PyPDE

Dec 18, 2019

Contents:

A Python library for solving any system of hyperbolic or parabolic Partial Differential Equations. The PDEs can have stiff source terms and non-conservative components.

Key Features:

- Any first or second order system of PDEs
- Your fluxes and sources are written in Python for ease
- Any number of spatial dimensions
- Arbitrary order of accuracy
- C++ under the hood for speed
- Based on the ADER-WENO method

Please feel free to message me with questions/suggestions: jackson.haran@gmail.com

Quickstart: check out the *core functionality* and *example code*.

CHAPTER 1

Installation

1.1 From PyPI

There are pre-built wheels for Linux, MacOS, and Windows (Python 3.6, 3.7, 3.8) avaiable on PyPI. To install them from your system, run:

```
pip install pypde
```

1.2 From source

Ensure you have a C++ compiler installed (e.g. Clang/g++ on Linux/MacOS, or MSVC on Windows). Then run:

```
$ git clone git://github.com/haranjackson/PyPDE.git
```

Then run the following commands:

```
$ cd PyPDE
$ pip install .
```

CHAPTER 2

Background

We can solve any system of PDEs of the form:

$$\begin{aligned}\frac{\partial \mathbf{Q}}{\partial t} + \frac{\partial}{\partial x_1}\mathbf{F}_1\left(\mathbf{Q}, \frac{\partial \mathbf{Q}}{\partial x_1}, \ldots, \frac{\partial \mathbf{Q}}{\partial x_n}\right) + \cdots + \frac{\partial}{\partial x_n}\mathbf{F}_n\left(\mathbf{Q}, \frac{\partial \mathbf{Q}}{\partial x_1}, \ldots, \frac{\partial \mathbf{Q}}{\partial x_n}\right) \\ + B_1(\mathbf{Q})\frac{\partial \mathbf{Q}}{\partial x_1} + \cdots + B_n(\mathbf{Q})\frac{\partial \mathbf{Q}}{\partial x_n} \\ = \mathbf{S}(\mathbf{Q})\end{aligned}$$

or, more succinctly:

$$\frac{\partial \mathbf{Q}}{\partial t} + \nabla \mathbf{F}(\mathbf{Q}, \nabla \mathbf{Q}) + B(\mathbf{Q}) \cdot \nabla \mathbf{Q} = \mathbf{S}(\mathbf{Q})$$

See *examples of such systems* .

If you give the values of $\mathbf{Q}$ at time $t = 0$ on a rectangular domain in $\mathbb{R}^n$, then PyPDE will calculate $\mathbf{Q}$ on the domain at a later time t_f that you specify.

The boundary conditions at the edges of the domain can be either transitive or periodic.

CHAPTER 3

Core Functionality

pypde.solvers.**pde_solver**(*Q0*, *tf*, *L*, *F=None*, *B=None*, *S=None*, *boundaryTypes='transitive'*, *cfl=0.9*, *order=2*, *ndt=100*, *flux='rusanov'*, *stiff=True*, *nThreads=-1*)

Solves PDEs of the following form:

$$\frac{\partial \mathbf{Q}}{\partial t} + \frac{\partial}{\partial x_1}\mathbf{F}_1\left(\mathbf{Q}, \frac{\partial \mathbf{Q}}{\partial x_1}, \ldots, \frac{\partial \mathbf{Q}}{\partial x_n}\right) + \cdots + \frac{\partial}{\partial x_n}\mathbf{F}_n\left(\mathbf{Q}, \frac{\partial \mathbf{Q}}{\partial x_1}, \ldots, \frac{\partial \mathbf{Q}}{\partial x_n}\right) + B_1(\mathbf{Q})\frac{\partial \mathbf{Q}}{\partial x_1} + \cdots + B_n(\mathbf{Q})\frac{\partial \mathbf{Q}}{\partial x_n} = \mathbf{S}(\mathbf{Q})$$

or, more succinctly:

$$\frac{\partial \mathbf{Q}}{\partial t} + \nabla \mathbf{F}(\mathbf{Q}, \nabla \mathbf{Q}) + B(\mathbf{Q}) \cdot \nabla \mathbf{Q} = \mathbf{S}(\mathbf{Q})$$

where $\mathbf{Q}, \mathbf{F}_i, \mathbf{S}$ are vectors of n_{var} variables and B_i are matrices of shape $(n_{var} \times n_{var})$. Of course, $\mathbf{F}, \mathbf{B}, \mathbf{S}$ can be 0.

