MATH 228b: HW7

April Novak

May 2, 2017

1 (a)

The `euler_fluxes.m` function returns the Euler fluxes F given inputs of the four conserved variables - density, two components of momentum, and total energy.

2 (b)

The `spectral_divergence.m` function calculates the divergence of a grid function using the FFT. Given a function $v(x)$, first $\hat{v}$ is computed, where $\hat{v}$ is the Fourier transform of v. Then, because the derivative of a Fourier transform is simply equal to the Fourier transform multiplied by ik, where k are the discrete frequencies, the derivative in the Fourier transform space can be computed as $\hat{w} = ik\hat{v}$. Finally, an inverse Fourier transform is performed to obtained w, where w is the derivative of v. This is the approach used to compute the divergence of a grid function. Note that this function only works for periodic grid functions - otherwise, the divergence "blows up," especially near the domain boundaries.

3 (c)

Similar to the use of the Bubnov-Galerkin formulation for convection-diffusion problems, stabilization is required for spectral methods. This is achieved by using a filter, which essentially filters the grid solution by damping out high frequencies by multiplying low frequencies by a number very close to 1.0 and high frequencies by numbers close to 0.0. This is not a conservative method. The spectral filter is implemented by first filtering in the x direction, and then using this semi-filtered result as input for filtering in the y direction.

4 (d)

The right-hand side of the Euler equations (the divergence of the flux vector) is computed using `euler_rhs.m` by calling `euler_fluxes` followed by `spectral_divergence` once for each of the four equations present.

5 (e)

Given an initial condition from the `euler_vortex.m` function, `euler_rk4step` steps the solutions once using the RK-4 time stepping method and applies filtering to all solution components.

6 (f)

Fig. 1 shows the infinity norm errors as a function of the grid spacing h for the four solution components for the Euler vortex. As can be seen, spectral convergence is obtained, since the rate of convergence is not constant as a function of h, but rather dramatically increases as h is refined. Fig. 2 shows the solution at

1.5615 seconds for $N = 32$. The code used to perform this convergence test is shown in `script.m` in the Appendix.

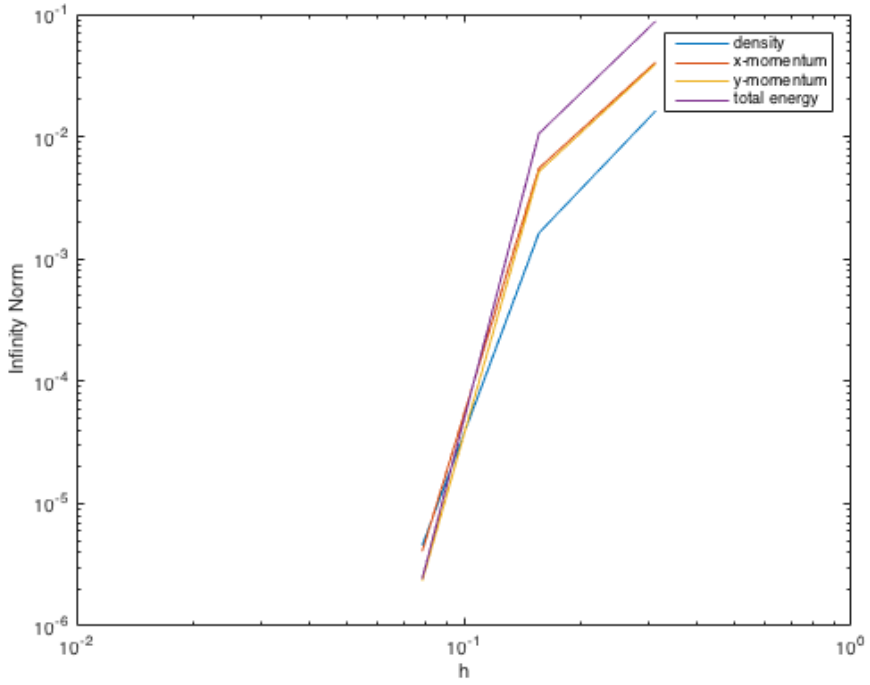

Figure 1. Infinity norm as a function of mesh spacing h for the four solution components.

