OpenSBLI user manual and developer guide

David J. Lusher, Satya P. Jammy, Neil D. Sandham

June 13, 2021

OpenSBLI user manual, major revision history:

- Update for CPC paper publication: DJL - June 2021.
- Added non-linear WENO filter methods: DJL - September 2020.
- Added Python 3 compatibility: DJL - May 2020.
- Initial version: DJL - March 2020.

Introduction

Quick start: General work flow

Below are the five main steps that must be performed to generate a code in OpenSBLI and run it within OPS. Please see section 1 for detailed instructions on how to install OpenSBLI, OPS, and run a simulation.

1. Run `python problem_file.py` for one of the problem files located in the apps folder in OpenSBLI to generate a C code.
2. Copy the C code `(*.cpp, *.h)` files to the machine where you have OPS installed to run the simulations.
3. Use the OPS source-to-source translator on the `opensbli.cpp` file to generate parallel versions of the code (OpenMP, OpenACC, MPI, CUDA, OpenCL, ...)
4. Copy a Makefile into the working directory and compile the code: `make opensbli_mpi`.
5. Run the executable: `mpirun -np 4 ./opensbli_mpi`.

Important note: The purpose of OpenSBLI is to generate a complete CFD simulation code in the C language from a set of user-defined equations and options. The execution of the simulation code to obtain results is performed with the OPS library. The OPS library must be built on whichever machine you wish to run the simulation on. Codes can be generated locally on a laptop in OpenSBLI and then copied onto a high-performance computing cluster to run with OPS. In many cases once a C code has been generated in OpenSBLI, changes can be made to the C code directly without needing to generate another code through the OpenSBLI Python interface.

Contents

1 Installation guide **5**
- 1.1 Installation of libraries on an Ubuntu local machine 6
- 1.2 OPS installation 6
- 1.3 OpenSBLI installation and code generation 7
- 1.4 Compiling and running the code 8
- 1.5 Things to note for HPC clusters: 8

2 List of source code files **10**
- 2.1 Directory structure 10
- 2.2 Description of the source code files 12

3 Example problem script for creating a simulation code **16**
- 3.1 Defining the governing equations 16
- 3.2 Defining constituent relations 17
- 3.3 Parsing and expanding the equations into symbolic expressions 17
- 3.4 Creating a simulation block and generating coordinate transformations . . . 18
- 3.5 Transformation of the simulation equations 18
- 3.6 Numerical scheme selection 18
- 3.7 Creating boundary conditions 19
- 3.8 Initialisation of the domain and grid 21
- 3.9 Simulation input/output data 22
- 3.10 Creating the simulation code and writing to C 23
- 3.11 Defining numerical values of constants for the simulation 23

4 Makefiles **25**

5 Procedure for restarting simulations **26**

6 Components of the generated OPS C code **27**
- 6.1 Preamble and constant declarations 27
- 6.2 Stencil declarations for relative data access 28
- 6.3 Parallel loops: ops_par_loop 29
- 6.4 Computational kernels 29
- 6.5 Iteration and sub-stage loops 30
- 6.6 HDF5 file output 31

7 Current scheme and boundary condition options **32**
- 7.1 Numerical scheme options 32
  - 7.1.1 Central schemes 32
  - 7.1.2 Shock-capturing WENO/TENO schemes 32

7.1.3 Non-linear WENO filter methods for shock-capturing 34
7.1.4 Time advancement 36
7.2 Boundary conditions 36
7.3 Miscellaneous filters 37
7.3.1 Explicit binomial filter 37
7.3.2 Selective Frequency Damping (SFD) filter 38

8 Example governing equations **39**
8.1 Central schemes with skew-symmetric formulation 39
8.2 Shock capturing schemes in conservative form 39

9 Monitoring quantities during the simulation **41**

10 Useful SymPy functions for OpenSBLI development **42**
10.1 Pretty printing of equations: pprint 42
10.2 Creating symbolic objects: symbols 42
10.3 Off-setting an indexed object 43
10.4 Substitution of symbolic quantities 43
10.5 Left and right hand sides of equations 44
10.6 De-nesting multiple iterables 44
10.7 Accessing attributes in Python objects: __dict__ 44
10.8 Using built-in SymPy functions (sin, cos, exp, ...) 45
10.9 Conditional expressions (if-else branching) 46
10.10String representation: srepr 46
10.11Breaking down an expression into its components: atoms/args 47
10.12Isolating terms of a certain type: isinstance 48
10.13Using GridVariables as temporary evaluations 48
10.14Simplifying long expressions and reducing operation counts 48

11 HPC set-up on CSD3 NVIDIA P100 GPU machine **49**
11.1 Example bashrc settings 49

12 Performance profiling of OPS **50**

13 Opening OpenSBLI HDF5 output files in post-processing software **50**
13.1 Python 50
13.2 MATLAB 50
13.3 Paraview 51
13.4 Tecplot 52

1 Installation guide

OpenSBLI requires a Python 2.7 install for code generation and the Oxford Parallel Structured software (OPS) library to build executables. The guide is for Linux platforms only, it hasn't been tested on OSX/Windows. Section 1 outlines the libraries that must be installed to use OpenSBLI + OPS, section 2 explains the build process of OPS, section 3 covers how to generate a code within OpenSBLI and section 4 gives the steps to compile and run the code. Section 5 has some additional information that may be applicable if running on an HPC cluster.

The installation of libraries will depend on whether you are using a local machine or an HPC cluster. If you are intending to use OpenSBLI on an HPC cluster you will need to load suitable system modules and point OPS to them via the environment variables outlined in section 1.2. MPI and HDF5 libraries must have been compiled with the same compiler or you will face errors during compilation of OPS executables. If you are not sure where the modules are located, please contact your system administrator. **Important note: SymPy version 1.1 is required, newer versions are currently not supported.** The requirements are:

OpenSBLI requirements (Python 2)

```
SymPy version 1.1 : Symbolic Python libary used by OpenSBLI
Matplotlib : Python plotting library
h5py : Python library to read in HDF5 output files for plotting
NumPy : Numerical Python library used in some plot scripts
SciPy: Scientific Python library used in some OpenSBLI classes
```

OPS requirements

```
git : Version control software used to clone the repository
A C compiler : (gcc, icc, Cray, ..)
An MPI distribution : (OpenMPI, MPICH, Intel MPI, ..)
HDF5 libraries : Used for I/O within OPS
CUDA/OpenCL libraries : Required only if you intend to run on GPUs
```

Important note: Codes can be generated with OpenSBLI on a local machine (laptop) and copied onto another machine to be translated, compiled and run with OPS. While you can install both OpenSBLI and OPS on an HPC cluster, it is not necessary to have both on the same machine. OPS however must be installed (and built) on the machine you wish to run simulations on.

1.1 Installation of libraries on an Ubuntu local machine

****Ignore this section if installing OpenSBLI on an HPC cluster****
This section was tested on a clean Ubuntu 18.04 LTS installation, similar packages can be found for other Linux distributions. Python packages are installed with pip, if pip is not already present on your system it can be installed with e.g. `sudo apt-get install python-pip`, on Ubuntu/Debian based systems. Alternatively, software distributions like Anaconda can be used to install the Python libraries. The guide is for installation on a local machine with the GNU C compiler (gcc) and OpenMPI.

1. Install OpenMPI and HDF5 libraries, git and pip/tk

   ```
   sudo apt-get install python-pip python-tk
   sudo apt-get install libhdf5-openmpi-dev
   sudo apt-get install git
   ```

2. Install the required Python packages for this user. Note that SymPy 1.1 is required, more recent versions will cause issues:

   ```
   pip install --user scipy numpy h5py matplotlib
   pip install --user sympy==1.1
   ```

1.2 OPS installation

In this section the installation of the OPS library is explained. **The paths that you put in step 2 will change depending on whether you are on a local machine or an HPC cluster.**

1. Clone the OPS library:
   `git clone https://github.com/OP-DSL/OPS.git`

2. Export OPS environment variables:
   Edit your `~/.bashrc` to include the lines:

   Local Ubuntu machine:

   ```
   export OPS_INSTALL_PATH=/home/<username>/OPS/ops/
   export OPS_COMPILER=gnu
   export OPS_TRANSLATOR=/home/<username>/OPS/ops_translator/c/
   export MPI_INSTALL_PATH=/usr/
   export HDF5_INSTALL_PATH=/usr/lib/x86_64-linux-gnu/hdf5/openmpi/
   export CUDA_INSTALL_PATH=/path/to/cuda-install/
   export USE_HDF5=1
   ```

HPC cluster: (fill these in)

```
export OPS_INSTALL_PATH=/home/<username>/OPS/ops/
export OPS_COMPILER=gnu/cray/intel (pick one)
export OPS_TRANSLATOR=/home/<username>/OPS/ops_translator/c/
export MPI_INSTALL_PATH=/path/to/mpi-install/
export HDF5_INSTALL_PATH=/path/to/hdf5-install/
export CUDA_INSTALL_PATH=/path/to/cuda-install/
export USE_HDF5=1
```

To make these changes active in the current shell type: `source ~/.bashrc`

3. Build the OPS libraries:

```
Change directory to ~/OPS/ops/c/
make seq mpi hdf5_seq hdf5_mpi
```

If you are using CUDA/OpenCL for GPUs you will also have to build:

```
make cuda mpi_cuda opencl mpi_opencl
```

You may see some warnings that can be ignored, the ./lib/ folder should now contain some OPS libraries. If you encounter errors at this point it will be due to improper configuration of the modules/environment variables.

1.3 OpenSBLI installation and code generation

1. Clone the latest OpenSBLI with your username and switch branch:

```
git clone https://github.com/opensbli/opensbli.git
```

2. Export the OpenSBLI folder to your PYTHONPATH in `~/.bashrc`:

```
export PYTHONPATH=$PYTHONPATH:/home/<username>/opensbli/
source ~/.bashrc
```

3. Generate a code in OpenSBLI (e.g. Taylor-Green vortex):

```
cd apps/taylor_green_vortex/
python taylor_green_vortex.py
```

You should now have four new files:
`opensbli.cpp, opensbliblock00_kernels.h, defdec_data_set.h, bc_exchanges.h` and some LaTeX files.

At this point the files may be copied onto another machine (if desired) where you have OPS installed. The OPS translation and compilation steps are described in the next section.

1.4 Compiling and running the code

1. Translate the code using OPS to generate parallel versions:

   ```
   python $OPS_TRANSLATOR/ops.py opensbli.cpp
   ```

   You should now have folders for CUDA, MPI and so on, plus an `opensbli_ops.cpp` file. You need to apply the translation step each time you make changes to the opensbli.cpp code generated by OpenSBLI. Ignore the clang-format warning, this is just an optional code indentation formatter and has no effect on the simulation.