Define $\Omega = [0, L_1] \times \cdots \times [0, L_n]$. Given $\mathbf{Q}(\mathbf{x}, 0)$ for $\mathbf{x} \in \Omega$, `pde_solver` finds $\mathbf{Q}(\mathbf{x}, t)$ for any $t > 0$.

Taking integers $m_1, \ldots, m_n > 0$ we split Ω into $(m_1 \times \cdots \times m_n)$ cells with volume $dx_1 dx_2 \ldots dx_n$ where $dx_i = \frac{L_i}{m_i}$.

Parameters

- **Q0** (*ndarray*) – An array with shape $(m_1, \ldots m_n, n_{var})$.

  `Q0[i1, i2, ..., in, j]` is equal to:

  $$\mathbf{Q}_{j+1}\left(\frac{(i_1 + 0.5)\, dx_1}{m_1}, \ldots, \frac{(i_n + 0.5)\, dx_n}{m_n}, 0\right)$$

- **tf** (*double*) – The final time at which to return the value of $\mathbf{Q}(\mathbf{x}, t)$.

- **L** (*ndarray or list*) – An array of length n, `L[i]` is equal to L_{i+1}.
- **F** (*callable, optional*) – The flux terms, with signature `F(Q, DQ, d) -> ndarray`, corresponding to $\mathbf{F}_{d+1}(\mathbf{Q}, \nabla\mathbf{Q})$.

  If $\mathbf{F}$ has no $\nabla\mathbf{Q}$ dependence, the signature may be `F(Q, d) -> ndarray`, corresponding to $\mathbf{F}_{d+1}(\mathbf{Q})$.

  If $n = 1$ and $\mathbf{F}$ has no $\nabla\mathbf{Q}$ dependence, the signature may be `F(Q) -> ndarray`, corresponding to $\mathbf{F}_1(\mathbf{Q})$.

  – `Q` is a 1-D array with shape $(n_{var},)$

  – `DQ` is an 2-D array with shape (n, n_{var}) (as `DQ[i]` is equal to $\frac{\partial \mathbf{Q}}{\partial x_i}$)

  – `d` is an integer (ranging from 0 to $n - 1$)

  – the returned array has shape $(n_{var},)$
- **B** (*callable, optional*) – The non-conservative terms, with signature `B(Q, d) -> ndarray`, corresponding to $\mathbf{B}_{d+1}(\mathbf{Q})$.

  If $n = 1$, the signature may be `B(Q) -> ndarray`, corresponding to $\mathbf{B}_1(\mathbf{Q})$.

  – `Q` is a 1-D array with shape $(n_{var},)$

  – `d` is an integer (ranging from 0 to $n - 1$)

  – the returned array has shape (n_{var}, n_{var})
- **S** (*callable, optional*) – The source terms, with signature `B(Q, d) -> ndarray`, corresponding to $\mathbf{S}(\mathbf{Q})$.

  – `Q` is a 1-D array with shape $(n_{var},)$

  – the returned array has shape $(n_{var},)$
- **boundaryTypes** (*string or list, optional*) –

  – If a string, must be one of `'transitive'`, `'periodic'`. In this case, all boundaries will take the stated form.

  – If a list, must have length n, containing strings taken from `'transitive'`, `'periodic'`. In this case, the first element of the list describes the boundaries at $x_1 = 0, L_1$, the second describes the boundaries at $x_2 = 0, L_2$, etc.
- **cfl** (*double, optional*) – The CFL number: $0 < CFL < 1$ (default 0.9).
- **order** (*int, optional*) – The order of the polynomial reconstructions used in the solver, must be an integer greater than 0 (default 2)
- **ndt** (*int, optional*) – The number of timesteps at which to return the value of the grid (default 100) e.g. if `tf=5`, and `ndt=4` then the value of the grid will be returned at `t=1.25,2.5,3.75,5`.
- **flux** (*string, optional*) – The kind of flux to use at cell boundaries (default `'rusanov'`) Must be one of `'rusanov'`, `'roe'`, `'osher'`.
- **stiff** (*bool, optional*) – Whether to use a stiff solver for the Discontinuous Galerkin step (default True) If the equations are stiff (i.e. the source terms are much larger than the other terms), then this is probably required to make the method converge. Otherwise, it can be turned off to improve speed.
- **nThreads** (*int, optional*) – The number of threads to use in the solver (default -1). If less than 1, the thread count will default to (number of cores) - 1.

Returns

out – An array with shape $(ndt, m_1, \ldots m_n, n_{var})$.

