dgswem-v2: User's Guide

March 7, 2018

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

2.1 Motivation

To understand, the design behind the Mesh class, we would like to begin by discussing one of the key problems currently faced in `dgswem`. The issue comes from Steven Brus' dissertation [?]. Brus goes on to show that the addition of curvilinear elements plays a significant role in obtaining high-order accuracy for real world problems. Simultaneously, these curvilinear elements incur a significantly higher computational cost. Thus in the interior of the finite element domain, where the solution doesn't exhibit mesh geometry to more computationally effient standard affine elements suffice. While computationally straight forward, this approach provides a software engineering challenge. Namely, something like the volume kernel now needs to be split up into two loops, i.e.

Listing 2.1: Naïve implementation of curved and linear finite element kernel

```
for ( auto& elt : linear_elements ) {
  //Evaluate volume kernel for each linear element
}

for ( auto& elt : curved_elements ) {
  //Evaluate volume kernel for each curved element
}
```

This can quickly provide a software engineering headache. For instance, the addition of quadrilateral elements, would suddenly require that each time step cover 4 loops. The addition of more features will lead to more code bloat ultimately resulting in maintainable code. The main goal of the Mesh class is to address specifically the problem, mentioned above.

2.2 The Element Class

The object-oriented solution approach is based on the fact that the discontinuous Galerkin algorithm ultimately, simply applies integrals over the elements, remaining mathematically agnostic to the actually implementation of the approximation. This lends itself to a polymorphic implementation. The idea behind polymorphism being that functionally all elements should behave identically. Thus, if we can agree to an interface that we would like to expose. We should be able to accomplish everything in one loop over the elements without having to worry about what's happening under the hood.

Regardless of the type of element, we assume that for any given element, the approximated solution u^h is given in the form

$$u^h(x,t) = \sum_{i=0}^{N} u_n(t)\phi_n(x),$$

where ϕ_n is some basis. The approximated solution is determined by the evolution of the functions $u_n(t)$. For the Galerkin Method, this evolution is ultimately determined by ensuring that the residuals of the approximate solution is orthogonal to the space spanned by $\{\phi_n\}$. Thus the key functionality that every element must possess is that ability to compute integrals of the form

$$\int_\Omega f\phi_n \underset{\sim}{\mathrm{x}}$$

for an arbitrary function f. In practice these integrations are approximated by quadrature rules. However, this provides the basis for the interface, we would like to expose from a generic element class.

Listing 2.2: Generic Element API

```
class Element {

  //Compute the value of F at the Gauss-points
  template <typename F>
  void ComputeFgp(const F& f, std::vector<double>& f_gp);

  //Compute the value of  a function u given by
  // basis coefficients u at the Gauss-Points
  void ComputeUgp(const std::vector<double>& u,
                  std::vector<double>& u_gp);

  //Compute the gradient of a function given by
  // basis coefficients u at the Gauss Points
  void ComputeDUgp(const uint dir,
                   const std::vector<double>& u,
                   std::vector<double>& du_gp);

  //Compute the integral of u
  double Integration(const std::vector<double>& u_gp);

  //Test u against basis function dof
  double IntegrationPhi(const uint dof,
                        const std::vector<double>& u_gp);

  //Test u against the derivative in direction dir
  // of basis function dof
  double IntegrationDPhi(const uint dir,
                         const uint dof,
                         const std::vector<double>& u_gp);

  //...
};
```

That is to say if every element satisfied this API, we would be able to write the discontinuous Galerkin kernel with only one loop regardless of the element.

2.2.1 Strong Typing

C++ is a strongly typed language. Thus, in exposing any kind of polymorphism there are two options. (1) Dynamic polymorphism and (2) Static polymorphism. Dynamic polymorphism is typically achieved through the use of the `virtual` keyword. This approach is typically considered the most readable, and typically be preferred in a first attempt at an implementation. However, virtual objects typically can't be resolved by the compiler at runtime. This means that when the executable is running, the compiler maintains a virtual look-up table, which it uses to determine which implementation to call, when provided one of these virtual calls. The cost of this is that there is a virtual overhead associated with each of these calls. Additionally, the compiler may have trouble inlining and optimizing virtual function calls. Typically, when the function execution is large enough this overhead may be treated as negligible. But for our applications, these function calls are expensive enough that virtualization would seriously degrade performance.

