""" David J. Lusher 08/2020: WENO non-linear filter for shock-capturing."""

from opensbli import \*

from sympy import symbols, exp, pprint, Piecewise, binomial, Min, sqrt, Equality, tanh

from opensbli.core.opensbliobjects import DataObject, ConstantObject, GroupedPiecewise

from opensbli.equation\_types.opensbliequations import OpenSBLIEquation, NonSimulationEquations, ConstituentRelations

from opensbli.postprocess.post\_process\_eq import \*

from opensbli.core.kernel import ConstantsToDeclare as CTD

from opensbli.code\_generation.algorithm.common import \*

from opensbli.utilities.user\_defined\_kernels import UserDefinedEquations

from opensbli.schemes.spatial.weno import \*

from opensbli.core.boundary\_conditions.bc\_core import WallBC

from sympy.functions.elementary.piecewise import ExprCondPair

class WENOFilter(NonSimulationEquations):

""" Class to apply a WENO-based non-linear filter after a full time-step of a non-dissipative high order base scheme. The dissipative

portion of a WENO procedure is used in characteristic space, by substracting a central difference flux approximation of order n+1. The shock location sensor

uses the absolute difference of the non-linear to ideal WENO weights. The amount of dissipation is controlled by Mach number or dilatation/vorticity sensors. The governing

equations in the user script should be central derivatives in a skew-symmetric formulation to improve numerical stability."""

def \_\_init\_\_(self, block, order, metrics=None, dissipation\_sensor='Ducros', Mach\_correction=False):

self.reconstruction\_kernels = []

self.residual\_kernels = []

block.shock\_filter = True

# Choice of how to evaluate the amount of dissipation to be added (varies spatially in the domain)

self.dissipation\_sensor = dissipation\_sensor

self.Mach\_correction = Mach\_correction

self.block = block

self.ndim = block.ndim

self.order = order

self.equation\_classes = []

self.nhalos = 5

# Store the value of the dissipation control sensor for debugging

self.store\_kappa = True

# Scheme used to form the non-linear filter

self.scheme\_type = "\*\*{\'scheme\':\'Weno\'}"

# Check if the problem needs a metric transformation of the equations

self.process\_metrics(metrics)

self.constants = ["Re", "Pr","gama", "Minf", "SuthT", "RefT"]

# Ensure gama has been added to the constants to define

self.gama = ConstantObject('gama')

CTD.add\_constant(self.gama)

# Einstein class to expand equations

self.EE = EinsteinEquation()

# Main class to generate the filter

self.main(order, block)

return

def detect\_wall\_boundaries(self):

""" The shock-filter is turned off in the near-wall region. This function detects which directions, if any, have

wall boundary conditions."""

self.wall\_boundaries = [[False, False] for \_ in range(self.ndim)]

try:

for direction in range(self.ndim):

for side in [0,1]:

if isinstance(self.block.boundary\_types[direction][side], WallBC):

self.wall\_boundaries[direction][side] = True

except:

raise ValueError("Please set boundary conditions on the block before calling the shock filter.")

return

def process\_metrics(self, metrics):

# Uniform mesh

if metrics is None:

self.metric\_class = None

self.stretched, self.curvilinear = False, False

# Stretched or curvlinear mesh

else:

self.metric\_class = metrics

# Check whether the mesh is only stretched or full curvilinear

if sum(self.metric\_class.stretching\_metric) > 0:

self.stretched = True

if sum(self.metric\_class.curvilinear\_metric) > 0:

self.curvilinear = True

else:

self.curvilinear = False

return

def create\_weno\_equations(self):

# Define the compresible Navier-Stokes equations in Einstein notation, depending on the metric input

scheme\_type = self.scheme\_type

# Uniform mesh, no stretching or curvilinear terms

if self.metric\_class is None:

coordinate\_symbol = "x"

mass = "Eq(Der(rho,t), - Conservative(rho\*u\_j,x\_j,%s))" % scheme\_type

momentum = "Eq(Der(rhou\_i,t) , -Conservative(rhou\_i\*u\_j + KD(\_i,\_j)\*p,x\_j , %s))" % scheme\_type

energy = "Eq(Der(rhoE,t), - Conservative((p+rhoE)\*u\_j,x\_j, %s))" % scheme\_type

output\_equations = flatten([self.EE.expand(eq, self.ndim, coordinate\_symbol, [], self.constants) for eq in flatten([mass, momentum, energy])])

else:

