OOPS Documentation

Jacob Fields

May 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 modified 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 Prior Preparation

OOPS tries to be a simple tool for scientific computing, but the documentation is neither a guide to scientific computing nor a guide to programming in C++. Therefore, the tutorials below do assume a basic knowledge of numerical methods (primarily how to solve ODEs and PDEs using finite-differencing techniques) and C++. No prior experience with CMake (for compiling) or JSON (for writing setup files) is assumed, and more complicated subjects like C++ templates are either ignored or hidden behind the scenes.

2.3 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.

2.4 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 Neumann (free) 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.

2.4.1 `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`.

2.4.2 `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;

  reallocateData();
}

// Destructor
Wave::~Wave(){
}
```

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 if it is not.

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.

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, we leave it empty.

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

```
void Wave::rhs(const Grid& grid, double **u, double **dudt){
  // Go ahead and define some stuff we will need.
  double stencil3[3] = {0.0, 0.0, 0.0};
  double stencil5[5] = {0.0, 0.0, 0.0, 0.0, 0.0};
  double dx = grid.getSpacing();
  int shp = grid.getSize();

  // Calculate the left boundary. We switch to a different ↩
      operator on the boundaries, which should
  // just be ghost points that will be overwritten, anyway.
  // Leftmost point.
  dudt[U_PHI][0] = u[U_PI][0];
  stencil3[0] = u[U_CHI][0];
  stencil3[1] = u[U_CHI][1];
  stencil3[2] = u[U_CHI][2];
  dudt[U_PI][0] = operators::dx_2off(stencil3, dx);
  stencil3[0] = u[U_PI][0];
  stencil3[1] = u[U_PI][1];
  stencil3[2] = u[U_PI][2];
  dudt[U_CHI][0] = operators::dx_2off(stencil3, dx);

  // Second leftmost point.
  dudt[U_PHI][1] = u[U_PI][1];
  stencil3[0] = u[U_CHI][0];
  stencil3[1] = u[U_CHI][1];
  stencil3[2] = u[U_CHI][2];
  dudt[U_PI][1] = operators::dx_2(stencil3, dx);
  stencil3[0] = u[U_PI][0];
  stencil3[1] = u[U_PI][1];
  stencil3[2] = u[U_PI][2];
  dudt[U_CHI][1] = operators::dx_2(stencil3, dx);

  // Now set all the interior points.
  for(int i = 2; i < shp - 2; i++){
    dudt[U_PHI][i] = u[U_PI][i];

    for(int j = 0; j < 5; j++){
      stencil5[j] = u[U_CHI][i - 2 + j];
    }
    dudt[U_PI][i] = operators::dx_4(stencil5, dx);

    for(int j = 0; j < 5; j++){
      stencil5[j] = u[U_PI][i - 2 + j];
    }
    dudt[U_CHI][i] = operators::dx_4(stencil5, dx);
  }

  // Second rightmost point.
  dudt[U_PHI][shp - 2] = u[U_PI][shp - 2];
  stencil3[0] = u[U_CHI][shp - 3];
  stencil3[1] = u[U_CHI][shp - 2];
  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[2] = u[U_CHI][shp - 3];
61    stencil3[1] = u[U_CHI][shp - 2];
62    stencil3[0] = u[U_CHI][shp - 1];
63    dudt[U_PI][shp - 1] = operators::dx_2off(stencil3, dx);
64    stencil3[2] = u[U_PI][shp - 3];
65    stencil3[1] = u[U_PI][shp - 2];
66    stencil3[0] = u[U_PI][shp - 1];
67    dudt[U_CHI][shp - 1] = operators::dx_2off(stencil3, dx);
68  }
```

Before we go any further, we need to talk about spatial discretization in OOPS. The `Domain` object we attached to `Wave` earlier has one or more `Grid` objects attached to it. The `Domain` describes the physical bounds of the problem and includes some details about discretization that all `Grid` objects need to follow. One of these is ghost points. All `Grid` objects include ghost points, which are nonphysical points extending past the bounds reported by each `Grid` object. These are required for transferring data between different `Grid` objects, but they are also used in some numerical schemes (such as those common in fluid dynamics) for applying boundary conditions. Strictly speaking, these ghost points do not *need* to be set in the righthand side routine, but they can be.

Moving forward to the code itself, it is somewhat long, but it should be very straightforward. The left and right computational (not necessarily physical) boundaries are set with second-order derivative operators, and which require a 3-point stencil, and all the interior points use a full fourth-order derivative operator with a 5-point stencil. Every derivative operator takes an appropriately sized stencil and spatial interval. An important point to notice here is that the points on the right boundary are loaded into the derivative stencil backwards; essentially, we are using a forward difference operator rather than defining a unique backward difference operator.

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();
```

```
    right = right_it->getData();
  }
  else{
    left = left_it->getIntermediateData();
    right = right_it->getIntermediateData();
  }
  unsigned int nr = right_it->getGrid().getSize();

  // Apply Neumann boundary condition.
  for(unsigned int i = 0; i < nb; i++){
    left[U_PHI][i] = left[U_PHI][nb];
    left[U_PI][i] = left[U_PI][nb];
    left[U_CHI][i] = 0.0;

    right[U_PHI][nr - 1 - i] = right[U_PHI][nr - nb - 1];
    right[U_PI][nr - 1 - i] = right[U_PI][nr - nb - 1];
    right[U_CHI][nr - 1 - i] = 0.0;
  }
}
```

This is a great time to explain how spatial data is stored in OOPS. As mentioned previously, every `Domain` contains one or more `Grid` objects. These are stored in an `std::set` 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 could also allow for adaptive mesh refinement, although that feature has not been implemented.

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.

By this point, some of you are probably asking, "Wait a minute—can I do boundaries without ghost point? How does that work?" The `applyBoundaries()` function was designed with fluid-style boundaries in mind, which are really Dirichlet boundaries cleverly designed to simulate other boundaries. More traditional PDEs can exclude the `applyBoundaries()` function altogether and set their boundaries in the `rhs()` if they so choose.

