Optimization and Calculus

To begin, there is a close relationship between finding the roots to a function and optimizing a function. In the former case, we solve

$$g(x) = 0$$

for x. In the latter, we solve:

$$f'(x) = 0$$

for x. Therefore, discussions about optimization often turn out to be discussions about finding roots. In that spirit, we look at a very basic algorithm for finding roots- An algorithm called "The Bisection Method".

The Bisection Method

The motivation for this method comes from the Intermediate Value Theorem from Calculus- In this case, suppose that our function g is continuous on $[a, b]$, and further $g(a)$ and $g(b)$ have different signs (more compactly $g(a)g(b) < 0$), then there is a c in the interval such that $g(c) = 0$.

Now, given such an interval $[a, b]$, we can get a better approximation by breaking the interval in half. Here is the algorithm written in Matlab:

```
function xc=bisect(f,a,b,tol)
% Bisection Method, xc=bisect(f,a,b,tol)
%  Computes an approximation to f(x)=0 given that the
%  root is bracketed in [a,b] with f(a)f(b)<0.  Will run
%  until TOL is reached, and will output the solution xc.
%
% EXAMPLE:  f=inline('x^3+x-1');
%          xc=bisect(f,0,1,0.00005);

%Error check and initialization:
fa=feval(f,a);  fb=feval(f,b);
if sign(fa)*sign(fb)>=0
    error('Root may not be bracketed');
end

iter=0;
while (b-a)/2>tol
    iter=iter+1;
    c=(a+b)/2;
    fc=feval(f,c);
    if fc==0  %This means that (a+b)/2 is the root-
```

```
        break     %Break out of the while loop and
                  %continue execution
    end
    if sign(fc)*sign(fa)<0  %New interval is [a,c] (reset b)
        b=c; fb=fc;
    else
        a=c; fa=fc;   %New interval is [c,b] (reset a)
    end
end
fprintf('Finished after %d iterates\n',iter);
xc=(a+b)/2;
```

The nice thing about the bisection method is that it is easy to implement. It takes a lot of iterations to converge, however.

Newton's Method

Another method we have from Calculus is Newton's Method. Newton's Method is a lot faster, but it also requires the computation of the derivative. The formula is given below. If you're not sure where the formula comes from, we'll review it in class (you can also look it up online or in a calculus text).

Newton's Method

Newton's method is used to locate a root to a function g. It uses an initial guess and produces a sequence of values that (hopefully) converge to a root.

Given $g(x)$, and an initial guess of the root, x_0, we iterate the following:

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

In Matlab, this is straightforward as long as we have a separate function that will input x and output both the value of the function g and the value of g'. Here's an example of how that would work:

```
function out=NewtonMethod(F,x0,numits,tol)
%  Inputs to Newton's Method:
%     F- Name of a function that returns BOTH
%        values of g(x) and g'(x)
%     x0= Initial value of x.
%     numits=  Maximum number of iterations
%     tol   =  Tolerance (like 1e-6) measures
%              change between iterates of x

for k=1:numits
```

```
   [g,gprime]=feval(F,x0);
   if abs(gprime<eps)
      error('Derivative is zero');
   end
   xnew=x0-g/gprime;
   d=abs(xnew-x0);
   if d<tol
      out=xnew;
      break
   end
   x0=xnew
end
fprintf('Newton used %d iterations\n',k);
out=xnew;
```

To use either the bisection or Newton's method Matlab functions, we'll need to define the function (that we're finding the roots to). You can either use an anonymous function handle (defined in-line) or use an $m-$file- It is useful to go ahead and program in both f and the derivative so you can use either bisection or Newton's method. Here's an example finding a root to $\mathrm{e}^x = 3x$.

First, we'll write a function file:

```
function [y,dy]=testfunc01(x)
% Test function - Outputs both y and the derivative
y=exp(x)-3*x;
dy=exp(x)-3;
```

To use bisection starting with the interval $[0, 1]$ and getting a tolerance of 10^{-6}, we would type (in the command window):

```
out01=bisect('testfunc01',0,1,1e-6);
```

Now, Matlab should say that it took 19 iterations and the value in `out01` is about 0.6191. Similarly, we can run Newton's Method by typing the following in the command window:

```
out02=NewtonMethod('testfunc01',0,100,1e-6);
```

And you should see that Newton's Method only used 5 iterations.

