Documentation for Numerical Derivative on Discontinuous Galerkin Space

Stefan Schnake

2014

1 Introduction

This documentation gives a guide to the syntax and usage of the functions in this package as simply as possible. If the user would like to read more about the analysis and implementation of this scheme it can be found in "Discontinuous Galerkin Finite Element differential calculus and applications to numerical solution of linear and nonlinear partial differential equations" by Xiaobing Feng, Thomas Lewis, and Michael Neilan. Given a mesh T_h and an element $K \in T_h$ and a function v the goal of this project is to find piecewise polynomials $\partial^+_{h,x_i} v$ and $\partial^-_{h,x_i} v$ such that

$$\int_K \left(\partial^{\pm}_{h,x_i} v\right) \phi \, \mathrm{d}x = \int_{\partial K} Q_i^{\pm}(v) \cdot \eta_K^{(i)} \phi \, \mathrm{d}x - \int_K v \cdot \frac{\partial}{\partial x_i} \phi \, \mathrm{d}x + \int_{\partial K \setminus \partial\Omega} \gamma_{i,e} \eta_e^i [v] \cdot \phi \, \mathrm{d}x \tag{1}$$

for all polynomials ϕ up to a specified degree. Here Ω is the domain of v, K is a subdomain, i is the x_i^{th} partial, $\gamma_{i,e}$ are piecewise constants, η_K is the unit outward normal vector vector of K. Lastly $Q_i^{\pm}(\cdot)$ and $[\cdot]$ is the trace operator and jump operator of function respectively (see the paper for more information). Finally we define $\partial_{h,x_i} v = \frac{1}{2}\left(\partial^+_{h,x_i} v + \partial^-_{h,x_i} v\right)$ to get the full numerical derivative of v.

2 Algorithm

Since $\partial^{\pm}_{h,x_i} v$ and ϕ from (1) both lie in a finite dimension polynomial subspace we can numerically compute $\partial^{\pm}_{h,x_i} v$ by converting (1) into a matrix problem. Let $\{\phi_i\}_{i=1}^{NE}$ be a basis for the finite dimensional polynomial DG (Discontinuous Galerkin) space with dimension NE. Then there exists constants $\{\alpha_i\}_{i=1}^{NE}$ such that $\partial^{\pm}_{h,x_i} v = \sum_{i=1}^{NE} \alpha_i \phi_i$. This basis expression and linearity of the integral turns (1) into

$$\alpha_i \int_K \phi_i \phi_j \, \mathrm{d}x = \int_{\partial K} Q_i^{\pm}(v) \cdot \eta_K^{(i)} \phi_j \, \mathrm{d}x - \int_K v \cdot \frac{\partial \phi_j}{\partial x_i} \, \mathrm{d}x + \int_{\partial K \setminus \partial\Omega} \gamma_{i,e} [v] \cdot \phi_j \, \mathrm{d}x \tag{2}$$

for all $j = 1 \ldots NE$. Letting $A_K = \left[\int_K \phi_i \phi_j \, \mathrm{d}x\right]_{i,j=1}^{NE}$ and β_j be equal to the right hand side of (2), we get the matrix problem

$$A_K \alpha_K = \beta_K$$

where $\alpha_K = [\alpha_i]_{i=1}^{NE}$ and $\beta_K = [\beta_j]_{j=1}^{NE}$. In practical computations we take $\gamma_{i,e} = 0$ for all i, e and map all of the integration to a simplex domain $K' = \{x \in \mathbb{R}^n : x_i \geq 0 \text{ for all } i = 1, \ldots, n, \ ||x||_1 < 1\}$ by an affine transformation.

For this computational implementation, the mass matrix A is dependent on the local basis chosen for V_h. For one dimension I have used the Legendre polynomials since they give an orthogonal mass matrix. In 2D I have chosen the monomial basis $\{1, x, y, x^2, xy, y^2, x^3, x^2y, y^2x, y^3, \ldots\}$ for its simplicity.

3 One Dimension Case

3.1 Syntax

The power of this package is broken up into two functions. `meshNumericalDerivative` is the function called to compute the numerical derivative data (plus, minus, or full) and `postProcessing` interprets this data and can be used to evaluate the derivative a specified points.

3.1.1 meshNumericalDerivative

The proper syntax for calling this function is:

```
polydata = meshNumericalDerivative(v, degree, mesh, ...)
```

where `polydata` is the numerical derivative outputted as a matrix with each column being the coefficient vector $a_{K'}$ corresponding to the polynomial basis on K'. The three required arguments are

- `v`: The input function. Note that the function must be continuous and have a weak derivative on each element into to have a numerical derivative. `v` can be several classes including:

  A discrete function represented as a vector of doubles. This input should correspond to $\{v(x_j)\}_{j=0}^{J}$ where $\{x_j\}_{j=0}^{J}$ is the mesh. Note that `meshNumericalDerivative` will only accept a vector the same size as the mesh inputted and computes a cubic interpolation of the data before calculating the numerical derivative. If the user has anymore information about the discrete function, midpoints for example, then the user should create a function that interpolates the data before input. See `interp1` in the MATLAB documentation for more information.

  A matrix the same size as `polydata` which has the basis coefficients corresponding to the polynomial basis on K'. This is the preferred and fastest method when computing second, third, and higher ordered derivatives.

  A `function_handle` loaded into the workspace.

  A string of the `function_handle` that is not loaded into the workspace, e.g., `'exp'`.

  Note: Any non-discrete function inputed must be a function of solely one argument. If the inputted function has more than one input - `func1(x,a)`, then the user should create an anonymous function to handle the extra argument - `testfunc = @(x) func1(x,3)`.

