dg1 Documentation

Release 0.0.1

Danny Hermes, Per Olof Persson

Sep 17, 2018

Contents

- Complete library index: genindex
- Index of all modules: modindex

CHAPTER 1

assignment1 package

1.1 Submodules

1.1.1 assignment1.dg1 module

Module for solving a 1D conservation law via DG.

Adapted from a Discontinuous Galerkin (DG) solver written by Per Olof-Persson.

Check out an example notebook using these utilities to solve the problem.

class assignment1.dg1.**DG1Solver**(*num_intervals*, *p_order*, *total_time*, *dt*, *get_initial_data=None*, *points_on_ref_int=None*)

Bases: object

Discontinuous Galerkin (DG) solver for the 1D conservation law.

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0$$

on the interval $[0, 1]$. By default, uses Gaussian-like initial data

$$u(x, 0) = \exp\left(-\left(\frac{x - \frac{1}{2}}{0.1}\right)^2\right)$$

but $u(x, 0)$ can be specified via `get_initial_data`.

We represent our solution via the $(p + 1) \times n$ rectangular matrix:

$$\mathbf{u} = \begin{bmatrix} u_0^1 & u_0^2 & \cdots & u_0^n \\ u_1^1 & u_1^2 & \cdots & u_1^n \\ \vdots & \vdots & \ddots & \vdots \\ u_p^1 & u_p^2 & \cdots & u_p^n \end{bmatrix}$$

where each column represents one of n sub-intervals and each row represents one of the $p+1$ node points within each sub-interval.

Parameters

- **num_intervals** (*int*) – The number n of intervals to divide $[0, 1]$ into.
- **p_order** (*int*) – The degree of precision for the method.
- **total_time** (*float*) – The amount of time to run the solver for (starts at $t = 0$).
- **dt** (*float*) – The timestep to use in the solver.
- **get_initial_data** (*callable*) – (Optional) The function to use to evaluate $u(x, 0)$ at the points in our solution. Defaults to *get_gaussian_like_initial_data()*.
- **points_on_ref_int** (function) – (Optional) The method used to partition the reference interval $[0, h]$ into node points. Defaults to *get_evenly_spaced_points()*.

get_mass_and_stiffness_matrices()

Get the mass and stiffness matrices for the current solver.

Uses pre-computed mass matrix and stiffness matrix for $p = 1$, $p = 2$ and $p = 3$ degree polynomials and computes the matrices on the fly for larger p.

Depends on the sub-interval width `h` and the order of accuracy `p_order`.

Return type tuple

Returns Pair of mass and stiffness matrices, both with `p_order + 1` rows and columns.

ode_func(*u_val*)

Compute the right-hand side for the ODE.

When we write

$$M\dot{\mathbf{u}} = K\mathbf{u} + \begin{bmatrix} u_p^2 & u_p^3 & \cdots & u_p^n & u_p^1 \\ 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 \\ -u_p^1 & -u_p^2 & \cdots & -u_p^{n-1} & -u_p^n \end{bmatrix}$$

we specify a RHS $f(u)$ via solving the system.

Parameters **u_val** (numpy.ndarray) – The input to $f(u)$.

Return type numpy.ndarray

Returns The value of the slope function evaluated at `u_val`.

update()

Update the solution for a single time step.

We use *ode_func()* to compute $\dot{u} = f(u)$ and pair it with an RK method (*low_storage_rk()*) to compute the updated value.

class assignment1.dg1.**MathProvider**

Bases: object

Mutable settings for doing math.

For callers that wish to swap out the default behavior – for example, to use high-precision values instead of floats – this class can just be monkey patched on the module.

The module assumes through-out that solution data is in NumPy arrays, but the data inside those arrays may be any type.

Note: The callers assume *exp_func* is a vectorized exponential that can act on a NumPy array containing elements of the relevant type.

Note: The *zeros* constructor should also be able to take the `order` argument (and should produce a NumPy array).

exp_func

linspace

mat_inv

num_type
alias of `__builtin__.float`

solve

zeros

`assignment1.dg1.`**find_matrices**(*p_order*, *points_on_ref_int=None*)
Find mass and stiffness matrices.