Linearization

Before continuing with these algorithms, it will be helpful to review the Taylor series for a function of one variable, and see how it extends to functions of more than one variable.

Recall that the Taylor series for a function $f(x)$, based at a point $x = a$, is given by the following, where we assume that f is analytic:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Therefore, we can *approximate* f using a constant:

$$f(x) \approx f(a)$$

or using a linear approximation (which is the tangent line to f at a):

$$f(x) \approx f(a) + f'(a)(x-a)$$

or using a quadratic approximation:

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2$$

We can do something similar if f depends on more than one variable. For example, in Calculus III we look at functions like

$$z = f(x, y)$$

In this case, the linearization of f at $x = a$, $y = b$ is given by the tangent plane:

$$z = f(a, b) + f_x(a, b)(x-a) + f_y(a, b)(y-b)$$

Similarly, if $y = f(x_1, x_2, \ldots, x_n)$, then the tangent plane at $x_1 = a_1, \ldots, x_n = a_n$ is given by:

$$\begin{aligned} z = & f(a_1, a_2, \ldots, a_n) + f_{x_1}(a_1, \ldots, a_n)(x_1 - a_1) + f_{x_2}(a_1, \ldots, a_n)(x_2 - a_2) + \ldots \\ & f_{x_n}(a_1, \ldots, a_n)(x_n - a_n) \end{aligned}$$

If we want to go further with a second order (quadratic) approximation, it looks very similar. First, if $z = f(x, y)$ at (a, b), the quadratic approximation looks like this:

$$\begin{aligned} z = f(a, b) + f_x(a, b)(x-a) + f_y(a, b)(y-b) + \frac{1}{2}[f_{xx}(a, b)(x-a)^2 + \\ 2f_{xy}(a, b)(x-a)(y-b) + f_{yy}(a, b)(y-b)^2] \end{aligned}$$

where we assume that $f_{xy}(a, b) = f_{yx}(a, b)$. For functions of n variables, the notation gets a little awkward. We'll define a new structure that will make the notation work a little better.

Gradient and Hessian

Let $y = f(x_1, \ldots, x_n)$ be a real valued function of n variables. To make the notation a bit easier to read, we'll denote the partial derivatives using subscript notation, where

$$\frac{\partial f}{\partial x_i} \doteq f_i \qquad \frac{\partial^2 f}{\partial x_i \partial x_j} \doteq f_{ji}$$

(Does the order of the differentiation matter? In the last equation, could I have written f_{ij}?)

We'll recall from Calculus III that the gradient of f is usually defined as a row vector of first partial derivatives:

$$\nabla f = [f_1, f_2, \ldots, f_n]$$

The $n \times n$ matrix of second partial derivatives is called the **Hessian matrix**, where the (i, j) element is f_{ij}, or

$$Hf = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & & & \vdots \\ f_{n1} & f_{n2} & \cdots & f_{nn} \end{bmatrix}$$

Using this notation, the **linear approximation** to $f(x_1, \ldots, x_n) = f(\vec{x})$ at the point $\vec{x} = \vec{a}$ is:

$$f(\vec{a}) + \nabla f(\vec{a})(\vec{x} - \vec{a})$$

(That last term is a dot product, or a row vector times a column vector) The **quadratic approximation** to f is:

$$f(\vec{a}) + \nabla f(\vec{a})(\vec{x} - \vec{a}) + \frac{1}{2}(\vec{x} - \vec{a})^T Hf(\vec{a})(\vec{x} - \vec{a})$$

As another example, suppose we want the quadratic approximation to the function

$$w = x \sin(y) + y^2 z + xyz + z$$

Find the quadratic approximation to w at the point $(1, 0, 2)$.

