```
#!/usr/bin/env python
# ----------------------------------------------------------------------------
from opensbli import *
from sympy import sin, cos, sinh, tanh, exp, pi, log, Piecewise, Or
from opensbli.utilities.helperfunctions import substitute_simulation_parameters
import chemistry
# ----------------------------------------------------------------------------

##########################################################################################################
##############################
# #
# Inputs #
# #
##########################################################################################################
##############################
Chemistry_model = 'Park01' # Choose from 'GGS' or 'Park01' or 'none'
Physical = {
'Re' :'1.0',
'Sc' :'0.71',
'uref' :'0.0', # '731.6071'
'pref' :'10000.0',
'rhoref' :'0.006',
'pexp' :'0.0',
}
Grid = {
'dt' :'0.0000002',
'niter' :'4000',
'block0np0' :'11', # 120
'block0np1' :'11', # 375
'Delta0block0' :'15.0*0.0133482/block0np0',
'Delta1block0' :'1.0/(block0np1-1)',
'Ly' :'100.0/15.0*0.133482',
'stretch' :'1.0', # 3.4
}
Gas = {
'Rhat' :'8314.3',
'MO' :'16.0',
'MO2' :'32.0',
'MN' :'14.0',
'MN2' :'28.0',
'MNO' :'30.0',
'dhO' :'59.544',
'dhN' :'112.951',
'dhNO' :'21.6009',
'thetavO2' :'2270.0',
'thetavN2' :'3390.0',
'thetavNO' :'2740.0',
}
set_constants = {**Physical , **Grid , **Gas} # REF[1]: Unpacks the dictionaries in a single one

##########################################################################################################
##############################
# #
# Simulation Equations #
# #
##########################################################################################################
##############################
# Number of simulation dimensions
ndim = 2
stats = False

# Define the compresible Navier-Stokes equations in Einstein notation
mass_O = "Eq(Der(rhoO,t), - Skew(rhoO*u_j,x_j) + Der(mu/(Re*Sc)*Der(yO,x_j),x_j) + wdotO )"
mass_O2 = "Eq(Der(rhoO2,t), - Skew(rhoO2*u_j,x_j) + Der(mu/(Re*Sc)*Der(yO2,x_j),x_j) + wdotO2 )"
mass_N = "Eq(Der(rhoN,t), - Skew(rhoN*u_j,x_j) + Der(mu/(Re*Sc)*Der(yN,x_j),x_j) + wdotN )"
mass_N2 = "Eq(Der(rhoN2,t), - Skew(rhoN2*u_j,x_j) + Der(mu/(Re*Sc)*Der(yN2,x_j),x_j) + wdotN2)"
mass_NO = "Eq(Der(rhoNO,t), - Skew(rhoNO*u_j,x_j) + Der(mu/(Re*Sc)*Der(yNO,x_j),x_j) + wdotNO)"
```

```
momentum = "Eq(Der(rhou_i,t), - Skew(rhou_i*u_j, x_j) - Der(p,x_i) + Der(tau_i_j,x_j))"
evib = "Eq(Der(rhoev,t), - Skew(rhoev*u_j,x_j) + ((rhoO2*eveqO2+rhoN2*eveqN2+rhoNO*eveqNO)*rho/(rhoO2+rhoN2+rhoNO) -
rhoev)/tau )" \
# " Der(mu/(Re*Sc)*(evO2*Der(yO2,x_j)+evN2*Der(yN2,x_j)+evNO*Der(yNO,x_j)),x_j) )" # ## - Der(qv_j,x_j)##+
wdotO2*evO2+wdotN2*evN2+wdotNO*evNO - Der(qv_j,x_j)
energy = "Eq(Der(rhoE,t), - Skew(rhoE*u_j,x_j) - Conservative(p*u_j,x_j) " \