Defining $dt = \frac{tf}{ndt}$, then `out[i, i1, i2, ..., in, j]` is equal to:

$$\mathbf{Q}_{j+1}\left(\frac{(i_1 + 0.5)\,dx_1}{m_1}, \ldots, \frac{(i_n + 0.5)\,dx_n}{m_n}, (i+1)\,dt\right)$$

Return type ndarray

CHAPTER 4

Example PDEs

The following PDEs all have the form solvable by PyPDE:

$$\frac{\partial \mathbf{Q}}{\partial t} + \nabla \mathbf{F}\left(\mathbf{Q}, \nabla \mathbf{Q}\right) + B\left(\mathbf{Q}\right) \cdot \nabla \mathbf{Q} = \mathbf{S}\left(\mathbf{Q}\right)$$

4.1 3D Navier-Stokes

$$\mathbf{Q} = \begin{pmatrix} \rho \\ \rho E \\ \rho v_1 \\ \rho v_2 \\ \rho v_3 \end{pmatrix} \quad \mathbf{F}_i = \begin{pmatrix} \rho v_i \\ \rho E v_i + \mathbf{\Sigma_i} \cdot \mathbf{v} \\ \rho v_i v_1 + \mathbf{\Sigma_{i1}} \\ \rho v_i v_2 + \mathbf{\Sigma_{i2}} \\ \rho v_i v_3 + \mathbf{\Sigma_{i3}} \end{pmatrix} \quad B_i = 0 \quad \mathbf{S} = 0$$

where:

$$\Sigma = pI - \mu\left(\nabla \mathbf{v} + \nabla \mathbf{v}^T - \frac{2}{3} tr\left(\nabla \mathbf{v}\right) I\right)$$

4.2 2D Reactive Euler

$$\mathbf{Q} = \begin{pmatrix} \rho \\ \rho E \\ \rho v_1 \\ \rho v_2 \\ \rho \lambda \end{pmatrix} \quad \mathbf{F}_i = \begin{pmatrix} \rho v_i \\ \left(\rho E + p\right) v_i \\ \rho v_i v_1 + \delta_{i1} p \\ \rho v_i v_2 + \delta_{i2} p \\ \rho v_i \lambda \end{pmatrix} \quad B_i = 0 \quad \mathbf{S} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -\rho \lambda K\left(T\right) \end{pmatrix}$$

where K is a (potentially large) function depending on temperature T.

4.3 3D Godunov-Romenski

$$\mathbf{Q} = \begin{pmatrix} \rho \\ \rho E \\ \rho v_1 \\ \rho v_2 \\ \rho v_3 \\ A_{11} \\ A_{12} \\ A_{13} \\ A_{21} \\ A_{22} \\ A_{23} \\ A_{31} \\ A_{32} \\ A_{33} \end{pmatrix} \quad \mathbf{F}_i = \begin{pmatrix} \rho v_i \\ \rho E v_i + \mathbf{\Sigma_i} \cdot \mathbf{v} \\ \rho v_i v_1 + \mathbf{\Sigma_{i1}} \\ \rho v_i v_2 + \mathbf{\Sigma_{i2}} \\ \rho v_i v_3 + \mathbf{\Sigma_{i3}} \\ \delta_{i1} \mathbf{A_1} \cdot \mathbf{v} \\ \delta_{i2} \mathbf{A_1} \cdot \mathbf{v} \\ \delta_{i3} \mathbf{A_1} \cdot \mathbf{v} \\ \delta_{i1} \mathbf{A_2} \cdot \mathbf{v} \\ \delta_{i2} \mathbf{A_2} \cdot \mathbf{v} \\ \delta_{i3} \mathbf{A_2} \cdot \mathbf{v} \\ \delta_{i1} \mathbf{A_3} \cdot \mathbf{v} \\ \delta_{i2} \mathbf{A_3} \cdot \mathbf{v} \\ \delta_{i3} \mathbf{A_3} \cdot \mathbf{v} \end{pmatrix} \quad B_i = v_i I_{14} - \begin{pmatrix} 0_5 & 0_3 & 0_3 & 0_3 \\ 0_3 & \delta_{i1} v_1 I_3 & \delta_{i1} v_2 I_3 & \delta_{i1} v_3 I_3 \\ 0_3 & \delta_{i2} v_1 I_3 & \delta_{i2} v_2 I_3 & \delta_{i2} v_3 I_3 \\ 0_3 & \delta_{i3} v_1 I_3 & \delta_{i3} v_2 I_3 & \delta_{i3} v_3 I_3 \end{pmatrix} \quad \mathbf{S} = -\frac{1}{\theta_1} \begin{pmatrix} \mathbf{0_5} \\ \frac{\partial \mathbf{E}}{\partial \mathbf{A_1}} \\ \frac{\partial \mathbf{E}}{\partial \mathbf{A_2}} \\ \frac{\partial \mathbf{E}}{\partial \mathbf{A_3}} \end{pmatrix}$$

where θ is a (potentially very small) function of A, and now:

$$\Sigma = pI + \rho A^T \frac{\partial E}{\partial A}$$

CHAPTER 5

Example Code

See more examples here.