  An aside about inputted functions: Since the input `v` can be discontinuous across edges the user must be careful about how the outputs of `v` are displayed. Take for example the function $g(x) = \begin{cases} 0, & x \geq 0 \\ 1, & x < 0 \end{cases}$. In order for the numerical derivative to work properly, $g(0)$

must output both 0 and 1. To clarify which value is which, the output will be a 2×1 column vector where the top element is the left hand limit and the bottom value is a right hand limit. For this example, `g(0) = [1;0]`. Because of this the code has been adapted to take a variety of inputs. `v` can either always output 2×1 column vector that is the same value on the interior of an element and (possibly) different on the edge, or a overloaded function that returns a single value on the interior and a 2×1 column vector on the edge. The code in this project does not make use of inputting vectors into `v` so this is okay. Also scalar-valued, piecewise continuous functions work as expected.

- `degree`: A nonnegative integer that specifies the degree of polynomial space. Increase this number to increase the accuracy of the numerical derivative or if the user wants to take multiple derivatives of the inputed function. The `degree` should any non-negative integer not exceeding 5.

- `mesh`: A vector of doubles that specifies the mesh of the domain - where the elements in the vector are increasing. In this case, the whole domain is an interval and each element is a subinterval. Note that the mesh does not have to be uniform - only increasing.

There are also optional arguments which must come in pairs. The first argument is the argument identifier and the second is the argument value. For example,

`poly_derv = meshNumericalDerivative('exp', 3, [-1:.1:1],'accuracy','high')`.

Here `'accuracy'` is the argument identifier and `'high'` is the argument value.

- `'derivative'`: (Default value: `'full'`). Specifies whether the output is $\partial_h^+ v$, $\partial_h^- v$, or $\partial_h v$. The possible argument values are `'plus'`, `'minus'`, and `'full'` which will give you $\partial_h^+ v$, $\partial_h^- v$, or $\partial_h v$ respectively.

- `'accuracy'`: (Default value: `'medium'`). Specifies the accuracy of the numerical quadrature used when computing the numerical derivative. This program takes advantage of a Gaussian quadrature scheme.

  `'low'`: A low order, 2 point, Gaussian quadrature scheme. While this is the fastest of the three options only use it with a `degree` of 0, 1, or 2.

  `'medium'`: A medium order, 3 point, Gaussian quadrature scheme. Use it with a `degree` of 2 or 3.

  `'high'`: A high order, 5 point, Gaussian quadrature scheme. Use it with a `degree` of 3 or 4.

  `'vhigh'`: A very high order, 7 point, Gaussian quadrature scheme. Use it with a `degree` of 5.

3.1.2 postProcessing

The proper syntax for calling this function is:

`pointvalues = postProcessing(polydata,mesh,format,x)`

where `pointvalues` is a matrix of function values of the derivative for input `x`. The 4 required arguments are

- `polydata`: The derivative polynomial data outputted from `meshNumericalDerivative`

- `mesh`: The same mesh used in the creation of `polydata`.
- `format`: This should be either a 0 or 1 depending on what the user wants outputted when one of the values in `x` is on the boundary of an element.

  0 will give only scalar outputs with values on the edge being the average of the left and right hand value, that is,

$$\frac{1}{2}\left(\lim_{x\to z^-}\partial_h v(x)+\lim_{x\to z^+}\partial_h v(x)\right).$$

  Note that values on the interior of an element will only have one value regardless. When 0 is specified, `x` can be any size matrix and `size(pointvalues) == size(x)`.

  1 will give a 3×1 column vector/matrix output. If the inputed value is on an edge of an element then the vector will be

$$\begin{bmatrix} \lim_{x\to z^-}\partial_h v(x) \\ \lim_{x\to z^+}\partial_h v(x) \\ \frac{1}{2}\left(\lim_{x\to z^-}\partial_h v(x)+\lim_{x\to z^+}\partial_h v(x)\right) \end{bmatrix}$$

  giving the left, right, and average derivatives values. If the inputed value is on the interior of an element then the vector will be the same value repeated thrice. Note that if 1 is specified `x` may only be inputted as a row vector and not a matrix with two or more rows.
- `x`: The specified points where the derivative will be evaluated. Look at the `format` paragraph above for how `x` should be entered.

3.2 Examples

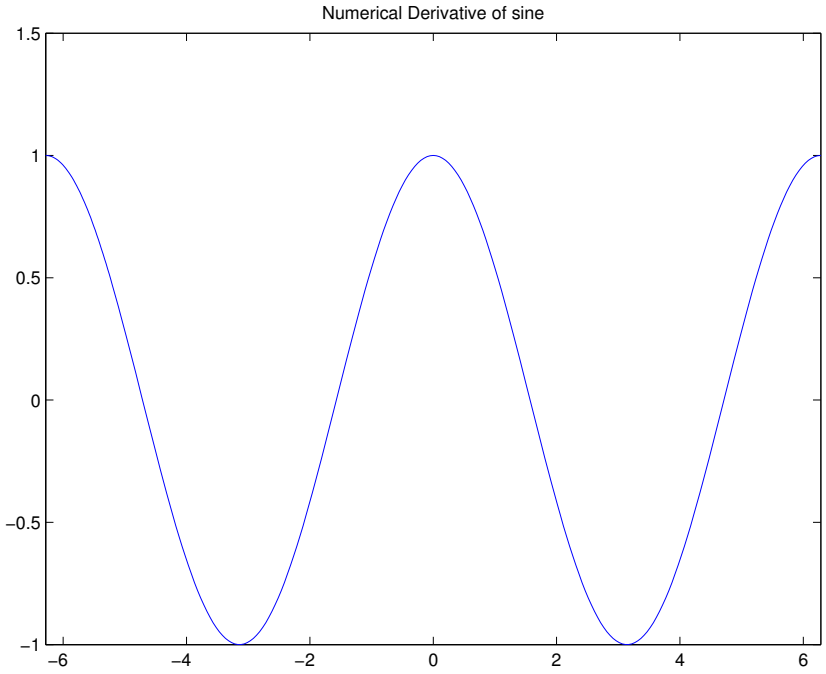
Below are a few examples of the code in action.

