```
from opensbli.utilities.numerical_functions import spline, splint
from sympy import Piecewise
from sympy import S, N, Rational, pprint, Eq, exp, tanh
from scipy.integrate import odeint
import numpy as np
import numpy.polynomial.polynomial as poly
import matplotlib.pyplot as plt
from opensbli.initialisation import GridBasedInitialisation
from opensbli.core.opensbliobjects import DataObject, ConstantObject
from opensbli.core.grid import GridVariable
from opensbli.core.kernel import Kernel
import warnings
# from scipy.optimize import curve_fit
from opensbli.equation_types.opensbliequations import OpenSBLIEq


plt.style.use('classic')


class Boundary_layer_profile(object):
""" Performs a similarity solution (Viscous fluid flow, F.White 1974),
to obtain u and T profiles for a laminar compressible boundary-layer.

:arg float xmach: Free-stream Mach numnber.
:arg float Pr: Prandtl number.
:arg float gama: Ratio of specific heats.
:arg float Tw: Wall temperature, use -1 for adibatic wall conditions.
:arg float Re: Free-stream Reynolds number.
:arg float Tinf: Dimensional free-stream temperature for Sutherland's law."""

def __init__(self, xmach, Pr, gama, Tw, Re, Tinf):
self.y, self.u, self.T, self.scale = self.generate_boundary_layer_profile(xmach, Pr, gama, Tw, Tinf, Re)
self.Re = Re
self.n = np.size(self.y)
return

def compbl(self, v, p=None):
""" Sets up the system of equations to be integrated by odeint.

:arg ndarray: v: Solution vector.
:arg None: p: Empty dummy argument required by the odeint function.
:returns: list: dv: System of equations."""
suth = self.suth
c = np.sqrt(v[3])*(1.0+suth)/(v[3]+suth)
dcdg = 1.0/(2.0*np.sqrt(v[3])) - np.sqrt(v[3])/(v[3]+suth)
dcdg *= (1.0+suth) / (v[3]+suth)
cp = dcdg*v[4]
dv = [v[1], v[2], -v[2]*(cp+v[0])/c, v[4],
-v[4]*(cp+self.pr*v[0])/c - self.pr*(self.gama-1)*self.xmach**2 * v[2]**2]
return dv

def generate_boundary_layer_profile(self, xmach, pr, gama, Tw, Tinf, Re):
""" Generates a boundary layer initial profile. Solves the mean flow
in a compressible boundary layer. (Equations 7.32) in White (1974).

:arg float xmach: Mach number.
:arg float pr: Prandtl number.
:arg float gama: Ratio of specific heats.
:arg float Tw: Wall temperature Tw/Tinf (< 0 for adiabatic)
:arg float Tinf: Freestream reference temperature (Kelvin)
:arg float Re: Reynolds number."""

n_iter, jmax = 5, 1001
v, dv, f, f1, f2 = np.zeros(2), np.zeros(2), np.zeros(2), np.zeros(2), np.zeros(2)
self.pr, self.gama, self.xmach, self.Re, self.Tw = pr, gama, xmach, Re, Tw
self.nvisc = 1 # 1 Sutherlands law, 2 Power law, 3 Chapman-Rubesin approximation.
etamax = 10.0
```

```
nstep = jmax-1
errtol = 1e-10
self.suth = 110.40/Tinf # Sutherland constant
self.soln = np.zeros((5, jmax))
self.eta = np.linspace(0, etamax, jmax)
vstart = np.zeros(5)
if Tw > 0:
vstart[3] = Tw
elif Tw < 0: # Adiabatic wall, initial guess
v[1] = 1.0 + 0.5*0.84*(gama-1)*xmach**2
v[0] = 0.47*v[1]**0.21
else: # Fixed wall temperature, initial guess. Tested for 1<Tw<4 and M<8 (Pr=0.72, gamma=1.4)
v[1] = 4.5 # !0.062*xmach**2-0.1*(Tw-1.0)*(1.0+xmach)/(0.2+xmach)
v[0] = 0.494891 # !0.45-0.01*xmach+(Tw-1.)*0.06
# Initial increments
dv[0] = v[0]*0.01
if Tw < 0:
dv[1] = v[1]*0.01
else:
dv[1] = 0.1
# Main loop
# ODE solver parameters
abserr = 1.0e-8
relerr = 1.0e-6
for k in range(n_iter):
vstart[2] = v[0] # Initial value
if Tw < 0:
vstart[3] = v[1]
else:
vstart[4] = 1.141575 # v[1]
