Math 597

Project #2

Damyn Chipman

Introduction

Now that we have the ability to interpolate and integrate in 1D, we will now look at solving a 1D Wave Equation using CG and DG Galerkin Methods.

In this project and writeup, we will look at how to go from the 1D wave equation to the linear systems we need to form in order to solve PDE. We'll look at the algorithms and code to generate the necessary matrices and vectors. And finally, we'll do a convergance study of our implementation to check how well it converges.

Again, the code for this project is housed on GitHub on the HydroForest repository.

Problem Statement

We will be solving the 1D wave equation, expressed as:

$$PDE: \ \frac{\partial q(x,t)}{\partial t} + \frac{\partial f(q(x,t))}{\partial x} = 0, \qquad \forall x \in \Omega = [-1,1] \tag{1}$$

$$IC: \ q(x,0) = e^{-64x^2} \tag{2}$$

$$BC: \ q(-1,t) = q(1,t) \tag{3}$$

with $f(q(x,t)) = uq(x,t)$, $u =$constant = 2, and periodic boundary conditions. We will run from $t = 0$ to $t = 1$, at which point the wave will have traveled around the domain and returned to the original starting position. This allows us to check the error compared to the exact solution at that time.

From PDE to Element Linear Systems

We'll walk through pieces of the derivation, with the reasoning and details provided in the text and class notes. We will do this for both CG and DG schemes at their point of

convergence.

We start by expressing q and f in terms of an expansion of basis functions:

$$q(x,t) \approx q_N^{(e)} = \sum_{j=0}^{N} \psi_j(x) q_j^{(e)}(t) \tag{4}$$

$$f(x,t) \approx f_N^{(e)} = f(q_N^{(e)}) \tag{5}$$

Now, multiply our PDE by a test function that is the same as our basis function (the Galerkin principle) and integrate over an individual element:

$$\int_{\Omega_e} \psi_i \frac{\partial q_N^{(e)}}{\partial t} d\Omega_e + \int_{\Omega_e} \psi_i \frac{\partial f_N^{(e)}}{\partial x} d\Omega_e = 0 \tag{6}$$

At this point, CG and DG diverge in approach, so we look at each individually.

Continuous Galerkin (CG) Method

With the CG method, we allow the function value at interfaces between elements to be continuous. From the equation above, we plug in our basis function expansion and rearrange:

$$\sum_{j=0}^{N} \frac{\partial q_j^{(e)}}{\partial t} \int_{\Omega_e} \psi_i(x) \psi_j(x) d\Omega_e + \sum_{j=0}^{N} f_j^{(e)} \int_{\Omega_e} \psi_i(x) \frac{\partial \psi_j(x)}{\partial x} d\Omega_e = 0 \tag{7}$$

Now we map from the physical element to the reference element:

$$\sum_{j=0}^{N} \frac{\partial q_j^{(e)}}{\partial t} \int_{\hat{\Omega}} \psi_i(\xi) \psi_j(\xi) d\hat{\Omega} + \sum_{j=0}^{N} f_j^{(e)} \int_{\hat{\Omega}} \psi_i(\xi) \frac{\partial \psi_j(\xi)}{\partial \xi} \frac{dx}{d\xi} d\hat{\Omega} = 0 \tag{8}$$

where our metric term $\frac{dx}{d\xi} = \frac{\Delta x^{(x)}}{2}$. We can write this in matrix form as follows:

$$M_{ij}^{(e)} \frac{\partial q_j^{(e)}}{\partial t} + D_{ij}^{(e)} f_j^{(e)} = 0 \tag{9}$$

where, using Guassian quadrature, we can express the matrices as:

$$M_{ij}^{(e)} = \int_{\hat{\Omega}} \psi_i(\xi)\psi_j(\xi) d\hat{\Omega} = \sum_{k=0}^{Q} w_k \psi_i(\xi_k)\psi_j(\xi_k) \tag{10}$$

$$D_{ij}^{(e)} = \int_{\hat{\Omega}} \psi_i(\xi) \frac{\partial \psi_j(\xi)}{\partial \xi} \frac{dx}{d\xi} d\hat{\Omega} = \sum_{k=0}^{Q} w_k \psi_i(\xi_k) \frac{\partial \psi_j(\xi_k)}{\partial x} \tag{11}$$

We can get either exact or inexact quadrature based on the number of quadrature points we use: For exact, we can use $Q = N + 1$ and for inexact, we can use $Q = N$.

Discontinuous Galerkin (DG) Method

In DG methods, we do not restrict the function value to at the interfaces between elements to be the same. We handle that information via a numerical flux that we will introduce.

Continuing from our divergance point, we use the product rule to rewrite the second term (with the flux)

$$\int_{\Omega_e} \psi_i \frac{\partial q_N^{(e)}}{\partial t} d\Omega_e - \int_{\Omega_e} \frac{\partial \psi_i}{\partial x} f_N^{(e)} d\Omega_e + \int_{\Omega_e} \frac{d}{dx}[\psi_i f_N^{(*,e)}] d\Omega_e = 0 \tag{12}$$

$$\int_{\Omega_e} \psi_i \frac{\partial q_N^{(e)}}{\partial t} d\Omega_e - \int_{\Omega_e} \frac{\partial \psi_i}{\partial x} f_N^{(e)} d\Omega_e + \left[\hat{n}^{(e)} \psi_i f_N^{(*,e)}\right]\Big|_{\Gamma_e} = 0 \tag{13}$$

transfering a derivative onto the basis function. We have also used the Fundamental Theorem of Calculus to change the term with the total derivative into a boundary integral (or just the integrand at the end points in 1D). We introduce a numerical flux f^* that we will look at shortly.

