1 - L_2) = 0 \\ \frac{\partial V}{\partial x_2} &= k_2(x_2 - x_1 - L_2) - k_3(x_3 - x_2 - L_3) = 0 \\ \frac{\partial V}{\partial x_3} &= k_3(x_3 - x_2 - L_3) - k_4(L_W - x_3 - L_4) = 0 \end{aligned}$$

which yields the same set of equations as in (1).

Notes about MATLAB's and Python's Inverse

If you have an equation $A\vec{x} = \vec{b}$, the solution for column vector $\vec{x}$ can be easily obtained as $x = inv(A)b$, however according to the MATLAB documentation a much faster and more accurate way is to use the back slash (\) command $x = A \setminus b$. If $\vec{x}$ is a row vector, then the matrix problem is written as $\vec{x}A = \vec{b}$ in which case you use the forward slash command $x = b / A$.

In MATLAB there is the `inv` function that inverts matrices as well as '\'. For example:

```
>> a=[1 3;5 8]
a =
     1     3
     5     8
>> b=[4;5]
b =
     4
     5
>> a\b
ans =
  -2.428571428571428
   2.142857142857143
>> inv(a)*b
ans =
  -2.428571428571429
   2.142857142857143
```

In general, '\' is faster.

In python you can use the numpy.linalg.inv function (np.linalg.inv). When you do matrix multiplication in numpy you have to use the np.dot function. For details, consult the online documentation. As an alternative, numpy also has matrix objects that you can use instead.

Notes about 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

$$\vec{e_i} \equiv \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$$

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

$$I = [\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots, \vec{e}_N]$$

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

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

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.
[In python numpy, the equivalent command is np.linalg.inv(), see http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.inv.html#numpy.linalg.inv for details.]

Singular and ill-conditioned matrices

Consider the simple matrix

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

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.

Aside - 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

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

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

$$M_2M_1A\,\vec{x} = M_2M_1\vec{b} = \begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & -1 & 1\end{pmatrix}\begin{pmatrix}1 & 1 & 1\\ 0 & -1 & -1\\ 0 & -1 & 1\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix}$$

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

so that we have

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

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

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, $U\vec{x} = \vec{c}$ or $L\vec{x} = \vec{c}$) then solution is easy. This is known as *LU* decomposition. One can show that any matrix can be expressed as A = LU. In the above example, it is straightforward to show that

$$L = M_1^{-1}M_2^{-1} \qquad (2)$$

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 (2). From above

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

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

which is L.

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 largest column sum

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

2-norm

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

∞-norm $\|\vec{x}\|_\infty \equiv \max|x_i|$

largest row sum

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 $\|A\|_\infty = 8$. In other words:

$\|A\|_1 = \max(6,2,5) = 6$ and $\|A\|_\infty = \max(4;2;8) = 8$

[In python, using numpy there is an equivalent command np.linalg.norm, see http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.norm.html for details.]

Non Linear Equations

(Section 4.3 of the Garcia 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 2. 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 which 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.

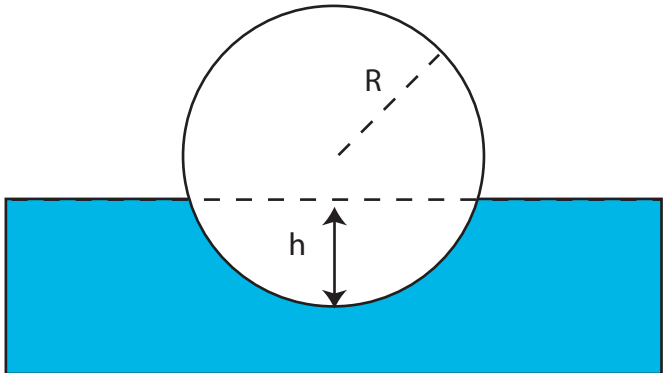

Figure 2: Floating Sphere problem

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}.

