OpenSBLI Documentation

Release 1.0.0

Satya P. Jammy, Christian T. Jacobs, Neil D. Sandham

Jul 05, 2018

Contents

1 Introduction **3**
- 1.1 Overview . . . 3
- 1.2 Licensing . . . 3
- 1.3 Citing . . . 3
- 1.4 Support . . . 4

2 Getting Started **5**
- 2.1 Dependencies . . . 5
- 2.2 Obtaining OPS . . . 5
- 2.3 Installing OpenSBLI . . . 6

3 Defining and Running a Problem **7**
- 3.1 Problem setup . . . 7
- 3.2 Equation specification . . . 8
- 3.3 Generating and compiling the model code . . . 8

4 Application: 1D wave propagation **9**
- 4.1 Equations . . . 9
- 4.2 Simulation setup . . . 9
- 4.3 Running and plotting results . . . 9

5 Citing **13**
- 5.1 Journal articles . . . 13
- 5.2 Datasets . . . 13

6 Indices and tables **15**

Contents:

CHAPTER 1

Introduction

1.1 Overview

OpenSBLI is an automatic code generator that expands a set of equations written in Einstein notation, and automatically generates code (in the OPSC language) which performs the finite difference approximation to obtain a solution. This OPSC code can then be targetted with the OPS library towards specific hardware backends, such as MPI/OpenMP for execution on CPUs, and CUDA/OpenCL for execution on GPUs.

The main focus of OpenSBLI is on the solution of the compressible Navier-Stokes equations with application to shock-boundary layer interactions (SBLI). However, in principle, any set of equations that can be written in Einstein notation may be solved using the code generation framework. This highlights one of the main advantages of such a high-level, abstract approach to computational model development.

From an implementation perspective, the OpenSBLI codebase is written in the Python (2.7.x) language and depends on the SymPy library to process the Einstein formulation of the equations. The code generator will then write out the model code which performs the finite difference approximations in any of the supported languages (currently only OPSC, although the structure of the codebase is such that other languages can be integrated with minimal effort).

The development of OpenSBLI was supported by EPSRC grants EP/K038567/1 (“Future-proof massively-parallel execution of multi-block applications”) and EP/L000261/1 (“UK Turbulence Consortium”). It was also supported by the ExaFLOW project (funded by the European Commission Horizon 2020 Framework grant 671571).

1.2 Licensing

1.3 Citing

If you use OpenSBLI, please consider citing the papers and other resources listed in the Citing section.

1.4 Support

The preferred method of reporting bugs and issues with OpenSBLI is to submit an issue via the repository's issue tracker. Users can also email the authors Satya P. Jammy and Christian T. Jacobs directly.

CHAPTER 2

Getting Started

2.1 Dependencies

You should first ensure that all the core dependencies listed in the `README.md` file are satisfied. Many of these packages can be installed either via a package manager such as `apt`, or via the Python package manager (pip) using

```
sudo pip install -r requirements.txt
```

from the OpenSBLI base/root directory.

2.2 Obtaining OPS

In order to target and compile the generated OPSC code, you will need to have OPS available. First, clone the OPS GitHub repository using

```
git clone https://github.com/gihanmudalige/OPS.git
```

and install it by running `sudo python setup.py install` from within the OPS directory that is created by the `git clone` process.

You will then need to set up your OPS-related environment variables, listed below. Note that the values given here are system-dependent and may need to be adapted depending on where the MPI or HDF5 libraries are installed. Furthermore, it is assumed that the OPS GitHub repository has been cloned in your home (~) directory.

```
export OPS_INSTALL_PATH=~/OPS/ops
export OPS_COMPILER=gnu
export MPI_INSTALL_PATH=/usr/
export HDF5_INSTALL_PATH=/usr/
```

You can include these export commands in your `~/.bashrc` file to save typing them out each time you open up a new terminal.

2.3 Installing OpenSBLI

First, clone the OpenSBLI GitHub repository using

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

You can install OpenSBLI using

```
sudo make install
```

from within the base directory of OpenSBLI. Alternatively, particularly for developers of OpenSBLI, you can simply point your `PYTHONPATH` environment variable to the OpenSBLI base directory using, for example,

```
export PYTHONPATH=$PYTHONPATH:~/opensbli
```

After installation, it is recommended that you run the test suite to check that OpenSBLI is performing as it should by using

```
make test
```

CHAPTER 3

Defining and Running a Problem

3.1 Problem setup

Essentially, OpenSBLI comprises the following classes and modules (emboldened below), which define the abstraction employed:

- A **Problem** defines the physical problem's dimension, the equations that must be solved, and any accompanying formulas, constants, etc.
- This Problem comprises many **Equations** representing the governing model equations and any constitutive formulas that need to be solved for. The Problem also performs the expansion on these equations and formulas about the Einstein indices.
- Once the equations are expanded, a numerical **Grid** of solution points and numerical **Scheme** s are created in order to discretise the expanded equations. Several Schemes are available, such as **RungeKutta** and **Explicit** for time-stepping schemes, and **Central** for central differencing in space. The spatial and temporal discretisation is handled by the **SpatialDiscretisation** and **TemporalDiscretisation** classes, respectively.
- The setting of any boundary conditions and initial conditions are handled by the **BoundaryConditions** and **GridBasedInitialisation** classes.
- The computational steps performed by the discretisation processes are described by a series of **Kernel** objects.
- All of the above classes come together to form a computational system which is written out as **OPSC** code.
- All LaTeX writing (mainly for debugging purposes) is handled by the **LatexWriter** class.

OpenSBLI will expect all these problem-specific settings and configurations (the governing equations, any constitutive formulas for e.g. temperature-dependent viscosity, what time-stepping scheme is to be used, the boundary conditions, etc.) to be defined in a separate Python script, which will eventually call the various OpenSBLI code generation routines. There are several examples provided in the applications (`apps`) directory of the OpenSBLI package.

3.2 Equation specification

Although the equations can be specified at a very abstract level in Einstein notation, certain rules are to be followed while writing them:

- All equations are written in the form `Eq(LHS,RHS)`, where `LHS` is the time dependant term in the equation and `RHS` are the terms of the equations that are equated to the time dependant governing equation.
- The Einstein indices should be prefixed with an underscore (_) and multiple indices should have multiple underscores. For example, a vector is written as `f_i` and a tensor is written as `f_i_j`.
- Derivatives that do not require special handling (e.g. single functions, chain rule applications for multiple derivatives) should be written in the form `Der(f,direction)`, where `f` is the function and `direction` is the direction.
- Derivatives involving more than one function that needs special handling like the conservative or skew-symmetric forms of the Navier-Stokes equations are handled using `Conservative` or `Skew`, respectively.
- OpenSBLI can handle all standard functions in SymPy (i.e. Kronecker Delta and Levi-Civita terms).

3.3 Generating and compiling the model code

Once defined, users can run the Python script defining the problem's configuration and generate the code for this particular problem:

```
python /path/to/directory/containing/problem_file.py
```

OpenSBLI will create two files written in the OPSC language: `simulation_name_here_block_0_kernel.h` and `simulation_name_here.cpp`. The latter file will automatically be passed through OPS's translator to target the OPSC code towards different backends, e.g. CUDA, MPI, OpenMP, etc; this yields a new file called `simulation_name_here_ops.cpp` and various directories corresponding to the different backends. It is this file that will be compiled to create the model's executable file. Note that, if OPS's translator cannot be called by OpenSBLI, you will need to run it manually using

```
python ~/OPS/translator/python/c/ops.py simulation_name_here.cpp
```

Finally, copy across the `Makefile` from one of the existing `apps`, and modify the simulation name appropriately so that it will compile the source for your simulation setup. To create a serial executable, run `make simulation_name_here_seq`. For MPI parallel executution, run `make simulation_name_here_mpi`. Similar commands can be run for GPU backends.

CHAPTER 4

Application: 1D wave propagation

4.1 Equations

This test case solves the numerical solution of the one-dimensional wave equation, written as

$$\frac{\partial \phi}{\partial t} + c\frac{\partial \phi}{\partial x} = 0,$$

where ϕ is the transported quantity and c is a known constant representing the wave speed (set to 0.5 m/s in this simulation).

4.2 Simulation setup

A domain of length $0 \leq x \leq 1$ m is considered, with grid spacing dx = 0.001 m, and periodic boundaries. An eighth-order accurate central differencing scheme is used to spatially discretise the domain, and a third-order Runge-Kutta timestepping scheme is used to march the equation forward in time.

The initial condition is defined by

$$\phi(x, t = 0) = \sin(2\pi x).$$

The simulation was run with a timestep of $dt = 4 \times 10^{-4}$ s until time t = 1 s (i.e. 2,500 iterations).

4.3 Running and plotting results

The simulation can be run sequentially using

```
python wave.py
cd wave_opsc_code
make wave_seq
./wave_seq
```

or by using the `run.py` file provided:

```
python run.py
```

The state of the solution field at the final iteration will be written to an HDF5 file called `wave_2500.h5`. This file can be read, and the results plotted, using

```
python plot.py
```

which will generate two figures; one showing the propagation of the initial sine wave (see `phi.pdf` and Figure *phi*), and one showing the error between the analytical solution (i.e. the initial wave translated to the right by $x = ct$) and the numerical solution (`phi_error.pdf` and Figure *phi_error*).