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:

```
void Wave::initData(){
  // The center of our Gaussian.
  double x0 = 0.5;

  // Loop through every grid and start assigning points.
```

```
  for(auto it = data.begin(); it != data.end(); ++it){
    const double *x = it->getGrid().getPoints();
    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.

2.4.3 `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;
  double bounds[2] = {0.0};
  bounds[0] = domain.getBounds()[0];
  bounds[1] = domain.getBounds()[1];
  domain.addGrid(bounds, 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().begin())->↩
      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;
```

```
32      ode.evolveStep(dt);
33
34      char buffer[12];
35      sprintf(buffer, "phi%05d.csv",i+1);
36      ode.dump_csv(buffer, t, 0);
37    }
38
39    return 0;
40  }
```

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.

2.4.4 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 your main project directory (`OOPS/WaveEquation`), then create a new file called `CMakeLists.txt` and add the following:

```
1 cmake_minimum_required(VERSION 3.0)
2 project(WaveEquation)
3
4 set(WAVE_INCLUDE_FILES
5     include/wave.h
```

```
    )
set(WAVE_SOURCE_FILES
    src/wave.cpp
    src/main.cpp
    )

set(SOURCE_FILES ${WAVE_INCLUDE_FILES} ${WAVE_SOURCE_FILES})
add_executable(Wave ${SOURCE_FILES})
target_include_directories(Wave PRIVATE ${↩
    CMAKE_CURRENT_SOURCE_DIR}/include)
target_include_directories(Wave PRIVATE ${CMAKE_SOURCE_DIR}/↩
    include)
target_link_libraries(Wave oops ${EXTRA_LIBS})
```

After that, rerun CMake, compile, and you should have an executable, `Wave`, in the directory `OOPS/build/WaveEquation`, which will solve your wave equation when you run it.

2.4.5 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 with the `doAfter...()` methods.

2.5 Adding Parameters

Right now, everything is hardcoded. If we wanted to try a different set of initial conditions, modify the number of `Grid` points, or change the `Domain`, we would have to recompile our code completely. This quickly becomes cumbersome as projects grow in size. An alternative would be reading in a file of parameters off the drive. To do so, we can use the `genParams.py` script to automatically generate `Parameters` and `ParamParser` classes when we compile the code.

2.5.1 Creating a Setup File

The first step is to make a setup file. In the `WaveEquation` directory, add a new directory called `scripts`. Inside `scripts`, make a new file called `wave.json`. JSON is a data format originally intended for storing JavaScript objects, but its simple structure and wide use in web programming (courtesy of JavaScript) means that many languages, including Python, can read and write JSON files. Because JSON files are stored as plain text, so can we. Therefore, inside `wave.json`, add the following code:

```
{
  "Wave":{
    "name":"Wave",
    "id":1,
    "members":[
      {
        "name":"GridPoints",
        "type":"int",
        "min":7,
        "max":128000,
        "default":101
      },
      {
        "name":"GaussianSigma",
        "type":"double",
        "min":1e-3,
        "max":1e6,
        "default":0.125
      }
    ]
  }
}
```

JSON stores all data as `"key":value` pairs, where `value` can be a string enclosed in quotes, a number, an array of comma-separated values (enclosed by square brackets, `[]`), or an object (denoted by parentheses). Therefore, this file declares an object `Wave` which has the variables `"name"`, `"id"`, and `"members"`. The `"name"` variable dictates the prefix that will be used for the custom `Parameters` and `ParamParser` objects we'll use to read in the parameters file we'll write later. The `"id"` variable is a sort of primitive type-checking to check which `ParamParser`, `Parameters`, `ODE` objects are compatible with one another. It's barely used at this point and will likely be deprecated in a future release, but

it's currently mandatory for the `genParams.py` script. The `"members"` variable is an array of objects, each one describing a different parameter.

Inside `"members"`, every object has a `"name"`, `"type"`, and `"default"`. The `"name"` field indicates how this parameter will be named in the code. For example, we will access `"GridPoints"` in the code with a function called `getGridPoints()`. The `"default"` field indicates the value that parameter should be initialized to within a `Parameters` object. The third variable, `"type"`, can be `"int"`, `"double"`, `"string"`, or `"enum"`, and tells `genParams.py` what the C++ type should be for this parameter. String objects are the easiest, as they require no additional parameters. The numerical types, `"double"` and `"int"`, require additional `"min"` and `"max"` fields to describe permissible parameter values. In this case, the `"GridPoints"` parameter will only be allowed to take values between `7` and `128000`, and `"GaussianSigma"` must lie between 10^{-3} and 10^{6}. We will discuss `"enum"` objects in more depth later on.

Next, navigate to `OOPS/scripts` run the following command:

```
python3 genParams.py ../WaveEquation/scripts/wave.json
```

Make sure that no error messages occur, then check that `OOPS/WaveEquation/include` contains `waveparameters.h` and `waveparser.h`. If that's the case, look inside `OOPS/WaveEquation/src` and verify that `waveparser.cpp` is there.

2.5.2 Setting up the Compiler

At this point, you *could* run `genParams.py` every single time you change `wave.json`, but forgetting to update your generated code is likely to become a common source of frustration and confusion as your project grows in size. It would be a lot more convenient to have `genParams.py` run whenever you compile your code. Inside the `CMakeLists.txt` file inside `OOPS/WaveEquation`, add the following lines before `set(WAVE_INCLUDE_FILES...)`:

```
set(PARAM_SRC ${CMAKE_CURRENT_SOURCE_DIR}/src/waveparser.cpp)
set(PARAM_INC
    ${CMAKE_CURRENT_SOURCE_DIR}/include/waveparameters.h
    ${CMAKE_CURRENT_SOURCE_DIR}/include/waveparser.h
   )
set(SETUP_SRC ${CMAKE_CURRENT_SOURCE_DIR}/scripts/wave.json)

add_custom_command(
  OUTPUT ${PARAM_INC}
         ${PARAM_SRC}
  DEPENDS ${SETUP_SRC}
  COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_SOURCE_DIR}/scripts/↩
      genParams.py ${SETUP_SRC}
  COMMENT "Generating custom Parameters files"
  WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
  VERBATIM USES_TERMINAL
)
```

Essentially, we declare three new variables: `${PARAM_SRC}` contains the source files we're generating, `${PARAM_INC}` contains a list of header files being generated, and `${SETUP_}` contains a list of JSON setup scripts. Following that, we tell CMake to execute a custom command, which, in this case, is running `genParams.py` on `${SETUP_SRC}`.