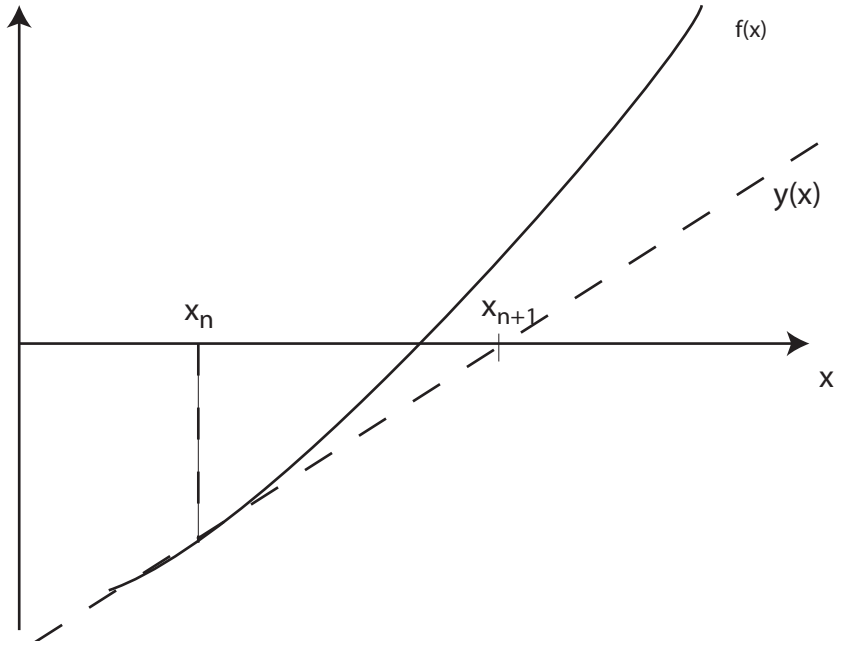

Figure 2: Newton's method

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

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}} \qquad (2)$$

This is Newton's method.

Notes on 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 (e.g., if f'(x)=0 at some location).
4. The basic idea behind the Newton method is that at the point (x_m, $f(x_n)$) we fit a straight line y(x) that goes through that point and has slope $f'(x_n)$ (Shown by the dashed line in figure 2). We can write *y(x)* as $y(x) = f'(x_n)(x - x_n) + f(x_n)$. We then find the location x_{n+1} where the line crosses the x-axis, i.e., where $y(x_{n+1}) = 0$, $f'(x_n)(x_{n+1} - x_n) + f(x_n) = 0$, solving for x_{n+1} we get equation (2).

One simple example of using Newton's method is to solve for the root of an equation, for example if you wanted to find x so that $x^m - a = 0$, the solution to which is the *m*-th root of a. Using Newton's method above the successive iterations for getting the root are

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

$$= x_n - \frac{x_n^m - a}{m x_n^{m-1}}$$

For example, if m=2 and a =2 and x_1 =1, then

$$x_2 = 1 - \frac{1^2 - 2}{2} = \frac{3}{2}$$

$$x_3 = \frac{3}{2} - \frac{\left(\frac{3}{2}\right)^2 - 2}{3} = \frac{17}{12} \approx 1.416$$

which is good to 2 decimal places after only 2 iterations.

Now, getting back to our sphere problem, then the function

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

Dividing both sides by R^3 and defining H=h/R

$g(H) = f(h)/R^3 = 4\rho - \left(3(H)^2 - (H)^3\right)$ and $g'(H) = -\left(6H - 3(H)^2\right)$

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

$$H_{n+1} = H_n + \frac{4\rho - (3(H_n)^2 - (H_n)^3)}{(6H_n / R - 3H_n^{\,2})}$$

The routine '`floatsphere.m`' computes the Newton iteration for this problem.

Secant Method

An alternative method when the derivative of the function is not known (or is hard or expensive to calculate) is to replace the derivative f'(x) with a finite difference approximation. So the equation becomes

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

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

This is known as the secant method, and usually does not converge as quickly as the Newton method.

An example MATLAB program '`floatsphere.m`' includes the secant method:

```
% 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=input(' please input rho ( < 1): ')

h_start = 1
hold_start = 0; % for the secant method

h = h_start; % first guess
eps = 0.0;
error_sol_newton =  abs(force_sphere(h,rho));
error_iter_new(1) = error_sol_newton;
error = 1e-7; % error tolerance

itermax = 1000; % max number of iterations

for iter=2:itermax
if(eps > error || error_sol_newton > error)
    hnew = h - force_sphere(h,rho)/(3*h^2-6*h)
    eps = abs(hnew - h)
    error_sol_newton = abs(force_sphere(hnew,rho))
    h = hnew;
    error_iter_new(iter) = error_sol_newton;
else
    break
end
end

hnewton=h;

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


% -----------now solve the same equation using the secant method-------
--

h = h_start; % first guess
h_old = hold_start; % need an hold

eps = 0;
error_sol_secant = abs(force_sphere(h,rho));
error_iter_sec(1) = error_sol_secant;

itermax = 1000; % max number of iterations

for iter=2:itermax
if(eps > error || error_sol_secant > error)
    hnew = h - force_sphere(h,rho)/(force_sphere(h,rho)-
force_sphere(h_old,rho))*(h-h_old);
    eps = abs(hnew - h);
    error_sol_secant = abs(force_sphere(hnew,rho));
    h = hnew;
    error_iter_sec(iter) = error_sol_secant;
else
    break
end
end
```

