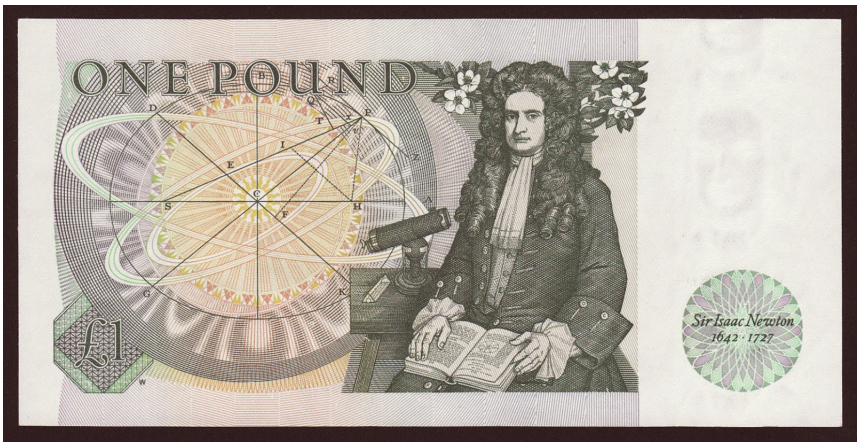
Nonlinear equations: Newton's method for systems

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/~jburkardt/classes/math2070_2019/nonlinear_newton_system/nonlinear_newton_system.p

---



Newton's method can be extended to multiple dimensions.

Rootfinding (Newton version)

Estimate a solution x^* of a system of n nonlinear equations $f(x) = 0$ in n unknowns, given the jacobian $f'(x)$, and a starting point x.

1 Newton's linear model for F(X)

Newton's method for solving a nonlinear equation $f(x) = 0$ can be generalized to the n-dimensional case. The value of the variable and the value of the function are now n-dimensional vectors, and when we can, we will write these as X and $F(X)$ to remind us that they are no longer scalars. Since our examples will all be two dimensional, we may sometimes write (x, y) instead of X. The derivative now becomes the jacobian matrix (or simply, "the jacobian"), which we will write as the $n \times n$ matrix $DF(X)$. The (i, j) entry is

$$DF_{i,j} = \frac{\partial f_i}{\partial x_j}$$

Assuming sufficient differentiability, we can again use a Taylor expansion at a point X near the root X^*:

$$F(X^*) = F(X) + DF(X) * (X^* - X) + O(||X^* - X||)^2$$

Noting that $F(X^*) = 0$, and dropping the error terms, we arrive at Newton's estimate for the root:

$$X^* = X - DF^{-1}(X)\, F(X)$$

where formally, $DF^{-1}(X)$ denotes the inverse matrix associated with $DF(X)$. In fact, it is preferred to write this system in the equivalent form:

$$\text{Solve for } DX^k\text{: } DF(X^k)\ DX^k = -F(X^k)$$
$$\text{Update: } X^{k+1} \leftarrow X^k + DX^k$$

By writing the process this way, we are suggesting that instead of computing the inverse matrix and multiplying, one should set up and solve the given linear system, which will be more efficient and more accurate.

As in the scalar case, we can generally only expect good convergence when our starting guess is near to a root, so that the linearized Taylor system is an adequate model.

We hope that, once the iteration is close to the root, the norm of the function will decrease on every step, and the size of the step $||X^k - X^{k-1}||$ will decrease quadratically, as it becomes similar to the size of the actual (but unknowable) error $||X^* - X^{k-1}||$.

2 Newton's method pseudocode

The pseudocode for the multidimensional Newton's method is very similar to that for the scalar algorithm, although we now need to do a linear solve in the middle of each step.

```
## newton_system1_pseudocode.txt

function newton_system1 ( )

Input: f(), fp(), x, xtol, ftol, itmax

it = 0

Begin loop

  it = it + 1
  Solve for DX: DF(X) * DX = - F(X)
  Set X = X + DX

  if ||DX|| is less than xtol and ||F(X)|| is less than ftol, success
  if it > itmax, failure

End loop

Output: values of updated X and it
```

Listing 1: Pseudocode for simple Newton's method for systems

This pseudocode is implemented in MATLAB as *newton_system1.m*.