The other typical approach is the use of static polymorphism. The idea being that we still maintain a unified API exposed by the class. However, now we specify the order in which we loop through the elements. That way the compiler is able to determine each of the function calls. Although ultimately, we will be able to get away with still only having one loop. The binary code generated should be equivalent to the code in Listing 2.1. Thus, static polymorphism provides us with the usage of dynamic polymorphism without the additional overhead. The downsides include a code base that is more difficult to maintain. However, given the performance critical nature of these function calls, this is a downside we've decided to accept.

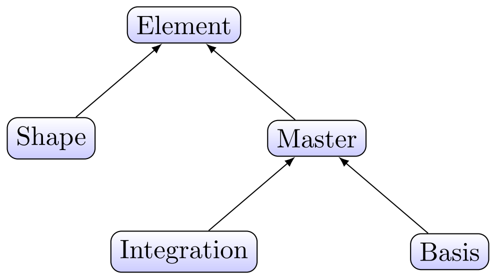

Figure 2.1: Composition of the Element classes.

2.2.2 Class Hierachy of Element

The last subsection describes the exact decomposition of the element. In designing `dgswem-v2`, we attempted to think of all possible features that one might want to implement and provide encapsulation to allow for adding of new features without having to be familiar with the entire code base. Figure 2.1 describes the major constituents of the element class.

- **Shape**: The shape class deals with deformations from the master element to the actual orientation within the mesh. Here features such as curvature of the mesh (e.g. solving in spherical coordinates) or the element (e.g. isoparametric or isogeometric) should be implemented.

- **Master**: The master element is rather similar to the generic element class. It exposes integration hooks over the master element.

- **Basis**: This class encapsulates all basis information. Potentially additional choices of basis include Bernstein, modal, and bases on quadrilateral elements.

- **Quadrature**: This class contains all information required to approximate an integral over the master element.

In addition to adding features by providing new instances of these classes. There also still remains the ability to add classes through template specialization and SFINAE. These techniques allow for special implementations to be written in certain element compositions. For example, one of the large advantages of a Bernstein basis is the fast matrix inversion formula. Since this formula drastically deviates from linear elements, one might want to write a specific implementation for linear triangles using the Bernstein basis.

2.3 `Mesh` Class

With the decision to strongly type the elements in the Mesh class, the next task is to develop a container, which represents the mesh. This requires the ability to iterate over interfaces, boundaries, elements, and distributed boundaries. The key object in allowing this to are the heterogeneous containers in the **Utilities** namespace. To explain, what in particular is happening in these containers assume we have three types of elements in our mesh: `EltA, EltB, EltC`. Morally, the heterogeneous vector can be written out as:

```
using HeterogeneousVector<EltA,EltB,EltC> = tuple<vector<EltA>,
                                                  vector<EltB>,
                                                  vector<EltC>>;
```

Now assuming, we have a function we would like to execute on each element regardless of type. We emulate the `std::for_each` API. So we will define a hook in the Mesh class, which will execute that function for every element in the HeterogeneousVector. We have demonstrated what specifically we would like in Listing 2.3.

Listing 2.3: Moral implementation of iterating over HeterogeneousVector

```
template<typename Element>
void SomeKernel(Element& e);

template<typename F>
void ForEachImpl( HeterogeneousVector<EltA,EltB,EltC>& v,
                   const F& f ) {

  vector<EltA>& vA = get<0>(v);
  for_each(vA.begin(),vA.end(), f);

  vector<EltB>& vB = get<1>(v);
  for_each(vB.begin(), vB.end(), f);

  vector<EltC>& vC = get<2>(v);
  for_each(vC.begin(), vC.end(), f);
}

//Apply SomeKernel to every element in v
MoralForImpl(v, SomeKernel);
```

In `dgswem-v2`, this type of implementation can be generalized to arbitrary Hetereogeneous vectors through the use of template metaprogramming. However, ultimately, the code is effectively executing the above code. Allowing the compiler to generate these loops is precisely the issue we wanted to address. The definition of the kernels to be passed into the mesh class are problem specific and will be addressed in the next chapter. This mechanism is used for boundaries, internal interfaces, and distributed interfaces, in addition to elements. In understanding, this key concept the Mesh class becomes nothing more than a container. Detailed API descriptions are provided in the doxygen documentation.

Chapter 3

`Problem` Class

Chapter 4

Annontated Examples

4.1 Preliminaries

This chapter demonstrates the core functionality of the code. In Section 4.2, we present steps necessary for running a manufactured solution problem.

