Table of Contents

Lab 6

```
fprintf(['asdf ' ...
    'asdf'])
```

```
asdf asdf
```

Problem 1

```
fprintf('lsq approximations from n=1 to n=15\n')
fprintf('n\tResult\t\tMax Error\tTime\t\n')
for i=1:1:15,
    [res, time, mxerror] = lsq(i);
    fprintf('%d\t%0.12f\t%0.12f\t%d\n', i, res, mxerror, time)
end
```

```
lsq approximations from n=1 to n=15
n Result  Max Error Time
1 0.000000000000 0.154845482803 1.019346e-03
2 0.000000000000 0.014981393570 7.365126e-05
3 0.000000000000 0.001050258652 9.417702e-05
4 0.000000000000 0.000058444302 1.285879e-04
5 0.000000000000 0.000192262678 1.726579e-04
6 0.000000000000 0.001464447439 2.176334e-04
7 0.000000000000 0.004836136992 2.447999e-04
8 0.000000000000 0.005145269809 2.994346e-04
9 0.000000000002 0.115781064803 3.459194e-04
10 0.000000000171 2.375262766082 4.077986e-04
11 0.000000004504 21.885619185256 4.699796e-04
12 0.000000085473 49.477311167257 5.149551e-04
13 0.000002474509 800.108145212394 6.033970e-04
14 0.000000096428 33.111082610996 7.425496e-04
15 0.000000122914 7.292378213866 7.485866e-04
```

1.a The error when m equals 1,2, and 3 decreases as m grow.

1.b There is an inflection point in the error when m reaches 4. When m reaches values larger than 4 it the error starts to increase as well.

1.c You can see errors when m is 10, 9, 7 (a little bit), 2 (a little bit), and 1.

1.d The error is almost cyclic. When 1 < n < 15 the error is bounded by 0.000058444302 (n = 4) and 800.108 (n = 15).

1.e In this case the lowest degree polynomial (n = 1) is not the best. The best polynomial for 'exp(x)' is when (n = 4). For this case lower degree polynomials are better when their degree is near 4.

```
fprintf('Jacobi\n')
for i=1:1:15,
    A=hilb(i+1);
    N=diag(diag(A));
    P=N-A;
    normv = norm(inv(N)*P);
    fprintf('n = %d\t norm = %0.12f\n', i, normv)

end
```

```
Jacobi
n = 1  norm = 1.500000000000
n = 2  norm = 2.502477790831
n = 3  norm = 3.580891042607
n = 4  norm = 4.673829413484
n = 5  norm = 5.771337441841
n = 6  norm = 6.870691726736
n = 7  norm = 7.970907326174
n = 8  norm = 9.071562772738
n = 9  norm = 10.172456330091
n = 10  norm = 11.273483462947
n = 11  norm = 12.374586758344
n = 12  norm = 13.475733289046
n = 13  norm = 14.576903557895
n = 14  norm = 15.678085756873
n = 15  norm = 16.779272634228
```

1.f No, you cannot use Jacobi iteration because the norm of a hilbert matrix is always greater than zero. For the Jacobi method (which is basically fixed point iteration) to work, g' or M needs to be less than 1.

```
fprintf('gauss-seidel\n')
for i=1:1:15,
    A=hilb(i+1);
    N=tril(A);
    P=N-A;
    normv = norm(inv(N)*P);
    fprintf('n = %d\t norm = %0.12f\n', i, normv);
end
```

```
gauss-seidel
n = 1  norm = 0.901387818866
n = 2  norm = 0.999000979633
n = 3  norm = 1.000105673292
n = 4  norm = 1.000291468939
n = 5  norm = 1.000512238416
n = 6  norm = 1.000739526346
n = 7  norm = 1.000959067810
n = 8  norm = 1.001164685021
n = 9  norm = 1.001354337869
n = 10  norm = 1.001527991969
n = 11  norm = 1.001686524849
n = 12  norm = 1.001831175615
n = 13  norm = 1.001963273164
n = 14  norm = 1.002084106704
n = 15  norm = 1.002194868940
```

1.g Yes, but only for matrixes of size n=1 and n=2. All other matricies have norms larger than one.

