OOPS Documentation

Jacob Fields

April 2020

Contents

1 Introduction

Let's talk about how scientific codes get written. Some graduate student named Steve has a tough problem to solve, so he learns just enough coding and numerical methods to solve his one problem. The code isn't pretty, but it doesn't have to be because it's simple and easy-to-use for Steve's one problem. Steve eventually graduates, but his advisor, Dr. Fred Smith, holds onto a copy of the code because it works so well.

Now enters Bob, a bright-eyed graduate student with enough ~~stupidity~~ ambition to pursue Dr. Smith's next tough problem and absolutely no coding experience. This tough problem happens to be somewhat similar to Steve's problem, so Dr. Smith starts Bob on Steve's old code. Roughly three months later, Bob has cannibalized Steve's code and extended it to work with his problem, too. It's ugly and gross in every way, but it still runs decently quickly and is simple enough that no one cares. He graduates and quickly forgets about the code.

Dr. Smith repeats this cycle with three or four more graduate students, plus about a dozen undergraduates who pop into his research group along the way. Halfway through, Annie the undergrad notices that the code is an absolute trainwreck and decides to rewrite portions of it. Unfortunately, she has to graduate, so she only rewrites the portions that she needs for her project, and now the code is a horrible amalgam of modern object-oriented programming, Steve's hard-coded relics, and everyone else's half-baked tweaks which never should have seen the light of day. Naturally, the documentation is more sparse than an SPS activity without free food, so Dr. Smith's students spend half their time trying to decipher what has become known as "The Frankencode."

The Object-Oriented PDE Solver, or OOPS, is designed to alleviate as many of these problems as possible. By creating a flexible object-oriented design, even those relatively inexperienced with coding and numerical methods can quickly solve their problems without ever having to worry about much more than the

PDE itself. At the same time, the framework is flexible enough that those who *do* need to worry about it can.

2 Getting Started with OOPS

Because the OOPS framework is currently so small and lightweight, the entire framework is compiled with each project without any extra library dependencies. We warn the user that this will likely change in a future release as the project grows in scope.

2.1 System Requirements

OOPS requires CMake 3.0 or later and a C++ compiler supporting C++11 or later. Some optional features require the `bbhutil` library from RNPL. This package is quite old and is only known to work on select Linux distributions (Ubuntu 16.04 and RHEL 7.0 have been tested) after a very onerous installation procedure. We recommend a Linux environment (native, virtual machines, or the Windows Subsystem for Linux should work), but any appropriately configured Windows or MacOS environment should work as well, albeit without the optional features depending on `bbhutil`.

The procedure for building OOPS in Linux is as follows:

```
cd <path to OOPS base directory>
mkdir build
cd build
ccmake ../
```

A configuration menu should appear. Press `c` to generate the configuration file. The default options should be sufficient. Press `c` again, then `g` to generate the make files.

```
make
```

After this, all generated executables should appear in the `build` directory.

2.2 Structure of OOPS

The most basic OOPS program contains the following components:

- A `Domain` object, which defines the space for your problem.
- One or more `Grid` objects, which subdivide a `Domain` into discrete points for numerical calculation.
- A `Solver` object, such as `RK4`, which will numerically solve any PDE fed into it.

- An `ODE` object, which provides the righthand side for a system of ODEs (such as a PDE discretized with finite-difference methods), boundary conditions, and initial conditions.

More complex codes will also make use of `Parameter` objects and `Interpolator` objects, but these are not much more difficult to add in. OOPS works as hard as possible to hide as many of the gory details behind the scenes.

2.3 Solving a Simple Initial Boundary Value Problem

For this project, we'll create a simple program using OOPS that solves the wave equation on a domain with outflow boundaries. Begin by building a new project directory for the program:

```
cd <path to OOPS base directory>
mkdir WaveEquation
cd WaveEquation
mkdir include
mkdir src
```

The `include` folder will contain any header files specific to the `WaveEquation` program, and `src` will hold the `main` file and any additional source files required.

Before we start writing our code, remember that the wave equation in one dimension takes the form

$$\partial_{tt}\phi - \partial_{xx}\phi = 0. \tag{1}$$

OOPS can't solve this in its current form because the time integrator expects it to be first order in space. Therefore, we make the definition $\pi = \partial_t \phi$, which suggests the ODE system

$$\partial_t \phi = \pi, \tag{2}$$
$$\partial_t \pi = \partial_{xx}\phi. \tag{3}$$

OOPS can actually solve this as is because we can just define a second-order derivative operator, but it's just as easy to reduce the spatial order by defining $\chi = \partial_x \phi$, giving us the system

$$\begin{aligned} \partial_t \phi &= \pi, \\ \partial_t \pi &= \partial_x \chi, \\ \partial_t \chi &= \partial_x \pi, \end{aligned} \tag{4}$$

where the third equation comes from the commutability of partial derivatives. Despite the two systems being mathematically identical and producing solutions to the wave equation, they exhibit very different numerical properties, with the first-order system being more dispersive and the second-order system being more diffusive.