In this section, we will go through several example problems for running DGSWEM-V2. In order to provide succint instructions to the reader, we introduce the following environment variables. We will assume that the following environment variables have been defined in the users environment:

```
export DGSWEMV2_REPO=<path/to/dgswemv2_repo>
export DGSWEMV2_BUILD=<path/to/dgswemv2_build>
export DGSWEMV2_EXAMPLES=${DGSWEMV_REPO}/examples
```

Additionally, to keep the compile time of the application short, the cmake configuration does not automatically build the examples. To build the following examples, you will need to rerun your CMAKE build command with the additional option `-DBUILD_EXAMPLES=On`.

4.2 Manufactured Solution

The method of manufactured solutions presents an easy means of determining whether the solution is converging at the rates stipulated by the theoretical error estimates. In addition, since we obtain exact error estimates, the method of manufactured solutions may also be used to determine, whether or not errors have been introduced between parallel and serial implementations.

To avoid the stability-related difficulties associated with greatly varying mesh refinements, we evaluate the manufactured solution from [?]. For the manufactured solution, we set the solution to be equal to

$$\zeta(x,y,t) = 2\zeta_0 \frac{\cos(\omega(x-x_1))\cos(\omega(y-y_1))\cos(\omega t)}{\cos(\omega(x_2-x_1))\cos(\omega(y_2-y_1))}$$
$$q_x(x,y,t) = \zeta_0 \frac{\sin(\omega(x-x_1))\cos(\omega(y-y_1))\sin(\omega t)}{\cos(\omega(x_2-x_1))\cos(\omega(y_2-y_1))}$$
$$q_y(x,y,t) = \zeta_0 \frac{\cos(\omega(x-x_1))\sin(\omega(y-y_1))\sin(\omega t)}{\cos(\omega(x_2-x_1))\cos(\omega(y_2-y_1))}$$

where $x_1 = 40{,}000\,\mathrm{m}$, $x_2 = 83{,}200\,\mathrm{m}$, $y_1 = 10{,}000\,\mathrm{m}$, $y_2 = 53{,}200\,\mathrm{m}$, $\zeta_0 = 0.25\,\mathrm{m}$, $\omega = 2\pi/43{,}200\,\mathrm{rad/s}$. Additionally, $g = 9.81\,\mathrm{m/s}^2$, and the bathymetry is constant with depth $H_0 = 2\,\mathrm{m}$. The source term is then obtained by applying the left hand side of the shallow water equations to the target solution.

The manufactured solutions can be run via the three following targets, which can be made as follows:

```
cd ${DGSWEMV2_BUILD}
make MANUFACTURED_SOLUTION_SERIAL
make MANUFACTURED_SOLUTION_HPX
make MANUFACTURED_SOLUTION_OMPI
```

Note that for the HPX and the OMPI manufactured targets, the project must have the `-DUSE_HPX` and `-DUSE_OMPI` options set to `On`, respectively.

The workflow for building the manufactured solution consists of three parts: (1) generating a mesh, (2) if necessary, partitioning the mesh, (3) running the simulation, and (4) interpreting the output.

4.2.1 Generating the mesh

For the generation of meshes, we use a `yaml`-formatted input file. We have included an input file in

```
${DGSWEMV2_EXAMPLES}/manufactured_solution/input_files/mesh_generator_input.yml
```

We have provided comments for the individual variables in the yaml files. One key aspect for improving the accuracy of the solution is the mesh resolution. To modulate these, `num_x_subdivisions` and `num_y_subdivisions` allows one to refine or coarsen the mesh appropriately. However, for now we leave them at their defaults. To generate the mesh, run the following commands

```
cd ${DGSWEMV2_EXAMPLES}/manufactured_solution/input_files/
${DGSWEMV2_BUILD}/mesh_generators/rectangular_mesh_generator \
    mesh_generator_input.yml
```

This will generate a `rectangular_mesh.14` file in the current directory. This is an ADCIRC-formatted meshfile.

4.2.2 Partitioning the mesh (optional)

Note that this section is only necessary if you are attempting to run the simulation in a distributed manner, i.e. using the OMPI or HPX executable. If you are only trying to run the simulation in serial, skip to the next subsection.