# Call the ODE solver.
self.soln = odeint(self.compbl, vstart, self.eta,
atol=abserr, rtol=relerr).T
err = np.abs(self.soln[1, nstep]-1.0) + np.abs(self.soln[3, nstep]-1.0)
if err < errtol:
break
f[0] = self.soln[1, nstep] - 1.0
f[1] = self.soln[3, nstep] - 1.0
# Increment v[0]
vstart[2] = v[0] + dv[0]
if (Tw < 0):
vstart[3] = v[1]
else:
vstart[4] = v[1]
self.soln = odeint(self.compbl, vstart, self.eta,
atol=abserr, rtol=relerr).T
f1[0] = self.soln[1, nstep] - 1.0
f1[1] = self.soln[3, nstep] - 1.0
# Increment v[1]
vstart[2] = v[0]
if (Tw < 0):
vstart[3] = v[1] + dv[1]
else:
vstart[4] = v[1] + dv[1]
self.soln = odeint(self.compbl, vstart, self.eta,
atol=abserr, rtol=relerr).T
f2[0] = self.soln[1, nstep] - 1.0
f2[1] = self.soln[3, nstep] - 1.0
# Solve the linear system
al11 = (f1[0] - f[0])/dv[0]
al21 = (f1[1] - f[1])/dv[0]
al12 = (f2[0] - f[0])/dv[1]
al22 = (f2[1] - f[1])/dv[1]
det = float(al11*al22 - al21*al12)
# New dv for improved solution
dv[0] = (-al22*f[0] + al12*f[1])/det
```

```
dv[1] = (al21*f[0] - al11*f[1])/det
v[0] += dv[0]
v[1] += dv[1]
# Write out improved estimate
# print 'it ', k, dv[0], dv[1], v[0], v[1]
# print "final vaues ", v[0], v[1], err
# Save the wall temperature value
self.Twall = v[1]
print("The wall temperature is :", self.Twall)
y, u, T, scale_factor = self.integrate_boundary_layer(nstep)
self.scale_factor = scale_factor
print("The scale factor is :", self.scale_factor)
print("Wall normal derivative of velocity at the wall is :", self.dudy)
return y, u, T, scale_factor

def integrate_boundary_layer(self, n):
""" Integrates the boundary-layer and calculates the scale factor from displacement thickness.

:arg int n: Iteration number from the iterative solver.
:returns: ndarray: y: Wall normal coordinates.
:returns: ndarray: u: Streamwise velocity component profile.
:returns: ndarray: T: Temperature profile.
:returns: float: scale: Scale factor of the boundary-layer."""
sumd, record_z = 0, 0
z = np.zeros(n+1)
d_eta = self.eta[1]*0.5
print('This is eta1')
# print(self.eta[1])
# self.soln[1,:] is the u velocity, should be 1 in free stream
for i in range(1, n+1):
z[i] = z[i-1] + d_eta*(self.soln[3, i] + self.soln[3, i-1])
dm = self.soln[3, i-1] - self.soln[1, i-1]
dd = self.soln[3, i] - self.soln[1, i]
sumd += d_eta*(dd+dm)
if(self.soln[1, i] > 0.999 and record_z < 1.0):
# print "recording at iteration: ", i
# dlta = z[i]
record_z = 2.0
scale = sumd
# print("delta is :", dlta)
print("conversion factor is: ", scale)
# print("scaled delta is: ", dlta/scale)
# Rescale with displacement thickness and convert to FLOWER variable normalisation
y, u, T = z/scale, self.soln[1, :], self.soln[3, :]
# Calculate du/dy at the wall
dy = y[1]
# self.dudy = (-3*u[0]+4*u[1]-u[2])/(2.0*dy)
self.dudy = (-1.83333333333334*u[0]+3.00000000000002*u[1]-1.50000000000003*u[2]+0.333333333333356*u[3]-
8.34657956545823e-15*u[4]+1.06910315192207e-15*u[5])/dy
self.dTdy = (-1.83333333333334*T[0]+3.00000000000002*T[1]-1.50000000000003*T[2]+0.333333333333356*T[3]-
8.34657956545823e-15*T[4]+1.06910315192207e-15*T[5])/dy
return y, u, T, scale


class Initialise_DimFlat(GridBasedInitialisation):
""" Generates the initialiastion equations for the boundary-layer profile.

:arg list npoints: Numerical values of the number of points in each direction.
:arg list lengths: Numerical values of the problem dimensions.
:arg list directions: Integer values of the problem directions.
:arg list betas: Stretching factors for stretched grids.
:arg int n_coeffs: Desired number of coefficients for the polynomial fit.
:arg float Re: Reynolds number.
