```
""" 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, DataSet
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
from opensbli.equation_types.metric import MetricsEquation
from opensbli.schemes.spatial.scheme import CentralHalos_defdec
from opensbli.core.grid import GridVariable as gv


class NonLinearFilterBase(object):
""" Base class for shared functionality between non-linear filter-step methods."""
def __init__(self, airfoil, block, metrics, optimize):
self.reconstruction_kernels, self.residual_kernels = [], []
self.airfoil = airfoil
self.hybrid = False
block.shock_filter = True
self.block = block
self.ndim = block.ndim
self.equation_classes = []
# Check if the problem needs a metric transformation of the equations
self.metrics = metrics
if metrics is not None:
try:
assert isinstance(block.get_metric_class, MetricsEquation)
except:
raise ValueError("Please set the metric class on the block before calling the WENO/TVD filter in the problem script.")
self.process_metrics(metrics)
self.EE = EinsteinEquation()
if metrics is not None:
optional_subs_dict = metrics.metric_subs
self.EE.optional_subs_dict = optional_subs_dict
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 detect_interface_boundaries(self):
""" The shock-filter is turned off close to block interfaces. This function detects which directions, if any, have
interface boundary conditions."""
self.interface_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], InterfaceBC) or isinstance(self.block.boundary_types[direction][side],
SharedInterfaceBC):
self.interface_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 Euler_equations_passive_scalar(self, block, scheme_type):
# Define the compresible Navier-Stokes equations in Einstein notation, depending on the metric input
scheme_type = "**{\'scheme\':\'%s\'}" % scheme_type
constants = ["Re", "Pr","gama", "Minf", "SuthT", "RefT"]
# Uniform mesh, no stretching or curvilinear terms
if self.metric_class is None:
coordinate_symbol = "x"
mass = "Eq(Der(rho,t), - Conservative(rhou_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
# Added passive scalar equation here for filter methods
ps = "Eq(Der(rhof,t), -Conservative(u_j*rhof, x_j, %s))" % scheme_type
output_equations = flatten([self.EE.expand(eq, self.ndim, coordinate_symbol, [], constants) for eq in flatten([mass, momentum,
energy, ps])])
else:
if self.passive_scalar:
raise ValueError("WARNING: Passive scalar has not been added to curvilinear equations yet.")
# Full curvilinear
if self.curvilinear:
coordinate_symbol = "xi"
optional_subs_dict = self.metric_class.metric_subs
self.EE.optional_subs_dict = optional_subs_dict
a = "Conservative(detJ * rho*U_j,xi_j,%s)" % scheme_type
mass = "Eq(Der(rho,t), - %s)" % (a)
a = "Conservative(detJ * (rhou_i*U_j + p*D_j_i), xi_j , %s)" % scheme_type
momentum = "Eq(Der(rhou_i,t) , - %s)" % (a)
a = "Conservative(detJ * (p+rhoE)*U_j,xi_j, %s)" % scheme_type
energy = "Eq(Der(rhoE,t), - %s)" % (a)

base_eqns = [mass, momentum, energy]
for i, base in enumerate(base_eqns):
base_eqns[i] = self.EE.expand(base, self.ndim, coordinate_symbol, [], constants)
if base==momentum:
for no, b in enumerate(base_eqns[i]):
base_eqns[i][no] = OpenSBLIEq(base_eqns[i][no].lhs, base_eqns[i][no].rhs)
else:
if base==energy:
base_eqns[i] = OpenSBLIEq(base_eqns[i].lhs, base_eqns[i].rhs)
# output_equations = flatten([self.EE.expand(eq, self.ndim, coordinate_symbol, [], constants) for eq in flatten([mass, momentum,
energy])])
output_equations = flatten(base_eqns)
# # Only stretching is applied
else: ### Only added non-conservative for this stretched case
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, [], constants) for eq in flatten([mass, momentum,
energy])])
```

```
output_equations = flatten([self.metric_class.apply_transformation(eqn) for eqn in (governing_eq)])
print("Using the following equations for the TVD/WENO filter.")