"+Der(mu/(Re*Sc)*Rhat*T*(5.0/(2.0*MO)*Der(yO,x_j)+5.0/(2.0*MN)*Der(yN,x_j)+7.0/(2.0*MO2)*Der(yO2,x_j)+7.0/
(2.0*MN2)*Der(YN2,x_j)" \
"+7.0/(2.0*MNO)*Der(YNO,x_j)),x_j) + Der(mu/(Re*Sc)*4.1868e6*(dhO/MO*Der(YO,x_j)+dhN/MN*Der(YN,x_j)+dhNO/
MNO*Der(YNO,x_j)),x_j) " \
"+ Der(mu/(Re*Sc)*(evO2*Der(yO2,x_j)+evN2*Der(yN2,x_j)+evNO*Der(yNO,x_j)),x_j) - Der(q_j,x_j) - Der(qv_j,x_j) +
Der(u_i*tau_i_j ,x_j))"
scalar = "Eq(Der(rhof,t), - Skew(rhof*u_j,x_j) + Der(mu/(Re*Sc)*Der(f,x_j),x_j) + Tvref*1.0e-12)" # non-reacting scalar is useful as a
(diffusing) marker of original fluid regions

# ----------------------------------------------------------------------------------------------------------------
# Constituent Equations #
# ----------------------------------------------------------------------------------------------------------------

# Variable relations used in the system
velocity = "Eq(u_i, rhou_i/rho)"
mixturefraction = "Eq(f, rhof/rho)"
pressure = "Eq(p, Rhat*T*(rhoO/MO+rhoO2/MO2+rhoN/MN+rhoN2/MN2+rhoNO/MNO))"
temperature = "Eq(T, (rhoE - rhoev*(rhoO2+rhoN2+rhoNO)/(rhoO+rhoO2+rhoN+rhoN2+rhoNO) - dhf -
rho*(1./2.)*(KD(_i,_j)*u_i*u_j))/(Rhat*(3.0/2.0*(rhoO/MO+rhoN/MN)+5.0/2.0*(rhoO2/MO2+rhoN2/MN2+rhoNO/MNO))) )"

# tempv = "Eq(Tv, thetavnum/(ysumM*log(1.0+thetavnum*Rhat/(rhoev*(rhoO2+rhoN2+rhoNO)/rho))))"
tempv = "Eq(Tvref, thetavnum/(ysumM*log(1.0+thetavnum*Rhat/(rhoev*(rhoO2+rhoN2+rhoNO)/
(rhoO+rhoO2+rhoN+rhoN2+rhoNO)))))"
tempvNR = "Eq(Tv, Tvref + " \
"(rhoev*(rhoO2+rhoN2+rhoNO)/(rhoO+rhoO2+rhoN+rhoN2+rhoNO) - ( (rhoO2*thetavO2*(Rhat/MO2)/(exp(thetavO2/Tvref)-1))+
(rhoN2*thetavN2*(Rhat/MN2)/(exp(thetavN2/Tvref)-1))+(rhoNO*thetavNO*(Rhat/MNO)/(exp(thetavNO/Tvref)-1)) ) )" \
"/" \
"( (rhoO2*thetavO2**(2.0)*(Rhat/MO2)*exp(thetavO2/Tvref)/(Tvref**(2.0)*(exp(thetavO2/Tvref)-1)**(2.0))) +
(rhoN2*thetavN2**(2.0)*(Rhat/MN2)*exp(thetavN2/Tvref)/(Tvref**(2.0)*(exp(thetavN2/Tvref)-1)**(2.0))) +
(rhoNO*thetavNO**(2.0)*(Rhat/MNO)*exp(thetavNO/Tvref)/(Tvref**(2.0)*(exp(thetavNO/Tvref)-1)**(2.0))) ) )"
evequilO2 = "Eq(eveqO2, thetavO2*Rhat/(MO2*(exp(thetavO2/T)-1.0)))"
evequilN2 = "Eq(eveqN2, thetavN2*Rhat/(MN2*(exp(thetavN2/T)-1.0)))"
evequilNO = "Eq(eveqNO, thetavNO*Rhat/(MNO*(exp(thetavNO/T)-1.0)))"
evvO2 = "Eq(evO2, thetavO2*Rhat/(MO2*(exp(thetavO2/Tv)-1.0)))"
evvN2 = "Eq(evN2, thetavN2*Rhat/(MN2*(exp(thetavN2/Tv)-1.0)))"
evvNO = "Eq(evNO, thetavNO*Rhat/(MNO*(exp(thetavNO/Tv)-1.0)))"
timefactorO2 = "Eq(ptauO2, (rhoO/MO*exp(129.0*(T**(-1.0/3.0)-0.0271)-18.42)" \
"+rhoO2/MO2*exp(129.0*(T**(-1.0/3.0)-0.0300)-18.42)" \
"+rhoN/MN*exp(129.0*(T**(-1.0/3.0)-0.0265)-18.42)" \
"+rhoN2/MN2*exp(129.0*(T**(-1.0/3.0)-0.0295)-18.42)" \
"+rhoNO/MNO*exp(129.0*(T**(-1.0/3.0)-0.0298)-18.42))/ysum)"
timefactorN2 = "Eq(ptauN2, (rhoO/MO*exp(220.0*(T**(-1.0/3.0)-0.0268)-18.42)" \
"+rhoO2/MO2*exp(220.0*(T**(-1.0/3.0)-0.0295)-18.42)" \
"+rhoN/MN*exp(220.0*(T**(-1.0/3.0)-0.0262)-18.42)" \
