1 Lagrange Interpolating Polynomial

We are trying to write a polynomial which, if we are considering the points $x_0, \ldots, x_n$, is equal to one at x_i, and equal to zero for all x_j, with $j \neq i$. We exhibit this polynomial below.

$$f_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \tag{1}$$

Note that $f_i(x_i) = \prod_{j \neq i} \frac{x_i - x_j}{x_i - x_j}$, and since all of the numerators and denominators are equal, we can see that $f_i(x_i) = 1$. For $f_i(x_k) = \prod_{j \neq i} \frac{x_k - x_j}{x_i - x_j}$, if $k \neq i$, since j ranges over all indices except i, one of the j's will be equal to k. That will make the numerator zero, and thus the whole product will be zero. Therefore, $f_i(x_k) = \delta_{ik}$.

If we want our polynomial to have the value y_i at each point x_i, we can sum several of these polynomials, so that the resultant polynomial has the characteristics which we desire:

$$\begin{aligned} f(x) &= \sum_i y_i f_i(x) && (2) \\ &= \sum_i y_i \left(\prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \right) && (3) \end{aligned}$$

This polynomial has the required values at each point.

2 Differentiating a Lagrange Interpolating Polynomial

Given the definition of $f_i(x)$ given above, we can compute the derivative of $f_i(x)$ with respect to x.

$$\frac{df_i(x)}{dx} = \sum_{\substack{k=0 \\ k \neq i}}^{n} \frac{1}{x_i - x_k} \left(\prod_{\substack{j=0 \\ j \neq k \\ j \neq i}}^{j=n} \frac{x - x_j}{x_i - x_j} \right) \tag{4}$$

Therefore, for the whole approximating function $f(x)$, we have:

$$\frac{df}{dx}(x) = \sum_{i=0}^{n} y_i \sum_{\substack{k=0\\k\neq i}}^{n} \frac{1}{x_i - x_k} \left(\prod_{\substack{j=0\\j\neq k\\j\neq i}}^{j=n} \frac{x - x_j}{x_i - x_j} \right) \tag{5}$$

This can be interpeted as a dot product. If we consider the vector $\vec{D}(x)$ given by:

$$D_i(x_l) = \sum_{\substack{k=0\\k\neq i}}^{n} \frac{1}{x_i - x_k} \left(\prod_{\substack{j=0\\j\neq k\\j\neq i}}^{j=n} \frac{x - x_j}{x_i - x_j} \right) \tag{6}$$

Then if we consider the vector $\vec{y} = \{y_0, \ldots, y_n\}$, then $\frac{df}{dx}(x) = \vec{D}(x) \cdot \vec{y}$.

2.1 Computing the Derivative at the Stencil Points $x_0, \ldots x_n$

Suppose $x = x_l$ is one of the coordinates. Then, $D_i(x_l)$ is given by:

$$D_i(x_l) = \sum_{\substack{k=0\\k\neq i}}^{n} \frac{1}{x_i - x_k} \left(\prod_{\substack{j=0\\j\neq k\\j\neq i}}^{j=n} \frac{x_l - x_j}{x_i - x_j} \right) \tag{7}$$

However, this can be greatly simplified. If $x_l = x_i$, then we see that the product terms all drop out, since the numerators and denomiators are all equal. If $x_l \neq x_i$, then the product would only be nonzero for the case where $k = l$. Therefore, only that term in the sum survives. We show the special cases for $D_i(x_l)$ below:

$$D_i(x_i) = \sum_{\substack{k=0 \\ k \neq i}} \frac{1}{x_i - x_k} \tag{8}$$

$$\underset{l \neq i}{D_i(x_l)} = \frac{1}{x_i - x_l} \prod_{\substack{j=0 \\ j \neq l \\ j \neq i}}^{j=n} \frac{x_l - x_j}{x_i - x_j} \tag{9}$$