2. Copy a Makefile into the directory and build the executable for the desired architecture (MPI, CUDA, OpenMP etc):

   ```
   cp ../Makefile ./
   make opensbli_mpi
   ```

3. Run the executable for number of processes 'np':

   ```
   mpirun -np 4 ./opensbli_mpi
   ```

   The simulation will now run, when finished the output data is written to an `opensbli_output.h5` HDF5 file for post-processing. If you wish to run the simulation again remove the HDF5 file from the directory before repeating the above steps as existing files won't be overwritten.

1.5 Things to note for HPC clusters:

1. For different compilers the `OPS_COMPILER` variable set in section 1.2 must be changed and you may need to modify some of the Makefiles. The easiest way is usually to export cc and CC to where your compiler is located, but you may need to explicitly edit the Makefiles to use icc/icpc for example. In the current OPS (02/2019) the Makefiles are all located in /OPS/makefiles/

2. If there is no suitable HDF5 available on the system you will have to build it yourself:

   ```
   Obtain the source: https://support.hdfgroup.org/HDF5/release/obtainsrc.html
   Extract the file, change to the directory and configure the build.
   For building parallel HDF5:
   ./configure --enable-parallel cc=/path/to/mpicc CC=/path/to/mpicc
   make
   make install
   ```

3. Depending on the machine you may need to add your HDF5/CUDA library locations to your `LD_LIBRARY_PATH` environment variable.

4. Job submission scripts will often need module load commands in them as the compute nodes are separate to login nodes. In addition `LD_LIBRARY_PATH` may need to be exported in the submission script with the HDF5/CUDA library locations.

5. If using gcc you may need to add:

   ```
   -fpermissive to CCFLAGS/CXXFLAGS in OPS/makefiles/Makefile.gnu
   ```

2 List of source code files

2.1 Directory structure

The directory tree below shows the structure of the OpenSBLI source code files. Folders are highlighted in blue and files in black.

```
OpenSBLI base directory
├─ apps
│  ├─ channel_flow
│  ├─ euler_wave
│  ├─ inviscid_shock_reflection
│  ├─ katzer_SBLI
│  ├─ kelvin_helmholtz
│  ├─ shu_osher
│  ├─ taylor_green_vortex
│  ├─ viscous_shock_tube
│  └─ wave
├─ docs
└─ opensbli
   ├─ code_generation
   │  └─ algorithm
   │     └─ algorithm.py
   ├─ latex.py
   ├─ opsc.py
   └─ core
      └─ bcs.py
         [cont.]
```

continued

```
    [cont.]
│
├── block.py
│
├── datatypes.py
│
├── grid.py
│
├── io.py
│
├── kernel.py
│
├── opensblifunctions.py
│
├── opensbliobjects.py
│
└── parsing.py

equation_types
│
├── metric.py
│
└── opensbliequations.py

initialisation
│
└── gridbasedinit.py

optimisations
│
└── optimisations.py

physical_models
│
├── euler_eigensystem.py
│
└── ns_physics.py

post_process
│
└── post_process_eq.py
          [cont.]
```

continued

```
        [cont.]
└─ removehalos.py

schemes
├─ spatial
│   ├─ averaging.py
│   ├─ hybrid.py
│   ├─ scheme.py
│   ├─ shock_capturing.py
│   ├─ shock_sensors.py
│   ├─ teno.py
│   └─ weno.py
└─ temporal
    ├─ rk_LS.py
    └─ rk_sbli.py

utilities
├─ helperfunctions.py
├─ katzer_init.py
├─ numerical_functions.py
├─ oblique_shock.py
└─ user_defined_kernels.py
```

2.2 Description of the source code files

This is a brief description of each of the source code files in alphabetical order.

1. **algorithm.py:** Placement and ordering of the different solution components in the output code. Defines the iteration and sub-stage loops and their respective compo-

nents.

2. **averaging.py:** Simple (mean) and Roe averaging routines for the characteristic decomposition. Used for the shock-capturing schemes.

3. **bcs.py:** All of the boundary conditions defined in OpenSBLI. Also contains the one-sided boundary derivative schemes.

4. **block.py:** The simulation block contains grid information and links all of the main OpenSBLI components together.

5. **datatypes.py:** Floating point precision definitions.

6. **euler_eigensystem.py:** Definitions of the transformation matrices used in the characteristic decomposition of the Euler equations. Used for improving the robustness of the shock-capturing schemes.

7. **grid.py:** Defines the parameters related to the grid (sizes, ranges) and the declaration of global work arrays.

8. **gridbasedinit.py:** The class used to initialise the solution at $t = 0$. Equations given to the initialisation class are calculated once at the start of the simulation.

9. **helperfunctions.py:** Miscellaneous functions frequently used throughout the internal structure of OpenSBLI. It also contains routines for adding the problem-specific HDF5 metadata required by OPS for reading/writing files.

10. **hybrid.py:** Routines for hybrid central-WENO schemes via a shock sensor.

11. **io.py:** The main input/output class controlling reading and writing of simulation data within OpenSBLI.

12. **katzer_init.py:** Numerical similarity solution of the compressible boundary-layer equations. Creates laminar boundary-layer profiles for wall-bounded domain initialisation.

13. **kernel.py:** The main class for OpenSBLI kernels which perform a computation for a given kernel range in parallel.

14. **latex.py:** Functionality to convert symbolic equations in OpenSBLI to TeX format. The output can be compiled with pdflatex to produce a pdf file.

15. **metric.py:** Classes to perform the metric transformation to generalised curvilinear coordinates.

16. **numerical_functions.py:** Numerical spline interpolation routines used for the similarity solution. These functions are for creating initial profiles in Python and are not part of the output simulation.

17. **ns_physics.py:** A physics object containing frequently used definitions related to the compressible Navier-Stokes equations.

18. **oblique_shock.py:** Oblique shock jump relation calculator.

19. **opensbliequations.py:** The main file for defining equations in OpenSBLI. Contains the routines for the SimulationEquations and ConstituentRelation classes.

20. **opensblifunctions.py:** Definitions of common derivative objects (Central, WENO, TENO) and the EinsteinStructure for indexing symbolic quantities in OpenSBLI.

21. **opensbliobjects.py:** Definitions of various symbolic objects within OpenSBLI that can be used to build up symbolic equations.

22. **opsc.py:** This controls the conversion of symbolic objects to C code compliant with the OPS domain-specific language.

23. **optimsations.py:** Optimisations for reduced work array usage.

24. **parsing.py:** Routines to parse equations written as Python strings into equations comprised of symbolic objects.

25. **post_process_eq.py:** A class to create a post processing OPS code.

26. **removehalos.py:** Functionality to strip halo points from OpenSBLI output HDF5 files.

27. **rk_LS.py:** Low-storage explicit Runge-Kutta schemes in the two-register Williamson formulation. Contains schemes for 3rd, 4th order and 3rd order with strong-stability-preservation (SSP).

28. **rk_sbli.py:** Low-storage explicit 3rd order Runge-Kutta scheme in the form used in the old SBLI Fortran code.

29. **scheme.py:** Contains the central differencing scheme and coefficient generator for the finite difference stencils.

30. **shock_capturing.py:** All of the common functionality for the characteristic decomposition shared between the various WENO/TENO shock-capturing schemes.

31. **shock_sensors.py:** The modified Ducros shock sensor.

32. **teno.py:** Routines for the Targeted Essentially Non-Oscillatory (TENO) shock-capturing schemes.

33. **user_defined_kernels.py:** Functionality to generate one-off OpenSBLI kernels. Useful for non-standard processing or filtering operations that the user may want to add to their simulation.

34. **weno.py:** Routines for the Weighted Essentially Non-Oscillatory (WENO) shock-capturing schemes. Including the WENO-JS and WENO-Z formulations.

3 Example problem script for creating a simulation code

In this section the components of an example problem script are shown for guidance of how to set up a problem in OpenSBLI. The selected problem is the laminar two-dimensional shock-wave/boundary-layer interaction Katzer case (Katzer (1989), JFM 206, pp. 477-496).

3.1 Defining the governing equations

```
#!/usr/bin/env python
from opensbli import *
import copy
from opensbli.utilities.katzer_init import Initialise_Katzer
from opensbli.utilities.helperfunctions import substitute_simulation_parameters
```

The first step imports all of the classes from the OpenSBLI source code and some individual components used for the Katzer SBLI case.

```
ndim = 2
sc1 = "**{\'scheme\':\'Teno\'}"
# Define the compresible Navier-Stokes equations in Einstein notation.
mass = "Eq(Der(rho,t), - Conservative(rhou_j,x_j,%s))" % sc1
momentum = "Eq(Der(rhou_i,t) , -Conservative(rhou_i*u_j + KD(_i,_j)*p,x_j , %s)
    + Der(tau_i_j,x_j) )" % sc1
energy = "Eq(Der(rhoE,t), - Conservative((p+rhoE)*u_j,x_j, %s) - Der(q_j,x_j) +
    Der(u_i*tau_i_j ,x_j) )" % sc1
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, (-mu/((gama-1)*Minf*Minf*Pr*Re))*Der(T,x_j))"
# Substitutions
substitutions = [stress_tensor, heat_flux]
constants = ["Re", "Pr", "gama", "Minf", "SuthT", "RefT"]
```

The user selects the number of dimensions and specifies the conservative derivatives should be discretized by a Targeted Essentially Non-Oscillatory (TENO) scheme by passing the scheme name as an argument into the equation strings. The continuity, momentum and energy equations are defined from the compressible Navier-Stokes equations. To improve readability the stress tensor $\tau_{i,j}$ and heat flux terms q_j are defined separately to be substituted into the momentum and energy equations. The equations are defined as strings to be parsed into symbolic quantities by OpenSBLI. The SymPy `Eq` class is used to define the equations, with the temporal and spatial derivatives separated into the left and right hand side of the equation respectively. The final line defines a list of constant quantities required by the simulation.

3.2 Defining constituent relations

```
# Define coordinate direction symbol (x) this will be x_i, x_j, x_k
coordinate_symbol = "x"
# Formulas for the variables used in the equations
velocity = "Eq(u_i, rhou_i/rho)"
pressure = "Eq(p, (gama-1)*(rhoE - rho*(1/2)*(KD(_i,_j)*u_i*u_j)))"
speed_of_sound = "Eq(a, (gama*p/rho)**0.5)"
temperature = "Eq(T, p*gama*Minf*Minf/(rho))"
viscosity = "Eq(mu, (T**(1.5)*(1.0+SuthT/RefT)/(T+SuthT/RefT)))"
```

A symbolic letter is selected to perform the spatial expansion over the indices in the governing equations. The next five lines are definitions to be used in the `ConstituentRelations` class. These formulae tell OpenSBLI how to evaluate quantities present in the governing equations that have not yet been defined. In this example the primitive variables of pressure, temperature, and directional velocity components are defined as functions of the conservative variables in the governing equations. As this problem is set up to use shock capturing schemes the speed of sound a must also be defined. For this supersonic test case the viscosity $\mu(T)$ is a function of temperature using Sutherland's law. If constant viscosity was required instead this equation could be omitted and mu could be defined as a constant as above.