Notice that, where the scalar Newton code set

```
dx = - f(x) / fp(x);
```

the Newton system code must write the analogous matrix/vector statement:

```
Solve for DX: DF * DX = - F
```

In MATLAB, this can be accomplished by a command like:

```
dx = - fp ( x ) \ f ( x );
```

Here the backslash indicates that a linear system is to be solved.

3 Example: The Trig2 problem

In class, the following system of two nonlinear equations was considered:

$$F_1(X) = \cos(x) - y$$
$$F_2(X) = x - \sin(y)$$

The corresponding jacobian matrix is:

$$DF = \begin{pmatrix} -\sin(x) & -1 \\ 1 & -\cos(y) \end{pmatrix}$$

To use Newton's method, we prepare these two functions as MATLAB files *trig2.m*

```
function value = trig2 ( xy )
  x = xy(1);
  y = xy(2);

  value = [ ...
    cos(x)-y; ...
    x-sin(y) ];

  return
end
```

Listing 2: trig2.m

and *trig2_fp.m*.

```
function value = trig2_fp ( xy )
  x = xy(1);
  y = xy(2);

  value = [ ...
    -sin(x), -1.0; ...
    1.0,     -cos(y) ];

  return
end
```

Listing 3: trig2_fp.m

Let us apply our Newton system code to this problem, using a starting point of $(1, 1)$:

```
x = [1.0; 1.0];
xtol = 0.00000001;
ftol = 0.00000001;
itmax = 50;

[ x, it ] = newton_system1 ( @(x)trig2(x), @(x)trig2_fp(x), x, xtol, ftol, itmax );
```

Listing 4: Newton's method for the trig2 system

The algorithm returns after just four steps. Here are the intermediate results:

it	x	y	$\|\|f(x,y)\|\|$
0	1.0	1.0	0.486265
1	0.7202728496702477	0.7756845865336229	0.0313296
2	0.6949521502342055	0.7683270627718570	0.000243455
3	0.6948196999805071	0.7681691575522178	1.09748e-08
4	0.6948196907307875	0.7681691567367960	0

While the values $f(x)$ seem to be showing quadratic convergence, the true quadratic behavior is in the successive values of x and y, in which the number of correct digits seem to roughly double on each step.

4 Exercise: The bill2 function

As an in-class exercise, let's see what we need to do to apply Newton's method to a new example, the *bill2()* function, defined by:

$$F_1(x,y) = x - y + \sin(y)$$
$$F_2(x,y) = 3\sin(x) - y$$

Do the following:

1. Copy the file *newton_system1.m* from the website;
2. Create a function file *bill2.m*;
3. Create a derivative file *bill2_fp.m*;
4. Assume that $(1,1)$ is a good starting point;
5. Write a script, or interactively issue commands, to call *newton_system()*;
6. Report your solution, and the number of iterations required.

5 Example: Estimating the convergence rate with poly2

Just as in the one-dimensional case, we are interesting in observing the convergence rate of the Newton method in higher dimensions. If we simply replace absolute values by norms, our calculations of `alpha` and `r` are pretty much the same.

The code becomes more complicated, because of the bookkeeping, but we did the very same calculations for the scalar case:

```
## newton_system2_pseudocode.txt

function newton_system2 ( f, fp, x, xtol, ftol, itmax )

it = 0
DX = 0
alpha = 0

Begin loop

  it = it + 1

  Set Xold to X
  Set DXold to DX
  Evaluate F(X) and DF(X)
  Solve DF * DX = - F for DX
  Set X = X + DX

  Set alpha_old to alpha
  if ||DX_old|| is not 0, set alpha to ||DX|| / ||DX_old||
    if alpha_old is not 0, set r = log ( alpha ) / log ( alpha_old )

  if ||DX|| is less than xtol and ||F(X)|| is less than ftol, success
  if it > itmax, failure

End loop
```

Listing 5: Pseudocode for newton code #2 to estimate convergence rate

This pseudocode is implemented in MATLAB as *newton_system2.m*.