Once this is done, we need to tell CMake to link `Wave` to the generated source files. The easiest way to do this is to modify the line `set(SOURCE_FILES...)` to read

```
set(SOURCE_FILES ${WAVE_INCLUDE_FILES} ${WAVE_SOURCE_FILES} ${↩
    PARAM_INC} ${PARAM_SRC})
```

If we wanted to add multiple setup scripts, or if our setup script contained multiple parameter sets, we could just modify `${PARAM_INC}`, `${PARAM_SRC}`, and `${SETUP_SRC}` to reflect the additional files. At this point, it's a good idea to delete `waveparameters.h`, `waveparser.h`, and `waveparser.cpp` in their respective directories, try compiling the code, and ensure that everything gets generated correctly.

2.5.3 Using Parameters in OOPS

At this point, we've discussed how to generate some code for a custom `Parameters` class when we compile our code, but we still haven't actually used them. First, let's go to `wave.h` and make some tweaks. Add the following code into their respective sections:

```
...
#include <waveparameters.h>
...
class Wave : public ODE {
  private:
  ...
    WaveParameters *params;
  public:
  ...
    void setParameters(WaveParameters *p);
    WaveParameters* getParameters();
};

#endif
```

This section is fairly straightforward: we need to maintain a pointer to our `WaveParameters` object, and so we also need a way to set it and retrieve it.

Next up is the `wave.cpp` file. The first change we'll make is to the constructor. Strictly speaking, this step is not necessary, but it is good practice. We need to initialize `params` to `nullptr`, otherwise its behavior is undefined.

```
Wave::Wave(...{
```

```
  ...
  params = nullptr;
}
```

We initialize `params` to `nullptr`, which guarantees that trying to use it without first setting it will result in a segmentation fault. Let's also go ahead and define `getParameters()` and `setParameters()`:

```
void Wave::setParameters(WaveParameters *p){
  params = p;
}

WaveParameters* Wave::getParameters(){
  return params;
}
```

Next, we need to modify `initData()` so that we can use the `GaussianSigma` parameter we defined:

```
void Wave::initData(){
  double sigma = params->getGaussianSigma();
  ...
  for(...){
    ...
    for(...){
      double val = std::exp(-(x[i] - x0)*(x[i] - x0)/(sigma*sigma))↩
          ;
      ...
      u[U_CHI][i] = -2.0/(sigma*sigma)*(x[i] - x0)*val;
    }
  }

}
```

Every parameter added to the JSON setup script will have both a getter and a setter in the custom generated `Parameters` class. In this case, because we called our function "`GaussianSigma`", we have functions `getGaussianSigma()` and `setGaussianSigma(double)`. Therefore, we can simply grab `GaussianSigma` from `params` and substitute that into our Gaussian initial data.

Now, if you try to run this code now, you'll get a segmentation fault. We haven't actually loaded or assigned a `WaveParameters` object to `Wave` yet, which will be our next step. We need to modify `main.cpp` as follows:

```
...
#include <iostream>
#include <waveparameters.h>
#include <waveparser.h>

int main(int argc, char* argv[]){
  if(argc < 2){
    std::cout << "Usage: ./Wave <parameter file>\n";
```

```
9       return 0;
10    }
11
12    WaveParameters params;
13    WaveParser parser;
14    parser.updateParameters(argv[1], &params);
15
16    // Construct our domain and a grid to fit on it.
17    Domain domain = Domain();
18    int N = params.getGridPoints();
19    ...
20
21    ...
22    ode.setParameters(&params);
23    ode.initData();
24    ...
25  }
```

The first few lines tell the code to expect a parameter file along with the executable, so if it's not present, it tells the user how to run the program properly and quits. If there is a parameter file, we can go ahead and try to run the program. We create `WaveParameters` and `WaveParser` objects, then we use the `WaveParser` to read in our parameters and update `WaveParameters` accordingly. The next change is setting the resolution of our `Grid` using the supplied parameter instead of hard-coding it to `101`. Finally, the last change is supplying the `WaveParameters` object to `ode` before we set the initial data.

At this point, make sure that your code compiles. It won't run yet because we haven't written a parameter file, but it should compile. Now, create a new file in the `OOPS/WaveEquation/pars` directory called `wave.par`:

```
1 [Wave]
2 GridPoints = 101
3 GaussianSigma = 0.125
```

The `[Wave]` header at the top indicates what `Parameters` object this belongs to. You can easily store multiple `Parameters` sets in a single file, which may be helpful if you want to maintain a set of parameters for your `Grid` and `Domain` setup, a different set for your `ODE`, and maybe a different set for output options.

Every parameter is specified as `parameter = value`. There is no semicolon at the end, and each new line is treated as a different parameter. Whitespace before and after parameter names and values is ignored, but everything is case-sensitive and must match the `"name"` variable specified in the JSON file to be read in correctly.

Now, let's run the code. You can either copy `wave.par` into the binary directory (`OOPS/build/WaveEquation`) or specify its relative path:

```
./Wave <relative path if necessary>/wave.par
```

Your code should run and generate a whole mess of `.csv` files, just as before.

If you examine the output using your favorite plotting program, it should look exactly the same. Here are some different things to try:

1. Go into `wave.par` and erase one or both parameter assignments (don't get rid of the `[Wave]` header). Run your code again. It should still run without an issue because `WaveParameters` will simply use some default values.

2. See what happens as you change `GridPoints`. What happens if you set it outside the acceptable range specified in `wave.json`?

3. Play around with different values of `GaussianSigma`. In particular, what happens to the quality of the solution if you make `GaussianSigma` very small?

4. Try adding some new parameters on your own. Some good examples might be `x0`, the center of the Gaussian, or the minimum and maximum bounds on the `Domain`.