We do this on the reference interval $[-1, 1]$. By default we use the evenly spaced points

$$x_0 = -1, x_1 = -(p-2)/p, \ldots, x_p = 1$$

but the set of nodes to use on the reference interval can be specified via the `points_on_ref_int` argument. With our points, we compute the polynomials $\varphi_j(x)$ such that $\varphi_j(x_i) = \delta_{ij}$. We do this by writing

$$\varphi_j(x) = \sum_{n=0}^{p} c_n^{(j)} L_n(x)$$

where $L_n(x)$ is the Legendre polynomial of degree n. With this representation, we need to solve

$$\begin{bmatrix} L_0(x_0) & L_1(x_0) & \cdots & L_p(x_0) \\ L_0(x_1) & L_1(x_1) & \cdots & L_p(x_p) \\ \vdots & \vdots & \ddots & \vdots \\ L_0(x_p) & L_1(x_p) & \cdots & L_p(x_p) \end{bmatrix} \begin{bmatrix} c_0^{(0)} & c_0^{(1)} & \cdots & c_0^{(p)} \\ c_1^{(0)} & c_1^{(1)} & \cdots & c_1^{(p)} \\ \vdots & \vdots & \ddots & \vdots \\ c_p^{(0)} & c_p^{(1)} & \cdots & c_p^{(p)} \end{bmatrix} = (\delta_{ij}) = I_{p+1}$$

Then use these to compute the mass matrix

$$M_{ij} = \int_{-1}^{1} \varphi_i(x)\varphi_j(x)\,dx$$

and the stiffness matrix

$$K_{ij} = \int_{-1}^{1} \varphi_i'(x)\varphi_j(x)\,dx$$

Utilizing the fact that

$$\langle L_n, L_m \rangle = \int_{-1}^{1} L_n(x)L_m(x)\,dx = \frac{2}{2n+1}\delta_{nm}$$

we can compute

$$M_{ij} = \langle \varphi_i, \varphi_j \rangle = \sum_{n,m} \left\langle c_n^{(i)} L_n, c_m^{(j)} L_m \right\rangle = \sum_{n=0}^{p} \frac{2}{2n+1} c_n^{(i)} c_n^{(j)}.$$

Similarly

$$\langle L_n'(x), L_m(x) \rangle = \begin{cases} 2 & \text{if } n > m \text{ and } n - m \equiv 1 \bmod 2 \\ 0 & \text{otherwise.} \end{cases}$$

gives

$$\begin{aligned} K_{ij} = \langle \varphi_i', \varphi_j \rangle &= \sum_{n,m} \left\langle c_n^{(i)} L_n', c_m^{(j)} L_m \right\rangle \\ &= 2 \left(c_0^{(j)} \left(c_1^{(i)} + c_3^{(i)} + \cdots \right) + c_1^{(j)} \left(c_2^{(i)} + c_4^{(i)} + \cdots \right) + \cdots + c_{p-1}^{(j)} c_p^{(i)} \right) \end{aligned}$$

(For more general integrals, one might use Gaussian quadrature. The largest degree integrand $\varphi_i \varphi_j$ has degree $2p$ so this would require $n = p + 1$ points to be exact up to degree $2(p + 1) - 1 = 2p + 1$.)

Parameters

- **p_order** (*int*) – The degree of precision for the method.
- **points_on_ref_int** (function) – (Optional) The method used to partition the reference interval $[0, h]$ into node points. Defaults to *get_evenly_spaced_points()*.

Return type tuple

Returns Pair of mass and stiffness matrices, square numpy.ndarray of dimension p_order + 1.

assignment1.dg1.**gauss_lobatto_points**(*start*, *stop*, *num_points*)

Get the node points for Gauss-Lobatto quadrature.

Using n points, this quadrature is accurate to degree $2n - 3$. The node points are $x_1 = -1$, $x_n = 1$ and the interior are $n - 2$ roots of $P_{n-1}'(x)$.

Though we don't compute them here, the weights are $w_1 = w_n = \frac{2}{n(n-1)}$ and for the interior points

$$w_j = \frac{2}{n(n-1)\left[P_{n-1}\left(x_j\right)\right]^2}$$

This is in contrast to the scheme used in Gaussian quadrature, which use roots of $P_n(x)$ as nodes and use the weights

$$w_j = \frac{2}{\left(1 - x_j\right)^2 \left[P_n'\left(x_j\right)\right]^2}$$

Note: This method is **not** generic enough to accommodate non-NumPy types as it relies on the numpy.polynomial.legendre.