Now we input our basis expansion for $q_N^{(e)}$ and $f_N^{(e)}$ and rearrange:

$$\sum_{j=0}^{N} \frac{\partial q_j^{(e)}}{\partial t} \int_{\Omega_e} \psi_i(x)\psi_j(x) d\Omega_e - \sum_{j=0}^{N} f_N^{(e)} \int_{\Omega_e} \frac{\partial \psi_i(x)}{\partial x} \psi_j(x) d\Omega_e + \left[\hat{n}^{(e)} \psi_i(x)\psi_j(x)\right.$$

Next, we do our mapping from the physical to the reference element:

$$\sum_{j=0}^{N} \frac{\partial q_j^{(e)}}{\partial t} \int_{\hat{\Omega}} \psi_i(\xi)\psi_j(\xi) d\hat{\Omega} - \sum_{j=0}^{N} f_N^{(e)} \int_{\hat{\Omega}} \frac{\partial \psi_i(\xi)}{\partial \xi} \psi_j(\xi) \frac{dx}{d\xi} d\hat{\Omega} + \left[\hat{n}^{(e)} \psi_i(\xi)\psi_j(\xi)\right]$$

which we can write as a linear system as follows:

$$M_{ij}^{(e)} \frac{\partial q_j^{(e)}}{\partial t} - \tilde{D}_{ij}^{(e)} f_j^{(e)} + F_{ij}^{(e)} f_j^{(*,e)} = 0 \tag{16}$$

Again using Gaussian quadrature, we can write these matrices as:

$$M_{ij}^{(e)} = \int_{\hat{\Omega}} \psi_i(\xi)\psi_j(\xi) d\hat{\Omega} = \sum_{k=0}^{Q} w_k \psi_i(\xi_k)\psi_j(\xi_k) \tag{17}$$

$$\tilde{D}_{ij}^{(e)} = (D_{ij}^{(e)})^T = \int_{\hat{\Omega}} \frac{\partial \psi_i(\xi)}{\partial \xi} \psi_j(\xi) \frac{dx}{d\xi} d\hat{\Omega} = \sum_{k=0}^{Q} w_k \frac{\partial \psi_i(\xi_k)}{\partial x} \psi_j(\xi_k) \tag{18}$$

$$F_{ij}^{(e)} = \left[\psi_i(\xi)\psi_j(\xi)\right]\Big|_{-1}^{1} \tag{19}$$

We need to address the numerical flux f^*. We introduce a numerical flux between elements that accounts for the discontinuity between the solutions at the interface point. We will be using the classic Rusanov flux:

$$f^{(*,e,k)} = \frac{1}{2}\left[f^{(e)} + f^{(k)} - \hat{n}^{(e,k)} |\lambda_{max}| \left(q^{(k)} - q^{(e)}\right)\right] \tag{20}$$

The superscripts denote element e and neighbors k. For the 1D wave equation, $\lambda_{max} = u$.

From Element Linear System to Global Linear System

The linear systems derived above are for a single element. We need to combine each local element matrix to form a global linear system that we can solve. We do this through Direct Stiffness Summation, or DSS. The process is similar for CG and DG, with the difference being the global indexing system.

In CG, we use a global numbering system that counts the interface points only once, as the solution is continuous there. In DG, the global numbering system counts the interface points twice, once for each element, as each is a different degree of freedom. We form ID matrices that take the following form in CG and DG to help us index from local to global and vice versa:

$$ID_{CG} = \begin{bmatrix} 0 & N & 2N & \dots & (N_e-1)N \\ \vdots & \vdots & \vdots & \dots & \vdots \\ N-1 & 2N-1 & 3N-1 & \dots & N_eN \end{bmatrix} \tag{21}$$

$$ID_{DG} = \begin{bmatrix} 0 & N-1 & 2N-1 & \dots & (N_e-1)N-1 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ N-1 & 2N-1 & 3N-1 & \dots & N_eN+N_e \end{bmatrix} \tag{22}$$

In general, the DSS operation takes the local element matrices and combines them into global matrices. Formally, we can express the DSS operation as:

$$M_{IJ} = \bigwedge_{e=0}^{N_e-1} M_{ij}^{(e)} \tag{23}$$

We'll detail the DSS operation for both CG and DG through the code below. The DSS operation is designed as a `struct` that takes as input a list of elements (each with information about the element's grid, quadrature grid, physical bounds, data vectors, etc.) and the ID matrix detailed above. When we DSS an element matrix, we can choose to include the metric term from the physical to reference domain mapping.

CG DSS

The points at the element interfaces need to be common points, so when we combine the local element matrices, the interface point gets added for both the left and right element matrices. We also need to account for the metric term in the mass matrix from our mapping. Thus, there are functions to do DSS with or without a metric term. Below is the code for the CG DSS operations, included in `CGFiniteElements.hpp`. For CG DSS, we follow Algorithm 5.4 for matrices and 5.7 for vectors.

```
template<typename NumericalType>
struct CGDirectStiffnessSummationOperator {

    std::vector<Element1D<NumericalType>>& elements;
    Matrix<int>& IDMatrix;


CGDirectStiffnessSummationOperator(std::vector<Element1D<NumericalType
 elements, Matrix<int>& IDMatrix) :
        elements(elements), IDMatrix(IDMatrix)
            {}

    Matrix<NumericalType> operate(Matrix<NumericalType>&
```

```
elementMatrix) {
        int nElements = (int) elements.size();
        int nOrder = (int) IDMatrix.nRows() - 1;
        int nPoints = nElements * nOrder + 1;

        Matrix<NumericalType> M(nPoints, nPoints, 0);

        for (auto e = 0; e < nElements; e++) {
            for (auto j = 0; j < nOrder + 1; j++) {
                int J = IDMatrix(j, e);
                for (auto i = 0; i < nOrder + 1; i++) {
                    int I = IDMatrix(i, e);
                    M(I,J) += elementMatrix(i,j);
                }
            }
        }

        return M;
    }

    Vector<NumericalType> operate(Vector<NumericalType>&
elementVector) {
        int nElements = (int) elements.size();
        int nOrder = (int) IDMatrix.nRows() - 1;
        int nPoints = nElements * nOrder + 1;

        Vector<NumericalType> r(nPoints, 0);

        for (auto e = 0; e < nElements; e++) {
            for (auto i = 0; i < nOrder; i++) {
                int I = IDMatrix(i,e);
                r[I] += elementVector[i];
            }
        }

        return r;
    }

    Matrix<NumericalType>
operateWithMetricTerm(Matrix<NumericalType>& elementMatrix) {
        int nElements = (int) elements.size();
        int nOrder = (int) IDMatrix.nRows() - 1;
        int nPoints = nElements * nOrder + 1;

        Matrix<NumericalType> M(nPoints, nPoints, 0);

        for (auto e = 0; e < nElements; e++) {
```

```
            double dx = elements[e].xUpper() -
elements[e].xLower();
            for (auto j = 0; j < nOrder + 1; j++) {
                int J = IDMatrix(j, e);
                for (auto i = 0; i < nOrder + 1; i++) {
                    int I = IDMatrix(i, e);
                    M(I,J) += (dx/2.0)*elementMatrix(i,j);
                }
            }
        }

        return M;
    }

    Vector<NumericalType>
operateWithMetricTerm(Vector<NumericalType>& elementVector) {
        int nElements = (int) elements.size();
        int nOrder = (int) IDMatrix.nRows() - 1;
        int nPoints = nElements * nOrder + 1;

        Vector<NumericalType> r(nPoints, 0);

        for (auto e = 0; e < nElements; e++) {
            double dx = elements[e].xUpper() -
elements[e].xLower();
            for (auto i = 0; i < nOrder; i++) {
                int I = IDMatrix(i,e);
                r[I] += (dx/2.0)*elementVector[i];
            }
        }

        return r;
    }

};
```