2.5.4 Adding an Enumerated Parameter

It happens quite frequently that we need a parameter describing a fixed set of values, such as different kinds of initial conditions, solution methods, or particular boundary conditions. A very common but inelegant solution is to assign each value a numerical index, then provide documentation on all the different cases. However, virtually every modern programming language, C++ included, allows for enumerated types, which are far more readable. The parameter generation system also supports enumerated types. Let's consider a new set of initial conditions: a sine wave. Go into `wave.json` and add the following object inside `"members"`:

```
{
  "name":"InitialConditions",
  "type":"enum",
  "value":[
    "GAUSSIAN",
    "SINE"
  ],
  "default":"GAUSSIAN"
}
```

Rather than specifying `"min"` and `"max"` variables, we add a `"values"` variable, which is an array of strings representing possible values for the enumerator. Go ahead and compile the code again to make sure that you've added everything correctly. Now, go into `wave.cpp`, and let's modify `initData()` again:

```
void Wave::initData(){
  ...
  if(params->getInitialConditions() == WaveParameters::GAUSSIAN){
```

```
    for(...){
      ...
      for(...){
        ...
      }
    }
  }
  else if(params->getInitialConditions() == WaveParameters::SINE){
    for(...){
      ...
      for(...){
        double pi = 3.1415926;
        u[U_PHI][i] = std::sin(x[i]*pi);
        u[U_PI ][i] = 0.0;
        u[U_CHI][i] = std::cos(x[i]*pi);
      }
    }
  }
}
```

After compiling the code to make sure it works, the last step is to add the initial condition into the `wave.par` parameter file:

```
[Wave]
GridPoints = 101
GaussianSigma = 0.125
InitialConditions = SINE
```

Now rerun the code. It should no longer evolve a Gaussian pulse, but rather a sine wave.

3 Documentation

3.1 `Domain` Class

The `Domain` class defines the physical region a PDE should be solved on. It acts as a factory for `Grid` objects and contains the necessary parameters to construct them appropriately.

Public Methods

`Domain()` The constructor for a `Domain` object. By default, it sets the number of ghost points to three, the boundaries to $[0, 1]$, and the CFL factor to 0.5.

`~Domain()` The destructor for a `Domain` object. It automatically clears all grids attached to the `Domain`.

`void setCFL(double cfl)` Set the CFL factor. At the moment, this doesn't affect anything in the code itself. Future releases may use this to calculate a recommended time step.

`double getCFL() const` Get the CFL factor.

`void setBounds(double bounds[2])` Set the bounds for the domain with an ordered pair. **This will also clear any `Grid` objects still on the `Domain`.**

`const double* getBounds() const` Get the bounds for the `Domain` as an ordered pair.

`void setGhostPoints(unsigned int n)` Set the number of ghost points to use while constructing new `Grid` objects. **This will clear any `Grid` objects still on the `Domain`.**

`unsigned int getGhostPoints() const` Get the number of ghost points used while constructing new `Grid` objects.

`std::set<Grid>& getGrids()` Get a `set` containing all the different `Grid` objects currently on the `Domain`.

`Result addGrid(double bounds[2], unsigned int n)` Using an ordered pair `bounds[]`, add a new `Grid` with `n` uniformly spaced points to the `Domain`. It returns `SUCCESS` if it succeeded and `BAD_ALLOC` if it failed.

`void clearGrids()` Clear all `Grid` objects currently on the `Domain`.

3.2 `Grid` Class

The `Grid` class defines a 1d grid with uniform spacing. These are the building blocks for basically all calculations in OOPS.

Public Methods

`Grid(const double bounds[2], unsigned int n, unsigned int nghosts)` The constructor for a new `Grid` with `n` points from `bounds[0]` to `bounds[1]`, plus an additional `nghosts` ghost points on either end.

`Grid(const Grid& other)` The copy construct for `Grid`. Because memory is allocated and deallocated in every `Grid` object, this performs a deep copy.

~`Grid()` This is the destructor for the `Grid` class. It frees all the memory allocated for the `Grid` points.

`Result rebuildGrid(const double bounds[2], unsigned int n, unsigned int nghosts)` A function to rebuild a `Grid`, with the arguments being the same as the constructor, if something needs to change. It returns `SUCCESS` on success and `BAD_ALLOC` if it fails.

`const double* getPoints() const` Get the array containing the physical location of every grid point.

`const unsigned int getSize() const` Get the number of points, including ghost points, currently on the grid.

`const double getSpacing() const` Get the space interval between points.

`const double* getBounds() const` Get the bounds, excluding ghost points, of the `Grid` as an ordered pair.

`bool operator < (const Grid& g) const` Compare two `Grid` objects to each other. If the right bound for the `Grid` on the left side of the operator is less than the left bound for the `Grid` on the right side of the operator, this returns true. If the two `Grid` objects overlap at all, this will always return false, even if you flip the arguments.

`Boundary whichNeighbor(const Grid& g) const` If two `Grid` objects share a boundary, this will return `Boundary::LEFT` or `Boundary::RIGHT`. Otherwise, it returns `Boundary::NONE`.

3.3 `Solver` Class

The `Solver` class is an interface for numerical integrators. As such, it cannot be instantiated, but any numerical integrator, such as the included `RK4` integrator, must be a descendant of the `Solver` class to work properly with the rest of the OOPS machinery. This particular class may be rewritten somewhat in the future to make it more consistent with the rest of OOPS.

Protected Fields

`const int nStages` The number of stages for the `Solver`, which is specified in the constructor.

Public Methods

`Solver(const int n)` A basic constructor included to set the number of stages for a `Solver`. **All subclasses must call this constructor in their initialization list.**

`int getNStages() const` Get the number of stages for the `Solver`.

Unimplemented Public Methods

`virtual Result setStageTime(double srcTime, double& destTime, double dt, unsigned int stage)` Using `srcTime` and `dt`, determine what the proper time for `stage` is and store the result in `destTime`. Ideally, this should be implemented to return `SUCCESS` unless an invalid stage is requested, then it should return `INVALID_STAGE`. This method is called before `calcStage()`.