Parameters

- **start** (*float*) – The beginning of the interval.
- **stop** (*float*) – The end of the interval.
- **num_points** (*int*) – The number of points to use.

Return type numpy.ndarray

Returns 1D array, the interior quadrature nodes.

assignment1.dg1.**get_evenly_spaced_points**(*start*, *stop*, *num_points*)
Get points on an interval that are evenly spaced.

This is intended to be used to give points on a reference interval when using DG on the 1D problem.

Parameters

- **start** (*float*) – The beginning of the interval.
- **stop** (*float*) – The end of the interval.
- **num_points** (*int*) – The number of points to use on the interval.

Return type numpy.ndarray

Returns The evenly spaced points on the interval.

assignment1.dg1.**get_gaussian_like_initial_data**(*node_points*)
Get the default initial solution data.

In this case it is

$$u(x,0) = \exp\left(-\left(\frac{x-\frac{1}{2}}{0.1}\right)^2\right)$$

Parameters **node_points** (numpy.ndarray) – Points at which evaluate the initial data function.

Return type numpy.ndarray

Returns The u-values at each node point.

assignment1.dg1.**get_legendre_matrix**(*points*, *max_degree=None*)
Evaluate Legendre polynomials at a set of points.

If our points are $x_0, \ldots, x_p$, this computes

$$\begin{bmatrix} L_0(x_0) & L_1(x_0) & \cdots & L_d(x_0) \\ L_0(x_1) & L_1(x_1) & \cdots & L_d(x_p) \\ \vdots & \vdots & \ddots & \vdots \\ L_0(x_p) & L_1(x_p) & \cdots & L_d(x_p) \end{bmatrix}$$

by utilizing the recurrence

$$nL_n(x) = (2n-1)xL_{n-1}(x) - (n-1)L_{n-2}(x)$$

Parameters

- **points** (numpy.ndarray) – 1D array. The points at which to evaluate Legendre polynomials.
- **max_degree** (*int*) – (Optional) The maximum degree of Legendre polynomial to use. Defaults to one less than the number of points (which will produce a square output).

Return type numpy.ndarray

Returns The 2D array containing the Legendre polynomials evaluated at our input points.

assignment1.dg1.**get_node_points**(*num_points*, *p_order*, *step_size=None*, *points_on_ref_int=None*)
Return node points to split unit interval for DG.

Parameters

- **num_points** (*int*) – The number n of intervals to divide $[0, 1]$ into.

- **p_order** (*int*) – The degree of precision for the method.
- **step_size** (*float*) – (Optional) The step size $1/n$.
- **points_on_ref_int** (function) – (Optional) The method used to partition the reference interval $[0, h]$ into node points. Defaults to *get_evenly_spaced_points()*.

Return type numpy.ndarray

Returns The x-values for the node points, with `p_order + 1` rows and n columns. The columns correspond to each sub-interval and the rows correspond to the node points within each sub-interval.

assignment1.dg1.**low_storage_rk**(*ode_func*, *u_val*, *dt*)

Update an ODE solutuon with an order 2/4 Runge-Kutta function.

The method is given by the following Butcher array:

$$\begin{array}{c|cccc} 0 & 0 & & & \\ 1/4 & 1/4 & 0 & & \\ 1/3 & 0 & 1/3 & 0 & \\ 1/2 & 0 & 0 & 1/2 & 0 \\ \hline & 0 & 0 & 0 & 1 \end{array}$$

It is advantageous because the updates k_j can be over-written at each step, since they are never re-used.

One can see that this method is **order 2** for general $\dot{u} = f(u)$ by verifying that not all order 3 node conditions are satisfied. For example:

$$\frac{1}{3} \neq \sum_i b_i c_i^2 = 0 + 0 + 0 + 1 \cdot \left(\frac{1}{2}\right)^2$$

However, for linear ODEs, the method is **order 4**. To see this, note that the test problem $\dot{u} = \lambda u$ gives the stability function

$$R(\lambda \Delta t) = R(z) = 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24}$$

which matches the Taylor series for e^z to order 4.