5.1 Reactive Euler (1D, hyperbolic, S0)

We must define our fluxes and source vector:

```
from numba import njit
from numpy import zeros

# material constants
Qc = 1
cv = 2.5
Ti = 0.25
K0 = 250
gam = 1.4

@njit
def internal_energy(E, v, lam):
    return E - (v[0]**2 + v[1]**2 + v[2]**2) / 2 - Qc * (lam - 1)

def F(Q, d):

    r = Q[0]
    E = Q[1] / r
    v = Q[2:5] / r
    lam = Q[5] / r

    e = internal_energy(E, v, lam)

    # pressure
    p = (gam - 1) * r * e
```

(continues on next page)

(continued from previous page)

```
    F_ = v[d] * Q
    F_[1] += p * v[d]
    F_[2 + d] += p

    return F_

@njit
def reaction_rate(E, v, lam):

    e = internal_energy(E, v, lam)
    T = e / cv

    return K0 if T > Ti else 0

def S(Q):

    S_ = zeros(6)

    r = Q[0]
    E = Q[1] / r
    v = Q[2:5] / r
    lam = Q[5] / r

    S_[5] = -r * lam * reaction_rate(E, v, lam)

    return S_
```

Under the Reactive Euler model, $\mathbf{F}_i$ has no $\nabla\mathbf{Q}$ dependence, thus `F` here has call signature `(Q, d)`. Note that any functions called by `F` or `S` must be decorated with `@nit`, as must any functions that they subsequently call.

The numba library is used to compile `F` and `S` before solving the system. numba is able to compile some numpy functions. As a general rule though, you should aim to write your functions in pure Python, with no classes. This is guaranteed to compile. It will not produce the performance hit usually associated with Python loops and other features.

We now set out the initial conditions for the 1D detonation wave test. We use 400 cells, with a domain length of 1. The test is run to a final time of 0.5.

```
from numpy import inner, array

def energy(r, p, v, lam):
    return p / ((gam - 1) * r) + inner(v, v) / 2 + Qc * (lam - 1)

nx = 400
L = [1.]
tf = 0.5

rL = 1.4
pL = 1
vL = [0, 0, 0]
lamL = 0
EL = energy(rL, pL, vL, lamL)

rR = 0.887565
pR = 0.191709
vR = [-0.57735, 0, 0]
lamR = 1
ER = energy(rR, pR, vR, lamR)
```

(continues on next page)

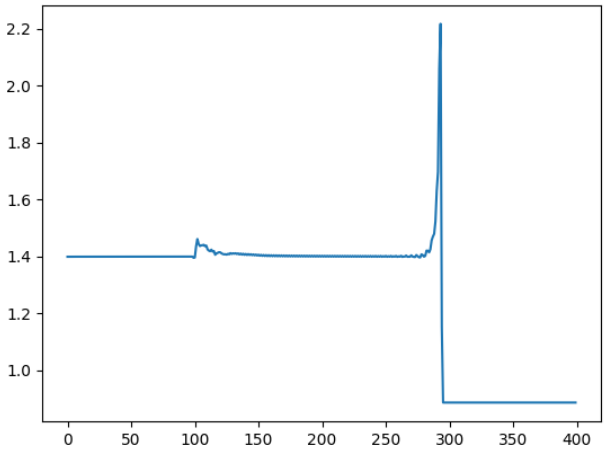
(continued from previous page)

```
QL = rL * array([1, EL] + vL + [lamL])
QR = rR * array([1, ER] + vR + [lamR])

Q0 = zeros([nx, 6])
for i in range(nx):
    if i / nx < 0.25:
        Q0[i] = QL
    else:
        Q0[i] = QR
```

We now solve the system. `pde_solver` returns an array `out` of shape $100 \times nx \times 6$. `out[j]` corresponds to the domain at $(j+1)\,\%$ through the simulation. We plot the final state of the domain for variable 0 (density):

```
import matplotlib.pyplot as plt

from pypde import pde_solver

out = pde_solver(Q0, tf, L, F=F, S=S, stiff=False, flux='roe', order=3)

plt.plot(out[-1, :, 0])
plt.show()
```

The plot is found below, in accordance with accepted numerical results:



5.2 Navier-Stokes (2D, parabolic)

We must define our fluxes and source vector:

```
from numba import njit
from numpy import dot, eye, zeros

# material constants
gam = 1.4
mu = 1e-2
```