CG Global Linear System

Doing the DSS operation on the element matrices gives us the following global system:

$$M_{IJ}\frac{\partial q_J}{\partial t} + D_{IJ} f_J = 0 \tag{24}$$

where I and J denote our global numbering system.

DG DSS

In DG, the interface point is counted twice, once for each sharing element. This means the DSS operation is just a block diagonal approach. It is designed similiar to the CG DSS, but we only have need to operate on matrices. Below is the code for the DG DSS included in `DGFiniteElements.hpp`:

```
template<typename NumericalType>
struct DGDirectStiffnessSummationOperator {

    std::vector<Element1D<NumericalType>>& elements;
    Matrix<int>& IDMatrix;


DGDirectStiffnessSummationOperator(std::vector<Element1D<NumericalType
 elements, Matrix<int>& IDMatrix) :
        elements(elements), IDMatrix(IDMatrix)
            {}

    Matrix<NumericalType> operate(Matrix<NumericalType>&
elementMatrix) {
        int nElements = (int) elements.size();
        std::vector<Matrix<NumericalType>> diag(nElements);
        for (auto e = 0; e < nElements; e++) {
            diag[e] = elementMatrix;
        }
        return blockDiagonalMatrix(diag);
    }

    Matrix<NumericalType>
operateWithMetricTerm(Matrix<NumericalType>& elementMatrix) {
        int nElements = (int) elements.size();
        std::vector<Matrix<NumericalType>> diag(nElements);
        for (auto e = 0; e < nElements; e++) {
            diag[e] = elementMatrix;
            double dx = elements[e].xUpper() -
elements[e].xLower();
            diag[e] *= (dx/2.0);
        }
        return blockDiagonalMatrix(diag);
    }

};
```

DG Global Linear System

Doing the DSS operation on the DG element linear system gives us the following global system:

$$M_{IJ}\frac{\partial q_J}{\partial t} - \tilde{D}_{IJ}f_J + F_{IJ}f_J^* = 0. \tag{25}$$

We need to look at how to form the global Rusanov flux vector. Using our global numbering system and ID_{DG}, we can iterate over the elements, and get the global ID for the element's right edge and the element's neighbor's left edge (i.e., the point in common). Then we compute the numerical flux, and assign the interface point on each element to that numerical flux. For this, we use Algorithm 6.6 and is implemented below:

```
template<typename NumericalType>
class DGGlobalRusanovFluxVector : public Vector<NumericalType> {

public:


DGGlobalRusanovFluxVector(std::vector<Element1D<NumericalType>>&
elements, Matrix<int>& IDMatrix, Vector<NumericalType>& q_global,
Vector<NumericalType>& f_global, NumericalType lambda) :
        Vector<NumericalType>(elements.size()*(IDMatrix.nRows()),
0) {

        int N = IDMatrix.nRows()-1;
        double schemeFlag = 1.0;
        for (auto e = 0; e < elements.size(); e++) {
            int L = e;
            int R = (e + 1) % elements.size();
            int I = IDMatrix(N, L);
            int J = IDMatrix(0, R);
            NumericalType f_star = 0.5*(f_global[I] + f_global[J]
- schemeFlag*lambda*(q_global[J] - q_global[I]));
            this->data_[I] = f_star;
            this->data_[J] = f_star;
        }

    }

};
```

Time Stepping

Now that we have a global linear system, we will solve for the time derivative and then time step with a time stepping scheme. For CG, we use Algorithm 5.5 as a guide, and for DG we use 6.7 as a guide.

Solving the CG global system yields:

$$\frac{\partial q_J}{\partial t} = (M_{IJ})^{-1} D_{IJ} f_J \tag{26}$$

And solving the DG global system yields:

$$\frac{\partial q_J}{\partial t} = (M_{IJ})^{-1} (\tilde{D}_{IJ} f_J - F_{IJ} f_J^*) \tag{27}$$

To time step, most schemes will work. We use a strong stability-preserving (SSP) RK3 method by Shu. For $\frac{dq}{dt} = R(q)$:

$$q^{(1)} = q^n + \Delta t R(q^n) \tag{28}$$
$$q^{(2)} = \frac{3}{4} q^n + \frac{1}{4} q^{(1)} + \frac{1}{4} \Delta t R(q^{(1)} \tag{29}$$
$$q^{n+1} = \frac{1}{3} q^n + \frac{2}{3} q^{(2)} + \frac{2}{3} \Delta t R(q^{(2)} \tag{30}$$