# Full curvilinear

if self.curvilinear:

coordinate\_symbol = "xi"

a = "Conservative(detJ \* rho\*U\_j,xi\_j,%s)" % scheme\_type

mass = "Eq(Der(rho,t), - %s/detJ)" % (a)

a = "Conservative(detJ \* (rhou\_i\*U\_j + p\*D\_j\_i), xi\_j , %s)" % scheme\_type

momentum = "Eq(Der(rhou\_i,t) , - %s/detJ)" % (a)

a = "Conservative(detJ \* (p+rhoE)\*U\_j,xi\_j, %s)" % scheme\_type

energy = "Eq(Der(rhoE,t), - %s/detJ)" % (a)

output\_equations = flatten([self.EE.expand(eq, self.ndim, coordinate\_symbol, [], self.constants) for eq in flatten([mass, momentum, energy])])

# Only stretching is applied

else:

coordinate\_symbol = "x"

mass = "Eq(Der(rho,t), - Conservative(rho\*u\_j,x\_j,%s))" % scheme\_type

momentum = "Eq(Der(rhou\_i,t) , -Conservative(rhou\_i\*u\_j + KD(\_i,\_j)\*p,x\_j , %s))" % scheme\_type

energy = "Eq(Der(rhoE,t), - Conservative((p+rhoE)\*u\_j,x\_j, %s))" % scheme\_type

governing\_eq = flatten([self.EE.expand(eq, self.ndim, coordinate\_symbol, [], self.constants) for eq in flatten([mass, momentum, energy])])

output\_equations = flatten([self.metric\_class.apply\_transformation(eqn) for eqn in (governing\_eq)])

return output\_equations

def create\_kernel(self, name, equations, halo\_type, block):

filter\_class = UserDefinedEquations()

filter\_class.algorithm\_place = InTheSimulation(frequency=False)

filter\_class.computation\_name = name

filter\_class.order = self.component\_counter

# Add the halo type to extend the range of evaluation

filter\_class.halos = halo\_type

filter\_class.add\_equations(equations)

return filter\_class

def add\_kernel(self, kernel):

""" Adds the finished kernels to the storage."""

if isinstance(kernel, list):

for ker in kernel:

self.equation\_classes.append(ker)

else:

self.equation\_classes.append(kernel)

return

def group\_by\_direction(self, eqs):

""" Groups the input equations by the direction (x0, x1, ...) they depend upon.

:arg list eqs: List of equations to group by direction.

:returns: dict: grouped: Dictionary of {direction: equations} key, value pairs for equations grouped by direction."""

all\_WDS = []

for eq in eqs:

all\_WDS += list(eq.atoms(WenoDerivative))

grouped = {}

for cd in all\_WDS:

direction = cd.get\_direction[0]

if direction in grouped.keys():

grouped[direction] += [cd]

else:

grouped[direction] = [cd]

return grouped

def constituent\_relations(self, block):

""" Evalutes the constiteunt relations on the state at the end of a full step

of the Runge-Kutta explicit time-stepper. Only the invscid terms are evaluted here (no viscosity relation)"""

CR\_eqns = []

# Conservative Q array entries from the current state

rho, rhoE = self.solution\_vector[0], self.solution\_vector[-1]

momentum\_components = [self.solution\_vector[i+1] for i in range(block.ndim)]

inv\_rho = GridVariable('inv\_rho')

CR\_eqns += [OpenSBLIEq(inv\_rho, 1.0/rho)]

# Primitive components and speed of sound

velocity\_components = [block.location\_dataset('u%d' % i ) for i in range(block.ndim)]

CR\_eqns += [OpenSBLIEq(x, y\*inv\_rho) for (x, y) in zip(velocity\_components, momentum\_components)]

# Pressure equation

p, a = block.location\_dataset('p'), block.location\_dataset('a')

# rhoE = p/(gama-1) + 0.5\*(rhou\*\*2)/rho

CR\_eqns += [OpenSBLIEq(p, (self.gama-1)\*(rhoE - 0.5\*sum([dset\*\*2 for dset in momentum\_components])\*inv\_rho))]

CR\_eqns += [OpenSBLIEq(a, sqrt(self.gama\*p\*inv\_rho))]

# Projected velocities if full curvilinear coordinates are being used

if self.curvilinear:

metric\_vel = "Eq(U\_i, D\_i\_j\*u\_j)"