3.2.1 Derivative of $\sin(x)$ over $[-2\pi, 2\pi]$

Since the sine function is already a built-in MATLAB function all we need to do is create a mesh and run the code. For this example we will use a quadratic approximation and will plot the derivative after.

```
>> mesh = [-2*pi:pi/8:2*pi];
>> poly_data = meshNumericalDerivative('sin',2,mesh);
>> poly = @(x) postProcessing(poly_data,mesh,0,x);
>> fplot(poly,[-2*pi,2*pi]);title('Numerical Derivative of sine');
```

Here is the graph outputted which does map cosine, the derivative of sine.

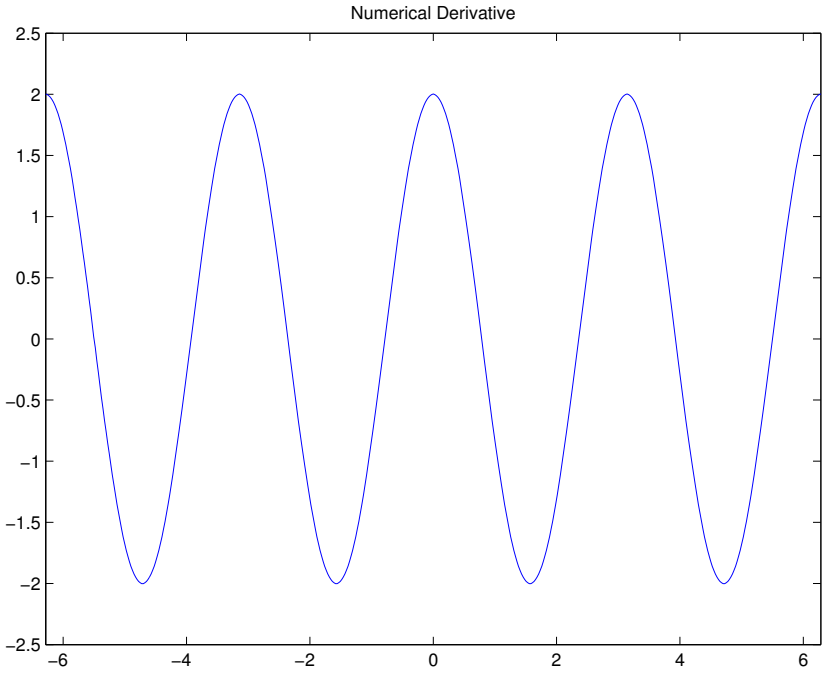


3.2.2 Derivative of $\sin(2x)$ over $[-2\pi, 2\pi]$

For this example we will use a `function_handle` that is loaded into the workspace. Again we will use quadratic approximation. We will also plot the derivative after.

```
>> mesh = [-2*pi:pi/8:pi];
>> test_sine = @(x) sin(2*x);
>> poly_data = meshNumericalDerivative(test_sine,2,mesh);
>> poly = @(x) postProcessing(poly_data,mesh,0,x);
>> fplot(poly,[-2*pi,2*pi]);title('Numerical Derivative');
```

Here is the graph outputted which does maps the derivative: $2\cos(2x)$.

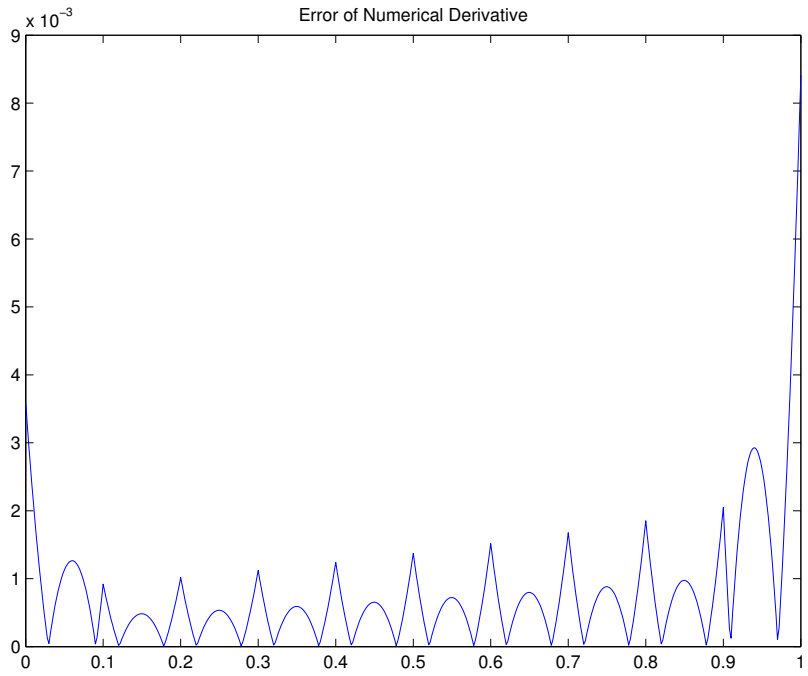


3.2.3 Derivative of e^x using a discrete input

Here we will compute the numerical derivative of e^x using only a discrete number of points. This time we will use cubic approximation. We will also plot the error of the numerical derivative and the true derivative (e^x).

```
>> mesh = [0:.1:1];
>> vp = exp(mesh);
>> poly_data = meshNumericalDerivative(vp,3,mesh);
>> error = @(x) abs(postProcessing(poly_data,mesh,0,x)-exp(x));
>> fplot(error,[0,1]);title('Error of Numerical Derivative');
```

Below is the graph of the error:

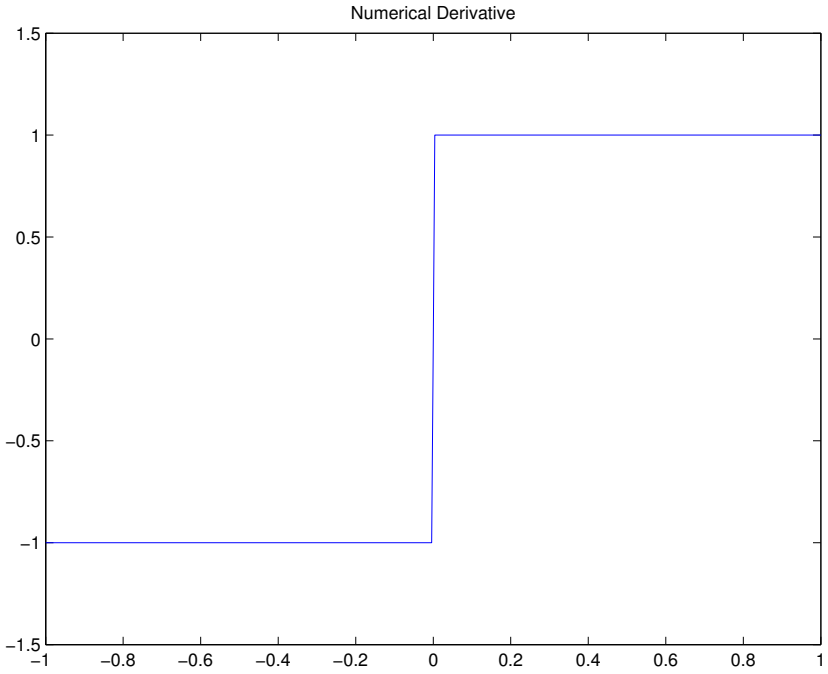


3.2.4 Derivative of $|x|$ with jumps

Here we will compute the numerical derivative of $|x|$ on the interval $[-1, 1]$. We will use a linear approximation and will adjust the `'accuracy'` to `'low'`. Note the derivative has a discontinuity at $x = 0$, with the left hand limit being -1 and the right hand limit being 1. We will show how the `'format'` argument in `postProcessing` allows the user to choose which value to use at $x = 0$.

```
>> mesh = [-1:.1:1];
>> poly_data = meshNumericalDerivative('abs',1,mesh,'accuracy','low');
>> poly = @(x) postProcessing(poly_data,mesh,1,x);
>> poly(0)

ans =

   -1.0000
    1.0000
   -0.0000

>> poly2 = @(x) postProcessing(poly_data,mesh,0,x);
>> fplot(poly2,[-1,1]);title('Numerical Derivative');
```

Also the plot of the numerical derivative.



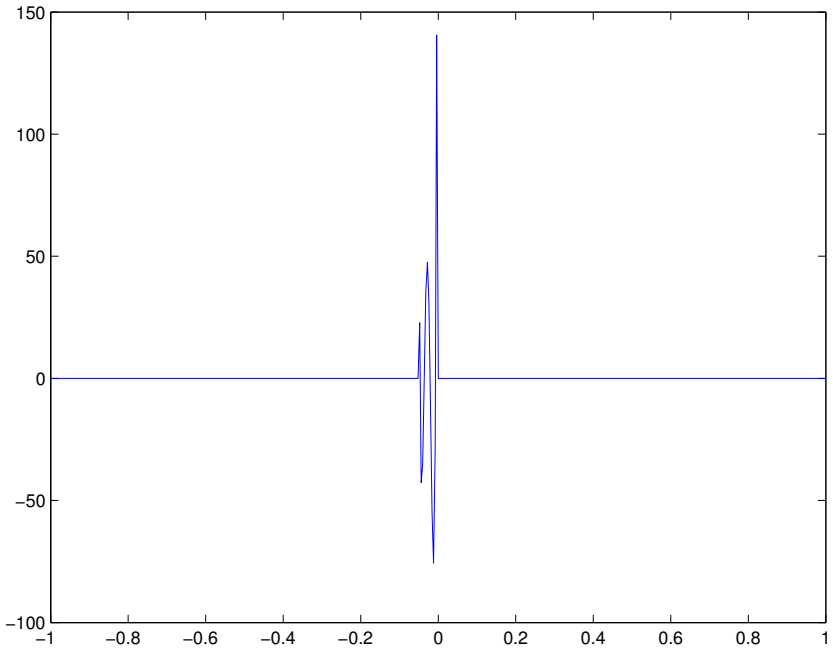
3.2.5 Numerical Derivative of Functions with no Weak Derivative

This example demonstrates what information the numerical derivative possesses when the inputted function does not have a weak derivative. Take for example, the heaviside function

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}.$$

This function has a distributional derivative, namely the Dirac-Delta function δ, but the Dirac-Delta function is not $L^1_{\text{loc}}([-1,1])$ so $H(x)$ does not have a weak derivative. We will calculate a high order numerical derivative of the heavy side function. The heavy side function `heavyside` used in this example can be found in the appendix.

```
>> mesh = [-1:.1:1];
>> poly_data = meshNumericalDerivative('heaviside',4,mesh,'accuracy','high');
>> poly = @(x) postProcessing(poly_data,mesh,0,x);
>> fplot(poly,[-1,1])
```



Since the Dirac-Delta function is not $L^1_{\text{loc}}([-1,1])$ no information is gained by looking at the plot. However, if we look at our numerical derivative (call it ψ) in the sense of distributions our output does approximate the Dirac-Delta function for appropriate $\phi \in C^1_c([-1,1])$ functions. Indeed $\int_{-1}^{1} \psi(x)\phi(x)\,\mathrm{d}x = \langle \psi, \phi \rangle \approx \langle \delta, \phi \rangle = \phi(0)$ as shown in a few examples below. For our candidate $C^1_c([-1,1])$ we will use `c1Cpt` provided in 5.2. Note that each integral should be the candidate function evaluated at 0.

