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. It is assumed that the reader has a basic familiarity with the DG method.

1.1 Spatial Discretization

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.

1.2 Code Layout

The general structure of the code is shown in Figure 1. The code is driven by an inputfile, 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.

1.3 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.

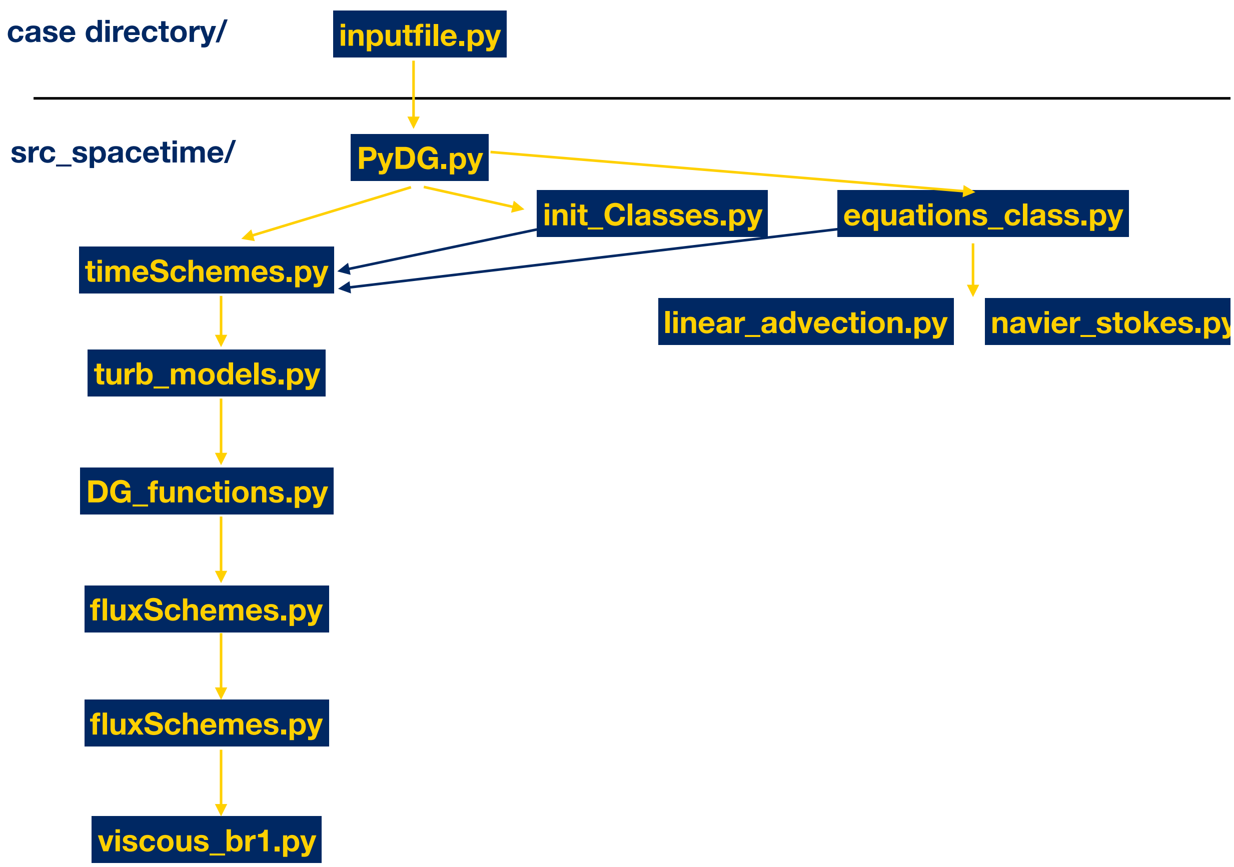

Figure 1: Layout for PyDG. Yellow arrows are function calls, blue arrows are passed classes

1.4 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

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

2 The input deck

A sample inputfile for the isentropic vortex is given. The input deck is in the format of a python file and requires specification of:

- The N_{el} array. This must be a 1D vector of length 4. Entries are elements in x, y, z, t
- The *order* array. Arguments are order of accuracy in each direciton
- The *quadpoints* array. Arguments are the number of quad points in each direction
- The viscosity, μ. This must be specified even if inviscid.
- The grid, x,y, and z. These must be passed in the format of a 3D array.

```
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 = []

# creation of BCs array for the solver
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]

# Misc strings that currently are required (from combustion)
source_mag= False
mol_str = False           #|

# Time stepping. The linear and nonlinear solvers are only used if implicit
time_integration = 'SSP_RK3'
linear_solver_str = 'GMRes'
nonlinear_solver_str = 'Newton'

#== Assign initial condition function. Note that you can alternatively define this here
#== layout is my_ic_function(x,y,z), where x,y,z are the quad points
IC_function = vortexICS                 #|
                                        #|
execfile('../../src_spacetime/PyDG.py')       #|  call the solver
```