Math 597

Project #1

Damyn Chipman

Introduction

When working with a finite element method, the operations of interpolation, integration, and differentiation need to be quick, efficient, and practical. To accomodate this, we use polynomial basis functions and nodal basis points generated from polynomials. There is solid mathematical theory to motivate the use of polynomials (and special types of polynomials) for these kinds of operations.

In this project and write up, we will explore how to use polynomial basis functions and points to interpolate, integrate, and differentiate a simple function. We will look at how well different combinations of polynomial basis functions and points do for each of these operations.

The code for this project is housed on GitHub on the HydroForest repository. Eventually, this will house the finite element code worked on as part of this course. Written in C++, instructions for configuring, building, and installing the code are provided there. For this write up, I will provide code snippets for context.

Problem 1

The goal of this problem is to interpolate the function

$$f(x) = \cos(\frac{\pi}{2}x) \tag{1}$$

using Lagrange polynomials as the nodal basis functions with the following as choices for nodal basis points:

1. Equally spaced points
2. Legendre-Gauss points
3. Legendre-Gauss-Lobatto points

4. (Extra) Chebyshev points.

We'll start by looking at the polynomials we will use (Lagrange and Legendre) and how to generate them.

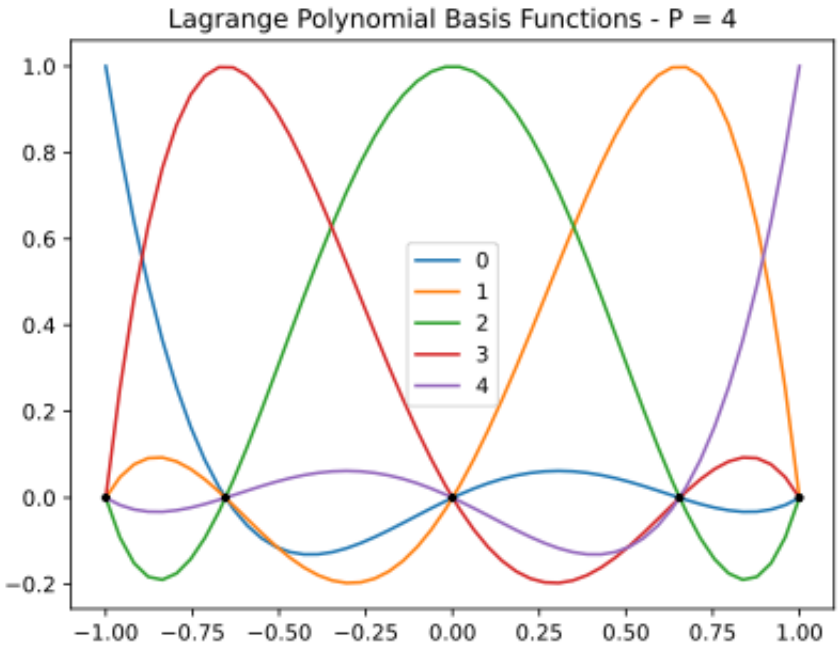
Nodal Basis Functions

Lagrange Polynomials

Lagrange polynomials are defined as:

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N} \frac{x - x_j}{x_i - x_j} \tag{2}$$

Lagrange polynomials require a set of points that act as basis points. If we have N points, then we get a set of N polynomials that are of order $p = N - 1$. For a set of Lobatto points and $p = 4$ (or $N = 5$), this looks like the following:



These kinds of polynomials have useful properties for our interpolation, integration, and differentiation operations. For example, they have a property called "partition of unity", which means that at each basis point, only one of the N polynomials is equal to one, with all others equal to zero (you can check this at each point in the figure above). This is useful for integration via quadrature because it allows us to use quadrature weights for

integration (more on that in problem 3).

In addition, there is a closed form of the derivative of Lagrange polynomials given by:

$$\frac{dL_i(x)}{dx} = \sum_{\substack{k=0 \\ k\neq i}}^{N} \left(\frac{1}{x_i - x_k}\right) \prod_{\substack{j=0 \\ j\neq i \\ j\neq k}}^{N} \frac{x - x_j}{x_i - x_j} \tag{3}$$

This will be useful for later when computing derivatives of a function.

