MATH 228b: HW5

April Novak

April 6, 2017

1

The first part of this question requires replacing equidistant node positions with the Chebyshev nodes, which should eliminate Runge phenomena, which is commonly observed as oscillations in high-degree polynomials. The Chebyshev node positions s_i are given as $s_i = \cos(\pi i/p)$, where $p = 0, \cdots, P$, where P is the polynomial order of the shape functions. The Chebyshev nodes are defined on $[-1, 1]$, so to transform these to a single element defined over $[0, h]$, the Chebyshev nodes are multiplied by $-2/h$, where 2 is the length of the domain of the Chebyshev nodes, and h the length of the desired domain.

Next, we need to compute the elementary mass and stiffness matrices. The strong form of the governing equation is:

$$\begin{aligned}\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} =&0\\ \frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} =&0\end{aligned} \tag{1}$$

where $f(u) = u$ is the flux function. We will expand the solution u in a single element:

$$u_h = \sum_{j=0}^{P} u_j \phi_j \tag{2}$$

and over the entire domain of K elements:

$$u_h = \sum_{k=1}^{K}\sum_{j=0}^{P} u_j^k \phi_j^k(x) \tag{3}$$

where ϕ_j are the expansion functions within a single element. Following the Galerkin approach, the test function v_h will also be expanded in the same set of basis functions. Integrate the flux term by parts, which due to the discontinuous nature of u_h and v_h cannot technically be performed. It is here that the discontinuous Galerkin method borrow from the finite volume method. For the discontinuities, approximate using a numerical flux function $F(u_h)$. At the left side of an element (at x_k), this numerical flux should depend (at a minimum at least) only on u_0^k and u_P^{k-1}, where now a new notation is introduced to describe the multiple expansion coefficients in each element (ranging from 0 to P).

$$\begin{aligned}\int_\Omega \frac{\partial u_h}{\partial t} v_h dx + \int_\Omega \frac{\partial f(u_h)}{\partial x} v_h dx =&0\\ \int_\Omega \frac{\partial u_h}{\partial t} v_h dx - \int_\Omega f(u_h)\frac{\partial v_h}{\partial x} dx + \left[f(u_h) v_h \cdot \hat{n}\right]_{x_{k-i}}^{x_k} =&0\\ \int_\Omega \frac{\partial u_h}{\partial t} v_h dx - \int_\Omega f(u_h)\frac{\partial v_h}{\partial x} dx + F(u_0^{k+1}, u_P^k) v_h(x_k) - F(u_0^k, u_P^{k-1}) v_h(x_{k-1}) =&0\end{aligned} \tag{4}$$

Substituting in the expansion over a single element into the equation above:

$$\begin{aligned}\int_\Omega \frac{\partial}{\partial t}\left(\sum_{j=0}^P u_j\phi_j(x)\right)\sum_{i=0}^P v_i\phi_i(x)dx - \int_\Omega f(u_h)\frac{\partial}{\partial x}\left(\sum_{i=0}^P v_i\phi_i(x)\right)dx + \\ F(u_0^{k+1},u_P^k)\sum_{i=0}^P v_i\phi_i(x_k) - F(u_0^k,u_P^{k-1})\sum_{i=0}^P v_i\phi_i(x_{k-1}) = 0\end{aligned} \tag{5}$$

Then, because v_i appears in every term and is arbitrary, it can be "cancelled" from each term by pulling the summation over i outside all integrals such that the integrand must be zero.

$$\begin{aligned}\int_\Omega \frac{\partial}{\partial t}\left(\sum_{j=0}^P u_j\phi_j(x)\right)\sum_{i=0}^P \phi_i(x)dx - \int_\Omega f(u_h)\frac{\partial}{\partial x}\left(\sum_{i=0}^P \phi_i(x)\right)dx + \\ F(u_0^{k+1},u_P^k)\sum_{i=0}^P \phi_i(x_k) - F(u_0^k,u_P^{k-1})\sum_{i=0}^P \phi_i(x_{k-1}) = 0\end{aligned} \tag{6}$$

Then, while the above equation is true, it is also true that it holds for each individual choice of i and j such that the summations can be removed with the implicit assumption of creating matrices that are of size $p+1\times p+1$ for the creation of a system of equations that holds for a single element. For the case of linear advection, $f(u_h) = u_h$, giving the second form below.

$$\begin{aligned}\int_\Omega \frac{\partial u_j}{\partial t}\phi_j(x)\phi_i(x)dx - \int_\Omega f(u_h)\frac{\partial\phi_i(x)}{\partial x}dx + F(u_0^{k+1},u_P^k)\phi_i(x_k) - F(u_0^k,u_P^{k-1})\phi_i(x_{k-1}) =& 0\\ \int_\Omega \frac{\partial u_j}{\partial t}\phi_j(x)\phi_i(x)dx - \int_\Omega u_j\phi_j(x)\frac{\partial\phi_i(x)}{\partial x}dx + F(u_0^{k+1},u_P^k)\phi_i(x_k) - F(u_0^k,u_P^{k-1})\phi_i(x_{k-1}) =& 0\end{aligned} \tag{7}$$