SOLUTION: We need to evaluate f, all the first partials, and all the second partials at the given point:

$$f(1, 0, 2) = 1 \cdot 0 + 0 \cdot 2 + 0 + 2 = 2$$

Now, using our new notation for partial derivatives:

$$f_1 = \sin(y) + yz \quad \Rightarrow \quad f_1(1, 0, 2) = 0$$

$$f_2 = x \cos(y) + 2yz + xz \quad \Rightarrow \quad f_2(1, 0, 2) = 3$$

$$f_3 = y^2 + xy + 1 \quad \Rightarrow \quad f_3(1, 0, 2) = 1$$

Therefore, $\nabla f(1,0,2) = [0,3,1]$. Now computing the Hessian, we get:

$$\left[\begin{array}{ccc} 0 & \cos(y)+z & y \\ \cos(y)+z & -x\sin(y)+2z & 2y+x \\ y & 2y+x & 0 \end{array}\right]\Bigg|_{x=1,y=0,z=2} = \left[\begin{array}{ccc} 0 & 3 & 0 \\ 3 & 4 & 1 \\ 0 & 1 & 0 \end{array}\right]$$

Now we can put these into the formula for the quadratic approximation:

$$2 + [0,3,1]\left[\begin{array}{c} x-1 \\ y \\ z-2 \end{array}\right] + \frac{1}{2}[x-1, y, z-2]\left[\begin{array}{ccc} 0 & 3 & 0 \\ 3 & 4 & 1 \\ 0 & 1 & 0 \end{array}\right]\left[\begin{array}{c} x-1 \\ y \\ z-2 \end{array}\right]$$

And expanded, we could write this as:

$$2 + 3y + (z-2) + \frac{1}{2}\left[6(x-1)y + 4y^2 + 2y(z-2)\right] = \\ -2y + z + 3xy + 2y^2 + yz$$

This is the quadratic approximation to $x\sin(y) + y^2z + xyz + z$ at the point $(1,0,2)$.

Linearization, Continued

Suppose we have a general function $\vec{G}$ that inputs n variables and outputs m variables. In that case, we might write $\vec{G}$ as:

$$\vec{G}(\vec{x}) = \left[\begin{array}{c} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ g_m(x_1, x_2, \ldots, x_n) \end{array}\right]$$

For example, here's a function that inputs two variables and outputs 3 nonlinear functions:

$$\vec{G}(x,y) = \left[\begin{array}{c} x^2 + xy + y^2 - 3x + 5 \\ \sin(x) + \cos(y) \\ 1 + 3x - 5y + 2xy \end{array}\right] \tag{1}$$

We need a new way of getting the derivative- In this case, it is called **the Jacobian matrix**. If $\vec{G}$ maps $\mathbb{R}^n$ to $\mathbb{R}^m$, then the Jacobian matrix is $m \times n$, where each row is the gradient of the corresponding function:

$$\vec{G}(\vec{x}) = \left[\begin{array}{c} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ g_m(x_1, x_2, \ldots, x_n) \end{array}\right] \quad \Rightarrow \quad JG = \left[\begin{array}{c} \nabla g_1(x_1, x_2, \ldots, x_n) \\ \nabla g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ \nabla g_m(x_1, x_2, \ldots, x_n) \end{array}\right]$$

Or, written out, the (i, j) element of the Jacobian matrix for $\vec{G}$ is:

$$(JG)_{ij} = \frac{\partial g_i}{\partial x_j}$$

Continuing with our previous example (Equation 1), we'll construct the Jacobian matrix:

$$\vec{G}(x, y) = \begin{bmatrix} x^2 + xy + y^2 - 3x + 5 \\ \sin(x) + \cos(y) \\ 1 + 3x - 5y + 2xy \end{bmatrix} \quad \Rightarrow \quad JG = \begin{bmatrix} 2x + y - 3 & x + 2y \\ \cos(x) & -\sin(y) \\ 3 + 2y & -5 + 2x \end{bmatrix}$$

As a second example, suppose that we have a function f that maps $\mathbb{R}^n$ to $\mathbb{R}$. Then we could let

$$\vec{G}(\vec{x}) = \nabla f(\vec{x})$$

Note that we have to think of the gradient as a column vector instead of a row, and the i^{th} row is:

$$f_{x_i}(x_1, x_2, \ldots, x_n)$$

so that $\vec{G}$ maps $\mathbb{R}^n$ to $\mathbb{R}^n$. Furthermore, in that case, the **Jacobian of** $\vec{G}$ is the **Hessian of** f:

$$(JG)_{ij} = \frac{\partial f_{x_i}}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_i}\right) = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

(We're using Clairaut's Theorem to simplify our expression).

