PyWENO

Release 0.11.2

December 17, 2014

Contents

The PyWENO project provides a set of open source tools for constructing high-order Weighted Essentially Non-oscillatory (WENO) methods and performing high-order WENO reconstructions.

PyWENO consists of four main parts:

- *WENO toolkit* - an easy to use toolkit to easily compute WENO reconstructions in Python.
- *Symbolics* - tools for exploring and constructing WENO methods.
- *Code generation* - tools for generating custom C, Fortran, and OpenCL WENO routines.
- *Non-uniform* - tools for generating WENO methods on non-uniform grids.

News

- December 4 2013: The kernel generator has been simplified a lot and the functional generator was removed. Several more (speed) improvements were made to the non-uniform module.
- November 12 2013: Several improvements were made to the non-uniform module. These were contributed by Ben Thompson.
- May 15 2012: Several routines were added for computing reconstructions of derivatives. These were contributed by Michael Welter.
- January 23 2012: The non-uniform codes have been resurrected.

Please check out the documentation (below) or the PyWENO project page for more information about using and contributing to PyWENO.

CHAPTER 1

Documentation

Main parts of the documentation

- *WENO tutorial* - basic WENO reconstructions.
- *Symbolics* - the symbolic tool kit.
- *Code generation* - the code generation tool kit.
- *Non-uniform* - the non-uniform grid tool kit.
- *Reference* - reference documentation.
- *Download* - download and installation instructions.

CHAPTER 2

Contributing

Contributions are welcome! Please send comments, suggestions, and/or patches to the primary author (Matthew Emmett). You will be credited.

2.1 WENO toolkit

2.1.1 WENO reconstructions

High-order WENO reconstructions for 1d arrays of cell-average quantities can be computed with the `pyweno.weno` module.

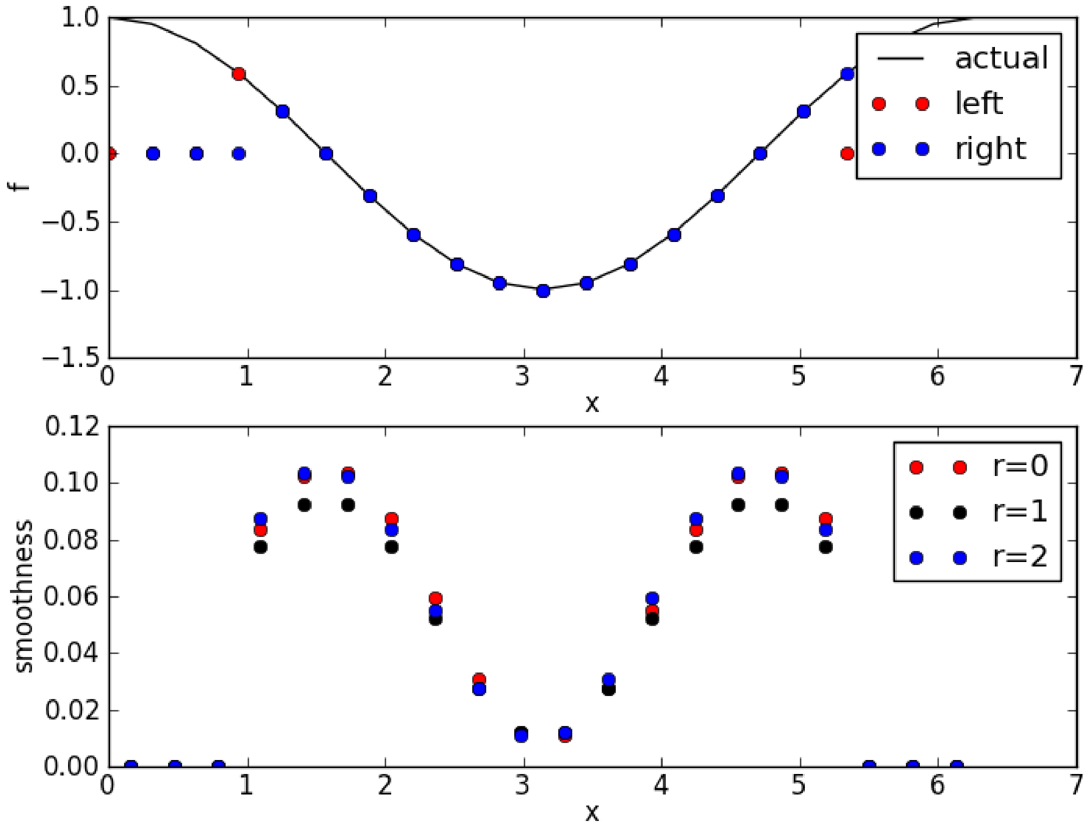
For example, to reconstruct *sin(x)* at the left edge of each cell to fifth order accuracy:

```
>>> import numpy as np
>>> import pyweno.weno
>>> x = np.linspace(0.0, 2*np.pi, 21)
>>> f = (np.cos(x[1:]) - np.cos(x[:-1])) / (x[1] - x[0])
>>> q = pyweno.weno.reconstruct(f, 5, 'left')
```