Define the mass matrix M_{ij} as:

$$M_{ij}^k = \int_{x_{k-1}}^{x_k}\phi_j(x)\phi_i(x)dx \tag{8}$$

and the stiffness matrix K_{ij}^k as:

$$K_{ij}^k = \int_{x_{k-1}}^{x_k}\phi_j(x)\frac{\partial\phi_i(x)}{\partial x}dx \tag{9}$$

For linear advection (i.e. with constant velocity), a good choice of numerical flux is to use upwinding such that the downwind value is always selected as the numerical flux. Then, the weak form becomes:

$$\int_\Omega \frac{\partial u_j}{\partial t}\phi_j(x)\phi_i(x)dx - \int_\Omega u_j\phi_j(x)\frac{\partial\phi_i(x)}{\partial x}dx + u_P^k\phi_i^k(x_k) - u_P^{k-1}\phi_i^k(x_{k-1}) = 0 \tag{10}$$

Then a system of equations that holds for a single element is:

$$M_{ij}^k \dot{u_j}^k - K_{ij}^k u_j^k + u_P^k\phi_i^k(x_k) - u_P^{k-1}\phi_i^k(x_{k-1}) = 0 \tag{11}$$

Because we will use nodal basis functions, $\phi_i^k(x_k)$ and $\phi_i^k(x_{k-1})$ are zero at all nodes except a single node. Hence, all of these extra terms will drop out except for one term each, and the shape function at that node will be unity.

Finally, to implement arbitrary order polynomials, the mass and stiffness matrices must be computed. Because very high order polynomials will be used, these should be computed in a domain over $[-1,1]$ so that Gaussian quadrature can easily be used. These matrices transform according to:

$$\begin{aligned}M_{ij}^k &= \int_{-1}^1 \phi_j(\xi)\phi_i(\xi)\frac{dx}{d\xi}d\xi\\ &= \int_{-1}^1 \phi_j(\xi)\phi_i(\xi)\frac{h}{2}d\xi\end{aligned} \tag{12}$$

where $-1 \leq \xi \leq 1$ represents the domain over which the integration will occur and the Jacobian of the transformation in 1-D is simply $h/2$ because the length of the domain in the physical domain is h and the length in the ξ domain is 2. Likewise, the stiffness matrix transforms according to:

$$\begin{aligned} K_{ij}^k &= \int_{-1}^{1} \phi_j(\xi) \frac{\partial \phi_i(\xi)}{\partial \xi} \frac{\partial \xi}{\partial x} \frac{\partial x}{\partial \xi} d\xi \\ &= \int_{-1}^{1} \phi_j(\xi) \frac{\partial \phi_i(\xi)}{\partial \xi} d\xi \end{aligned} \tag{13}$$

Legendre polynomials are used as the expansion functions for this problem because they will eliminate the oscillatory nature of higher-degree monomials. However, the stand-alone Legendre polynomials are not a nodal basis. To make these polynomials a nodal basis, we write our basis functions ϕ_i as a sum of P Legendre polynomials:

$$\phi_i(\xi) = \sum_{i=0}^{P} c_i^j P_j(\xi) \tag{14}$$

For $P = 2$ (just for example), the coefficients c_i^j can be determined by requiring that the basis function goes to zero at all nodes except one. For nodes s_p (of which there are $p + 1$):

$$\begin{bmatrix} P_0(s_0) & P_1(s_0) & P_2(s_0) \\ P_0(s_1) & P_1(s_1) & P_2(s_1) \\ P_0(s_2) & P_1(s_2) & P_2(s_2) \end{bmatrix} \begin{bmatrix} c_0^0 & c_1^0 & c_2^0 \\ c_0^1 & c_1^1 & c_2^1 \\ c_0^2 & c_1^2 & c_2^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

This must be performed for each element to determine the basis functions for each element. Alternatively, because the integrals are to be performed in the ξ domain anyways, these can be found a single time in the ξ domain (and then reused for every element). After these coefficients are determined, the elemental mass and stiffness matrices are determined according to the equations given previously. The `legendre_poly.m` function is used for evaluating these polynomials and their derivatives over the reference element. Then, the remainder of the solve does not need to be modified, since at this point it is a "turn-the-crank" procedure.

To plot the solution using $3P$ points per element requires using the nodal values to determine the solution in each element. We need to evaluate the solution at these points in order to be able to interpolate the solution values at these $3P$ points with straight lines (as requested). Because all of the integrals were transformed to the reference element appropriately, all of the solution values `u` obtained during the time-stepping loop are valid in the physical domain. However, for plotting, the expansion functions in the physical domain must be determined - up until now, they have only been defined for a reference element. The solution over an element in the reference domain for $P = 2$ is:

$$u_h^{e=1} = C_1^0 \phi_0(\xi) + C_1^1 \phi_1(\xi) + C_1^2 \phi_2(\xi) \tag{16}$$

