Selected Solutions to Assignment #4

Exercise 1: (*It is o.k. for there to be a fair number of variations on this solution among your work. Here is just my sample.*)

(a) We apply Newton's method to $f(x) = x^2 - S$ for $S > 0$, to get

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - S}{2x_n} = x_n - \left(\frac{x_n}{2} - \frac{S}{2x_n}\right) = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right).$$

Thus x_{n+1} is an average, of x_n and the ratio S/x_n. (The ratio is x_n if $x_n = \sqrt{S}$.) To compute x_{n+1} we do exactly three operations, the division S/x_n, the sum, and then another division by two.

Because $f(x)$ is increasing and concave up for $x > 0$, Newton's method will converge for any positive initial guess. Regarding a *good* initial guess, however, note that S is represented inside a computer as $S = a \times 2^b$, where $1 \le a < 2$ and b is an integer, so $\sqrt{S} = \sqrt{a} \times 2^{b/2}$ exactly. In fact, the smallest such a used in representing S is $1.000\ldots000_2$ and the largest is $1.111\ldots111_2$. Therefore I suggest that $x_0 = 1.000\ldots000_2 \times 2^{b/2}$ if b is even and $x_0 = 1.000\ldots000_2 \times 2^{(b-1)/2}$. Thus we actually ignor a in generating x_0; the main idea is to get the magnitude about right for an initial guess about the square root $\sqrt{S}$.

(b) *My longer-than-necessary sample lesson*:

The square root of a number is a number whose square is the first number. For example, the square root of 9 is 3 because $3 \times 3 = 9$. We write $\sqrt{9} = 3$.

Some integers are *perfect squares*, like $4, 9, 36, 81$ which are the squares of $2, 3, 6, 9$. Perfect squares have integer square roots, of course. Other numbers, like 28, for example, will have square roots which are not integers.

How should we find $\sqrt{28}$ in practice? The first thing to notice is that $25 < 28 < 36$ so $5 < \sqrt{28} < 6$. Therefore we expect that $\sqrt{28} \approx 5.2$, or something, just by where 28 lies between 25 and 36. To get better accuracy, there is a rule called the *mechanic's rule* for computing more digits. It gives new values from old values:

$$[\text{new}] = \frac{1}{2}\left([\text{old}] + \frac{28}{[\text{old}]}\right).$$

It requires an initial guess, the first value for "[old]". If that first value is close to the right number then doing the mechanic's rule just a couple of times gets a very accurate estimate of the square root.

Let us use 5 as our initial guess for $\sqrt{28}$, though 5.2 would work better. Then

$$[\text{new}] = \frac{1}{2}\left(5 + \frac{28}{5}\right) = \frac{1}{2}\left(\frac{25}{5} + \frac{28}{5}\right) = \frac{53}{10} = 5.3.$$

Again, and using long division to compute $28/5.3 = 280/53 \approx 5.2830$,

$$[\text{new}] = \frac{1}{2}\left(5.3 + \frac{28}{5.3}\right) = \frac{1}{2}\left(5.3000 + 5.2830\right) = 5.2915.$$

This is a correct five digit approximation to $\sqrt{28}$. We can check its accuracy by multiplying $5.2915 \times 5.2915 = 27.99997225$. (More complete calculations say $\sqrt{28} \approx 5.29150262$.)

Why does the mechanic's rule work? It replaces a guess for the square root by the average of two numbers, the guess itself and the ratio of the original number and the guess. If the guess is low then the average is with a larger number, and vice versa. So the next value is sure to be a better estimate of the root.

Exercise 2: **(a)** A more efficient code, including an example within the "help file" comments (which is a recommended programming practice!):

secantfast.m

```
function x = secantfast(f,x0,x1)
% SECANTFAST  More efficient secant method.  Solves f(x) = 0.
%   Requires two initial guesses x0 ~= x1.  Seeks only 8 digit accuracy.
%   Only one function evaluation per step.  Reports intermediate estimates for x.
% Example:
% >> format long
% >> secantfast(@(x) x-cos(x), 0.7, 0.8)
% x =  0.738565440250903
% x =  0.739078362144669
% x =  0.739085133992362
% x =  0.739085133215159
% ans =  0.739085133215159
% >> fsolve(@(x) x-cos(x), 0.7)   % compare to built-in
% ans =  0.739085133215161

f0 = feval(f,x0);
while abs(x0-x1)/abs(x1) > 1e-8
  f1 = feval(f,x1);
  x = x1 - f1 * (x1 - x0) / (f1 - f0)
  x0 = x1;
  f0 = f1;
  x1 = x;
end
```