:arg float xMach: Free-stream Mach number"""
def __new__(cls, bl_directions, n_coeffs, Re, xMach, Tinf, rhoref, uref, ydomain, blthickness, coordinate_evaluations=None):
ret = super(Initialise_DimFlat, cls).__new__(cls)
print("Polynomial boundary-layer initialiastion called with Re = %f, Mach = %f, T_inf = %f." % (Re, xMach, Tinf))
```

```
ret.coordinates = [x[1] for x in bl_directions]
ret.bl_directions = bl_directions
ret.n_coeffs = n_coeffs
ret.coordinate_evaluations = coordinate_evaluations
ret.Re = ret.find_constant_values([Re])[0]
ret.Tinf = ret.find_constant_values([Tinf])[0]
ret.equations = []
ret.xMach = ret.find_constant_values([xMach])[0]

# gnsa1e21 edit -------------------------------------------
# function: get inputs from the main python script
ret.rhoref = ret.find_constant_values([rhoref])[0]
ret.uref = ret.find_constant_values([uref])[0]
ret.ydomain = ret.find_constant_values([ydomain])[0]
ret.blthickness = ret.find_constant_values([blthickness])[0]
# gnsa1e21 edit -------------------------------------------

return ret

def find_constant_values(self, input):
outlist = []
for l in input:
if isinstance(l, ConstantObject):
if isinstance(l.value, str):
raise ValueError("")
else:
outlist += [l.value]
else:
outlist += [l]
return outlist

def check_inputs(self, block):
bl_directions = [x[0] for x in self.bl_directions]
if sum(bl_directions) < 1:
raise ValueError("Provide the directions to apply a boundary layer profile in.")
if len(bl_directions) != block.ndim:
raise ValueError("The list of polynomial directions must match the dimensions of the problem.")
if self.n_coeffs < 10:
raise ValueError("Higher number of polynomial coefficients are required for a good polynomial fit.")
return

def spatial_discretisation(self, block):
self.equations = []
self.block = block
self.idxs = block.grid_indexes
self.check_inputs(block)

# Check if user has passed equations to evaluate coordinates, and add them to the kernel
if self.coordinate_evaluations:
self.equations += self.coordinate_evaluations
self.initial = self.generate_initial_condition()
# Add polynomial equations to initialise the solution
self.equations += self.eqns

self.equations = block.dataobjects_to_datasets_on_block(self.equations)
# Ensure coordinate arrays are also restarted when required
self.check_coordinate_evaluation(block)
print(self.order)
kernel1 = Kernel(block, computation_name="Grid_based_initialisation%d" % self.order)
kernel1.set_grid_range(block)
schemes = block.discretisation_schemes
for d in range(block.ndim):
for sc in schemes:
if schemes[sc].schemetype == "Spatial":
kernel1.set_halo_range(d, 0, schemes[sc].halotype)
kernel1.set_halo_range(d, 1, schemes[sc].halotype)
kernel1.add_equation(self.equations)
```

```
kernel1.update_block_datasets(block)
self.Kernels = [kernel1]
return

def generate_initial_condition(self):
n_coeffs = self.n_coeffs
# Load from similarity solution class
y, u, T, rho, n = self.load_similarity()
y0, u0, T0, rho0, n0 = self.load_similarity()
# Solve continuity equation to obtain v
rho = (10000.0/(8.3143))*(1.0/T)
v0, dvdy = self.solved_continuity(y, u, rho)
v0 = v0 / rho
print('velocity')
print('\n----------------------------------------------------------------------')
print(y)
print(rho)
print(u)
print(v0)
print('----------------------------------------------------------------------\n')

y, u, v, T, rho, n = y*self.blthickness , u*self.uref, v0*self.uref, T*self.Tinf, self.rhoref, n # dimensionalise all variables - gnsa1e21,
2022
self.dudy, dvdy = self.dudy*self.uref / (self.blthickness ), dvdy*self.uref / (self.blthickness )