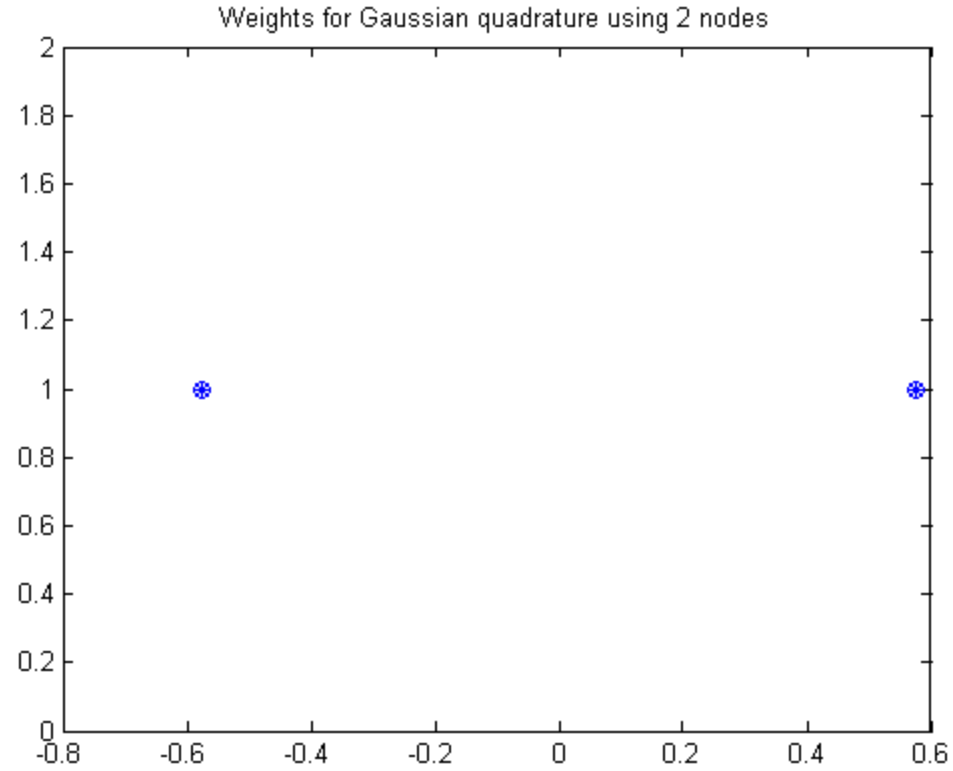
Problem 2

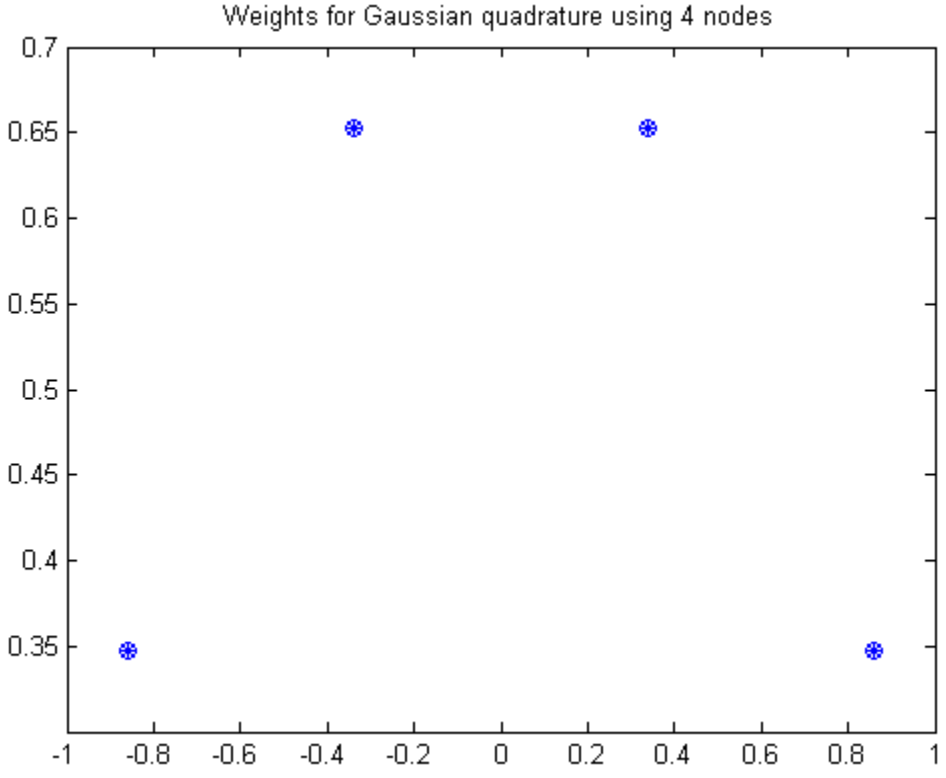
```
Ns = [2, 4, 8, 32, 64];
for i=1:length(Ns),
    n = Ns(i);
    [compweights, A] = gaussweights(n);
    N_GS=tril(A);
    P_GS=N_GS-A;
    GS_norm = norm(inv(N_GS)*P_GS);

    N_J=diag(diag(A));
    P_J=N_J-A;
    J_norm = norm(inv(N_J)*P_J);
    fprintf(['==============\nn = %d\t\ncondition number = %0.5e\nJacobi' ...
     'norm = %0.5e\nGauss-seidel norm = %0.5e'], n, cond(A, inf), J_norm, GS_norm
    compweights
    fprintf('\n')
end
```