For this SSP RK3 method, the CFL number for stability is:

$$CFL = u \frac{\Delta t}{\Delta x} \leq \frac{1}{3} \tag{31}$$

The time stepping is designed as an abstract interface class that allows for expansion to other methods. It allows for a RHS function that takes q^n, a RHS matrix that multiplies q^n, or a vector already formed by the operation of $R(q)$. Each scheme allows for a suggested maximum time step, given a characteristic speed and characteristic length. This allows for an adaptive grid later on.

```
template<typename FloatingDataType>
class TimeIntegration {

public:

    virtual Vector<FloatingDataType> update(FloatingDataType t,
FloatingDataType dt, Vector<FloatingDataType> q_n,
Matrix<FloatingDataType> Rq_n) = 0;
```

```
    virtual Vector<FloatingDataType> update(FloatingDataType t,
FloatingDataType dt, Vector<FloatingDataType> q_n,
Vector<FloatingDataType> Rq_n) = 0;
    virtual Vector<FloatingDataType> update(FloatingDataType t,
FloatingDataType dt, Vector<FloatingDataType> q_n,
std::function<Vector<FloatingDataType>(Vector<FloatingDataType>)>
Rq_n) = 0;
    virtual FloatingDataType getCFLStabilityNumber() = 0;
    virtual FloatingDataType getMaxTimeStep(FloatingDataType
characteristicSpeed, FloatingDataType characteristicLength) = 0;

};

template<typename FloatingDataType>
class RungeKutta3 : public TimeIntegration<FloatingDataType> {

public:

    RungeKutta3() {}

    Vector<FloatingDataType> update(FloatingDataType t,
FloatingDataType dt, Vector<FloatingDataType> q_n,
Matrix<FloatingDataType> Rq_n) {

        Vector<FloatingDataType> q_temp;
        q_temp = Rq_n*q_n;
        Vector<FloatingDataType> q_1 = q_n + dt*q_temp;
        q_temp = Rq_n*q_1;
        Vector<FloatingDataType> q_2 = (3.0/4.0)*q_n +
(1.0/4.0)*q_1 + (1.0/4.0)*dt*q_temp;
        q_temp = Rq_n*q_2;
        Vector<FloatingDataType> q_np1 = (1.0/3.0)*q_n +
(2.0/3.0)*q_2 + (2.0/3.0)*dt*q_temp;
        return q_np1;

    }

    Vector<FloatingDataType> update(FloatingDataType t,
FloatingDataType dt, Vector<FloatingDataType> q_n,
std::function<Vector<FloatingDataType>(Vector<FloatingDataType>)>
Rq_n) {

        Vector<FloatingDataType> q_1 = q_n + dt*Rq_n(q_n);
        Vector<FloatingDataType> q_2 = (3.0/4.0)*q_n +
(1.0/4.0)*q_1 + (1.0/4.0)*dt*Rq_n(q_1);
        Vector<FloatingDataType> q_np1 = (1.0/3.0)*q_n +
(2.0/3.0)*q_2 + (2.0/3.0)*dt*Rq_n(q_2);
```

```
        return q_np1;

    }

    Vector<FloatingDataType> update(FloatingDataType t,
FloatingDataType dt, Vector<FloatingDataType> q_n,
Vector<FloatingDataType> Rq_n) {

        Vector<FloatingDataType> q_1 = q_n + dt*Rq_n;
        Vector<FloatingDataType> q_2 = (3.0/4.0)*q_n +
(1.0/4.0)*q_1 + (1.0/4.0)*dt*Rq_n;
        Vector<FloatingDataType> q_np1 = (1.0/3.0)*q_n +
(2.0/3.0)*q_2 + (2.0/3.0)*dt*Rq_n;
        return q_np1;

    }

    FloatingDataType getCFLStabilityNumber() { return cfl_; }
    FloatingDataType getMaxTimeStep(FloatingDataType
characteristicSpeed, FloatingDataType characteristicLength) {
        return (characteristicLength*cfl_) / characteristicSpeed;
    }

protected:

    FloatingDataType cfl_ = 1.0/3.0;

};
```

Results

Element Mesh

With the linear systems formed and a way to time step, we are ready to solve the problem at hand. The CG and DG versions are fairly similar. We first form the mesh made up of `Element1D`s. Each `Element1D` contains the reference element grid, a quadrature grid (with options for exact or inexact quadrature), the physical bounds, and storage for data matrices and vectors like the solution, flux, or other element matrices like mass or derivative matrices. The `Element1D` class has member functions for transforming from the local reference element to the global domain and vice versa.

The `ElementMesh1D` class contains a `std::vector<Element1D>` that has the main storage for the mesh. It also has utility functions like `setInitialCondition` and `plot`.

Each of these are implemented in `Element1D.hpp` and `Mesh1D.hpp`.

Advance Routines

The bulk of the work is done in the `advanceCG` and `advanceDG` routines provided in `examples/math597_P2/main.cpp`. These routines roughly follow Algorithms 5.5 for CG and 6.7 for DG.

Linear Element Matrices

As a sanity check, when we run `advanceCG` or `advanceDG` with $N = 1$ and $N_e \leq 4$, the element matrices are printed out for us to compare with the derivations in the book. After debugging, we can run and observe that the element matrices and the global matrices via DSS are, in fact, correct for these parameters:

```
--=== CG MATRICES ===--
nElements = 4  N = 1
M_ij =   [2 x 2]
      0.6667      0.3333
      0.3333      0.6667

D_ij =   [2 x 2]
       -0.5         0.5
       -0.5         0.5

ID =   [2 x 4]
           0           1           2           3
           1           2           3           0

M_IJ =   [5 x 5]
      0.3333     0.08333           0     0.08333           0
     0.08333      0.3333     0.08333           0           0
           0     0.08333      0.3333     0.08333           0
           0           0     0.08333      0.3333     0.08333
           0           0           0           0           1

D_IJ =   [5 x 5]
           0         0.5           0        -0.5           0
        -0.5           0         0.5           0           0
           0        -0.5           0         0.5           0
           0           0        -0.5           0         0.5
           0           0           0           0           0

Dhat =   [5 x 5]
      0.9231       2.769           0      -2.769     -0.9231
      -3.462      0.1154           3     -0.1154      0.4615
      0.9231      -3.231           0       3.231     -0.9231
     -0.2308      0.8077          -3     -0.8077       3.231
           0           0           0           0           0

--=== DG MATRICES ===--
nElements = 4  N = 1
M_ij =   [2 x 2]
      0.6667      0.3333
      0.3333      0.6667

D_ij =   [2 x 2]
       -0.5        -0.5
        0.5         0.5

F_ij =   [2 x 2]
```