Now we get to the **linearization** of the function $\vec{G}$ at a point $\vec{x} = \vec{a}$:

$$\vec{G}(\vec{a}) + JG(\vec{a})(\vec{x} - \vec{a})$$

To illustrate this notation, let's linearize Equation 1 at the origin $(0, 0)$.

$$\vec{G}(0, 0) = \begin{bmatrix} 5 \\ 1 \\ 1 \end{bmatrix} \qquad JG(0, 0) = \begin{bmatrix} -3 & 0 \\ 1 & 0 \\ 3 & -5 \end{bmatrix}$$

Therefore, the linearization at the origin is:

$$\begin{bmatrix} 5 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -3 & 0 \\ 1 & 0 \\ 3 & -5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 - 3x \\ 1 + x \\ 1 + 3x - 5y \end{bmatrix}$$

You might notice that each row is the linearization of the corresponding function from $\vec{G}$- For example, $1 + x$ is the linearization of $\sin(x) + \cos(y)$ at the point $(0, 0)$.

Now we can get back to Newton's Method.

Multivariate Newton's Method

In the general case, we are solving for the critical points of some function f,

$$\nabla f(\vec{x}) = \vec{0}$$

That is, if f depends on n variables, then this is a system of n equations (in n unknowns).

As we did in the previous section, we can think of the gradient itself as a function:

$$\vec{G}(\vec{x}) = \nabla f(\vec{x})$$

and think about how Newton's Method will apply in this case.

Recall that in Newton's Method in one dimension, we begin with a guess x_0. We then solve for where the *linearization* of f crosses the $x-$axis:

$$f(x_0) + f'(x_0)(x - x_0) = 0$$

From which we get the formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Now we do the same thing with the vector-valued function $\vec{G}$, where $\vec{G} : \mathbb{R}^n \to \mathbb{R}^n$. Start with an initial guess, $\vec{x}_0$, then we linearize $\vec{G}$ and solve the following for $\vec{x}$:

$$\vec{G}(\vec{x}_0) + JG(\vec{x}_0)(\vec{x} - \vec{x}_0) = 0$$

Remember that JG, the Jacobian of $\vec{G}$, is now an $n \times n$ matrix, so this is a system of n equations in n unknowns. Using the recursive notation, and solving for $\vec{x}$, we now get the formula for **multidimensional Newton's Method**:

$$\vec{x}_{i+1} = \vec{x}_i - JG^{-1}(\vec{x}_i)G(\vec{x}_i)$$

We should notice the beautiful way that method generalizes to multiple dimensions- The reciprocal $1/f'(x_i)$ becomes the matrix inverse: $JG^{-1}(\vec{x}_0)$.

Nonlinear Optimization with Newton

Now we go back to our original question: We want to optimize a function whose domain is multidimensional:

$$\min_{\mathbf{x}} \ f(\mathbf{x}) \qquad \text{or} \qquad \max_{\mathbf{x}} \ f(\mathbf{x})$$

where it is assumed that $f : \mathbb{R}^n \to \mathbb{R}$. To use Newton's Method, we let $\vec{G} = \nabla f$, and look for the zeros of the gradient. Here's a summary:

Newton's Method
For solving $\nabla f(\mathbf{x}) = 0$

Given an initial $\mathbf{x}_0$, compute a sequence of better approximations to the solution:

$$\vec{x}_{i+1} = \vec{x}_i - Hf^{-1}(\vec{x}_i)\nabla f(\vec{x}_i)$$

where Hf is the Hessian matrix ($n \times n$ matrix of the second partial derivatives of f).

Example: Minimize $f(x, y) = \frac{1}{4}x^4 - \frac{1}{2}x^2 + \frac{1}{2}y^2$.

SOLUTION: First, we can use Calculus to check the computer output. The gradient is:

$$\nabla f = [x^3 - x, \quad y]$$

so that the critical points are $(-1, 0), (1, 0)$ and $(0, 0)$.