Finally, let's address how to generate Lagrange polynomials. Each polynomial is different based on the basis points and sample points. As such, we'll write an algorithm that generates a matrix. The purpose of this matrix is to operate on a vector of function values to compute a function approximation. For the sake of completeness, we'll also do the same for the Lagrange polynomial derivatives:

```
struct LagrangePolynomial {

    int order;
    Vector<double> nodalPoints;
    LagrangePolynomial(Vector<double> nodalPoints) :
order(nodalPoints.size()), nodalPoints(nodalPoints) {}

    Matrix<double> operator()(Vector<double> xSample) {
        std::size_t Q = xSample.size();
        std::size_t N = nodalPoints.size();
        Matrix<double> L_il(N, Q);
        for (auto l = 0; l < Q; l++) {
            double x_l = xSample[l];
            for (auto i = 0; i < N; i++) {
                double x_i = nodalPoints[i];
                L_il(i, l) = 1.0;
                for (auto j = 0; j < N; j++) {
                    if (i != j) {
                        double x_j = nodalPoints[j];
                        L_il(i,l) *= (x_l - x_j) / (x_i - x_j);
                    }
                }
            }
        }
        return L_il;
    }

    Matrix<double> derivative(Vector<double> xSample) {
        std::size_t Q = xSample.size();
```

```
            std::size_t N = nodalPoints.size();
            Matrix<double> dL_il(N, Q);
            for (auto l = 0; l < Q; l++) {
                double x_l = xSample[l];
                for (auto i = 0; i < N; i++) {
                    double x_i = nodalPoints[i];
                    for (auto j = 0; j < N; j++) {
                        double x_j = nodalPoints[j];
                        double prod = 1.0;
                        if (j != i) {
                            for (auto k = 0; k < N; k++) {
                                double x_k = nodalPoints[k];
                                if (k != i && k != j) {
                                    prod *= (x_l - x_k) / (x_i - x_k);
                                }
                            }
                            dL_il(i,l) += prod / (x_i - x_j);
                        }
                    }
                }
            }
            return dL_il;
        }

};
```

When we create an instance of a `LagrangePolynomial`, we provide the nodal basis points (uniform, Legendre, Lobatto, Chebyshev, etc.). When we wish to compute the interpolation matrix, we provide the sample points (plotting points, the same nodal points, etc.). The same can be said for the derivative.

Legendre Polynomials

Another useful set of polynomials are the Legendre polynomials. They are generated from the following recurrence relation:

$$\phi_0^{Leg}(x) = 1 \tag{4}$$
$$\phi_1^{Leg}(x) = x \tag{5}$$
$$\phi_N^{Leg}(x) = \frac{2N-1}{N} x \phi_{N-1}^{Leg}(x) - \frac{N-1}{N} \phi_{N-2}^{Leg}(x), \forall N \geq 2 \tag{6}$$

Legendre polynomials, and their roots, form excellent sets of basis points for use with the operations we will need to do.

To generate them, we use the following algorithm and `struct`:

```
struct LegendrePolynomial {

    int order = 0;
    LegendrePolynomial(int order) : order(order) {}

    double operator()(double x) {
        double l1 = 0.0;
        double l0 = 1.0;
        for (int i = 1; i <= order; i++) {
            double ii = (double) i;
            double l2 = l1;
            double a = (2.0*ii - 1.0) / ii;
            double b = (ii - 1.0) / ii;
            l1 = l0;
            l0 = a*x*l1 - b*l2;
        }
        return l0;
    }

    Vector<double> operator()(Vector<double> x) {
        Vector<double> f(x.size());
        for (int l = 0; l < x.size(); l++) {
            f[l] = operator()(x[l]);
        }
        return f;
    }

    Vector<double> derivatives012(double x) {
        double l1 = 0.0; double l1_1 = 0.0; double l1_2 = 0.0;
        double l0 = 1.0; double l0_1 = 0.0; double l0_2 = 0.0;
        for (int i = 1; i <= order; i++) {
            double ii = (double) i;
            double l2 = l1; double l2_1 = l1_1; double l2_2 =
l1_2;
            double a = (2.0*ii - 1.0) / ii;
            double b = (ii - 1.0) / ii;
            l1 = l0;
            l1_1 = l0_1;
            l1_2 = l0_2;
            l0 = a*x*l1 - b*l2;
            l0_1 = a*(l1 + x*l1_1) - b*(l2_1);
            l0_2 = a*(2.0*l1_1 + x*l1_2) - b*l2_2;
        }
        return {l0, l0_1, l0_2};
    }

};
```

Next, let's look at each of the nodal basis points we will be evaluating.

Nodal Basis Points

In theory, any set of points can be used as a basis for interpolation and differentiation. But certain choices are better than others due to their specific properties.

Uniform Points

It doesn't get simplier than this... For a given domain $x \in [a, b]$, we pick N points and get the following set:

$$x_i = a + \Delta x i, i = 0, \dots, N-1 \tag{7}$$

where $\Delta x = \frac{b-a}{N-1}$.

