MATH 232 Homework 4

1. **Problem 1**

SOLUTION:

(a) Solving $u''(x) = e^x$ with $u(0) = 3, u'(1) = -5$, we can simply integrate 2 times.

$$\int u''(x)dx = \int e^x dx \implies u'(x) = e^x + C \implies u'(1) = -5 = e + C \therefore C = -5 - e$$

Integrating again yields:

$$\int u'(x)dx = \int e^x - (5+e)dx = e^x - (5+e)x + C \implies 3 = u(0) = e^0 - 0 + C \therefore C = 2$$

Hence, we see that $u(x) = e^x - (5+e)x + 2$.

Preface: All parts (b)-(f) can have their results attained by running the file `hw04.m`, the individual questions are done in separate files as detailed below.

(b) Using the standard second-order finite difference method, we solve the given BVP using the script `solve_p1a.m` found in the appendix/attached files. Observing Figure 1, we see that the line $e1$ is the error produced by running the script `hw04.m`. We verify that it is parallel to a line with slope -2 in the loglog plot, hence verifying that it is a second order method. In the approximated solution plot, it corresponds to u_{2nd}, and we see that it does a very good job at approximating the true solution, with slightly more apparent error towards the end of the interval, but still a good approximation nonetheless.

(c) In `hw04.m`, we perform a Richardson's extrapolation using the script `solve_p1a.m` from part (b). This is denoted in the `ur` assignment statement. We plot the errors of this method in the loglog plot and denote it er. We verify that it is parallel to a line with slope -4 in the loglog plot, verifying that it is a fourth order method. In the approximated solution plot, it corresponds to u_{ext}, and we see it does an extremely accurate job at approximating the true solution.

(d) In the script `solve_p1d.m` found in the appendix/attached files, we implement the deferred correction solution as presented in lecture. We see in Figure 1 that the errors in the loglog plot, denoted ed are parallel to a line with slope -4, verifying that the deferred correction method is fourth order accurate. We also find that in Figure 2, that the true solution is approximated very well by this method.

(e) In the script `solve_p1e.m`, we implement a fourth order finite difference method. In Figure 1, we see the the error produced in the loglog plot is given by $e4$, and that this line in general is parallel to a line with slope -4, verifying that our method is fourth order accurate. We see in Figure 2 that the solution given by u_4 is extremely accurate at approximating the true solution.

(f) Using the script `solve_p1f.m` found in the appendix, we implement the spectral collocation method using a Chebyshev grid. In the original rendition of the code, provided by `SpectralMethod1.m`, we see that if we increase the number of nodes that the method eventually does not produce accurate solutions, namely when increased to past node count of 65 it does not approximate the true solution correctly. Using the modified Chebyshev grid, we see that the errors are produced in the loglog plot by es. Again, we see in Figure 2 that this method is very accurate at approximating the true solution, with errors being still magnitudes better than the previous methods (all $< 10^{-10}$), despite it increasing with the amount of nodes used. In Figure 2, we see that the solution curve given by u_s approximates the true solution very accurately.

Commentary:
In terms of accuracy, we clearly see that using the spectral method on a Chebyshev grid results in the best accuracy. Additionally, the spectral method computes the solution with the best efficiency as well. With only 10 grid points, the error is less than 1e-12. For this particular problem, the spectral method does exceptionally well. However, this may not be the case on more complicated geometries.

The second-order finite difference scheme performs as expected (second order) and has a very simple implementation. While it is not the most accurate of the methods tested, it is by far the simplest to implement. Furthermore, this method can be augmented by using Richardson extrapolation or differed correction to yield a 4th order scheme. Both of these augmentation techniques are also fairly simple to implement (after you know the process).

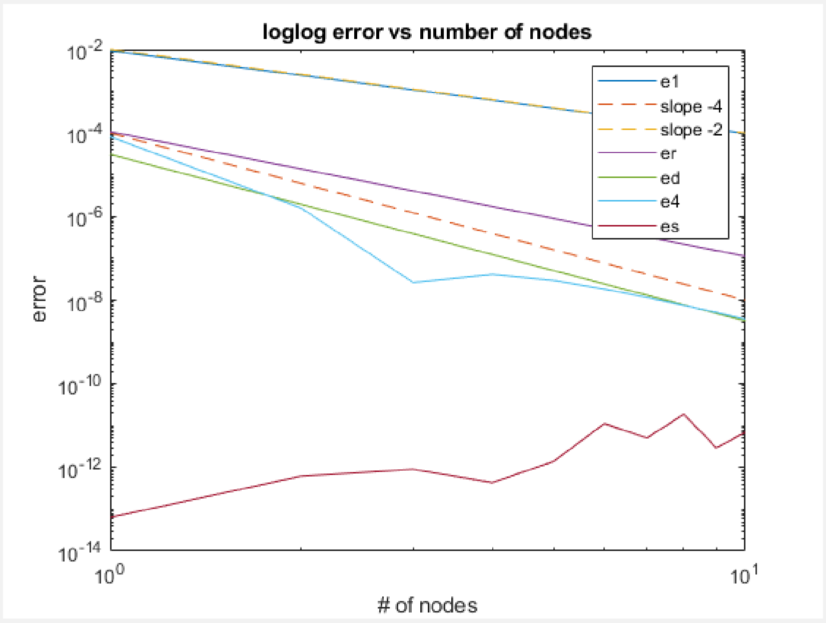

Figure 1: loglog plot of the error vs the number of nodes used per numerical method.