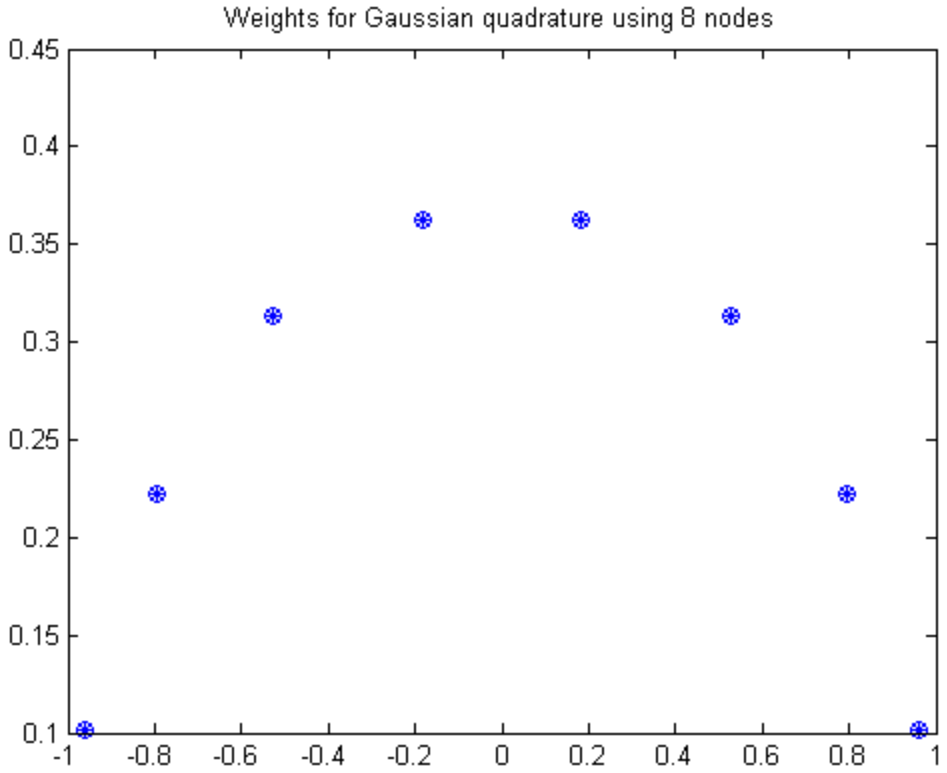
```
==============
n = 2
condition number = 2.73205e+00
Jacobinorm = 1.00000e+00
Gauss-seidel norm = 1.41421e+00
compweights =

     1     1


==============
n = 4
condition number = 2.19310e+01
Jacobinorm = 9.19218e+00
Gauss-seidel norm = 7.31311e+00
compweights =

    0.3479    0.6521    0.6521    0.3479


==============
n = 8
condition number = 7.98958e+02
Jacobinorm = 1.17920e+03
Gauss-seidel norm = 9.63263e+02
compweights =

  Columns 1 through 7

    0.1012    0.2224    0.3137    0.3627    0.3627    0.3137    0.2224

  Column 8
```

