COMPUTER SCIENCE 349A, SPRING 2018
ASSIGNMENT #1 - SOLUTION

Question #1 - 10 marks.

(a)
```
function [t,v] = skyfall_euler_matrices( m,c,g,t0,v0,tn,n )
          % print headings and initial conditions
          fprintf('values of t approximations v(t)\n')
          fprintf('%8.3f',t0),fprintf('%19.4f\n',v0)
          % compute step size h
          h=(tn-t0)/n;
          % set t,v to the initial values
          t=[t0:h:tn];
          v=zeros(1, n);
          v(1)=v0;
          % compute v(t) over n time steps using Eulers method
          for i=1:n
               v(i+1)=v(i)+(g-c/m*v(i))*h;
                 fprintf('%8.3f',t(i+1)),fprintf('%19.4f\n',v(i+1))
          end
end
```

(b) The following command can be used to calculate the results for the parameters provided:

```
>> [t,v]=skyfall_euler_matrices(73.5,13.1,9.81,0,0,16,80);
values of t approximations v(t)
   0.000              0.0000
   0.200              1.9620
   0.400              3.8541
   0.600              5.6787
   0.800              7.4383
   1.000              9.1351
   1.200             10.7715
   1.400             12.3495
   1.600             13.8713
   1.800             15.3388
   2.000             16.7541
   2.200             18.1188
   2.400             19.4350
   2.600             20.7042
   2.800             21.9282
   3.000             23.1085
   3.200             24.2468
```

3.400	25.3445
3.600	26.4030
3.800	27.4239
4.000	28.4083
4.200	29.3577
4.400	30.2732
4.600	31.1560
4.800	32.0074
5.000	32.8285
5.200	33.6203
5.400	34.3838
5.600	35.1202
5.800	35.8303
6.000	36.5151
6.200	37.1754
6.400	37.8123
6.600	38.4264
6.800	39.0187
7.000	39.5898
7.200	40.1406
7.400	40.6717
7.600	41.1839
7.800	41.6779
8.000	42.1542
8.200	42.6136
8.400	43.0565
8.600	43.4837
8.800	43.8957
9.000	44.2930
9.200	44.6761
9.400	45.0456
9.600	45.4019
9.800	45.7455
10.000	46.0768
10.200	46.3963
10.400	46.7045
10.600	47.0016
10.800	47.2882
11.000	47.5646
11.200	47.8311
11.400	48.0881
11.600	48.3359
11.800	48.5749
12.000	48.8054
12.200	49.0277

```
  12.400            49.2420
  12.600            49.4487
  12.800            49.6481
  13.000            49.8403
  13.200            50.0257
  13.400            50.2044
  13.600            50.3768
  13.800            50.5431
  14.000            50.7034
  14.200            50.8580
  14.400            51.0071
  14.600            51.1509
  14.800            51.2896
  15.000            51.4233
  15.200            51.5523
  15.400            51.6766
  15.600            51.7965
  15.800            51.9122
  16.000            52.0237
```

(c) The function: Note, they may build the vectors explicitly like in (a). This is okay although harder. Also, they do not have to have the fprints, but it looks better if they do. Also, I used the t built in part (a) so I didn't have to include it in the code.

```
function [ v ] = skyfall( g,m,c,t,n)
%skyfall Uses the analytic solution to find the velocity
%   If you pass it t as a vector it will return v as a vector of
%   corresponding velocities.

 v=(g*m/c)*(1-exp(-(c*t/m)));

 % print headings and initial conditions
        fprintf('values of t and values of v(t)\n')
        for i=1:n+1
                fprintf('%8.3f',t(i)),fprintf('%19.4f\n',v(i))
        end
end
```

The call and results:

```
>> [u] = skyfall(9.81,73.5,13.1,t,80);
values of t and values of v(t)
   0.000             0.0000
   0.200             1.9274
```