# Tolerance for finding the edge of the boundary layer.
tolerance = 1e-10
bl_directions = [x[0] for x in self.bl_directions]
print(bl_directions)

if sum(bl_directions) == 1: # 2D MixFlat and 3D spanwise periodic MixFlat, boundary layer in one direction
# Create a large array of coordinates for this direction to interpolate the profile onto
dire = [i for i, x in enumerate(self.bl_directions) if x[0]][0]
poly_coordinates0 = self.uniform_1d_coordinate()
poly_coordinates = self.uniform1d_coordinate()

# Interpolate u, T, rho onto the grid
u_nondim = self.interpolate_onto_grid(y0, poly_coordinates0, u0, self.dudy, 0)
u_new = self.interpolate_onto_grid(y, poly_coordinates, u, self.dudy, 0)
T_new = self.interpolate_onto_grid(y, poly_coordinates, T, 0, 0)
v_new = self.interpolate_onto_grid(y, poly_coordinates, v, dvdy, 0)


rho_new = (10000.0/(8.3143))*(1.0/T_new) # create array of varying rhoref values - gnsa1e21, 2023
rhou_new = rho_new*u_new
rhov_new = rho_new*v_new
edge = self.find_edge_of_bl(u_new, tolerance)

# Obtain polynomial fit coefficients
rhou_coeffs = self.fit_polynomial(poly_coordinates, rhou_new, edge, n_coeffs)
rhov_coeffs = self.fit_polynomial(poly_coordinates, rhov_new, edge, n_coeffs)
T_coeffs = self.fit_polynomial(poly_coordinates, T_new, edge, n_coeffs)
self.generate_one_wall_equations([rhou_new, rhov_new, T_new], [rhou_coeffs, rhov_coeffs, T_coeffs], dire, edge,
poly_coordinates)

elif sum(bl_directions) == 2: # 3D with one side wall in x2 # WARNING: This currently does the same thing twice,
edges, coeffs, profiles, normal_coeffs, normal_profiles = [], [], [], [], []
directions = [i for i, x in enumerate(self.bl_directions) if x[0]]
for dire in directions: # For different bl thicknesses on different walls this still needs to be a loop
poly_coordinates = self.uniform_1d_coordinate()
# Interpolate u, and T onto the grid
u_new = self.interpolate_onto_grid(y, poly_coordinates, u, self.dudy, 0)
T_new = self.interpolate_onto_grid(y, poly_coordinates, T, 0, 0)
# Temperature scaling function in region [0, 1]
Tw, Tinf = self.Twall, 1.0 # Free-stream normalised temperature of 1.0, Wall temp taken from similarity solution
g = self.temperature_scaling(T_new, Tw, Tinf)
# rho_new = 1.0/T_new
rho_new = (10000.0/(8.3143))*(1.0/T_new)
```

```
rhou_new = rho_new*u_new
# Solve continuity equation to obtain rho*wall_normal_velocity
rho_vel_normal = self.solve_continuity(poly_coordinates, u_new, rho_new)
profiles.append([rhou_new, T_new])
normal_profiles.append(rho_vel_normal)
edge = self.find_edge_of_bl(u_new, tolerance)
edges.append(edge)
# Obtain polynomial fit coefficients
rhou_coeffs = self.fit_polynomial(poly_coordinates, rhou_new, edge, n_coeffs)
g_coeffs = self.fit_polynomial(poly_coordinates, g, edge, n_coeffs)
rho_vel_normal_coeffs = self.fit_polynomial(poly_coordinates, rho_vel_normal, edge, n_coeffs)
coeffs.append([rhou_coeffs, g_coeffs])
normal_coeffs.append(rho_vel_normal_coeffs)
self.generate_two_wall_equations(profiles, coeffs, directions, edges, normal_profiles, normal_coeffs, poly_coordinates)
else:
raise NotImplementedError("Boundary layer initialisation is not implemented for walls in 3 dimensions.")
return

def uniform_1d_coordinate(self):
n_elem = 10000
return np.linspace(0, 20.0, n_elem)

def uniform1d_coordinate(self):
n_elem = 10000
return np.linspace(0, 20.0e-4, n_elem)

def temperature_scaling(self, temp_profile, Tw, Tinf):
""" Computes the temperature profile between [0,1]

:arg ndarray temp_profile: Temperature profile values between wall temperature and freestream.
:arg float Tw: Wall temperature.
:arg float Tinf: Free-stream temperature.
:returns: ndarray: g: Temperature profile ranging from 0 to 1. """
g = (temp_profile - Tw)/(Tinf - Tw)
return g

def form_equation(self, variable, name, coefficients, direction, edge, poly_coordinates):
""" Creates the piecewise equations for the cases of 2D and 3D span-periodic boundary-layer profiles.

:arg ndarray variable: Array of values for a given flow variable, used to obtain the free-stream value.
:arg string name: Name of the variable.
:arg ndarray coefficients: Coefficients for the polynomial fit.
:arg int direction: Spatial direction to apply the equation to.
:arg int edge: Grid index for the edge of the boundary-layer.
returns: Eq: eqn: OpenSBLI equation to add to the initialisation kernel."""
bl_edge_coordinate = poly_coordinates[edge]
powers = [i for i in range(np.size(coefficients))][::-1]
eqn = sum([coeff*self.coordinates[direction]**power for (coeff, power) in zip(coefficients, powers)]) # TODO set to exactl 1.0 if
required
eqn = OpenSBLIEq(GridVariable('%s' % name), Piecewise((eqn, self.coordinates[direction] < bl_edge_coordinate), (variable[edge],
True)))
return eqn

def form_mixed_equation(self, profiles, names, coefficients, directions, edges, normal_profiles, normal_coeffs,
poly_coordinates):
""" Generates the equations for the 3D SBLI sidewall case.

:arg list profiles: Arrays of values for the rhou and [0,1] temperature profiles.
:arg list names: Variable names as strings.
:arg list coefficients: Coefficients for the polynomial fit for rhou and temperature profiles.
:arg list directions: Directions that contain a wall.
:arg list edges: Indices for the boundary layer edges in each direction.
:arg list normal_profiles: Arrays of values for the rhov and rhow profiles.
:arg list normal_coeffs: Coefficients for the polynomial fit for rhov and rhow.