"+rhoN2/MN2*exp(220.0*(T**(-1.0/3.0)-0.0290)-18.42)" \
"+rhoNO/MNO*exp(220.0*(T**(-1.0/3.0)-0.0293)-18.42))/ysum)"
timefactorNO = "Eq(ptauNO, (rhoO/MO*exp(168.0*(T**(-1.0/3.0)-0.0270)-18.42)" \
"+rhoO2/MO2*exp(168.0*(T**(-1.0/3.0)-0.0298)-18.42)" \
"+rhoN/MN*exp(168.0*(T**(-1.0/3.0)-0.0264)-18.42)" \
"+rhoN2/MN2*exp(168.0*(T**(-1.0/3.0)-0.0293)-18.42)" \
"+rhoNO/MNO*exp(168.0*(T**(-1.0/3.0)-0.0295)-18.42))/ysum)"

# fluid properties
viscosity = "Eq(mu, ((yO2+yN2+yNO)*0.1*exp(-11.2202)*T**(0.021823*log(T)+0.34357)+(yO+yN)*0.1*exp(-
11.7344)*T**(0.022652*log(T)+0.342509))*(1.0-exp(-0.010568*T)) )" # NDS simplication of Blottner/Sutherland
conductivity = "Eq(kappa, (1410.0*(yO2+yN2+yNO)*0.1*exp(-11.2202)*T**(0.021823*log(T)+0.34357)+2210.0*(yO+yN)*0.1*exp(-
11.7344)*T**(0.022652*log(T)+0.342509))*(1.0-exp(-0.010568*T)) )" # NDS simplification of Blottner/Sutherland/Eucken (see
Viscosity_model3.m and Viscosity_model_v2_optimise.m)
conductivity_vib = "Eq(kappav, (286.7*(yO2+yN2+yNO)*0.1*exp(-11.2202)*T**(0.021823*log(T)
+0.34357)+519.6*(yO+yN)*0.1*exp(-11.7344)*T**(0.022652*log(T)+0.342509))*(1.0-exp(-0.010568*T)) )" # NDS simplification of
```

```
Blottner/Sutherland/Eucken (see Viscosity_model3.m and Viscosity_model_v2_optimise.m) vibration is just taken as one factor of
Rhat/Mhat(needs a closer look maybe)

# chemistry
molefractionO = "Eq(yO, rhoO/(MO*ysum))"
molefractionO2 = "Eq(yO2, rhoO2/(MO2*ysum))"
molefractionN = "Eq(yN, rhoN/(MN*ysum))"
molefractionN2 = "Eq(yN2, rhoN2/(MN2*ysum))"
molefractionNO = "Eq(yNO, rhoNO/(MNO*ysum))"


# ------------------------------------------------------------------------------------------------------------
# Substitutions #
# ------------------------------------------------------------------------------------------------------------

# Substitutions used in the equations
stress_tensor = "Eq(tau_i_j, (mu/Re)*(Der(u_i,x_j)+ Der(u_j,x_i)- (2/3)* KD(_i,_j)* Der(u_k,x_k)))"
heat_flux = "Eq(q_j, -(kappa/Re)*Der(T,x_j))"
heat_flux_vib = "Eq(qv_j, -(kappav/Re)*Der(Tv,x_j))"
evibration = "Eq(ev, rhoev/rho)"
density = "Eq(rho, (rhoO+rhoO2+rhoN+rhoN2+rhoNO))"
molesum = "Eq(ysum, (rhoO/MO+rhoO2/MO2+rhoN/MN+rhoN2/MN2+rhoNO/MNO))"
molesumM = "Eq(ysumM, (rhoO2/MO2+rhoN2/MN2+rhoNO/MNO))"
hformation = "Eq(dhf, 4.1868e6*(dhO*rhoO/MO+dhN*rhoN/MN+dhNO*rhoNO/MNO))"
timeconst = "Eq(tau, (rhoO2/MO2+rhoN2/MN2+rhoNO/MNO)*101325.0/(p*(rhoO2/(MO2*ptauO2)+rhoN2/
(MN2*ptauN2)+rhoNO/(MNO*ptauNO))))"
thetavset = "Eq(thetavnum, (thetavO2*rhoO2/MO2+thetavN2*rhoN2/MN2+thetavNO*rhoNO/MNO))"


# make substitutions
substitutions = [stress_tensor, heat_flux, heat_flux_vib, evibration, density, molesum, molesumM, hformation, timeconst,
thetavset]
# Chemistry
substitutions += chemistry.substitutions(Chemistry_model)