See Problem Set 3 from Persson's Math 228A for more details.

Parameters

- **ode_func** (*callable*) – The RHS in the ODE $\dot{u} = f(u)$.
- **u_val** (numpy.ndarray) – The input to $f(u)$.
- **dt** (*float*) – The timestep to use.

Return type numpy.ndarray

Returns The updated solution value.

1.1.2 assignment1.dg1_high_prec module

Helpers to use *assignment1.dg1* with high-precision numbers.

High-precision is achieved by using mpmath.

class assignment1.dg1_high_prec.**HighPrecProvider**
Bases: object

High-precision replacement for *assignment1.dg1.MathProvider*.

Implements interfaces that are essentially identical (at least up to the usage in *dg1*) as those provided by NumPy.

All matrices returned are numpy.ndarray with mpmath.mpf as the data type and all matrix inputs are assumed to be of the same form.

static exp_func(*value*)
Vectorized exponential function.

static linspace(*start*, *stop*, *num=50*)
Linearly spaced points.

Points are computed with mpmath.linspace() but the output (a list) is converted back to a numpy.ndarray.

static mat_inv(*mat*)
Matrix inversion, using mpmath.

static num_type(*value*)
The high-precision numerical type: mpmath.mpf.

classmethod solve(*left_mat*, *right_mat*)
Solve Ax = b for x.

A is given by left_mat and b by right_mat.

This method seeks to mirror mpmath.matrices.linalg.LinearAlgebraMethods.lu_solve(), which uses mpmath.matrices.linalg.LinearAlgebraMethods.LU_decomp(), mpmath.matrices.linalg.LinearAlgebraMethods.L_solve() and mpmath.matrices.linalg.LinearAlgebraMethods.U_solve(). Due to limitations of mpmath we use modified helpers to accomplish the upper- and lower-triangular solves. We also cache the LU-factorization for future uses.

It's worth pointing out that numpy.linalg.solve() works in exactly this fashion. From the C source there is a lapack_func that gets defined and is eventually used in Python as gufunc. Notice that the lapack_func is dgesv for doubles. Checking the LAPACK docs verifies the dgesv does an LU and then two triangular solves.

static zeros(*shape*, ***kwargs*)
Produce a matrix of zeros of a given shape.

assignment1.dg1_high_prec.**gauss_lobatto_points**(*start*, *stop*, *num_points*)
Get the node points for Gauss-Lobatto quadrature.

Rather than using the optimizations in *dg1.gauss_lobatto_points()*, this uses mpmath utilities directly to find the roots of $P_n'(x)$ (where n is equal to num_points - 1).

Parameters

- **start** (mpmath.mpf (or float)) – The beginning of the interval.
- **stop** (mpmath.mpf (or float)) – The end of the interval.
- **num_points** (*int*) – The number of points to use.

Return type numpy.ndarray

Returns 1D array, the interior quadrature nodes.

1.1.3 assignment1.dg1_symbolic module

Symbolic helper for *assignment1.dg1*.

Provides exact values for stiffness and mass matrices using symbolic algebra.

For example, *assignment1.dg1* previously used pre-computed mass and stiffness matrices from this module. These were created using evenly spaced points on $[0, 1]$ for small p. These values can be verified by *find_matrices_symbolic()* below.

assignment1.dg1_symbolic.**find_matrices_symbolic**(*p_order*, *start=0*, *stop=1*, *x_vals=None*)

Find mass and stiffness matrices using symbolic algebra.

We do this on the reference interval $[0, 1]$ with the evenly spaced points

$$x_0 = 0, x_1 = 1/p, \ldots, x_p = 1$$

and compute the polynomials $\varphi_j(x)$ such that $\varphi_j(x_i) = \delta_{ij}$. Since we are using symbolic rationals numbers, we do this directly by inverting the Vandermonde matrix V such that

$$\begin{bmatrix} 1 & x_0 & \cdots & x_0^p \\ 1 & x_1 & \cdots & x_1^p \\ \vdots & & & \vdots \\ 1 & x_p & \cdots & x_p^p \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_p \end{bmatrix} = (\delta_{ij}) = I_{p+1}$$