(b) My robustified code. I learned again in doing this that how "robust" your code is mostly has to do with how many cases you have tested it on! Frankly, it is just too complicated. Note I used sub-functions to do jobs within the code:

frobust.m

```
function c = frobust(f,a,b)
% FROBUST  Putatively robust combination of secant method
%          and bisection.  To solve f(x) = 0.  Requires bracket
%          to get started.  Goal is to produce bracket with
%          length less than 10^-12.
% Easy example; much faster than bisection:
%   >> frobust(@(x) x-cos(x), 0.7, 0.8)
% Easy example but motivates subtle cases below:
%   >> frobust(@(x) abs(x) - 2, -1, 3)
% Hard example; complex behavior but success; plot this fcn to see:
%   >> frobust(@(x) abs(sin(4*x))- 0.2*x - 0.1,-1,3)
% Impossible example; THINKS it has a root, but it doesn't; evaluate f to see:
%   >> frobust(@(x) sin(1e16*x), -1, 1.05)

fa = feval(f,a);
fb = feval(f,b);
if sign(fa) == sign(fb), error('not a bracket!'), end
printf('                    initial bracket [%15.12f, %15.12f]\n', a, b)

while abs(b-a)/max([abs(a) abs(b)]) > 1e-12
  c = b - fb * (b - a) / (fb - fa)
  fc = feval(f,c);
  if (c >= a) & (abs(c-a) < 1e-12)
    % c is nearly same as a; try to move b very close to a
    [b, fb] = tryreduce_moveb(f,a,b,fb,min(a + 3e-12,b));
```

```
  elseif (c <= b) & (abs(b-c) < 1e-12)
    % c is nearly same as b; try to move a very close to b
    [a, fa] = tryreduce_movea(f,a,b,fa,max(b - 3e-12,a));
  end
  if (c > a) & (c < b) & (abs(fc) < min(abs(fa),abs(fb)))
    % accept c because it is in (a,b) and has better fcn value
    if (b - c) < 0.1 * (b - a)
      % if c is close to b, try to shrink the bracket by moving a
      [a, fa] = tryreduce_movea(f,a,b,fa,b - 3 * (b - c));
    elseif (c - a) < 0.1 * (b - a)
      % if c is close to a, try to shrink the bracket by moving b
      [b, fb] = tryreduce_moveb(f,a,b,fb,a + 3 * (c - a));
    end
    % now update bracket
    if sign(fc) == sign(fa)
      a = c;   fa = fc;
      printf('  secant step (move a); new bracket [%15.12f, %15.12f]\n', a, b)
    else
      b = c;   fb = fc;
      printf('  secant step (move b); new bracket [%15.12f, %15.12f]\n', a, b)
    end
  else
    % reject c; do bisection
    c = (a + b)/2;
    fc = feval(f,c);
    if sign(fc) == sign(fa)
      a = c;   fa = fc;
    else
      b = c;   fb = fc;
    end
    printf('        bisection step; new bracket [%15.12f, %15.12f]\n', a, b)
  end
end

    function [a, fa] = tryreduce_movea(f, a, b, fa, atry);
    fatry = feval(f,atry);
    if (sign(fatry) == sign(fa)) & (abs(fatry) < abs(fa))
      a = atry;   fa = fatry;
      printf('  reduce step (move a); new bracket [%15.12f, %15.12f]\n', a, b)
    end

    function [b, fb] = tryreduce_moveb(f, a, b, fb, btry);
    fbtry = feval(f,btry);
    if (sign(fbtry) == sign(fb)) & (abs(fbtry) < abs(fb))
      b = btry;   fb = fbtry;
      printf('  reduce step (move b); new bracket [%15.12f, %15.12f]\n', a, b)
    end
```

Moler exercise 2.1: We solve the system

$$\begin{aligned} 3a + 12b + c &= 2.36 \\ 12a + 2c &= 5.26 \\ 2b + 3c &= 2.77 \end{aligned}$$

using MATLAB/OCTAVE. In few symbols:

```
>> format bank
>> [3 12 1; 12 0 2; 0 2 3] \ [2.36 5.26 2.77]'
ans =
  0.29
```

```
  0.05
  0.89
```

So apples are 29 cents, bananas 5 cents, and cantaloupes 89 cents. (Not AK prices ...)

Moler exercise 2.3: We set up the system with f_3, f_6, f_7, f_8, f_{11}, f_{13} not present, because we either know their values or can get them from other values:

$$\begin{aligned}
\alpha f_1 - f_4 - \alpha f_5 &= 0 \\
\alpha f_1 + \alpha f_5 &= -10 \\
f_2 + \alpha f_5 - \alpha f_9 - f_{10} &= 0 \\
\alpha f_5 + \alpha f_9 &= 15 \\
f_4 + \alpha f_9 - \alpha f_{12} &= 0 \\
\alpha f_9 + \alpha f_{12} &= -20 \\
f_{10} + \alpha f_{12} &= 0
\end{aligned}$$

The unknowns in this system are $f_1, f_2, f_4, f_5, f_9, f_{10}, f_{12}$.