`virtual Result calcStage(ODE *ode, double *data0[], double *dataint[], double *dest[], const Grid& grid, double dt, unsigned int stage)`
This is the method currently used by the OOPS framework for calculating each stage of a `Solver` object. The first argument is the `ODE`, which contains the righthand-side function `Solver` should use. The argument `data0` is a 2d array containing the current solution, `dataint` is a 2d array for the intermediate solution, and `dest` is the destination for the work data/righthand side calculated during this particular stage. The `grid` argument specifies what `Grid` the solver is operating on, `dt` is the size of the timestep, and `stage` is the stage for a multi-stage solver like `RK4`. Although `ODE` technically contains all the data and the `Grid`, it must be passed in because `calcStage` occurs in the middle of a loop over all the `Grid` objects. Be aware that this is likely to change in order to make the `Solver` class more consistent with the rest of OOPS.

`virtual Result combineStages(double **data[], double *dest[], const Grid& grid, double dt, const std::vector<unsigned int> evolutionIndices)`
This function combines all the stages together. The `data[]` array is organized by `[STAGE][VAR][INDEX]` and contains all the work data from the previous calculations. The `dest[]` array contains the final result of the calculation. The `Grid` for this calculation is supplied, as is the time step and the number of variables for the equation. Because some systems, such as fluids, may evolve fewer equations than they actually contain, the `evolutionIndices` argument contains an `std::vector` that describes all the equations which should be evolved.

3.3.1 `RK4` Child Class

An implementation of the classic 4th-order Runge-Kutta integrator. It is a four-stage solver and has no unique functions. It can be declared with its default constructor, `RK4()`.

3.4 `ODE` Class

`ODE` is an abstract class describing a generic system of differential equations. As a brief note, no `ODE` object is copyable because they store a considerable amount of dynamically allocated memory.

Protected Fields

`const unsigned int nEqs` The number of independent equations in this system.

`const unsigned int pID` The ID associated with this `ODE`. This is used for identifying compatible `Parameter` objects.

`Domain *domain` The `Domain` this `ODE` is being solved on. This is how the `ODE` accesses all of the `Grid` objects it needs to construct the `SolverData` objects.

`std::set<SolverData> data` A `set` containing all the `SolverData` objects for each `Grid`.

`Solver *solver` The `Solver` that should be used to calculate each step.

`Interpolator *interpolator` The `Interpolator` used to transfer solution data between differently sized `Grid` objects.

`max_dx` The maximum grid spacing across all `Grid` objects on the `Domain`.

`time` The current evolution time.

`std::vector<unsigned int> evolutionIndices` An `std::vector` containing the indices for the variables that should be evolved. By default, this is initialized to loop over all variables.

Protected Methods

`virtual void applyBoundaries(bool intermediate)` A function to apply fluid-style boundary conditions after data is transferred between `Grid` objects. It is empty by default, so initial value problems and PDEs which apply their boundaries through the righthand side can simply not overwrite this function.

`virtual void doAfterStage(bool intermediate)` An empty function which can be overwritten to perform specific tasks after the `Solver` stage completes but before the `Grid` exchange occurs.

`virtual void doAfterExchange(bool intermediate)` An empty function which can be overwritten to perform specific tasks after the `Grid` exchange occurs but before the boundaries are applied.

`virtual void doAfterBoundaries(bool intermediate)` An empty function which can be overwritten to perform specific tasks after the boundary conditions are applied but before the next stage is calculated.

`Result reallocateData()` A function that clears and reallocates all of the data. This may be important if the `Domain` or any of its `Grid` objects change.

`void performGridExchange()` This exchanges data between the ghost points of neighboring `Grid` objects, including taking care of interpolation.

`void exchangeGhostPoints(const SolverData &data1, const SolverData &data2)` A helper function that performs a `Grid` exchange between two specific data sets.

`void interpolateLeft(const SolverData &datal, const SolverData &datar)` A helper function for interpolation that assumes the `SolverData` to the left of the interface is finer.

`void interpolateRight(const SolverData &datal, const SolverData &datar)` A helper function for interpolation that assumes the `SolverData` to the right of the interface is finer.

`Result addAuxiliaryField(std::string name)` Add a new auxiliary field to the `ODE` object. The `reallocateData()` function must be called to allocate memory for the new field. It returns `SUCCESS` if the field was created successfully or `FIELD_EXISTS` if a field with that name already exists.

`Result removeAuxiliaryField(std::string name)` Remove an auxiliary field from the `ODE` object. Returns `SUCCESS` if the field was created successfully or `UNRECOGNIZED_FIELD` if the field does not exist.

`std::set<ODEData>* ODE::getAuxiliaryField(std::string name)` Get a pointer to the specified auxiliary field. If the field does not exist, it returns `nullptr` instead.

Public Methods

`ODE(const unsigned int n, const unsigned int id)` The constructor for `ODE`, which all subclasses must call in their initialization lists, which defines a new `ODE` object with `n` equations utilizing `Parameter` objects with the provided `id`.

`~ODE()` The destructor for the `ODE`, which clears all the data.

`Result setDomain(Domain *domain)` Set the `Domain` for the `ODE`. This will clear all existing data and reallocate it for the new `Domain`. It returns `SUCCESS` on success and an error, usually `BAD_ALLOC`, on failure.

`Result setSolver(Solver *solver)` Set the `Solver` for the `ODE`. This will clear all existing data and reallocate it for the new `Solver`. It returns `SUCCESS` on success and an error, usually `BAD_ALLOC`, on failure.

`Result setInterpolator(Interpolator *interp)` Set the interpolation method used by this `ODE`. It returns `SUCCESS` on success and an error on failure, usually `BAD_ALLOC`.

`virtual Result evolveStep(double dt)` The evolution function for a single step. It is overwriteable in case a specific use-case scenario requires more flexibility than the default evolution function provides, but this is not a typical use-case scenario.

`unsigned int getNEqs() const` Get the number of equations in the `ODE` system.

`Domain *getDomain()` Get the current `Domain` used by the `ODE`.