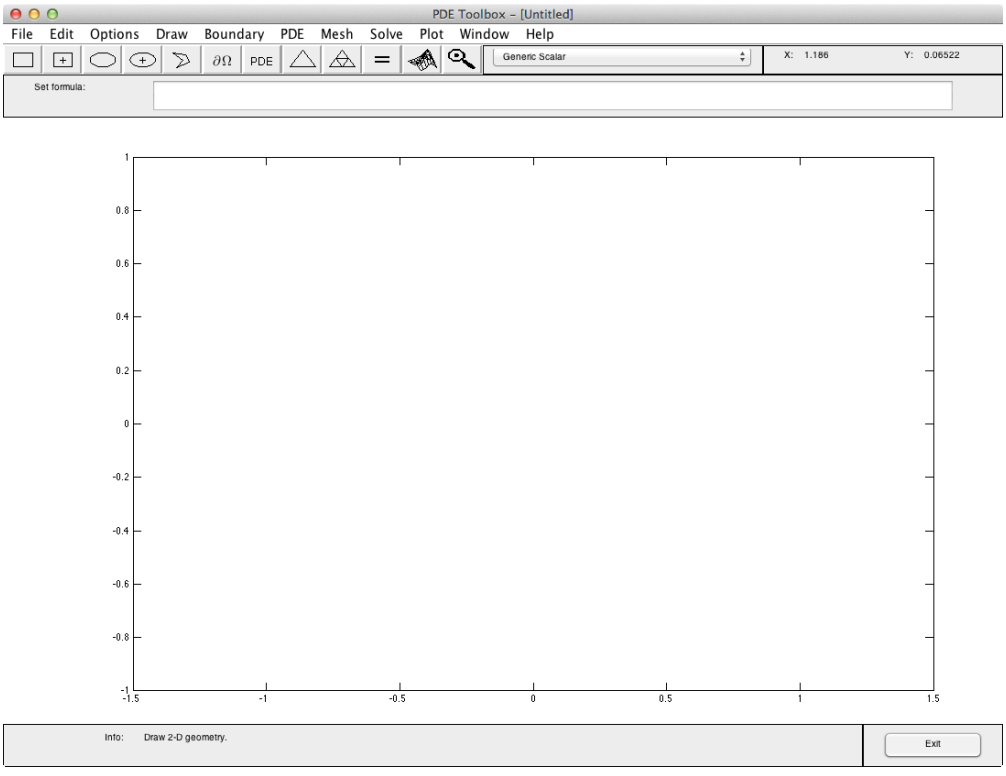
```
>> format long
>> test1 = @(x) c1Cpt(x,-.5,.5,2) .* poly(x);
>> integral(test1,-1,1)

ans =

   1.999999254385638

>> c1Cpt(0,-.5,.5,2)

ans =

     2

>> test2 = @(x) c1Cpt(x,-.43,.75,-30) .* poly(x);
>> integral(test2,-1,1)

ans =
```

```
 -57.304586160017060

>> c1Cpt(0,-.43,.75,-30)

ans =

 -57.304610703052091
```