Please see the *reference documentation* for more information.

Smooth reconstruction

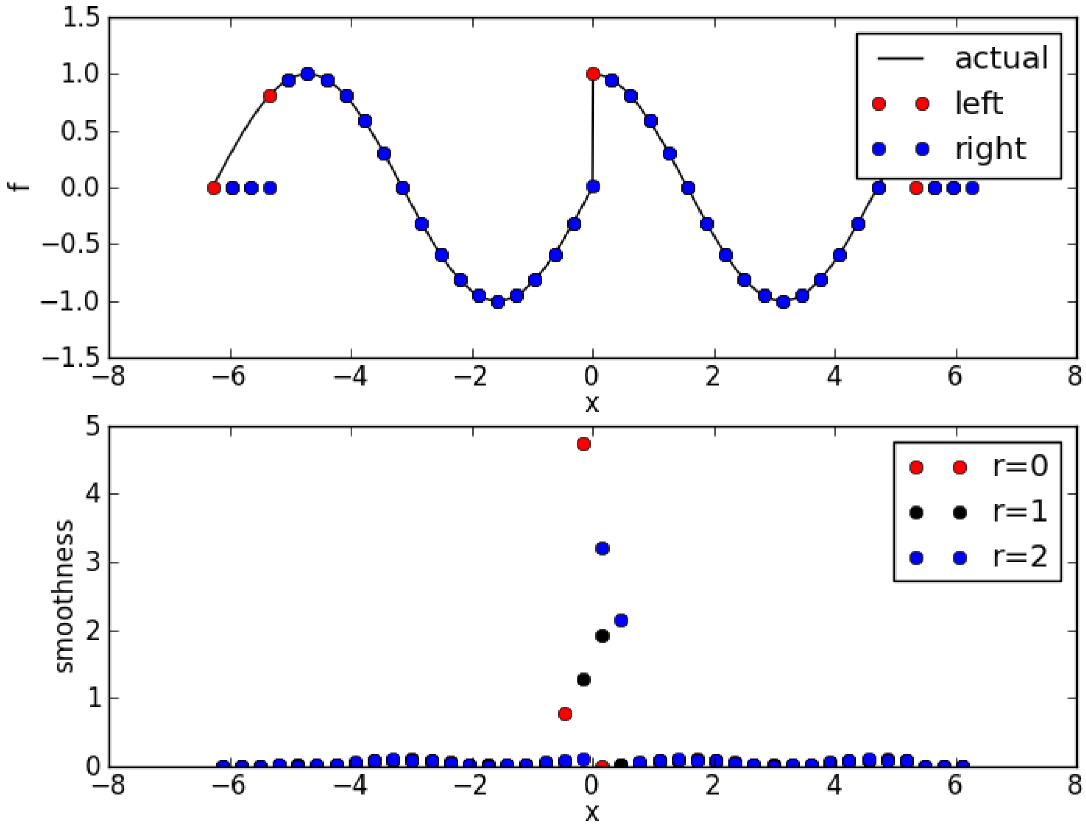
Here we reconstruct *sin(x)* at the left and right edges of each cell to fifth order accuracy and plot the results:



The code to generate the above is in `smooth.py`.

Discontinuous reconstruction

Here we reconstruct a discontinuous function (*sin(x)* for *x<0*, *cos(x)* for *x>0*) at the left and right edges of each cell to fifth order accuracy and plot the results:



The code to generate the above is in `discontinuous.py`.