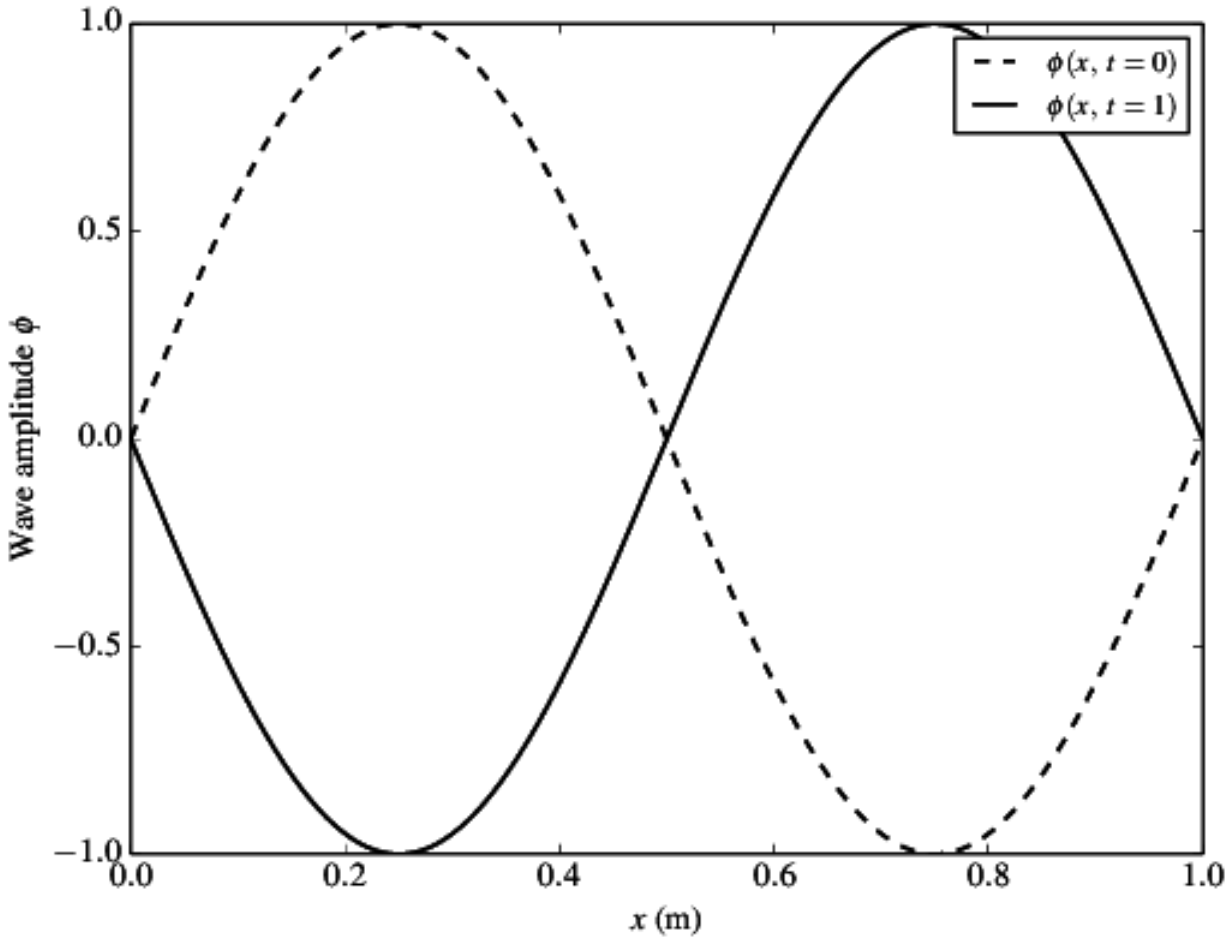

Fig. 1: The solution field ϕ at time t = 0 s and t = 1 s

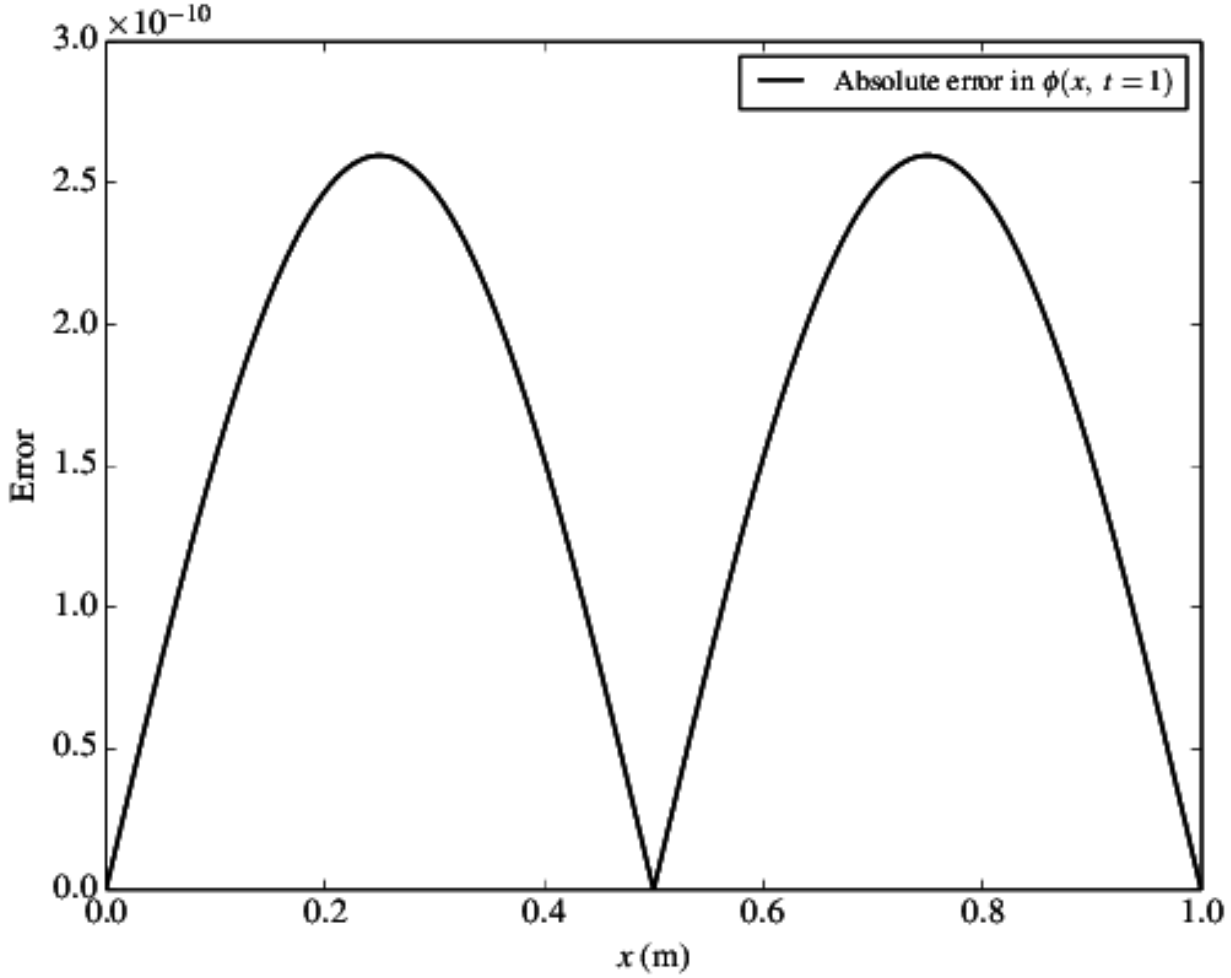

Fig. 2: The error between the analytical solution and the numerical solution at time t = 1 s.

CHAPTER 5

Citing

5.1 Journal articles

- Jacobs, C. T., Jammy, S. P., Sandham N. D. (2017). **OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures**. *Journal of Computational Science*, 18:12-23, DOI: 10.1016/j.jocs.2016.11.001
- Jammy, S. P., Jacobs, C. T., Sandham N. D. (In Press). **Performance evaluation of explicit finite difference algorithms with varying amounts of computational and memory intensity**. *Journal of Computational Science*, DOI: 10.1016/j.jocs.2016.10.015

5.2 Datasets

- Jammy, S. P., Jacobs, C. T., Sandham N. D. (2016). **Enstrophy and kinetic energy data from 3D Taylor-Green vortex simulations**. *University of Southampton ePrints repository*. DOI: 10.5258/SOTON/401892
- Jacobs, C. T., Jammy, S. P., Sandham N. D. (2016). **Solution field data from a three-dimensional Taylor-Green vortex simulation**. *University of Southampton ePrints repository*. DOI: 10.5258/SOTON/402073
- Jacobs, C. T., Jammy, S. P., Sandham N. D. (2016). **Solution field data from a one-dimensional wave propagation simulation**. *University of Southampton ePrints repository*. DOI: 10.5258/SOTON/402070
- Jacobs, C. T., Jammy, S. P., Sandham N. D. (2016). **Data from a convergence study based on the Method of Manufactured Solutions**. *University of Southampton ePrints repository*. DOI: 10.5258/SOTON/402072

CHAPTER 6

Indices and tables

- genindex
- modindex
- search