Then use these to compute the mass matrix

$$M_{ij} = \int_0^1 \varphi_i(x)\varphi_j(x)\,dx$$

and the stiffness matrix

$$K_{ij} = \int_0^1 \varphi_i'(x)\varphi_j(x)\,dx$$

Some previously used precomputed values for evenly spaced points on $[0, 1]$ are

$$M_1 = \frac{1}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \qquad K_1 = \frac{1}{2}\begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix}$$

$$M_2 = \frac{1}{30}\begin{bmatrix} 4 & 2 & -1 \\ 2 & 16 & 2 \\ -1 & 2 & 4 \end{bmatrix} \qquad K_2 = \frac{1}{6}\begin{bmatrix} -3 & -4 & 1 \\ 4 & 0 & -4 \\ -1 & 4 & 3 \end{bmatrix}$$

$$M_3 = \frac{1}{1680}\begin{bmatrix} 128 & 99 & -36 & 19 \\ 99 & 648 & -81 & -36 \\ -36 & -81 & 648 & 99 \\ 19 & -36 & 99 & 128 \end{bmatrix} \qquad K_3 = \frac{1}{80}\begin{bmatrix} -40 & -57 & 24 & -7 \\ 57 & 0 & -81 & 24 \\ -24 & 81 & 0 & -57 \\ 7 & -24 & 57 & 40 \end{bmatrix}$$

In addition, when $p = 3$, the Gauss-Lobatto nodes

$$x_0 = -1,\ x_1 = -\frac{1}{\sqrt{5}},\ x_2 = \frac{1}{\sqrt{5}},\ x_4 = 1$$

are **not** evenly spaced for the first time. These produce

$$M_3 = \frac{1}{42}\begin{bmatrix} 6 & \sqrt{5} & -\sqrt{5} & 1 \\ \sqrt{5} & 30 & 5 & -\sqrt{5} \\ -\sqrt{5} & 5 & 30 & \sqrt{5} \\ 1 & -\sqrt{5} & \sqrt{5} & 6 \end{bmatrix}$$

and

$$K_3 = \frac{1}{24}\begin{bmatrix} -12 & -5 & -5 & -2 \\ 5 & 0 & 0 & -5 \\ 5 & 0 & 0 & -5 \\ 2 & 5 & 5 & 12 \end{bmatrix} + \frac{\sqrt{5}}{24}\begin{bmatrix} 0 & -5 & 5 & 0 \\ 5 & 0 & -10 & 5 \\ -5 & 10 & 0 & -5 \\ 0 & -5 & 5 & 0 \end{bmatrix}$$

Parameters

- **p_order** (*int*) – The degree of precision for the method.
- **start** (sympy.core.expr.Expr) – (Optional) The beginning of the interval. Defaults to 0.
- **stop** (sympy.core.expr.Expr) – (Optional) The end of the interval. Defaults to 1.
- **x_vals** (*list*) – (Optional) The list of x-values to use. If not given, defaults to `p_order + 1` evenly spaced points on $[0, 1]$.

Return type tuple

Returns Pair of mass and stiffness matrices, square sympy.Matrix with rows/columns equal to `p_order + 1`.

assignment1.dg1_symbolic.**get_symbolic_vandermonde**(*p_order*, *x_vals=None*)

Get symbolic Vandermonde matrix of evenly spaced points.

Parameters

- **p_order** (*int*) – The degree of precision for the method.
- **x_vals** (*list*) – (Optional) The list of x-values to use. If not given, defaults to `p_order + 1` evenly spaced points on $[0, 1]$.

Return type tuple

Returns Pair of vector of powers of x and Vandermonde matrix. Both are type sympy.Matrix, the `x_vec` is a row vector with `p_order + 1` columns and the Vandermonde matrix is square of dimension `p_order + 1`.

1.1.4 assignment1.plotting module

Plotting helpers for `dg1` solver.

class assignment1.plotting.**DG1Animate**(*solver*, *fig=None*, *ax=None*, *interp_points=None*)

Bases: object

Helper for animating a solution.

Assumes the solution (which is updated via `solver`) produces a solution that remains in the same bounding box as $u(x, 0)$ (give or take some noise).