2.1.2 Version information

Here we obtain the version of PyWENO:

```
>>> import pyweno.version
>>> pyweno.version.version()
>>> pyweno.version.git_version()
```

2.2 Symbolics

PyWENO contains a symbolic module to help authors develop their own WENO methods and/or explore the basics of WENO methods. Below are a few quick examples demonstrating how the symbolic routines of PyWENO are used.

2.2.1 Interpolating polynomials

First, let's build some grid points and y-values:

```
>>> import sympy
>>> import pyweno
```

```
>>> (x0, x1, x2) = sympy.var('x0 x1 x2')
>>> (y0, y1, y2) = sympy.var('y0 y1 y2')
```

Then, the polynomial that interpolates the points `(x0, y0)`, `(x1, y1)`, and `(x2, y2)` is given by:

```
>>> p = pyweno.symbolic.polynomial_interpolator([x0, x1, x2], [y0, y1, y2])
>>> p
y0*(x - x1)*(x - x2)/((x0 - x1)*(x0 - x2)) + y1*(x - x0)*(x - x2)/((x1 - x0)*(x1 - x2)) + y2*(x - x0
```

and is a function of the SymPy variable `x`. For example:

```
>>> x = sympy.var('x')
>>> p.subs(x, x2)
y2
```

The polynomial that interpolates the primitive function of f such that

$$f(x_i) = \sum_j y_j (x_{j+1} - x_j)$$

is given by:

```
>>> P = pyweno.symbolic.primitive_polynomial_interpolator([x0, x1, x2], [y1, y2])
>>> P
y1*(x - x0)*(x - x2)/(x1 - x2) + (x - x0)*(x - x1)*(y1*(x1 - x0) + y2*(x2 - x1))/((x2 - x0)*(x2 - x1
```

and is also a function of the SymPy variable `x`. For example:

```
>>> P.subs(x, x1)
y1*(x1 - x0)
```

For uniform grids, one could define the grid points by:

```
>>> (x, dx) = sympy.var('x dx')
>>> xs = [ dx, 2*dx, 3*dx ]
>>> p = pyweno.symbolic.polynomial_interpolator(xs, [y0, y1, y2])
>>> p
y0*(x - 3*dx)*(x - 2*dx)/(2*dx**2) + y2*(x - dx)*(x - 2*dx)/(2*dx**2) - y1*(x - dx)*(x - 3*dx)/dx**2
```

2.2.2 Reconstruction coefficients

Hereafter we assume that the grid is uniform. Furthermore, to specify a point within a cell, the interval [-1, 1] is used as a reference.

The reconstruction coefficients for a 5th (=2k-1 where k=3) order WENO scheme corresponding to the reconstruction point at the left side (ξ = -1) of each grid cell are given by:

```
>>> c = pyweno.symbolic.reconstruction_coefficients(k=3, xi=[ -1 ])
>>> c
{'k': 3,
 'n': 1,
 (0, 0, 0): 11/6,
 (0, 0, 1): -7/6,
 (0, 0, 2): 1/3,
 (0, 1, 0): 1/3,
 (0, 1, 1): 5/6,
 (0, 1, 2): -1/6,
 (0, 2, 0): -1/6,
```

```
 (0, 2, 1): 5/6,
 (0, 2, 2): 1/3
```

Note that the return value `c` is a dictionary of SymPy objects, indexed according to `c[l,r,j]` where `l` is the index of the reconstruction point and `r` is the left-shift of the stencil.

Recall that the reconstruction coefficients c are used to reconstruct the original (unknown) function f at each point ξ_l in `xi` according to

$$f^r(\xi^l) \approx \sum_{j=0}^{k-1} c^l_{r,j}\, \bar{f}_{i-r+j}$$

for each l from 0 to *len(xi)*, where $\bar{f}_{i-r+j}$ is the cell average of f in the cell $i - r + j$.

2.2.3 Optimal weights

The optimal weights for a 5th (=2k-1 where k=3) order WENO scheme corresponding to the reconstruction point at the left side of each grid cell are given by:

```
>>> w = pyweno.symbolic.optimal_weights(3, [ -1 ])
>>> w
({'k': 3, 'n': 1, (0, 0): 1/10, (0, 1): 3/5, (0, 2): 3/10}, {0: False, 'n': 1})
```

Note that the return value *w* is a tuple of dictionaries of SymPy objects. The first dictionary contains the weights, and is indexed according to `w[l,r]`. the second dictionary contains boolean values determining if the weights are split (negative).