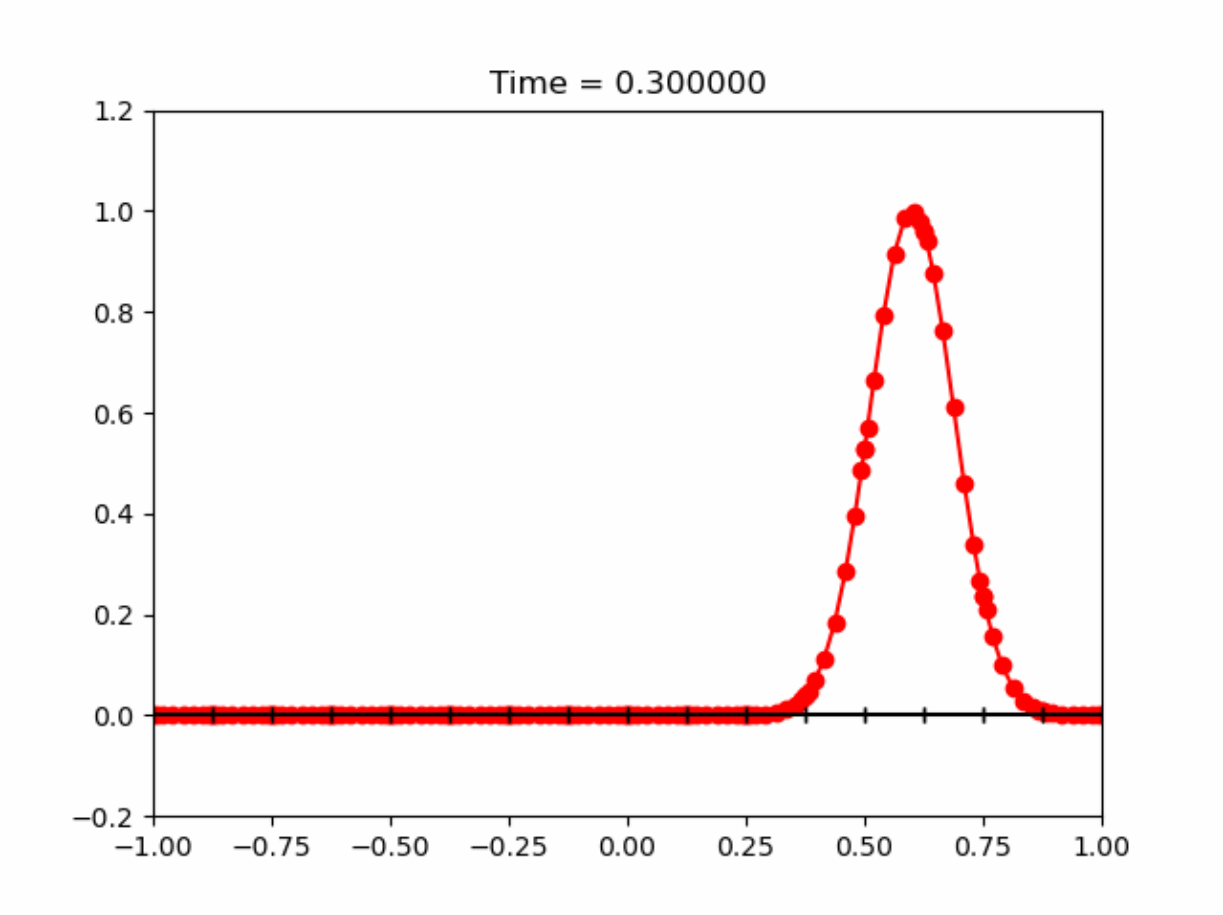
```
          -1           0
           0           1

ID =   [2 x 4]
           0           2           4           6
           1           3           5           7

M_IJ =   [8 x 8]
      0.1667     0.08333           0           0           0
0           0           0
     0.08333      0.1667           0           0           0
0           0           0
           0           0      0.1667     0.08333           0
0           0           0
           0           0     0.08333      0.1667           0
0           0           0
           0           0           0           0      0.1667
0.08333           0           0
           0           0           0           0     0.08333
0.1667           0           0
           0           0           0           0           0
0      0.1667     0.08333
           0           0           0           0           0
0     0.08333      0.1667

D_IJ =   [8 x 8]
        -0.5        -0.5           0           0           0
0           0           0
         0.5         0.5           0           0           0
0           0           0
           0           0        -0.5        -0.5           0
0           0           0
           0           0         0.5         0.5           0
0           0           0
           0           0           0           0        -0.5
-0.5           0           0
           0           0           0           0         0.5
0.5           0           0
           0           0           0           0           0
0        -0.5        -0.5
           0           0           0           0           0
0         0.5         0.5

F_IJ =   [8 x 8]
          -1           0           0           0           0
0           0           0
           0           1           0           0           0
```

```
0           0           0
           0           0          -1           0           0
0           0           0
           0           0           0           1           0
0           0           0
           0           0           0           0          -1
0           0           0
           0           0           0           0           0
1           0           0
           0           0           0           0           0
0          -1           0
           0           0           0           0           0
0           0           1

Dhat =    [8 x 8]
          -6          -6           0           0           0
0           0           0
           6           6           0           0           0
0           0           0
           0           0          -6          -6           0
0           0           0
           0           0           6           6           0
0           0           0
           0           0           0           0          -6
-6            0           0
           0           0           0           0           6
6           0           0
           0           0           0           0           0
0          -6          -6
           0           0           0           0           0
0           6           6

Fhat =    [8 x 8]
          -8          -4           0           0           0
0           0           0
           4           8           0           0           0
0           0           0
           0           0          -8          -4           0
0           0           0
           0           0           4           8           0
0           0           0
           0           0           0           0          -8
-4            0           0
           0           0           0           0           4
8           0           0
           0           0           0           0           0
0          -8          -4
```

```
          0          0          0          0          0
0          4          8
```

Visualization

We can run `advanceCG` or `advanceDG` with a `plotFlag` to turn on plotting at designated time steps. Saving each plot and converting it to an animation gives us the following satisfying visualization:



Or it can be viewed on Github here.

