# swerve Documentation

***Release***

**Alice Harpole**

**May 18, 2017**

# Contents:

**S**hallow **W**ater **E**quations for **R**elati **V**istic **E**nvironments

`swerve` is a set of software designed to investigate the general relativistic form of the shallow water equations. The code is developed in the notebook `Shallow_Water_Equations.ipynb`, before being implemented in an optimized C++/CUDA version which runs on the GPU. MPI is used to run the code on multiple GPUs (if available).

CHAPTER 1

# Installation and usage

The CUDA version can be built using the Makefile and run using the parameters in the file `input_file.txt`. Before compiling, make sure that the variables `CUDA_PATH` and `MPI_PATH` at the top of the Makefile point to the correct locations of CUDA and MPI on your system. The code can then be compiled by executing `make` (or *make debug* to include debug flags).

To run on e.g. 2 GPUs/processors, execute

```
mpirun -np 2 ./gr_cuda
```

or to use the custom input file `custom_input.txt`,

```
mpirun -np 2 ./gr_cuda custom_input.txt
```

This code outputs into an HDF5 file which can be viewed using the notebook `Plotting.ipynb` (inadvisable except for very small files) or using the python script `plot.py`.

A version of the code which evolves a section of the domain using the compressible fluid equations on a finer grid can be compiled and run using `make mesh` and `./mesh`.

CHAPTER 2

# Documentation

In order to build the documentation, you must first ensure that `doxygen` and `sphinx` are installed on your system. From the main swerve directory, then execute

```
doxygen Doxyfile
cd docs
make
```

This will provide a list of the possible formats for the documentation. Follow the instructions to build the documentation in the format of your choice.

CHAPTER 3

# Testing

A set of tests can be compiled by going to the main `swerve` directory and executing

```
make test
```

then a test case can be run:

```
cd testing
./flat
```

This test case provides initial data that is flat with a static gravitational field and no burning. It then tests that this data remains unchanged after being evolved through 100 timesteps.

Unit tests can be run by compiling the tests then running

```
cd testing
./unit_tests
```

This will run a set of tests on the majority of the individual functions used and output to screen whether each function tested has passed or failed.

# CHAPTER 4

# Input Files

Initial data in swerve is described in two ways: the first is an input file, describing the parameters of the system, the second is a C++ function which describes the initial data on the coarsest multilayer shallow water grid.

The input file is a text file which provides swerve with the system parameters. This input file is read in at the beginning of the program and used to set up the necessary data structures. Input data is validated at this point and will terminate if invalid parameters are encountered. The filename of this input file can be provided as an argument at runtime - if no argument is provided, then the program defaults to the file `mesh_input.txt`. The standard form of the input file is as follows:

**`nx`** Number of grid points in the $x$ dimension of the coarsest grid

**`ny`** Number of grid points in the $y$ dimension of the coarsest grid

**`nt`** Number of timesteps

**`ng`** Number of ghost cells

**`r`** Refinement ratio

**`nlevels`** Number of levels of mesh refinement

**`models`** List of physical models to be used on each level, where

`S` = single layer shallow water,

`M` = multilayer shallow water,

`C` = compressible and

`L` = Low Mach

**`nzs`** Number of layers / grid points in the vertical direction for each grid

**`df`** Fraction of the domain each level should cover with respect to the previous level

**`xmin`** Minimum $x$ coordinate of coarsest grid

**`xmax`** Maximum $x$ coordinate of coarsest grid

**`ymin`** Minimum $y$ coordinate of coarsest grid

**ymax** Maximum $y$ coordinate of coarsest grid

**zmin** Height of sea floor

**zmax** Maximum $z$ coordinate of coarsest compressible grid

**rho** List of densities $\rho$ of multilayer shallow water layers

**Q** Energy release rate

**E_He** Binding energy of reactant

**Cv** Specific heat capacity at constant volume

**gamma** Adiabatic index $\gamma$

**alpha** Lapse function $\alpha$

**beta** Shift vector $\beta$

**gamma_down** Covariant spatial metric $\gamma_{ij}$

**periodic** Are the boundary conditions periodic (`t`) or outflow (`f`)

**burning** Do we include burning reactions (`t`) or not (`f`)

**dprint** Number of timesteps between outputting data to file

**outfile** Path to output file (must be HDF5)

**n_print_levels** Number of levels to be printed to file

**print_levels** List of indices of levels to be printed to file

The specific form of the initial data is described in the `main` function of the file `run_mesh_cuda.cpp`. The initial state vector must be provided for all points in the coarsest multilayer shallow water grid.

CHAPTER 5

# Classes and functions

## Mesh_cuda.h

**class `Sea`**

*#include* <*Mesh_cuda.h*> A class that manages the simulation.

Implements *Sea* class.