3.3 Parsing and expanding the equations into symbolic expressions

```
# Instatiate equation classes
eq = EinsteinEquation()
base_eqns = [mass, momentum, energy]
constituent_eqns = [velocity, pressure, speed_of_sound, temperature, viscosity]
# Expand the base equations
for i, base in enumerate(base_eqns):
    base_eqns[i] = eq.expand(base, ndim, coordinate_symbol, substitutions,
        constants)
# Expand the constituent relations
for i, CR in enumerate(constituent_eqns):
    constituent_eqns[i] = eq.expand(CR, ndim, coordinate_symbol, substitutions,
        constants)
```

Having defined the governing equations and constituent relations, the next task is to use OpenSBLI to parse and expand these equations over the number of specified dimensions. The `EinsteinEquation` class is instantiated to perform expansions, and the equations defined previously are added to lists so they can be expanded iteratively in a Python loop. The expansion routine takes the input equation, number of dimensions, coordinate symbol and any substitutions and constants defined. The output is a symbolic version of each of the governing equations. The same approach is used for the five constituent relations defined by the user.

3.4 Creating a simulation block and generating coordinate transformations

```
block = SimulationBlock(ndim, block_number=0)
# Create metrics before the scheme selection
metriceq = MetricsEquation()
metriceq.generate_transformations(ndim, coordinate_symbol, [(False, False),
    (True, False)], 2)
```

In this step the simulation block is created based on the number of dimensions. For this wall bounded flow we apply grid stretching normal to the wall to improve the grid resolution in the boundary layer. To do this the `MetricsEquation` class is created and then transformation matrices are generated by specifying which directions of the problem require a coordinate transformation. The tuples `(False, False)` refer to whether grid stretching and curvilinear coordinates are required for a given direction. In this example curvilinear coordinates are turned off and only grid stretching is requested in the second spatial direction.

3.5 Transformation of the simulation equations

```
# Create SimulationEquations and Constituent relations, add the expanded
    equations
simulation_eq = SimulationEquations()
constituent = ConstituentRelations()
for eqn in base_eqns:
    simulation_eq.add_equations(eqn)
for eqn in constituent_eqns:
    constituent.add_equations(eqn)
# Grid is stretched normal to the wall
simulation_eq.apply_metrics(metriceq)
```

To store the governing equations a `SimulationEquations` class is instantiated and the previously expanded equations are added to it with the `add_equation` routine. The same procedure is applied to store the expanded constituent relation equations. In the final line the previously created metric class is used to transform the derivatives in the simulation equations.

3.6 Numerical scheme selection

```
# Adaptive TENO with modified Ducros sensor
SS = ShockSensor()
shock_sensor, sensor_array = SS.ducros_equations(block, coordinate_symbol,
    metriceq)
# Add shock Ducros sensor to constituent relations
constituent.add_equations(shock_sensor)
store_sensor = True
teno_order = 5
Avg = RoeAverage([0, 1])
LLF = LLFTeno(teno_order, formulation='adaptive', averaging=Avg,
    sensor=sensor_array, store_sensor=True)
schemes = {}
schemes[LLF.name] = LLF
```

At this point we specify the numerical schemes to be used to discretize the governing equations. For this example we are using the adaptive TENO scheme which requires a shock sensor to be provided. The shock sensor class is instantiated and the Ducros sensor is selected. As the Ducros sensor contains derivative quantities the metric class must be provided to perform the necessary coordinate transformations. The equation to evaluate the shock sensor is then added to the constituent relations to be computed at the start of each sub-stage in the time loop. A 5th order scheme is selected with the Roe averaging option for the characteristic decomposition. The `LLFTeno` class is selected to perform local Lax-Friedrichs flux splitting with a TENO scheme. An empty dictionary is created to hold the schemes, to which the LLF class is added. Note that when using the TENO schemes a value of $\epsilon = 1 \times 10^{-15}$ should be specified in the simulation constants list.

```
cent = Central(4)
schemes[cent.name] = cent
rk = RungeKuttaLS(3, formulation='SSP')
schemes[rk.name] = rk
block.set_discretisation_schemes(schemes)
```

Having selected the scheme to discretize the convective terms NS equations, we must now select a scheme for the viscous and heat-flux terms. This is done by creating a Central differencing class of order 4 and adding it to the scheme dictionary. A 3rd order low-storage SSP Runge-Kutta scheme is then selected to perform the explicit time integration. The final step is to set the chosen schemes on the simulation block for later use.

3.7 Creating boundary conditions

For a 2D problem we have to specify four boundary conditions. The convention we use for each boundary condition is to apply the coordinate direction and then 0 or 1 for the two possible sides in that direction. For this 2D shock-wave/boundary-layer interaction the possible combinations are $[0, 0]$ and $[0, 1]$ for the inlet and outlet in the x_0 direction, plus $[1, 0]$ and $[1, 1]$ for the no-slip wall and upper shock jump conditions in x_1.

```python
boundaries = [[0, 0] for t in range(ndim)]
# Left pressure extrapolation at x= 0, inlet conditions
direction = 0
side = 0
boundaries[direction][side] = InletPressureExtrapolateBC(direction, side)
# Right extrapolation at outlet
direction = 0
side = 1
boundaries[direction][side] = ExtrapolationBC(direction, side, order=0)
```

The first step creates a list of lists to hold the four boundary conditions. The inlet $[0, 0]$ is then selected as a pressure extrapolation boundary condition that only has the direction and side as input arguments. A simple zero order spatial extrapolation condition is selected for the outlet boundary.

```python
local_dict = {"block": block, "GridVariable": GridVariable, "DataObject":
    DataObject}
# Bottom no-slip isothermal wall
direction = 1
side = 0
wall_const = ["Minf", "Twall"]
for con in wall_const:
    local_dict[con] = ConstantObject(con)
# Isothermal wall condition
rhoE_wall = parse_expr("Eq(DataObject(rhoE),
    DataObject(rho)*Twall/(gama*(gama-1.0)*Minf**2.0))", local_dict=local_dict)
wall_eqns = [rhoE_wall]
boundaries[direction][side] = IsothermalWallBC(direction, 0, wall_eqns)
```

At the bottom of the domain an isothermal no-slip wall is selected. A local dictionary with common OpenSBLI objects is created as the isothermal wall requires an equation for how the user wants to evaluate energy for this boundary condition. Two constants relating to the freestream Mach number and a constant wall temperature are created using the `ConstantObject` class. In this example the parser is used directly to parse the energy equation that has been defined by the user. The last step creates the isothermal wall boundary condition and passes the energy equation as the final function argument.

```python
# Top dirichlet shock generator condition
direction = 1
side = 1
x_loc = parse_expr("Eq(GridVariable(x0),
    block.deltas[0]*block.grid_indexes[0])", local_dict=local_dict)
```

```
rho = parse_expr("Eq(DataObject(rho), Piecewise((1.129734572, (x0)>40.0),
    (1.00000596004, True)))", local_dict=local_dict)
rhou0 = parse_expr("Eq(DataObject(rhou0), Piecewise((1.0921171, (x0)>40.0),
    (1.00000268202, True)))", local_dict=local_dict)
rhou1 = parse_expr("Eq(DataObject(rhou1), Piecewise((-0.058866065, (x0)>40.0),
    (0.00565001630205, True)))", local_dict=local_dict)
rhoE = parse_expr("Eq(DataObject(rhoE), Piecewise((1.0590824, (x0)>40.0),
    (0.94644428042, True)))", local_dict=local_dict)

upper_eqns = [x_loc, rho, rhou0, rhou1, rhoE]
boundaries[direction][side] = DirichletBC(direction, side, upper_eqns)
block.set_block_boundaries(boundaries)
```

The fourth boundary condition is a Dirichlet condition to impose flow conditions before and after the oblique shock-wave. As this condition has a dependence on the x_0 coordinate we create a local equation to evaluate the current x_0 position during the simulation. The next four equations give the pre and post shock values for a $\theta = 3.08°$ oblique shock at Mach 2. The SymPy class `Piecewise` is used to create an if-else condition dependent on the local x_0 coordinate value.

It is important to note here that the x_0 value is stored as a temporary `GridVariable` before being used to set the conservative arrays of type `DataObject` which are stored globally throughout the simulation. The parsed equations are added to a list and then passed as an argument to the `DirichletBC` class. Having selected all four boundary conditions for the problem, they are set on the block in the last line.

3.8 Initialisation of the domain and grid

At this stage we need to select how the domain should be initialised and details of the grid equations. In OpenSBLI a grid can either be read in from a pre-computed HDF5 file or evaluated at runtime in the initialisation kernel. For this example the grid is specified as equations and added to the initialisation.

```
# Reynolds number, Mach number and free-stream temperature for the initial
    profile
Re, xMach, Tinf = 950.0, 2.0, 288.0
polynomial_directions = [(False, DataObject('x0')), (True, DataObject('x1'))]
n_poly_coefficients = 50
grid_const = ["Lx1", "by"]
for con in grid_const:
    local_dict[con] = ConstantObject(con)
gridx0 = parse_expr("Eq(DataObject(x0), block.deltas[0]*block.grid_indexes[0])",
    local_dict=local_dict)
gridx1 = parse_expr("Eq(DataObject(x1),
    Lx1*sinh(by*block.deltas[1]*block.grid_indexes[1]/Lx1)/sinh(by))",
    local_dict=local_dict)
coordinate_evaluation = [gridx0, gridx1]
initial = Initialise_Katzer(polynomial_directions, n_poly_coefficients, Re,
    xMach, Tinf, coordinate_evaluation)
```

For this SBLI case we initialise the domain with a similarity solution laminar boundary layer. The inputs for this routine are the Reynolds number based on inlet displacement thickness, the freestream Mach number and the reference temperature. The initialisation numerically solves the compressible boundary-layer equations in Python to get numerical profiles for u velocity and temperature. A high-order polynomial curve is then fitted to the profiles to create an algebraic expression in x_1 that can be evaluated at runtime throughout the domain.

In this case there is only one boundary-layer so the polynomial fit is selected in the x_1 direction only. Two constant values are created relating to the domain length in x_1 and a stretching factor used in the grid equations. Grid equations are created which will populate the x_0 and x_1 coordinate arrays at the start of the simulation. Here `block.deltas[0]` is the uniform grid spacing Δx_0 in the x_0 direction and `block.grid_indexes[0]` is the range of integer values $i = 0, \dots, N_x - 1$. For the x_1 direction an equation is provided to generate coordinates stretched hyperbolically with the `sinh` function. The initial boundary-layer class `Initialse_Katzer` is called with all of the input arguments. For general simulations not using the boundary-layer initialisation, the `GridBasedInitialisation` class would be used here instead with a set of user defined equations for the initial flow conditions at $t = 0$.