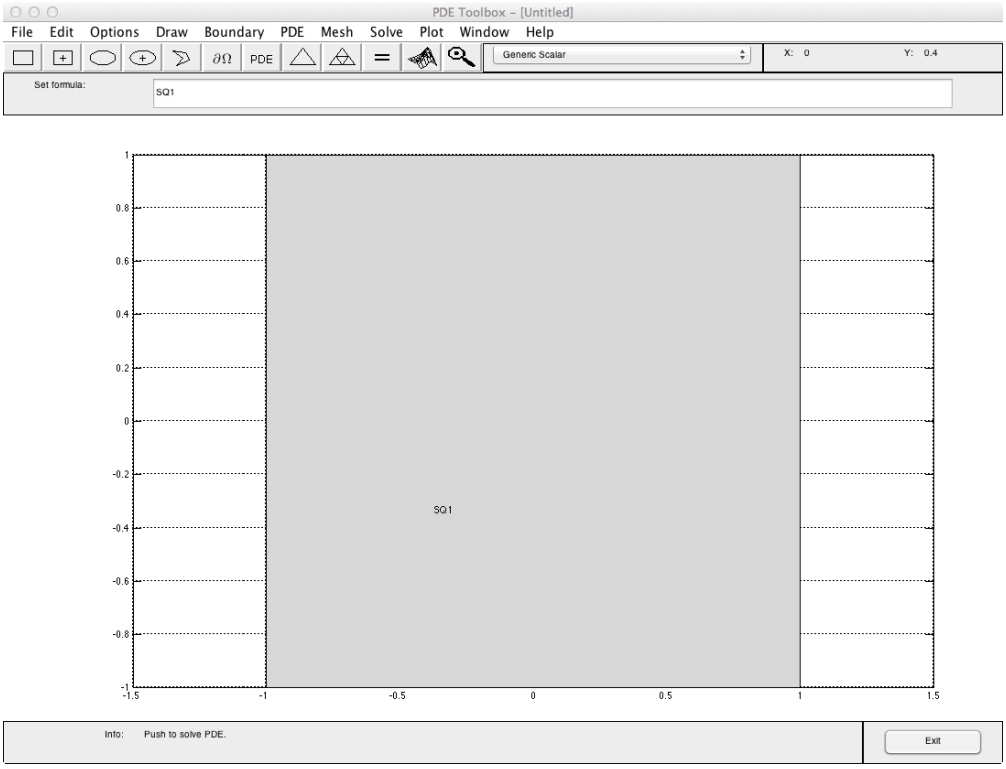
4 Two Dimension Case

4.1 Prerequisites

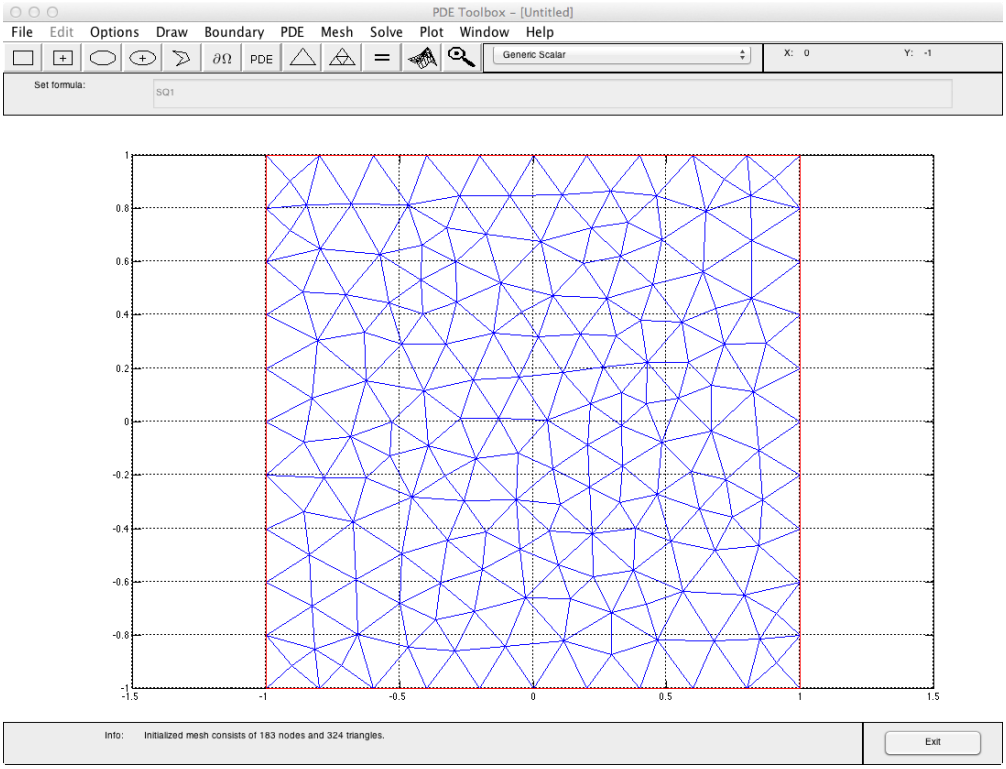
The mesh in the 1-dimensional case is trivial to create since it is a vector of points and geometrically the edge of each element is the boundary points. The 2-D case requires some additional prep work before we can dive into the numerical derivative. This program is designed to work with MATLAB's Partial Differential Equation Toolbox. Specifically we will be using the `pdetool` command to create an appropriate mesh. The Partial Differential Equation Toolbox documentation is a great place to look for more information but for now we will create a simple mesh. First we will type `pdetool` into the command window and this screen below will appear.



Next we will enable "Grid" and "Snap" from the Options menu. Finally using any of the five geometry tools from the row below the menu bar we will create our shape. In this case we will create a square with a side length of 2 center at the origin.



To create the mesh, we will click "Initialize Mesh" under the Mesh menu. The "Parameters..." option is useful for further mesh properties.



Finally we will click "Initialize Mesh" under the Mesh menu and confirm with "OK". We have now created a point matrix (p), an edge matrix (e), and a triangle matrix (t) from the mesh

generator. The point and triangle matrices are required for the numerical derivative software; however since this software is for the Discontinuous Galerkin framework we need the edge data for every element instead of just the boundary (which is what (e) provides). To get this data run

```
ee = gatherEdgeData( p, e, t)
```

where `p, e, t` are from above. This command will output a cell with 6×3 matrix for each triangle in (t). Each column in the 6×3 matrix corresponds to the following edge data:

```
% | 1 first point of edge                            |
% | 2 second point of edge                           |
% | 3 triangle # of the edges shared partner         |
% | 4 x-coordinate of unit outward normal vector     |
% | 5 y-coordinate of unit outward normal vector     |
% | 6 element normal vector == edge normal vector    |
% |   is the edge on the boundary of the mesh        |
```

4.2 Syntax

Like the one dimensions case this package is broken up into two functions. `meshNumericalDerivative` is the function called to compute the numerical derivative data (plus, minus, or full) and `postProcessing` interprets this data and can be used to evaluate the derivative a specified points.

`meshNumericalDerivative` is the function called to compute the numerical derivative (plus, minus, or full). The proper syntax for calling this function is:

```
polydata = meshNumericalDerivative( v, p, ee, t, degree, position,...)
```

where `polydata` is the numerical derivative outputted as a matrix with each column being the coefficient vector $a_{K'}$ corresponding to the polynomial basis on K'. The required arguments are

- `v`: The input function. Note that the function must be continuous and have a weak derivative on each element into to have a numerical derivative. `v` can be several classes including:

  A discrete function represented as a vector of doubles. This input should correspond to $\{v(x_j, y_j)\}_{j=0}^{N}$ where $[x_j; y_j]$ is the jth column of p and N is the total number of points in the mesh. Note a cubic interpolation of the data will be computed before calculating the numerical derivative. See `griddata` in the MATLAB documentation for more information.

  A matrix the same size as `polydata` which has the basis coefficients corresponding to the polynomial basis on K'. This is the preferred and fastest method when computing second, third, and higher ordered derivatives.

  A `function_handle` loaded into the workspace.

  A string of the `function_handle` that is not loaded into the workspace, e.g., `'harmonic'` (See appendix).

