MATH2070: LAB 5: Multidimensional Newton's Method

1 Introduction

Last time we discussed Newton's method for nonlinear equations in one real or complex variable. In this lab, we will extend the discussion to two or more dimensions. One of the examples will include a common application of Newton's method, *viz.*, nonlinear least squares fitting.

This lab will take three sessions. If you print it, you might find the pdf version more convenient.

2 Modifications to `newton.m` for vector functions

Suppose $\mathbf{x}$ is a vector in $\mathbb{R}^N$, $\mathbf{x} = (x_1, x_2, \ldots, x_N)$, and $\mathbf{f}(\mathbf{x})$ a differentiable vector function,

$$\mathbf{f} = (f_m(x_1, x_2, \ldots, x_N)), \text{ for } m = 1, 2, \ldots, N,$$

with Jacobian matrix J,

$$J_{mn} = \frac{\partial f_m}{\partial x_n}.$$

Recall that Newton's method can be written

$$\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} = -J(\mathbf{x}^{(k)})^{-1}\mathbf{f}(\mathbf{x}^{(k)}). \tag{1}$$

(The superscript indicates iteration number. A superscript is used to distinguish iteration number from vector index.)

As an implementation note, the inverse of J should not be computed. Instead, the system

$$J(\mathbf{x}^{(k)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = -\mathbf{f}(\mathbf{x}^{(k)}).$$

should be solved. As you will see later in this course, this will save about a factor of three in computing time over computing the inverse. Matlab provides a special, division-like symbol for this solution operation: the backslash (\) operator. If you wish to solve the system

$$J\boldsymbol{\Delta}\mathbf{x} = -\mathbf{f}$$

then the solution can be written as

$$\mathbf{\Delta x} = -J\backslash\mathbf{f}.$$

Note that the divisor is written so that it appears "underneath" the backslash. Mathematically speaking, this expression is equivalent to

$$\mathbf{\Delta x} = -J^{-1}\mathbf{f}$$

but using \ is about three times faster.

You might wonder why Matlab needs a new symbol for division when we have a perfectly good division symbol already. The reason is that matrix multiplication is not commutative, so order is important. Multiplying a matrix times a (column) vector requires the vector to be on the right. If you wanted to "divide" the vector $\mathbf{f}$ by the matrix J using the ordinary division symbol, you would have to write $\mathbf{f}/J$, But this would seem to imply that $\mathbf{f}$ is a *row* vector because it is to the left of the matrix. If $\mathbf{f}$ is a column vector, you would need to write $\mathbf{f}^T/J$, but this expression is awkward, and does not really mean what you want because the result would be a row vector. The Matlab method:

$$J\backslash\mathbf{f}$$

is better.

Exercise 1:

(a) Start from your version of `newton.m` from Lab 4, or download my version. Change its name to `vnewton.m` and change its signature to

```
function [x,numIts]=vnewton(func,x,maxIts)
% comments
```

and modify it so that it: (a) Uses the Matlab \ operator instead of scalar division for `increment`; (b) uses the `norm` of the increment and `norm` of the old increment instead of `abs` to determine `r1` and the error estimate; (c) has no `disp` statements; and, (d) has appropriately modified comments. (Leaving the `disp` statements in will cause syntax errors when vector arguments are present, so be sure to eliminate them.)

(b) Test your modifications by comparing the value of the root and the number of iterations required for the complex scalar (not vector) case from last time ($f_8(z) = z^2 + 9$). This shows that `vnewton` will still work for scalars. Your results should agree with those from `newton`. Fill in the following table (same one as last lab, except the final line). Recall that the column `norm(error)` refers to the *true* error, the absolute value of the difference between the computed solution and the true solution.

Initial guess	numIts	norm(error)	numIts(newton)
1+i	-------	---------	--------------
1-i	-------	---------	--------------
10+5i	-------	---------	--------------
10+1.e-25i	-------	---------	--------------

3 A complex function revisited

It is possible to rewrite a complex function of a complex variable as a *vector* function of a *vector* variable. This is done by equating the real part of a complex number with the first component of a vector and the imaginary part of a complex number with the second component of a vector.

Reminder: In Matlab, you denote subscripts using parentheses. For example, the 13^{th} element of a vector f would be denoted f_{13} mathematically and `f(13)` in Matlab.

Consider the function $f_8(z) = z^2 + 9$. Write $z = x_1 + x_2 i$ and $f(z) = f_1 + f_2 i$. Plugging these variables in yields

$$f_1 + f_2 i = (x_1^2 - x_2^2 + 9) + (2x_1 x_2)i.$$

This can be written in an equivalent matrix form as

$$\begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_1^2 - x_2^2 + 9 \\ 2x_1 x_2 \end{bmatrix} \tag{2}$$

Exercise 2:

(a) Write a function m-file `f8v.m` to compute the vector function described above in Equation (2) and its Jacobian. It should be in the form needed by `vnewton` and with the signature

```
function [f,J]=f8v(x)
% comments
```

where `f` is the two-dimensional column vector from Equation (2) and `J` is its Jacobian matrix. **Hint:** Compute, by hand, the formulas for `df1dx2` ($= \frac{\partial f_1}{\partial x_2}$), `df1dx1`, `df2dx1`, and `df2dx2`. Then set

```
J=[df1dx1 df1dx2
   df2dx1 df2dx2];
```

(b) Test your `vnewton.m` using `f8v.m` starting from the column vector `[1;1]` and comparing with the results of `newton.m` using `f8.m` and starting from `1+1i`. Both solution and number of iterations should agree exactly. If they do not, use the debugger to compare results after 1, 2, *etc.* iterations.