3.9 Simulation input/output data

```
kwargs = {'iotype': "Write"}
h5 = iohdf5(**kwargs)
h5.add_arrays(simulation_eq.time_advance_arrays)
h5.add_arrays([DataObject('x0'), DataObject('x1'), DataObject('D11'),
    DataObject('TENO')])
block.setio(copy.deepcopy(h5))
```

We must now tell OpenSBLI which input/output arrays are required to be read or written to disk. In the above example an `iohdf5` class is called with the `write` keyword to denote that this relates to data written out of the simulation. The `add_arrays` routine is used to add the conservative variables, coordinate arrays, metric term for stretching and the storage of the shock sensor locations. Depending on the requirements of the user, additional input/output quantities can be added here via the same method. The `read` keyword can also be used here to read in pre-computed grid files.

3.10 Creating the simulation code and writing to C

```
sim_eq = copy.deepcopy(simulation_eq)
CR = copy.deepcopy(constituent)
# Set equations on the block
block.set_equations([CR, sim_eq, initial, metriceq])
# Begin the discretisation process symbolically
block.discretise()

alg = TraditionalAlgorithmRK(block)
SimulationDataType.set_datatype(Double)
OPSC(alg)
```

Up until this point we have defined many of the components of the simulation but none of the symbolic differentiation has been performed. The simulation equations, constituent relations, initial condition and metric transformation equations are now all set on the block. To initiate the main discretisation process we call `block.discretise`. Once this is complete all of the governing equations have been discretized with the chosen numerical schemes and each of the major computations have been converted into OpenSBLI kernels.

An algorithm class is created using the simulation block that links all of the components together. The purpose of the algorithm class is to place the different components of the simulation in the correct part of the output C code. The simulation is then set to be double precision. Finally, the `OPSC` code-writer class is called with the algorithm. The OPSC class converts all of the symbolic quantities to OPS compliant C code. The current file directory now contains several *C* code files to be compiled with OPS.

3.11 Defining numerical values of constants for the simulation

```
# Substitute simulation parameter values
constants = ['gama', 'Minf', 'Pr', 'Re', 'Twall', 'dt', 'niter', 'block0np0',
    'block0np1',
               'Delta0block0', 'Delta1block0', 'SuthT', 'RefT', 'eps',
                   'TENO_CT', 'Lx1', 'by', 'teno_a1', 'teno_a2', 'epsilon']
values = ['1.4', '2.0', '0.72', '950.0', '1.67619431', '0.04', '25000', '500',
    '250',
            '400.0/(block0np0-1)', '115.0/(block0np1-1)', '110.4', '288.0',
                '1e-15', '1e-5', '115.0', '5.0','10.5', '4.5', '1.0e-30']
substitute_simulation_parameters(constants, values)
print_iteration_ops(NaN_check='rho_B0')
```

By default the generated code has no numerical values for simulation constants and parameters. These are populated by using the `substiute_simulation_parameters` function. Two aligned lists must be provided that contain the name of the constant and the numerical value. Parameters such as number of grid points, number of iterations and the time-step are specified here. The C code is now ready to use. Once you have generated a code using OpenSBLI you can manually edit the C code if required. The simulation parameters should be changed as required in the C code rather than generating a new code in Python. The last line in the code-snippet adds an optional in-simulation NaN check and termination of the simulation if one is detected.

4 Makefiles

For recent versions of OPS, compile flags and other Makefile settings are located in compiler-specific files located in `OPS/makefiles`. To compile a code in OpenSBLI the Makefile below should be present in the working directory of the simulation code.

```
# The following environment variables should be predefined:
#
# OPS_INSTALL_PATH
# OPS_COMPILER (gnu,intel,etc)
#
USE_HDF5=1
include $(OPS_INSTALL_PATH)/../makefiles/Makefile.common
include $(OPS_INSTALL_PATH)/../makefiles/Makefile.mpi
include $(OPS_INSTALL_PATH)/../makefiles/Makefile.cuda
include $(OPS_INSTALL_PATH)/../makefiles/Makefile.hdf5

HEADERS =  opensbliblock00_kernels.h defdec_data_set.h
OTHER_FILES =
OPS_GENERATED = opensbli_ops.cpp
OPS_FILES = opensbli.cpp

APP=opensbli
MAIN_SRC=opensbli

include $(OPS_INSTALL_PATH)/../makefiles/Makefile.c_app
```

5 Procedure for restarting simulations

This section outlines the procedure for restarting simulations from a previous OpenSBLI output file. For restarting simulations there is a `generate_restart.py` script available in the OpenSBLI repository. This file modifies how arrays in the simulation are declared. By default storage arrays are declared as zeros using the `ops_decl_dat` function in OPS. When restarting a simulation we instead want to initialise the values of the time advance arrays $(\rho, \rho u, \rho v, \rho w, \rho E)$ from a previous simulation checkpoint file. The `generate_restart.py` script performs a simple text replace to use `ops_decl_dat_hdf5` instead of `ops_decl_dat` with the appropriate input parameters.

1. Perform a simulation in OpenSBLI to obtain an `opensbli_output.h5` file.
2. Modify the `generate_restart.py` script available in the repository for the arrays you wish to restart.
3. Move the `opensbli_output.h5` HDF5 file to `restart.h5`.
4. Run `python generate_restart.py` in the directory where your `defdec_data_set.h` file is located.
5. Comment out the grid based initialization kernel in your main `opensbli.cpp` file to prevent the initial condition being applied again.
6. Change the number of iterations to perform in `opensbli.cpp` if necessary.
7. Translate and compile the modified code within OPS to generate a new executable.

Important note: If step 5 is not performed then the simulation will overwrite the restarted data arrays in memory with the initial condition. If the grid coordinates (x_0, x_1, x_2) were originally generated in the initialisation kernel, they should also be written to the output file and added to the `generate_restart.py` script to be reinitialised.

6 Components of the generated OPS C code

It's important to develop an understanding of how OpenSBLI uses the OPS library to perform simulations in parallel. To be able to debug simulations within OpenSBLI the user must know their way around the generated C code and the various components of OPS. The OPS library has its own set of documentation and example test cases which can be found here: `https://github.com/OP-DSL/OPS`. In this section the main features of the OPS DSL are highlighted in the context of OpenSBLI.

6.1 Preamble and constant declarations

```
int block0np0;
int block0np1;
int block0np2;
#define OPS_3D
#include "ops_seq.h"
#include "opensbliblock00_kernels.h"
int main(int argc, char **argv)
{
block0np0 = 64;
block0np1 = 64;
block0np2 = 64;
```

The start of the program defines all of the constants in the simulation and any header files that are used. Inside the main program are the numerical values for the constants which have to be filled in by the user. If changes are made to the values here, the translation step in OPS must be performed again before the code can be compiled again. Within the constants section there are values for rational constants. These are factors appearing in the equations such as $rc10 = 1.0/12.0$ which are pre-computed once at the start of the simulation. The rational constants are substituted during the code generation process and don't need to be modified by the user.

```
// Initializing OPS
ops_init(argc,argv,1);
ops_decl_const("block0np0" , 1, "int", &block0np0);
ops_decl_const("block0np1" , 1, "int", &block0np1);
ops_decl_const("block0np2" , 1, "int", &block0np2);
```

The OPS part of the code begins with the `ops_init(argc, argv, 1)` call, where the integer 1 is the level of diagnostics to perform. For further details on diagnostics see the section on performance profiling in section 12. Each of the constants must then be defined in OPS using the `ops_decl_const` function.

```
// Define and Declare OPS Block
ops_block opensbliblock00 = ops_decl_block(3, "opensbliblock00");
#include "defdec_data_set.h"
```

At this point a simulation block must be created using the `ops_decl_block` function, passing it the number of dimensions of the problem and a block name. The final line includes the `defdec_data_set.h` file which contains the declarations of storage arrays. Below is an example of an array declaration in OPS. The number of arrays in the `defdec_data_set.h` file will vary depending on the nature of the problem.

```
ops_dat rho_B0;
{
int halo_p[] = {5, 5, 5};
int halo_m[] = {-5, -5, -5};
int size[] = {block0np0, block0np1, block0np2};
int base[] = {0, 0, 0};
double* value = NULL;
rho_B0 = ops_decl_dat(opensbliblock00, 1, size, base, halo_m, halo_p, value,
    "double", "rho_B0");
}
```

An `ops_dat` storage array is declared with size $(\text{block0np0}, \text{block0np1}, \text{block0np2})$. By default all arrays in OpenSBLI are padded with five halo points on each side of the data. The `ops_decl_dat` call initialises the array to zeros. These declarations can be replaced with `ops_decl_dat_hdf5` if the user wishes to read in a pre-computed grid or previous restart file. For further details see the simulation restarting procedure in section 5.

6.2 Stencil declarations for relative data access

As OPS is a stencil-based framework, the relative data access for any computation must be specified beforehand. This is done by declaring a set of stencils at the start of the program with the `ops_decl_stencil` routine. The convention for the stencil layout is $\{x_0, y_0, z_0, \dots, x_n, y_n, z_n\}$ for accessing $i = 0, \dots n-1$ points in the x, y, z directions. In the example below the y direction is being accessed at $[-1, 0, 1]$ grid locations relative to $[0, 0, 0]$. If manual changes are made to the equations in a computational kernel, the stencil must also be updated accordingly. OpenSBLI generates all of the stencil access patterns automatically during the code generation process. The last line of the code snippet calls the `ops_partition` routine which performs the domain decomposition based on the number of MPI processes being used.

```
// Define and declare stencils
int stencil_0_01temp[] = {0, -1, 0, 0, 0, 0, 0, 1, 0};
ops_stencil stencil_0_01 =
    ops_decl_stencil(3,3,stencil_0_01temp,"stencil_0_01temp");
```

```
// Init OPS partition
ops_partition("");
```

6.3 Parallel loops: ops_par_loop

Each computation to perform in parallel within OPS must have a call to an `ops_par_loop`. These parallel regions require: a kernel function to compute, the range over which to perform the calculation, the input/output arrays and their read/write status, and the stencil data access for each array.

```
int iteration_range_31_block0[] = {-2, block0np0 + 2, -2, block0np1 + 2, -2,
    block0np2 + 2};
ops_par_loop(opensbliblock00Kernel031, "CRp_B0", opensbliblock00, 3,
    iteration_range_31_block0,
ops_arg_dat(rhoE_B0, 1, stencil_0_00, "double", OPS_READ),
ops_arg_dat(rho_B0, 1, stencil_0_00, "double", OPS_READ),
ops_arg_dat(u0_B0, 1, stencil_0_00, "double", OPS_READ),
ops_arg_dat(u1_B0, 1, stencil_0_00, "double", OPS_READ),
ops_arg_dat(u2_B0, 1, stencil_0_00, "double", OPS_READ),
ops_arg_dat(p_B0, 1, stencil_0_00, "double", OPS_WRITE));
```

The first line defines the range of grid points in x, y, z to iterate over. For this example the entire grid range plus two halos on either side are being used. The second line makes the call to the `ops_par_loop` for a kernel named `opensbliblock00Kernel031`. Each computational kernel is defined in the `opensbliblock00_kernels.h` header file explained in the next section. In this case the variables $(\rho, u_0, u_1, u_2, \rho E)$ are being read in and used to evaluate pressure to `p_B0`. As no relative data access is required for this evaluation, all of the arrays are accessed with the default $\{0, 0, 0\}$ stencil. The parallel code that is generated for this parallel region will vary depending on the computational architecture used and is determined within OPS.