0.400	3.7874
0.600	5.5822
0.800	7.3142
1.000	8.9855
1.200	10.5983
1.400	12.1546
1.600	13.6564
1.800	15.1056
2.000	16.5041
2.200	17.8536
2.400	19.1558
2.600	20.4124
2.800	21.6251
3.000	22.7952
3.200	23.9244
3.400	25.0141
3.600	26.0656
3.800	27.0802
4.000	28.0594
4.200	29.0042
4.400	29.9160
4.600	30.7958
4.800	31.6448
5.000	32.4641
5.200	33.2547
5.400	34.0176
5.600	34.7538
5.800	35.4643
6.000	36.1498
6.200	36.8113
6.400	37.4497
6.600	38.0657
6.800	38.6602
7.000	39.2338
7.200	39.7873
7.400	40.3215
7.600	40.8369
7.800	41.3343
8.000	41.8143
8.200	42.2775
8.400	42.7244
8.600	43.1557
8.800	43.5719
9.000	43.9736
9.200	44.3611

9.400	44.7351
9.600	45.0960
9.800	45.4442
10.000	45.7803
10.200	46.1046
10.400	46.4175
10.600	46.7195
10.800	47.0109
11.000	47.2921
11.200	47.5634
11.400	47.8253
11.600	48.0780
11.800	48.3218
12.000	48.5571
12.200	48.7841
12.400	49.0032
12.600	49.2147
12.800	49.4187
13.000	49.6156
13.200	49.8055
13.400	49.9889
13.600	50.1658
13.800	50.3365
14.000	50.5012
14.200	50.6602
14.400	50.8136
14.600	50.9616
14.800	51.1045
15.000	51.2423
15.200	51.3754
15.400	51.5037
15.600	51.6276
15.800	51.7471
16.000	51.8624

(d) The command:

```
>> plot(t,u,t,v)
```

The plot:

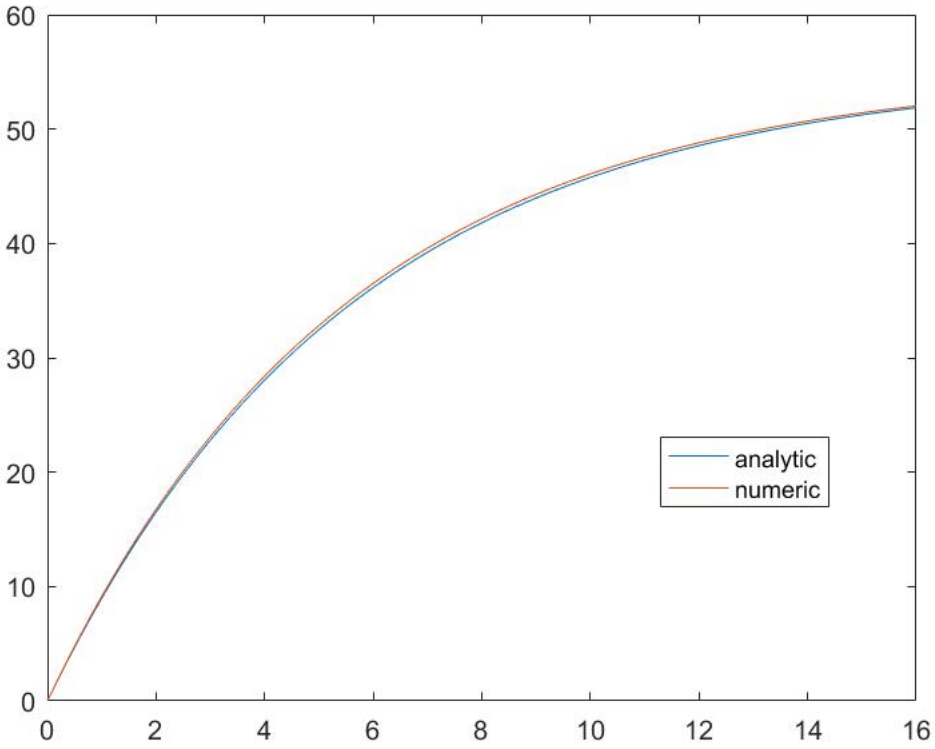

Figure 1: Comparison of numerical and analytical solution

(e) The exact analytical solution is:

$$v(t) = \frac{gm}{c}(1 - e^{-(c/m)t}) \tag{1}$$

When we let $t \to \infty$ then we get $\frac{gm}{c}$ giving a terminal velocity of:

$$\frac{9.81 * 73.5}{13.1} = 55.0408 \tag{2}$$

In MATLAB, running Euler with higher values for tn and n I get:

```
>> [t,v]=skyfall_euler_matrices(73.5,13.1,9.81,0,0,100,100);
values of t approximations v(t)
   0.000              0.0000
```