:returns: list: piecewise_eqns: Piecewise initialisation equations to be added to the initialisation class."""

# Assuming we have the same number of poly coefficients in each direction, change later if required
powers = [i for i in range(np.size(coefficients[0][0]))][::-1]
```

```
# Loop over rhou, and T profiles
piecewise_eqns = []
direction1 = directions[0]
direction2 = directions[1]
# x variables and coordinates at the edge of the boundary layer in each
coord1, coord2 = self.coordinates[direction1], self.coordinates[direction2]
bl_coord1, bl_coord2 = poly_coordinates[edges[0]], poly_coordinates[edges[1]]
# Create rhou profiles
eqn1 = sum([coeff*coord1**power for (coeff, power) in zip(coefficients[0][0], powers)])
eqn2 = sum([coeff*self.local_coordinate**power for (coeff, power) in zip(coefficients[1][0], powers)]) # profiles[0][max(edges)]
u_var1, u_var2 = GridVariable('%s_1' % names[0]), GridVariable('%s_2' % names[0])
# freestream_value = np.max([profiles[0][0][edges[0]], profiles[1][0][edges[1]]])
freestream_value = 1.0
piecewise_eqns.append(OpenSBLIEq(u_var1, Piecewise((eqn1, coord1 < bl_coord1), (freestream_value, True))))
piecewise_eqns.append(OpenSBLIEq(u_var2, Piecewise((eqn2, self.local_coordinate < bl_coord2), (freestream_value, True))))
piecewise_eqns.append(OpenSBLIEq(GridVariable('%s' % names[0]), u_var1*u_var2))
# Create T profiles, g = (T-Tw)/(Tinf - Tw) ---> T = g*(Tinf-Tw) + Tw
Tw = ConstantObject('Twall')
eqn1 = sum([coeff*coord1**power for (coeff, power) in zip(coefficients[0][1], powers)])
eqn2 = sum([coeff*self.local_coordinate**power for (coeff, power) in zip(coefficients[1][1], powers)]) # profiles[0][max(edges)]
T_var1, T_var2 = GridVariable('%s_1' % names[1]), GridVariable('%s_2' % names[1])
freestream_value = np.max([profiles[0][1][edges[0]], profiles[1][1][edges[1]]])
freestream_value = 1.0 # CHECK THIS VALUE
piecewise_eqns.append(OpenSBLIEq(T_var1, Piecewise((eqn1, coord1 < bl_coord1), (freestream_value, True))))
piecewise_eqns.append(OpenSBLIEq(T_var2, Piecewise((eqn2, self.local_coordinate < bl_coord2), (freestream_value, True))))
piecewise_eqns.append(OpenSBLIEq(GridVariable('%s' % names[1]), T_var1*T_var2*(freestream_value - Tw) + Tw))
# Create normal velocity component profiles, rhov:
eqn1 = sum([coeff*coord1**power for (coeff, power) in zip(normal_coeffs[0], powers)])
temp1 = GridVariable('%s' % names[2])
rhov_inf = normal_profiles[0][edges[0]]*u_var2 # Multiplying by rhou from the other direction
piecewise_eqns.append(OpenSBLIEq(temp1, Piecewise((eqn1*u_var2, coord1 < bl_coord1), (rhov_inf, True))))
# rhow:
eqn2 = sum([coeff*self.local_coordinate**power for (coeff, power) in zip(normal_coeffs[1], powers)])
temp2 = GridVariable('%s' % names[3])
# rhow should reduce to zero at the symmetry plane
rhow_inf = normal_profiles[1][edges[1]]*(1 - (self.local_coordinate-bl_coord2)/(0.5*self.domain_length-bl_coord2))*u_var1
piecewise_eqns.append(OpenSBLIEq(temp2, self.side_fact*Piecewise((eqn2*u_var1, self.local_coordinate < bl_coord2), (rhow_inf,
True))))
return piecewise_eqns

def generate_one_wall_equations(self, data, coeffs, direction, edge, poly_coordinates):
""" Generates the equations for 2D SBLI and 3D span-periodic cases.
:arg list data: Profile arrays for rhou0, rhou1 and temperature.
:arg list coeffs: Coefficients for the polynomial fits.
:arg list direction: Direction normal to the wall.
