`dgswem-v2`: Users' Guide

October 5, 2017

Contents

Chapter 1

Introduction

This document aims to orient new users about the design of `dgswem-v2`. Before beginning to discuss what this document is, we will briefly preface what it is not. Firstly, this document does not contain installation instructions. Those can be found in the `README.md` file in the root directory of this repository. Secondly, this guide is not intended as an API reference. To see API documentation, we have annotated our functions using doxygen, which can be built via

```
cd ${dgswem-v2-root}/documentation
doxygen Doxyfile
```

Rather this document aims to achieve the following 3 goals:

1. Provide insight into the design philosophy of `dgswem-v2`,
2. Document input formats and intended usage through annotated examples.
3. Provide a reference for `dgswem`-users trying to get oriented in this new code.

1.1 Problems with `dgswem`

Before discussing the features of `dgswem-v2`, we will briefly dwell on the problems with the predecessor of this code. To quote George Santayana, "Those who cannot remember the past are condemned to repeat it".

1.1.1 Difficulty of Adding Features

The first two problems outline with `dgswem` center on issues surrounding developer productivity. The first issue is centered around the difficulty of adding features. This is partially centered around limitations of Fortran, but also partially a criticism of the code bases design.

The first point I would like to highlight is the difficulty associated with implementing new features is the difficulty in maintain old ones. One poignant

example of this can be found in `src/DG_hydro_timestep.F`. The code currently supports two timestepping mechanisms: Strong Stability Preserving Runge-Kutta (SSPRK) methods and Runge-Kutta Chebychev methods. In an attempt to preserve good performance, the methods can be swapped out at compile time using compiler macros. However, the SSPRK implementation spans roughly 170 lines of code, and the RKC implementation a similar number of lines. The issue now being that any modification that happens to one timestepping scheme needs to be duplicated below. This form of code duplication can be confusing and complicates supporting the full set of features in any given combination.

The second issue is a more underlying limitation of the use of Fortran. Steven Brus' work [**?**] demonstrates the drastic performance improvements made by the introduction of curvilinear elements. Due to the varying Jacobians, separate loops are required to update curvilinear elements. This would then require that two loops be written into `dgswem`. However, it is convievable that one might be interested in implementing quadratic elements as well. Now the number of loops grow in a comninatorial manner (linear/triangles, curved/triangles, linear/quadrilateral, and curved/quadrilaterals).

1.1.2 Not Testable

As a procedural code written in a traditional fortran manner. The data of the simulation are allocated as globally accessible structs of arrays. The global scoping of the structs obfuscates the actually dependencies. Developing a unit testing framework would require meticulous teasing out of explicit dependencies for a given function, and then furthermore require convoluted tests cases to achieve reasonable code coverage. In practice, previous attempts at implementing continuous integration testing for `dgswem` have centered around implementing correctness tests. While these would at least verify if adding a change to the code base doesn't compromise the correctness of the code, these tests fail to locate the source of the error.

1.1.3 Poor Fortran/C++ Interoperability

This problem has initially arisen out of the STORM project whose objective was to implement storm surge codes using HPX, an Asynchronous runtime system. One of the key difficulties in this project however, remains the interoperability of Fortran–what `dgswem` is written in – and C++ –what HPX is written in. The works by Byerly et al. [**?**] illustrate some of the difficulties that go into this. However, this fix is places relatively restrictive limitations on what the code can do. Furthermore, the development of new C++ libraries present interesting solutions, which are currently not accessible to fortran based implementations. In particular, examples include explicit data parallel programming libraries such as `Boost.simd` and `Vc`, as well as novel GPU abstraction mechanisms such as Kokkos. While certainly pains have been taken to implement a dgswem-hpx implementation. This is not really affordable, when looking at developer

cost. The choice of Fortran as a programming language restricts our abilities to explore novel solutions to HPC problems.

1.2 Proposed Solution

In an attempt to directly address the issues above `dgswem-v2` has been design with the following features: written in C++14, object-oriented, and test driven design.

1.2.1 Written in C++14

The simplest solution to address issues with C++ interoperability is to write the application in C++. Additionally, we have chosen the C++14 standard. Typical reasons, for not using newer standards hinge around legacy support issues. However, since the code base is being rewritten in its entirety, we have no such limitations.

1.2.2 Object Oriented

One of the key features of C++ is it's support of object oriented programming. The main principal of object oriented programming is to partition the program into objects that are responsible for discrete tasks. This modularization allows users to work on one part of the code without having to worry about unintended effects in the rest of the code base. This is hopefully shorten the amount of time required for a new user to become productive. Additionally, it allows us to reasonably partition up the timesteppers. In terms of the previous example regarding the Runge-Kutta timesteppers, each time stepper would be implemented as a unique class, and its arguments would contain all the functionality required for function evaluations. This would allow a user to make a change in the code, and have it automatically propagated to both timesteppers, and expose an API that the user could use two develop a third timestepper.

1.2.3 Test driven design

The other advantage of object oriented programming is that it strictly scopes the data it's working on. This allows us to write tests for small units of code, e.g. integration or polynomial basis implementations. These changes will then hopefully indicate where and when code breaks greatly simplifying debugging. Additionally, existing continuous integration software allows us to run these unit tests for each commit. Allowing developers to precisely pinpoint the commit that broke functionality.

1.3 Target Audience

There certainly already exist a vast set of finite element libraries out there, e.g. `deal.ii`, `dune`, `feel++`. Certainly, there are a lot of arguments for using on of these libraries. However, ultimately we have settled on implementing their functionality ourselves. We are aiming to solve a much smaller subset of the problems these libraries attempt to solve. Libraries like `deal.ii` have quite intense dependencies, which we simply are not interested in at the moment. Furthermore, it is worth mentioning that `dgswem-v2` isn't even library, but rather an application. Our target audience really falls into two user groups:

1. People who are interesting in modeling shallow water equations. In the same, spirit as adcirc, we hope to provide the functionality, that might allow a coastal engineer to supply the code with a mesh geometry, and run a simulation.

2. People who are interested in developing new numerical methods. The bare-bones API has been designed to expose hooks that mirror the numerical algorithms as closely as possible. The hope being that these hooks allow users to rapidly explore new algorithms.

1.3.1 Limitations

While attempts have been made to leave `dgswem-v2`'s API as open as possible. It is important to list some of the governing principles. These principals dictate limitations with what can be naturally achieved with the code.

1. Adding new features should be more important than maintaining the best performance. While we certainly are not ignore performance in the design of `dgswem-v2`, it is of note that we should prefer flexibility of the code over diehard performance.

2. The code is centered around solving conservation laws using the discontinuous Galerkin Method. The main ramifications of this item, is that all of our solvers are set-up as explicit. Presumably, there would have to be significant refactoring to incorporate implicit solvers.

1.4 Remainder of the Guide

The remainder of this guide aims to underscore the design of `dgswem-v2`. The two main abstractions are the `Mesh` and `Problem` class. The `Mesh` class abstracting the underlying mesh, and the `Problem` abstracting the PDE/ discretization thereof. These two concepts are the intellectual pillars upon which `dgswem-v2` is based. Thereafter, we include annotated examples, which will hopefully get users started.

This is followed-up with a description of the input format, and a chapter describing where `dgswem` functionality can be found in the new code.

Chapter 2

`Mesh` class

Chapter 3

`Problem` Class

Chapter 4

Annontated Examples

Chapter 5

Input Format

Chapter 6

dgswem to dgswem-v2 Look-up Table