CR\_eqns += flatten([self.EE.expand(metric\_vel, self.ndim, "x", [], self.constants)])

# Optiional Low Mach number correction

if self.Mach\_correction:

M\_var, M\_eqns = self.evaluate\_Yee\_Mach\_sensor(block)

CR\_eqns += M\_eqns

CR\_halos = []

for \_ in range(block.ndim):

CR\_halos.append([self.halo\_type, self.halo\_type])

CR\_kernel = self.create\_kernel('Constituent Relations evaluation', CR\_eqns, CR\_halos, block)

self.component\_counter += 1

# Add the kernel

self.add\_kernel(CR\_kernel)

return

def evaluate\_Yee\_Mach\_sensor(self, block):

""" Sensor controlling the amount of dissipation to apply. Turns the filter off in low-Mach regions.

(High Order Filter Methods for Wide Range of Compressible Flow Speeds, Yee, 2010). Primitive variables are computed locally

to reduce the number of input/output arrays to the kernel."""

Mach\_equations = []

# Evaluate the local Mach number

M, inv\_a, p, inv\_rho = symbols('M inv\_a p inv\_rho', \*\*{'cls' : GridVariable})

# Conservative quantities

rho, rhoE = self.solution\_vector[0], self.solution\_vector[-1]

momentum\_components = [self.solution\_vector[i+1] for i in range(block.ndim)]

velocity\_components = [GridVariable('u%d' % i ) for i in range(block.ndim)]

Mach\_equations += [OpenSBLIEq(inv\_rho, 1.0/rho)]

Mach\_equations += [OpenSBLIEq(x, y\*inv\_rho) for (x, y) in zip(velocity\_components, momentum\_components)]

Mach\_equations += [OpenSBLIEq(p, (self.gama-1)\*(rhoE - 0.5\*sum([dset\*\*2 for dset in momentum\_components])\*inv\_rho))]

Mach\_equations += [OpenSBLIEq(inv\_a, 1.0/sqrt(self.gama\*p\*inv\_rho))]

Mach\_equations += [OpenSBLIEq(M, sqrt(sum(dset\*\*2 for dset in velocity\_components))\*inv\_a)]

# Evaluation of the kappa parameter to control the amount of dissipaton

Mach\_correct = block.location\_dataset('Mach\_sensor')

Mach\_equations += [OpenSBLIEq(Mach\_correct, Min(0.5\*M\*\*2 \* sqrt(4+(1-M\*\*2)\*\*2) / (1+M\*\*2), 1))]

return Mach\_correct, Mach\_equations

def evaluate\_Ducros\_sensor(self, block):

# Add a shock sensor for the WENO filter

SS = ShockSensor()

output\_eqns, kappa = SS.ducros\_equations(block, "x", metrics=self.metric\_class, name='kappa')

# Low Mach number correction

if self.Mach\_correction:

sensor\_evaluation = output\_eqns[-1]

del output\_eqns[-1]

output\_eqns += [OpenSBLIEq(sensor\_evaluation.lhs, block.location\_dataset('Mach\_sensor')\*sensor\_evaluation.rhs)]

# Halo points for the sensor kernel

sensor\_halos = []

for \_ in range(block.ndim):

sensor\_halos.append([self.halo\_type, self.halo\_type])

sensor\_kernel = self.create\_kernel('Shock sensor', flatten(output\_eqns), sensor\_halos, block)

# Add the kernel

self.add\_kernel(sensor\_kernel)

self.component\_counter += 1

return kappa

def wall\_control(self):

""" Turns off the filter close to any of the walls in the problem."""

wall\_var = GridVariable('Wall')

wall\_conditions, wall\_equations = [], []

indexes = [OpenSBLIEq(GridVariable('Grid\_%d' % direction), self.block.grid\_indexes[direction]) for direction in range(self.ndim)]

wall\_equations += indexes

# Disable the shock filter at any wall boundaries

for direction in range(self.ndim):

for side in [0,1]:

wall = self.wall\_boundaries[direction][side]

if wall:

if side == 0:

wall\_conditions += [ExprCondPair(0, indexes[direction].lhs <= 5)]

else:

wall\_conditions += [ExprCondPair(0, indexes[direction].lhs >= self.block.ranges[direction][side] - 6)]

# No wall, default condition is the sensor is not turned off

wall\_conditions += [ExprCondPair(1, True)]

wall\_equations += [OpenSBLIEq(wall\_var, Piecewise(\*wall\_conditions))]

return wall\_var, wall\_equations

def filter\_application(self, solution\_vector, block):