Parameters

- **solver** (*dg1.DG1Solver*) – The solver which computes and updates the solution.
- **fig** (matplotlib.figure.Figure) – (Optional) A figure to use for plotting. Intended to be passed when creating a matplotlib.animation.FuncAnimation.
- **ax** (matplotlib.artist.Artist) – (Optional) An axis to be used for plotting.
- **interp_points** (*int*) – (Optional) The number of points to use to represent polynomials on an interval. Defaults to *INTERVAL_POINTS*.

Raises `ValueError` if one of `fig` or `ax` is passed, but not both.

init_func()
An initialization function for the animation.

Plots the initial data **and** stores the lines created.

Return type `list` of `matplotlib.lines.Line2D`

Returns List of the updated matplotlib line objects, with length equal to n (coming from `solver`).

update_plot(*frame_number*)
Update the lines in the plot.

First advances the solver and then uses the updated value to update the `matplotlib.lines.Line2D` objects associated to each interval.

Parameters **frame_number** (*int*) – (Unused) The current frame.

Return type `list` of `matplotlib.lines.Line2D`

Returns List of the updated matplotlib line objects, with length equal to n (coming from `solver`).

Raises `ValueError` if the frame number doesn't make the current step on the solver.

assignment1.plotting.**INTERVAL_POINTS = 10**
Number of points to use when plotting a polynomial on an interval.

class assignment1.plotting.**PolynomialInterpolate**(*x_vals*, *num_points=None*)
Bases: `object`

Polynomial interpolation from node points.

Assumes the first and last x-value are the endpoints of the interval.

Using Lagrange basis polynomials we can write our polynomial as

$$p(x) = \sum_j y_j \ell_j(x)$$

and we can compute $\ell_j(x)$ of our data without ever computing the coefficients. We do this by computing all pairwise differences of our x-values and the interpolating values. Then we take the products of these differences (leaving out one of the interpolating values).

Parameters

- **x_vals** (`numpy.ndarray`) – List of x-values that uniquely define a polynomial. The degree is one less than the number of points.
- **num_points** (*int*) – (Optional) The number of points to use to represent the polynomial. Defaults to *INTERVAL_POINTS*.

classmethod from_solver(*solver*, *num_points=None*)
Polynomial interpolation factory from a solver.

The reference interval for the interpolation is assumed to be in the first column of the x-values stored on the solver.

Parameters

- **solver** (*dg1.DG1Solver*) – A solver containing x-values.
- **num_points** (*int*) – (Optional) The number of points to use to represent the polynomial. Defaults to *INTERVAL_POINTS*.

Return type *PolynomialInterpolate*

Returns Interpolation object for the reference

interpolate(*y_vals*)

Evaluate interpolated polynomial given y-values.

We've already pre-computed the values $\ell_j(x)$ for all the x-values we use in our interval (`num_points` in all, using the interpolating x-values to compute the $\ell_j(x)$). So we simply use them to compute

$$p(x) = \sum_j y_j \ell_j(x)$$

using the y_j from `y_vals`.

Parameters **y_vals** (numpy.ndarray) – Array of y-values that uniquely define our interpolating polynomial. If 1D, converted into a column vector before returning.

Return type numpy.ndarray

Returns 2D array containing $p(x)$ for each x-value in the interval (`num_points` in all). If there are multiple columns in `y_vals` (i.e. multiple $p(x)$) then each column of the result will corresponding to each of these polynomials evaluated at `all_x`.

assignment1.plotting.**make_lagrange_matrix**(*x_vals*, *all_x*)

Make matrix where $M_{ij} = \ell_j(x_i)$.

This matrix contains the Lagrange interpolating polynomials evaluated on the interval given by `x_vals`. The x_i (corresponding to rows in M) are the `num_points` possible x-values in `all_x` and the ℓ_j (corresponding to columns in M) are the Lagrange interpolating polynomials interpolated on the points in `x_vals`.

Parameters

- **x_vals** (numpy.ndarray) – 1D array of x-values used to interpolate data via Lagrange basis functions.
- **all_x** (numpy.ndarray) – 1D array of points to evaluate the $\ell_j(x)$' at.

Return type numpy.ndarray

Returns The matrix M.

assignment1.plotting.**plot_convergence**(*p_order*, *interval_sizes*, *colors*, *solver_factory*, *interval_width=1.0*, *total_time=1.0*, *points_on_ref_int=None*)

Plot a convergence plot for a given order.