And in the physical domain, it is:

$$u_h^{e=1} = u_1^0 \phi_0(x) + u_1^1 \phi_1(x) + u_1^2 \phi_2(x) \tag{17}$$

But, the expansion functions are only known in the reference element. Because the Legendre polynomials form a basis, we can express any P-order polynomial using a summation of P Legendre polynomials. Hence, for $P = 2$ for instance, the solution in each element is a cubic of the form:

$$u_h^e = A_0^e + A_1^e x + A_2^e x^2 \tag{18}$$

where once these coefficients are determined, the solution over the element is fully known. For an element $e = 1$ with nodes at $x = 0.25, 0.5, 0.75$, we can solve for these coefficients A by:

$$\begin{bmatrix} 1 \cdot 0.25^0 & 1 \cdot 0.25^1 & 1 \cdot 0.25^2 \\ 1 \cdot 0.50^0 & 1 \cdot 0.50^1 & 1 \cdot 0.50^2 \\ 1 \cdot 0.75^0 & 1 \cdot 0.75^1 & 1 \cdot 0.75^2 \end{bmatrix} \begin{bmatrix} A_0^1 \\ A_1^1 \\ A_2^1 \end{bmatrix} = \begin{bmatrix} u_1^0 \\ u_1^1 \\ u_1^2 \end{bmatrix} \tag{19}$$

This is then repeated for every element until the solution over each element is fully specified. Then, the monomial defined using the coefficients A is sampled at the desired interpolation points for plotting purposes (at $3P$ points).

Finally, to compute the error, the L^2 norm is computed over the entire domain. Because the solution is discontinuous, this is straightforward, and the solution over each element is integrated from x_{k-1} to x_k and then summed (and then square-rooted). To compute the norm, the exact solution is interpolated into a discontinuous Galerkin basis (i.e. it is expressed in the same form as `u` from the original `dgconvect0.m` code), and then trapezoidal integration is performed over each element by simply taking the square of the difference entry-wise between the discontinuous Galerkin solution and the analytical solution. To ensure that accurate norms are obtained, the interpolation of the analytical and numerical solutions into new spatial domains of length 500 nodes/element is performed. Because the error can be found at each time step, for comparison it is only evaluated and compared for the last time step.

The `errors` matrix returned by `dgconvect_convergence` reports errors in the format such that each row corresponds to the errors for a given p. Fig. **??** shows the raw data obtained for the requested points (and more to obtain better slope estimates) for all five of the polynomial orders. As can be seen, points below a logarithm of the L^2 norm of about 10^{-8} are affected by numerical rounding, since theoretically the error should monotonically decrease. Fig. **??** shows the same data, but with points with an L^2 norm less that 10^{-8} removed from the data set. As can be seen, once these spurious data points are removed, all polynomial orders except 16 obtain fairly close to theoretical convergence rates. Like the continuous Galerkin method, the rate of convergence in the L^2 norm should be the order of the polynomial expansion plus one. A poor rate of convergence is observed for $p = 16$ (and somewhat also for $p = 8$, which converges at a slope less than theoretical) because at such a fine solution, the time error associated with the Runge-Kutta stepping likely dominates the total error. To completely eliminate the time error, a method of manufactured solutions test could be performed to show that for an analytical, *linear*, dependence on time, all of the polynomial orders should obtain theoretical convergences. All of the code for this section is included in the Appendix. For the purpose of replicating these convergence results, the plotting section of `dgconvect` should be commented out.

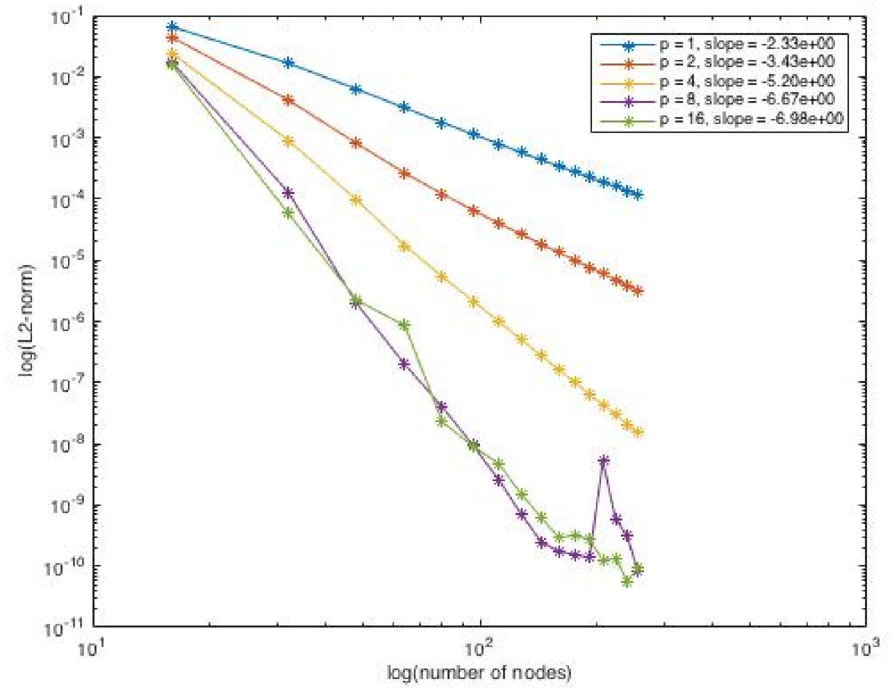

Figure 1. L^2 norm as a function of the number of nodes.