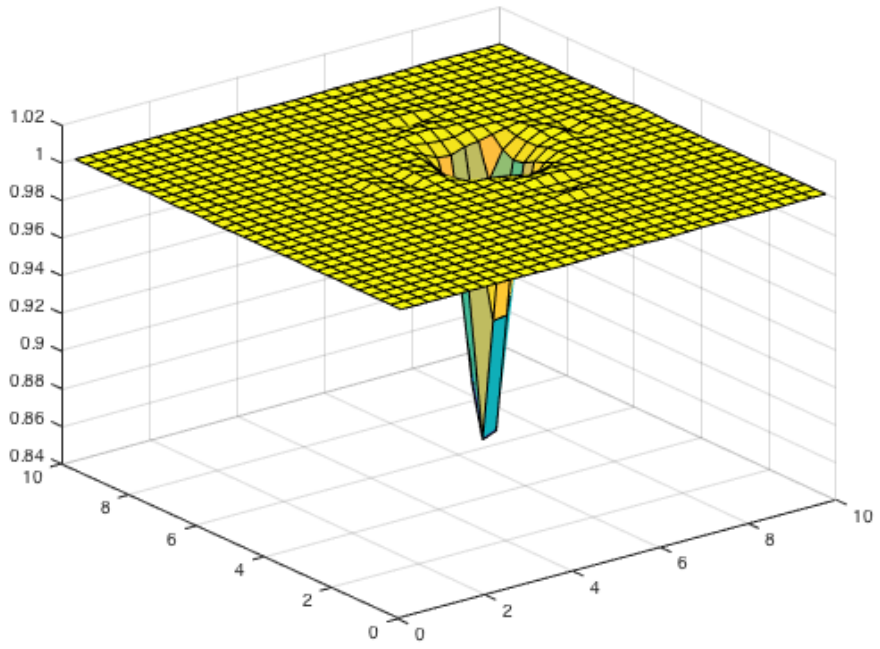

Figure 2. Solution for $N = 32$ at $t = 1.5615$.

7 (g)

Fig. 3 shows the Kelvin-Helmholtz instability at 2 seconds, where the maximum and minimum values of ρ are 2.3312 and 0.5243, respectively. The script developed for this section is shown in the Appendix as `kh.m`.