6.4 Computational kernels

In the previous section a kernel: `opensbliblock00Kernel031` was used within an `ops_par_loop`. The equations evaluated in this kernel can be found in the `opensbliblock00kernels.h` file.

```
 void opensbliblock00Kernel031(const double *rhoE_B0, const double *rho_B0, const
     double *u0_B0, const double *u1_B0,
const double *u2_B0, double *p_B0)
{
   p_B0[OPS_ACC5(0,0,0)] = (gama - 1)*(rhoE_B0[OPS_ACC0(0,0,0)] -
     rc12*rho_B0[OPS_ACC1(0,0,0)]*pow(u0_B0[OPS_ACC2(0,0,0)], 2) -
```

```
    rc12*rho_B0[OPS_ACC1(0,0,0)]*pow(u1_B0[OPS_ACC3(0,0,0)], 2) -
    rc12*rho_B0[OPS_ACC1(0,0,0)]*pow(u2_B0[OPS_ACC4(0,0,0)], 2));

}
```

The kernel function evaluates the result of the pressure equation and stores it in the `p_B0[OPS_ACC5(0,0,0)]` storage array. The `OPS_ACC` identifiers correspond to the ordering of the input arguments to the function. If additional arrays are manually added to the kernel, then the access ordering must be consistent in both the function call and the equations. In this example only the $(0, 0, 0)$ location is being accessed. If manual changes are made to the data access, this must be reflected in the stencils passed to the `ops_par_loop`. When generating a code, the access patterns and stencil declarations are generated automatically by OpenSBLI.

6.5 Iteration and sub-stage loops

The first computational kernel called in the main `opensbli.cpp` file is the initialisation of the domain. This is the kernel that must be commented out when restarting a simulation as in section 5. If coordinate transformations are used in the simulation, metric terms are computed once based on the input grid.

```
double cpu_start0, elapsed_start0;
ops_timers(&cpu_start0, &elapsed_start0);
for(int iter=0; iter<=niter - 1; iter++)
{
```

After the metric calculations the main time loop begins, with `ops_timers` to record the simulation time. Boundary conditions are called at the start of each iteration and then the sub-stage loop is entered to perform the stages of the Runge-Kutta algorithm.

```
ops_halo_transfer(exchange118);
ops_halo_transfer(exchange119);
for(int stage=0; stage<=2; stage++)
{
int iteration_range_29_block0[] = {-2, block0np0 + 2, -2, block0np1 + 2, -2,
    block0np2 + 2};
ops_par_loop(opensbliblock00Kernel029, "CRu2_B0", opensbliblock00, 3,
    iteration_range_29_block0,
ops_arg_dat(rho_B0, 1, stencil_0_00, "double", OPS_READ),
ops_arg_dat(rhou2_B0, 1, stencil_0_00, "double", OPS_READ),
ops_arg_dat(u2_B0, 1, stencil_0_00, "double", OPS_WRITE));
```

In this example periodic boundary conditions are applied with calls to the `ops_halo_transfer` exchange. The number of stages for the sub-stage loop will depend on the order of the

Runge-Kutta scheme chosen; for 3rd and 4th order schemes there are 3 and 5 stages respectively. Inside the sub-stage loop the first kernel is an evaluation of one of the constituent relations for evaluating a primitive variable u_2. From this point onwards all of the computational kernels required for the simulation are called in turn. Placement of where these kernel calls are located in the code is controlled in the `algorithm.py` file.

6.6 HDF5 file output

```
char name[80];
sprintf(name, "opensbli_output.h5");
ops_fetch_block_hdf5_file(opensbliblock00, name);
ops_fetch_dat_hdf5_file(rho_B0, name);
ops_fetch_dat_hdf5_file(rhou0_B0, name);
ops_fetch_dat_hdf5_file(rhou1_B0, name);
ops_fetch_dat_hdf5_file(rhou2_B0, name);
ops_fetch_dat_hdf5_file(rhoE_B0, name);
ops_fetch_dat_hdf5_file(x0_B0, name);
ops_fetch_dat_hdf5_file(x1_B0, name);
ops_fetch_dat_hdf5_file(x2_B0, name);
ops_exit();
```

All input/output of data to file within OpenSBLI must be in the HDF5 file format. At the end of the simulation, data is written out using the `ops_fetch_dat_hdf5_file` routine. An HDF5 file is created for a simulation block by passing a file name to the `ops_fetch_block_hdf5_file` function. In this case the conservative variables and the grid are written to the output file. For intermediate data written to file during the simulation, the `save_every` keyword can be given to the `io_hdf5` class in the main problem script.

HDF5 files are easily read into Tecplot and MATLAB, the user has to strip the halo points off when accessing the data. See the `removehalos.py` file for further details. To read the data into Python the `h5py` library must be used. Example usage can be found in the provided `plot.py` files in the applications folder. For Paraview an XDMF file must be generated to load the data, a script `paraview.py` for generating these XDMF files is given in the repository. For quick viewing of HDF5 files, software such as HDFCompass (`https://support.hdfgroup.org/projects/compass/download.html`) can be used to inspect slices of the data.

7 Current scheme and boundary condition options

In this section a list of the currently available numerical schemes and boundary conditions is given with example usage. For detailed information about the boundary conditions please refer to the `bcs.py` file in the core source directory.

7.1 Numerical scheme options

7.1.1 Central schemes

For shock-free problems, central differencing schemes can be generated in OpenSBLI for any arbitrary even order.

```
order = 4
cent = Central(order)
schemes[cent.name] = cent
```

If no scheme is explicitly selected in the derivative terms in the governing equations, then the central scheme will be used by default. Halos are currently set to 5 in OpenSBLI by default, which would support a central scheme of up to 10th order. The number of halo points can be modified in the `scheme.py` file.

For applications that have non-periodic boundaries, a one-sided Carpenter scheme replaces central near the boundary. This one-sided derivative scheme modifies 5 points above the boundary to achieve 4th order derivatives. The base central scheme should therefore be limited to $\leq$ 6th order for non-periodic problems at present. Calculation of metric terms and derivatives in the constituent relations are also performed with the central scheme. A central scheme must also be chosen for the viscous terms if using the shock capturing options for the convective terms.

7.1.2 Shock-capturing WENO/TENO schemes

Shock capturing is performed via Weighted Essentially Non-Oscillatory (WENO) and Targeted Essentially Non-Oscillatory (TENO) schemes. Both options make use of the Local Lax-Friedrich's flux splitting method. A transformation of the Euler equations to characteristic space is performed before the reconstruction is applied. The transformation must be performed on an averaged state at a cell interface, selecting either `SimpleAverage` or `RoeAverage`. Note that a constituent relation for the speed of sound must be provided when using the shock capturing schemes.

WENO schemes of arbitrary odd order can be generated in the code. For orders > 7, the number of halo points in `scheme.py` should be increased. To select WENO derivatives in the governing equations the convention to use is:

```
sc1 = "**{\'scheme\':\'Weno\'}"
```

```
mass = "Eq(Der(rho,t), - Conservative(rhou_j,x_j,%s))" % sc1
```

A WENO scheme must then be selected and set on the block:

```
weno_order = 5
Avg = RoeAverage([0, 1])
LLF = LLFWeno(weno_order, formulation='Z', averaging=Avg)
schemes = {}
schemes[LLF.name] = LLF
```

Two WENO formulations are currently implemented in the code, WENO-JS and the improved WENO-Z which can be selected with the formulation argument. TENO schemes are selected in a similar manner:

```
sc1 = "**{\'scheme\':\'Teno\'}"
mass = "Eq(Der(rho,t), - Conservative(rhou_j,x_j,%s))" % sc1
```

```
teno_order = 5
Avg = RoeAverage([0, 1])
LLF = LLFTeno(teno_order, averaging=Avg)
schemes = {}
schemes[LLF.name] = LLF
```

TENO schemes have a constant user-specified parameter `TENO_CT`, which controls the numerical dissipation of the scheme. For problems containing shock-waves the recommended values for 5th and 6th TENO schemes are 1×10^{-5} and 1×10^{-6} respectively. Lower values reduce the numerical dissipation of the scheme but may generate oscillations and stability issues at shock-waves. For shock-free compressible turbulence values of 1×10^{-9} can be used to improve the resolution of the scheme.

In addition to the base TENO method, there is an adaptive TENO scheme (TENO-A) which varies the value of `TENO_CT` throughout the domain. A modified version of the Ducros sensor is used to detect discontinuities and raise the value of `TENO_CT`. In smooth regions of the flow the value drops to reduce the numerical dissipation where it is not required. The adaptive scheme can be selected with:

```
# Adaptive TENO with modified Ducros sensor
SS = ShockSensor()
shock_sensor, sensor_array = SS.ducros_equations(block, coordinate_symbol,
    metriceq)
# Add shock Ducros sensor to constituent relations
constituent.add_equations(shock_sensor)
store_sensor = True
teno_order = 6
Avg = RoeAverage([0, 1])
```

```
LLF = LLFTeno(teno_order, formulation='adaptive', averaging=Avg,
    sensor=sensor_array, store_sensor=True)
schemes = {}
schemes[LLF.name] = LLF
```

The evaluation of the shock sensor must be added to the constituent relations class for the adaptive scheme to function. The adaptive scheme defines two new constants instead of `TENO_CT`. The constants `teno_a1, teno_a2` define the upper and lower bounds of the `TENO_CT` parameter. Recommended values are `teno_a1 = 10.5`, `teno_a2 = 4.5`, which would give 1×10^{-6} around shocks and 1×10^{-10} elsewhere. For more information about the implementation and performance of the schemes, refer to `https://arc.aiaa.org/doi/abs/10.2514/6.2019-3208`.

7.1.3 Non-linear WENO filter methods for shock-capturing

OpenSBLI also contains non-linear filter methods to stabilise central schemes for problems containing strong discontinuities. These methods are more computationally efficient than using WENO/TENO for the governing equations, as the shock-capturing is only applied once at the end of a full time-step. The governing equations are solved by a non-dissipative central base scheme. At the end of each time-step the dissipative part of a high-order WENO scheme is used to filter the solution. Further details of the method can be found in: *H.C. Yee, B. Sjogreen. Recent developments in accuracy and stability improvement of nonlinear filter methods for DNS and LES of compressible flows (C&F, 2018).*

Example usage is shown in the supersonic cylinder, and Taylor-Green vortex cases in the application directory. The governing equations should be in the the skew-symmetric formulation shown in section 8.1. The filter is selected in the problem script and added to the equations list on the block. Example usage is shown below:

```
ShockFilter = WENOFilter(block, order=5, metrics=metriceq,
    dissipation_sensor='Ducros', Mach_correction=True)
block.set_equations([constituent, simulation_eq, initial, metriceq] +
    ShockFilter.equation_classes)
```

The input arguments are:

1. **block**: An OpenSBLI Simulation Block.
2. **order**: The order of the WENO scheme used as the filter (3, 5, or 7).
3. **metrics**: An OpenSBLI metric class if the grid is stretched or curvilinear. Defaults to none.
4. **dissipation_sensor**: Choice of sensor to localise the dissipation to discontinuities. Defaults to a modified version of the Ducros sensor.