Chebyshev Points

Included here for completeness, another common set of points in the Chebyshev points:

$$x_i = \cos\left(\left[\frac{2i+1}{2N+2}\right]\pi\right), i = 0, \dots, N \tag{8}$$

Legendre Points

Legendre points are generated from the roots of Legendre polynomials:

$$\phi_N^{Leg}(x) = \frac{2N-1}{N} x \phi_{N-1}^{Leg}(x) - \frac{N-1}{N}\phi_{N-2}^{Leg}(x) \tag{9}$$

In order to compute the roots, we use Newton's method. This algorithm is housed under `NewtonRaphsonSolver.hpp` and has a Newton's method and a Newton's method combined with a bisection method. These algorithms were adopted from Numerical Recipes.

Lobatto Points

Finally, Lobatto points expand on the usefulness of Legendre points to include the end points of the domain. Thus, Lobatto points are the roots of Lobatto polynomials:

$$\phi_N^{Lob}(x) = (1-x^2)\frac{d}{dx}[\phi_{N-1}^{Leg}(x)] \tag{10}$$

Again, we find these roots via the Newton-Raphson method provided in `NewtonRaphsonSolver.hpp`. For both Legendre and Lobatto points, we use a Chebyshev point as the initial guess for the Newton method.

For the implementation, each of these "grids" can be found in `Grid1D.hpp`. We define a pure virtual interface class `Grid1DBase` to have each grid inherit from to take advantage of polymorphism. Each grid class needs to be able to return the bounds of the domain, the number of points, the actual points, the weights for quadrature (if they exist), and an `operator[]` to index into the grid points.

Interpolation Results

Overview

With the polynomial functions and basis points outlined, let's look at how to use these basis functions and points to interpolate another function. We start by expressing a function in terms of a (general) polynomial expansion:

$$f(x) \approx \sum_{i=0}^{N} \phi_i(x) f(x_i) \tag{11}$$

where $\phi_i(x)$ is the polynomial basis function, points $x_i, i = 0, \dots, N$ are the nodal basis points, and $f(x_i) = f_i$ is the function evaluated at the nodal points. We will use Lagrange polynomials, so $\phi_i(x) = L_i(x)$. Truncating the sum to N terms and substituting the Lagrange polynomials gives an interpolating function:

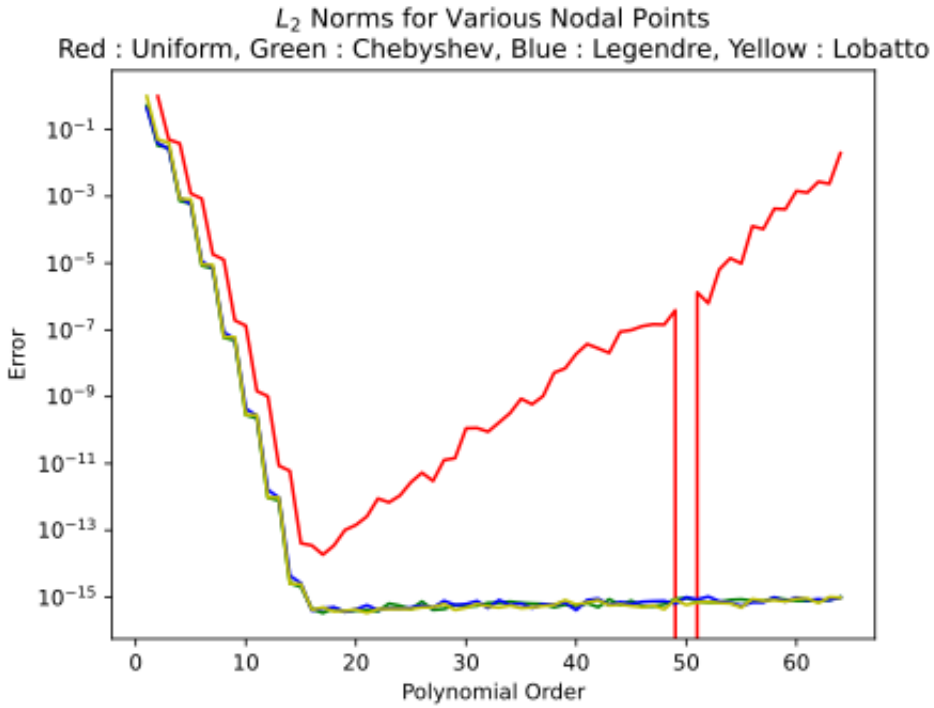
$$f_N(x) = \sum_{i=0}^{N} L_i(x) f_i. \tag{12}$$