I put this into an m-file, instead of editing it at the command line, because of the ability to format for clarity:

exertwo3.m

```
% EXERTWO3  Exercise 2.3 in Moler.

alf = 1/sqrt(2);
A = [alf   0  -1 -alf    0   0    0;
     alf   0   0  alf    0   0    0;
       0   1   0  alf -alf  -1    0;
       0   0   0  alf  alf   0    0;
       0   0   1    0  alf   0 -alf;
       0   0   0    0  alf   0  alf;
       0   0   0    0    0   1  alf];
b = [0 -10 0 15 0 -20 0]';
x = A \ b;
f = [x(1) x(2) 10 x(3) x(4) x(2) 0 x(3) x(5) x(6) 20 x(7) x(6)]'
```

The result is this list, $f_1, \ldots, f_{13}$:

```
f =
  -28.28427
   20.00000
   10.00000
  -30.00000
   14.14214
   20.00000
    0.00000
  -30.00000
    7.07107
   25.00000
   20.00000
  -35.35534
   25.00000
```

Now, in "real" engineering life, the linear system would never be seen by the engineer. Rather, a structure would be entered into a computer-aided design (CAD) system element-by-element by the engineer; that

would be the actual work specific to the structure. (Each "element" is a beam with certain dimensions and material properties, in this case, but such elements might also include fasteners and cables and such.) Then a system of linear equations would be automatically assembled by part of the CAD program, and then this set of linear equations would be solved by a routine, like "`A\b`", but invisibly to the engineer *except* that the computed forces would appear on each element, in the display of the structure. My point is that both the *assembly* of the linear system, and its *solution* by numerical methods, are parts of engineering software.

Moler exercise 2.8: It is easy to get this wrong! First I modified the code:

modlutx.m

```
function [L,U,p] = modlutx(A)
%MODLUTX  Triangular factorization.  Modified lutx; this has bland for loops.
%   [L,U,p] = modlutx(A) produces a unit lower triangular matrix L,
%   an upper triangular matrix U, and a permutation vector p,
%   so that L*U = A(p,:)

[n,n] = size(A);
p = (1:n)';

for k = 1:n-1

   % Find index of largest element below diagonal in k-th column
   [r,m] = max(abs(A(k:n,k)));
   m = m+k-1;

   % Skip elimination if column is zero
   if (A(m,k) ~= 0)

      % Swap pivot row
      if (m ~= k)
         for j = 1:n
           temp = A(k,j);
           A(k,j) = A(m,j);
           A(m,j) = temp;
         end
         temp = p(k);
         p(k) = p(m);
         p(m) = temp;
      end

      % Compute multipliers
      for i = k+1:n
        A(i,k) = A(i,k)/A(k,k);
      end

      % Update the remainder of the matrix
      for i = k+1:n
        for j = k+1:n
          A(i,j) = A(i,j) - A(i,k)*A(k,j);
        end
      end
   end
end

% Separate result
L = tril(A,-1) + eye(n,n);
U = triu(A);
```

Then I tested it. In particular,

```
>> A=rand(5);  [L,U,p]=lutx(A); [L U],  [L,U,p]=modlutx(A); [L U],  [L,U,p]=lu(A); [L U]
```

produces three versions of the LU decomposition for the same, which should be identical. After correcting the bugs I introduced, they were.

Now the timing. I used OCTAVE, and your results may vary! I tried merely to get the time between 10 and 11 seconds, and I repeated the runs at least once to convince myself of the timing:

```
>> A=rand(100);  tic, modlutx(A); toc
Elapsed time is 10.825 seconds.
>> A=rand(100);  tic, modlutx(A); toc
Elapsed time is 10.795 seconds.
>> A=rand(760);  tic, lutx(A); toc
Elapsed time is 10.181 seconds.
>> A=rand(760);  tic, lutx(A); toc
Elapsed time is 10.28 seconds.
>> A=rand(1850);  tic, lu(A); toc
Elapsed time is 10.636 seconds.
>> A=rand(1850);  tic, lu(A); toc
Elapsed time is 10.619 seconds.
```

Clearly the built-in `lu` is much faster than the others, and the `for` loops in `modlutx` slow it down a lot.

We will learn that the underlying algorithm of LU decomposition requires $O(n^3)$ operations, so for a fixed code we expect that doubling the size of the matrix will cause a slow-down by a factor of 8.

I will explain in class more about the LU decomposition, which is the numerical analyst's preferred form of Gauss elimination.

Moler exercise 2.9: *Too hard, sorry. Not graded. Will appear on a later assignment, after I have returned your work to you.*