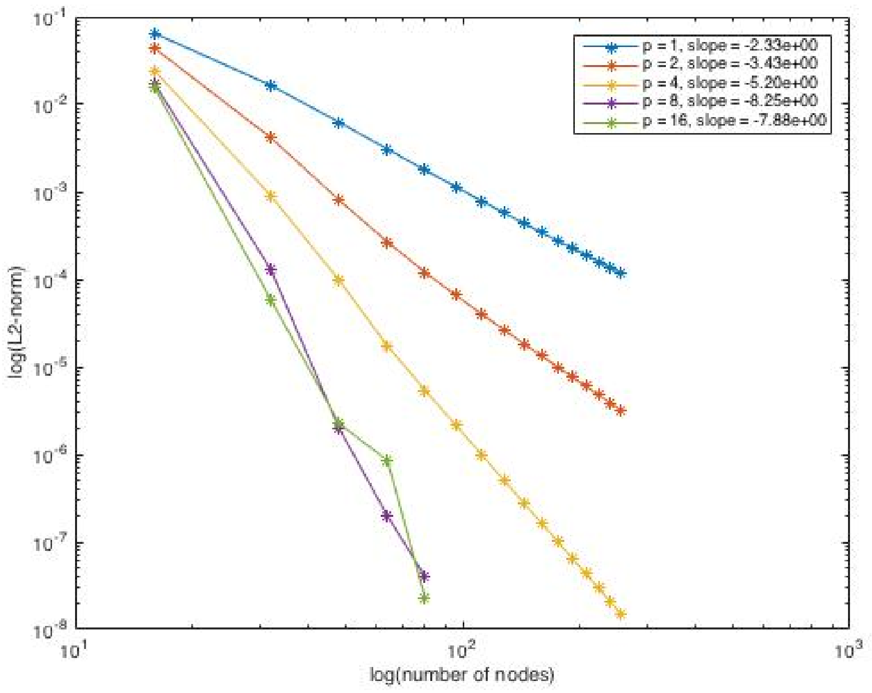

Figure 2. L^2 norm as a function of the number of nodes with data points with an L^2 norm less that 10^{-9} removed from the data.

2

This problem modifies the code developed in the first part by solving the convection-diffusion equation:

$$\frac{\partial u}{\partial t}+\frac{\partial f(u)}{\partial x}-k\frac{\partial^2 u}{\partial x^2}=0 \tag{20}$$

The solution approach follows that in question 1, where again $f(u)=u$, except that the above will be split into a system of two first-order equations, because otherwise we would need to determine a numerical "flux function" for specifying $\partial u/\partial x$ on the boundary, which can be difficult.

$$\begin{gathered}\frac{\partial u}{\partial t}+\frac{\partial f(u)}{\partial x}-k\frac{\partial \sigma}{\partial x}=0\\ \sigma=\frac{\partial u}{\partial x}\end{gathered} \tag{21}$$

Then, the approach from here is similar to that taken in the first problem. Express u and the test function v as an expansion, integrate over space, and integrate by parts when possible. The shape function used in the equation for σ is denoted as τ.

$$\begin{gathered}\int_\Omega \frac{\partial u_h}{\partial t}v_h dx+\int_\Omega \frac{\partial f(u_h)}{\partial x}v_h dx-\int_\Omega k\frac{\partial \sigma_h}{\partial x}v_h=0\\ \int_\Omega \sigma_h\tau_h dx=\int_\Omega \frac{\partial u_h}{\partial x}\tau_h dx\end{gathered} \tag{22}$$

Now, integrate by parts to give:

$$\begin{aligned}\int_\Omega \frac{\partial u_h}{\partial t} v_h dx - \int_\Omega \frac{\partial v_h}{\partial x} f(u_h) dx + \int_\Omega k\frac{\partial v_h}{\partial x}\sigma_h + \left[f(u_h) v_h \cdot \hat{n}\right]_0^1 - \left[k\sigma_h v_h \cdot \hat{n}\right]_0^1 = 0 \\ \int_\Omega \sigma_h \tau_h dx = -\int_\Omega \frac{\partial \tau_h}{\partial x} u_h dx + \left[u_h \tau_h \cdot \hat{n}\right]_0^1\end{aligned} \tag{23}$$

Next, we assume that u_h, v_h, σ_h, and τ_h all come from the same space of functions. Inserting these expansions into the above gives, for a single element:

$$\begin{aligned}\int_{x_{k-1}}^{x_k} \frac{\partial}{\partial t}\left(\sum_{j=0}^P u_j\phi_j\right)\sum_{i=0}^P v_i\phi_i dx - \int_{x_{k-1}}^{x_k}\frac{\partial}{\partial x}\left(\sum_{i=0}^P v_i\phi_i\right)\sum_{j=0}^P u_j\phi_j dx + \\ \int_{x_{k-1}}^{x_k} k\frac{\partial}{\partial x}\left(\sum_{i=0}^P v_i\phi_i\right)\sum_{j=0}^P \sigma_j\phi_j + \left[\sum_{j=0}^P u_j\phi_j\sum_{i=0}^P v_i\phi_i\cdot\hat{n}\right]_{x_{k-1}}^{x_k} - \left[k\sum_{j=0}^P \sigma_j\phi_j\sum_{i=0}^P v_i\phi_i\cdot\hat{n}\right]_{x_{k-1}}^{x_k} = 0\end{aligned} \tag{24}$$

$$\int_{x_{k-1}}^{x_k}\sum_{j=0}^P\sigma_j\phi_j\sum_{i=0}^P\tau_i\phi_i dx = -\int_{x_{k-1}}^{x_k}\frac{\partial}{\partial x}\left(\sum_{i=0}^P\tau_i\phi_i\right)\sum_{j=0}^P u_j\phi_j dx + \left[\sum_{j=0}^P u_j\phi_j\sum_{i=0}^P\tau_i\phi_i\cdot\hat{n}\right]_{x_{k-1}}^{x_k} \tag{25}$$