We may recall the "second derivatives test" from Calculus III- It actually uses the Hessian:

Let $D = f_{xx}(a, b)f_{yy}(a, b) - f_{xy}^2(a, b)$. If

- If $D > 0$ and $f_{xx}(a, b) > 0$, then $f(a, b)$ is a local min.
- If $D > 0$ and $f_{xx}(a, b) < 0$, then $f(a, b)$ is a local max.
- If $D < 0$, then $f(a, b)$ is a saddle point.

The Hessian of f:

$$Hf = \begin{bmatrix} f_{xx}(x, y) & f_{xy}(x, y) \\ f_{yx}(x, y) & f_{yy}(x, y) \end{bmatrix} = \begin{bmatrix} 3x^2 - 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Therefore, the second derivatives test is a test using the determinant of the Hessian (more generally, we would need the eigenvalues of the Hessian).

At $(\pm 1, 0)$, the determinant is positive and $f_{xx}(\pm 1, 0)$ is positive. Therefore, $f(\pm 1, 0) = 1/4$ is the local minimum. We have a saddle point at the origin.

Newton's Method, Matlab

Here is a function that will implement the multidimensional Newton's method in Matlab. When you write the file for your function, you will need to include the gradient and the Hessian.

In the previous example, here's the test function file:

```
function [y,dy,hy]=newttest(x)
%  A test function for Newton's Method for Optimization
%  The input is the VECTOR x (with elements x,y below)
%
%     y = (1/4)x^4-(1/2)x^2+(1/2)y^2
%    dy = Gradient = [x^3-x; y]  (The gradient will output as a COLUMN)
%    hy = Hessian  = [3x^2-1, 0;0,1]
```

```
y=(1/4)*x(1)^4-(1/2)*x(1)^(1/2)*x(2)^2;
dy=[x(1)^3-x(1); x(2)];
hy=[3*x(1)^2-1, 0;0,1];
```

Here's the main function we'll use:

```
function out=MultiNewton(F,x0,numits,tol)
%  Newton's Method defined for optimizing multi-dimensional F.
%
%     F- Name of a function that returns
%        F, the gradient AND the Hessian
%
%     x0= Initial value of x.
%
%     numits=  Maximum number of iterations
%
%     tol   =  Tolerance (like 1e-6) measures
%              change between iterates of x

for k=1:numits

   [g,gradg,hessg]=feval(F,x0);

   %
   % This is an error check to see if the matrix is
   %  non-invertible.  You don't need to know this,
   %  but we're using the "condition number"
   %
   if cond(hessg)>1000000
       error('The Hessian Matrix is not invertible');
   end

   xnew=x0-inv(hessg)*gradg;

   d=norm(xnew-x0);
   if d<tol
      out=xnew;
      break
   end

   x0=xnew;
end
fprintf('Newton used %d iterations\n',k);
```

```
out=xnew;
```

Now, to run our code, we would type something like this into the command window:

```
>> out=MultiNewton('newttest',[0.65;0.5],100,1e-6);
Newton used 8 iterations
out =
     1
     0
```

Steepest Descent

The method of steepest descent is an algorithm that will search for local extrema by taking advantage of the fact about gradients:

> The gradient of a function at a given point will point in the direction of fastest increase (of the function).

Therefore, if we started at a point (or vector) $\vec{a}$, if we want to find a local maximum, we should move in the direction of the gradient. To locate a local minimum, we should move in the direction negative to the gradient.

Exercise: Prove the statement given.

Therefore, the method of steepest descent proceeds as follows: Given a starting point $\vec{a}$ and a scalar α (which is referred to as the step size), we iterate the following:

$$\vec{x}_{i+1} = \vec{x}_i - \alpha \nabla f(\vec{x}_i)$$

The method is fairly quick, but can have bad convergence properties- Mainly because we have that step size. There is a way to get "best" step size α. In the following, we'll continue with finding the local **minimum**.