for eqn in output_equations:
pprint(eqn)
# exit()
return output_equations

def Euler_equations(self, block, scheme_type):
# Define the compresible Navier-Stokes equations in Einstein notation, depending on the metric input
scheme_type = "**{\'scheme\':\'%s\'}" % scheme_type
constants = ["Re", "Pr","gama", "Minf", "SuthT", "RefT"]
# Uniform mesh, no stretching or curvilinear terms
if self.metric_class is None:
coordinate_symbol = "x"
mass = "Eq(Der(rho,t), - Conservative(rhou_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, [], constants) for eq in flatten([mass, momentum,
energy])])
else:
# Full curvilinear
if self.curvilinear:
coordinate_symbol = "xi"
optional_subs_dict = self.metric_class.metric_subs
self.EE.optional_subs_dict = optional_subs_dict
a = "Conservative(detJ * rho*U_j,xi_j,%s)" % scheme_type
mass = "Eq(Der(rho,t), - %s)" % (a)
a = "Conservative(detJ * (rhou_i*U_j + p*D_j_i), xi_j , %s)" % scheme_type
momentum = "Eq(Der(rhou_i,t) , - %s)" % (a)
a = "Conservative(detJ * (p+rhoE)*U_j,xi_j, %s)" % scheme_type
energy = "Eq(Der(rhoE,t), - %s)" % (a)

base_eqns = [mass, momentum, energy]
for i, base in enumerate(base_eqns):
base_eqns[i] = self.EE.expand(base, self.ndim, coordinate_symbol, [], constants)
if base==momentum:
for no, b in enumerate(base_eqns[i]):
base_eqns[i][no] = OpenSBLIEq(base_eqns[i][no].lhs, base_eqns[i][no].rhs)
else:
if base==energy:
base_eqns[i] = OpenSBLIEq(base_eqns[i].lhs, base_eqns[i].rhs)
# output_equations = flatten([self.EE.expand(eq, self.ndim, coordinate_symbol, [], constants) for eq in flatten([mass, momentum,
energy])])
output_equations = flatten(base_eqns)
# # Only stretching is applied
else: ### Only added non-conservative for this stretched case
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, [], constants) for eq in flatten([mass, momentum,
energy])])
output_equations = flatten([self.metric_class.apply_transformation(eqn) for eqn in (governing_eq)])
# for eqn in output_equations:
# pprint(eqn)
# exit()
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
for eqn in equations:
pprint(eqn)
```

```
# exit()
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 reduction_operations(self, reduction_equations):
""" Get reduction kernels for global reductions if needed."""
reduction_halos = []
for _ in range(self.ndim):
reduction_halos.append([self.halo_type, self.halo_type])
reduction_kernel = self.create_kernel('Global wave-speed reduction evaluations', reduction_equations, reduction_halos, block)
self.component_counter += 1
self.add_kernel(reduction_kernel)
return

def constituent_relations(self, block, kappa=None):
""" 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 = []
if kappa is not None:
CR_eqns += [OpenSBLIEq(kappa, 1)]
# Ensure gama has been added to the constants to define
gamma = ConstantObject('gama')
CTD.add_constant(gamma)
# Conservative Q array entries from the current state
rho, energy = block.location_dataset('rho'), block.location_dataset('rhoE')
# Pressure and speed of sound
p, a = block.location_dataset('p'), block.location_dataset('a')
inv_rho = gv('inv_rho')
CR_eqns += [OpenSBLIEq(inv_rho, 1.0/rho)]
velocity_components = [block.location_dataset('u%d' % i ) for i in range(self.ndim)]

if block.conservative:
momentum_components = [self.solution_vector[i+1] for i in range(self.ndim)]
# Primitive components and speed of sound
CR_eqns += [OpenSBLIEq(x, y*inv_rho) for (x, y) in zip(velocity_components, momentum_components)]
# rhoE = p/(gama-1) + 0.5*(rhou**2)/rho
CR_eqns += [OpenSBLIEq(p, (gamma-1)*(energy - 0.5*sum([dset**2 for dset in momentum_components])*inv_rho))]
else:
# E = p/((gamma-1)*rho) + 0.5*u**2
CR_eqns += [OpenSBLIEq(p, rho*(gamma-1)*(energy - 0.5*sum([dset**2 for dset in velocity_components])))]
# Ideal gas, speed of sound
CR_eqns += [OpenSBLIEq(a, sqrt(gamma*p*inv_rho))]
# Wide halos
CR_halos = []
for _ in range(self.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
self.add_kernel(CR_kernel)
return

def zero_work_arrays(self, block):
""" Ensure all the temporary arrays are zeroed before calculating the filter."""
resid_kernel = self.residual_kernels[0]
zero_halos = []
for _ in range(self.ndim):
zero_halos.append([CentralHalos_defdec(), CentralHalos_defdec()])
```

```
zeroed_equations = flatten([OpenSBLIEq(dset, 0.0) for dset in self.SF.temp_wk_arrays[direction]] for direction in
range(block.ndim))
zero_kernel = self.create_kernel('Zero the work arrays', zeroed_equations, zero_halos, 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 wall_control(self, depth):
""" Turns off the filter close to any of the walls or block interfaces in the problem."""