1.000	9.8100
2.000	17.8716
3.000	24.4963
4.000	29.9403
5.000	34.4140
6.000	38.0903
7.000	41.1114
8.000	43.5941
9.000	45.6343
10.000	47.3108
11.000	48.6885
12.000	49.8207
13.000	50.7511
14.000	51.5157
15.000	52.1440
16.000	52.6603
17.000	53.0846
18.000	53.4332
19.000	53.7198
20.000	53.9552
21.000	54.1487
22.000	54.3077
23.000	54.4384
24.000	54.5458
25.000	54.6340
26.000	54.7065
27.000	54.7661
28.000	54.8151
29.000	54.8553
30.000	54.8884
31.000	54.9155
32.000	54.9379
33.000	54.9562
34.000	54.9713
35.000	54.9837
36.000	54.9939
37.000	55.0023
38.000	55.0091
39.000	55.0148
40.000	55.0194
41.000	55.0232
42.000	55.0264
43.000	55.0290
44.000	55.0311
45.000	55.0328

46.000	55.0342
47.000	55.0354
48.000	55.0364
49.000	55.0372
50.000	55.0378
51.000	55.0384
52.000	55.0388
53.000	55.0392
54.000	55.0395
55.000	55.0397
56.000	55.0399
57.000	55.0401
58.000	55.0402
59.000	55.0403
60.000	55.0404
61.000	55.0405
62.000	55.0406
63.000	55.0406
64.000	55.0406
65.000	55.0407
66.000	55.0407
67.000	55.0407
68.000	55.0408
69.000	55.0408
70.000	55.0408
71.000	55.0408
72.000	55.0408
73.000	55.0408
74.000	55.0408
75.000	55.0408
76.000	55.0408
77.000	55.0408
78.000	55.0408
79.000	55.0408
80.000	55.0408
81.000	55.0408
82.000	55.0408
83.000	55.0408
84.000	55.0408
85.000	55.0408
86.000	55.0408
87.000	55.0408
88.000	55.0408
89.000	55.0408
90.000	55.0408

```
  91.000            55.0408
  92.000            55.0408
  93.000            55.0408
  94.000            55.0408
  95.000            55.0408
  96.000            55.0408
  97.000            55.0408
  98.000            55.0408
  99.000            55.0408
 100.000            55.0408
```

Question #2 - 6 Marks.

(a)
```
function [t,v] = Euler2( m,k,g,t0,v0,tn,n )
        % print headings and initial conditions
        fprintf('values of t approximations v(t)\n')
        fprintf('%8.3f',t0),fprintf('%19.4f\n',v0)
        % compute step size h
        h=(tn-t0)/n;
        % set t,v to the initial values
        t=[t0:h:tn];

        v(1)=v0;
        % compute v(t) over n time steps using Eulers method
        for i=1:n
                v(i+1)=v(i)+(g-k/m*v(i)^2)*h;
                fprintf('%8.3f',t(i+1)),fprintf('%19.4f\n',v(i+1))
        end
end
```

(b)
```
>> [t,v]=Euler2(73.5,0.234,9.81,0,0,18,72);
values of t approximations v(t)
   0.000             0.0000
   0.250             2.4525
   0.500             4.9002
   0.750             7.3336
   1.000             9.7433
   1.250            12.1202
   1.500            14.4558
   1.750            16.7420
   2.000            18.9714
   2.250            21.1374
   2.500            23.2343
   2.750            25.2572
```