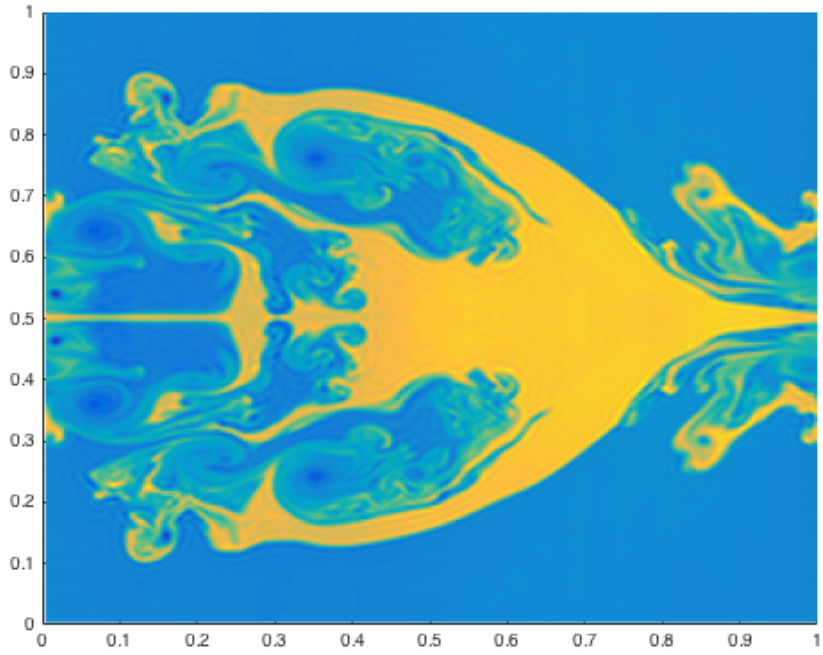

Figure 3. Density solution to the Kelvin-Helmholtz instability at 2 seconds.

8 Appendix

8.1 euler_fluxes.m

```
function [Frx, Fry, Frux, Fruy, Frvx, Frvy, FrEx, FrEy] = euler_fluxes(r, ru, rv, rE)

u = ru ./ r; v = rv ./ r; E = rE ./ r;

% compute pressure from ideal gas law
y = 7/5;
P = (y - 1) .* r .* (E - (u.^2 + v.^2) / 2);

Frx = ru;               Fry = rv;
Frux = r .* u.^2 + P;   Fruy = ru .* v;
Frvx = ru .* v;         Frvy = r .* v.^2 + P;
FrEx = u .* (rE + P);   FrEy = v .* (rE + P);
```

8.2 spectral_divergence.m

```
function divF = spectral_divergence(Fx, Fy, h)

N = length(Fx(1, :));
L = N * h;
k = ones(N, 1) * [0:N/2-1 0 -N/2+1:-1];

% compute x derivative
dx = fft(Fx, [], 2);
```

```
w_hat_x = 1i * k .* dx;
wx = real(ifft(w_hat_x, [], 2));

% compute y derivative
dy = fft(Fy, [], 1);
w_hat_y = 1i * k' .* dy;
wy = real(ifft(w_hat_y, [], 1));

divF = (wx + wy) * (2*pi) / L;
```

8.3 spectral_filter.m

```
function w = spectral_filter(u)

N = length(u(1, :));
k0 = 3 * N / 8;
k = ones(N, 1) * [0:N/2-1 0 -N/2+1:-1];

% filter along the x direction
T = 1 ./ (1 + (k ./ k0) .^ 16);
u_hat = fft(u, [], 2);
w_hat = T .* u_hat;
w = real(ifft(w_hat, [], 2));

% then filter along the y direction using the x-filtered solution
T = 1 ./ (1 + (k' ./ k0) .^ 16);
u_hat = fft(w, [], 1);
w_hat = T .* u_hat;
w = real(ifft(w_hat, [], 1));
```

8.4 euler_rhs.m

```
function [fr, fru, frv, frE] = euler_rhs(r, ru, rv, rE, h)

% determine the Euler fluxes
[Frx, Fry, Frux, Fruy, Frvx, Frvy, FrEx, FrEy] = euler_fluxes(r, ru, rv, rE);

% determine the spectral divergence
fr  = - spectral_divergence(Frx, Fry, h);
fru = - spectral_divergence(Frux, Fruy, h);
frv = - spectral_divergence(Frvx, Frvy, h);
frE = - spectral_divergence(FrEx, FrEy, h);
```

