MATH 517 Finite Differences Homework 7

Caleb Logemann

April 21, 2016

1. (a) Implement the alternating direction implicit (ADI) scheme for this problem. Use the backslash operator in MATLAB to invert the necessary matrices.

```
function [u, Ux, Uy, k] = ADI(N, T, f, g)
    h = 1/(N+1);
    x = @(i) i*h;
    y = @(j) j*h;

    % find k such that k = O(h)
    Nt = ceil(T/h);
    k = T/Nt;

    t = @(n) n*k;

    % create functions to swap between column-wise ordering and i,j ordering
    %kFun = @(i, j) i + (j-1)*N;
    iFun = @(k) floor((k-1)/N) + 1;
    jFun = @(k) mod(k-1,N)+1;

    % permutation matrix to change from natural row wise ordering to natural
    % column wise ordering or vice versa
    % uCol = P*uRow or uRow = P*uCol
    i = 1:N^2;
    j = repmat(0:N:N^2-N, 1, N) + kron(1:N,ones(1, N));
    P = sparse(i, j, ones(1, N^2));

    % matrix to represent diffusion in one direction
    % D*uRow = Dx2*U and D*uCol = Dy2*U
    e = ones(N, 1);
    tridiagonal = 1/h^2*spdiags([e, -2*e, e], [-1, 0, 1], N, N);
    D = kron(speye(N), tridiagonal);

    % N^2 by N^2 identity matrix
    I = speye(N^2);

    % initiate matrix to store u at times 1:Nt
    u = zeros(N^2, Nt+1);
    % create initial conditions
    % vector of x-values for column-wise ordering k = 1:N^2
    Ux = x(iFun(1:N^2));
    % vector of y-values for column-wise ordering k = 1:N^2
    Uy = y(jFun(1:N^2));
    % find initial values in column-wise ordering
    u(:, 1) = f(Ux, Uy);

    % create index arrays for boundary
    % k-indices for bottom boundary in column-wise ordering or
    % left boundary in row-wise ordering
    % ie when either x or y is zero
    zeroIndices = 1:N:N^2;
    % k-indices for top boundary in column-wise ordering or
    % right boundary in row-wise ordering
    % ie when either x or y is one
    oneIndices = N:N:N^2;
    for n = 1:1
        % u starts in column-wise ordering
        % First stage
        % (I + k/2 Dy2)*u
        rhs = (I + k/2*D)*u(:, n);
        % add boundary conditions for y-direction at time t = tn
```

```
        bottomBoundary = k/(2*h^2)*g(t(n-1), x(1:N), 0);
        topBoundary = k/(2*h^2)*g(t(n-1), x(1:N), 1);
        rhs(zeroIndices) = rhs(zeroIndices) + bottomBoundary';
        rhs(oneIndices) = rhs(oneIndices) + topBoundary';
        % change to row-wise ordering
        rhs = P*rhs;
        % add boundary conditions for x-direction at time t = tn + k/2
        leftBoundary = k/(2*h^2)*g(t(n-1)+k/2, 0, y(1:N));
        rightBoundary = k/(2*h^2)*g(t(n-1)+k/2, 1, y(1:N));
        rhs(zeroIndices) = rhs(zeroIndices) + leftBoundary';
        rhs(oneIndices) = rhs(oneIndices) + rightBoundary';
        % solve for uStar which approximates u at t = tn + k/2
        % uStar is in row-wise ordering
        tic
        uStar = (I - k/2*D)\rhs;

        % second stage
        % rhs row-wise ordering
        rhs = (I + k/2*D)*uStar;
        % add boundary conditions for x-direction at time t = tn + k/2
        % left and right boundaries same as before
        rhs(zeroIndices) = rhs(zeroIndices) + leftBoundary';
        rhs(oneIndices) = rhs(oneIndices) + rightBoundary';
        % change to column-wise ordering
        rhs = P*rhs;
        % add boundary conditions for y-direction at time t = tn + k
        bottomBoundary = k/(2*h^2)*g(t(n-1)+k, x(1:N), 0);
        topBoundary = k/(2*h^2)*g(t(n-1)+k, x(1:N), 1);
        rhs(zeroIndices) = rhs(zeroIndices) + bottomBoundary';
        rhs(oneIndices) = rhs(oneIndices) + topBoundary';
        % solve for u at time t = tn + k
        % u is in column wise ordering
        u(:,n+1) = (I - k/2*D)\rhs;
        toc
    end
end
```