5. **Mach_correction**: An optional correction to turn the filter off in low-speed regions. Defaults to False.

The sensor $\kappa \in [0, 1]$ is evaluated as

$$\kappa = \frac{1}{2}\left(1 - \tanh\left(2.5\left(1 + a\nabla \cdot \mathbf{u}\right)\right)\right)\frac{(\nabla \cdot \mathbf{u})^2}{(\nabla \cdot \mathbf{u})^2 + (\omega)^2 + \epsilon}, \tag{1}$$

with $a = 100$, $\epsilon = 1.0e^{-12}$. The tanh function filters out all regions of positive dilatation (expansion). The optional low-Mach correction is evaluated as

$$\kappa_0(M) = \min\left(\frac{M^2}{2}\frac{\sqrt{4 + (1 - M^2)^2}}{1 + M^2}, 1\right) \tag{2}$$

At the end of a full time-step U_j^*, with the non-dissipative base scheme, the filter is subtracted from the solution vector as

$$U_j^{n+1} = U_j^* - \frac{\Delta t}{\Delta x}\left[H_{j+1/2} - H_{j-1/2}\right]. \tag{3}$$

The m components of the flux $H_{j+1/2}$, are computed in each direction of the problem as

$$\bar{h}_{j+1/2} = \frac{\kappa_{j+1/2}^m}{2}\left(s_{j+1/2}^m\right)\left(g_{j+1/2}^m - b_{j+1/2}^m\right), \tag{4}$$

where $g_{j+1/2}^m$, and $b_{j+1/2}^m$ are the fluxes evaluated with a WENO scheme and a central difference respectively. Subtracting a central difference from the WENO scheme isolates the dissipative part of the WENO. For a selected WENO scheme of order n, the subtracted central approximation is of order $(n + 1)$. The $k_{j+1/2}^m$ term controls the amount of dissipation to apply, as determined by the sensor in (1). Within each of the WENO reconstruction kernels, a discontinuity sensor $s_{j+1/2}^m$, is evaluated for each of the characteristic waves. The sensor inside the WENO kernels is evaluated for non-linear and ideal WENO weights ω_r, d_r, as

$$s_{j+1/2}^m = \sum_{r=0}^{k-1} |\omega_r - d_r|. \tag{5}$$

The algorithm can be summarised as follows:

1. Recompute the primitive variables (constituent relations) based on the conservative variables at the end of a full time-step of the base scheme.

2. Evaluate the WENO reconstructions, subtract the central difference, and filter the solution using the discontinuity sensor.

3. Evaluate the modified Ducros sensor globally, to control the amount of dissipation to apply at discontinuities.

4. Filter the solution vector and proceed to the next time-step.

In addition to turning the shock-filtering off away from discontinuities, it is also disabled for the six nearest grid points at no-slip walls. The WenoFilter class detects any no-slip walls during the code-generation, and generates C code to disable the filter in the near-wall region.

7.1.4 Time advancement

OpenSBLI uses explicit low-storage Runge-Kutta schemes for time advancement. The current options are:

```
rk = RungeKutta(3)
rk = RungeKuttaLS(3)
rk = RungeKuttaLS(3, formulation='SSP')
rk = RungeKuttaLS(4)
```

The first option is the 3rd order scheme from the previous SBLI Fortran code. The second class `RungeKuttaLS` is an alternative 2-register Williamson formulation that is more computationally efficient. The `RungeKuttaLS` class is implemented for 3rd and 4th order schemes, plus a 3rd order scheme with Strong-Stability-Preserving (SSP) coefficients. All of the 3rd order schemes perform 3 stages per iteration, whereas the 4th order scheme requires 5 stages.

7.2 Boundary conditions

Many of the boundary conditions were implemented using the formulations provided by

1. **PeriodicBC:** Applies periodicity using the OPS halo exchange interface. Data is copied from the end of the domain at one side to the halo points of the other.

2. **SymmetryBC:** Applies a symmetry condition by creating a ghost flow in the halo points.

3. **DirichletBC:** Applies a constant value (time-independent) Dirichlet condition to the boundary plane and all of the halo points. The condition to be evaluated is provided by the user as a set of equations to evaluate. Spatial dependence of the condition can be included via use of the `Piecewise` function. See section 10.9 for further information.

4. **IsothermalWallBC:** No-slip solid wall condition held at a constant temperature. The user must specify the manner in which they wish to evaluate energy on the wall; refer to the `katzer_SBLI.py` problem script for guidance. Momentum components are set to zero on the wall and the density is left to be calculated by the scheme. If using shock capturing schemes, halo points are populated to create the necessary ghost flow.

5. **InletTransferBC:** A simple inflow boundary condition that copies an initial condition from the halos onto the inlet plane of the simulation. Commonly used with the laminar boundary-layer initialisation to maintain a constant inflow boundary-layer.

6. **ExtrapolationBC:** A simple low-order spatial extrapolation condition applied to each of the conservative variables. Used mainly for outflow boundaries. Zero and first order schemes are implemented.

7. **AdiabaticWallBC:** Adiabatic no-slip solid wall condition. Similar to the isothermal wall but does not enforce a constant wall temperature. The adiabatic condition is created by generating a mirrored ghost flow in the halo points.

8. **InletPressureExtrapolateBC:** Inlet boundary condition that splits based on the local sound speed on the boundary. In the subsonic region of the boundary pressure is extrapolated from the internal domain into the halo points. In the supersonic region all conservative variables are copied in from the initial condition in the halo points.

9. **ForcingStripBC:** An isothermal no-slip wall with a time-dependent blowing/suction forcing strip. The user must provide the form of the wall-normal velocity component. See the transitional SBLI case for further guidance.

A few additional boundary conditions will be included in the next release version.

7.3 Miscellaneous filters

7.3.1 Explicit binomial filter

One method of filtering oscillations in the freestream is to apply an explicit filter at the end of each full time-step. This method has been used to damp disturbances close to far-field boundaries for the circular cylinder and aerofoil cases. For a solution vector U, and filter strength σ, the filter is applied as

$$U = (1 - \sigma)U + \sigma U_f, \tag{6}$$

where the average filtered solution U_f is given by

$$U_f = (U_{fx} + U_{fy} + U_{fz})/3. \tag{7}$$

In each direction the filter U_{fx_i} is an explicit filter of order n, with stencil width of $(n+1)$ points. The coefficients for the stencil points are taken from the binomial coefficients from the expansion of $(a-b)^n/2^n$. The filter can be selected in a problem script by:

```
# Add an explicit filter in a specified region of the domain
j = block.grid_indexes[1]
grid_condition = j >= 169
BF = BinomialFilter(block, order=10, grid_condition=grid_condition, sigma=0.01)
block.set_equations([constituent, simulation_eq, initial, metriceq]+
    BF.equation_classes)
```

The input arguments are:

1. **block:** An OpenSBLI simulation block.
2. **order:** The order of the filter. Even orders between 2 and 10.
3. **grid_condition:** An optional spatial dependence to limit the filter to specific regions of the domain. The input can be any (or multiple) conditions based on grid indices or coordinates (x_0, x_1, x_2).
4. **sigma:** The percentage blending strength of the filter between 0 and 1.

7.3.2 Selective Frequency Damping (SFD) filter

A second method is available for filtering certain frequencies above a threshold value. Further details are given in *Richez, Leguille and Marquez, "Selective frequency damping method for steady RANS solutions of turbulent separated flows around an airfoil at stall", Computer and Fluids, Vol. 132, pp. 51-61 (2016)*

```
# Add selective frequency damping
SFD = SFD(block, chifilt=0.1, omegafilt=1.0/0.75)
```

The input arguments are:

1. **block:** An OpenSBLI simulation block.
2. **chifilt:** The damping coefficient.
3. **omegafilt:** The threshold cut-off frequency.

8 Example governing equations

8.1 Central schemes with skew-symmetric formulation

An example of the compressible Navier-Stokes equations written in skew-symmetric form to improve stability. An alternative skew formulation can be found in the turbulent channel flow application.

```
# Define the compresible Navier-Stokes equations in Einstein notation, by
    default the scheme is Central no need to
mass = "Eq(Der(rho,t), - Skew(rho*u_j,x_j))"
momentum = "Eq(Der(rhou_i,t) , - Skew(rhou_i*u_j, x_j) - Der(p,x_i) +
    Der(tau_i_j,x_j))"
energy = "Eq(Der(rhoE,t), - Skew(rhoE*u_j,x_j) - Conservative(p*u_j,x_j) +
    Der(q_j,x_j) + Der(u_i*tau_i_j ,x_j))"
# 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, (mu/((gama-1)*Minf*Minf*Pr*Re))*Der(T,x_j))"
substitutions = [stress_tensor, heat_flux]
# Constants that are used
constants = ["Re", "Pr", "gama", "Minf"]
# symbol for the coordinate system in the equations
coordinate_symbol = "x"
# Constituent relations used in the system
velocity = "Eq(u_i, rhou_i/rho)"
pressure = "Eq(p, (gama-1)*(rhoE - rho*(1/2)*(KD(_i,_j)*u_i*u_j)))"
temperature = "Eq(T, p*gama*Minf*Minf/(rho))"
viscosity = "Eq(mu, 1.0)" # Note: constant viscosity here
```

8.2 Shock capturing schemes in conservative form

An example of using the shock capturing schemes with the Navier-Stokes equations written in conservative form. Dynamic viscosity is being evaluated with Sutherland's law. Alternatively a power law could be used here. Note that the speed of sound must be specified as a constituent relation for the characteristic decomposition.

```
sc1 = "**{\'scheme\':\'Teno\'}"
# Define the compresible Navier-Stokes equations in Einstein notation.
mass = "Eq(Der(rho,t), - Conservative(rhou_j,x_j,%s))" % sc1
momentum = "Eq(Der(rhou_i,t) , -Conservative(rhou_i*u_j + KD(_i,_j)*p,x_j , %s)
    + Der(tau_i_j,x_j) )" % sc1
energy = "Eq(Der(rhoE,t), - Conservative((p+rhoE)*u_j,x_j, %s) - Der(q_j,x_j) +
    Der(u_i*tau_i_j ,x_j) )" % sc1
```

```
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, (-mu/((gama-1)*Minf*Minf*Pr*Re))*Der(T,x_j))"
# Substitutions
substitutions = [stress_tensor, heat_flux]
constants = ["Re", "Pr", "gama", "Minf", "SuthT", "RefT"]
# Define coordinate direction symbol (x) this will be x_i, x_j, x_k
coordinate_symbol = "x"
# Formulas for the variables used in the equations
velocity = "Eq(u_i, rhou_i/rho)"
pressure = "Eq(p, (gama-1)*(rhoE - rho*(1/2)*(KD(_i,_j)*u_i*u_j)))"
speed_of_sound = "Eq(a, (gama*p/rho)**0.5)"
temperature = "Eq(T, p*gama*Minf*Minf/(rho))"
viscosity = "Eq(mu, (T**(1.5)*(1.0+SuthT/RefT)/(T+SuthT/RefT)))"
```