8.5 euler_rk4step.m

```
function [r, ru, rv, rE] = euler_rk4step(r, ru, rv, rE, h, dt)

[fr, fru, frv, frE] = euler_rhs(r, ru, rv, rE, h);
k1_fr  = dt .* fr;
k1_fru = dt .* fru;
k1_frv = dt .* frv;
k1_frE = dt .* frE;

[fr, fru, frv, frE] = euler_rhs(r + k1_fr./2, ru + k1_fru./2, ...
```

```
    rv + k1_frv./2, rE + k1_frE./2, h);
k2_fr  = dt .* fr;
k2_fru = dt .* fru;
k2_frv = dt .* frv;
k2_frE = dt .* frE;

[fr, fru, frv, frE] = euler_rhs(r + k2_fr./2, ru + k2_fru./2, ...
    rv + k2_frv./2, rE + k2_frE./2, h);
k3_fr  = dt .* fr;
k3_fru = dt .* fru;
k3_frv = dt .* frv;
k3_frE = dt .* frE;

[fr, fru, frv, frE] = euler_rhs(r + k3_fr, ru + k3_fru, ...
    rv + k3_frv, rE + k3_frE, h);
k4_fr  = dt .* fr;
k4_fru = dt .* fru;
k4_frv = dt .* frv;
k4_frE = dt .* frE;

r  =  r +  (k1_fr + 2 .* k2_fr + 2 .* k3_fr + k4_fr)    ./ 6;
ru = ru + (k1_fru + 2 .* k2_fru + 2 .* k3_fru + k4_fru) ./ 6;
rv = rv + (k1_frv + 2 .* k2_frv + 2 .* k3_frv + k4_frv) ./ 6;
rE = rE + (k1_frE + 2 .* k2_frE + 2 .* k3_frE + k4_frE) ./ 6;

% filter each solution component (x and y for the four parts)
r = spectral_filter(r);
ru = spectral_filter(ru);
rv = spectral_filter(rv);
rE = spectral_filter(rE);
end
```

8.6 script.m

```
clear all
N = [32, 64, 128];
inf_norm = zeros(length(N), 4);

for i = 1:length(N)
    n = N(i);
    h = 10 / N(i);
    x = h:h:10; y = h:h:10;
    [X, Y] = meshgrid(x, y);

    final = sqrt(20^2 + 10^2);
    dt = final/(ceil(final / (0.2 * h)));
    dt = dt / 4;
    num_dt = final/dt;

    % obtain the initial condition from euler_vortex
    pars = [0.5, 1, 0.5, atan2(1,2), 5, 5];
    [r, ru, rv, rE] = euler_vortex(X, Y, 0, pars);
    r_ex = r; ru_ex = ru; rv_ex = rv; rE_ex = rE;

    % RK-4 timestepping
    for it = 1:num_dt
        [r, ru, rv, rE] = euler_rk4step(r, ru, rv, rE, h, dt);
```

```
        if (mod(it, 100) == 0)
            surf(X, Y, r)
        end
    end

    inf_norm(i, :) = [max(max(abs(r_ex - r))), max(max(abs(ru_ex - ru))), ...
        max(max(abs(rv_ex - rv))), max(max(abs(rE_ex - rE)))];
end

loglog(10./N, inf_norm(:, 1))
hold on
loglog(10./N, inf_norm(:, 2))
hold on
loglog(10./N, inf_norm(:, 3))
hold on
loglog(10./N, inf_norm(:, 4))
hold on
legend('density', 'x-momentum', 'y-momentum', 'total energy')
```

8.7 kh.m

```
clear all
N = 256;
h = 1 / N;
dt = 0.2 * h;

x = h:h:1; y = h:h:1;
[X, Y] = meshgrid(x, y);

num_dt = 2.0/dt;
P = 3; gamma = 7/5;

% set the initial condition
r = zeros(length(x), length(y));
for i = 1:length(y)
    for j = 1:length(x)
        if (abs(y(i) - 0.5) < (0.15 + sin(2 * pi * x(j)) / 200))
            r(i, j) = 2;
        else
            r(i, j) = 1;
        end
    end
end

ru = r .* (r - 1);
rv = 0 .* r;
rE = P/(gamma - 1) + (ru.^2 + rv.^2)./(2.*r);

% RK-4 timestepping
for it = 1:num_dt
    [r, ru, rv, rE] = euler_rk4step(r, ru, rv, rE, h, dt);
    if (mod(it, 20) == 0)
        s = surf(X, Y, r);
        s.EdgeColor = 'none';
        view(0, 90)
        drawnow
    end
```

```
end
```