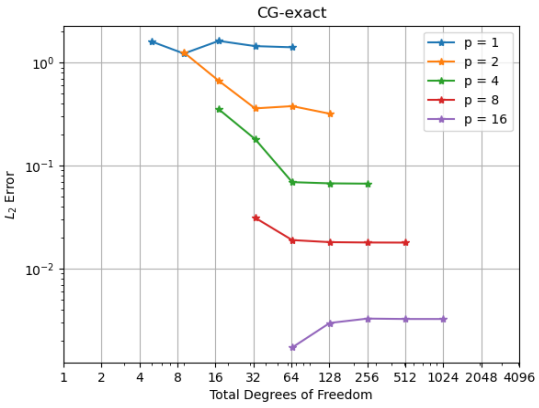
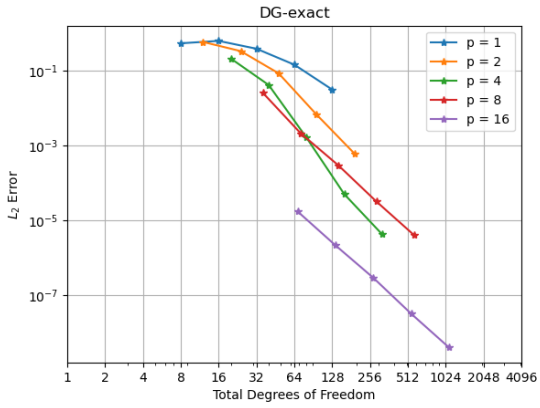
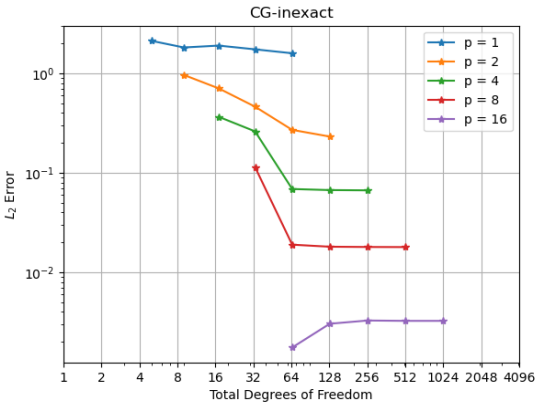
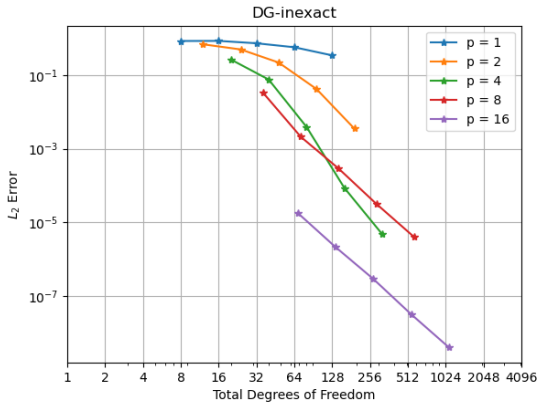
Convergance Study

The `main` program in `examples/math597-P2/main.cpp` runs a convergance study for the following parameter sweeps:

- Scheme : CG, DG
- Integration Method : Exact, Inexact
- Basis Order : 1, 2, 4, 8, 16
- Number of Elements : 4, 8, 16, 32, 64

This is done with MPI on 4 ranks with each rank taking one of the scheme/integration method combos. Running it produces plots of the number of degrees of freedom vs. the L_2 error compared to the exact solution, which is the initial solution advected around the domain once.

The plots are shown below:

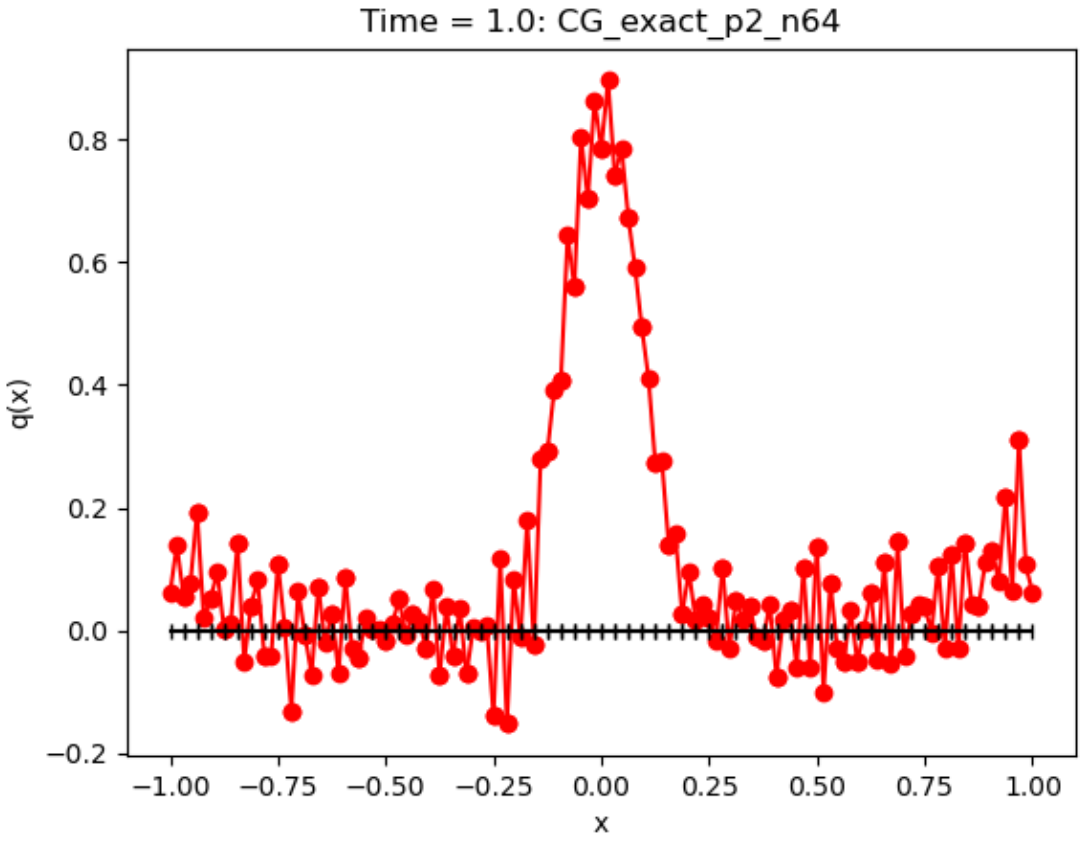
CG	DG
	
	