Next, we select a set of sample points to evaluate our interpolate at. We simply use a uniform grid of 50 points from $x = [-1, 1]$. We call these points x_k. When we evaluate the Lagrange polynomials of nodal basis points at the sample points, we get a matrix $L_{ik} \in \mathbb{R}^{N \times Q}$, where N is the number of nodal points, and Q is the number of sample points. Thus, our interpolation reduces to a matrix-vector multiply of L_{ik} (or really, it's transpose) and the function evaluated at our nodal points:

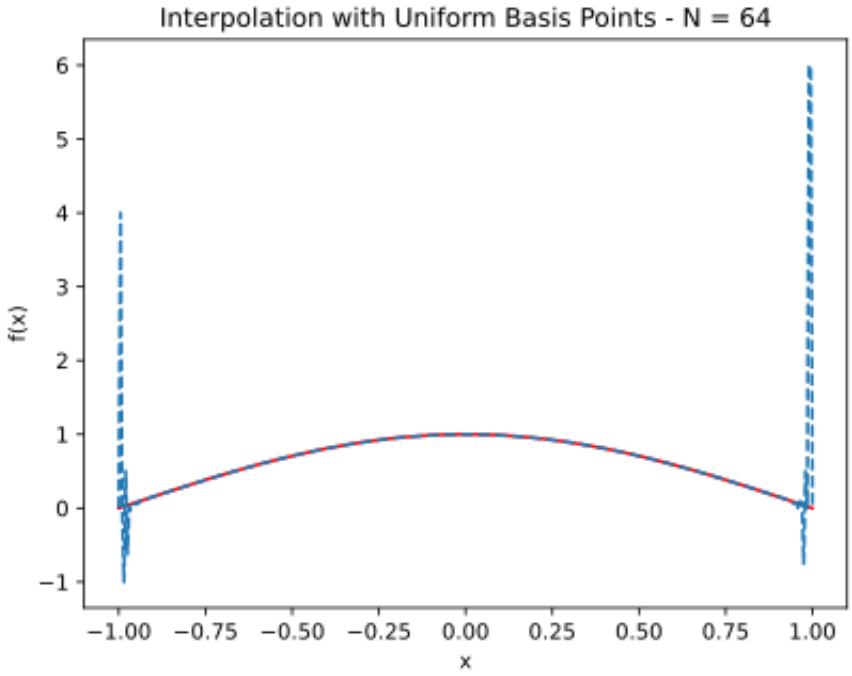
$$f_k = (L_{ik})^T f_i = L_{ki} f_i \tag{13}$$

Results

We explore the performance of each set of basis points with Lagrange basis functions for $N = 1, \dots, 64$. The results are plotted below:



As we increase N, each nodal basis point set decreases in error until about machine precision. Around this point, the uniform grid begins accumulating round-off error (probably at the edges of the domain). If we plot the function $f(x)$ that we are looking at, and the interpolation results for the uniform basis at $N = 64$, we see what's going on:



There are not enough points near the boundary of the domain to capture the function well. This is one of the advantages of using higher order basis points like Chebyshev, Legendre, and Lobatto becuase they have more points collocated near the boundary than in the center of the domain.

The other basis points do a better job at interpolating the function. As N increases, the error drops at a fast rate, and once it reaches machine precision, it stays there. In fact, for $N > 16$, the error is at machine precision. Thus, in the future with elements, we'll really only need $N \approx 10 - 20$ for each element to get the most out of the polynomial basis functions. Any more and we'd be doing more work for nothing in return.

(Note: I'm not sure why there is a drop in the uniform error around $N = 50$...)