3 Finite Difference Scheme Code

The code snippet below implements a stencil to compute the derivative at an arbitrary point x.

```
/*  FUNCTION: 1st derivative of f(x).
    Given a stencil, it generates the finite diff coefficients

    x = point at which dfdx is evaluated
    ns = number of points on the stencil
    xs = array of locations of stencil points
    D = array to save coefficients
*/
void dfdx(double x, int ns, double *xs, double *D){
    // For the k-th stencil point, calculate its
    //      coefficient and store it in D[k].
    double aux;
    for(int i=0; i<ns; i++){
        D[i] = 0;
        for(int k=0; k<ns; k++){
            aux = 1/(xs[i]-xs[k]);
            for(int j=0; j<ns; m++){
                if(j!=i && j!=k)
                    aux *= (x - xs[j])/(xs[i] - xs[j]);
            }
            D[k] += aux;
        }
    }
}
```

4 Formal Accuracy of Finite Difference Scheme

Given a stencil, which is a series of points $x_0, \ldots, x_n$, and a point at which the derivative is to be computed, x, we compute coefficients which can be used to compute the derivative at x based on the values of the function at the points $x_0, \ldots, x_n$. The scheme comes from the differentiation of an interpolating polynomial of order $n-1$. The difference between the actual derivative, and the approximating polynomial, is a polynomial of order n.

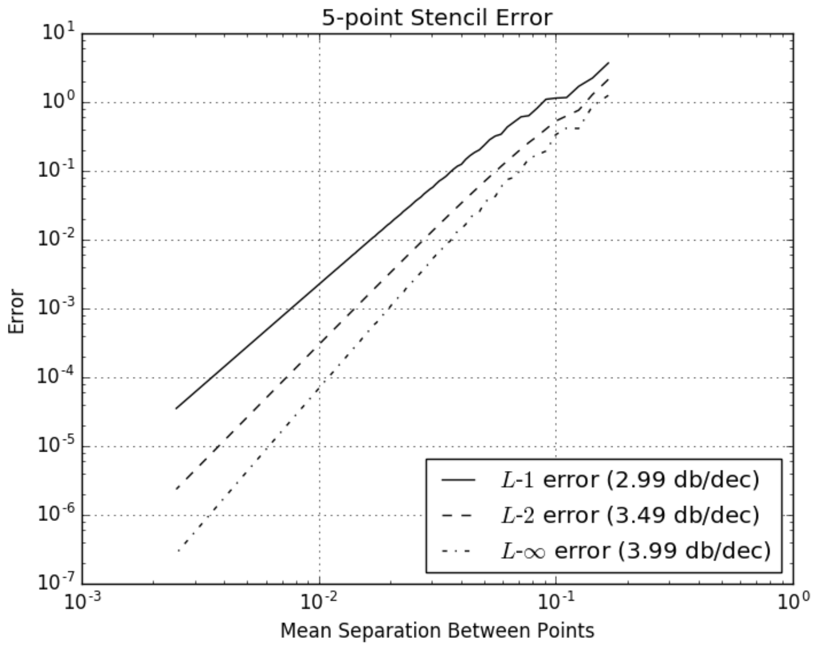
5 Computation of mesh

In this section, we consider a mesh in which the point x_i, i ranging from 0 to n, the points are distributed as:

$$x_i = \frac{i}{n} + \frac{1}{10}\sin\left(\frac{2\pi i}{n}\right) \tag{10}$$

We are using our finite-difference scheme code to generate a five-point computational stencil. This stencil varies spatially, as the point distribution changes.

Using the function $f(x) = \tanh(10x - 5)$, we compare the analytical derivative with the numerical derivative.



For some reason, the L-1 error has a slope of 3, the L-2 error has a slope of 3.5, and the L-∞ error has a slope of 4. This result surprised us, and we aren't quite sure why the errors decrease in this way. If every individual point's error is decreased in the 4th order of the mean point separation, then this accounts for the L-∞ error's slope.

5.1 Boundaries of the Mesh

The computation above was performed only on the interior points, with at least two neighbors. We referred earlier to the D matrix, where $D_{ij} = D_i(x_j)$. This was done by computing the full D matrix for a 5-point stencil, and then selecting the row corresponding to the center point. In order to compute the derivatives on the boundary, one need only select the row corresponding to the boundary point. This is a 4th order method throughout the simulation region.