  Note: Any non-discrete function inputed must be a function of exactly two arguments. If the inputted function has more than two inputs - `func2(x,y,a)`, then the user should create an anonymous function to handle the extra argument - `testfunc = @(x) func2(x,y,3)`. Also, these functions must be vectorized, accept column vectors for `x` and `y`, and output solutions as a matrix with number of rows the same as the number of rows in `x`.

An aside about inputted functions: This algorithm does not explicitly use the function values at each vertex since there can be 5 or more elements that share the same vertex. However we cannot avoid edges as they are exclusively used in the calculation of the edge integrals. Since the inputted function can be discontinuous over an edge, the output can be a 1×2 vector where the first output corresponds with the element which has the lower global labeling of the two. Inside each element, the inputted function is continuous can output a scalar or 1×2 vector (having the same value in both entries) - the program can accept both options.

- `t,p,ee`: The triangle matrix, point matrix, and edge data respectively created from the `pdetool` mesh generator and `gatherEdgeData`. Please see 4.1 for more information.

- `degree`: A nonnegative integer that specifies the degree of polynomial space. Here the sum of the degrees of each component specifies the polynomial space. For example, the polynomials x^3, x^2y, xy^2, y^3 are all degree 3. Increase this number to increase the accuracy of the numerical derivative or if the user wants to take multiple derivatives of the inputed function. The `degree` should any non-negative integer not exceeding 4.

- `position`: This flag of `1` or `2` specifies which the program will compute $\frac{\partial}{\partial x}$ or $\frac{\partial}{\partial y}$ respectfully.

There are also optional arguments which must come in pairs. The first argument is the argument identifier and the second is the argument value. For example,

`poly_derv = meshNumericalDerivative('exp', 3, [-1:.1:1],'format','first').`

Here `'format'` is the argument identifier and `'first'` is the argument value.

- `'format'`: (Default value `'average'`) A string that determines what value of the numerical derivative uses on the boundary of an element. The numerical derivative may have a discontinuity on the boundary of an element (for example, the numerical derivative of $f(x) = |x|$ is discontinuous at 0), but must be continuous on each element. Because of this there can be "two" values of the numerical derivative at each boundary point. Since this does not create a function suitable for plotting, the `'format'` option allows us to weight the value of each point. The possible options are `'first'`, `'last'`, `'average'`, and `same`. Below is a description of each option with an example point z and example function $\partial_h v$.

  `'left'` specifies the left handed value of the function, that is, $\lim_{x \to z^-} \partial_h v(x)$.

  `'right'` specifies the right handed value of the function, that is, $\lim_{x \to z^+} \partial_h v(x)$.

  `'average'` specifies the average value of the function, that is,

$$\frac{1}{2}\left(\lim_{x \to z^-} \partial_h v(x) + \lim_{x \to z^+} \partial_h v(x)\right).$$

  `'same'` handles the case where the polynomial derivative has discontinuities over the edge. The polynomial outputted is column vector valued. If evaluated on an edge the top element is the left hand limit and the bottom is the right hand limit. If evaluated on the interior of an element both the top and bottom values are the same.

- `'derivative'`: (Default value: `'full'`). Specifies whether the output is $\partial_h^+ v$, $\partial_h^- v$, or $\partial_h v$. The possible argument values are `'plus'`, `'minus'`, and `'full'` which will give you $\partial_h^+ v$, $\partial_h^- v$, or $\partial_h v$ respectively.

- `'accuracy'`: (Default value: `'medium'`). Specifies the accuracy of the numerical quadrature used when computing the numerical derivative. This program takes advantage of a Gaussian quadrature scheme.

  `'low'`: A low order, 2 point, Gaussian quadrature scheme. While this is the fastest of the three options only use it with a `degree` of 0, 1, or 2.

  `'medium'`: A medium order, 3 point, Gaussian quadrature scheme. Use it with a `degree` of 3 or 4.

  `'high'`: A high order, 5 point, Gaussian quadrature scheme. Use it with a `degree` of 5 through 8.

4.2.1 postProcessing

The proper syntax for calling this function is:

```
pointvalues = postProcessing(polydata,p,t,format,x,y)
```

where `pointvalues` is a matrix of function values of the derivative for input `x`. The 4 required arguments are

- `polydata`: The derivative polynomial data outputted from `meshNumericalDerivative`
- `p,t`: The same p and t used in the creation of `polydata`.
- `format`: This should be either a 0 or 1 depending on what the user wants outputted when one of the values in `x` is on the boundary of an element.

  0 will give only scalar outputs with values on the edge being the average of the left and right hand value, that is,

  $$\frac{1}{2}\left(\partial_h v|_{K^+}(x) + \partial_h v|_{K^-}(x)\right).$$

  Note that values on the interior of an element will only have one value regardless. When 0 is specified, `x` can be any size matrix and `size(pointvalues) == size(x)`.

  1 will give a 3×1 column vector/matrix output. If the inputed value is on an edge of an element then the vector will be

  $$\begin{bmatrix} \partial_h v|_{K^-}(x) \\ \partial_h v|_{K^+}(x) \\ \frac{1}{2}\left(\partial_h v|_{K^+}(x) + \partial_h v|_{K^-}(x)\right). \end{bmatrix}$$

  giving the left, right, and average derivatives values. If the inputed value is on the interior of an element then the vector will be the same value repeated thrice. Note that if 1 is specified `x` may only be inputted as a row vector and not a matrix with two or more rows.

- `x,y`: The specified x and y values where the derivative will be evaluated. Look at the `format` paragraph above for how `x,t` should be entered.