Because τ_i is arbitrary, it can be "cancelled" from every term in the second equation by grouping all of the terms such that τ_i multiplies an integrand. Then, that integrand must be zero, allowing τ_i to be cancelled. The same argument applies for v_i. Then, while the equations are true when using the entire summation, it is also true for each individual p.

$$\begin{aligned}\int_{x_{k-1}}^{x_k} \frac{\partial}{\partial t}\left(\sum_{j=0}^P u_j\phi_j\right)\sum_{i=0}^P \phi_i dx - \int_{x_{k-1}}^{x_k}\frac{\partial}{\partial x}\left(\sum_{i=0}^P \phi_i\right)\sum_{j=0}^P u_j\phi_j dx + \\ \int_{x_{k-1}}^{x_k} k\frac{\partial}{\partial x}\left(\sum_{i=0}^P \phi_i\right)\sum_{j=0}^P \sigma_j\phi_j + \left[\sum_{j=0}^P u_j\phi_j\sum_{i=0}^P \phi_i\cdot\hat{n}\right]_{x_{k-1}}^{x_k} - \left[k\sum_{j=0}^P \sigma_j\phi_j\sum_{i=0}^P \phi_i\cdot\hat{n}\right]_{x_{k-1}}^{x_k} = 0\end{aligned} \tag{26}$$

$$\begin{aligned}\int_{x_{k-1}}^{x_k}\frac{\partial u_j}{\partial t}\phi_j\phi_i dx - \int_{x_{k-1}}^{x_k}\frac{\partial\phi_i}{\partial x}u_j\phi_j dx + \int_{x_{k-1}}^{x_k}k\frac{\partial\phi_i}{\partial x}\sigma_j\phi_j + \left[u_j\phi_j\phi_i\cdot\hat{n} - k\sigma_j\phi_j\phi_i\cdot\hat{n}\right]_{x_{k-1}}^{x_k} = 0 \\ \int_{x_{k-1}}^{x_k}\sum_{j=0}^P\sigma_j\phi_j\sum_{i=0}^P\phi_i dx = -\int_{x_{k-1}}^{x_k}\frac{\partial}{\partial x}\left(\sum_{i=0}^P\phi_i\right)\sum_{j=0}^P u_j\phi_j dx + \left[\sum_{j=0}^P u_j\phi_j\sum_{i=0}^P\phi_i\cdot\hat{n}\right]_{x_{k-1}}^{x_k} \\ \int_{x_{k-1}}^{x_k}\sigma_j\phi_j\phi_i dx = -\int_{x_{k-1}}^{x_k}\frac{\partial\phi_i}{\partial x}u_j\phi_j dx + \left[u_j\phi_j\phi_i\cdot\hat{n}\right]_{x_{k-1}}^{x_k}\end{aligned} \tag{27}$$

Define a mass matrix as:

$$M_{ij}^k = \int_{x_{k-1}}^{x_k}\phi_j\phi_i dx \tag{28}$$

and a stiffness matrix as:

$$K_{ij}^k = \int_{x_{k-1}}^{x_k}\frac{\partial\phi_i}{\partial x}\phi_j dx \tag{29}$$

I assume that the method shown in the DG slides on BCourses show the way to interpret the cryptic instructions that $C_{11} = 0$ and $C_{12} = 1/2$. This corresponds to pure upwinding in the first equation and pure downwinding in the second equation. With these specifications for the numerical fluxes, the system of equations becomes, for a single element:

$$M_{ij}^k \dot{u_j}^k = K_{ij}^k \left(u_j - k\sigma_j\right) - \begin{bmatrix} -(u_p^{k-1} - k\sigma_p^{k-1}) \\ \vdots \\ u_p^k - k\sigma_p^k \end{bmatrix} \tag{30}$$

$$M_{ij}^k \sigma_j^k = -K_{ij}^k u_j^k + \begin{bmatrix} -u_0^k \\ \vdots \\ u_0^{k+1} \end{bmatrix} \tag{31}$$

where the $\cdots$ notation indicates that all of the other entries are zero due to the use of a nodal basis. From here, the same general process is performed as done in question 1. After calculating the mass and stiffness matrices, Eq. (**??**) is solved for σ_j using the initial condition for u_h. Then, Eq. (**??**) is solved to determine u_j. This process is repeated for every time step. So, the only part of `dgconvect.m` that must be modified is the time-stepping loop.

The exact solution provided must be shifted in space corresponding to the local velocity in order to account for the convective part of the solution. The diffusive kernel causes the solution to spread, while the convection kernel causes this entire spreading to move to the right at velocity $u = 1$. The higher the thermal conductivity, the faster the solution spreads out to reach a flat profile. For the convergence study, the plotting should again be commented out to save time. The same basic structure of the convergence program is used to assess convergence here. Fig. **??** shows the convergence rates obtained using my code - as can be seen, optimal convergence rates are not obtained (and I'm not sure why they are not).