9 Monitoring quantities during the simulation

OpenSBLI has a *SimulationMonitor* class to print output of point values (data probes) during a simulation if required. The output displays the current simulation time, iteration number, and a series of column data for each of the probes, printed at a given iteration frequency. The simulation monitor class can be added to a problem script as follows.

```
arrays = ['u0', 'p']
arrays = [block.location_dataset('%s' % dset) for dset in arrays]
indices = [('block0np0/2', 'block0np1/2', 0), ('block0np0/2', 'block0np1/2', 0)]
SM = SimulationMonitor(arrays, indices, block, print_frequency=250,
    fp_precision=12, output_file='output.log')
alg = TraditionalAlgorithmRK(block, simulation_monitor=SM)
```

In this example there are two probes, one for streamwise velocity, and one for pressure. Each quantity has a corresponding grid location (i, j, k), given as a tuple in line 3. The grid indices can either be given as integers or as factors of the total grid points in each direction (block0np0, block0np1, block0np2). The simulation monitor is added to the algorithm in the final line and will appear at the end of the time-loop in the generated C code. The input options are:

1. **arrays:** The physical quantities to be monitored. These can be any array defined within the simulation.
2. **indices:** The (i, j, k) grid indices for the probe's location.
3. **block:** An OpenSBLI simulation block.
4. **print_frequency:** The desired printing frequency.
5. **fp_precision:** Floating point precision of the output.
6. **output_file:** Name of the log file. If this argument is not provided, the output is printed to standard out.

10 Useful SymPy functions for OpenSBLI development

To be able to develop new features for OpenSBLI, a good knowledge of Python and the SymPy library is required. The best resource is the SymPy online documentation: `https://docs.sympy.org/1.1/index.html`. Some commonly used functionality in OpenSBLI is listed here with examples.

10.1 Pretty printing of equations: pprint

At any point within the code generation process the user is able to take a list of symbolic equations and print them to screen to be checked. The best way to do this is with SymPy's `pprint` function. Many of the classes in OpenSBLI will store equations as an `equations` attribute to an object, which can be iterated over and printed to screen:

```
from sympy import pprint
for equation in kernel.equations:
   pprint(equation)
```

The pretty print function can be used throughout the entire source-code and in the problem set up files.

10.2 Creating symbolic objects: symbols

There are three main ways of generating symbolic objects in OpenSBLI. The first is to simply call one of the OpenSBLI object classes with a string as a name:

```
p = DataObject('p')
```

This creates a `DataObject` which will later be converted into a `DataSet` defined on a block. Note that the variable p on the left hand side is simply a Python variable used to refer to this object during code generation, it has no effect on the output code. The name passed to the `DataObject` class as a string will be written out in the simulation code, so ensure that this variable has been defined elsewhere in the script. If you have already created a `SimulationBlock` you can create a `DataSet` directly on that block with:

```
p = block.location_dataset('p')
pprint(p)
p_B0[0,0,0]
```

Note that the DataSet has indices as it is an indexed object with i indices for the i number of dimensions in the problem. The `DataSet` has also been given a '_B0' suffix to denote that it is defined on a block numbered 0. The third method is a quick way to create several symbols of the same type using the `symbols` class of SymPy.

```
p, u, v, w = symbols('p u v w', **{'cls': DataObject})
```

The type of object created here could be `DataObject`, `GridVariable` or `ConstantObject` depending on the application.

10.3 Off-setting an indexed object

OPS uses a stencil-based approach for data access. As the `ops_par_loop` iterates over the discrete grid points, the current grid point being evaluated is always denoted as [0,0,0] in three dimensions. To build finite-difference approximations we need the ability to access data that is offset from the [0,0,0] location. OpenSBLI contains a function to let users index DataSets for a given number of grid points away from the current location in a certain direction. For example to index the above DataSet in the $i = 1$ dimension by three grid points:

```
from opensbli.utilities.helperfunctions import increment_dataset
p = block.location_dataset('p')
pprint(p)
p_B0[0,0,0] # The base [0,0,0] location for a generic stencil
direction, increment = 1, 3 # index direction 1 by 3 grid points
p_shifted = increment_dataset(p, direction, increment)
pprint(p_shifted)
p_B0[0,3,0]
```

This functionality is frequently used when developing numerical schemes or boundary conditions, where the expression requires data offset from the current [0,0,0] grid point. For example to create a 4th order central difference approximation for a given direction, the user would create a list [-2, -1, 1, 2] and build the finite-difference approximation by looping over the offsets with four applications of the `increment_dataset` function.

10.4 Substitution of symbolic quantities

SymPy has various methods for substituting terms into existing symbolic expressions. Most of these rely on setting up either a Python dictionary or tuples to create pairs of the existing terms and the new term to be substituted in to the expression. An example with the `replace` function would be:

```
for old, new in zip(to_be_replaced, new_terms):
    term = term.replace(old, new)
```

Alternatively the SymPy function `subs` can be used with a substitution dictionary containing `{key: value}` pairs such that:

```
expression.subs(substitution_dictionary)
```

Always `pprint` the expression before and after the substitution to check that you are getting the correct behaviour. Sometimes for more complex expressions multiple applications of the substitution functions will be required.

10.5 Left and right hand sides of equations

When manipulating equations in OpenSBLI, the user will often need to access the left and right hand sides of an equation (of type `Eq`) individually. This is true for the substitutions in the previous section, where it is often the right hand side of an equation that needs to be modified. The left and right hand sides of an equation can be accessed simply with:

```
print(example_equation)
Eq(p, (gama - 1)*(-rho*u0**2/2 - rho*u1**2/2 - rho*u2**2/2 + rhoE))
print(example_equation.lhs)
p
print(example_equation.rhs)
(gama - 1)*(-rho*u0**2/2 - rho*u1**2/2 - rho*u2**2/2 + rhoE)
```

10.6 De-nesting multiple iterables

Often in OpenSBLI you will encounter iterables that contain iterables. A common example of this would be a list of lists in Python. To be able to iterate over all of the individual items directly in a single loop it is often convenient to use the SymPy function `flatten`. This function collapses the list of list structure into a single list containing all of the items to iterate over.

```
>>> from sympy import flatten
>>> x = [[1, 2, 3], [4, 5,6]]
>>> print(x)
[[1, 2, 3], [4, 5, 6]]
>>> print(flatten(x))
[1, 2, 3, 4, 5, 6]
```

10.7 Accessing attributes in Python objects: __dict__

All objects in Python have a `.__dict__` routine to show the attributes contained in that object. This is frequently used when developing features for OpenSBLI. The `.__dict__` routine returns a Python dictionary `{key: value}` pair. For example, in an OpenSBLI `Kernel` class the equations defined in that kernel will be stored and can be accessed in the attribute `kernel.equations`. In the code snippet below the `.__dict__` function is showing the attributes defined in the Runge-Kutta time-stepping class after it has been instantiated.

```
>>> rk = RungeKutta(3)
>>> print(rk.__dict__)
{'solution_coeffs': rkold[stage], 'stage_coeffs': rknew[stage], 'name':
    'RungeKutta', 'iteration_number': iter, 'temporal_iteration': iter,
    'solution': {}, 'schemetype': 'Temporal', 'constant_time_step': True,
    'time_step': dt, 'order': 3, 'nloops': 2, 'stage': stage}
```

10.8 Using built-in SymPy functions (sin, cos, exp, ...)

It's important to understand the differences between the common mathematics functions that are available in Python. In the majority of cases for OpenSBLI, common math functions should be imported from the symbolic SymPy library. A common mistake is to use either the Python `math` library or `NumPy`. Both of these will throw errors when trying to build symbolic expressions in OpenSBLI.

```
>>> import math, numpy, sympy
>>> print([math.cos(0.5), numpy.cos(0.5), sympy.cos(0.5)])
[0.8775825618903728, 0.8775825618903728, 0.877582561890373]
>>> x = sympy.symbols('x')
>>> sympy.cos(x)
cos(x)
>>> math.cos(x)
Traceback (most recent call last):
   raise TypeError("can't convert expression to float")
TypeError: "can't convert expression to float"
```

Sometimes it can be useful to ask SymPy not to evaluate numerical expressions, or explicitly use symbolic expressions in a function:

```
>>> sympy.cos(0.5)
0.877582561890373
>>> sympy.cos(0.5, evaluate=False)
cos(0.5)
>>> sympy.cos(sympy.Rational(1,2))
cos(1/2)
```

Other common mathematical functions in SymPy can be used in OpenSBLI, which will be converted into C code at the code generation stage. Examples include `Abs, Min, Max, ceil`.

10.9 Conditional expressions (if-else branching)

SymPy also has a boolean logic system that allows us to symbolically create branching if-else statements in the C code. A simple conditional function such as

$$\rho = \begin{cases} 10, & \text{if } x < 3 \\ 5, & \text{if } x \geq 5 \\ 0, & \text{otherwise} \end{cases}$$

would be defined in OpenSBLI using SymPy's `Piecewise` function as:

```
x = DataObject('x0')
eqn = Eq(DataObject('rho'), Piecewise((10.0, x<3), (5.0, x>=5), (0.0,True)))
```

Each `(value, condition)` tuple pair will be written out as branching if and else-if statements. The `(value, True)` tuple is evaluated if all of the other conditions are not met in the simulation code. The Piecewise function will throw an error if an else statement is not provided in this format. More sophisticated conditional expressions can be built up using the SymPy `And`, `Or` and `Equality` classes. Each of these logic functions will be translated to the C equivalent in the final code.

OpenSBLI also has a `GroupedPiecewise` class derived from the base SymPy version. GroupedPiecewise is useful when multiple equations are to be evaluated inside one of the conditional branches. Usage can be found for example in the `teno.py` file. Tuple pairs of values and conditions can be built up using SymPy's `ExprCondPair` class to be used to form a set of branching statements. Note that SymPy will simplify conditional logic if possible so you may not see every condition entered into the function.

10.10 String representation: srepr

Often it is useful to find out how SymPy is storing the symbolic quantities through its string representation functionality. The function `srepr` breaks an expression down into its individual components and can be useful for debugging purposes. For example the pressure evaluation

$$p = (\gamma - 1)\left(\frac{u_0^2}{2} + \frac{u_1^2}{2} + \frac{u_2^2}{2} + \rho E\right) \tag{8}$$

Has the string representation of:

```
OpenSBLIEquation(DataObject('p'), Mul(Add(ConstantObject('gama'), Integer(-1)),
    Add(Mul(Integer(-1), Rational(1, 2), DataObject('rho'), Pow(DataObject('u0'),
    Integer(2))), Mul(Integer(-1), Rational(1, 2), DataObject('rho'),
    Pow(DataObject('u1'), Integer(2))), Mul(Integer(-1), Rational(1, 2),
    DataObject('rho'), Pow(DataObject('u2'), Integer(2))), DataObject('rhoE'))))
```

This is a good way of finding out which quantities are DataObjects, DataSets, ConstantObjects and GridVariables within an OpenSBLI expression. Finding the type of a single object can be done via the standard `type()` function in Python.