if self.airfoil:
wall_buffer = depth
buffer = depth
else:
wall_buffer = depth
buffer = depth
wall_var = gv('Wall')
wall_conditions, wall_equations = [], []
indexes = [OpenSBLIEq(gv('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 or block interfaces
for direction in range(self.ndim):
for side in [0,1]:
wall = self.wall_boundaries[direction][side]
interface = self.interface_boundaries[direction][side]
if wall:
if side == 0:
wall_conditions += [ExprCondPair(0, indexes[direction].lhs <= wall_buffer)]
else:
wall_conditions += [ExprCondPair(0, indexes[direction].lhs >= self.block.ranges[direction][side] - (wall_buffer+1))]
if interface:
if side == 0:
wall_conditions += [ExprCondPair(0, indexes[direction].lhs <= buffer)]
else:
wall_conditions += [ExprCondPair(0, indexes[direction].lhs >= self.block.ranges[direction][side] - (buffer+1))]
# No wall or interface, 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 update_periodic_boundary(self, block, halos):
""" Apply periodic boundary conditions again if required before applying the WENO filter."""
print("Applying periodic boundary for WENO/TVD")
bc_kernels = []
for direction in range(block.ndim):
for side in [0, 1]:
if isinstance(block.boundary_types[direction][side], PeriodicBC):
bc_kernels.append(PeriodicBC(direction, side, halos=halos, corners=False).apply(self.solution_vector, block))
# Swap periodic boundary before applying WENO filter method
self.equation_classes[0].Kernels = bc_kernels + self.equation_classes[0].Kernels
return

class WENOFilter(NonSimulationEquations, NonLinearFilterBase):
""" 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, flux_type='LLF', airfoil=False, formulation='Z', optimize=False):
print("Using non-linear WENO filtering on block {:}.".format(block.blocknumber))
```

```
# self.passive_scalar = passive_scalar
# Get the shared functionality between TVD/WENO non-linear filters
NonLinearFilterBase.__init__(self, airfoil, block, metrics, optimize=optimize)
self.flux_type = flux_type
self.formulation = formulation
self.optimize = optimize
# Main class to generate the filter
self.main(order, block)
return

def main(self, scheme_order, block):
""" Main calling function to generate the kernels for the WENO filter."""
# Counter to order the kernels. Put the WENO filtering kernels at the very end of the time loop
self.component_counter = 1000 + block.blocknumber*1000
# Create the equations for WENO
# if self.passive_scalar:
# eqn = self.Euler_equations_passive_scalar(block, "Weno")
# else:
eqn = self.Euler_equations(block, "Weno")
# Convert the equations to datasets on this block
self.equations = self.convert_to_datasets(block, eqn)
# Create a WENO scheme
if self.flux_type == 'LLF' or self.flux_type == 'GLF':
self.SF = LFWeno(scheme_order, formulation=self.formulation, flux_type=self.flux_type, averaging=RoeAverage([0, 1]),
shock_filter=True, conservative=block.conservative)
elif self.flux_type == 'HLLC' or self.flux_type == 'HLLC-LM':
self.SF = HLLCWeno(scheme_order, formulation=self.formulation, flux_type=self.flux_type, averaging=RoeAverage([0, 1]),
shock_filter=True, conservative=block.conservative)
else:
raise ValueError("Please input a valid flux splitting type: LLF, GLF, HLLC, HLLC-LM.")
self.halo_type = set()
self.halo_type.add(self.SF.halotype)
# Start the discretisation and create residual arrays for the equations
self.Kernels = []
self.create_residual_arrays(block)
CR, solution_vector, reductions = self.SF.discretise(self, block)
# Q vector
self.solution_vector = flatten(self.time_advance_arrays)
# Swap over the WENO stencil if periodic boundaries
# bc_kernels = self.update_periodic_boundary(block, self.SF.halotype)
# Shock sensor evaluation to find which points to evaluate the WENO scheme on
if self.optimize:
self.kappa = self.evaluate_shock_sensor(block)
else:
self.kappa = 1
# Reductions if needed
if len(reductions) > 0:
self.reduction_operations(reductions)
# Zero the work arrays
self.zero_work_arrays(block)
# Create the WENO reconstruction kernels
reconstruction_kernels = []
for direction, ker in enumerate(self.reconstruction_kernels):
# Hybrid mode
if self.optimize:
ker = self.hybrid_condition(ker, block, direction)
halo_ranges = ker.halo_ranges
reconstruction_kernels.append(self.create_kernel('WENO reconstruction direction %d' % direction, ker.equations, halo_ranges,
block))
self.component_counter += 1

self.add_kernel(reconstruction_kernels)
# Check if there any wall boundary conditions or interfaces defined on the block.
self.detect_wall_boundaries()
self.detect_interface_boundaries()
# # Create the residual kernel
self.filter_application(block)
```

```
return

# def evaluate_Yee_Mach_sensor(self, velocity_components, pressure, speed_of_sound, 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)."""