(c) Fill in the table below. Note that the norm of the error here should be the same as the absolute value of the error for f_8 in the previous exercise, and the correct solutions are

$$\begin{bmatrix} 0 \\ \pm 3 \end{bmatrix}.$$

Initial guess	numIts	norm(error)
[1;1]	_____	_____
[1;-1]	_____	_____
[10;5]	_____	_____
[10;1.e-25]	_____	_____

At this point, you should be confident that `vnewton` is correctly programmed and yields the same results as `newton` from the previous lab. In the following exercise, you will apply `vnewton` to a simple nonlinear problem.

Exercise 3: Use Newton's method to find the two intersection points of the parabola $x_2 = x_1^2 - x_1$ and the ellipse $x_1^2/16 + x_2^2 = 1$.

(a) Plot both the parabola and the ellipse on the same plot, showing the intersection points. Be sure to show the whole ellipse so you can be sure there are exactly two intersection points. Please include this plot when you send me your work.

(b) Write a Matlab function m-file `f3v.m` to compute the vector function that is satisfied at the intersection points. This function should have the signature

```
function [f,J]=f3v(x)
% [f,J]=f3v(x)
% f and x are both 2-dimensional column vectors
% J is a 2 X 2 matrix
```

```
% more comments

% your name and the date
```

Hint: The vector `f=[f1;f2]`, where `f1` and `f2` are each zero at the intersection points.

(c) Starting from the initial column vector guess `[2;1]`, what is the intersection point that `vnewton` finds, and how many iterations does it take?

(d) Find the other intersection point by choosing a different initial guess. What initial guess did you use, what is the intersection point, and how many iterations did it take?

4 Slow to get started

In Exercise 2, you saw that the guess `[10;1.e-25]` resulted in a relatively large number of iterations. While not a failure, a large number of iterations is unusual. In the next exercise, you will investigate this phenomenon using Matlab as an investigative tool.

Exercise 4: In this exercise, we will look more carefully at the iterates in the poorly-converging case from Exercise 2. Because we are interested in the sequence of iterates, we will generate a special version of `vnewton.m` that returns the full sequence of iterates. It will return the iterates as a matrix whose columns are the iterates, with as many columns as the number of iterates. (One would not normally return so much data from a function call, but we have a special need for this exercise.)

(a) Make a copy of `vnewton.m` and call it `vnewton1.m`. Change its signature line to the following

```
function [x, numIts, iterates]=vnewton1(func,x,maxIts)
% comments
```

(b) Just *before* the loop, add the following statement

```
iterates=x;
```

to initialize the variable `iterates`.

(c) Add the following statement *just after* the new value of `x` has been computed.

```
iterates=[iterates,x];
```

Since `iterates` is a matrix and `x` is a column vector, this Matlab statement causes one column to be appended to the matrix `iterates` for each iteration.

(d) Replace the `error` function call at the end of the function with a `disp` function call. The reason for this change is that you will be using `vnewton1` later in this lab to see how Newton's method can fail.

(e) Test your modified function `vnewton1` using the same `f8v.m` from above, starting from the vector `[2;1]`. You should get the same results as before for solution and number of iterations. Check that the size of the matrix is 2×(number of iterations+1), *i.e.*, it has 2 rows and (`numIts`+1) columns.

The study of the poorly-converging case continues with the following exercise. The objective is to try to understand what is happening. The point of the exercise is two-fold: on the one hand to learn something about Newton iterations, and on other hand to learn how one might use Matlab as an investigative tool.

Exercise 5:

(a) Apply `vnewton1.m` to the function `f8v.m` starting from the column vector `[10;1.e-25]`. This is the poorly-converging case.

(b) Plot the iterates on the plane using the command

```
plot(iterates(1,:),iterates(2,:),'*-')
```

It is very hard to interpret this plot, but it is a place to start. You do not have to send me a copy of your plot. Note that most of the iterates lie along the x-axis, with some quite large.

(c) Use the zoom feature of plots in Matlab (the magnifying glass icon with a + sign) to look at region with horizontal extent around [-20,20]. It is clear that most of the iterates appear to be on the x-axis. Please send me a copy of this plot.

(d) Look at the formula you used for the Jacobian in `f8v.m`. Explain why, if the initial guess starts with $x_2 = 0$ exactly, then all subsequent iterates also have $x_2 = 0$. **Remark:** You have used Matlab to help formulate a hypothesis that can be proved.