# ------------------------------------------------------------------------------------------------------------
# Expanding the Equations #
# ------------------------------------------------------------------------------------------------------------

# Instantiate EinsteinEquation class for expanding the Einstein indices in the equations
eq = EinsteinEquation()
# symbol for the coordinate system in the equations
coordinate_symbol = "x"

# Constants that are used
constants = ["Re", "Sc", "uref", "pref", "rhoref", "pexp", "Rhat", "MO", "MO2", "MN", "MN2", "MNO","rhoEref",
"dhO", "dhN", "dhNO", "thetavO2", "thetavN2", "thetavNO"]

# Add the constants used for the chemistry model - (their names)
constants += list(chemistry.constants(Chemistry_model))

# Expand the simulation equations
simulation_eq = SimulationEquations()
base_eqns = [mass_O, mass_O2, mass_N, mass_N2, mass_NO, momentum, evib, energy, scalar]
for i, base in enumerate(base_eqns):
base_eqns[i] = eq.expand(base, ndim, coordinate_symbol, substitutions, constants)
for eqn in base_eqns:
simulation_eq.add_equations(eqn)

# Expand the constituent relations
constituent = ConstituentRelations()
constituent_eqns = [velocity, pressure, temperature, mixturefraction, viscosity, conductivity, conductivity_vib,
molefractionO, molefractionO2, molefractionN, molefractionN2, molefractionNO, evequilO2, evequilN2,
evequilNO, evvO2, evvN2, evvNO, timefactorO2, timefactorN2, timefactorNO, tempv,tempvNR] #,tempvNR
```

```
# Chemistry constiuent equations
constituent_eqns+= chemistry.constituent(Chemistry_model) # rate of reaction

for i, CR in enumerate(constituent_eqns):
constituent_eqns[i] = eq.expand(CR, ndim, coordinate_symbol, substitutions, constants)
for eqn in constituent_eqns:
constituent.add_equations(eqn)

# Create a simulation block
block = SimulationBlock(ndim, block_number=0)

# Local dictionary for parsing the expressions
local_dict = {"block": block, "GridVariable": GridVariable, "DataObject": DataObject}

###########################################################################################################
##############################
# #
# Grid #
# #
###########################################################################################################
##############################

# Set the discretisation schemes - low storage
schemes = {}
fns = 'u0 u1 T'
cent = StoreSome(4,fns)
schemes[cent.name] = cent
rk = RungeKuttaLS(4)
# rk = RungeKutta(3)
schemes[rk.name] = rk
block.set_discretisation_schemes(schemes)

# ----------------------------------------------------------------------------------------------------------

dx, dy = block.deltas
x, y = symbols('x0:%d' % ndim, **{'cls': DataObject})
i, j = block.grid_indexes
nx, ny, Ly, stretch = symbols('block0np0 block0np1 Ly stretch', **{'cls': ConstantObject})
Lx = nx*dx
q_vector = flatten(simulation_eq.time_advance_arrays)