2.3.1 `wave.h`

The next step is to construct our PDE. In the `include` directory, create a new file `wave.h` that reads as follows:

```
1  #ifndef WAVE_H
2  #define WAVE_H
3
4  #include <ode.h>
5
6  class Wave : public ODE {
7    private:
8      // Variable labels
9      static const unsigned int U_PHI = 0;
10     static const unsigned int U_PI = 1;
11     static const unsigned int U_CHI = 2;
12
13   protected:
14     virtual void applyBoundaries(bool intermediate);
15
16     virtual void rhs(const Grid& grid, double **u, double↩
           **dudt);
17
18   public:
19     Wave(Domain& d, Solver& s);
20     virtual ~Wave();
21
22     virtual void initData();
23 };
24
25 #endif
```

The `ode.h` file defines the `ODE` object, which defines an abstract class representing a system of ODEs (or a discretized system of PDEs). We declare a new class, `Wave`, which inherits from `ODE` and therefore reduces a lot of the work that we have to do.

Underneath the `private` label, we assign some variable names to our specific indices. This isn't necessary, strictly speaking, but it makes your code a lot more readable when we start writing the righthand-side routine.

For our `protected` methods, we have two virtual functions, `applyBoundaries()` and `rhs()`, which are both inherited from `ODE`. The method `applyBoundaries()` is used in the evolution function (which is also virtual and can be overwritten, but the default definition works well enough for this case) to fix the boundaries between each stage of the `Solver` object we'll attach later. The `intermediate` flag will be explained in more detail later, but it has to do with figuring out which data set needs to be modified. The `rhs()` method contains the righthand side of Eq. (4).

Lastly, inside the `public` region, we have a constructor and a destructor as well as `initData()`, which contains the initial data for the ODE. This is made public for convenience, as more advanced implementations may have a custom `Parameter` object, which can be used, among other things, to control the initial

conditions for a specific `ODE`.

2.3.2 wave.cpp

The next file we need to create is `wave.cpp`, which is located inside `src` and contains the definitions for `wave.h`. We'll take this one piece at a time:

```
#include <wave.h>
#include <operators.h>
#include <iostream>
#include <cmath>
```

Clearly `wave.h` is the header file we just created. The file `operators.h` contains a set of pre-defined derivative operators. Next, we need access to the I/O and math functions from the Standard Template Library, so we go ahead and include those.

```

// Constructor
Wave::Wave(Domain& d, Solver& s) : ODE(3, 0){
  if(d.getGhostPoints() < 2){
    std::cerr << "Warning: domain has fewer ghost points than expected. Expect incorrect behavior.\n";
  }
  domain = &d;
  solver = &s;

  // Set some default parameters.
  params = new Parameters();

  reallocateData();
}

// Destructor
Wave::~Wave(){
  delete params;
}
```

When we define our constructor, we have to make sure that we call the `ODE` constructor, `ODE(const unsigned int n, const unsigned int id)`. The first argument, `n`, is the number of variables for our `ODE` object, and `id` is an identifier. For our simple program, `id` isn't really important, but it's a sort of baroque method for manually type-checking `Parameters` types and `ODE` objects. Basically `id=0` just means that we can use the default `Parameters` object.

We are going to construct our ODE method with fourth-order derivative operators, so we need at least two ghost points at the boundaries to ensure that we get fourth-order accuracy. The `if` statement guarantees that this is the case and spits out an error promising that the solution will behave incorrectly.

Next, we need to set the domain and solver for our `ODE`. This should always be done in the constructor so that memory is allocated properly.

We then build a `Parameters` object for our `ODE`. The default `Parameters` object is very simple and basically just stores information about what sort of `Interpolator` should be used, (which is not important for our particularly simple example), but a custom `Parameters` object could also contain information about what order derivative operators to use, initial conditions, and so forth.