3.000	27.2019
3.250	29.0655
3.500	30.8456
3.750	32.5408
4.000	34.1505
4.250	35.6748
4.500	37.1143
4.750	38.4705
5.000	39.7450
5.250	40.9402
5.500	42.0587
5.750	43.1033
6.000	44.0770
6.250	44.9832
6.500	45.8252
6.750	46.6063
7.000	47.3300
7.250	47.9995
7.500	48.6182
7.750	49.1894
8.000	49.7161
8.250	50.2013
8.500	50.6480
8.750	51.0588
9.000	51.4363
9.250	51.7831
9.500	52.1013
9.750	52.3933
10.000	52.6609
10.250	52.9062
10.500	53.1309
10.750	53.3366
11.000	53.5249
11.250	53.6971
11.500	53.8547
11.750	53.9988
12.000	54.1305
12.250	54.2509
12.500	54.3608
12.750	54.4613
13.000	54.5531
13.250	54.6369
13.500	54.7134
13.750	54.7833
14.000	54.8471

```
  14.250            54.9053
  14.500            54.9584
  14.750            55.0069
  15.000            55.0512
  15.250            55.0915
  15.500            55.1284
  15.750            55.1620
  16.000            55.1926
  16.250            55.2206
  16.500            55.2461
  16.750            55.2693
  17.000            55.2905
  17.250            55.3099
  17.500            55.3275
  17.750            55.3436
  18.000            55.3583
```

(c) They can do this by hand or in MATLAB. I did it in MATLAB. First, I wrote a new function for the analytic solution.

```
function [ v ] = Euler2_Analytic(g,m,k,t)
%Analytic solution using 2nd order drag coefficient
    v = sqrt(g*m/k)*tanh(sqrt(g*k/m)*t);
end
```

Then, I ran it with m=73.5, k=0.234, and t=18.

```
>> v18 = Euler2_Analytic(9.81,73.5,0.234,18)

v18 =

   55.3186
```

Finally, I calculated the relative error.

```
>> abs(v(73)-v18)/abs(v(73))

ans =

   7.1624e-04
```

Note, if they calculate by hand it may be slightly different as MATLAB is using more significant digits than it is showing here. For example, using exactly 6 significant digits gives.

```
>> (55.3583-55.3186)/55.3583

ans =

   7.1715e-04
```

Question #3 - 4 Marks

I did this question in MATLAB. They could do it in MATLAB or by hand. First, I will present my three functions, enegx_Taylor1 for the McLaurin series of e^{-x}, enegx_Taylor2 for the McLaurin series of $\frac{1}{e^x}$, and rel_error, my function for calculating the relative error.

```
function [ fx ] = enegx_Taylor1(x,n)
%enegx_Taylor1 Approximates e^(-x) with the
%   nth degree McLaurin polynomial for e^(-x)

    fx = 0;
    for i=0:n
        fx = fx + (-1)^i * x^i / factorial(i);
    end
end

function [ fx ] = enegx_Taylor2(x,n)
%enegx_Taylor2 Approximates e^(-x) with 1 over
%   the nth degree McLaurin polynomial for e^x

    sum = 0;
    for i=0:n
        sum = sum + x^i / factorial(i);
    end
    fx = 1 / sum;
end

function [ ept ] = rel_error(p,p_star)
%rel_error Calculates the relative error between
%   the true value p and the approximation p_star

    ept = abs(1 - p_star / p);
end
```

Then, I ran enegx_Taylor1 for values of $n = 1$ to 5 and stored the results in vector f1x. The entries in this vector correspond to the 1st degree polynomial through 5th degree polynomial approximations. That is, when $n = 1$, then $e^{-x} \approx 1 + x$, when $n = 2$, then $e^{-x} \approx 1 + x + \frac{x^2}{2!}$, etc.

```
>>  for n = 1:5
f1x(n) = enegx_Taylor1(2,n);
end
>> f1x

f1x =

   -1.0000    1.0000   -0.3333    0.3333    0.0667
```

I then calculated the true value storing it in fx and calculated the relative errors corresponding to each of the above approximations. I stored those in vector e1t.

```
>> fx = exp(-2)

fx =

    0.1353

>> e1t = rel_error(fx,f1x)

e1t =

    8.3891    6.3891    3.4630    1.4630    0.5074
```

Finally, I did the same for the second approximating McLaurin series using enegx_Taylor2, storing the approximations in f2x and the corresponding relative errors in e2t.

```
>> for n = 1:5
f2x(n) = enegx_Taylor2(2,n);
end
>> f2x

f2x =

    0.3333    0.2000    0.1579    0.1429    0.1376

>> e1t = rel_error(fx,f2x)

e1t =

    1.4630    0.4778    0.1667    0.0556    0.0168
```

Then, they need to point out that the second function is clearly a better approximation of e^{-x} for this particular value of x.