(e) You should be able to see what is happening. I chose an initial guess with $x_2 \approx 0$, so subsequent iterates stayed near the x-axis. Since the root is at $x_2 = 3$, it takes many iterates before the x_2 component of the iterate can "rise to the occasion," and enter the region of quadratic convergence.

(f) Use a semilog plot to see how the iterates grow in the vertical direction:

```
semilogy( abs( iterates(2,:) ) )
```

The plot shows the x_2 component seems to grow exponentially (linearly on the semilog plot) with seemingly random jumps superimposed. Please send me a copy of this plot.

You now have a rough idea of how this problem is behaving. It converges slowly because the initial guess is not close enough to the root for quadratic convergence to be exhibited. The x_2 component grows exponentially, however, and eventually the iterates become close enough to the root to exhibit quadratic convergence. It is possible to prove these observations, although the proof is beyond the scope of this lab.

5 Nonlinear least squares

A common application of Newton's method for vector functions is to nonlinear curve fitting by least squares. This application falls into the general topic of "optimization" wherein the extremum of some function $F : \mathbb{R}^n \to \mathbb{R}$ is sought. If F is differentiable, then its extrema are given by the solution of the system of equations $\partial F/\partial x_k = 0$ for $k = 1, 2, \ldots, n$, and the solution can be found using Newton's method.

The differential equation describing the motion of a weight attached to a damped spring without forcing is

$$m\frac{d^2v}{dt^2} + c\frac{dv}{dt} + kv = 0,$$

where v is the displacement of the weight from equilibrium, m is the mass of the weight, c is a constant related to the damping of the spring, and k is the spring stiffness constant. The physics of the situation indicate that m, c and k should be positive. Solutions to this differential equation are of the form

$$v(t) = e^{-x_1 t}(x_3 \sin x_2 t + x_4 \cos x_2 t),$$

where $x_1 = c/(2m)$, $x_2 = \sqrt{k - c^2/(4m^2)}$, and x_3 and x_4 are values depending on the position and velocity of the weight at $t = 0$. One common practical problem (called the "parameter identification" problem) is to estimate the values $x_1 \ldots x_4$ by observing the motion of the spring at many instants of time.

After the observations have progressed for some time, you have a large number of pairs of values (t_n, v_n) for $n = 1, \ldots, N$. The question to be answered is, "What values of x_k for $k = 1, 2, 3, 4$ would best reproduce the observations?" In other words, find the values of $\mathbf{x} = [x_1, x_2, x_3, x_4]^T$ that minimize the norm of the differences between the formula and observations. Define

$$F(\mathbf{x}) = \sum_{n=1}^{N} (v_n - e^{-x_1 t_n}(x_3 \sin x_2 t_n + x_4 \cos x_2 t_n))^2 \tag{3}$$

and we seek the minimum of F.

Note: The particular problem as stated can be reformulated as a linear problem, resulting in reduced numerical difficulty. However, it is quite common to solve the problem in this form and in more difficult cases it is not possible to reduce the problem to a linear one.

In order to solve this problem, it is best to note that when the minimum is achieved, the gradient $\mathbf{f} = \nabla F$ must be zero. The components f_k of the gradient $\mathbf{f}$ can be written as

$$
\begin{aligned}
f_1 &= \frac{\partial F}{\partial x_1} = 2\sum_{k=1}^{K}(v_k - e^{-x_1 t_k}(x_3 \sin x_2 t_k + x_4 \cos x_2 t_k))t_k e^{-x_1 t_k}(x_3 \sin x_2 t_k + x_4 \cos x_2 t_k) \\
f_2 &= \frac{\partial F}{\partial x_2} = -2\sum_{k=1}^{K}(v_k - e^{-x_1 t_k}(x_3 \sin x_2 t_k + x_4 \cos x_2 t_k))e^{-x_1 t_k}(x_3 \cos x_2 t_k t_k - x_4 \sin x_2 t_k t_k) \qquad (4) \\
f_3 &= \frac{\partial F}{\partial x_3} = -2\sum_{k=1}^{K}(v_k - e^{-x_1 t_k}(x_3 \sin x_2 t_k + x_4 \cos x_2 t_k))e^{-x_1 t_k} \sin(x_2 t_k) \\
f_4 &= \frac{\partial F}{\partial x_4} = -2\sum_{k=1}^{K}(v_k - e^{-x_1 t_k}(x_3 \sin x_2 t_k + x_4 \cos x_2 t_k))e^{-x_1 t_k} \cos(x_2 t_k)
\end{aligned}
$$

(One rarely does this kind of calculation by hand any more. The Matlab symbolic toolbox, or Maple or Mathematica can greatly reduce the manipulative chore.)

To apply Newton's method to $\mathbf{f}$ as defined in (4), the sixteen components of the Jacobian matrix are also needed. These are obtained from f_i above by differentiating with respect to x_j for $j = 1, 2, 3, 4$.