Recall that the optimal weights are used to obtain an optimally high-order reconstruction of the original function f given the low-order reconstructions $f^r(\xi^l)$ according to

$$f(\xi^l) \approx \sum_{r=0}^{k-1} \varpi^{l,r} f^r(\xi_l).$$

2.2.4 Smoothness coefficients

The Jiang-Shu smoothness coefficients for a 5th (=2k-1 where k=3) order WENO scheme are given by:

```
>>> beta = pyweno.symbolic.jiang_shu_smoothness_coefficients(3)
```

The return value *beta* is a dictionary of SymPy objects, and is indexed according to `beta[r,m,n]` (see the reference documentation for details).

Recall that the smoothness coefficients `beta[r, m, n]` are used to compute the non-linear weights $\omega^{l,r}$ (used in place of $\varpi^{l,r}$ in non-smooth regions) according to

$$\omega^{l,r} = \frac{\alpha^{l,r}}{\alpha^{l,0} + \cdots + \alpha_{l,k-1}}$$

where

$$\alpha^{l,r} = \frac{\varpi^{l,r}}{(\epsilon + \sigma^r)^p}$$

and

$$\sigma^r = \sum_{m=1}^{2k-1} \sum_{n=1}^{2k-1} \beta_{r,m,n}\, \overline{f}_{i-k+m}\, \overline{f}_{i-k+n}.$$

2.3 Code generation

PyWENO contains two modules to help authors generate WENO codes in C-like languages (ie, C, C++, OpenCL, and CUDA), and Fortran. The code-generation process is done in two stages:

1. Use the *symbolic* package to compute reconstruction coefficients and optimal weights (or provide your own).
2. Generate either code snippets (kernels) to use as building blocks for your own code.

2.3.1 Kernels

First, let's generate a Fortran kernel to reconstruct at the left and right endpoints of a cell to fifth order:

```
>>> import pyweno
>>> k = pyweno.kernels.KernelGenerator('Fortran', order=5, xi=[-1,1])
>>> print k.smoothness()
sigma0 = 3.33333333333333333333333333333333d0*f(i+0)*f(i+0) - &
      10.3333333333333333333333333333333d0*f(i+0)*f(i+1) + &
      3.66666666666666666666666666666667d0*f(i+0)*f(i+2) + &
      8.33333333333333333333333333333333d0*f(i+1)*f(i+1) - &
      6.33333333333333333333333333333333d0*f(i+1)*f(i+2) + &
      1.33333333333333333333333333333333d0*f(i+2)*f(i+2)
sigma1 = 4.33333333333333333333333333333333d0*f(i+0)*f(i+0) - &
      4.33333333333333333333333333333333d0*f(i+0)*f(i+1) - &
      4.33333333333333333333333333333333d0*f(i+0)*f(i-1) + &
      1.33333333333333333333333333333333d0*f(i+1)*f(i+1) + &
      1.66666666666666666666666666666667d0*f(i+1)*f(i-1) + &
      1.33333333333333333333333333333333d0*f(i-1)*f(i-1)
sigma2 = 3.33333333333333333333333333333333d0*f(i+0)*f(i+0) - &
      10.3333333333333333333333333333333d0*f(i+0)*f(i-1) + &
      3.66666666666666666666666666666667d0*f(i+0)*f(i-2) + &
      8.33333333333333333333333333333333d0*f(i-1)*f(i-1) - &
      6.33333333333333333333333333333333d0*f(i-1)*f(i-2) + &
      1.33333333333333333333333333333333d0*f(i-2)*f(i-2)
>>> print k.weights()
omega0 = 0.1d0/(sigma0 + 0.000000999999999999999995474811182588625869d0)*(sigma0 + &
      0.000000999999999999999995474811182588625869d0)
omega1 = 0.6d0/(sigma1 + 0.000000999999999999999995474811182588625869d0)*(sigma1 + &
      0.000000999999999999999995474811182588625869d0)
omega2 = 0.3d0/(sigma2 + 0.000000999999999999999995474811182588625869d0)*(sigma2 + &
      0.000000999999999999999995474811182588625869d0)
acc = omega0 + omega1 + omega2
omega0 = omega0/acc
omega1 = omega1/acc
omega2 = omega2/acc
omega3 = 0.3d0/(sigma0 + 0.000000999999999999999995474811182588625869d0)*(sigma0 + &
      0.000000999999999999999995474811182588625869d0)
omega4 = 0.6d0/(sigma1 + 0.000000999999999999999995474811182588625869d0)*(sigma1 + &
      0.000000999999999999999995474811182588625869d0)
omega5 = 0.1d0/(sigma2 + 0.000000999999999999999995474811182588625869d0)*(sigma2 + &
      0.000000999999999999999995474811182588625869d0)
acc = omega3 + omega4 + omega5
omega3 = omega3/acc
omega4 = omega4/acc
omega5 = omega5/acc
>>> print k.reconstruction()
fr0 = 1.83333333333333333333333333333333d0*f(i+0) - &
```

```
      1.16666666666666666666666666666666667d0*f(i+1) + &
      0.33333333333333333333333333333333333d0*f(i+2)
fr1 = 0.83333333333333333333333333333333333d0*f(i+0) - &
      0.16666666666666666666666666666666667d0*f(i+1) + &
      0.33333333333333333333333333333333333d0*f(i-1)
fr2 = 0.33333333333333333333333333333333333d0*f(i+0) + &
      0.83333333333333333333333333333333333d0*f(i-1) - &
      0.16666666666666666666666666666666667d0*f(i-2)
fr3 = 0.33333333333333333333333333333333333d0*f(i+0) + &
      0.83333333333333333333333333333333333d0*f(i+1) - &
      0.16666666666666666666666666666666667d0*f(i+2)
fr4 = 0.83333333333333333333333333333333333d0*f(i+0) + &
      0.33333333333333333333333333333333333d0*f(i+1) - &
      0.16666666666666666666666666666666667d0*f(i-1)
fr5 = 1.83333333333333333333333333333333333d0*f(i+0) - &
      1.16666666666666666666666666666666667d0*f(i-1) + &
      0.33333333333333333333333333333333333d0*f(i-2)
fs0 = fr0*omega0 + fr1*omega1 + fr2*omega2
fs1 = fr3*omega3 + fr4*omega4 + fr5*omega5
```