# # Evaluate the local Mach number
# M = symbols('M', **{'cls' : GridVariable})
# Mach_equations = [OpenSBLIEq(M, sqrt(sum(dset**2 for dset in velocity_components))/speed_of_sound)]
# # Evaluation of the kappa parameter to control the amount of dissipaton
# Mach_equations += [OpenSBLIEq(block.location_dataset('Mach_sensor'), Min(0.5*M**2 * sqrt(4+(1-M**2)**2) / (1+M**2), 1.0))]
# return Mach_equations

def evaluate_shock_sensor(self, block):
# Add a shock sensor for the WENO filter
SS = ShockSensor()
if block.ndim > 1: # no shock sensor defined for ndim=1 currently
# Ducros dilatation part
sensor_evaluations, kappa = SS.ducros_equations(block, "x", metrics=self.metric_class, name='kappa')
else:
raise ValueError("No hybrid method for ndim=1.")
# Update the CRs needed, and set the shock sensor to 1 on the outer boundaries
self.constituent_relations(block, kappa)
# Halo points for the sensor kernel
sensor_halos = []
for _ in range(self.ndim):
sensor_halos.append([self.halo_type, self.halo_type])
sensor_kernel = self.create_kernel('Shock sensor', flatten(sensor_evaluations), sensor_halos, block)
self.add_kernel(sensor_kernel)
self.component_counter += 1
return kappa

def filter_application(self, block):
""" Applies the non-linear filter by subtracting from the q vector after a full RK time-step."""
resid_kernel = self.residual_kernels[0]
nvars = len(self.solution_vector)
# Previous in conservative form
rho = self.solution_vector[0]
modified_equations = []
# Turn off the sensor at the walls
wall_detection, wall_equations = self.wall_control(depth=5)
modified_equations += wall_equations
# Find the maximum of nearby points
kappa_fact = self.kappa
if isinstance(kappa_fact, DataSet):
check = self.kappa
for direction in range(self.ndim):
for loc in [-1, -1, 0, 1, 2]:
check = Max(check, increment_dataset(self.kappa, direction, loc)) # for direction in range(self.ndim):
kappa_fact = block.location_dataset('WENO_filter')
# Selection of which kappa points to apply the filter to
DS = ConstantObject('Ducros_select')
DS.value = 0.05
CTD.add_constant(DS)
check = check >= DS
cond1 = ExprCondPair(1, check)
cond2 = ExprCondPair(0.0, True)
modified_equations += [OpenSBLIEq(kappa_fact, Piecewise(*[cond1, cond2]))]
else:
kappa_fact = 1
# detJ if needed, need to improve these scaling
if self.curvilinear and self.airfoil:
if self.ndim == 3:
modified_equations += [OpenSBLIEq(gv('inv_detJ'), 1 / (Abs(block.location_dataset('detJ')) / self.block.deltas[2])) ] ## Assumes
span-periodic for now, for scaling
else:
modified_equations += [OpenSBLIEq(gv('inv_detJ'), 1 / Abs(block.location_dataset('detJ')))]
detJ_term = gv('inv_detJ')
```

```
else:
detJ_term = 1

# Apply the filter
for i, eqn in enumerate(resid_kernel.equations):
# Turn shock-capturing off only for the reconstruction normal to the wall, currently assume direction = 1 for the wall. Fix later
weno_eqn = eqn.rhs.xreplace({ConstantObject('inv_rfact%d_block%d' % (1, block.blocknumber)) : ConstantObject('inv_rfact
%d_block%d' % (1, block.blocknumber))*wall_detection})
rhs = kappa_fact*ConstantObject('dt')*weno_eqn * detJ_term
modified_equations.append(OpenSBLIEq(self.solution_vector[i], self.solution_vector[i] + rhs))

# Finish creating the kernel
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 hybrid_condition(self, kernel, block, direction):
""" Checks the Ducros sensor, if it is a shock we perform the WENO reconstruction, else do nothing."""
from sympy import And, Or
input_equations = flatten(kernel.equations)
kernel.equations = []

if self.optimize:
""" Only evaluate the WENO kernels at certain points, based on the shock sensor result. Improves performance."""
DC = ConstantObject('Ducros_check')
DC.value = 0.05
CTD.add_constant(DC)
locations = [-3, -2, -1, 1, 2]
term = self.kappa
# for dire in range(block.ndim):
dire = direction # Only check 1D kappa
for loc in locations:
term = Max(term, increment_dataset(self.kappa, dire, loc))
check = term > DC
cond1 = ExprCondPair(input_equations, check)
zeroed_equations = flatten([OpenSBLIEq(dset, 0.0) for dset in self.SF.temp_wk_arrays[direction]])
cond2 = ExprCondPair(zeroed_equations, True)
kernel.add_equation([GroupedPiecewise(cond1, cond2)])
else:
kernel.add_equation(input_equations)
return kernel
```