Remark: The function F is a real-valued function of a vector. Its gradient, $\mathbf{f} = \nabla F$, is a vector-valued function. The gradient of $\mathbf{f}$ is a matrix-valued function. The gradient of $\mathbf{f}$, called the "Jacobian" matrix in the above discussion, is the second derivative of F, and it is sometimes called the "Hession" matrix. In that case, the term "Jacobian" is reserved for the gradient. This latter usage is particularly common in the context of optimization.

In the following exercise, you will see that Newton's method applied to this system can require the convergence neighborhood to be quite small.

Exercise 6:

(a) Download my code for the least squares objective function F, its gradient $\mathbf{f}$, and the Jacobian matrix J. The file is called `objective.m` because a function to be minimized is often called an "objective function." (Our objective is to minimize the objective function.)

(b) Use the command `help objective` to see how to use it.

(c) Compute `f` at the solution `x=[0.15;2.0;1.0;3]` to be sure that the function is zero there.

(d) Compute the determinant of `J` at the solution `x=[0.15;2.0;1.0;3]` to see that it is nonsingular.

(e) Since this particular problem seeks the minimum of a quadratic functional, the matrix `J` must be positive definite and symmetric. Check that `J` is symmetric and then use the Matlab `eig` function to find the four eigenvalues of `J` to see they are each positive.

Exercise 7: The point of this exercise is to see how sensitive Newton's method can be when the initial guess is changed. Fill in the following table with either the number of iterations required or the word "failed," using `vnewton.m` and the indicated initial guess. Note that the first line is the correct solution. *For this exercise, restrict the number of iterations to be no more than 100.*

Initial guess	Number of iterations
[0.15; 2.0; 1.0; 3]	

```
[0.15;  2.0; 0.9; 3]       ______________
[0.15;  2.0; 0.0; 3]       ______________
[0.15;  2.0;-0.1; 3]       ______________
[0.15;  2.0;-0.3; 3]       ______________
[0.15;  2.0;-0.5; 3]       ______________

[0.15;  2.0; 1.0; 4]       ______________
[0.15;  2.0; 1.0; 5]       ______________
[0.15;  2.0; 1.0; 6]       ______________
[0.15;  2.0; 1.0; 7]       ______________

[0.15; 1.99; 1.0; 3]       ______________
[0.15; 1.97; 1.0; 3]       ______________
[0.15; 1.95; 1.0; 3]       ______________
[0.15; 1.93; 1.0; 3]       ______________
[0.15; 1.91; 1.0; 3]       ______________

[0.17;  2.0; 1.0; 3]       ______________
[0.19;  2.0; 1.0; 3]       ______________
[0.20;  2.0; 1.0; 3]       ______________
[0.21;  2.0; 1.0; 3]       ______________
```

You can see from the previous exercise that Newton can require a precise guess before it will converge. Sometimes some iterate is not far from the ball of convergence, but the Newton step is so large that the next iterate is ridiculous. In cases where the Newton step is too large, reducing the size of the step might make it possible to get inside the ball of convergence, even with initial guesses far from the exact solution. This is the strategy examined in the following section.

6 Softening (damping)

The exercise in the previous section suggests that Newton gets in trouble when its increment is too large. One way to mitigate this problem is to "soften" or "dampen" the iteration by putting a fractional factor on the iterate.

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(k)} - \alpha J(\mathbf{x}^{(k)})^{-1}\mathbf{f}(\mathbf{x}^{(k)}) \tag{5}$$

where α is a number smaller than one. It should be clear from the convergence proofs you have seen for Newton's method that introducing the softening factor α destroys the quadratic convergence of the method. This raises the question of stopping. In the current version of `vnewton.m`, you stop when `norm(increment)` gets small enough, but if `increment` has been multiplied by `alpha`, then convergence could happen immediately if `alpha` is very small. It is important to make sure `norm(increment)` is not multiplied by `alpha` before the test is done.

Try softening in the following exercise.

Exercise 8:

(a) Starting from your `vnewton.m` file, copy it to a new file named `snewton0.m` (for "softened Newton"), change its signature to

```
function [x,numIts]=snewton0(f,x,maxIts)
```

and change it to conform with Equation (5) with the fixed value $\alpha = 1/2$. Don't forget to change the comments and the convergence criterion. **Matlab warning:** The backslash operator does

not observe the "operator precedence" you might expect, so you need parentheses. For example, `3*2\4=0.6667`, but `3*(2\4)=6`!

(b) Returning to Exercise 7, to the nearest 0.01, how large can the first component of the initial guess get before the iteration diverges? (Leave the other three values at their correct values.)

Softening by a constant factor can improve the initial behavior of the iterates, but it destroys the quadratic convergence of the method. Further, it is hard to guess what the softening factor should be. There are tricks to soften the iteration in such a way that when it starts to converge, the softening goes away ($\alpha \to 1$). One such trick is to compute

$$\begin{aligned} \Delta x &= -J(\mathbf{x}^{(k)})^{-1}\mathbf{f}(\mathbf{x}^{(k)}) \\ \alpha &= \frac{1}{1+\beta\|\Delta x\|} \\ \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \alpha\Delta x \end{aligned} \tag{6}$$

where the value $\beta = 10$ is a conveniently chosen constant. You should be able to see how you might prove that this softening strategy does not destroy the quadratic convergence rate, or, at least, allows a superlinear rate.