grid_equations = []
stretch_eqn = 0.5*Ly*sinh(stretch*(j-(ny-1)/2)/((ny-1)/2))/sinh(stretch)
grid_equations += [Eq(x, i*dx), Eq(y,stretch_eqn)]

# Metrics: Stretching or curvature of the grid
metriceq = MetricsEquation()
metriceq.generate_transformations(ndim, coordinate_symbol, [(False, False), (True, False)], 2)
simulation_eq.apply_metrics(metriceq)

# Boundary Conditions
boundaries = []
direction = 0
boundaries += [PeriodicBC(direction, side=0)]
boundaries += [PeriodicBC(direction, side=1)]
direction = 1
boundaries += [SymmetryBC(direction, 0)]
boundaries += [SymmetryBC(direction, 1)]
block.set_block_boundaries(boundaries)

# # Binomial Filter
# j = block.grid_indexes[1]
# nfilter = 20
# grid_condition = Or(j<=nfilter,j>=ny-nfilter)
# BF = BinomialFilter(block, order=2, grid_condition=grid_condition ,sigma=0.20)


###########################################################################################################
##############################
```

```
# #
# Initial Conditions #
# #
#############################################################################################################
uref, pref, rhoref,rhoEref, Rhat, MO, MO2, MN, MN2, MNO, dhO, dhN, dhNO, thetavO2, thetavN2, thetavNO \
= symbols('uref pref rhoref rhoEref Rhat MO MO2 MN MN2 MNO dhO dhN dhNO thetavO2 thetavN2 thetavNO', **{'cls':
ConstantObject})
rhoO, rhoO2, rhoN, rhoN2, rhoNO,rho, u, v, p, T ,f, ev, evequilO2, evequilN2, evequilNO,evO2, evN2, evNO,evO2eq, evN2eq,
evNOeq, Tv \
= symbols('rhoO, rhoO2, rhoN, rhoN2, rhoNO,rho, u, v, p, T, f, ev, evequilO2, evequilN2, evequilNO,evO2, evN2, evNO,evO2eq,
evN2eq, evNOeq, Tv', **{'cls': GridVariable})


initial_equations = [

# Velcity/Scalar
Eq(u, 0.0), # uref*tanh(2.0*y/0.0133482)
Eq(v, 0.0), # 0.01*uref*cos(2.0*pi*x/Lx)*exp(-(y/0.0133482)**2.0/10.0)
Eq(f, 0.0), # 0.5 * (1.0 + tanh(2.0 * y / 0.0133482))

# Species/Density
Eq(rhoO,rhoref*0.000),
Eq(rhoO2,rhoref*0.21),
Eq(rhoN,rhoref*0.000),
Eq(rhoN2,rhoref*0.79),
Eq(rhoNO,rhoref*0.000),
Eq(rho,rhoO+rhoO2+rhoN+rhoN2+rhoNO),

# q Vectors
Eq(q_vector[0],rhoO),
Eq(q_vector[1],rhoO2),
Eq(q_vector[2],rhoN),
Eq(q_vector[3],rhoN2),
Eq(q_vector[4],rhoNO),
Eq(q_vector[5],rho*u),
Eq(q_vector[6],rho*v),
Eq(q_vector[7],1000.0), # Edited this to ev(Tv)
Eq(q_vector[8],5.0e+6*rho),
Eq(q_vector[9],rho*f),
]

# Parse the initial conditions
initial = GridBasedInitialisation()
initial.add_equations(grid_equations + initial_equations)


#############################################################################################################
# #
# Printouts & Monitor Points / Latex #
# #
#############################################################################################################

# set the IO class to write out arrays
kwargs = {'iotype': "Write"}
h5 = iohdf5(save_every=500, **kwargs)
h5.add_arrays(simulation_eq.time_advance_arrays + [x, y])