Code

The following function demonstrates the implementation of the above.

```
void runProblem1() {
    // Create plotting grid and sample function at points
    double xLower = -1;
    double xUpper = 1;
    int nPlot = 50;
    HydroForest::UniformGrid1D<double> plotGrid(xLower, xUpper,
nPlot);
    HydroForest::Vector<double> samplePoints =
plotGrid.getPoints();
    HydroForest::Vector<double> fSample(nPlot);
    for (auto i = 0; i < nPlot; i++) {
        fSample[i] = f(samplePoints[i]);
    }

    // Iterate through basis order
    HydroForest::Vector<double> uniformErrors(64);
    HydroForest::Vector<double> chebyshevErrors(64);
    HydroForest::Vector<double> legendreErrors(64);
    HydroForest::Vector<double> lobattoErrors(64);
    std::vector<HydroForest::Vector<double>*> errorVectors = {
        &uniformErrors,
        &chebyshevErrors,
        &legendreErrors,
        &lobattoErrors
    };
    HydroForest::Vector<int> basisOrders =
HydroForest::vectorRange(1, 64);
    for (auto n = 0; n < basisOrders.size(); n++) {
        auto basisOrder = basisOrders[n];
```

```
        // Create basis points
        HydroForest::UniformGrid1D<double> uniformBasis(xLower,
xUpper, basisOrder);
        HydroForest::ChebyshevGrid1D<double>
chebyshevBasis(basisOrder);
        HydroForest::LegendreGrid1D<double>
legendreBasis(basisOrder);
        HydroForest::LobattoGrid1D<double>
lobattoBasis(basisOrder);
        std::vector<HydroForest::Grid1DBase<double>*>
basisPoints(4);
        basisPoints[0] = &uniformBasis;
        basisPoints[1] = &chebyshevBasis;
        basisPoints[2] = &legendreBasis;
        basisPoints[3] = &lobattoBasis;

        // Iterate through basis functions
        // [uniform, chebyshev, legendre, lobatto]
        for (auto basisIndex = 0; basisIndex < basisPoints.size();
basisIndex++) {
            auto& basisPointGrid = *basisPoints[basisIndex];
            auto& errorVector = *errorVectors[basisIndex];

            // Create Lagrange nodal points and interpolation
matrix
            HydroForest::Vector<double> nodalPoints =
basisPointGrid.getPoints();
            HydroForest::LagrangePolynomial
lagrangePolyBasis(nodalPoints);
            HydroForest::Matrix<double> L_ik =
lagrangePolyBasis(samplePoints);
            HydroForest::Matrix<double> L_ki = L_ik.T(); //
Transpose for inner product

            HydroForest::Vector<double>
fNodal(nodalPoints.size());
            for (auto i = 0; i < fNodal.size(); i++) {
                fNodal[i] = f(nodalPoints[i]);
            }
            HydroForest::Vector<double> fInterpolated = L_ki *
fNodal; // f_i = L_ki * f_k

            double l2Error =
HydroForest::computeL2Norm(fInterpolated, fSample);
            errorVector[n] = l2Error;
        }
```

```
    }

    // Plot polynomial order vs norm for all basis functions
    plt::semilogy(basisOrders.data(), uniformErrors.data(), "-r");
    plt::semilogy(basisOrders.data(), chebyshevErrors.data(), "-
g");
    plt::semilogy(basisOrders.data(), legendreErrors.data(), "-
b");
    plt::semilogy(basisOrders.data(), lobattoErrors.data(), "-y");
    plt::title("$L_2$ Norms for Various Nodal Points\nRed :
Uniform, Green : Chebyshev, Blue : Legendre, Yellow : Lobatto");
    plt::xlabel("Polynomial Order");
    plt::ylabel("Error");
    plt::save("poly_order_vs_error_function.pdf");
    plt::show();

    return;
}
```

Problem 2

Overview

The goal for this problem is to look at how we can use the polynomial basis functions and points we established above to differentiate a function. Becuase differentiation is a linear operator, and we are using a linear basis, we can simply move the derivative operator onto the basis functions:

$$f(x) \approx \sum_{j=0}^{N} \phi_j(x) f(x_j) \tag{14}$$

$$\Rightarrow \frac{df(x)}{dx} \approx \frac{d}{dx} \sum_{j=0}^{N} \phi_j(x) f(x_j) \tag{15}$$

$$= \sum_{j=0}^{N} \frac{d\phi_j(x)}{dx} f(x_j). \tag{16}$$

As we are using Lagrange basis functions, and we already have a closed form of the derivative, we can reuse most of the functionality in the code and algorithms above to get results for the derivative. Recall that we have the following function to get a matrix for the Lagrange polynomial derivatives at the nodal points and sample points:

```
// In struct LagrangePolynomial
Matrix<double> derivative(Vector<double> xSample) {
```