:arg int edge: Grid index for the edge of the boundary-layer."""
self.eqns = []
rhou0_eqn = self.form_equation(data[0], 'rhou0', coeffs[0], direction, edge, poly_coordinates)
rhou1_eqn = self.form_equation(data[1], 'rhou1', coeffs[1], direction, edge, poly_coordinates)
T_eqn = self.form_equation(data[2], 'T', coeffs[2], direction, edge, poly_coordinates)
# Set conservative values
rho, rhou0, rhou1, T = GridVariable('rho'), GridVariable('rhou0'), GridVariable('rhou1'), GridVariable('T')
# rho_eqn = OpenSBLIEq(rho, 1.0/T)
rho_store = OpenSBLIEq(DataObject('rho'), rho)
rhou0_store = OpenSBLIEq(DataObject('rhou0'), rhou0)
rhou1_store = OpenSBLIEq(DataObject('rhou1'), rhou1)

# set chemical densities
# rhoO, rhoN, rhoO2, rhoN2, rhoNO = GridVariable('rhoO'), GridVariable('rhoN'), GridVariable('rhoO2'), GridVariable('rhoN2'),
GridVariable('rhoNO')
# # set chemical densities
# rhoO, rhoN, rhoO2, rhoN2, rhoNO = GridVariable('rhoO'), GridVariable('rhoN'), GridVariable('rhoO2'), GridVariable('rhoN2'),
GridVariable('rhoNO')
rhoev = GridVariable('rhoev')
evequilO2, evequilN2, evequilNO = GridVariable('evequilO2'), GridVariable('evequilN2'), GridVariable('evequilNO')

# set dimensional grid constants ----------
```

```
uref, rhoref, pref, Rhat, Twall = ConstantObject('uref'), ConstantObject('rhoref'), ConstantObject('pref'), ConstantObject('Rhat'),
ConstantObject('Twall')
thetavO, thetavN, thetavO2, thetavN2, thetavNO =
ConstantObject('thetavO'),ConstantObject('thetavN'),ConstantObject('thetavO2'),ConstantObject('thetavN2'),ConstantObject('the
tavNO')
MO, MN, MO2, MN2, MNO = ConstantObject('MO'), ConstantObject('MN'), ConstantObject('MO2'), ConstantObject('MN2'),
ConstantObject('MNO')
dhO, dhN, dhO2, dhN2, dhNO = ConstantObject('dhO'), ConstantObject('dhN'), ConstantObject('dhO2'), ConstantObject('dhN2'),
ConstantObject('dhNO')

concO, concN, concO2, concN2, concNO = ConstantObject('concO'), ConstantObject('concN'), ConstantObject('concO2'),
ConstantObject('concN2'), ConstantObject('concNO')

u = GridVariable('u')
rho_eqn = OpenSBLIEq(u, uref) # dimensional velocity
rho_store = OpenSBLIEq(rho, (10000.0/(8.3143)*(1.0/T)))
# initial concentrations
rhoOi, rhoNi, rhoO2i, rhoN2i, rhoNOi = GridVariable('rhoO'), GridVariable('rhoN'), GridVariable('rhoO2'), GridVariable('rhoN2'),
GridVariable('rhoNO')
rhou0_store = OpenSBLIEq(DataObject('rhou0'), rhou0)
rhou1_store = OpenSBLIEq(DataObject('rhou1'), rhou1)
# rhou2_store = OpenSBLIEq(DataObject('rhou2'), rhou2)
Tinf = GridVariable('T')
tempeq = OpenSBLIEq(Tinf, pref/(Rhat*(rho)))
Tempwall = GridVariable('Twalld')
tempwall = OpenSBLIEq(Tempwall, Twall)
rhof = OpenSBLIEq(DataObject('rhof'), rhoNi)
gama, Minf = ConstantObject("gama"), ConstantObject("Minf")
uref = ConstantObject("uref")
# Minf = uref*sqrt.()
# rhoE_store = OpenSBLIEq(DataObject('rhoE'),
Rhat*T*(3.0/2.0*(rhoOi/MO+rhoNi/MN)+5.0/2.0*(rhoO2i/MO2+rhoN2i/MN2+rhoNOi/MNO))+4.1868e6*(dhO*rhoOi/
MO+dhN*rhoNi/MN+dhNO*rhoNOi/MNO)+rhoO2i*evequilO2+rhoN2i*evequilN2+rhoNOi*evequilNO+0.5*((rhou0/rho)**2+
(rhou1/rho)**2)*rho)
rhoE_store = OpenSBLIEq(DataObject('rhoE'), rho*T/(gama*(gama-1)*Minf**2) + 0.5*(rhou0**2 + rhou1**2 )/rho)
self.eqns += [tempwall, tempeq, rhou0_eqn, rhou1_eqn, T_eqn, rho_store, rhou0_store, rhou1_store, rhoE_store, rhof]#rhoO,
rhoN, rhoO2, rhoN2, rhoNO,
if self.block.ndim == 3: # Periodic case, rhow = 0
self.eqns += [OpenSBLIEq(DataObject('rhou2'), 0.0)]
return

def generate_two_wall_equations(self, profiles, coeffs, directions, edges, normal_profile, normal_coeffs, poly_coordinates):
""" Generates the equations for 3D case with sidewalls.

:arg list profiles: Profile arrays for rhou0 and temperature.
:arg list coeffs: Coefficients for the polynomial fits.
:arg list directions: Directions normal to the wall.
:arg list edges: Grid indexes for the edge of the boundary-layer.