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

% now plot the solition

hplot = 0:0.1:2*h;

figure(1)
plot(hplot,force_sphere(hplot,rho),'-
',hnewton,force_sphere(hnewton,rho),'+')
xlabel('depth')
ylabel('force curve')
legend('force curve','solution')
title('solution from Newton')
grid on

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

itermax=max(length(error_iter_new),length(error_iter_sec));
if(itermax > 1)
figure(3)
clf
semilogy([1:length(error_iter_new)],error_iter_new,'r-
',[1:length(error_iter_sec)],error_iter_sec,'b-')
xlabel('iteration')
ylabel('error')
legend('Newton','Secant')
axis([1,itermax,0 2])
grid
else
    fprintf('Lucky guess, got it first time!\n')
end
```

```
function f=force_sphere(h,rho)
% force term for the floating spshere problem'
% frt 2/08
  f = 4*rho - (h.^2).*(3-h);
end
```

The program iterates until the solution stops changing within some specified error tolerance. Figures 3&4 shows an example output for a density of 0.3. In this case the Newton Method is clearly superior.

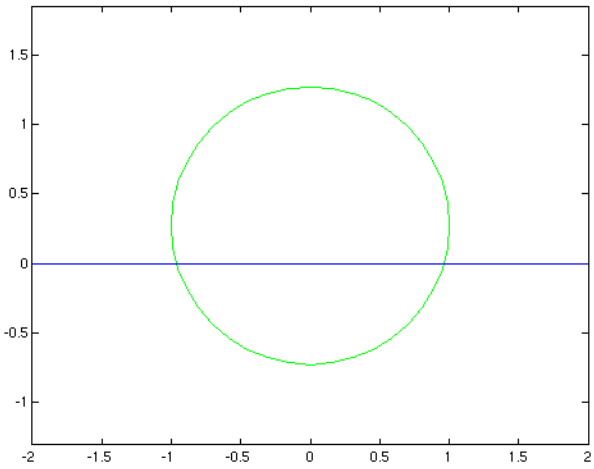

Figure 3: Output for the `floatsphere.m` program for ρ=0.3.

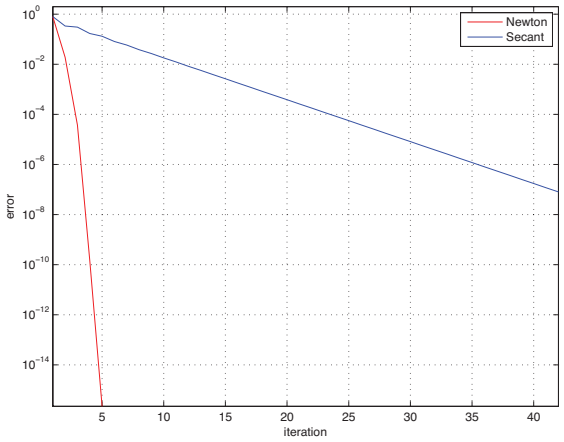

Figure 4: Error as a function of iteration for the Newton and Secant method for the `floatsphere.m` program for ρ=0.3.

Multivariable Newton's Method

Newton's method can be generalized to an N-variable problem. If the unknowns are now a vector $\vec{x}$ that we want to find 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

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

An example of this method is in the `newtn.m` program from the text. It is solving to find the stationary points for the Lorenz model from chapter 3b. In this case f is a column vector defined as

$$\mathbf{f} = \begin{bmatrix} \sigma(y-x) & rx-y-xz & xy-bz \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x & y & z \end{bmatrix}$$

where as $D_{ij} = \frac{\partial f_j}{\partial x_i}$ where the 'i' corresponds to rows and 'j' to columns of the matrix D, so we have

$$\mathbf{D} = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_2}{\partial x} & \frac{\partial f_3}{\partial x} \\ \frac{\partial f_1}{\partial y} & \frac{\partial f_2}{\partial y} & \frac{\partial f_3}{\partial y} \\ \frac{\partial f_1}{\partial z} & \frac{\partial f_2}{\partial z} & \frac{\partial f_3}{\partial z} \end{bmatrix} = \begin{bmatrix} -\sigma & r-z & y \\ \sigma & -1 & z \\ 0 & -x & -b \end{bmatrix}$$

The solution for **f = 0** is analytic as

$x = y = \sqrt{b(r-1)} \quad z = r-1$ for $\sigma \neq 0$ which you can readily verify from the program.