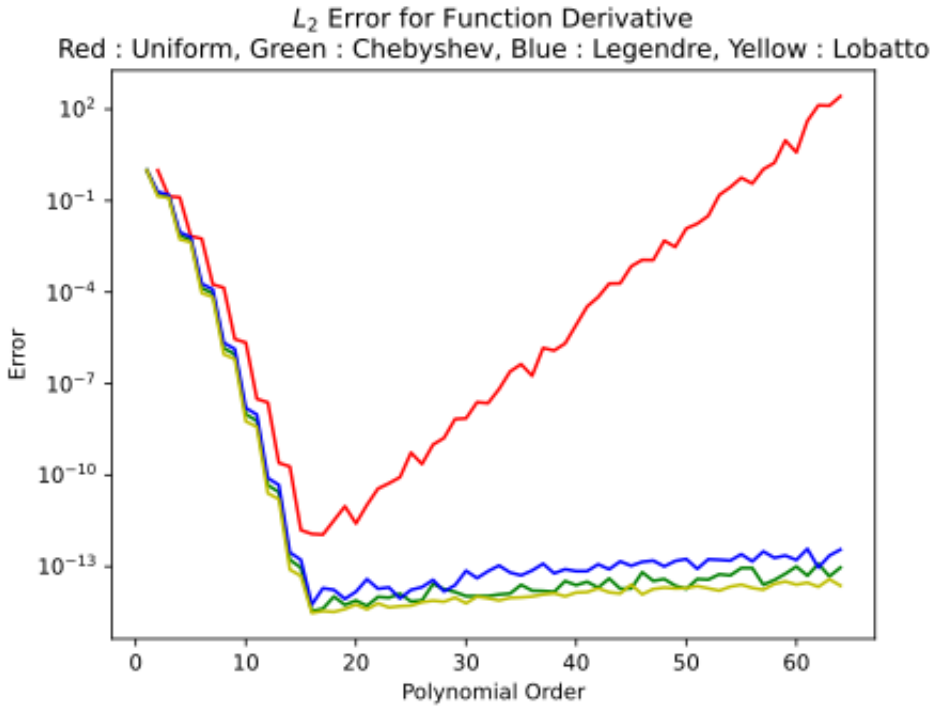
```
        std::size_t Q = xSample.size();
        std::size_t N = nodalPoints.size();
        Matrix<double> dL_il(N, Q);
        for (auto l = 0; l < Q; l++) {
            double x_l = xSample[l];
            for (auto i = 0; i < N; i++) {
                double x_i = nodalPoints[i];
                for (auto j = 0; j < N; j++) {
                    double x_j = nodalPoints[j];
                    double prod = 1.0;
                    if (j != i) {
                        for (auto k = 0; k < N; k++) {
                            double x_k = nodalPoints[k];
                            if (k != i && k != j) {
                                prod *= (x_l - x_k) / (x_i - x_k);
                            }
                        }
                        dL_il(i,l) += prod / (x_i - x_j);
                    }
                }
            }
        }
        return dL_il;
    }
```

Thus evaluating the derivative reduces again to a matrix-vector multiplication:

$$f'_k = (L'_{ik})^T f_i = L'_{ki} f_i \tag{17}$$

Results

Running the same process to iterate over $N = 1, \ldots, 64$, we get the following error results:



As we increase N, the error drops for all basis point sets, but again the error rises sharply for the uniform grid. This is again due to the lack of points collocated on the boundary of the domain. All other sets drop to about 10^{13} and stay there, with the Lobatto points just slightly better. This may be due to the fact that the Lobatto points have points on the boundary of the domain.