`void setTime(double t)` Set the current evolution time for the `ODE`.

`double getTime()` Get the current evolution time.

`void output_frame(char *name, double t, unsigned int var)` Write out the data from variable `var` labeled with time `t` to `<name>.sdf`. This requires the `bbhutil` library and must be explicitly enabled with the `USE_SDF` option during the configuration. Otherwise, a warning is printed the first time it is called and no data is saved.

`dump_csv(char *name, double t, unsigned int var)` All the data is dumped to the specified file for the corresponding variable `var` in the format (t, x, data).

Unimplemented Public Methods

`virtual void initData()` The initial conditions for the `ODE`.

`virtual void rhs(const Grid& grid, double** data, double** dudt)` The righthand-side function for this `ODE`. It takes a `Grid` object and the `data` corresponding to that `Grid`, then stores the calculated righthand-side in `dudt`.

3.5 `ODEData` Class

The `ODEData` class is a utility used by `ODE` for storing general solution data. Note that the `ODEData` copy constructor is protected, so no instance of `ODEData` can be copied directly.

Public Methods

`ODEData(unsigned int eqCount, const Grid& grid)` The constructor for a `ODEData` object. It saves a reference to the `Grid` object it should be associated with and allocates memory for `eqCount` equations on that `Grid`.

`virtual ~ODEData()` The destructor for `ODEData`. It deallocates all the memory allocated in the constructor. Since this is virtual, `ODEData` can be safely used as a base class for more complex data structures, such as the `SolverData` class below.

`double** getData() const` Get a pointer to a 2d array of data organized by `[vars][points]` representing `ODE` information on the `Grid` associated with the `ODEData` object.

`unsigned int getEqCount() const` Get the number of equations that the `ODEData` object is storing data for.

`const Grid& getGrid()` Get the `Grid` object associated with the `ODEData` object.

3.5.1 `SolverData` Child Class

The `SolverData` class is a utility used by `ODE` to store solution data for a particular `Solver`. Note that the `SolverData` copy constructor is private, so no instance of `SolverData` can be copied directly. It descends directly from the `ODEData` class, whose `double** data` field is assumed to be the primary solution.

Public Methods

`SolverData(unsigned int eqCount, unsigned int nStages, const Grid& grid)` The constructor for a `SolverData` object. It saves a reference to the `Grid` object it should be associated with and allocates memory for `eqCount` equations for `nStages` `Solver` stages.

~`SolverData()` The destructor for `SolverData`. It deallocates all the memory allocated in the constructor.

`double** getIntermediateData() const` Get a pointer to the 2d array of data organized by [vars][points] which stores the intermediate data used in a `Solver`.

`double*** getWorkData()` Get a pointer to the 3d array of data organized by [stage][vars][points]. This is used to store the various results needed from each stage of a `Solver`.

3.6 `Interpolator` Class

An `Interpolator` object provides machinery for a generic interpolation scheme. Typically this should be a centered scheme, although there is nothing preventing a subclass from doing a non-centered stencil. However, such `Interpolator` schemes should not be used to interpolate `Grid` objects.

Protected Fields

`const unsigned int nStencil` The size of the interpolation stencil.

`double *stencil` The stencil for this `Interpolator`. This is allocated dynamically when the scheme is created and destroyed when the destructor is called.

Public Methods

`Interpolator(const unsigned int n)` The constructor for this `Interpolator`, which builds a new scheme with a stencil of size `n`. This must be called in the initialization list of subclasses.

~`Interpolator` The destructor for this `Interpolator`. It releases the allocated memory.

`double* getStencil()` Get a pointer to the interpolation stencil, which will be an array of size `nStencil`.

`unsigned int getStencilSize()` Get the size of the interpolation stencil.

Unimplemented Public Methods

`virtual double interpolate()` Interpolate to calculate a point. It's generally expected that this will be a centered scheme with an even number of points in the stencil (linear, cubic, etc.), but this is by no means required if the `Interpolator` will not be used for `Grid` interpolation.

3.6.1 `PolynomialInterpolator` Child Class

An interpolator for n-point centered polynomial interpolation.

Public Methods

`PolynomialInterpolator(const unsigned int n)` Construct a new `PolynomialInterpolator` for an `n`-point polynomial interpolation scheme.

3.7 `operators` Namespace

The `operators.h` file defines a number of finite-difference operators inside the `operators` namespace. All operators unless otherwise specified assume a uniform grid spacing.

Differential Operators All differential operators take a fixed-size `double` array as a stencil and a `double` for the grid spacing.

`double dx_2(const double u[3], const double dx)` A 2nd-order centered first derivative operator.

`double dx_2off(const double u[3], const double dx)` A 2nd-order forward-difference first derivative operator. The corresponding backward-difference operator is used by entering the points in backwards.

`double dx_4(const doube u[5], const double dx)` A 4th-order centered first derivative operator.

`double dxx_2(const double u[3], const double dx)` A 2nd-order centered second derivative operator.

`double dxx_2off(const double u[4], const double dx)` A 2nd-order forward-difference second derivative operator. The corresponding backward-difference operator is used by entering the points in backwards.

`doubl dxx_4(const double u[5], const double dx)` A 4th-order centered second derivative operator.

Kreiss-Oliger Dissipation Operators All Kreiss-Oliger dissipation operators take a fixed-size `double` array as a stencil and a `double` for the grid spacing.

`double ko_dx(const double u[7], const double dx)` A (4th?)-order centered-difference operator for Kreiss-Oliger dissipation.

`double ko_dx_2(const double u[5], const double dx)` A 2nd-order centered-difference operator for Kreiss-Oliger dissipation.

`double ko_dx_off1(const double u[4], const double dx)` A (1st?)-order operator for Kreiss-Oliger dissipation at the leftmost point in the stencil.

`double ko_dx_off2(const double u[5], const double dx)` A (2nd?)-order operator for Kreiss-Oliger dissipation at the 2nd-leftmost point in the stencil.

`double ko_dx_off3(const double u[6], const double dx)` A (3rd?)-order operator for Kreiss-Oliger dissipation at the 3rd-leftmost point in the stencil.