Once we choose a direction of travel, the idea will be to follow that line until we find the minimum of the function (along the line). That is, we have a small optimization problem:

$$\arg\min_t f(\vec{a} - t\nabla f(\vec{a}))$$

This is called a *line search*, and can be time consuming. Before continuing, let's look at this more closely. Define a new function $\phi(t)$ as the output along the line:

$$\phi(t) = f(\vec{a} - t\vec{u}) = f\begin{pmatrix} a_1 - tu_1 \\ a_2 - tu_2 \\ \vdots \\ a_n - tu_n \end{pmatrix}$$

where $\vec{u}$ is a vector and f depends on $x_1, \ldots, x_n$.

Then optimizing this means to find where the derivative of ϕ is zero. Let's compute the derivative using the chain rule:

$$\phi'(t) = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \cdots + \frac{\partial f}{\partial x_n}\frac{\partial x_n}{\partial t} = \frac{\partial f}{\partial x_1}u_1 + \cdots + \frac{\partial f}{\partial x_n}u_n = -\nabla f(\vec{a} - t\vec{u}) \cdot \vec{u}$$

(Note the minus sign from differentiating $-t\vec{u}$). Thus, setting this to zero and solving should give us an optimal step size.

Example: Apply the method of steepest descent to the following, and try to determine the optimal step size:

$$f(x, y) = 4x^2 - 4xy + 2y^2$$

with initial point $(2, 3)$. Try to determine the best step size at each iteration.

SOLUTION: Compute the gradient and evaluate it at $(2, 3)$:

$$\nabla f = [8x - 4y, -4x + 4y] \quad \Rightarrow \quad \nabla f(2, 3) = [4, 4]$$

For some fixed h, our next point will be:

$$\vec{x}_1 = \vec{x}_0 - h\nabla f(\vec{x}_0) = \begin{bmatrix} 2 \\ 3 \end{bmatrix} - h \begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 - 4h \\ 3 - 4h \end{bmatrix}$$

To find the optimal path, we define $\phi(t)$ to be the function restricted to the line:

$$\phi(t) = f(\vec{x}_0 - t\nabla f(\vec{x}_0))$$

We differentiate to find the minimum:

$$\phi'(t) = -\nabla f(\vec{x}_0 - h\nabla f(\vec{x}_0)) \cdot \nabla f(\vec{x}_0) = -[4 - 16t,\ 4] \begin{bmatrix} 4 \\ 4 \end{bmatrix} = -32 + 64t$$

Therefore, the derivative is zero for $t = 1/2$, and the second derivative is positive, so this is indeed where the minimum occurs.

Now, using that step size:

$$\vec{x}_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix} - \frac{1}{2} \begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Exercises

1. Use the bisection method (in Matlab) to find the root correct to 6 decimal places: $3x^3 + x^2 = x + 5$.

2. Suppose we have a function f so that $f(0, 1) = 3$ and $\nabla f((0, 1)) = [1, -2]$. Use the linearization of f to estimate $f(1/2, 3/2)$.

3. Let $f(x, y) = x^2y + 3y$. Compute the linearization of f at $x = 1, y = 1$.

4. Let $f(x, y) = xy+y^2$. At the point $(2, 1)$, in which direction is f increasing the fastest? How fast is it changing?

5. Use the second derivatives test to classify the critical points of the following functions:

   (a) $f(x, y) = x^2 + xy + y^2 + y$

   (b) $f(x, y) = xy(1 - x - y)$

6. Use Matlab and our Newton's Method program to find two critical points of f correct to three decimal places.

$$f(x, y) = x^4 + y^4 - 4x^2y + 2y$$

   Hint: There are three of them in $-1 \leq x \leq 1, -1 \leq y \leq 1$, so try several starting points.

7. Perform the computations involved in the second step of the method of steepest descent example (with $f(x, y) = 4x^2 - 4xy + 2y^2$).

8. We said that, in order to optimize the step size, we set the derivative of ϕ to zero:

$$\phi'(t) = \nabla f(\vec{a} - t\vec{u}) \cdot \vec{u} = 0$$

   Show that this implies the following theorem about the method of steepest descent:

$$\nabla f(\vec{x}_{i+1}) \cdot \nabla f(\vec{x}_i) = 0$$

   or, the gradient vectors from one step to the next are orthogonal to each other.