Finally, we see that the standard 4th order scheme performs as expected with 4th order accuracy. By performing some additional tests (using the `tic` and `toc` commands in `MATLAB`), we can also see that the 4th order method is faster in implementation than Richardson extrapolation, but not the deferred correction scheme. Though, we should note that implementing the 4th order scheme is more straight forward and the standard coefficients are readily available online, whereas the deferred correction implementation needs to be computed by completing the truncation error analysis.

Overall, we see that the Spectral Method performs the best for this particular problem. It is straightforward to implement and is the most computationally efficient. However, we do acknowledge that may not be the case for other boundary conditions. As a second choice (for most preferred method), the 2nd order scheme using the deferred correction augmentation should be considered. By including one or two additional terms in the computation of our right hand side, we can increase the order of our method by 2. Though, this may not be the most straightforward process if our function isn't as nice as e^x.

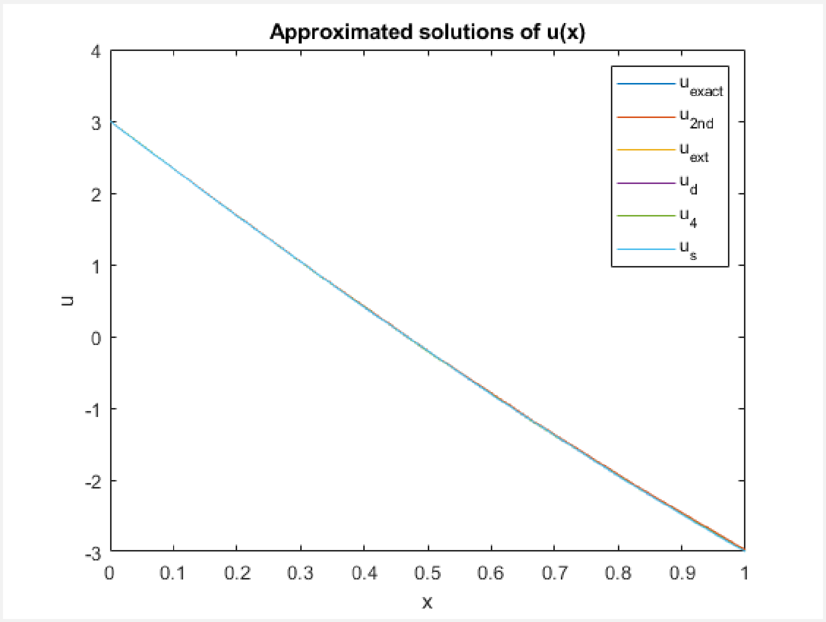

Figure 2: Approximated solutions and true solution.

Contributions:

Each group member completed the assignment individually, then the team we met together to discuss the answers complete the write up in Latex.

APPENDIX:

hw04.m

```
close all; clear; clc;

% define grid
x0 = 0;
xf = 1;
m = 6;

% define init conditions
u0 = 3;
du1 = -5;
rhs_fun = @(x) exp(x);

% solve
u1 = solve_p1a(u0,du1,rhs_fun,x0,xf,m);
u2 = solve_p1a(u0,du1,rhs_fun,x0,xf,2*m);
ur = (1/3)*(4*u2(1:2:end)-u1);
ud = solve_p1d(u0,du1,rhs_fun,x0,xf,m);
u4 = solve_p1e(u0,du1,rhs_fun,x0,xf,m);
[us,xs] = solve_p1f(u0,du1,rhs_fun,x0,xf,m);

% compute exact solution
u_exact = @(t) exp(t) - (5+exp(1))*t + 2;

% plot sols
plot(linspace(x0,xf,m),u_exact(linspace(x0,xf,m)))
hold on
plot(linspace(x0,xf,length(u1)),u1)
plot(linspace(x0,xf,length(ur)),ur)
plot(linspace(x0,xf,length(ud)),ud)
plot(linspace(x0,xf,length(u4)),u4)
plot(xs,us)
xlabel 'x'; ylabel 'u'
title 'Approximated solutions of u(x)'
legend 'u_{exact}' 'u_{2nd}' 'u_{ext}' 'u_d' 'u_4' 'u_s'

% plot errors
for j = 1:10
    m = j*10;

    u1 = solve_p1a(u0,du1,rhs_fun,x0,xf,m);
    u2 = solve_p1a(u0,du1,rhs_fun,x0,xf,2*m);
    ur = (1/3)*(4*u2(1:2:end)-u1);
    ud = solve_p1d(u0,du1,rhs_fun,x0,xf,m);
    u4 = solve_p1e(u0,du1,rhs_fun,x0,xf,m);
    [us,xs] = solve_p1f(u0,du1,rhs_fun,x0,xf,m);

    e1(j) = norm(u1(end)-u_exact(xf));
    er(j) = norm(ur(end)-u_exact(xf));
    ed(j) = norm(ud(end)-u_exact(xf));
    e4(j) = norm(u4(end)-u_exact(xf));
    es(j) = norm(us(end)-u_exact(xf));
end
```

```
ms = [10 20 30 40 50 60 70 80 90 100];
figure
loglog(e1)
hold on
sl4 = @(x) x.^(-4);
ys4 = sl4(ms);
loglog(ys4, '--')
sl2 = @(x) x.^(-2);
ys2 = sl2(ms);
loglog(ys2, '--')
loglog(er)
loglog(ed)
loglog(e4)
loglog(es)
xlabel '# of nodes'
ylabel 'error'
title 'loglog error vs number of nodes'
legend 'e1' 'slope -4' 'slope -2' 'er' 'ed' 'e4' 'es'
```

```
solve_p1a.m
function [u] = solve_p1a(u0,du1,rhs_fun,x0,xf,m)
h = (xf-x0)/(m);

F = zeros(m+1,1);

A = spdiags([ones(m+1,1) -2*ones(m+1,1) ones(m+1,1)], -1:1, m+1, m+1);
A(1,:) = 0;
A(1,1) = h^2;
A(m+1,:) = 0;
A(m+1,m-1) = h/2;
A(m+1,m) = -2*h;
A(m+1,m+1) = 3*h/2;

A = A/h^2;

for i = 2:m
    F(i) = rhs_fun((i-1)*h+x0);
end
F(1) = u0;
F(m+1) = du1;
u = A\F;

end
```

```
solve_p1d.m
function [u] = solve_p1d(u0,du1,rhs_fun,x0,xf,m)

h = (xf-x0)/(m);

F = zeros(m+1,1);

A = spdiags([ones(m+1,1) -2*ones(m+1,1) ones(m+1,1)], -1:1, m+1, m+1);
A(1,:) = 0;
A(1,1) = h^2;
A(m+1,:) = 0;
A(m+1,m-1) = h/2;
A(m+1,m) = -2*h;
A(m+1,m+1) = 3*h/2;

A = A/h^2;

for i = 2:m
    F(i) = rhs_fun((i-1)*h+x0)+h^2*rhs_fun((i-1)*h+x0)/12;
end
F(1) = u0;
F(m+1) = du1 - h^2*rhs_fun(xf)/3 + h^3*rhs_fun(xf)/4;
u = A\F;

end
```

```
solve_p1e.m
function [u] = solve_p1e(u0,du1,rhs_fun,x0,xf,m)

h = (xf-x0)/(m);

F = zeros(m+1,1);

A = spdiags([(-1/12)*ones(m+1,1) (4/3)*ones(m+1,1) (-5/2)*ones(m+1,1)
    (4/3)*ones(m+1,1) (-1/12)*ones(m+1,1)], -2:2, m+1, m+1);
A(1,:) = 0;
A(1,1) = h^2;

A(2,:) = 0;
A(2,2) = 15/4;
A(2,3) = -77/6;
A(2,4) = 107/6;
A(2,5) = -13;
A(2,6) = 61/12;
A(2,7) = -5/6;

A(m,:) = 0;
A(m,m) = 15/4;
A(m,m-1) = -77/6;
A(m,m-2) = 107/6;
A(m,m-3) = -13;
A(m,m-4) = 61/12;
A(m,m-5) = -5/6;

A(m+1,:) = 0;
A(m+1,m+1) = 25*h/12;
A(m+1,m) = -4*h;
A(m+1,m-1) = 3*h;
A(m+1,m-2) = -4*h/3;
A(m+1,m-3) = h/4;

A = A/h^2;

for i = 2:m
    F(i) = rhs_fun((i-1)*h+x0);
end
F(1) = u0;
F(m+1) = du1;
u = A\F;

end
```

```
solve_p1f.m
function [u,x] = solve_p1f(u0,du1,rhs_fun,x0,xf,m)

% construct grid
h = 1/(m+1);
x = zeros(m+2,1);

for i = 1 : m+2
    %x(i) = .5*(1+cos(pi*(1-(i-1)*h)));
    x(i) = x0 + 0.5*(xf-x0)*(1+cos(pi*(1-(i-1)*h)));
end

% construct matrix
A = zeros(m+2);

A(1,1) = 1;

for j = 2 : m + 1
    A(j,:) = fdcoeffF( 2, x(j), x );
end

A(m+2,:) = fdcoeffF( 1, x(m+2), x );

% compute rhs
F = zeros(m+2,1);
F(1) = u0;

for i = 2:m+1
    F(i) = rhs_fun(x(i));
end

F(m+2) = du1;

u = A\F;

end
```