Creates a side-by-side of error plots and the solutions as the mesh is refined.

Parameters

- **p_order** (*int*) – The order of accuracy desired.
- **interval_sizes** (numpy.ndarray) – Array of n values to use for the number of sub-intervals.
- **colors** (*list*) – List of triples RGB (each a color). Expected to be the same length as `interval_sizes`.
- **solver_factory** (*type*) – Class that can be used to construct a solver.
- **interval_width** (*float*) – (Optional) The width of the interval where the solver works. Defaults to 1.0.
- **total_time** (*float*) – (Optional) The total time to run the solver. Defaults to 1.0.

- **points_on_ref_int** (function) – (Optional) The method used to partition the reference interval $[0, h]$ into node points. Defaults to `get_evenly_spaced_points()`.

assignment1.plotting.**plot_solution**(*color*, *num_cols*, *interp_func*, *solver*, *ax*)

Plot the solution and return the newly created lines.

Helper for *DG1Animate*.

Parameters

- **color** (*str*) – The color to use in plotting the solution.
- **num_cols** (*int*) – The number of columsn in the `solution`.
- **interp_func** (*PolynomialInterpolate*) – The polynomial interpolation object used to map a solution onto a set of points.
- **solver** (*dg1.DG1Solver*) – A solver containing a `solution` and `node_points`.
- **ax** (matplotlib.artist.Artist) – An axis to be used for plotting.

Return type list of matplotlib.lines.Line2D

Returns List of the updated matplotlib line objects.

1.2 Module contents

Package for first assignment in M273.

CHAPTER 2

class_preso package

2.1 Submodules

2.1.1 class_preso.weno_computations module

Helper functions for `weno_computations` notebook.

Slides can be seen on nbviewer.

`class_preso.weno_computations.`**`discontinuity_to_volume`**`()`
Make plots similar to introductory, but with a discontinuity.

`class_preso.weno_computations.`**`discontinuity_to_volume_single_cell`**`(`*stopping_point=None*`)`
Plot a piecewise constant function w/discontinuity towards the left.

Parameters **`stopping_point`** (*int*) – (Optional) The transition point to stop at when creating the plot. By passing in 0, 1, 2, ... this allows us to create a short slide-show.

`class_preso.weno_computations.`**`interp_simple_stencils`**`()`
Return interpolated values for $u_{j+1/2}$ using simple stencils.

First uses three sets of interpolating values,

$$\{\overline{u}_{j-2}, \overline{u}_{j-1}, \overline{u}_j\}, \{\overline{u}_{j-1}, \overline{u}_j, \overline{u}_{j+1}\}, \{\overline{u}_j, \overline{u}_{j+1}, \overline{u}_{j+2}\},$$

to give local order three approximations.

Then uses all five points $\{x_{j-2}, x_{j-1}, x_j, x_{j+1}, x_{j+2}\}$ to give an order five approximation on the whole stencil.

Return type tuple

Returns Quadraple of LaTeX strings, one for each set of interpolating points, in the order described above.

`class_preso.weno_computations.`**`make_intro_plots`**`(`*stopping_point=None*`)`
Make introductory plots.

Uses

$$\overline{u}_{-2} = 0, \overline{u}_{-1} = 3, \overline{u}_0 = 2, \overline{u}_1 = -1, \overline{u}_2 = 2$$

And plots the interpolations by quadratics (on the three contiguous subregions) and by a quartic that preserve the interval.

class_preso.weno_computations.**make_shock_plot**()
Make plots similar to introductory, but with a discontinuity.

class_preso.weno_computations.**make_shock_plot_single_cell**()
Plot the reconstructed polynomials that occur near a shock.

class_preso.weno_computations.**to_latex**(*value*, *replace_dict*)
Convert an expression to LaTeX.

This method is required so we can get a specified ordering for terms that may not have the desired lexicographic ordering.

Parameters

- **value** (sympy.core.expr.Expr) – A
- **replace_dict** (*dict*) – Dictionary where keys are old variable names (as strings) and values are the new variable names to replace them with.

Return type str

Returns The value as LaTeX, with all variables replaced.

2.2 Module contents

Package for class presentation in M273.

Python Module Index

a

c

Index