Remark: Note that the expression in (6) is designed to keep the largest step below one tenth of the Newton step. This is a very conservative strategy. Note also that another quantity could be placed in the denominator, such as $\|\mathbf{f}\|$, so long as it becomes zero at the solution.

Exercise 9:

(a) Starting from your `snewton0.m` file, copy it to a new file named `snewton1.m`, change its signature to

```
function [x,numIts]=snewton1(f,x,maxIts)
```

and change it so that α is not the constant 1/2, but is determined from Equation (6). Don't forget to change the comments.

(b) How many iterations are required to converge, starting from `x(1)=0.20` and the other components equal to their converged values? How many iterations were required by `snewton0`? You should see that `snewton1` has a slightly larger ball of convergence than `snewton0`, and converges much faster.

(c) Using `snewton1.m`, to the nearest 0.01, how large can the first component of the initial guess get before the iteration diverges? (Leave the other three values at their correct values.)

Another strategy is to choose α so that the objective function almost always decreases on each iteration. This strategy can be useful when the the previous approach is too slow. One way to implement this strategy is to add an additional loop *inside* the Newton iteration loop. Begin with a full Newton step, but check that this step reduces the objective function. If it does not, halve the step size and try again. Keep halving the step size until either the objective function is reduced or some fixed maximum number of halvings (*e.g.*, ten) have been tried. This can be written as an algorithm:

1. On Newton iteration $k+1$, start with $\alpha = 1$.
2. Compute a trial update:
$$\tilde{\mathbf{x}}^{(\ell)} = \mathbf{x}^{(k)} - \alpha J(\mathbf{x}^{(k)})^{-1}\mathbf{f}(\mathbf{x}^{(k)})$$
3. If $|\mathbf{f}(\tilde{\mathbf{x}}^{(\ell)})| \le |\mathbf{f}(\mathbf{x}^{(k)})|$ or if $\alpha < 1/1024$, then set $\mathbf{x}^{(k+1)} = \tilde{\mathbf{x}}^{(\ell)}$ and continue with the next Newton iteration,
otherwise replace α with $\alpha/2$ and return to Step 2.

The limit $\alpha < 1/1024$ represents the (arbitrary) choice of a maximum of ten halvings.

Exercise 10:

(a) Starting from your `snewton1.m` file, copy it to a new file named `snewton2.m`, and employ the strategy described above to repeatedly halve α until $\mathbf{f}(\mathbf{x})$ is reduced. If ten halving steps are taken without reducing $\mathbf{f}$, accept the smallest value of α and continue with the Newton iteration.
Remark: The above algorithm can be written either as a `while` loop inside a `for` loop or as two `for` loops, one inside the other. Indentation should help you keep the loops organized.
Remark: Pay careful attention when writing this program! The value of α should start over at $\alpha = 1$ on *each* Newton iteration.

(b) Using `snewton2.m`, to the nearest 0.01, how large can the first component of the initial guess get before the iteration diverges? (Leave the other three values at their correct values.)
Warning: You may find this strategy is not better than the previous one!

7 Continuation or homotopy methods

As can be seen in the previous few exercises, there are ways to improve the radius of convergence of Newton's method. For some problems, such as the curve-fitting problem above, they just don't help enough. There is a class of methods called continuation or homotopy methods (or Davidenko's method, Ralston and Rabinowitz, p. 363f) that can be used to find good initial guesses for Newton's method. These methods do not appear to be discussed in Quarteroni, Sacco, and Saleri. Some other references:

- `http://www.math.uic.edu/%7ejan/srvart/node4.html`
- An online article (apparently originally by Davidenko) and the references therein, `https://www.encyclopediaofmath.o` and also `https://www.encyclopediaofmath.org/index.php/Continuation_method_(to_a_parametrized_family,_f`
- Werner Rheinboldt, *Methods for solving systems of nonlinear equations*, Section 7.3, p84ff.

In the previous section, we were concerned with solving for the minimum value of an objective function $F(x)$. Suppose there is another, much simpler, objective function $\Phi(x)$, whose minimum is easy to find using Newton's method. One possible choice would be $\Phi(x) = \|x - x_0\|^2$, for some choice of x_0. For $0 \le p \le 1$, consider the new objective function

$$G(x, p) = pF(x) + (1 - p)\Phi(x). \tag{7}$$

When $p = 0$, G reduces to Φ and is easy to minimize (to a result that we already know) but when $p = 1$, G is equal to F and its minimum is the desired minimum. All you need to do is minimize $G(x, p)$ for the sequence $0 = p_1 < p_2 < \cdots < p_{n-1} < p_n = 1$ where you use the solution x_k from parameter p_k as the initial guess for the p_{k+1} case. For properly chosen sequences, the result p_k from step k will be within the radius of convergence for the next step p_{k+1} and the final minimum will be the desired one. The trick is to find a "properly chosen sequence," and there is considerable mathematics involved in doing so. In this lab, we will simply take a uniform sequence of values. This method can work quite nicely, as you see in the following exercise.

