Computer Assignment 2 – Solutions

1. a) Solving linear system with `gaussel`:

```
>> A = [5 3 36 7; 15 9 28 -1; -23 -4 -13 7; 9 3 40 -2]; b = [67; 63; -52; 88];
>> x = gaussel(A, b)
x =
   NaN
   NaN
   NaN
   NaN
```

The naïve GE doesn't work because it produces a zero pivot in the 2nd column:

```
>> M1 = eye(4) - [0; A(2:4,1)./A(1,1)]*[1 0 0 0]
M1 =
    1.0000         0         0         0
   -3.0000    1.0000         0         0
    4.6000         0    1.0000         0
   -1.8000         0         0    1.0000
>> M1*A
ans =
    5.0000    3.0000   36.0000    7.0000
         0         0  -80.0000  -22.0000
         0    9.8000  152.6000   39.2000
         0   -2.4000  -24.8000  -14.6000
```

b) Matlab m-file `gaussel_spp.m`

```
function [x,l] = gaussel_spp(A,b)
% [x, l] = gaussel_spp(A,b)
%
%   This subroutine will perform Gaussian elimination with scaled partial
%   pivoting and back substitution to solve the system Ax = b.
%   INPUT :  A - matrix for the left hand side.
%            b - vector for the right hand side
%
%   OUTPUT : x - the solution vector.
%            l - vector of pivot indices

N = max(size(A));
l = (1:N)';  % Define variable index l
% Compute row scale factors s
 for i=1:N
     s(i) = max(abs(A(i,:)));
 end
% Another way to compute s:
%   s = max(abs(A),[],2);
% Perform Gaussian Elimination
 for j=2:N
% Determine the pivot row k
     mm = abs(A(l(j-1),j-1))./s(l(j-1)); k = j-1;
     for i=j:N
         if mm < abs(A(l(i),j-1))./s(l(i))
             mm = abs(A(l(i),j-1))./s(l(i)); k = i;
         end
     end
% Another way to determine k
%     [~,k] = max(abs(A(l(j-1:N),j-1))./s(l(j-1:N))); k = k+j-2;
     if k ~= j-1
         kk=l(j-1); l(j-1) = l(k); l(k) = kk;  % Swap values of l(j-1) and l(k)
     end
     for i=j:N
        m = A(l(i),j-1)/A(l(j-1),j-1);
        A(l(i),:) = A(l(i),:) - A(l(j-1),:)*m;
        b(l(i)) = b(l(i)) - m*b(l(j-1));
     end
 end

% Perform back substitution
```

```
 x = zeros(N,1);
 x(N) = b(l(N))/A(l(N),N);
 for j=N-1:-1:1
   x(j) = (b(l(j))-A(l(j),j+1:N)*x(j+1:N))/A(l(j),j);
 end
% End of function
```

c) Using GE with scaled partial pivoting:

```
>> [x,l] = gaussel_spp(A, b)
x =
    1.0000
   -1.0000
    2.0000
   -1.0000
l =
     3
     2
     4
     1
```

Compare to the 'exact' result:

```
>> x_exact = [1; -1; 2; -1];
>> norm(x-x_exact)
ans =
   1.9547e-15
```

Solve by '\':

```
>> z = A\b
z =
    1.0000
   -1.0000
    2.0000
   -1.0000
>> norm(z-x_exact)
ans =
   1.8578e-15
```

The accuracy of the result obtained with `gaussel_spp` is nearly the same as that obtained with '\'.

2. a) Matlab m-file `run_newton.m`

```
function x = run_newton(f, fp, x0, N, tol)
  x = x0;  n = 0;
  while n <= N
    fx = f(x);
    if abs(fx) < tol
      break;  % Terminate execution of the 'while' loop
    end
    fpx = fp(x);
    if abs(fpx) < tol
      warning('Warning: f''(x) is small; giving up.');
      return; % Exit from the function
    end
    x = x - fx./fpx;
    n = n + 1;
  end
  if n <= N
    disp(['Solution found after ' num2str(n) ' iterations']);
  else
    disp('Warning: Number of allowed iterations exceeded');
  end
```

b) Use `run_newton` to fine the root of the polynomial:

```
>> f = @(x)(x.^5 - x.^4 - 4*x.^3 + 4*x.^2 + 5*x - 5);
>> fp = @(x)(5*x.^4 - 4*x.^3 - 12*x.^2 + 8*x + 5);
>> x0 = 2.2;
>> x = run_newton(f, fp, x0, 20, 1e-16)
Solution found after 10 iterations
x =
    1.0000
```

c) Modified function outputting all iterates in x:

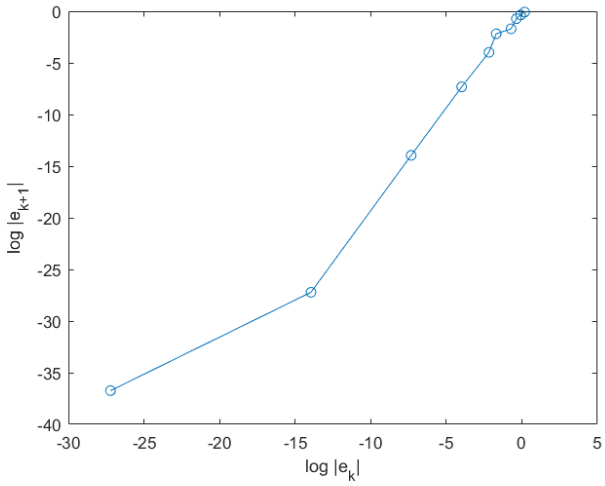
```
function x = run_newton_itr(f, fp, x0, N, tol)
  x = nan(N+2,1); % Allocate vector x for output of all iterates
  x(1) = x0;  n = 0;
  while n <= N
    fx = f(x(n+1));
    if abs(fx) < tol
      break;  % Terminate execution of the 'while' loop
    end
    fpx = fp(x(n+1));
    if abs(fpx) < tol
      disp('Warning: f''(x) is small; giving up.');
      return; % Exit from the function
    end
    x(n+2) = x(n+1) - fx./fpx;
    n = n + 1;
  end
  if n <= N
    disp(['Solution found after ' num2str(n) ' iterations']);
    x = x(1:n+1); % Trim x to the size equal the number of iterates
  else
    disp('Warning: Number of allowed iterations exceeded');
  end
```

Use this function:

```
>> x = run_newton_itr(f, fp, x0, 20, 1e-16)
Solution found after 10 iterations
x =
    2.2000
    1.9215
    1.6968
    1.4885
    1.1831
    0.8866
    0.9813
    0.9993
    1.0000
    1.0000
    1.0000
```

Calculate e_k and plot $\log|e_{k+1}|$ as a function of $\log|e_k|$:

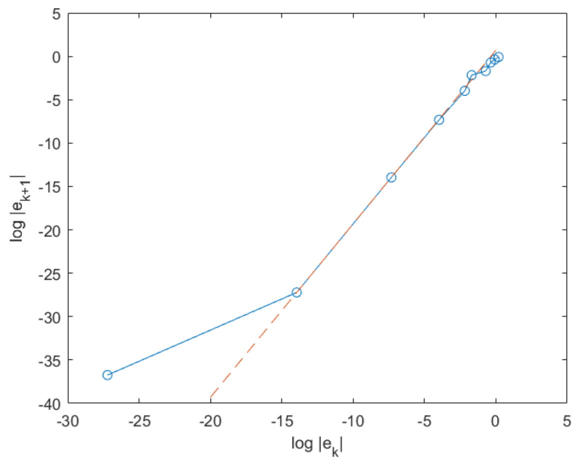
```
>> err = abs(x - 1);
>> plot(log(err(1:end-1)),log(err(2:end)),'o-');
>> xlabel('log |e_k|'); ylabel('log |e_{k+1}|');
```



We expect that $e_{k+1} = Ce_k^r + O(e_k^q)$, where $q > r$. If $O(e_k^q)$ is much smaller than Ce_k^r, then we should observe linear dependence between $\log|e_{k+1}|$ and $\log|e_k|$, that is $\log|e_{k+1}| \approx r\log|e_k| + \log|C|$. When e_k is relatively large, the $O(e_k^q)$ term cannot be neglected, leading to the deviation from linear dependence, while when e_k is very small, the calculation of e_{k+1} suffers from the round-off error (due to finite precision of floating point calculations). This also leads to the deviation from the linear dependence.