(continues on next page)

(continued from previous page)

```
@njit
def sigma(dv):
    return mu * (dv + dv.T - 2 / 3 * (dv[0, 0] + dv[1, 1] + dv[2, 2]) * eye(3))


@njit
def pressure(r, E, v):
    return r * (gam - 1) * (E - dot(v, v) / 2)


def F(Q, DQ, d):

    F_ = zeros(5)

    r = Q[0]
    E = Q[1] / r
    v = Q[2:5] / r

    dr_dx = DQ[0, 0]
    drv_dx = DQ[0, 2:5]
    dv_dx = (drv_dx - dr_dx * v) / r

    dv = zeros((3, 3))
    dv[0] = dv_dx

    p = pressure(r, E, v)
    sig = sigma(dv)

    vd = v[d]
    rvd = r * vd

    F_[0] = rvd
    F_[1] = rvd * E + p * vd
    F_[2:5] = rvd * v
    F_[2 + d] += p

    sigd = sig[d]
    F_[1] -= dot(sigd, v)
    F_[2:5] -= sigd

    return F_
```

Under the Navier-Stokes model, $\mathbf{F}_i$ has a $\nabla\mathbf{Q}$ dependence, thus `F` here has call signature `(Q, DQ, d)`. Note that any functions called by `F` must be decorated with `@nit`, as must any functions that they subsequently call.

The numba library is used to compile `F` before solving the system. numba is able to compile some numpy functions. As a general rule though, you should aim to write your functions in pure Python, with no classes. This is guaranteed to compile. It will not produce the performance hit usually associated with Python loops and other features.

We now set out the initial conditions for the 2D Taylor-Green vortex test. We use 50x50 cells, with a domain length of 2π. The test is run to a final time of 1.

```
from numpy import cos, pi, sin

def total_energy(r, p, v):
```

(continues on next page)

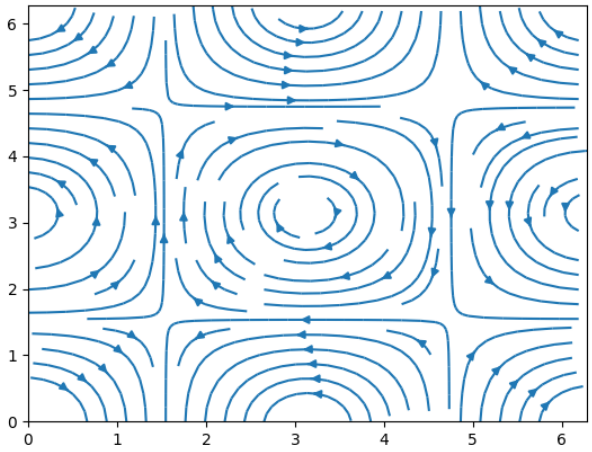
(continued from previous page)

```
    return p / (r * (gam - 1)) + dot(v, v) / 2


def make_Q(r, p, v):
    """ Returns the vector of conserved variables, given the primitive variables
    """
    Q = zeros(5)
    Q[0] = r
    Q[1] = r * total_energy(r, p, v)
    Q[2:5] = r * v
    return Q


L = [2 * pi, 2 * pi]

nx = 50
ny = 50
tf = 1

C = 100 / gam
r = 1
v = zeros(3)

u = zeros([nx, ny, 5])
for i in range(nx):
    for j in range(ny):
        x = (i + 0.5) * L[0] / nx
        y = (j + 0.5) * L[1] / ny
        v[0] = sin(x) * cos(y)
        v[1] = -cos(x) * sin(y)
        p = C + (cos(2 * x) + cos(2 * y)) / 4
        u[i, j] = make_Q(r, p, v)
```

We now solve the system. `pde_solver` returns an array `out` of shape $100 \times nx \times ny \times 5$. `out[j]` corresponds to the domain at $(j+1)\,\%$ through the simulation. We plot the final state of the domain for velocity:

```
import matplotlib.pyplot as plt
from numpy import linspace

from pypde import pde_solver

out = pde_solver(u,
                 tf,
                 L,
                 F=F,
                 cfl=0.9,
                 order=2,
                 boundaryTypes='periodic')

x = linspace(0, L[0], nx)
y = linspace(0, L[1], ny)

ut = out[-1, :, :, 2] / out[-1, :, :, 0]
vt = out[-1, :, :, 3] / out[-1, :, :, 0]
plt.streamplot(x, y, ut, vt)
plt.show()
```

The plot is found below, in accordance with accepted numerical results:



Index

P