However, this is not necessarily what we might want. If we want to preserve the pentadiagonal structure, the first row and last row can only contain three nonzero entries. Similarly, the second and second-to-last rows can only contain four nonzero entries. The solution is to make a D matrix for the first three (or four) points, and then select the row corresponding to the relevant point. While this preserves the diagonal structure of the matrix, this is not a 4th order matrix all throughout the simulation region.

The final way to compute the boundaries is if we perscribe the values of the function past the boundary of the simulation region, or if we perscribe the value of the function and the derivative at the boundary, we can preserve the 5th order character of the method, as well as the pentadiagonal structure.

A Code Appendix

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress

def get_D(xs):
    n = len(xs)-1
    D = np.empty((n+1,n+1))
    for l in range(n+1):
        for i in range(n+1):
            if i == l:
                term = 0.0
                for k in range(n+1):
                    if k == i:
                        continue
                    term += 1.0/(xs[i] - xs[k])
                D[l,l] = term
            else:
                term = 1.0/(xs[i] - xs[l])
                for j in range(n+1):
                    if (j == l) or (j == i):
                        continue
                    term *= (xs[l] - xs[j])/(xs[i] - xs[j])
                D[l,i] = term
    return D

def mesh(n):
    i = np.arange(n+1, dtype=np.float)
    return i/n + 0.1*np.sin(2*np.pi*i/n)

def five_point_mesh(xs):
    assert len(xs) == 5
    D = get_D(xs)
    return D[2,:]

def get_stencils(xs):
    n = len(xs)-1
    Dstencils = np.empty((n-3, 5))
    for i in range(n-3):
        Dstencils[i,:] = five_point_mesh(xs[i:i+5])
    return Dstencils

def apply_stencil(Dstencils, fvals):
    fprime_numerical = np.empty(len(fvals)-4)
    for i in range(len(fvals)-4):
        fprime_numerical[i] = np.dot(Dstencils[i,:], fvals[i:i+5])
    return fprime_numerical

def errors(xs, f, fprime, norm_orders):
    n = len(xs)-1
    Dstencils = get_stencils(xs)
    fvals = f(xs)
    fprimevals = fprime(xs[2:-2])
    fmatvals = apply_stencil(Dstencils, fvals)
    return np.array([
            np.linalg.norm(fprimevals - fmatvals, ord=norm_order)
            for norm_order in norm_orders
        ])

def f(x):
    return np.tanh(10*x - 5)

def fprime(x):
    return 10/np.cosh(5-10*x)**2

nmin = 6
nmax = 400
norm_errors = np.empty((nmax - nmin, 3))
ns = np.arange(nmin, nmax)
for n in range(nmin, nmax):
    norm_errors[n-nmin,:] = errors(mesh(n), f, fprime, [1,2,float('inf')])

nregressionmin = 50
slopes = np.empty(3)
for i in range(3):
    slopes[i], _, _, _, _ = linregress(np.log(1.0/ns[nregressionmin:]),
                                       np.log(norm_errors[nregressionmin:,i]))

plt.loglog(1.0/np.arange(nmin, nmax), norm_errors[:,0],
           label=r"$L$-$1$ error ({:1.3} db/dec)".format(slopes[0]),
           linestyle="solid", color="black")
plt.loglog(1.0/np.arange(nmin, nmax), norm_errors[:,1],
           label=r"$L$-$2$ error ({:1.3} db/dec)".format(slopes[1]),
           linestyle="dashed", color="black")
plt.loglog(1.0/np.arange(nmin, nmax), norm_errors[:,2],
           label=r"$L$-$\infty$ error ({:1.3} db/dec)".format(slopes[2]),
           linestyle='-.', color="black")
plt.legend(loc="lower right")
plt.grid(True)
plt.title("5-point Stencil Error")
plt.xlabel("Mean Separation Between Points")
plt.ylabel("Error")

plt.savefig("ErrorDecrease.png")
```