The last line needs a little bit of explanation. Whenever we assign a `Domain` object to our `ODE`, we need to allocate memory for every single variable at every single point. If this is done for a multi-stage `Solver`, such as `RK4`, we also need to allocate this same amount of memory for each stage. Therefore, any time the `Domain` or `Solver` objects are changed, `ODE` automatically calls `reallocateData()`. When an `ODE` object is destroyed, it automatically deallocates this memory. However, the constructor doesn't automatically call this routine because we assign the `Domain` and `Solver` manually rather than through the `setDomain()` and `setSolver()` methods, therefore we must explicitly call `reallocateData()`.

The destructor doesn't need to do much because most of the heavy lifting is handled by `ODE`. Therefore, the only thing we need to do is call `delete` on the `Parameters` object we created.

The next piece we need to implement is the `rhs()` function:

```
1         void Wave::rhs(const Grid& grid, double **u, double **↩
              dudt){
2           // Go ahead and define some stuff we will need.
3           double stencil3[3] = {0.0, 0.0, 0.0};
4           double stencil5[5] = {0.0, 0.0, 0.0, 0.0, 0.0};
5           double dx = grid.getSpacing();
6           int shp = grid.getSize();
7
8           // Calculate the left boundary. We switch to a ↩
                different operator on the boundaries, which should
9           // just be ghost points that will be overwritten, ↩
                anyway.
10          // Leftmost point.
11          dudt[U_PHI][0] = u[U_PI][0];
12          stencil3[0] = u[U_CHI][0];
13          stencil3[1] = u[U_CHI][1];
14          stencil3[2] = u[U_CHI][2];
15          dudt[U_PI][0] = operators::dx_2off(stencil3, dx);
16          stencil3[0] = u[U_PI][0];
17          stencil3[1] = u[U_PI][1];
18          stencil3[2] = u[U_PI][2];
19          dudt[U_CHI][0] = operators::dx_2off(stencil3, dx);
20
21          // Second leftmost point.
22          dudt[U_PHI][1] = u[U_PI][1];
23          stencil3[0] = u[U_CHI][0];
24          stencil3[1] = u[U_CHI][1];
25          stencil3[2] = u[U_CHI][2];
26          dudt[U_PI][1] = operators::dx_2(stencil3, dx);
27          stencil3[0] = u[U_PI][0];
28          stencil3[1] = u[U_PI][1];
29          stencil3[2] = u[U_PI][2];
```

```
30            dudt[U_CHI][1] = operators::dx_2(stencil3, dx);
31
32            // Now set all the interior points.
33            for(int i = 2; i < shp - 2; i++){
34              dudt[U_PHI][i] = u[U_PI][i];
35
36              for(int j = 0; j < 5; j++){
37                stencil5[j] = u[U_CHI][i - 2 + j];
38              }
39              dudt[U_PI][i] = operators::dx_4(stencil5, dx);
40
41              for(int j = 0; j < 5; j++){
42                stencil5[j] = u[U_PI][i - 2 + j];
43              }
44              dudt[U_CHI][i] = operators::dx_4(stencil5, dx);
45            }
46
47            // Second rightmost point.
48            dudt[U_PHI][shp - 2] = u[U_PI][shp - 2];
49            stencil3[0] = u[U_CHI][shp - 3];
50            stencil3[1] = u[U_CHI][shp - 2];
51            stencil3[2] = u[U_CHI][shp - 1];
52            dudt[U_PI][shp - 2] = operators::dx_2(stencil3, dx);
53            stencil3[0] = u[U_PI][shp - 3];
54            stencil3[1] = u[U_PI][shp - 2];
55            stencil3[2] = u[U_PI][shp - 1];
56            dudt[U_CHI][shp - 2] = operators::dx_2(stencil3, dx);
57
58            // Rightmost point.
59            dudt[U_PHI][shp - 1] = u[U_PI][shp - 1];
60            stencil3[0] = u[U_CHI][shp - 3];
61            stencil3[1] = u[U_CHI][shp - 2];
62            stencil3[2] = u[U_CHI][shp - 1];
63            dudt[U_PI][shp - 1] = operators::dx_2off(stencil3, dx);
64            stencil3[0] = u[U_PI][shp - 3];
65            stencil3[1] = u[U_PI][shp - 2];
66            stencil3[2] = u[U_PI][shp - 1];
67            dudt[U_CHI][shp - 1] = operators::dx_2off(stencil3, dx);
68          }
```

This piece of code is somewhat long, but it's quite straightforward. Basically, we manually set the left and right boundaries using second-order derivative operators, which require a 3-point stencil, and all the interior points use a full fourth-order operator with a 5-point stencil. Every derivative operator takes an appropriately sized stencil and spatial interval.