In the figure above, we observe linear dependence when $-14 < \log|e_k| < -2$. We can fit the straight line in this interval with the choice $r = 2$ and $\log|C| = 0.7$.

```
>> r = 2.0;  logC = 0.7;
>> hold on;
>> plot([-20 0], r*[-20 0] + logC,'--');
```



The fit looks good, so we observe quadratic convergence ($r = 2$) and $|C| = e^{0.7} \approx 2.0$. From the convergence analysis of Newton's method

$$C = \frac{f''(x^*)}{2f'(x^*)}$$

For the given polynomial function

```
>> fp = @(x)(5*x.^4 - 4*x.^3 - 12*x.^2 + 8*x + 5);
>> fpp = @(x)(20*x.^3 - 12*x.^2 - 24*x + 8);
>> C = fpp(1)/2/fp(1)
C =
    -2
```

This agrees with our estimate.

We can also determine the values of r and $\log|C|$ from each of the three line segments:

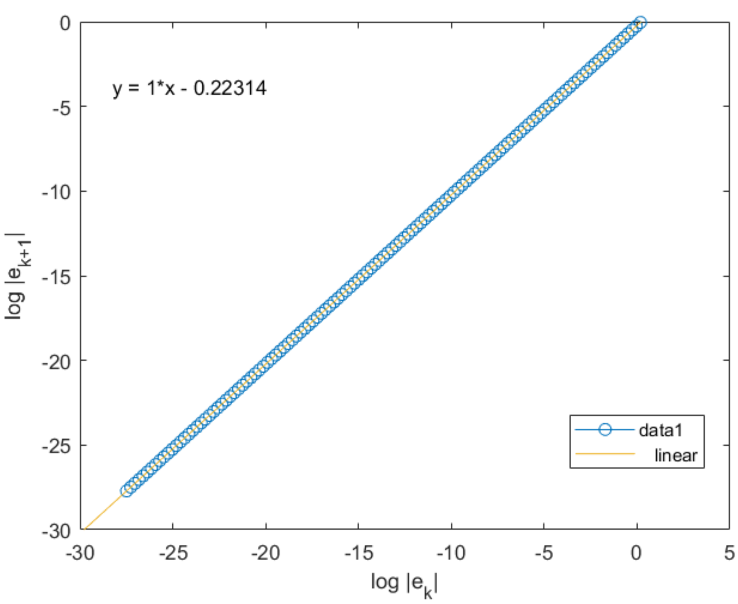
$$r = \frac{\log|e_{k+1}| - \log|e_k|}{\log|e_k| - \log|e_{k-1}|}, \quad \log|C| = \log|e_{k+1}| - r\log|e_k|.$$

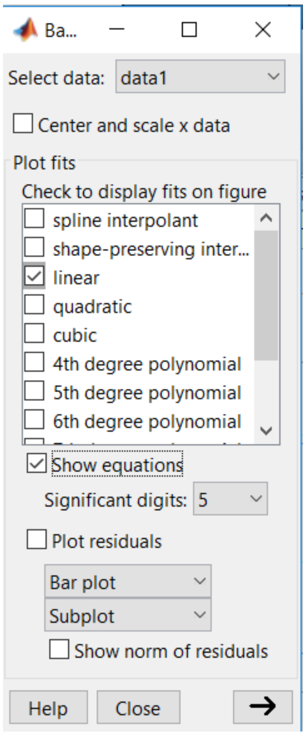
```
>> r = (log(err(3:end))-log(err(2:end-1)))./(log(err(2:end-1))-log(err(1:end-2)))
r =
    1.0579
    1.2710
    2.7631
    0.4885
    3.7609
    1.8538
    1.9840
    1.9997
    0.7181

>> log(err(3:end))-r.*log(err(2:end-1))
ans =
   -0.2747
   -0.2573
    0.2817
   -1.3476
    4.2076
    0.0558
    0.5739
    0.6890
  -17.1933
```

d) Repeat calculations for the real root of $(x-1)^5$.

```
>> f = @(x)(x-1).^5;  fp = @(x)5*(x-1).^4;
>> x = run_newton_itr(f, fp, x0, 200, 1e-60);
Solution found after 125 iterations
>> err = abs(x - 1);
>> plot(log(err(1:end-1)),log(err(2:end)),'o-');
>> xlabel('log |e_k|'); ylabel('log |e_{k+1}|');
```





Using 'Tools → Basic Fitting' from the Figure menu, we obtain a linear fit to the plotted data points with slope $r = 1$ and intercept $\log|C| = -0.22314$, so that $|C| = 0.8$.

This is consistent with the convergence analysis of Newton's method for roots with multiplicity m. In this case $m = 5$.