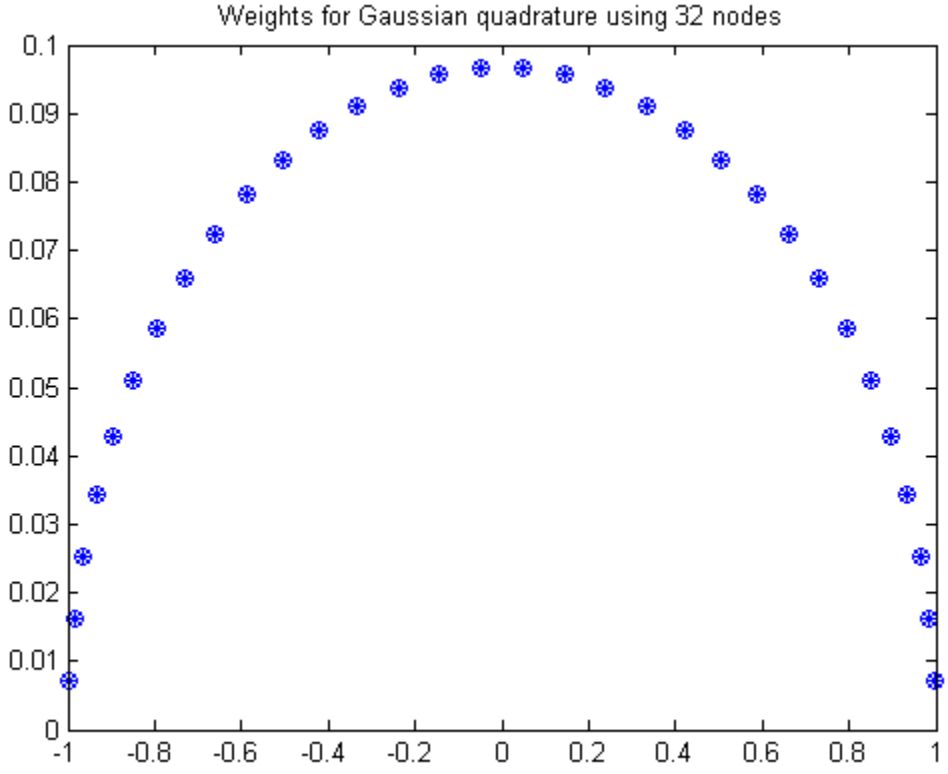
```
    0.1012


Warning: Matrix is close to singular or badly scaled. Results may be inacc
RCOND =  2.866894e-23.
Warning: Matrix is close to singular or badly scaled. Results may be inacc
RCOND =  8.795425e-22.
==============
n = 32
condition number = 1.28267e+12
Jacobinorm = 2.28212e+21
Gauss-seidel norm = 2.20770e+21
compweights =

  Columns 1 through 7

    0.0070    0.0163    0.0254    0.0343    0.0428    0.0510    0.0587

  Columns 8 through 14

    0.0658    0.0723    0.0782    0.0833    0.0877    0.0912    0.0938

  Columns 15 through 21

    0.0956    0.0965    0.0965    0.0956    0.0938    0.0912    0.0877

  Columns 22 through 28

    0.0833    0.0782    0.0723    0.0658    0.0587    0.0510    0.0428

  Columns 29 through 32

    0.0343    0.0254    0.0163    0.0070


Warning: Matrix is close to singular or badly scaled. Results may be inacc
RCOND =  3.727595e-54.
Warning: Matrix is close to singular or badly scaled. Results may be inacc
RCOND =  2.334178e-52.
Warning: Matrix is close to singular or badly scaled. Results may be inacc
RCOND =  7.132669e-21.
==============
n = 64
condition number = 1.23383e+20
Jacobinorm = 1.04013e+52
Gauss-seidel norm = 1.12629e+52
compweights =

  Columns 1 through 7

    0.0572   -0.1622    0.2423   -0.1803   -0.0105    0.4171   -0.8978

  Columns 8 through 14
```