(b) Do a convergence study of your method for the following exact solution:

$$u(t,x,y) = e^{-32\pi^2 t}\cos(4\pi x)\cos(4\pi y)$$

```
%% Problem 1
    uExact = @(t, x, y) exp(-32*pi^2*t)*cos(4*pi*x).*cos(4*pi*y);
    f = @(x, y) uExact(0, x, y);
    T = 1;
    E = [];
    H = [];
    for N = 10*2.^(1:5) - 1
        [u, Ux, Uy, k] = ADI(N, 1, f, uExact);
        % create exact solution
        t = @(n) n*k;
        uExactMatrix = cell2mat(arrayfun(@(n) uExact(t(n), Ux, Uy)', 0:T/k, ...
            'UniformOutput', false));
        H = [H; Uy(2) - Uy(1)];
        E = [E; norm(u(:,end) - uExactMatrix(:,end), inf)];
    end
    hRatios = H(1:end-1)./H(2:end);
    errorRatios = E(1:end-1)./E(2:end);
    order = log(errorRatios)./log(hRatios);
    table(hRatios, errorRatios, order)
```

```
ans =

    hRatios    errorRatios    order
    -------    -----------    ------

    2            8.132        3.0236
    2           9.4968        3.2474
    2           12.937        3.6935
    2           15.027        3.9095
```

(c) For the problem in (b), put a tic command immediately before you solve the first tridiagonal system, and a toc command immediately after the second tridiagonal solve. Create a table of run times for various N in a single timestep of your solver. Comment on your results.

N	time (sec)
19	0.002496
39	0.002069
79	0.005204
159	0.011467
319	0.054771
639	0.221379

The time for one iteration seems to be growing with N^2, or at the very least it is growing faster than N is growing. When N doubles the time for one iteration increases by more than 2 times probably closer to 4 times. I believe that the time is growing by N^2 because the size of the system being solved is N^2, because we are in 2 dimensions. This seems to indicate that this method will not scale well to large problems.

2. Third Order Lax-Wendroff

(a) Construct a third order accurate Lax-Wendroff-type method for the advection equation.
First I will expand $u(t+k,x)$ using a Taylor series.

$$u(t+k,x) = u(t,x) + ku_t(t,x) + \frac{k^2}{2}u_{tt}(t,x) + \frac{k^3}{6}u_{ttt}(t,x) + O(k^4)$$

The advection equation states that $u_t = -au_x$, therefore it can also be states that $u_{tt} = a^2u_{xx}$ and $u_{ttt} = -a^3u_{xxx}$. Thus the Taylor expansion for the advection equation becomes

$$u(t+k,x) = u(t,x) - aku_x + \frac{(ak)^2}{2}u_{xx} - \frac{(ak)^3}{6}u_{xxx} + O(k^4)$$

In order to approximate the spacial derivatives, I will create a cubic polynomial that interpolates U^n_{j-2}, U^n_{j-1}, U^n_j and U^n_{j+1}. I will express this polynomial in the form

$$p(x) = a(x-x_j)^3 + b(x-x_j)^2 + c(x-x_j) + d.$$

This form will make finding the derivatives, $u_x(t^n,x_i)$, $u_{xx}(t^n,x_i)$, and $u_{xxx}(t^n,x_i)$, easier in the future. In order to find the coefficients a, b, c, and d, the following four equations must be solved.