Code

With only slight changes, we can use the same methodology from problem 1. The code is provided below in the following function:

```
void runProblem2() {
    // Create plotting grid and sample function at points
    double xLower = -1;
    double xUpper = 1;
    int nPlot = 50;
    HydroForest::UniformGrid1D<double> plotGrid(xLower, xUpper,
nPlot);
    HydroForest::Vector<double> samplePoints =
plotGrid.getPoints();
    HydroForest::Vector<double> fSample(nPlot);
    HydroForest::Vector<double> dfSample(nPlot);
    for (auto i = 0; i < nPlot; i++) {
        fSample[i] = f(samplePoints[i]);
        dfSample[i] = df(samplePoints[i]);
```

```
    }

    // Iterate through basis order
    HydroForest::Vector<double> uniformErrors(64);
    HydroForest::Vector<double> chebyshevErrors(64);
    HydroForest::Vector<double> legendreErrors(64);
    HydroForest::Vector<double> lobattoErrors(64);
    std::vector<HydroForest::Vector<double>*> errorVectors = {
        &uniformErrors,
        &chebyshevErrors,
        &legendreErrors,
        &lobattoErrors
    };
    HydroForest::Vector<int> basisOrders =
HydroForest::vectorRange(1, 64);
    for (auto n = 0; n < basisOrders.size(); n++) {
        auto basisOrder = basisOrders[n];

        // Create basis points
        HydroForest::UniformGrid1D<double> uniformBasis(xLower,
xUpper, basisOrder);
        HydroForest::ChebyshevGrid1D<double>
chebyshevBasis(basisOrder);
        HydroForest::LegendreGrid1D<double>
legendreBasis(basisOrder);
        HydroForest::LobattoGrid1D<double>
lobattoBasis(basisOrder);
        std::vector<HydroForest::Grid1DBase<double>*>
basisPoints(4);
        basisPoints[0] = &uniformBasis;
        basisPoints[1] = &chebyshevBasis;
        basisPoints[2] = &legendreBasis;
        basisPoints[3] = &lobattoBasis;

        // Iterate through basis functions
        // [uniform, chebyshev, legendre, lobatto]
        for (auto basisIndex = 0; basisIndex < basisPoints.size();
basisIndex++) {
            auto& basisPointGrid = *basisPoints[basisIndex];
            auto& errorVector = *errorVectors[basisIndex];

            // Create Lagrange nodal points and interpolation
matrix
            HydroForest::Vector<double> nodalPoints =
basisPointGrid.getPoints();
            HydroForest::LagrangePolynomial
lagrangePolyBasis(nodalPoints);
```

```
            HydroForest::Matrix<double> dL_ik =
lagrangePolyBasis.derivative(samplePoints);
            HydroForest::Matrix<double> dL_ki = dL_ik.T(); //
Transpose for inner product

            HydroForest::Vector<double>
fNodal(nodalPoints.size());
            for (auto i = 0; i < fNodal.size(); i++) {
                fNodal[i] = f(nodalPoints[i]);
            }
            HydroForest::Vector<double> dfInterpolated = dL_ki *
fNodal; // df_i = dL_ki * f_k

            double l2Error =
HydroForest::computeL2Norm(dfInterpolated, dfSample);
            errorVector[n] = l2Error;
        }
    }

    // Plot polynomial order vs norm for all basis functions
    plt::semilogy(basisOrders.data(), uniformErrors.data(), "-r");
    plt::semilogy(basisOrders.data(), chebyshevErrors.data(), "-
g");
    plt::semilogy(basisOrders.data(), legendreErrors.data(), "-
b");
    plt::semilogy(basisOrders.data(), lobattoErrors.data(), "-y");
    plt::title("$L_2$ Error for Function Derivative\nRed :
Uniform, Green : Chebyshev, Blue : Legendre, Yellow : Lobatto");
    plt::xlabel("Polynomial Order");
    plt::ylabel("Error");
    plt::save("poly_order_vs_error_derivative.pdf");
    plt::show();

    return;
}
```

Problem 3

Finally, let's look at how to integrate a function using these polynomial basis functions and points. We will do so using Gaussian quadrature.

Overview

Numerically integrating a function over a set domain using a specific set of points is called a quadrature rule. Because of the properties of the polynomials we are using, we can integrate quite effectively using the quadrature weights generated from each

polynomial and basis set. While this would work for Chebyshev points, and even uniform points with the right weights and weighting function, we will be looking at Legendre-Gauss and Legendre-Gauss-Lobatto quadrature points.

Legendre-Gauss Quadrature

The weights of an LG quadrature are given by:

$$w_i = \frac{2}{(1-x_i^2)(\phi'_N(x_i))^2} \tag{18}$$

Legendre-Gauss-Lobatto Quadrature

The weights of an LGL quadrature are given by:

$$w_i = \frac{2}{N(N+1)(\phi_N(x_i))^2} \tag{19}$$

Once we have the weights, we can use them to interpolate a function as follows:

$$I = \int_{-1}^{1} f_N(x)dx = \int_{-1}^{1} \sum_{i=0}^{N} L_i(x) f_i dx \tag{20}$$