In order to run the simulation in parallel, the mesh must be broken into smaller pieces, which can then be assigned to individual processors. Since there exist stark contrasts in the send latencies between two processors on the same node versus processors that might be connected via an interconnect, we require the user to specify the number of localities (i.e. private memory address spaces) in addition to the number of partitions to allow for the mesh partitioner to optimize these interconnect effects.

The partitioner executable can be found in

```
${DGSWEMV2_BUILD}/partitioner/partitioner
```

and running the executable without any arguments will provide usage information. In particular, the variables mean the following:

- `<input filename>`: The name of the input file used to run the execution.
- `<number of partitions>`: The number of partitions the mesh is to be partitioned into.
- `<number of nodes>`: The number of hardware localities the simulation is to be run on.
- `<ranks per locality>`: The number of ranks per locality.
- `<rank balanced>`: Whether or not the constraints should be balanced across the individual submeshes. Note that the entry options are `true` or `false`. Recommended for OpenMP/MPI.

Our two parallelization strategies rely on different parallel execution models. Thus the inputs for either version need to be slightly modified. In the following two subsections we outline, the differences in general, but also provide concrete numbers to allow the user to proceed.

Partitioning for HPX

The HPX execution model varies from that of traditional parallelization strategies in that the number of partitions does not correspond to any hardware concept. For example, traditional MPI implementations assigned one rank per core, and insofar the number of partitions equalled the number of cores. For HPX, the number of partitions is roughly proportional to the number of tasks executed on that locality. *Oversubscribing* meshes — assigning more meshes to the locality than there are cores — can lead to desirable behavior in the form of hiding of send latencies. However, the user needs to be careful to avoid exposing too fine grain parallelism in the form of too many meshes, because task overhead will dominate the execution time and lead performance degradation.

To continue the example for an HPX parallelization, we recommend running:

```
cd ${DGSWEMV2_EXAMPLES}/manufactured_solution/input_files
${DGSWEMV2_BUILD}/partitioner/partitioner\
    dgswemv2_input.15 4 1
```

This should generate, 4 `meta`-formatted mesh files. The `meta`-format isn't an official mesh format, but rather a simple method of representing the mesh internally for the DGSWEM-v2 application, and 4 `dbmd`-formatted files, which encapsulate the distributed metadata information required to ensure that submeshes appropriately communication with one another. In addition, we will have generated an updated input file specifically for running parallel meshes. For this example, it's `dgswemv2_input_parallelized.15`.

Partitioning for OpenMP/MPI

Partitioning for the MPI+OpenMP implementation is slightly, different. Ratio of number of partitions to number of threads should correspond to the number of threads available on each node. However, it's also possible to run a flat MPI implementation by modifying the `<ranks per locality>` option. This option will correspond to the number MPI ranks on each node.

Additionally, since submeshes are mapped statically to threads, we recommend setting the `<rank balanced>` option to `true`. For applications with varying load across the elements, e.g. in the case of wetting and drying, we would like to balance load and memory constraints across the submeshes. Ultimately, the performance will be constrained along the critical path, for statically mapped parallelizations optimal performance is achieved when each submesh roughly has the same amount of work. Note that this is not necessary for the HPX parallelization, which implements aggressive on-node work stealing.

To generate a mesh partitioning for this parallelization, we run

```
cd ${DGSWEMV2_EXAMPLES}/manufactured_solution/input_files
${DGSWEMV2_BUILD}/partitioner/partitioner\
   dgswemv2_input.15 2 1 2 true
```

This configuration will result in a flat MPI run. With 2 MPI ranks. Note that similar to the HPX run, we generate both `meta` mesh files and `dbmd` connectivity information and an updated `dgswemv2_input_paralllelized.15` input file.

4.2.3 Running the simulation

For each of the three execution modes — serial, with HPX, and with MPI+OpenMP — we have a separate executable. To execute the serial implementation, run

```
cd ${DGSWEMV2_EXAMPLES}/manufactured_solution/input_files
mkdir -p output
${DGSWEMV2_BUILD}/examples/MANUFACTURED_SOLUTION_SERIAL\
    dgswemv2_input.15
```

For the MPI and HPX versions, we assume that the parallel launcher, would be accessible through some `<mpirun>` command, e.g. on a slurm based system this might `srun`. The MPI-version can then be executed via