$$\begin{aligned}
p(x_j - 2h) &= a(-2h)^3 + b(-2h)^2 + c(-2h) + d = U_{j-2}^n \\
p(x_j - h) &= a(-h)^3 + b(-h)^2 + c(-h) + d = U_{j-1}^n \\
p(x_j) &= d = U_j^n \\
p(x_j + h) &= a(h)^3 + b(h)^2 + c(h) + d = U_{j+1}^n
\end{aligned}$$

After solving these equations in Mathematica, I found the coefficients to be

$$\begin{aligned}
a &= -\frac{U_{j-2}^n - 3U_{j-1}^n + 3U_j^n - U_{j+1}^n}{6h^3} \\
b &= \frac{U_{j-1}^n - 2U_j^n + U_{j+1}^n}{2h^2} \\
c &= \frac{U_{j-2}^n - 6U_{j-1}^n + 3U_j^n + 2U_{j+1}^n}{6h} \\
d &= U_j^n
\end{aligned}$$

Now note that $u(t^n, x) \approx p(x)$ when $x \in [x - 2h, x + h]$. Therefore

$$\begin{aligned}
u_x(t^n, x_j) &\approx p'(x_j) = c \\
u_{xx}(t^n, x_j) &\approx p''(x_j) = 2b \\
u_{xxx}(t^n, x_j) &\approx p'''(x_j) = 6a
\end{aligned}$$

Now substituting into the Taylor expansion to actually create a numerical method we get

$$\begin{aligned}
U_j^{n+1} = U_j^n &- \frac{ak}{6h}\Big(U_{j-2}^n - 6U_{j-1}^n + 3U_j^n + 2U_{j+1}^n\Big) \\
&+ \frac{(ak)^2}{2h^2}\Big(U_{j-1}^n - 2U_j^n + U_{j+1}^n\Big) \\
&+ \frac{(ak)^3}{6h^3}\Big(U_{j-2}^n - 3U_{j-1}^n + 3U_j^n - U_{j+1}^n\Big)
\end{aligned}$$

(b) Verify that the truncation error is $O(k^3)$ if $h = O(k)$.

3. Third-Order Method of Lines with RK3:
   The following method is a third order accurate Runge-Kutta method for $u' = f(u)$:

$$U^{n+1} = U^n + \frac{k}{9}(2Y_1 + 3Y_2 + 4Y_3)$$

$$Y_1 = f(U^n), \quad Y_2 = f\left(U^n + \frac{k}{2}Y_1\right), \quad Y_3 = f\left(U^n + \frac{3k}{4}Y_2\right)$$

   (a) Construct a third order accurate method for the advection equation.
   In order to approximate the first derivative, u_x I will use the interpolation polynomial that I found in problem 2:

$$p(x) = a(x - x_j)^3 + b(x - x_j)^2 + c(x - x_j) + d.$$

where

$$
\begin{aligned}
a &= -\frac{U^n_{j-2} - 3U^n_{j-1} + 3U^n_j - U^n_{j+1}}{6h^3} \\
b &= \frac{U^n_{j-1} - 2U^n_j + U^n_{j+1}}{2h^2} \\
c &= \frac{U^n_{j-2} - 6U^n_{j-1} + 3U^n_j + 2U^n_{j+1}}{6h} \\
d &= U^n_j.
\end{aligned}
$$

Originally the advection equation states, on the spacially discretized system, that

$$U_j'(t) = a\frac{\mathrm{d}}{\mathrm{d}x}U_j(t)$$

However we can now replace $\frac{\mathrm{d}}{\mathrm{d}x}U_j(t)$ with $\frac{\mathrm{d}}{\mathrm{d}x}p(x)$ at $x = x_j$, which is

$$\frac{U^n_{j-2} - 6U^n_{j-1} + 3U^n_j + 2U^n_{j+1}}{6h}.$$

Now for each j, there is an ODE of the form

$$U_j'(t) = \frac{a}{6h}\left(U^n_{j-2} - 6U^n_{j-1} + 3U^n_j + 2U^n_{j+1}\right)$$