After setting up the righthand side, we still need to define some boundary conditions. We're assuming a Neumann boundary, so this is easy to set up.

```
1           void Wave::applyBoundaries(bool intermediate){
2             unsigned int nb = domain->getGhostPoints();
3             // Grab the data at the leftmost grid and the rightmost↩
                   grid.
4             auto left_it = data.begin();
5             auto right_it = --data.end();
6
```

```
7           double **left;
8           double **right;
9
10          if(!intermediate){
11            left = left_it->getData();
12            right = right_it->getData();
13          }
14          else{
15            left = left_it->getIntermediateData();
16            right = right_it->getIntermediateData();
17          }
18          unsigned int nr = right_it->getGrid().getSize();
19
20          // Apply Neumann boundary condition.
21          for(unsigned int i = 0; i < nb; i++){
22            left[U_PHI][i] = left[U_PHI][nb];
23            left[U_PI][i] = left[U_PI][nb];
24            left[U_CHI][i] = 0.0;
25
26            right[U_PHI][nr - 1 - i] = right[U_PHI][nr - 1 - i];
27            right[U_PI][nr - 1 - i] = right[U_PI][nr - 1 - i];
28            right[U_CHI][nr - 1 - i] = 0;
29          }
30        }
```

This is a great time to explain how spatial data is stored in OOPS. Every `Domain` contains one or more `Grid` objects, which are stored in a `std::treeset` and sorted from left to right. Every `ODE` object maintains a set of `SolverData` objects, each of which are assigned a specific grid, and are also sorted from left to right. This makes it possible to store multiple grids on a single domain, thus allowing for different-sized grids in different regions of the solution. Hypothetically, this allows for adaptive mesh refinement, although that feature has not been implemented yet.

In any case, we need to make sure that we retrieve the leftmost and rightmost grids on the domain, because those will be the ones containing the physical boundaries. If we're using a multi-stage `Solver` object, the intermediate flag tells us whether this data needs to come from the intermediate data stored between stages or the original data from the beginning of the time step. After that, applying the boundary condition itself is nearly trivial. Because we have a Neumann boundary, we set $\chi = \partial_x \phi = 0$, and we enforce this same condition in in ϕ and π by copying the physical boundary into the ghost points.

The last piece for this file is setting the initial conditions. For simplicity, we'll assume a Gaussian centered around $x = 0.5$. The code looks like this:

```
1         void Wave::initData(){
2           // The center of our Gaussian.
3           double x0 = 0.5;
4
5           // Loop through every grid and start assigning points.
6           for(auto it = data.begin(); it != data.end(); ++it){
7             const double *x = it->getGrid().getPoints();
8             unsigned int nx = it->getGrid().getSize();
```

```
            double **u = it->getData();
            for(unsigned int i = 0; i < nx; i++){
              double val = std::exp(-(x[i] - x0)*(x[i] - x0)*64.0);
              u[U_PHI][i] = val;
              u[U_PI][i] = 0.0;
              u[U_CHI][i] = -128.0*(x[i] - x0)*val;
            }
          }
        }
```

After our discussion on boundary conditions, the purpose of the loop should be more straightforward: we need to loop over every grid on our domain so we can set each of their points fit along a Gaussian.

2.3.3 main.cpp

We're in the home stretch! With our ODE set up, all that's left is our `main()` function. Start by making a new file in the `src` directory called `main.cpp`.

```
#include <domain.h>
#include <grid.h>
#include <rk4.h>
#include <cmath>
#include <cstdio>
#include <wave.h>
#include <polynomialinterpolator.h>

int main(int argc, char* argv[]){
  // Construct our domain and a grid to fit on it.
  Domain domain = Domain();
  int N = 101;
  domain.addGrid(domain.getBounds(), N);

  // Set up our ODE system.
  RK4 rk4 = RK4();
  PolynomialInterpolator interpolator = ↩
      PolynomialInterpolator(4);
  Wave ode = Wave(domain, rk4);
  ode.setInterpolator(&interpolator);
  ode.initData();

  double ti = 0.0;
  double tf = 5.0;
  doublt dt = domain.getCFL()*(--domain.getGrids().end())->↩
      getSpacing();
  unsigned int M = (tf - ti)/dt;
  ode.dump_csv("phi00000.csv", 0, 0);
  for(unsigned int i = 0; i < M; i++){
    double t = (i + 1)*dt;
    ode.evolveStep(dt);

    char buffer[12];
    sprintf(buffer, "phi%05d.csv",i+1);
    ode.dump_csv(buffer, t, 0);
  }
```

```
35
36          return 0;
37      }
```