Exercise 11:

(a) Suppose that x_0 is a fixed four-dimensional vector and x is a four-dimensional variable. Define

$$\Phi(x) = \sum_{k=1}^{4} (x_k - (x_0)_k)^2.$$

Its gradient is

$$\phi_k = \frac{\partial \Phi}{\partial x_k}$$

and the Jacobian matrix is given by

$$J_{k,\ell} = \frac{\partial \phi_k}{\partial x_\ell}$$

The following code outline computes Φ and its derivatives in a manner similar to `objective.m`.

```
function [f,J,F]=easy_objective(x,x0)
% [f,J,F]=easy_objective(x-x0)
% more comments

% your name and the date

if norm(size(x)-size(x0)) ~= 0
  error('easy_objective: x and x0 must be compatible.')
end
F=sum((x-x0).^2);
% f(k)=derivative of F with respect to x(k)
f=zeros(4,1);
f= ???
% J(k,ell)=derivative of f(k) with respect to x(ell)
J=diag([2,2,2,2]);
```

Copy it into a file named `easy_objective.m` and complete the expression for the vector `f`.

Remark: The chosen value for `x0` is problem-related and amounts to a vague approximation to the final solution.

(b) What are the values of `f, J` and `F` for `x0=[0;2;1;2]` and `x=[0;0;0;0]` and also for `x0=[0;2;1;2]` and `x=[1;-1;1;-1]`? You should be able to confirm that these values are correct by an easy calculation.

(c) Use cut-and-paste to copy the following code to a file named `homotopy.m`, and complete the code.

```
function [f,J,F]=homotopy(x,p,x0)
% [f,J,F]=homotopy(x,p,x0)
% computes the homotopy or Davidenko objective function
% for 0<=p<=1

[f1,J1,F1]=objective(x);
[f2,J2,F2]=easy_objective(x,x0);
f=p*f1+(1-p)*f2;
J=???
F=???
```

(d) Place the following code into a Matlab script file named `exer11.m`.

```
x0=[0.24; 2; 1; 3];
x=ones(4,1);
STEPS=1000;
MAX_ITERS=100;
p=0;
% print out table headings
fprintf('   p    n   x(1)    x(2)    x(3)    x(4)\n');
```

```
for k=0:STEPS
  p=k/STEPS;
  [x,n]=vnewton(@(xx) homotopy(xx,p,x0),x,MAX_ITERS);
  % the fprintf statement is more sophisticated than disp
  if n>3 || k==STEPS || mod(k,20)==0
    fprintf('%6.4f %2d %7.4f %7.4f %7.4f %7.4f\n',p,n,x);
  end
end
```

Try this code with `x0=[0.24; 2; 1; 3]`. Does it successfully reach the value `p=1`? Does it get the same solution values for `x` as in Exercises 5,6,8 and 9?

(e) Explain what the expression

```
@(xx) homotopy(xx,p,x0)
```

means in the context of `exer11.m`. Why is this construct used instead of simply using `@homotopy`?

(f) Test `x0=[.25;2;1;3];` in `easy_objective`. Does the `exer11` script successfully reach the value `p=1`? To the nearest 0.05, how far can you increase the first component and still have it successfully reach `p=1`?

(g) Success of the method depends strongly on the number of steps taken in moving from the simple objective to the true objective. Change `STEPS` from 1000 to 750, thus increasing the size of the steps. Starting from `x0=[.25;2;1;3];` in `easy_objective`, to the nearest 0.05, how far can you increase the first component and still have it successfully reach `p=1`?

(h) Returning to `STEPS=1000`, can you start from `x0=[0;2;1;3]` and reach the solution? How about `x0=[-0.5;2;1;3]`?

As you can see, the idea behind continuation methods is powerful. The best that could be done in Exercise 10 is deviate from the exact answer by a few percent, while in Exercise 11 you more than tripled the first component and still achieved a correct solution. Nonetheless, some care must be taken in choosing `x0` and `STEPS`. Nothing is free, however, and you probably noticed that 1000 steps takes a bit of time to complete.

8 Quasi-Newton methods

The term "quasi-Newton" method basically means a Newton method using an approximate Jacobian instead of an exact one. You saw in Lab 4 that approximating the Jacobian can result in a linear convergence rate instead of the usual quadratic rate, so quasi-Newton methods can take more iterations than true Newton methods will take. On the other hand, inverting an $N \times N$ matrix takes time proportional to N^3, while solving a matrix (after inverting it) takes time proportional to N^2.

For very large linear systems, then, enough time could be saved by solving an approximate Jacobian system to make up for the extra iterations required because true Newton converges faster. In the exercise below, the inverse of the Jacobian matrix will be saved from iteration to iteration, and under proper circumstances, re-used in later iterations because it is "close enough" to the inverse of the true Jacobian matrix.