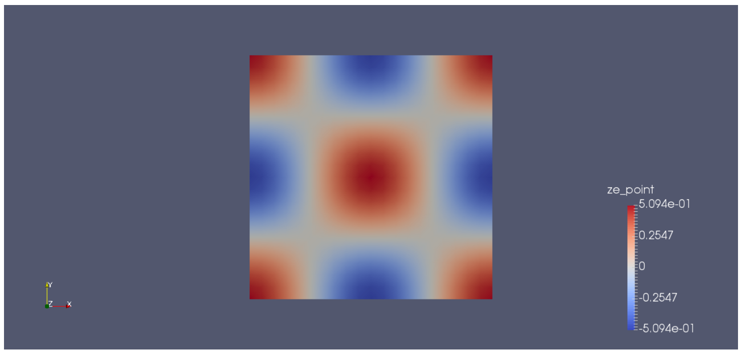

Figure 4.1: Sample output of the manufactured solution output.

```
cd ${DGSWEMV2_EXAMPLES}/manufactured_solution/input_files
mkdir -p ouptut
<mpirun> ${DGSWEMV2_BUILD}/examples/MANUFACTURED_SOLUTION_OMPI\
    dgswemv2_input.15
```

For the HPX version, the execution command depends on the set-up of the system. If there is no distributed aspect to your computing environment, you do not need a distributed launcher like `srun`. Thus, for the reader running the simulation on their laptop, i.e. with one locality, the HPX example can be run as

```
cd ${DGSWEMV2_EXAMPLES}/manufactured_solution/input_files
mkdir -p output
${DGSWEMV2_BUILD}/examples/MANUFACTURED_SOLUTION_HPX\
    dgswemv2_input.15
```

4.2.4 Intepreting the Output

DGSWEM-V2 writes output in several forms. Firstly, in the `output` folder that was created in the `${DGSWEMV2}_EXAMPLES/manufactured_solution` directory. You will find a log file name `log` and `vtk`-formatted output. The output can be visualized using a package like PARAVIEW. A sample image of what the output should look like is shown in Figure 4.1.

4.3 1D Inlet

Chapter 5

Input Format

Chapter 6

`dgswem` to `dgswem-v2` Look-up Table

This chapter is aimed at helping users proficient with `dgswem` orient themselves in `dgswem-v2`. Table provides means to look up subroutines from `dgswem` and determine the location in `dgswemv2`. Certain functionality is not present at all in `dgswem-v2`. These files and subroutines therein have been omitted from the table. If you are searching for a subroutine from:

- dg.F
- Diff45_41.F
- flow_edge_sed.F
- fparser.F90
- fparser.F90
- global_3dvs.F
- globalio.F
- harm.F
- mkodal2nodal.F
- nodalattr.F
- parameters.f90
- pdg_debug.F
- sizes.F
- slopelimiter.F

- wind.F
- write_output.F

the desired functionality has not been implemented in `dgswem-v2`.

`dgswem` subroutine	`dgswem-v2` equivalent function	Location in Repository
calc_normal	Shape::StraightTriangle::GetSurfaceNormal	source/shape/shapes_2D/shape_straighttriangle.cpp
check_errors	Not supported	
coldstart	Not supported	
create_edge_data	initialize_mesh_interfaces_boundaries	source/preprocessor/initialize_mesh.hpp
dg_hydro_timestep	Inlined	source/simulation/simulation.hpp source/hpx_simulation.hpp source/simulation/ompi_simulation.hpp
dg_timestep	Inlined	source/simulation/simulation.hpp source/hpx_simulation.hpp source/simulation/ompi_simulation.hpp
dgswem	main	source/simulation/simulation.hpp source/hpx_simulation.hpp source/simulation/ompi_simulation.hpp
ebarrier_edge_hydro	Not supported	
edge_int_ldg_hydro	Not supported	
edge_int_ldg_sediment	Not supported	
errorelevsum	Not supported	
flow_edge_hydro	SWE::Flow::GetEX†	source/problem/SWE/swe_kernels_boundary_conditions.hpp
flow_edge_ldg_hydro	Not supported	
fort_dg_setup	Not supported	
Hllc_flux	Not supported	
hotstart	Not supported	
ibarrier_edge_hydro	Not supported	
ibarrier_fluxes	Not supported	
internal_edge_hydro	SWE::Problem::interface_kernel†	source/problem/SWE/swe_kernels_processor.hpp
internal_edge_ldg_hydro	Not supported	
land_edge_hydro	SWE::Land::GetEX†	source/problem/SWE/swe_kernels_boundary_conditions.hpp
land_edge_ldg_hydro	Not supported	
LDG_hydro	Not supported	
llf_flux	SWE::LLF_flux	source/problem/SWE/swe_LLF_flux.hpp
Message_abort	Not supported	
Message_fini	Inlined	source/problem/SWE/ompi_main_swe.cpp
message_init	Inlined	source/problem/SWE/ompi_main_swe.cpp
met_forcing	Not supported	
msg_blocksync_finish	Not supported	
msg_blocksync_start	Not supported	
msg_start_elem	OMPICommunicator:InitializeCommunication	source/communication/ompi_communicator.cpp
msg_table_elem	OMPICommunicator::OMPICommunicator	source/communication/ompi_communicator.cpp
msg_types_elem	Not supported	
ncp_flux	Not supported	
numerical_flux	Inlined	
nws112interp_region	Not supported	
nws12get	Not supported	
nws12init	Not supported	
nws12interp_basin	Not supported	
ocean_edge_hydro	Not supported	
ocean_edge_hydro_post	Not supported	
ocean_edge_hydro_strict	Not supported	
ocean_edge_ldg_hydro	Not supported	
ocean_flow_edge_hydro	Not supported	
orthobasis	Basis::Dubiner_2D::GetPhi Basis::Dubiner_2D::GetDPhi	source/basis/bases_2D/basis_dubiner_2D.cpp
orthobasis_area	Basis::Dubiner_2D::GetPhi Basis::Dubiner_2D::GetDPhi	source/basis/bases_2D/basis_dubiner_2D.cpp
orthobasis_edge	Basis::Dubiner_2D::GetPhi	source/basis/bases_2D/basis_dubiner_2D.cpp
p_enrichment	Not supported	
para_max	Not supported	