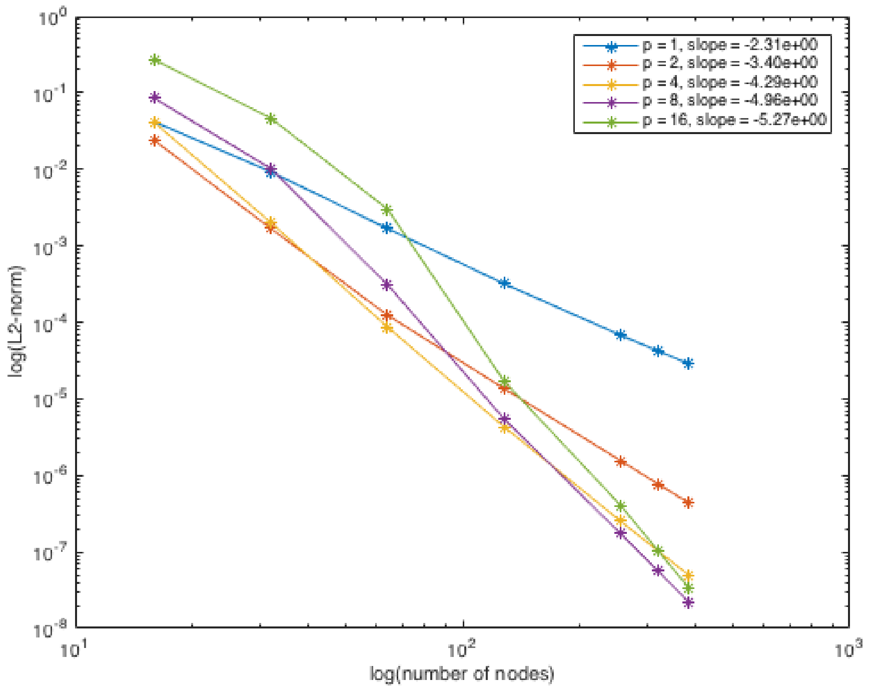

Figure 3. L^2 norm as a function of the number of nodes.

3 Appendix

3.1 Question 1

3.1.1 dgconvect.m

```
function [u, error] = dgconvect(n, p, T, dt)
% n = number of elements
% p = polynomial order
% T = end time
% dt = time step

% number of points to resolve per discontinuous element
if n <= 4
    fine_el = 5000;
else
    fine_el = 1000;
end
% number of points to resolve for analytical solution
exact_el = fine_el * n - (n - 1);

h = 1/n;
s_master = - cos(pi * (0:p) / p);
s = ((s_master + 1).* h / 2)';                   % Chebyshev nodes
x = s * ones(1,n) + ones(p+1,1) * (0:h:1-h);    % Entire mesh
uinit = @(x) exp(-(x - 0.5).^2 / 0.1^2);        % Initial solution
xx = linspace(0, 1, exact_el);                   % Fine grid for exact soln

% find basis function coefficients (c_i^j) for an element in master domain
A = [];
I = eye(length(s));

for i = 1:length(s)
    [y, dy] = legendre_poly(s_master(i), p);
    A(i, :) = y;
end

c = A \ I;

% quadrature points and weights
[qp, wt] = gauss_quad(2*p);

Mel = zeros(p + 1, p + 1); Kel = zeros(p + 1, p + 1);

for q = 1:length(qp)
    [yi, dyi] = legendre_poly(qp(q), p);
    [yj, dyj] = legendre_poly(qp(q), p);
    for i = 1:(p + 1)
        for j = 1:(p + 1)
            Mel(i, j) = Mel(i, j) + wt(q) * dot(c(:, i), yi) * dot(c(:, j), yj) * h /
                ↪ 2;
            Kel(i, j) = Kel(i, j) + wt(q) * dot(c(:, j), yj) * dot(c(:, i), dyi);
        end
    end
end

u = uinit(x);
```

```
% create matrix of evaluation points for plotting
x_plot = [];
for el = 1:n
    x_plot(:, el) = linspace(x(1, el), x(end, el), 3*p);
end

horiz = [[0, x(end, :)]; [0, x(end, :)]];
vert = [0; 1] * ones(1, length([0, x(end, :)]));

for it = 1:T/dt
    k1 = dt * rhs(u, Kel, Mel);
    k2 = dt * rhs(u + k1/2, Kel, Mel);
    k3 = dt * rhs(u + k2/2, Kel, Mel);
    k4 = dt * rhs(u + k3, Kel, Mel);
    u = u + (k1 + 2*k2 + 2*k3 + k4) / 6;

    if mod(it, T/dt/1000) == 0                  % 1000 frames
        uexact = uinit(mod(xx - dt*it, 1.0));   % Exact solution
        u_plot = []; A = []; coeff_plot = [];

        for el = 1:n
            for i = 1:(p + 1)
                A(i, 1:(p + 1)) = x(i, el) .^ ((1:(p + 1)) - 1);
            end

            coeff_plot(:, el) = A \ u(:, el);

            for i = 1:length(x_plot(:, 1))
                u_plot(i, el) = sum(coeff_plot(:, el)' .* (x_plot(i, el) .^ ((1:(p +
                    ↪ 1)) - 1)));
            end
        end

        plot(x_plot, u_plot, 'b*-', x, u, 'r', xx, uexact, 'k', horiz, vert, '--')
        grid on
        axis([0, 1, -0.1, 1.1])
        drawnow
    end
end

uexact = uinit(mod(x - T, 1.0));                % Exact final solution

% determines the solution in the physical domain for plotting
x_norm = [];
for el = 1:n
    x_norm(:, el) = linspace(x(1, el), x(end, el), fine_el);
end

A = []; coeff_plot = []; u_norm = [];
for el = 1:n
    for i = 1:(p + 1)
        A(i, 1:(p + 1)) = x(i, el) .^ ((1:(p + 1)) - 1);
    end

    coeff_plot(:, el) = A \ u(:, el);

    for i = 1:length(x_norm(:, 1))
        u_norm(i, el) = sum(coeff_plot(:, el)' .* (x_norm(i, el) .^ ((1:(p + 1)) - 1))
            ↪ );
    end
end

% u_norm : DG solution
% u_exact_norm : exact solution
u_exact_n = uinit(mod(xx - T, 1.0));
u_exact_norm = [];

i = 1;
j = fine_el;
for el = 1:n
    u_exact_norm(:, el) = u_exact_n(i:j)';
    i = i + fine_el - 1;
    j = j + fine_el - 1;
end

% L-2 norm for each element is determined using trapz() at the last time
% step only (follows the original code)
L2_elem = [];
for el = 1:n
    L2_elem(el) = trapz(x_norm(:, el), (u_exact_norm(:, el)' - u_norm(:, el)') .^ 2);
end

error = sqrt(sum(L2_elem));

end
```