3.8 `ParamReader` Class

The `ParamReader` class is designed to read a parameter file from the disk.

3.8.1 Parameter File Structure

Parameter files are reminiscent of (and slightly more stringent than) classic Windows `.ini` files. Each file is divided into sections denoted by square brackets, such as `[Section]`. The `#` character is a comment character, and anything past it to the end of the line will be ignored. Parameter names are denoted by a single alphanumeric phrase and are assigned to a value by an equals sign. A simple file might look like this:

```
# A parameter file for some simulation.
[Grid]
# The number of grid points to use.
nx = 64

[InitialConditions]
# Set up Gaussian initial data.
init_id = ID_GAUSSIAN
init_gaussian_sigma = 10.0
init_gaussian_A = 10.0
```

Whitespace before and after parameter and value names is ignored. Values may contain whitespace inside their names, but parameters may not. Therefore, `string_data = Some String of Data` is a valid entry, but `string data = Some String of Data` will return an `INVALID_PARAMETER` error. Each line is treated as a new parameter.

3.8.2 Reading Parameter Files

Parameter files are parsed into an `std::map` of `std::map` data structures. The first `map` contains all the parameters for a specific section, with that section name acting as the key. The second `map` contains the values for each parameter, with its name acting as the key. Therefore, to read a parameter file, you must know the section and parameter name.

Public Methods

`ParamReader()` The default constructor for a `ParamReader` object.

~`ParamReader()` The destructor for a `ParamReader` object.

`ParamResult readFile(std::string fname)` Read a file `fname` off the disk and load it into memory. It returns a `ParamResult` indicating success or failure.

`bool hasSection(std::string section)` Check if a particular section exists. Returns true if so, false if not.

`bool hasParameter(std::string section, std::string parameter)` Check if a particular parameter exists in a section. Returns true if so, false if not.

`std::string readAsString(std::string section, std::string parameter)` Read `parameter` in `section` as a string. If the parameter exists, it returns the value. Otherwise, it prints a warning to `std::cout` and returns `"NULL"`.

`double readAsDouble(std::string section, std::string parameter)` Read a parameter value as a double. If it exists and is a numerical value, it will return its value. If the parameter does not exist or is not a number, it will print a warning to `std::cout` and return `0.0`.

`double readAsInt(std::string section, std::string parameter)` Read a parameter value as an integer. If it exists and is an integer, it will return its value. If the parameter does not exist or is not an integer, it will print a warning to `std::cout` and return `0`.

`void clearData()` Clear all the data that this `ParamReader` has read in.

`ParamResult` Enumerator

`SUCCESS` The file was loaded successfully.

`BAD_FILENAME` The file does not exist or could not be opened.

`SYNTAX_ERROR` A syntax error, such as a missing `=` sign, an unclosed section bracket, or a parameter declared on the same line as a section.

`BAD_TYPE` Currently unused.

`MULTIPLE_DEFINITIONS` The parameter file contains a previously written section.

`UNSECTIONED_PARAMETER` A parameter was defined at the head of the file before a section was declared.

`INVALID_PARAMETER` A parameter name contains invalid characters.

`INVALID_VALUE` A value name contains invalid characters.

The `ParamReader` class can read in multiple files and store them, but they cannot share section names without first clearing the data. Parameters defined multiple times within a single section will be overwritten by the last defintion.

3.9 `Parameters` Class

The `Parameters` class is a simple class for representing the parameters an `ODE` object might need. While it is a concrete class and can be declared, it is most useful as a base class for more complicated `ODE` objects. It only has a few useful methods in its basic form.

Methods

`Parameters(unsigned int id) /*Protected*/` A constructor designed to build a `Parameters` object with a specific `id`. This should be used in the initialization list for a subclass, otherwise `id=0` and cannot be changed. It is protected so that it can only be used in this fashion.

`Parameters() /*Public*/` A public default constructor.

`unsigned int getId() const /*Public*/` Get the `id` for this `Parameters` object. This is useful for making sure that a `Parameters` object assigned to a specific `ODE` is actually compatible; if the `id` does not match, OOPS will not allow the assignment.

3.10 `ParamParser` Class

The `ParamParser` class is an abstract class which should be used to fill out `Parameter` objects with information collected by `ParamReader`. Except in rare circumstances, it is unlikely you will ever have to extend `ParamParser` to read your parameter files; the `genParams.py` script described below should be used for that instead.

Public Methods

`ParamParser(unsigned int id)` A constructor which sets the `id` for this `ParamParser`. It must match the `id` in the `Parameters` objects it is designed to work with.

`bool checkId(Parameters* params) const` Check if `params` has an `id` that matches this particular `ParamParser`.

`unsigned int getId() const` Get the `id` for this `ParamParser`.

Unimplemented Public Methods

`void updateParameters(std::string fname, Parameters* params)` Update `params` with the data specified in the parameter file `fname`. The `id` of `params` must match the `id` for this particular `ParamParser`.

3.11 `genParams.py` Script

OOPS also features a Python script, `genParams.py`, that will accept a `JSON` setup file and generate custom `Parameters` and `ParamParser` objects.

3.11.1 Writing Setup Files

Because Python has a wonderfully simple `JSON` parser available, the setup file for a new `Parameters` object and its parser is written in `JSON`. A sample is available in the `scripts` folder in the OOPS base directory, and it reads as follows:

```
{
  "Test":{
    "name":"Test",
    "id":1,
    "members":[
      {
        "name":"InitialConditions",
        "type":"enum",
        "value":[
          "GAUSSIAN",
          "FLAT"
        ],
        "default":"GAUSSIAN"
      },
      {
        "name":"KOSigma",
        "type":"double",
        "min":0.0,
        "max":1.0,
        "default":0.0
      }
    ]
  }
```

```
}
```

The first layer is expected to be the definition for a `Parameters` object. It must define three variables: `"name"`, `"id"`, and `"members"`.

The `"name"` variable is the head for all generated files. For example, `"name":"Test"` will generate the class `TestParameters` inside `testparameters.h`. The `"id"` field is an integer that indicates what `id` should be used for this `Parameters` object and any `ODE` object that uses it. The `"members"` field is a list of objects, each object representing a new parameter.