para_min	Not supported	
para_sum	Not supported	
prep_DG	initialize_mesh	source/preprocessor/initialize_mesh.hpp
prep_slopelim	Not supported	
quad_pts_area	Integration::Dunabant_2D::GetRule	source/integration/integrations_2D/integration_dunavant_2D.cpp
quad_pts_edge	Integration::GaussLegendre_1D::GetRule	source/integration/integrations_1D/integration_gausslegendre_1D.cpp
radiation_edge_hydro	Not supported	
radiation_edge_ldg_hydro	Not supported	
read_fixed_fort_dg	Not supported	
read_input	InputParameters::InputParameters	source/preprocessor/input_parameters.hpp
read_keyword_fort_dg	Not supported	
rhs_dg_hydro	SWE::Problem::volume_kernel†	source/problem/SWE/swe_kernels_processor.hpp
	SWE::Problem::source::kernel†	
rhs_ldg_hydro	Not supported	
rk_time	Stepper::Stepper	source/simulation/stepper.cpp
roe_flux	Not supported	
scrutinize_solution	SWE::Problem::scrutinize_solution_kernel†	source/problem/SWE/swe_kernels_processor.hpp
sta_basis	Integration::Dunabant_2D::GetRule	source/integration/integrations_2D/integration_dunavant_2D.cpp
sta_location	Not supported	
tidal_potential	Not supported	
updatei_elem	Not supported	
updatelz_elem	Not supported	
updatemz_elem	Not supported	
updater_elem	Not supported	
updater_elem_mod	OMPICommunicator::SendAll	source/communication/ompi_communicator.hpp
	OMPICommunicator::ReceiveAll	
	OMPICommunicator::WaitAllSends	
	OMPICommunicator::WaitAllReceives	
updater_elem_mod2	OMPICommunicator::SendAll	source/communication/ompi_communicator.hpp
	OMPICommunicator::ReceiveAll	
	OMPICommunicator::WaitAllSends	
	OMPICommunicator::WaitAllReceives	
updater_elem_mod3	OMPICommunicator::SendAll	source/communication/ompi_communicator.hpp
	OMPICommunicator::ReceiveAll	
	OMPICommunicator::WaitAllSends	
	OMPICommunicator::WaitAllReceives	
wetdry	Not supported	
write_results	SWE::Problem::extract_VTK_data_kernel	source/problem/SWE/swe_kernels_postprocessor.hpp
	SWE::Problem::extract_modal_data_kernel	

Table 6.1: The names of Fortran subroutines with the corresponding `dgswem-v2` function calls and the location of their implementation. Not supported routines are not implemented in `dgswem-v2`. The dagger (†) signifies that a given implementation is defined element-wise, whereas typical `dgswem` implementations loop over the entire mesh.