4.3 Examples

4.3.1 Partial Derivative of $x^3 + xy + y^2$ in x direction

We will first convert the polynomial $f(x, y) = x^3 + xy + y^2$ into an anonymous function and then compute it's numerical partial derivative $\frac{\partial f}{\partial x}$ which is $\frac{\partial f}{\partial x} = 3x^2 + y$ using a quadratic approximation. We will then compute the L^2 error of the derivative and it's numerical estimate. Our domain in this problems is a square centered at (.5,.5) with side length of 1.

```
>> f = @(x,y) x.^3 + x.*y + y.^2;
>> polydata = meshNumericalDerivative(f,p,ee,t,2,1);
>> error = @(x,y) (postProcessing(polydata,p,t,0,x,y) - derivative(x,y)).^2;
>> integral2(error,0,1,0,1)^(1/2)

ans =

   1.6531e-12
```

4.3.2 Laplacian of a Harmonic Function

This example shows the accuracy of second order numerical derivatives. We will take the numerical laplacian of the `harmonic` function $\ln(|x|)$ listed in 5.3. Note the laplacian of $\ln(|x|)$ is the identical function 0. In order to speed up the computations, we will not use the first derivatives' global function but instead the local functions defined on each element. Our mesh is a square centered at $(.75, .75)$ and side lengths of one. We will output the time it takes to compute each derivative, then to test accuracy, we will measure the error with the discrete norm defined by

$$||f|| = \frac{1}{N}\left(\sum_{j=0}^{N} f(x_j, y_j)^2\right)^{\frac{1}{2}}$$

where (x_j, y_j) are the points given in the point matrix, p, and N is the number of points in the point matrix.

```
>> tic;[dx] = meshNumericalDerivative('harmonic',p,ee,t,3,1);toc;
Elapsed time is 0.121740 seconds.
>> tic;[dxx] = meshNumericalDerivative(dx,p,ee,t,3,1);toc;
Elapsed time is 0.246442 seconds.
>> tic;[dy] = meshNumericalDerivative('harmonic',p,ee,t,3,2);toc;
Elapsed time is 0.109847 seconds.
>> tic;[dyy] = meshNumericalDerivative(dy,p,ee,t,3,2);toc;
Elapsed time is 0.251037 seconds.
>> f = @(x,y) postProcessing(dxx,p,t,0,x,y) + postProcessing(dyy,p,t,0,x,y);
>> norm(f(p(1,:)',p(2,:)'))/188

ans =

   8.0858e-05
```

5 Appendix

5.1 Heaviside function

```
function [ y ] = heaviside( x )
%The heaviside function
err_tol = 10^(-11);
if abs(x-err_tol) < 0
    y = [0;1];
elseif x < 0
    y = 0;
else
    y = 1;
end
end
```

5.2 C^1 function with compact support

```
function [ z ] = c1Cpt( x,a,b,c )
%Function that is a polynomial on (a,b), obtains a value of c at (a+b)/2,
%has compact support [a,b], and is C^1(R).

%Let m = (a+b)/2.  poly_vector, r, is a vector that satisfies
%A*r = C where
%     | a^4  a^3  a^2  a  1|           |0|
%     | b^4  b^3  b^2  b  1|           |0|
% A = |4a^3 3a^2 2a^1  1  0|  and  C = |0|
%     |4b^3 3b^2 2b^1  1  0|           |0|
%     | m^4  m^3  m^2  m  1|           |c|.
%Note that A is non-singular as long as a and b are not the same.
poly_vector = [-(16*(3*a^3-2*a^2-2*a*b+b^2))*c/((a-b)*(9*a^6-3*a^5*b-...
    21*a^4*b^2+15*a^3*b^3-10*a^5+8*a^4*b+11*a^3*b^2-5*a^2*b^3-5*a*b^4+...
    b^5)),(32*(a+b))*c/(9*a^5+6*a^4*b-15*a^3*b^2-10*a^4-2*a^3*b+9*a^2*...
    b^2+4*a*b^3-b^4),(16*(3*a^5+6*a^4*b+9*a^3*b^2-4*a^4-8*a^3*b-3*a^2*...
    b^2-2*a*b^3-b^4))*c/(9*a^7-12*a^6*b-18*a^5*b^2+36*a^4*b^3-15*a^3*...
    b^4-10*a^6+18*a^5*b+3*a^4*b^2-16*a^3*b^3+6*a*b^5-b^6),-32*a*b*(3*...
    a^4+6*a^3*b+3*a^2*b^2-4*a^3-5*a^2*b-2*a*b^2-b^3)*c/((a-b)*(9*a^6-3*...
    a^5*b-21*a^4*b^2+15*a^3*b^3-10*a^5+8*a^4*b+11*a^3*b^2-5*a^2*b^3-5*a*...
    b^4+b^5)),16*a^2*b^2*(3*a^3+6*a^2*b-4*a^2-4*a*b-b^2)*c/(9*a^7-12*...
    a^6*b-18*a^5*b^2+36*a^4*b^3-15*a^3*b^4-10*a^6+18*a^5*b+3*a^4*b^2-16*...
    a^3*b^3+6*a*b^5-b^6)];
%Create supported entries (entries that are in (a,b)).
supported_entires = (a < x) & (x < b);
%Multiply the created polynomial values with whether the value was in (a,b)
z = supported_entires .* polyval(poly_vector,x);
end
```

5.3 Harmonic function

Below is a 2-D function that is harmonic on $\mathbb{R}^2 \setminus \{0\}$.

```
function [ z ] = harmonic( x,y )
%A variant to the fundamental solution for Laplace's equation in two dimensions.
z = log((x.^2+y.^2).^(1/2));
end
```

6 Concluding Remarks

A thanks to Salmon Rogers for his `polyval2` code used in this endeavor. It can be found at

`http://www.mathworks.com/matlabcentral/fileexchange/13719-2d-weighted-polynomial-fitting-and-evaluation/content/polyfitweighted2/polyval2.m`