1 PyDG Manual

PyDG is a high-order discontinuous Galerkin code written in python. The purpose of this document will be to familiarize the reader with PyDG. We will first discuss how to run the code, and then provide details into the inner workings of the code.

1.1 Running PyDG

To run PyDG, we need to construct an input file. This input file specifies things like the grid, order of accuracy, equation sets, etc. A sample inputfile for the isentropic vortex is given:

```
import numpy as np
import sys
sys.path.append("../../src_spacetime") #link the source directory for PyDG
from ic_functions_premade import vortexICS  #import the IC function
def savehook(main): # function called every save_freq iterations
  pass
## Make square grid
L = 10.                          #|  length
Nel = np.array([32,32,1,1])      #|  elements in x,y,z,t
order =np.array([2,2,1,1])       #|  spatial order
quadpoints = order*1.            #|  number of quadrature points
mu = 0.                          #|  viscocity
x = np.linspace(0,L,Nel[0]+1) #|  x, y, and z
y = np.linspace(0,L,Nel[1]+1) #|
z = np.linspace(0,L,Nel[2]+1) #|
x,y,z = np.meshgrid(x,y,z,indexing='ij')
t = 0                            #|  simulation start time
dt = 0.0125                      #|  time step
et = 10.                         #|  simulation end time
save_freq = 10.                 #|  frequency to save output and print to screen
eqn_str = 'Navier-Stokes'    #|  equation set
schemes = ('roe','Inviscid')    #|  inviscid and viscous flux schemes
procx = 2                           #|  processor decomposition in x
procy = 2      #|  same in y. Note procx*procy needs to equal total number of
    procs

## Boundary conditions
right_bc = 'periodic'
left_bc = 'periodic'
top_bc = 'periodic'
bottom_bc = 'periodic'
front_bc = 'periodic'
back_bc = 'periodic'

## Arguments for boundary conditions
right_bc_args = []
left_bc_args = []
top_bc_args = []
bottom_bc_args = []
front_bc_args = []
back_bc_args = []
```

```
41
42 # creation of BCs array for the solver
43 BCs = [right_bc,right_bc_args,top_bc,top_bc_args,left_bc,left_bc_args,
       bottom_bc,bottom_bc_args,front_bc,front_bc_args,back_bc,back_bc_args]
44
45 # Misc strings that currently are required (from combustion)
46 source_mag= False
47 mol_str = False          #|
48
49 # Time stepping. The linear and nonlinear solvers are only used if implicit
50 time_integration = 'SSP_RK3'
51 linear_solver_str = 'GMRes'
52 nonlinear_solver_str = 'Newton'
53
54 #== Assign initial condition function. Note that you can alternatively define
       this here
55 #== layout is my_ic_function(x,y,z), where x,y,z are the quad points
56 IC_function = vortexICS                #|
57                                        #|
58 execfile('../../src_spacetime/PyDG.py')      #|  call the solver
```

To PyDG with the above script, move to the directory of "inputfile.py" and type into the command line:

```
1   $ mpirun -np 4 python inputfile.py
```

1.1.1 Post Processing

Running PyDG will lead to the creation of a folder called "Solution" in the directory where the inputfile is. In this "Solution" directory, PyDG will put saved states of the solution - for example "npsol0.npz". These states are saved every "$save_freq$" iterations; this is defined in the input deck. The solution files are saved in ".npz" format and may be visualized in python. A post processing script that converts the solution files to ".vts" files for visualization in paraview is also available. To run this script, one should move to the "Solution" directory and run the postProcessing.py script with the appropriate equation set that is present in the src_spacetime folder:

```
1   $ cd Solution
2   $ python (PyDG home directory)/src_spacetime/postProcessing.py (Eqn set)
```

Currently implemented equation sets are:

1. Navier-Stokes
   - This runs the script for the classic Navier-Stokes (or Euler) equations. It outputs fields of $\rho, \rho u_i$, and ρE.
2. Entropy
   - This runs the script for the Navier-Stokes equations with entropy variables. It outputs fields of $\rho, \rho u_i$, and ρE.
3. Scalar

- This runs the script for a scalar equation. It outputs the field with the name u.

1.2 The Code

Here we dive into details about how the code works.