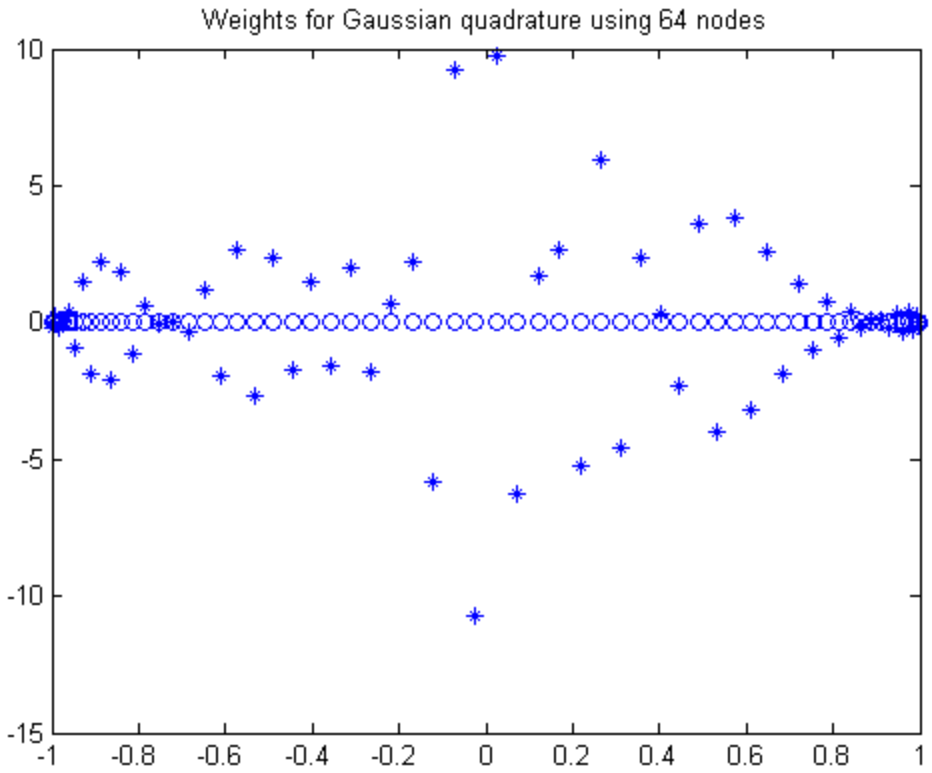
```
  1.4791   -1.9049    2.2038   -2.1179    1.8182   -1.1832    0.6059

Columns 15 through 21

 -0.0786    0.0529   -0.3540    1.1675   -1.9366    2.6211   -2.6597

Columns 22 through 28

  2.3834   -1.7498    1.5161   -1.5566    2.0070   -1.8302    0.6451

Columns 29 through 35

  2.2147   -5.8144    9.2520  -10.7129    9.7656   -6.2526    1.7344

Columns 36 through 42

  2.6331   -5.2237    5.9092   -4.5707    2.3355    0.2944   -2.2814

Columns 43 through 49

  3.6344   -3.9850    3.8568   -3.1967    2.5699   -1.8479    1.4037

Columns 50 through 56

 -0.9705    0.7783   -0.5307    0.4218   -0.2095    0.0916    0.1091

Columns 57 through 63

 -0.2072    0.3386   -0.3575    0.3771   -0.3045    0.2334   -0.1227

Column 64

  0.0399
```


Weights for Gaussian quadrature using 2 nodes
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6


Weights for Gaussian quadrature using 4 nodes
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1


Weights for Gaussian quadrature using 8 nodes
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1


Weights for Gaussian quadrature using 32 nodes
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1



2.a The values for the weights in the book and the weights generated in the table are exactly the same for up to three digits of accuracy.

2.b The condition values for n = 32 and n = 64 are relativly very large compared to when n is equal to 8, 4, and 2. Specifically, when n = 32 the condition number is 1282671494874.59 and when n = 64 K is 48500196769728348160. The high condition numbers mean the matrixs are ill-conditioned which mean there will be a lot of error when solving them.