In order to demonstrate the estimation of the convergence rate, we will use a new example function called *poly2*:

$$F_1(x,y) = x^2 + 4y^2 - 9.0$$
$$F_2(x,y) = 18y - 14x^2 + 45.0$$

We use a starting value of $X = (x,y) = [1,-1]$ and call `newton_system2()`. Because we are using *newton_system2()*, we can set the global variable `print_r` to `true` so that the convergence rate estimates are printed. Our results are reassuring:

it	α	log(α)	r
2	0.177797	-1.7271	
3	0.0329564	-3.41257	1.97588
4	0.00110876	-6.80451	1.99396
5	1.21189e-06	-13.6233	2.0021

The starting point must already have been close enough to the region of quadratic convergence, since all our estimates for the convergence rate are close to the optimal value of 2.

6 Example: A failure with multi2

Let us consider a new test function *multi2()*, which has the form:

$$F_1(x,y) = x + 3 * \log(|x|) - y^2;$$
$$F_2(x,y) = 2 * x^2 - x * y - 5 * x + 1$$

If we start from $X = [2,2]$, the iteration does not proceed well:

it	x	y	$\|\|f(x,y)\|\|$
0	2.000000	2.000000	5.000631
1	-18.158883	-10.579441	572.197274
2	-8.371007	-5.228726	142.299878
3	-3.552491	-2.719070	35.078319
4	-1.201458	-1.472826	8.600258
5	-0.000412	0.094904	23.408007
6	0.045101	-1865.231218	3479096.751689
7	0.022073	-932.612712	869777.890008
8	0.010018	-466.299349	217448.883733

The iteration runs to the limit of 50 steps without convergence; the norm of the function value rises and falls as the method seems unable to converge. Such a problem can arise because the starting point is too far from the root, because of errors in the jacobian, because the linear approximation breaks down or is only valid extremely close to the root, or because the function has several distinct but adjacent roots which are "confusing" the iteration.

7 Damping an unsatisfactory Newton step

In the results for the *multi2* function, we see on iterations 0 and 4, on the next iterate, the function norm increases. Assuming the jacobian `DF` has been correctly calculated, the quantity $DX = -DF^{-}1\,F(X)$ must

be a *descent direction*, that is, if we start at X and take a small enough step β in the DX direction, the norm $||F(X + \beta * DX)||$ must be smaller than $||F(X)||$. Since the standard Newton step is taking $\beta = 1$, if we notice that this step results in a rising function norm, we can try recomputing the new iterate with a smaller value of β. If that doesn't help, then we can try a few more reductions before we just give up. This kind of stepsize reduction is known as *damping*.

```
## damped_system_pseudocode.txt

function damped_system ( f, fp, x, xtol, ftol, itmax )

it = 0

Begin loop

  it = it + 1

  Set Xold to X
  Solve for DX: DF(X) * DX = - F(X)

   Initialize beta to 1.0
   begin loop
     Set X = Xold + beta * DX
     if ( ||F(X)|| <= ||F(Xold)||
       Optionally print it, beta, ||F(X)||
       break from this damping loop
     if ( beta < 1.0 / 1024 )
       return with failure
     beta = beta / 2.0
   end loop

  if ||DX|| is less than xtol and |F(X)| is less than ftol, return success
  if it > itmax, return with failure

End loop
```

Listing 6: Pseudocode for damped Newton

8 Assignment #5: A damped Newton method

1. Starting with the code *newton_system1.m* and the pseudocode for a damped Newton method, create a new code *damped_system.m* that implements the damping idea.
2. Apply your code to the *multi2* function, again using the starting point of $X = [2, 2]$. Insert a print statement that displays the values of `it`, *beta* and `||f(x)||`.
3. Run your code and demonstrate that it converges to a root of the *multi2* function.

Turn in: your file *damped_system.m* by Friday, September 27.