Now the RK3 method can be applied where

$$f(U^n) = \frac{a}{6h}\left(U^n_{j-2} - 6U^n_{j-1} + 3U^n_j + 2U^n_{j+1}\right)$$

(b) Verify that the truncation error is $O(k^3)$ if $k = O(h)$

(c)

4. First I will implement the Lax-Wendroff method.

```
function [u, h, k] = LaxWendroff3(a, N, T, f)
    % discretize space
    h = 1/(N+1);
    x = @(i) i*h;

    % discretize time
    % find k such that k = O(h), but not exact k = h
    Nt = ceil(T/(.9*h));
    k = T/Nt;
    t = @(n) n*k;

    % initiate matrix to store u at times 1:Nt
    u = zeros(N+2, Nt+1);
    % add initial conditions
    u(:,1) = f(x(0:N+1));

    for n=1:Nt
        % create U^n_{j-2}, U^n_{j-1}, and U^n_{j+1} considering periodic
        % boundary conditions
        Uj2m = [u(end-1,n); u(end, n); u(1:end-2,n)];
        Uj1m = [u(end, n); u(1:end-1,n)];
        Uj1p = [u(2:end,n); u(1,n)];
        u(:,n+1) = u(:,n) - (a*k)/(6*h)*(Uj2m - 6*Uj1m + 3*u(:,n) + 2*Uj1p)...
            + (a*k)^2/(2*h^2)*(Uj1m - 2*u(:,n) + Uj1p)...
            + (a*k)^3/(6*h^3)*(Uj2m - 3*Uj1m + 3*u(:,n) - Uj1p);
    end
end
```

Testing both initial conditions

```
%% Problem 4 for LaxWendroff3 created in Problem 2
    a = 1;
    T = 1;
    uExact = @(t, x) 2*exp(-200*(mod((x-a*t),1) - 1/2).^2);
    f = @(x) uExact(0,x);
    E = [];
    H = [];
    for N = 10*2.^(1:8) - 1
        [u, h, k] = LaxWendroff3(a, N, T, f);
        % create exact solution
        t = @(n) n*k;
        x = @(i) i*h;
        uExactMatrix = cell2mat(arrayfun(@(n) uExact(t(n), x(0:N+1))', 0:T/k, ...
            'UniformOutput', false));
        H = [H; h];
        E = [E; norm(u(:,end) - uExactMatrix(:,end), inf)];
    end
    hRatios = H(1:end-1)./H(2:end);
    errorRatios = E(1:end-1)./E(2:end);
    order = log(errorRatios)./log(hRatios);
    table(hRatios, errorRatios, order)

    uExact = @(t, x) 2*exp(-200*(mod((x-a*t),1) - 1/2).^2).*cos(40*pi*mod((x-a*t),1));
    f = @(x) uExact(0,x);
    E = [];
    H = [];
    for N = 10*2.^(1:8) - 1
        [u, h, k] = LaxWendroff3(a, N, T, f);
        % create exact solution
        t = @(n) n*k;
        x = @(i) i*h;
        uExactMatrix = cell2mat(arrayfun(@(n) uExact(t(n), x(0:N+1))', 0:T/k, ...
            'UniformOutput', false));
        H = [H; h];
        E = [E; norm(u(:,end) - uExactMatrix(:,end), inf)];
    end
    hRatios = H(1:end-1)./H(2:end);
    errorRatios = E(1:end-1)./E(2:end);
    order = log(errorRatios)./log(hRatios);
    table(hRatios, errorRatios, order)
```

```
ans =

    hRatios    errorRatios     order
    -------    -----------    -------

    2          1.5373         0.62036
    2          1.9039         0.92894
    2          1.9901          0.9928
    2          1.9993         0.99951
    2               2         0.99999
    2               2               1
    2               2               1
```

```
ans =

    hRatios    errorRatios     order
    -------    -----------    -------

    2          0.44201        -1.1778
    2           1.0798         0.1108
    2           1.4695        0.55536
    2           1.7229        0.78484
    2           1.9306        0.94903
    2            1.984         0.9884
    2           1.9987        0.99906
```