3.1.2 dgconvect_convergence.m

```
function [errors, slopes] = dgconvect_convergence

T = 1;                      % end simulation time
dt = 2e-4;                  % time step size
P = [1, 2, 4, 8, 16];       % polynomial orders for convergence study
N = 16:16:256;              % numbers of nodes

errors = zeros(length(P), length(P));

for p = 1:length(P)
    en = 1;
    for n = N ./ P(p)
        [u, errors(p, en)] = dgconvect(n, P(p), T, dt);
        en = en + 1;
    end
end


slopes = [];
leg = cell(1, length(P)); leg_text = ''; % to hold the legend labels

for p = 1:length(P)
    p_error = errors(p, :);
    % to remove points affected by rounding:
    p_error = p_error(p_error > 1e-8);
    N = N(1:length(p_error));
    % find the rates of convergence
    fit = polyfit(log(N), log(p_error), 1);
    slopes(p) = fit(1);
    loglog(N, p_error, '*-')
    leg{p} = sprintf('p␣=␣%i,␣slope␣=␣%.2i', P(p), slopes(p));
    hold on

    if p == length(P)
        leg_text = strcat(leg_text, sprintf('␣leg{%i}', p));
    else
        leg_text = strcat(leg_text, sprintf('␣leg{%i},', p));
    end
end

leg_text = strcat(strcat('legend(', leg_text), ')');
eval(leg_text)
xlabel('log(number␣of␣nodes)')
ylabel('log(L2-norm)')

end
```