Inside the main function, we can see the procedure for constructing a `Domain`. When we create a new `Domain` object, it is generated with some defaults. The boundaries are automatically defined at $x = 0$ and $x = 1.0$, the number of ghost points is set to 3, and we have a Courant-Friedrichs-Lewy (CFL) factor of 0.5. This helps us set the time step to something guaranteed to be stable for our chosen spatial interval, although in practice this depends a lot on the particular equation we're solving. We can add a `Grid` to the `Domain` with the method `addGrid(double bounds[2], unsigned int n)`, where `bounds` defines the spatial location of the `Grid` (which, in this case, is just the entire domain) and the number of physical points for the `Grid`. Now is a good time to note that the number of actual points on the grid is going to be `n + 2*nghosts`, where `nghosts` is the number of ghost points, which will extend slightly beyond the region specified by `bounds[]`.

The next step is constructing our `Wave ODE` object. We first start by picking a `Solver` for the time integration, which is `RK4` in this case. We then construct an `Interpolator` object because it's required by the `ODE` class to transfer data between different-sized `Grid` objects, although this functionality is not important for our purposes. We then build our `Wave` object, assign the new `Interpolator`, and set the initial data.

Finally, we need to run our main loop. We calculate our time step, the number of total steps (because calculating `t` from the number of steps is less prone to numerical error than adding `dt` over and over), and run `evolveStep(dt)`. We also dump our data to a `.csv` file every step so we can look at the results when we're done.

2.3.4 Compiling the Project

By the end of this, you're asking, "We're done, right?"

The answer to that question is yes and no. We're done writing C++, but we still have one more step before we can compile our code and run the project. So, navigate to the OOPS base directory and open up `CMakeLists.txt`. Append the following to the end of the file:

```
1       set(WAVE_INCLUDE_FILES
2           Wave/include/wave.h
3          )
4       set(TEST_SOURCE_FILES
5           Wave/src/wave.cpp
6           Wave/src/main.cpp
7          )
8
9       set(SOURCE_FILES \${GRID_SOURCE_FILES} \${↩
            GRID_INCLUDE_FILES} \${TEST_SOURCE_FILES} \${↩
            WAVE_INCLUDE_FILES})
```

```
10          add_executable(Wave \${SOURCE_FILES})
11          target_link_libraries(Wave \${EXTRA_LIBS})
12          target_include_directories(Wave PUBLIC \${↩
                CMAKE_CURRENT_SOURCE_DIR}/include
13                                      \${CMAKE_CURRENT_SOURCE_DIR}/↩
                                            Wave/include \${↩
                                            PROJECT_SOURCE_DIR})
```

After that, rerun CMake, compile, and you should have an executable, `Wave`, which will solve your wave equation when you run it.

2.3.5 Afterthoughts

By this time, you're saying, "You said this would be simple! Why did it take so long? This seems like overkill for such a simple problem."

You're probably right. If all you're trying to do is solve the wave equation, you really don't need all this machinery. But let's start by talking about all the things you *didn't* have to do. You *didn't* have to write your own numerical integrator, first of all. You just loaded up the standard `RK4` integrator, attached it to the `Wave` object, and you were good to go. You also *didn't* have to worry about discretizing the `Domain` yourself. This isn't a terribly difficult process if all you need is a single uniform grid, but it takes out one more failing point in your code. You didn't have to write your own derivative operators. Again, these aren't difficult, but they do take time, especially if you want something more than the traditional second-order centered finite-difference operators. You didn't have to worry about boundary conditions getting applied correctly between the different stages of the numerical integrator, either; you just overloaded `applyBoundaries`, wrote out the boundaries you wanted, and let OOPS do the rest for you.

Now let's talk about what you *can* do thanks to OOPS. By just changing a few lines of code in your main function, you could change your problem from a single grid to three or four grids, perhaps to give you good resolution in the middle where the peak of the Gaussian is located and less resolution toward the edges where the data is more flat. OOPS automatically handles the data transfer between neighboring grids, including interpolating between different-sized grids (so long as they're a multiple of two apart). If the `RK4` integrator isn't good enough for your problem, you can write a new integrator using the `Solver` base class with just three functions (two of which are nearly identical). It will automatically work with all the rest of the OOPS machinery. If you're working on a complicated problem that requires some extra steps in the evolution method (`ODE::evolveStep()`), you can do that, and it's still easy because OOPS has encapsulated everything into a convenient framework for you.