1.2.1 Spatial Discretization: The Discontinuous Galerkin Approach

PyDG utilizes Legendre polynomials for its spatial discretization. The Legendre polynomials consist of a hierarchical set of polynomials and are orthogonal on the unit cube. For multidimensional problems, PyDG uses tensor products. The remainder of this section will outline the discontinuous Galerkin approach;

We consider the system of conservation laws,

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \mathbf{F} = 0 \tag{1}$$

with $\mathbf{x} \in \Omega$. In the discontinuous Galerkin approach, the domain Ω is subdivided into individual elements, Ω_k. The state variable $\mathbf{u}$ is then approximated by basis functions over Ω,

$$\mathbf{u}(\mathbf{x}, t) = \sum_{k=0}^{N-1} \sum_{j=0}^{p} \mathbf{a}_{j,k}(t) \mathbf{w}_{k,j}(\mathbf{x}),$$

where $\mathbf{a}$ are the basis coefficients and $\mathbf{w}$ are the basis functions. Note that in the DG approach, the basis functions have local support and are hence discontinuous between the elements.

The DG approach proceeds by multiplying Eq. 1 by the basis functions and integrating over the support of each basis function,

$$\int_{\Omega_k} \mathbf{w}_{k,j} \left[\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \mathbf{F} \right] d\Omega = 0. \tag{2}$$

The flux term is then integrated by parts. This integration by parts couples all the elements and establishes the finite-volume aspect of the DG approach. One has [**?**],

$$\int_{\Omega_k} \mathbf{w}_{k,j} \frac{\partial \mathbf{u}}{\partial t} d\Omega - \int_{\Omega_k} \nabla \mathbf{w}_{k,j} \cdot \mathbf{F} d\Omega + \int_{d\Omega_k} \mathbf{w}^{+}_{k,j} \hat{\mathbf{F}}(\mathbf{u}^{+}, \mathbf{u}^{-}, \hat{n}) dl = 0, \tag{3}$$

where $+$ and $-$ refers to the element interior and exterior, respectively. The term $\hat{\mathbf{F}}$ is the numerical flux between the faces.

The numerical implementation of the DG method has three parts:

1. Evaluation of the volumetric integrals (second term in Eq. 3.)
2. Evaluation of the fluxes (third term in Eq. 3.)
3. Coupling to a time marching scheme.

1.3 Code Layout

The general structure of the code is shown in Figure 1. The code is driven by an input-file, callled "inputfile.py". In this file, aspects such as the grid, order of accuracy, equation set, boundary conditions, etc. are defined. This file proceeds to call the driving script, "PyDG.py". The script "PyDG.py" first initializes variables and equation sets and then runs the main time loop. Inside the time loop, the script "timeSchemes.py" is called. The "timeSchemes.py" script contains the time scheme routines. In each stage of a time scheme routine, the script "turb_model.py" is called. This "turb_models.py" script contains manipulations that are performed in turbulence models. Note that the code mainly focuses on residual-based-type closure models and as such the "turb_models.py" script exists outside of the traditional RHS evaluation files. The "turb_models.py" script will call the script "DG_functions.py". This script is responsible for calculating the flux and volume terms in a DG formulation.