:arg list normal_profile: Profile for the wall normal velocity components.
:arg list normal_coeffs: Coefficients for the wall normal polynomial fit."""
self.eqns = []
names = ['rhou0', 'T', 'rhou1', 'rhou2']
# Hard code to spanwise direction for 2 walls
direction = 2
array_coord = self.coordinates[direction]
self.domain_length = self.block.deltas[direction]*(self.block.shape[direction]-1)
self.side_fact = GridVariable('side_fact')
side_fact_eqn = OpenSBLIEq(self.side_fact, Piecewise((1, array_coord <= self.domain_length/2.0), (-1, True)))

self.local_coordinate = GridVariable('local_coord')
local_coord_eqn = OpenSBLIEq(self.local_coordinate, Piecewise((array_coord, array_coord <= self.domain_length/2.0),
(self.domain_length-array_coord, True)))
self.eqns += [local_coord_eqn]
self.eqns += [side_fact_eqn]
# Create the piecewise equations formed from the boundary layers
bl_equations = self.form_mixed_equation(profiles, names, coeffs, directions, edges, normal_profile, normal_coeffs,
poly_coordinates)
```

```
# Set conservative values
rho, rhou0, rhou1, rhou2, T = GridVariable('rho'), GridVariable('rhou0'), GridVariable('rhou1'), GridVariable('rhou2'),
GridVariable('T')
# rho_eqn = OpenSBLIEq(rho, 1.0/T)
rho_eqn = OpenSBLIEq(rho, (10000.0/(8.3143))*(1.0/T))
rho_store = OpenSBLIEq(DataObject('rho'), rho)
rhou0_store = OpenSBLIEq(DataObject('rhou0'), rhou0)
rhou1_store = OpenSBLIEq(DataObject('rhou1'), rhou1)
rhou2_store = OpenSBLIEq(DataObject('rhou2'), rhou2)
gama, Minf = ConstantObject("gama"), ConstantObject("Minf")
rhoE_store = OpenSBLIEq(DataObject('rhoE'), rho*T/(gama*(gama-1)*Minf**2) + 0.5*(rhou0**2 + rhou1**2 + rhou2**2)/rho)
self.eqns += bl_equations + [rho_eqn, rho_store, rhou0_store, rhou1_store, rhou2_store, rhoE_store]
return

def fit_polynomial(self, coords, variable, bl_edge, n_coeffs):
""" Fits a polynomial to the input data, coefficients are returned.

:arg ndarray coords: Independent variable of the input data.
:arg ndarray variable: Dependent variable of the input data.
:arg int bl_edge: Array index at the edge of the boundary-layer.
:arg int n_coeffs: Desired number of coefficients for the polynomial.
:returns: ndarray: coeffs: Coefficients of the polynomial fit."""
coords = coords[0:bl_edge]
variable = variable[0:bl_edge]
# with warnings.catch_warnings():
# warnings.filterwarnings('error')
# try:
warnings.filterwarnings("ignore")
coeffs = poly.polyfit(coords, variable, n_coeffs)
# except np.RankWarning:
# print "Poorly conditioned fit"
# ffit = poly.polyval(coords, coeffs)
# plt.plot(coords, ffit, label='fit')
# plt.plot(coords, variable, label='original_data')
# plt.legend(loc="best")
# plt.show()
# Reverse coefficients so they are in descending order
return coeffs[::-1]

def solved_continuity(self, y, u, rho):
""" Solves the continuity equation to obtain the wall normal velocity profile.

:arg ndarray y: Dependent coordinate values.
:arg ndarray u: Streamwise velocity component values.
:arg ndarray rho: Density values.
:returns: ndarray: rhov: Array of values for the wall normal velocity components. """
# Grid offset delta to form derivative approximation
n = np.size(y)
ya2 = y[:]
delta, scale, re = 0.001, self.scale_factor, self.Re
rex0 = 0.5*(re/scale)**2
x0 = 0.5*re/scale**2
print(scale)

print('Reynolds number', re)
print('Scale', scale)
drudx, rhov = np.zeros_like(y), np.zeros_like(y)
# Local Reynolds number scaling to obtain a v profile
sqrex = np.sqrt(rex0)
delsx = np.sqrt(2.0)*scale*(x0)/sqrex
ya2 = delsx*ya2
d2y_u = spline(ya2, u, n, self.dudy, 0)
d2y_rho = spline(ya2, rho, n, 0, 0)
for j in range(0, n):
ya2[j] = delsx*ya2[j]
dstarp = delsx*np.sqrt((x0+delta)/(x0))
dstarm = delsx*np.sqrt((x0-delta)/(x0))
```

```
yp, ym = ya2[j]/dstarp, ya2[j]/dstarm
uxp = splint(ya2, u, d2y_u, n, yp)
uxm = splint(ya2, u, d2y_u, n, ym)
rhoxp = splint(ya2, rho, d2y_rho, n, yp)
rhoxm = splint(ya2, rho, d2y_rho, n, ym)
drudx[j] = (rhoxp*uxp-rhoxm*uxm)/(2.0*delta)
rhov[j] = rhov[j-1]-0.5*(ya2[j]-ya2[j-1])*(drudx[j]+drudx[j-1])

dy = y[1]
dvdy = (-1.83333333333334*rhov[0]+3.00000000000002*rhov[1]-1.50000000000003*rhov[2]+0.333333333333356*rhov[3]-
8.34657956545823e-15*rhov[4]+1.06910315192207e-15*rhov[5])/dy
v_n = rhov
dvdy = (-1.83333333333334*v_n[0]+3.00000000000002*v_n[1]-1.50000000000003*v_n[2]+0.333333333333356*v_n[3]-
8.34657956545823e-15*v_n[4]+1.06910315192207e-15*v_n[5])/dy

return v_n, dvdy

def solve_continuity(self, y, u, rho):
""" Solves the continuity equation to obtain the wall normal velocity profile.