Next I will implement the third-order Runge Kutta method.

```
function [u, h, k] = RungeKutta3(a, N, T, icFun)
    % discretize space
    h = 1/(N+1);
    x = @(i) i*h;

    % discretize time
    % find k such that k = O(h), but not exact k = h
    Nt = ceil(T/(.9*h));
    k = T/Nt;
    t = @(n) n*k;

    % initiate matrix to store u at times 1:Nt
    u = zeros(N+2, Nt+1);
    % add initial conditions
    u(:,1) = icFun(x(0:N+1));

    f = @(u) uxFun(u, a, h);

    for n = 1:Nt
        Y1 = f(u(:,n));
        Y2 = f(u(:,n) + k/2*Y1);
        Y3 = f(u(:,n) + 3*k/4*Y2);
        u(:,n+1) = u(:,n) + k/9*(2*Y1 + 3*Y2 + 4*Y3);
        %plot(x(0:N+1),u(:,n+1));
        %title(num2str(N));
        %pause(k)
    end
end
```

```
function [f] = uxFun(u, a, h)
    % f =
    f = -a/(6*h)*([u(end-1); u(end); u(1:end-2)] - 6*[u(end); u(1:end-1)] + 3*u + ...
        2*[u(2:end); u(1)]);
end
```

Testing both initial conditions

```matlab
%% Problem 4 for RungeKutta3 created in Problem 3
    a = 1;
    T = 1;
    uExact = @(t, x) 2*exp(-200*(mod((x-a*t),1) - 1/2).^2);
    f = @(x) uExact(0,x);
    E = [];
    H = [];
    for N = 10*2.^(1:8) - 1
        [u, h, k] = RungeKutta3(a, N, T, f);
        % create exact solution
        t = @(n) n*k;
        x = @(i) i*h;
        uExactMatrix = cell2mat(arrayfun(@(n) uExact(t(n), x(0:N+1))', 0:T/k, ...
            'UniformOutput', false));
        H = [H; h];
        E = [E; norm(u(:,end) - uExactMatrix(:,end), inf)];
    end
    hRatios = H(1:end-1)./H(2:end);
    errorRatios = E(1:end-1)./E(2:end);
    order = log(errorRatios)./log(hRatios);
    table(hRatios, errorRatios, order)

    uExact = @(t, x) 2*exp(-200*(mod((x-a*t),1) - 1/2).^2).*cos(40*pi*mod((x-a*t),1));
    f = @(x) uExact(0,x);
    E = [];
    H = [];
    for N = 10*2.^(1:8) - 1
        [u, h, k] = RungeKutta3(a, N, T, f);
        % create exact solution
        t = @(n) n*k;
        x = @(i) i*h;
        uExactMatrix = cell2mat(arrayfun(@(n) uExact(t(n), x(0:N+1))', 0:T/k, ...
            'UniformOutput', false));
        H = [H; h];
        E = [E; norm(u(:,end) - uExactMatrix(:,end), inf)];
    end
    hRatios = H(1:end-1)./H(2:end);
    errorRatios = E(1:end-1)./E(2:end);
    order = log(errorRatios)./log(hRatios);
    table(hRatios, errorRatios, order)
```

```
ans =

    hRatios    errorRatios     order 
    -------    -----------    -------

    2          1.6631         0.73389
    2          1.8306         0.87232
    2          2.1462          1.1018
    2          2.0825          1.0583
    2          2.0244          1.0175
    2          2.0062          1.0045
    2          2.0016          1.0011


ans =
```

hRatios	errorRatios	order
2	0.5243	-0.93154
2	1.0001	0.00010375
2	1.0191	0.027309
2	1.6093	0.68644
2	2.7795	1.4748
2	2.2621	1.1776
2	2.027	1.0193

For both of these methods I only got first order convergence, with the second set of initial conditions converging slower than the first set of initial conditions. I felt like both were pretty easy to implement. The third order Runge Kutta was probably slightly easier to derive.