10.11 Breaking down an expression into its components: atoms/args

Being able to manipulate symbolic expressions in OpenSBLI relies on knowing how to isolate certain components. Two useful functions within SymPy are `atoms` and `args`.

```
>>> print(expression)
(gama - 1)*(-rho*u0**2/2 - rho*u1**2/2 - rho*u2**2/2 + rhoE)
>>> print(expression.atoms())
set([-1, -1/2, 2, gama, rho, rhoE, u0, u1, u2])
>>> print(expression.atoms(DataObject)))
set([rho, rhoE, u0, u1, u2])
>>> print(expression.atoms(ConstantObject))
set([gama])
```

A call to the `atoms()` routine on any expression will return a set of symbolic quantities in it. This includes integers and rational constants. To extract a certain type of object from the expression to iterate over, an object type can be passed to atoms such as `atoms(DataObject)`. Note that the output is a Python `set`, which is an unordered collection of unique objects. These differ from a Python `list`, which are an ordered collection that may contain duplicate entries. Useful operations can be performed on multiple sets such as `intersection`, `union` and `difference`. These operations are often used in OpenSBLI to find all of the unique objects that have been defined throughout the code.

```
# Take the right hand side of the continuity equation
>>> continuity_equation = simulation_eq.equations[0].rhs
>>> print(continuity_equation)
-CD rhou0_B0 x0 - CD rhou1_B0 x1 - CD rhou2_B0 x2
>>> print(continuity_equation.args)
(-CD rhou0_B0 x0, -CD rhou1_B0 x1, -CD rhou2_B0 x2)
# Take the first term d(rhou0)/dx
term = continuity_equation.args[0]
# Iterate over the components of the derivative operator
>>> print([x for x in term.args])
[-1, CD rhou0_B0 x0]
# Iterate over the derivative term
>>> derivative = term.args[1]
>>> print([x for x in derivative.args])
[rhou0_B0, x0]
```

The example above shows how to break down an expression into its individual terms and then isolate the individual components that make up those terms.

10.12 Isolating terms of a certain type: isinstance

The Python function `isinstance` is frequently used to isolate objects of a certain type for modification. A use case for this would be if the user wanted to perform a modification only on a certain type of object in an expression while leaving the rest untouched. For example to modify only DataObjects in a given expression one could use:

```
for term in equation.rhs.atoms():
    if isinstance(term, DataObject):
        # Perform some operation on the term
```

10.13 Using GridVariables as temporary evaluations

The purpose of the `GridVariable` object is to evaluate a quantity locally for the current grid point, use it in an expression to set a `DataSet` and then discard it. The `GridVariable` will take different values at different grid points, but unlike a `DataSet` it is not stored globally for later use. Examples of this are seen throughout OpenSBLI, for example in the definition of boundary conditions.

10.14 Simplifying long expressions and reducing operation counts

The final form of more complex expressions is not always the most efficient from a computational performance point of view. SymPy has a number of functions that can be used to try and simplify an expression in OpenSBLI. It is not always obvious which one is suitable (if any), without printing the expression before and after the modification. Among the possibilities for a given expression are:

1. `simplify(expression)`: Attempts to simplify an expression.
2. `factor(expression)`: Attempts to factor polynomials.
3. `horner(expression)`: Reduces a polynomial into a form requiring the minimum number of additions and multiplications by the Horner algorithm.

One method of assessing the impact of the attempted simplification is the `count_ops()` routine. Using `expression.count_ops()` gives the number of operations in the expression. This can be applied before and after one of the simplification methods to assess the output. Further methods for simplifying expressions can be found here: `https://docs.sympy.org/1.1/tutorial/simplification.html`.

11 HPC set-up on CSD3 NVIDIA P100 GPU machine

These guidelines are for running OpenSBLI on the University of Cambridge 'Service for Data Driven Discovery' (CSD3) system operated by the University of Cambridge Research Computing Service (`http://www.hpc.cam.ac.uk`), funded by EPSRC Tier-2 capital grant **EP/P020259/1**. The code runs on the NVIDIA P100 GPUs using the Intel compiler and OpenMPI for multi-GPU configurations.

11.1 Example bashrc settings

```
# Loading the necessary modules
module unload openmpi-1.10.7-gcc-5.4.0-jdc7f4f
module unload gcc-5.4.0-gcc-4.8.5-fis24gg
module load intel/bundles/complib/2017.4
module unload intel/impi/2017.4/intel
module load openmpi-1.10.7-intel-17.0.4-exlc3jx
module load hdf5/openmpi-gdr/1.8.19
module unload hdf5/impi/1.8.16

# OPS specific library paths
export OPS_COMPILER=intel
export NV_ARCH=Pascal
export HDF5_INSTALL_PATH=/usr/local/Cluster-Apps/hdf5/openmpi-gdr/1.8.19/
export OPS_INSTALL_PATH=~/software/OPS/ops/
export MPI_INSTALL_PATH=/usr/local/software/spack/spack-0.11.2/opt/
spack/linux-rhel7-x86_64/intel-17.0.4/openmpi-1.10.7-exlc3jxza6tcpa5eqfdnfibf34obiukz/
export USE_HDF5=1

# Direct the Python installation to OpenSBLI if generating codes
export PYTHONPATH=$PYTHONPATH:~/software/opensbli/
```

Example settings for the `~/.bashrc` file are given above. Once these modules have been loaded and the paths exported, the OPS libraries can be built as in section 1. The same modules should be loaded in the submit script for batch submission of jobs. To run on a single GPU, the user would compile: `make opensbli_cuda` and run: `./opensbli_cuda`. For a 4 GPU job compile: `make opensbli_mpi_cuda` and run `mpirun -np 4 ./opensbli_mpi_cuda`. The Makefile in section 4 should be in the working directory of the simulation code.

12 Performance profiling of OPS

OPS has built-in functionality for profiling the performance of the code on different architectures. Any existing code generated by OpenSBLI can be altered to obtain the performance output. For full details please see the OPS user manual: `https://op-dsl.github.io/docs/OPS/user.pdf`. The main steps are:

1. In your `opensbli.cpp` file find `ops_init(argc,argv,1);`.
2. Change the diagnostics level from 1 to 5.
3. At the bottom of your simulation code before `ops_exit();` add `ops_timing_output(std::cout);`
4. Recompile and run the code for a small number of iterations. The timing output will be printed to standard out.

The output gives a breakdown of how much time was spent on each kernel and the associated MPI communication time. Refer to the kernel numbering in your code to assess which components are taking the majority of the simulation time.

13 Opening OpenSBLI HDF5 output files in post-processing software

OpenSBLI HDF5 output files can be post-processed with the following third-party software.

13.1 Python

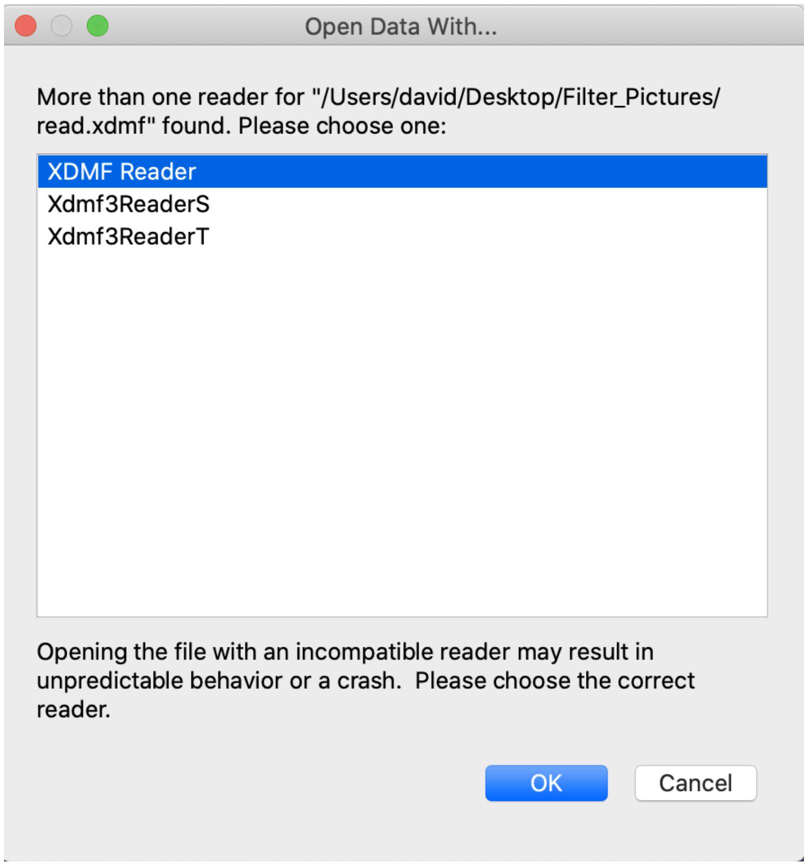
OpenSBLI HDF5 files can be read into Python using the h5py library. Example usage to open a file:

```
f = h5py.File('opensbli_output.h5', 'r')
group = f["opensbliblock00"]
```

Many of the example applications have Python plotting scripts that demonstrate how to remove halo points and access the variables within the HDF5 file.

13.2 MATLAB

MATLAB can read or display the contents of HDF5 files using the `h5read`, and `h5disp` files respectively.


Open Data With...

More than one reader for "/Users/david/Desktop/Filter_Pictures/read.xdmf" found. Please choose one:

Opening the file with an incompatible reader may result in unpredictable behavior or a crash. Please choose the correct reader.

Figure 1: XDMF option for Paraview.

13.3 Paraview

Paraview requires an XDMF file to read HDF5 files. A *paraview.py* script is included in the repository to generate XDMF files. Example usage:

```
python paraview.py opensbli_output.h5
```

The script removes the halos from the data file, and generates an XDMF file to be opened in Paraview. Note that the grid coordinates (x_0, x_1, x_2) must be written to the output file for Paraview to set up the domain geometry.

Once in Paraview, the file can be loaded by selecting `File -> Open -> XDMF reader`, as in Figure 1. Click Apply on the left hand side, and the geometry will be visible.

13.4 Tecplot

OpenSBLI output files can be read into Tecplot natively, selecting the HDF5Loader option when prompted. To remove the halo points select `Data -> Extract -> Subzone`, and for each direction set the start and end values to 6, and $N - 5$ respectively. Click 'Zone Style' on the left, and deselect 'Zone 1' to show the data without halos. This process can be repeated for multiple blocks in succession for multi-block cases. To choose the correct coordinates, select `Plot -> Assign XY`, and select the coordinate arrays in the output file.