h5.add_arrays([DataObject('T'),DataObject('Tv'),DataObject('p'),DataObject('evO2'),DataObject('evN2'),DataObject('evNO')])
block.setio(copy.deepcopy(h5))

# set monitor points
arrays = ['rhoO','rhoO2','rhoN','rhoN2','rhoNO','rhoE','rhoev','T','Tv']
# arrays = ['wdotO','wdotO2','wdotN','wdotN2','wdotNO','rhoE','rhoev','T','Tv']
arrays = [block.location_dataset('%s' % dset) for dset in arrays]
indices = [(1,1),(1,1),(1,1),(1,1),(1,1),(1,1),(1,1),(1,1),(2,2)] # ('block0np0/4', '(block0np1-1)/2')
```

```
SM = SimulationMonitor(arrays, indices, block, print_frequency=2,fp_precision=12, output_file='output.log')

# ----------------------------------------------------------------------------------------------------------

# Write the expanded equations to a Latex file with a given name and titile
latex = LatexWriter()
latex.open('equations.tex', "Einstein Expansion of the simulation equations")
latex.write_string("Simulation equations\n")
for index, eq in enumerate(flatten(simulation_eq.equations)):
latex.write_expression(eq)
latex.write_string("Constituent relations\n")
for index, eq in enumerate(flatten(constituent.equations)):
latex.write_expression(eq)
latex.close()

#############################################################################################################
##############################
# #
# Miscellaneous #
# #
#############################################################################################################
##############################
# set chemical model constants
set_constants.update(chemistry.constants(Chemistry_model))

# set the equations to be solved on the block
block.set_equations([copy.deepcopy(constituent), copy.deepcopy(simulation_eq), initial, metriceq]) # "BF.equation_classes" is for
the filter

# Discretise the equations on the block
block.discretise()

# create an algorithm from the discretised computations
alg = TraditionalAlgorithmRK(block, simulation_monitor=SM)

# set the simulation data type, for more information on the datatypes see opensbli.core.datatypes
SimulationDataType.set_datatype(Double)

# Write the code for the algorithm
OPSC(alg) # ,OPS_V2=True

# Add the data from "Input" section
substitute_simulation_parameters(set_constants.keys(), set_constants.values())


print_iteration_ops(NaN_check='rhoN2')



#############################################################################################################
################################################
# #
# Refrences #
# #
# [1]: Unpacks the dictionaries in a single one #
# https://stackoverflow.com/questions/13361510/typeerror-unsupported-operand-types-for-dict-items-and-dict-items #
#############################################################################################################
################################################
```