Remark: You will see next semester that one almost never constructs the inverse of a matrix because simply solving a linear system takes much less time than constructing the inverse and multiplying by it. Furthermore, solving a linear system by a direct method involves constructing two matrices, one lower triangular and one upper triangular. These two matrices can be solved efficiently, and the two matrices can be saved and used again. In this lab, however, we will be constructing the inverse matrix and saving it for future use because it is conceptually simpler.

One choice of nonlinear vector function can be given by the following expression for its components, for $k = 1, \ldots, N$,

$$(f_{10}(x))_k = (d_k + \varepsilon)x_k^n - \sum_{j=k+1}^{N} \frac{x_j^n}{j^2} - \sum_{j=1}^{k-1} \frac{x_{k-j}^n}{j^2} - \frac{k}{N(1+k)}$$

where $n = 2$, $N = 3000$, $\epsilon = 10^{-5}$ and

$$d_k = \sum_{j \neq k} \frac{1}{j^2}.$$

Note that $d_k < \sum_{k=1}^{\infty} 1/k^2 = \pi^2/6 = B_1$, the first Bernoulli number.

Exercise 12:

(a) Copy the following code to a function m-file `f10.m` and fill in values for the Jacobian matrix `J` by taking derivatives "in your head" and using the resulting formulæ in place of `???`.

```
function [f,J]=f10(x)
% [f,J]=f10(x)
% large function to test quasi-Newton methods

% your name and the date

[N,M]=size(x);
if M ~= 1
  error(['f10: x must be a column vector'])
end

n=2;
f=zeros(N,1);
J=zeros(N,N);

for k=1:N
  d=0;
  for j=k+1:N
    d=d+1/j^2;
    f(k)=f(k) - x(j)^n/j^2;
    % J(k,j)=df(k)/dx(j)
    J(k,j)= ???
  end
  for j=1:k-1
    d=d+1/j^2;
    f(k)=f(k) - x(k-j)^n/j^2;
    % J(k,k-j)=df(k)/dx(k-j)
    J(k,k-j)= ???
  end
  f(k)=f(k) + (d+1.e-5)*x(k)^n - k/(1+k)/N;
  % J(k,k)=df(k)/dx(k)
  J(k,k)= ???
end
```

(b) It is extremely important for this exercise that the Jacobian matrix is correctly computed. The terms in $f_{10}(x)$ are all quadratic in x, and for any quadratic function $g(x)$,

$$\frac{dg}{dx} = \frac{g(x + \Delta x) - g(x - \Delta x)}{2\Delta x}$$

exactly for any reasonable value of Δx.

Choose $x = [1; 2; 3]$, $\Delta x = 0.1$, and $N = 3$, use this formula to compute the nine derivatives

$$\frac{d(f_{10})_k}{dx_j}, \qquad \text{for } k, j = 1, \ldots, 3$$

Compare these nine values with the with the values in the Jacobian matrix. If they do not agree, be sure to find your error.

Hint: You can do this in three steps by choosing the three column vectors $\Delta x = [0.1; 0; 0]$, $\Delta x = [0; 0.1; 0]$ and $\Delta x = [0; 0; 0.1]$,

You have seen that when your `vnewton` function is converging quadratically, the ratio `r1` becomes small. One way to improve speed might be to stop using the current Jacobian matrix when `r1` is small, and just use the previous one. If you save the inverse Jacobians from step to step, this can improve speed. In the next exercise, you will construct a quasi-Newton method that does just this.

Exercise 13:

(a) Make a copy of your `vnewton.m` file called `qnewton.m` and change the signature to read

```
function [x,numIts]=qnewton(func,x,maxIts)
% [x,numIts]=qnewton(func,x,maxIts)
% more comments

% your name and the date
```

(b) Just before the start of the loop, initialize a variable

```
skip = false;
```

(c) Replace the two lines defining `oldIncrement` and `increment` with the following lines

```
if ~skip
  tim=clock;
  Jinv=inv(derivative);
  inversionTime=etime(clock,tim);
else
  inversionTime=0;
end
oldIncrement=increment;
increment = -Jinv*value;
```

(d) Add the following lines just after the computation of `r1`.

```
skip = r1 < 0.2;
fprintf('it=%d, r1=%e, inversion time=%e.\n',numIts,r1, ...
inversionTime)
```

Make sure there are no other print statements inside `qnewton.m`.

(e) In addition, make a copy of your completed `qnewton.m` to a new file `qnewtonNoskip.m` (or use "save as"). Change the name inside the file, and replace The line

```
skip = r1 < 0.2;
```

with

```
skip = false;
```

This will give you a file for comparing times for true Newton against those for quasi-Newton.

(f) The following command both solves a moderately large system and gives the time it takes:

```
tic;[v,its]=qnewton(@(x) f10(x),linspace(1,10,3000)');toc
```

How long does it take? How many iterations? How many iterations were skipped (took no time)? What is the total time for inversion as a percentage of the total time taken?

(g) Repeat the same experiment but use `qnewtonNoskip`.

```
tic;[v0,its0]=qnewtonNoskip(@(x) f10(x),linspace(1,10,3000)');toc
```

How long does it take? How many iterations? How far apart are the solutions (`norm(v-v0)/norm(v)`?