""" Applies the non-linear filter by subtracting from the q vector after a full RK time-step."""

resid\_kernel = self.residual\_kernels[0]

modified\_equations = []

# Global parameter to control the dissipation to give extra control of the dissipation in the C code

FC = ConstantObject('shock\_filter\_control')

FC.value = 1 # Default condition has no scaling

CTD.add\_constant(FC)

# The amount of dissipation to apply, using a local flow sensor

if self.dissipation\_sensor == 'Ducros':

kappa = self.evaluate\_Ducros\_sensor(block)

elif self.dissipation\_sensor == 'Constant': # No flow sensor for the dissipation control, only a global parameter

kappa, kappa\_evaluation = 1, []

if self.Mach\_correction:

kappa\_Yee, Mach\_eqns = self.evaluate\_Yee\_Mach\_sensor(block)

kappa \*= kappa\_Yee

else:

raise NotImplementedError("Please enter a valid dissipation sensor option: 'Ducros', or 'Constant'.")

# Take the maximum of the sensor over nearby grid points

if self.dissipation\_sensor == 'Ducros':

formula = kappa

for direction in range(self.ndim):

for location in [-2, -1, 0, 1, 2]:

formula = Max(formula, increment\_dataset(kappa, direction, location))

kappa\_max = GridVariable('kappa\_max')

modified\_equations += [OpenSBLIEq(kappa\_max, formula)]

else:

kappa\_max = kappa

# Turn off the sensor at the walls

wall\_detection, wall\_equations = self.wall\_control()

modified\_equations += wall\_equations

# Apply the filter

for i, eqn in enumerate(resid\_kernel.equations):

q = self.solution\_vector[i]

modified\_equations.append(OpenSBLIEq(q, q + wall\_detection\*FC\*kappa\_max\*ConstantObject('dt')\*eqn.rhs))

resid\_kernel.equations = modified\_equations

residual\_kernel = self.create\_kernel('Non-linear filter application', resid\_kernel.equations, resid\_kernel.halo\_ranges, block)

self.component\_counter += 1

self.add\_kernel(residual\_kernel)

return

def zero\_work\_arrays(self, block):

resid\_kernel = self.residual\_kernels[0]

zeroed\_equations = []

for i in range((self.ndim+2)\*self.ndim):

zeroed\_equations += [OpenSBLIEq(block.location\_dataset('wk%d' % i), 0.0)]

zero\_kernel = self.create\_kernel('Zero the work arrays', zeroed\_equations, resid\_kernel.halo\_ranges, block)

self.component\_counter += 1

self.add\_kernel(zero\_kernel)

return

def convert\_to\_datasets(self, block, equations):

output\_equations = []

for eqn in flatten(equations):

output\_equations += [eqn.convert\_to\_datasets(block)]

return output\_equations

def main(self, scheme\_order, block):

# Counter to order the kernels

self.component\_counter = block.blocknumber\*10

# Create the equations for WENO

eqn = self.create\_weno\_equations()

# Convert the equations to datasets on this block

self.equations = self.convert\_to\_datasets(block, eqn)

# Create a WENO scheme

WS = LLFWeno(scheme\_order, formulation='JS', averaging=SimpleAverage([0, 1]), shock\_filter=True)

self.halo\_type = set()

self.halo\_type.add(WS.halotype)

# Start the discretisation and create residual arrays for the equations

self.Kernels = []

self.create\_residual\_arrays(block)

CR, solution\_vector = WS.discretise(self, block)

# Q vector

self.solution\_vector = flatten(self.time\_advance\_arrays)

# Constituent relations evaluations on the Q vector at the end of the full RK time-step

self.constituent\_relations(block)

# Zero the work arrays

# self.zero\_work\_arrays(block)

# Create the WENO reconstruction kernels

reconstruction\_kernels = []

for code\_gen\_order, ker in enumerate(self.reconstruction\_kernels):

halo\_ranges = ker.halo\_ranges

reconstruction\_kernels.append(self.create\_kernel('WENO reconstruction direction %d' % code\_gen\_order, ker.equations, halo\_ranges, block))

self.component\_counter += 1

self.add\_kernel(reconstruction\_kernels)

# Check if there any wall boundary conditions defined on the block.

self.detect\_wall\_boundaries()

# # Create the residual kernel

self.filter\_application(solution\_vector, block)

return