2.c and 2.d Jacobi and Gauss-Seidel can be applied but only when the norm of the matrix is strictly less than 1.

Problem 3

To reduce the amount of code in my lab, I modified cubspline.m to accept a new argument 'sol_method' that chose which method to use when solving the system

```
%   +  switch sol_method
%   +    case 'GEpivot'
%   +        [M,lu,piv] = GEpivot(A,b);
%   +    case 'Jacobi'
%   +        [M, iflag, itnum] = Jacobi(A,b,b,eps,100000);
%   +    case 'GS'
%   +        [M, iflag, itnum] = GS(A,b,b,eps,100000);
%   +    case 'CG'
%   +        [M, iflag, itnum] = CG(A,b,b,eps,100000);
%   +    case 'triadag'
```

```
    %   +        [M, alpha, beta, ier] = tridiag(ld,di,ud,b,n-2,0);
    %   +  end
    %   +
```

```
%Ns = [8, 16, 64, 128, 256];
Ns = [64, 128, 256];
methods = {'GEpivot', 'Jacobi', 'GS', 'CG', 'triadag'};
for m=1:length(methods),
    method = methods{m};

    sprintf(['Using *%s* to solve tridiagonal system corresponding to' ...
        'natural cubic spline interpolation\n'], method)
    for i=1:length(Ns),
        n = Ns(i);
        time_e = 0;
        for j=1:50,
            [res,tme,itnum] = cubspline(n, method, 0);
            time_e = time_e + tme;
        end
        fprintf(['n = %d\titerations = %d\t Average run time over 50' ...
            'runs:%0.5e\t\n'], n, itnum, time_e/50.0);
    end
end
```

```
ans =

Using *GEpivot* to solve tridiagonal system corresponding tonatural cubic


n = 64 iterations = 1  Average run time over 50runs:8.08721e-03
n = 128 iterations = 1  Average run time over 50runs:4.73153e-02
n = 256 iterations = 1  Average run time over 50runs:1.83243e-01

ans =

Using *Jacobi* to solve tridiagonal system corresponding tonatural cubic s


n = 64 iterations = 45  Average run time over 50runs:2.61510e-04
n = 128 iterations = 45  Average run time over 50runs:4.64220e-04
n = 256 iterations = 46  Average run time over 50runs:8.89617e-04

ans =

Using *GS* to solve tridiagonal system corresponding tonatural cubic splin


n = 64 iterations = 31  Average run time over 50runs:1.28325e-02
n = 128 iterations = 31  Average run time over 50runs:3.24036e-02
n = 256 iterations = 31  Average run time over 50runs:7.39473e-02

ans =
```

```
Using *CG* to solve tridiagonal system corresponding tonatural cubic splin


n = 64 iterations = 35  Average run time over 50runs:1.47416e-03
n = 128 iterations = 35  Average run time over 50runs:2.06292e-03
n = 256 iterations = 35  Average run time over 50runs:3.06037e-03

ans =

Using *triadag* to solve tridiagonal system corresponding tonatural cubic


n = 64 iterations = 1  Average run time over 50runs:1.65220e-04
n = 128 iterations = 1  Average run time over 50runs:1.64924e-04
n = 256 iterations = 1  Average run time over 50runs:3.21796e-04
```

3.a See printed table

3.b In time, Jacobi preformed much better than Gaussian Elimination with Pivoting. When n was 256 Jocobi took roughly 0.00129 seconds and GEpivot took 0.1814 seconds. Jacobi beat GEpivot when n was 26 and 128 too. Jacobi usually took 45 iterations and GEpivot took 1 iteration.

3.c GS used 31 iterations and Jacobi used 45 iterations. GS is better because it uses calculated Jacobi values which accelerates its convergence.

3.d GC uses less iterations than Jacobi and has a faster average time of completing when n is 64 and 128. When n is 256 Jacobi actually solves it's system faster than GC does.

3.e Triadag is faster than all other methods and only takes 1 iteration (one round of forward and back substitution).

Published with MATLAB® 8.0