:arg ndarray y: Dependent coordinate values.
:arg ndarray u: Streamwise velocity component values.
:arg ndarray rho: Density values.
:returns: ndarray: rhov: Array of values for the wall normal velocity components. """
# Grid offset delta to form derivative approximation
n = np.size(y)
ya2 = y[:]
delta, scale, re = 0.001, self.scale_factor, self.Re
rex0 = 0.5*(re/scale)**2
x0 = 0.5*re/scale**2
# print "Domain inlet is when the boundary-layer has developed for a length of x0 = %.10f" % x0
drudx, rhov = np.zeros_like(y), np.zeros_like(y)
# Local Reynolds number scaling to obtain a v profile
sqrex = np.sqrt(rex0)
delsx = np.sqrt(2.0)*scale*(x0)/sqrex
ya2 = delsx*ya2
d2y_u = spline(ya2, u, n, self.dudy, 0)
d2y_rho = spline(ya2, rho, n, 0, 0)
for j in range(0, n):
ya2[j] = delsx*ya2[j]
dstarp = delsx*np.sqrt((x0+delta)/(x0))
dstarm = delsx*np.sqrt((x0-delta)/(x0))
yp, ym = ya2[j]/dstarp, ya2[j]/dstarm
uxp = splint(ya2, u, d2y_u, n, yp)
uxm = splint(ya2, u, d2y_u, n, ym)
rhoxp = splint(ya2, rho, d2y_rho, n, yp)
rhoxm = splint(ya2, rho, d2y_rho, n, ym)
drudx[j] = (rhoxp*uxp-rhoxm*uxm)/(2.0*delta)
rhov[j] = rhov[j-1]-0.5*(ya2[j]-ya2[j-1])*(drudx[j]+drudx[j-1])
return rhov

def load_similarity(self):
""" Solves the compressible boundary-layer equations via similarity solution."""
Re, xMach, Tinf = self.Re, self.xMach, self.Tinf
Pr, gama = 0.72, 1.4 # Prandtl number, ratio of specific heats
# bl = Boundary_layer_profile(xMach, Pr, gama, -1, Re, Tinf) # -1 for Tw sets an adiabatic wall
bl = Boundary_layer_profile(xMach, Pr, gama, -1, Re, Tinf) # -1 for Tw sets an adiabatic wall
y, u, T, rho, n = bl.y, bl.u, bl.T, 1.0/bl.T, np.size(bl.y)
self.Twall, self.scale_factor = bl.Twall, bl.scale_factor # Wall temperature and scale factor from the similarity solution
self.dudy = bl.dudy # du/dy at the wall
return y, u, T, rho, n

def interpolate_onto_grid(self, y_in, y_out, var_in, y0, yn):
# Create interpolating second derivative spline
# n = size of the original data
n = np.size(y_in)
n_out = np.size(y_out)
```

```
d2y = spline(y_in, var_in, n, y0, yn)
# Array for variable interpolated onto the grid
var_out = np.zeros_like(y_out)
for i in range(n_out):
var_out[i] = splint(y_in, var_in, d2y, n, y_out[i])
return var_out

def find_edge_of_bl(self, variable, tolerance):
""" Finds the edge of the boundary layer and returns the index of that grid point.

:arg ndarray variable: Array of values for a given flow variable.
:arg float tolerance: Stopping tolerance for the difference between two successive grid points.
:returns: int: index: Index of the boundary-layer edge."""
index = 1
while np.abs(variable[index]-variable[index-1]) > tolerance:
index += 1
return index

def freestream_value(self, variable, index):
""" Returns the value of a flow variable for a given grid index.

:arg ndarray variable: Array of values for a given flow variable.
:arg index Array index to use.
:returns: float: variable[index]: Value of the flow variable at that index."""
return variable[index]
```