3.2 Question 2

3.2.1 dgconvdiff.m

```
function [u, error] = dgconvdiff(n, p, T, dt, k)
% n = number of elements
% p = polynomial order
% T = end time
% dt = time step
% k = thermal conductivity

% number of points to resolve per discontinuous element
if n <= 4
    fine_el = 5000;
else
    fine_el = 1000;
end
% number of points to resolve for analytical solution
exact_el = fine_el * n - (n - 1);

h = 1/n;
s_master = - cos(pi * (0:p) / p);
s = ((s_master + 1).* h / 2)';                   % Chebyshev nodes
x = s * ones(1,n) + ones(p+1,1) * (0:h:1-h);    % Entire mesh
uinit = @(x) exp(-(x - 0.5).^2 / 0.1^2);        % Initial solution
xx = linspace(0, 1, exact_el);                  % Fine grid for exact soln

% find basis function coefficients (c_i^j) for an element in master domain
A = [];
I = eye(length(s));

for i = 1:length(s)
    [y, dy] = legendre_poly(s_master(i), p);
    A(i, :) = y;
end

c = A \ I;

% quadrature points and weights
[qp, wt] = gauss_quad(2*p);
```

```
Mel = zeros(p + 1, p + 1); Kel = zeros(p + 1, p + 1);

for q = 1:length(qp)
    [yi, dyi] = legendre_poly(qp(q), p);
    [yj, dyj] = legendre_poly(qp(q), p);
    for i = 1:(p + 1)
        for j = 1:(p + 1)
            Mel(i, j) = Mel(i, j) + wt(q) * dot(c(:, i), yi) * dot(c(:, j), yj) * h /
                ↪ 2;
            Kel(i, j) = Kel(i, j) + wt(q) * dot(c(:, j), yj) * dot(c(:, i), dyi);
        end
    end
end

u = uinit(x);

% create matrix of evaluation points for plotting
x_plot = [];
for el = 1:n
    x_plot(:, el) = linspace(x(1, el), x(end, el), 3*p);
end

horiz = [[0, x(end, :)]; [0, x(end, :)]];
vert = [0; 1] * ones(1, length([0, x(end, :)]));

for it = 1:T/dt
    % solve for u
    k1 = dt * rhsdiff(u, Kel, Mel, k);
    k2 = dt * rhsdiff(u + k1/2, Kel, Mel, k);
    k3 = dt * rhsdiff(u + k2/2, Kel, Mel, k);
    k4 = dt * rhsdiff(u + k3, Kel, Mel, k);
    u = u + (k1 + 2*k2 + 2*k3 + k4) / 6;

    if mod(it, T/dt/500) == 0                  % 1000 frames
        uexact = zeros(1, length(xx));

        for z = -1:1:1
            tt = it * dt;
            xxx = mod(xx - tt, 1.0);
            term = 1 + 400 * k * tt;
            uexact = uexact + (1 / sqrt(term)) * exp(-100 .* ((xxx - 0.5 + z).^ 2) ./
                ↪ term);
        end

        u_plot = []; A = []; coeff_plot = [];

        for el = 1:n
            for i = 1:(p + 1)
                A(i, 1:(p + 1)) = x(i, el) .^ ((1:(p + 1)) - 1);
            end

            coeff_plot(:, el) = A \ u(:, el);

            for i = 1:length(x_plot(:, 1))
                u_plot(i, el) = sum(coeff_plot(:, el)' .* (x_plot(i, el) .^ ((1:(p +
                    ↪ 1)) - 1)));
            end
        end
        plot(x_plot, u_plot, 'b*-', xx, uexact, 'k', horiz, vert, '--')
        grid on
        axis([0, 1, -0.1, 1.1])
        drawnow
    end
end

% exact final solution
uexact = zeros(1, length(xx));
for z = -1:1:1
    tt = T;
    xxx = mod(xx - tt, 1.0);
    term = 1 + 400 * k * tt;
    uexact = uexact + (1 / sqrt(term)) * exp(-100 .* ((xxx - 0.5 + z).^ 2) ./ term);
end

% determines the solution in the physical domain for plotting
x_norm = [];
for el = 1:n
    x_norm(:, el) = linspace(x(1, el), x(end, el), fine_el);
end

A = []; coeff_plot = []; u_norm = [];
for el = 1:n
    for i = 1:(p + 1)
        A(i, 1:(p + 1)) = x(i, el) .^ ((1:(p + 1)) - 1);
    end

    coeff_plot(:, el) = A \ u(:, el);

    for i = 1:length(x_norm(:, 1))
        u_norm(i, el) = sum(coeff_plot(:, el)' .* (x_norm(i, el) .^ ((1:(p + 1)) - 1))
            ↪ );
    end
end

% u_norm : DG solution
% u_exact_norm : exact solution in a DG basis
% u_exact_n : exact solution in a continuous basis
u_exact_n = uexact;
u_exact_norm = [];

i = 1;
j = fine_el;
for el = 1:n
    u_exact_norm(:, el) = u_exact_n(i:j)';
    i = i + fine_el - 1;
    j = j + fine_el - 1;
end

% L-2 norm for each element is determined using trapz() at the last time
% step only (follows the original code)
L2_elem = [];
for el = 1:n
    L2_elem(el) = trapz(x_norm(:, el), (u_exact_norm(:, el)' - u_norm(:, el)') .^ 2);
end

error = sqrt(sum(L2_elem));
```

```
end
```

3.2.2 rhsdiff.m

```
function [r] = rhs(u, Kel, Mel, k)
    % solve for sigma
    r = - Kel * u;
    r(end, :) = r(end, :) + u(1, [2:end, 1]);   % right-end flux
    r(1, :) = r(1, :) - u(1, :);                % left-end flux
    sigma = Mel \ r;

    % solve for u
    r = Kel * (u - k * sigma);                                  % Integral
    r(end, :) = r(end, :) - (u(end, :) - k * sigma(end, :));    % right flux
    r(1, :) = r(1, :) + (u(end, [end, 1:end-1]) - ...
                k * sigma(end, [end, 1:end-1]));                % left flux
    r = Mel \ r;
end
```

3.2.3 dgconvdiff_convergence.m

```
function [errors, slopes] = dgconvect_convergence
clear all
T = 1;                       % end simulation time
dt = 2e-4;                   % time step size
P = [1, 2, 4, 8, 16];        % polynomial orders for convergence study
N = [16, 32, 64, 128, 256]; % numbers of nodes
k = 1e-3;                    % thermal conductivity

errors = zeros(length(P), length(N));

for p = 1:length(P)
    en = 1;
    for n = N ./ P(p)
        [u, errors(p, en)] = dgconvdiff(n, p, T, dt, k);
        en = en + 1;
    end
end

slopes = [];
leg = cell(1, length(P)); leg_text = ''; % to hold the legend labels

for p = 1:length(P)
    p_error = errors(p, :);
    fit = polyfit(log(N), log(p_error), 1);
    slopes(p) = fit(1);
    loglog(N, p_error, '*-')
    leg{p} = sprintf('p␣=␣%i,␣slope␣=␣%.2i', P(p), slopes(p));
    hold on

    if p == length(P)
        leg_text = strcat(leg_text, sprintf('␣leg{%i}', p));
    else
        leg_text = strcat(leg_text, sprintf('␣leg{%i},', p));
    end
```

```
end

leg_text = strcat(strcat('legend(', leg_text), ')');
eval(leg_text)
xlabel('log(number␣of␣nodes)')
ylabel('log(L2-norm)')

end
```