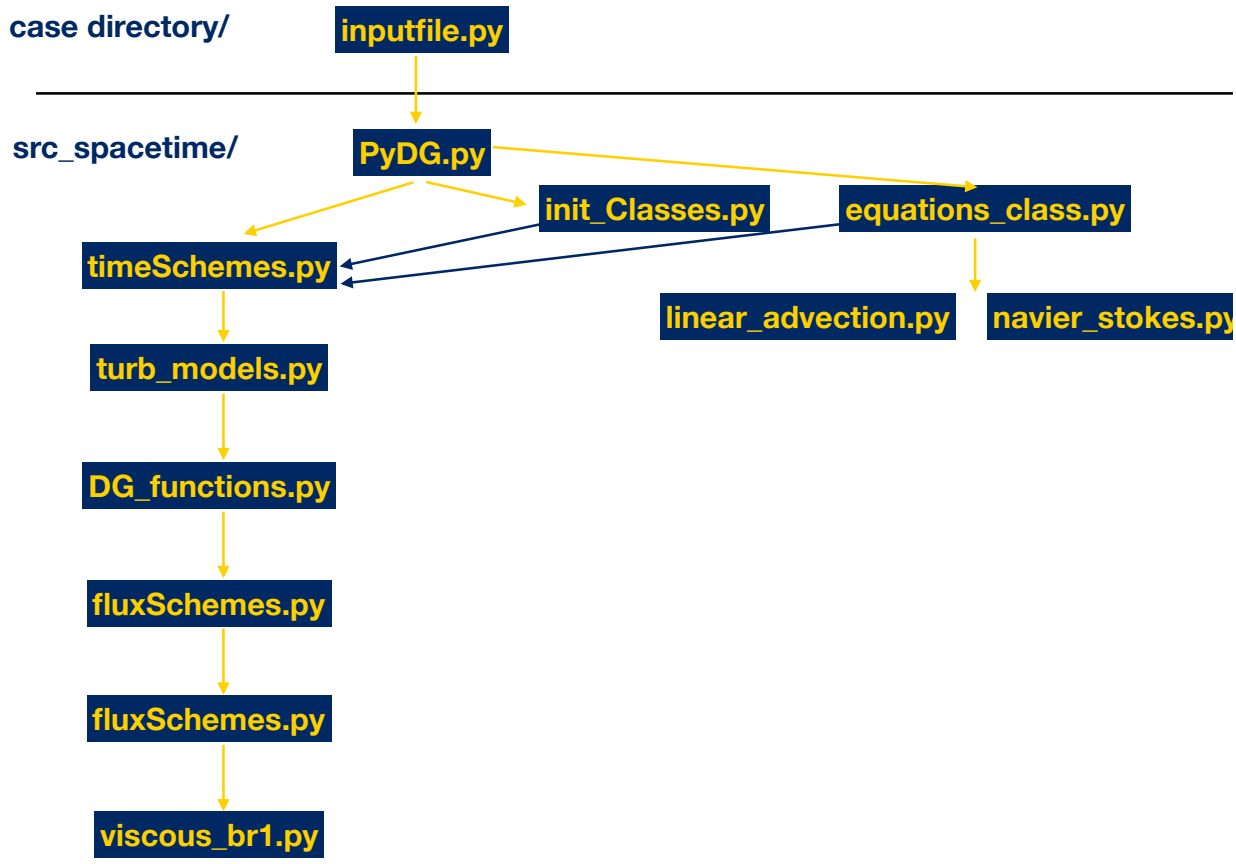

Figure 1: Layout for PyDG. Yellow arrows are function calls, blue arrows are passed classes

1.4 General Dimension of Variables

The variables in PyDG primarily consist of variables in modal space and variables in physical space. The variables in modal space, which are typically referenced by a command similar to "classname.a", are of dimension,

$$classname.a = N_{vars} \times N_{px} \times N_{py} \times N_{pz} \times N_{pt} \times N_{elx} \times N_{ely} \times N_{elz} \times N_{elt}.$$

Yes, they are $8D$ arrays! The variable N_{vars} is the number of unknown variables (eg. 5 for compressible Euler). The term N_{px} is the order of the polynomial in the x direction, with N_{py}, N_{pz}, and N_{pt} being the same for the y, z, and temporal direction, respectively. Similarly, the N_{el} terms refer to the number of elements in each direction.

Variables in physical space have a similar structure. Typically referenced by a command similar to "classname.u", they are of dimension

$$classname.u = N_{vars} \times N_{qx} \times N_{qy} \times N_{qz} \times N_{qt} \times N_{elx} \times N_{ely} \times N_{elz} \times N_{elt}.$$

Similar to N_{px}, terms such as N_{qx} refer to the number of quadrature points in each direction. Typically for the Euler equations, one uses twice the quadrature points as the order of the polynomial.

1.5 The variables class

The variables class, typically assigned "main", is the primary class used in the solver. The variables class is defined in "init_Classes.py". Variables contains several important subvariables and subclasses

- variables.a: This is a subclass that represents the main state variables.
  - a.a: These are the Legendre coefficients associated with the state variable in a. The dimensions are

    $$a.a = N_{vars} \times N_{px} \times N_{py} \times N_{pz} \times N_{pt} \times N_{elx} \times N_{ely} \times N_{elz} \times N_{elt}$$

  - a.u: These are the physical space values associated with the state variable in a. The dimensions are

    $$a.u = N_{vars} \times N_{qx} \times N_{qy} \times N_{qz} \times N_{qt} \times N_{elx} \times N_{ely} \times N_{elz} \times N_{elt}$$