(h) For this case, which is faster (takes less total time) qnewton or qnewtonNoskip? **Remark:** On my computer, `qnewton` is about 5 seconds faster. If you have a faster computer, you may observe a smaller difference, or even that `qnewton` is slower. If so, you can try the same comparison with `N=4000` or larger.

Remark 1: As mentioned earlier, `qnewtonNoskip` is slower than `vnewton` because the inverse matrix is constructed explicily. Instead of computing and saving `Jinv`, you could compute and save the factors of the matrix. A version of `qnewtonNoskip` programmed this way would take about the same amount of time as `vnewton`, and `qnewton` would be faster yet.
Remark 2: The choice `skip = (r1 < 0.2)` is very much problem-dependent. There are more reliable ways of deciding when to skip inverting the Jacobian, but they are beyond the scope of this lab.
Remark 3: For large matrix systems, especially ones arising from partial differential equations, it is faster to solve the system using iterative methods. In this case, there are two iterations: the nonlinear Newton iteration and the linear solution iteration. For these sysems, it can be more efficient to stop the linear solution iteration before full convergence is reached, saving the time for unnecessary iterations. This approach is also called a "quasi-Newton method."
Remark 4: When a sequence of similar problems is being solved, such as in Davidenko's method or in time-dependent partial differential equations, quasi-Newton methods can save considerable time in the solution at each step because it is often true that the Jacobian changes relatively slowly.

9 Extra Credit: More on minimization (8 points)

In Section 5 above, you used an objective function that I provided for you. In this section, you will see how a (simpler) objective function is generated.

Suppose you have experimental data that, based on physical grounds, you expect to behave as $ce^{t/\tau}$, where c is a constant and τ a time constant. This would be the case, for example, if the quantity satisfies a first-order differential equation. Given the experimental data the task is to find values for c and τ that best fit the data.

Denoting the experimental data set as the pairs $\{(t_n, v_n)\}$ for $n = 1, \ldots, N$, then the objective function can be written as

$$F(\mathbf{x}) = \sum_{n=1}^{N} (v_n - x_1 e^{x_2 t_n})^2 \tag{8}$$

and the task is to find that value of $\mathbf{x} = (x_1, x_2)$ that minimizes F. This objective function is similar to, but simpler than, the one given above (3). Like that problem, it can be phrased as a linear problem (take logs), but we will treat it as a nonlinear problem.

In the following exercise, you will construct a function m-file named `extra.m`, analogous to `objective.m` that computes the function F in (8). Newton iteration applied to this objective function is sensitive to initial guesses in much the same way that is discussed in Section 5.

Exercise 14:

(a) Begin `extra.m` with the signature

```
function [f,J,F]=extra(x)
```

and add appropriate comments.

(b) Construct 200 artificial data values for `t` between 0 and 1, and for `v` based on values $v = ce^{t/\tau}$ with $c = 1$ and $\tau = 5$.

(c) Construct the objective value F according to (8).

(d) Compute the derivatives $f_1 = \partial F/\partial x_1$ and $f_2 = \partial F/\partial x_2$. You should take these derivatives using pencil and paper (or some computer algebra program such as Maple, Mathematica or the symbolic toolbox in Matlab) and then write Matlab code in `extra.m`.

(e) Compute the Jacobian matrix (or Hessian matrix) by computing the partial derivatives $J_{k,\ell} = \partial f_k/\partial x_\ell$ and writing Matlab code for them.

(f) Use your `extra.m` to compute `f`, `J` and `F` for `x=[1;.2]`. This should yield $F = 0$, $\mathbf{f} = \mathbf{0}$.

(g) Check `f` using *first order* finite differences with $\Delta x_1 = .001$ and $\Delta x_2 = .0001$ in the following way.

i. Compute `[fa,Ja,Fa]=extra([2;.3])`.

ii. Compute `[fb,Jb,Fb]=extra([2.001;.3])`.

iii. Compute `[fc,Jc,Fc]=extra([2;.3001])`.

iv. `(Fb-Fa)/.001` should approximate `fa(1)`, and `(Fc-Fa)/.0001` should approximate `fa(2)`.

(h) Similarly, check that the finite difference approximations for the derivatives $\partial f_i/\partial x_j$ are in approximate agreement with the components of `J`.

(i) Double-check that at $x = [1; .2]$, J a positive definite, symmetric (nonsingular) matrix.

(j) Check that `vnewton` can find the correct solution starting from the correct solution ($x = [1; .2]$).

(k) Show that `vnewton` requires fewer than 10 iterations to find the correct minimum starting from `[1;0.9]` but that starting from `[1;1.0]` requires more than 100 iterations. (If you do not find these vaues, your `J` is probably not correct. Repeat the finite difference checks with $\Delta x_1/2$ and $\Delta x_2/2$ and make sure the approximation error is roughly halved.)

Last change `$Date: 2016/09/18 00:27:23 $`