Discussion

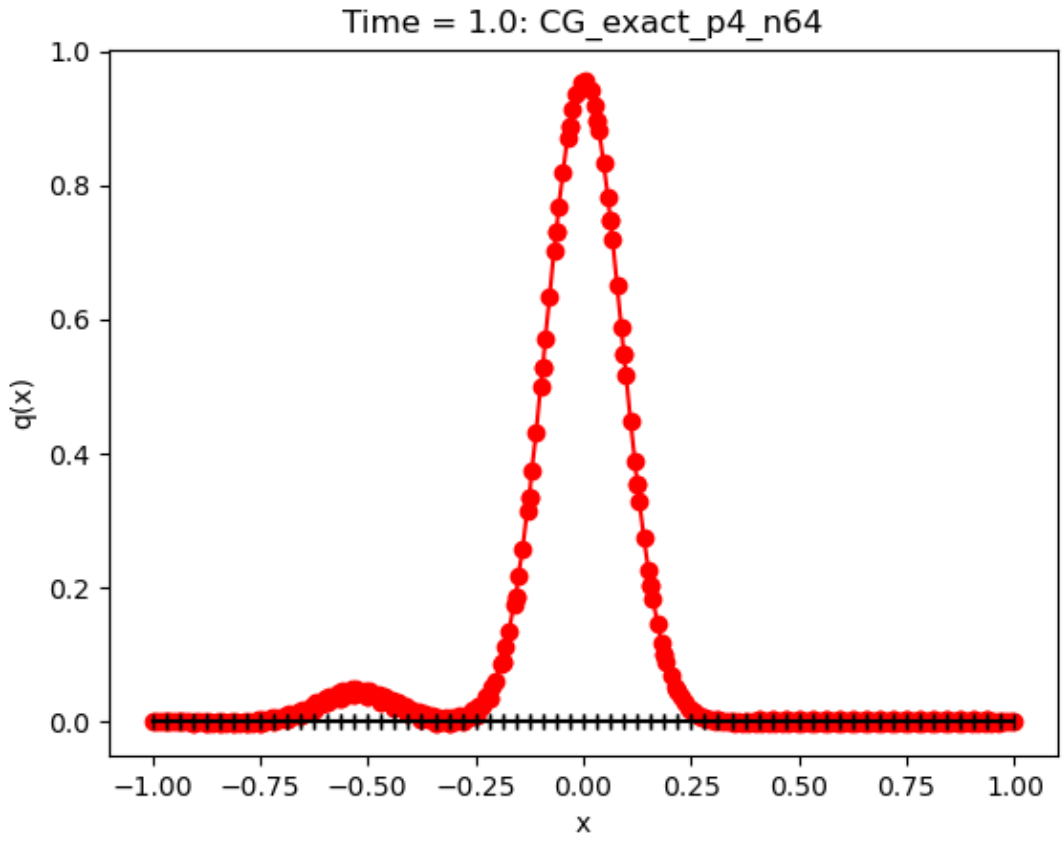
Now let's discuss these results. We'll mainly be looking at the error performance, as timing or memory weren't yet considered in these implementations. However, more efficient and more performant approaches certainly exists!

CG vs. DG

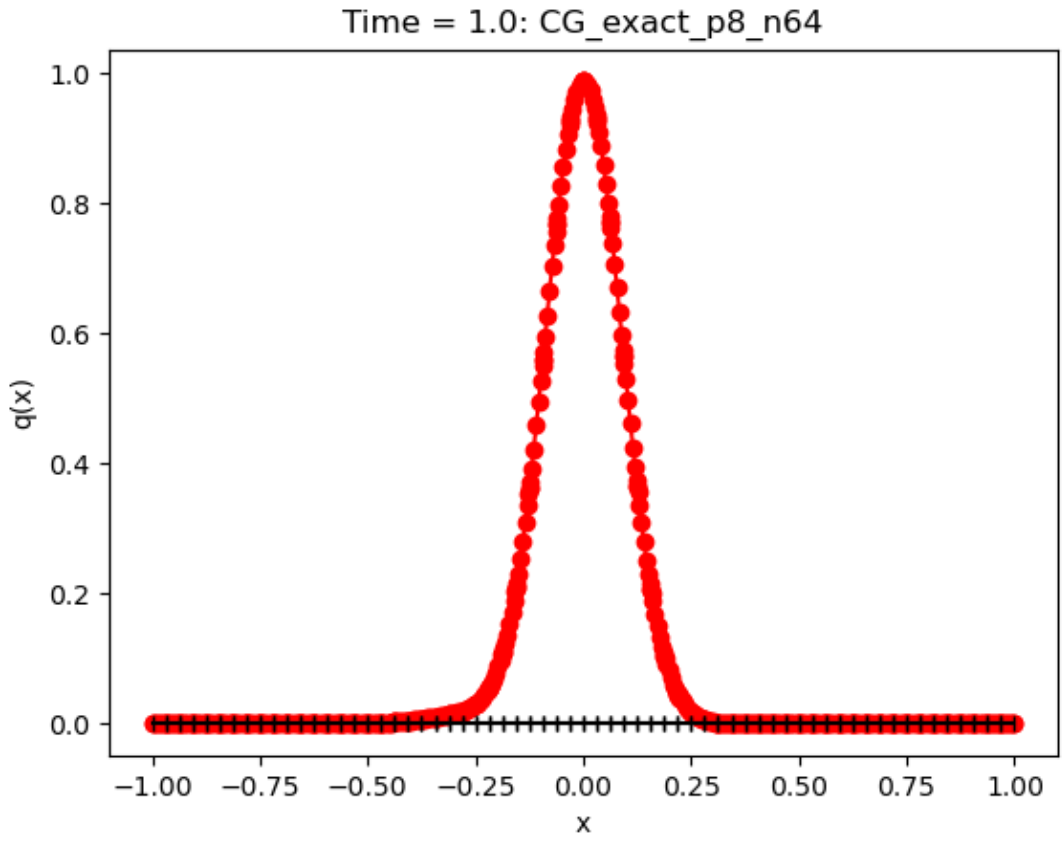
When time stepping, I found the CG results to be highly sensitive to stability issues. Even at higher resolutions and with a generously small time step, most CG results suffered from instabilities. For example, look at the following plot of a CG solution at the final time:



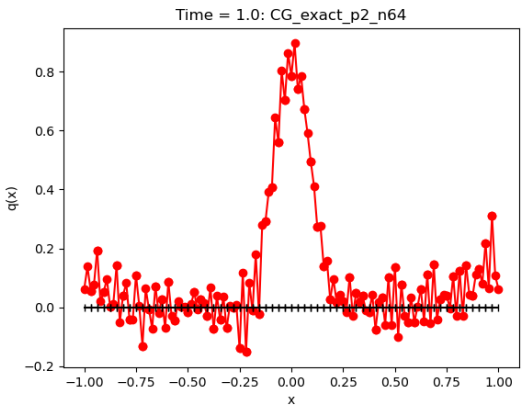
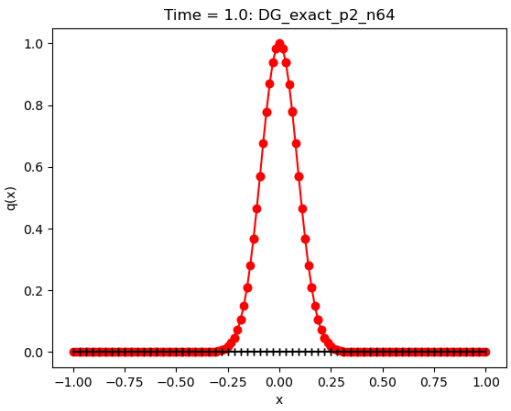
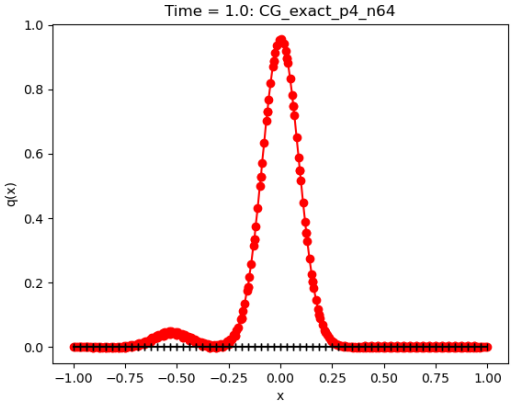
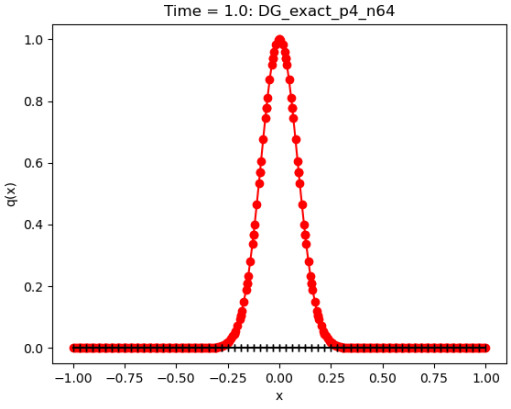
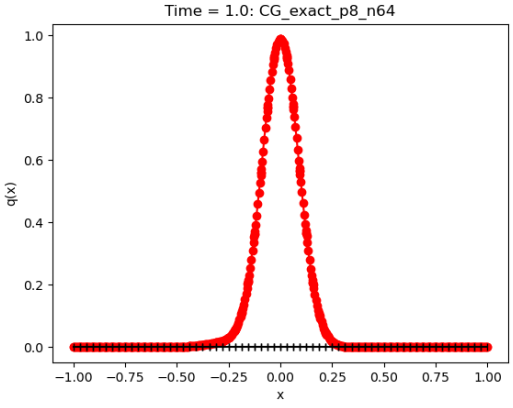
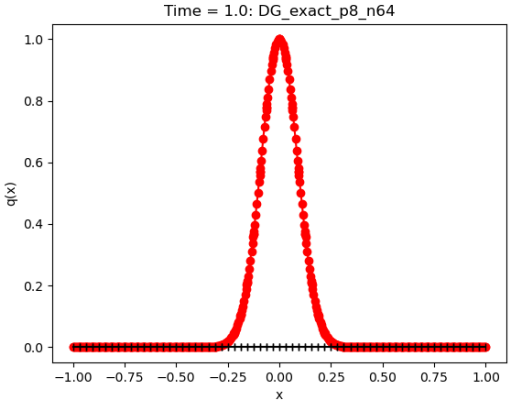
With many elements and second order basis functions, we get instabilities in the form of oscillations between elements. Increasing the polynomial order helps:



But we still have byproducts of the instabilities persist. It's only at even higher orders that we begin to see these instabilities die down:



Contrast that to the DG solutions at the same number of elements and same orders:

CG	DG
	
	
	

The instabilities are near non-existant for the DG versions. This is likely due to the allowing of discontinuities across element boundaries; the solution is allowed to have jumps. And the size of the solution discontinuity shrinks as you increase the number of degrees of freedom, nearing continuity and converging to the exact solution.

This can also be seen in the convergance plots, as the error for similar degrees of freedom is much lower.

Inexact vs Exact Integration

I can see very little difference between exact vs. inexact integration between the CG and DG methods. They are different, and this can be seen when viewing the matrices formed by the exact vs. inexact quadrature. The inexact quadrature results in purely diagonal matrices for mass and derivative matrices, both local and global, whereas the matrices formed from exact quadrature are block diagonal. I did not time or do anything to take advantage of the structure of these matrices (my linear algebra classes use a dense implementation and are wrappers around `LAPACK`). That being said, in future versions, I could take advantage of this structure to hopefully achieve some speedup.

Implementation

Once the bugs were worked out, there is a simplicity and beauty to how these Galerkin methods work. They allow for some nice, clean, and robust code. I've worked with other FE packages (MOOSE, MFEM, some other LLNL codes), and find them to be well-designed and easy to go from math to code... once the math is correct! Even my code, which tends to be naive to larger applications, seems to have a flow to it. At least as the developer it seems that way! I'm looking forward to future development!

Conclusion

We explored the 1D wave equation by solving it via CG and DG Galerkin methods. For both CG and DG, a linear system is formed first on the reference element, and then on the global domain. The solution is advanced in time via a SSB RK3 method. We looked at a convergence study and compared the CG and DG methods, with both exact and inexact integration.

References

Giraldo, Francis X. *An Introduction to Element-Based Galerkin Methods on Tensor-Product Bases: Analysis, Algorithms, and Applications*. Vol. 24. Springer Nature, 2020.