All `"member"` entries must define `"name"`, `"type"`, and `"default"`. The `"name"` field corresponds to the name to print into the variable. The `"type"` field indicates the C++ type for the variable. Finally, the `"default"` field indicates how `"name"` should be set inside the constructor of the new `Parameters` object. **You are responsible for ensuring that your variables and their corresponding values lead to valid C++ code.** A future version of `genParams.py` may perform some limited syntax checking to ensure that names do not include forbidden characters, but this is rather low on the list of priorities at the moment.

If `"type"` is `"enum"`, the `"value"` variable must also be defined as a list of enumerator values. Number types (e.g., `"int"` or `"double"`) will also require `"min"` and `"max"` fields for generating the parser, but they are not currently necessary.

The output `Parameters` file of the setup file described above is

```
#ifndef TEST_PARAMETERS_H
#define TEST_PARAMETERS_H

#include <parameters.h>

class TestParameters : public Parameters {
 public:
   enum InitialConditions{
     GAUSSIAN,
     FLAT,
   };

   TestParameters() : Parameters(1){
     mInitialConditions = GAUSSIAN;
     mKOSigma = 0.0;
   }

   inline void setInitialConditions(InitialConditions val){
     mInitialConditions = val;
   }

   inline InitialConditions getInitialConditions(){
     return mInitialConditions;
   }

   inline void setKOSigma(double KOSigma){
     mKOSigma = KOSigma;
```

```
28    }
29
30    inline double getKOSigma(){
31      return mKOSigma;
32    }
33
34  private:
35    InitialConditions mInitialConditions;
36    double mKOSigma;
37 };
38
39 #endif
```

As you can see, it will define a new class declaring all member variables as private with public getters and setters. The constructor will assign each variable its proper default value and set the correct id in the initialization list. Enumerators will be defined in the public section. For a small setup file like this, obviously writing a single header file isn't much work, but very complex `ODE` objects may require a `Parameters` object with dozens of different settings corresponding to different initial conditions, and you'll also have to write a parser to read all those parameters from a parameter file, check for invalid settings, and ignore unspecified parameters. It quickly becomes extremely tedious to write it by hand, and it's much simpler to generate a setup file and let `genParams.py` do all the work.

3.11.2 Running `genParams.py`

Running the script is easy:

```
python3 genParams.py <filename1.json> <filename2.json> ...
```

The script will load in each setup file one at a time and generate C++ code for each `Parameters` object specified in the JSON setup file consisting of three files: header files for `Parameters` and `ParamParser`, and a source file for `ParamParser`. It expects that your JSON setup file will be located either in the root of your project directory or within a `scripts` file in your project directory. From here, it will automatically place the header and source files in their proper directories. We highly recommend that you modify your local `CMakeLists.txt` file to run `genParams.py` automatically during the build. See Sec. 2.5.2 for details.

3.12 `plotData.py` Script

You've got a great OOPS code written, but now you need to check that your output actually works. OOPS doesn't feature any sophisticated data analysis or plotting tools (nor are any planned for the immediate future), but it does contain a simple Python script to let you look at all those `.csv` files you just generated. Copy `OOPS/scripts/plotData.py` into the directory with all your data files, and run

```
python3 plotData.py <filenames>
```

(Yes, it does support regex arguments. You will not need to manually input all the data files.) The script will scan through all the data files, look for the maximum and minimum values along the vertical axis, plot each frame, and save it to a `.png` image. For example, if your data all have names like `data*.csv`, all your plots will be saved in `plots_data/data*.png`.

This script is not intended to be a full analysis or plotting utility. If you need something more sophisticated, there are plenty of tutorials online for using Python, MATLAB, Mathematica, and other software packages to read in and manipulate `.csv` data.

The `plotData.py` script does require the Matplotlib and NumPy modules to work. These can be installed via `pip`, `conda`, or your favorite package manager.

4 Question and Answer

4.1 How do I...

I need to add a `Solver` that doesn't work like `ODE` expects. Can I do this? Sure. The `evolveStep()` method in `ODE` is virtual and can be overwritten for this purpose. Really byzantine solvers may not work well with `SolverData`, but the `ODEData` class is a more basic data structure containing only a `Grid` reference and an array of solution data, which may work better depending on the situation.

How do I plot OOPS `.csv` data? Each row has the format `(time, position, value)`. These `.csv` files can be loaded into your favorite data analysis software and manipulated that way. If you just need a quick look at your data, try using `plotData.py` in the `OOPS/scripts` directory. Instructions on using it are located in Sec. **??**.

4.2 Extending OOPS

I want to add to this project. Can I? Possibly. In most circumstances, forking the project is more appropriate. As the goal of OOPS is to be an easy-to-use PDE solver with a self-consistent design, code maintenance is a serious responsibility. Jacob, as the owner and maintainer for the project, doesn't have the time to patrol ten different people's work for code quality. Consequently, if you don't have an institutional connection to him, permission to edit the repository directly is unlikely.

If I can't edit it, can I request a feature? We can't guarantee anyone on the project has the time or interest to add new features, but feel free to make requests or suggestions.

What if I have an idea to do something easier? We really want this project to be easy to use. If you have good ideas to make things more consistent, more streamlined, or easier to use without sacrificing flexibility, don't keep it a secret. Let us know. Fork the project and show us how it will work.

4.3 Troubleshooting

4.3.1 Parameters

I keep getting segmentation faults when I try to access a parameter inside my `ODE` class. Make sure `ODE.setParameters()` is called before `ODE.initData()` or any other function that may access the `Parameters` object.

4.3.2 Python

None of the OOPS Python scripts will run! Make sure that your Python installation is up to date. OOPS should complain if you don't have at least Python 3.0 installed. If a more modern installation doesn't work, please let us know.

I'm having problems getting the `plotData.py` script to run. The `plotData.py` script requires both Matplotlib and NumPy to work. Install them using `pip`, `conda`, or another package manager. If you have them installed, try updating them. If you still encounter issues running them, please let us know.