As you can see, the code snippets above assume that you have defined several variables, and that the cell-averaged unknown is stored in the `f` vector (indexed by `i`). Finally, the last snippet stores the reconstructed values in `fs0` and `fs1`.

You can optionally change the names of the variables used above, and you can also supply your own reconstruction coeffs, optimal weights, and smoothness coefficients instead of having them automactially computed.

Please see the reference documentation for more information.

2.4 Non-uniform grids

PyWENO contains a module to compute all of the various WENO reconstruction coefficients on non-uniform grids. The routines to compute the WENO coefficients in the non-uniform module are very similiar to those in the symbolic module, except they accept lists (or NumPy arrays) of cell edges and return NumPy arrays.

Let's compute the reconstruction coefficients to reconstruct at the left and right endpoints of a cell to fifth order:

```
>>> import pyweno.nonuniform
>>> edges = [ 0.0, 1.0, 2.5, 3.9, 4.7, 5.5, 6.3, 7.8, 8.8, 9.9, 10.5 ]
>>> c, beta, varpi = pyweno.nonuniform.coefficients(3, [ -1, 1 ], edges)
>>> c[5]
array([[[ 1.59025033, -0.81328063,  0.2230303 ],
        [ 0.37096774,  0.71879383, -0.08976157],
        [-0.16666667,  0.83333333,  0.33333333]],

       [[ 0.49407115,  0.6513834 , -0.14545455],
        [-0.24193548,  1.06241234,  0.17952314],
        [ 0.33333333, -1.16666667,  1.83333333]]])
```

Note that the `c` array is indexed according to `c[i,l,r,j]` where `i` is the cell number, `l` is the reconstruction point (eg, 0 is the left edge and 1 is the right edge), `r` is the left-shift of the stencil, and `j` is the summation index.

2.5 Reference

2.5.1 WENO toolkit

2.5.2 Symbolics

2.5.3 Code generation

Kernels

2.5.4 Non-uniform reconstructions

2.5.5 Version

2.6 Downloading and installing

2.6.1 From PyPI

The PyWENO package is registered on the Python package index. If you have pip installed, you can install PyWENO by:

```
$ pip install pyweno
```

2.6.2 From github

The latest source distribution is also available in either zip or tar format. Finally, you can also obtain the source code on GitHub through the PyWENO project page.

2.6.3 Tracking the development repo

You can clone the project by running:

```
$ git clone git://github.com/memmett/PyWENO
```