$$= \sum_{k=0}^{N} w_k \Big(\sum_{i=0}^{N} L_i(x) f_i \Big) \tag{21}$$

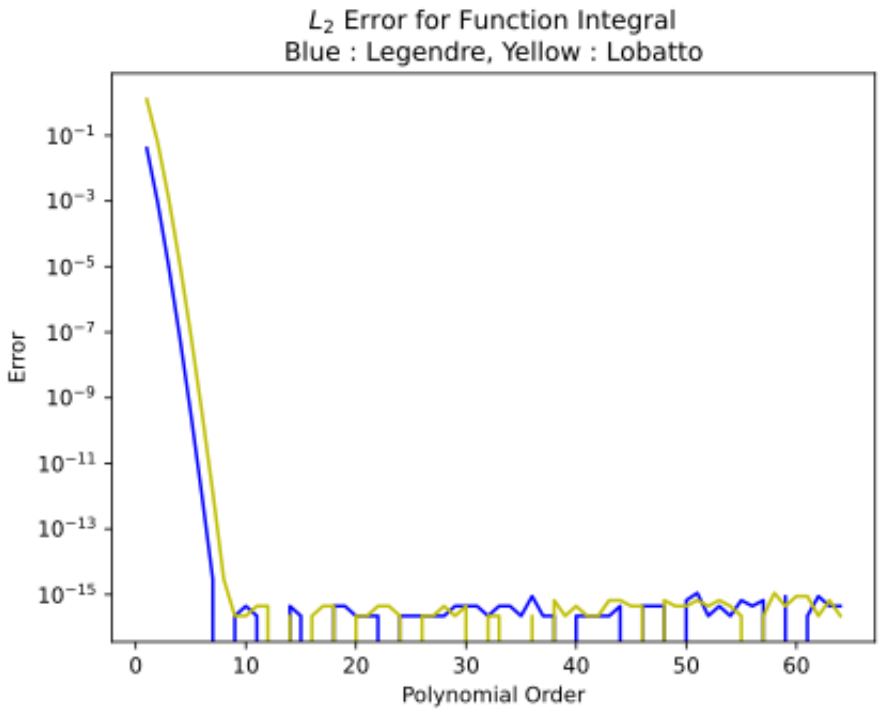
or rewriting in Einstein notation:

$$I = w_k (L_{ik})^T f_i = w_k L_{ki} f_i \tag{22}$$

To implement this, the difference now is we use the basis points for both the sample and basis points when computing L_{ik}. This now gives a matrix to map nodal points to nodal points where the weights are also collocated. By summing up the product of the weights and the nodal function points, we get the value of the integral via Gaussian quadrature.

Results

Again running from $N = 1, \dots, 64$, we get the following error results:



As we can expect, the error gets driven to machine precision and stays there for both LG and LGL quadrature. It only taks about 8-12 order polynomials to accomplish this, which is quite good. We'll use this when we start doing finite element integrals!

Code

Again, the code is similar to the methodology from above, with the difference being the sample points we use to evaluate the matrix for the Lagrange polynomials:

```
void runProblem3() {
    // Iterate through basis order
    HydroForest::Vector<double> legendreErrors(64);
    HydroForest::Vector<double> lobattoErrors(64);
    std::vector<HydroForest::Vector<double>*> errorVectors = {
        &legendreErrors,
        &lobattoErrors
    };
    HydroForest::Vector<int> basisOrders =
HydroForest::vectorRange(1, 64);
    for (auto n = 0; n < basisOrders.size(); n++) {
        auto basisOrder = basisOrders[n];

        // Create basis points
        HydroForest::LegendreGrid1D<double>
legendreBasis(basisOrder);
        HydroForest::LobattoGrid1D<double>
```

```
lobattoBasis(basisOrder);
        std::vector<HydroForest::Grid1DBase<double>*>
basisPoints(2);
        basisPoints[0] = &legendreBasis;
        basisPoints[1] = &lobattoBasis;

        // Iterate through basis functions
        // [legendre, lobatto]
        for (auto basisIndex = 0; basisIndex < basisPoints.size();
basisIndex++) {
            auto& basisPointGrid = *basisPoints[basisIndex];
            auto& errorVector = *errorVectors[basisIndex];

            // Create Lagrange nodal points and interpolation
matrix
            HydroForest::Vector<double> nodalPoints =
basisPointGrid.getPoints();
            HydroForest::Vector<double> nodalWeights =
basisPointGrid.getWeights();
            HydroForest::LagrangePolynomial
lagrangePolyBasis(nodalPoints);
            HydroForest::Matrix<double> L_ik =
lagrangePolyBasis(nodalPoints);
            HydroForest::Matrix<double> L_ki = L_ik.T(); //
Transpose for inner product

            HydroForest::Vector<double>
fNodal(nodalPoints.size());
            for (auto i = 0; i < fNodal.size(); i++) {
                fNodal[i] = f(nodalPoints[i]);
            }
            HydroForest::Vector<double> fInterpolated = L_ki *
fNodal; // f_i = L_ki * f_k
            double fIntegrated = fInterpolated * nodalWeights;
            double absoluteDiff = fabs(fIntegrated - fIntegral);
            errorVector[n] = absoluteDiff;
        }
    }

    // Plot polynomial order vs norm for all basis functions
    plt::semilogy(basisOrders.data(), legendreErrors.data(), "-
b");
    plt::semilogy(basisOrders.data(), lobattoErrors.data(), "-y");
    plt::title("$L_2$ Error for Function Integral\nBlue :
Legendre, Yellow : Lobatto");
    plt::xlabel("Polynomial Order");
    plt::ylabel("Error");
```

```
    plt::save("poly_order_vs_error_integral.pdf");
    plt::show();

    return;
}
```

References

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