### Public Functions

**`Sea`**(int *_nx*, int *_ny*, int *_nt*, int *_ng*, int *_r*, float *_df*, float *xmin*, float *xmax*, float *ymin*, float *ymax*, float *zmin*, float *zmax*, float **_rho*, float *_Q*, float *_gamma*, float *_E_He*, float *_Cv*, float *_alpha*, float **_beta*, float **_gamma_down*, bool *_periodic*, bool *_burning*, int *_dprint*, int *_print_level*)
Constructor from list of parameters.

**`Sea`**(stringstream &*inputFile*, char **filename*)
Constructor for *Sea* class using inputs from file.

Data is validated: an error will be thrown and the program terminated if any of the inputs are found to be invalid.

**Parameters**

- `filename`: name of input file

**`Sea`**(char **filename*)

**`Sea`**(**const** *Sea*&)
Copy constructor

void **`initial_swe_data`**(float **D0*, float **Sx0*, float **Sy0*)
Initialise D, Sx, Sy.

**Parameters**

- `D0`: conserved density
- `Sx0`: conserved x-velocity
- `Sy0`: conserved y-velocity

void **initial_compressible_data**(float **D0*, float **Sx0*, float **Sy0*, float **Sz0*, float **tau0*)
Initialise D, Sx, Sy, Sz, tau.

**Parameters**

- `D0`: conserved density
- `Sx0`: conserved x-velocity
- `Sy0`: conserved y-velocity
- `Sz0`: conserved z-velocity
- `tau`: conserved energy

void **bcs**(float **grid*, int *n_x*, int *n_y*, int *n_z*, int *vec_dim*)
Enforce boundary conditions on grid of quantities with dimension vec_dim.

**Parameters**

- `grid`: grid on which boundary conditions are to be enforced
- `n_xn_yn_z`: grid dimensions
- `vec_dim`: dimension of state vector

void **print_inputs**()
Print some input and runtime parameters to screen.

void **run**(MPI_Comm *comm*, MPI_Status **status*, int *rank*, int *size*, int *tstart*)
Run simulation.

**Parameters**

- `comm`: MPI communicator
- `status`: MPI status flag
- `rank`: MPI process rank number
- `size`: Total number of MPI processes
- `tstart`: Start time

**~Sea**()
Deconstructor. Clean up member arrays.

### Public Members

int **nx**
number of gridpoints in x-direction of coarsest grid

int **ny**
number of gridpoints in y-direction of coarsest grid

int ***nxs**
number of gridpoints in x-direction of grids

int ***nys**
number of gridpoints in y-direction of grids

int ***nzs**
Number of layers to have in each grid

int **ng**
Number of ghost cells

int **nlevels**
Number of levels of mesh refinement

char ***models**
Array describing the physical model to use on each level. S = single layer SWE, M = multilayer SWE, C = compressible, L = Low Mach

int ***vec_dims**
Dimensions of state vectors on each grid

float **gamma**
Adiabatic index

float **alpha0**
Lapse function

float **R**
Radius of star

float **dz**
Gridpoint separation in the z-direction of fine (compressible grid)

float **zmin**
Height of sea floor

float **zmax**
Maximum height of sea surface

float ***xs**
Vector of x-coordinates of coarsest gridpoints

float ***ys**
Vector of y-coordinates of coarsest gridpoints

float ****Us**
Array of pointers to grids.

float ***p_const**
Array of constant pressures on shallow water grids.

### Public Static Functions

**static** void **invert_mat**(float *A, int *m*, int *n*)
Invert the m x n matrix M in place using Gaussian elimination.

**Parameters**

- A: Matrix to be inverted
- mn: Dimensions of matrix

### Private Functions

void **init_sea**(stringstream &*inputFile*, char **filename*)

### Private Members

int **nt**
Total number of timesteps to run simulation for

int **r**
refinement ratio

int ***matching_indices**
Location of fine grids wrt coarser grid coordinates

float **dx**
Gridpoint separation in x-direction on coarsest grid

float **dy**
Gridpoint separation in y-direction on coarsest grid

float **dt**
Timestep

float **df**
Fraction of coarse grid covered by fine grid

float ***rho**
Vector of density in each of the shallow water layers

float **Q**
Mass transfer rate

float **E_He**
Energy release per unit mass of helium burning

float **Cv**
Specific heat at constant volume

float **beta**[3]
Shift vector

float **gamma_down**[3 *3]
Covariant spatial metric

float **gamma_up**[3 *3]
Contravariant spatial metric

bool **periodic**
Are the boundaries periodic (true) or outflow (false)

bool **burning**
Do we include burning? (True)

int **dprint**
number of timesteps between printouts

int **n_print_levels**
number of the level to be output to file

int ***print_levels**
number of the level to be output to file

char **outfile**[200]
Name of (hdf5) file to print output data to

char **paramfile**[200]
Name of parameter file

## mesh_cuda_kernel.h

### Typedefs

**typedef** void (***flux_func_ptr**) (float *q, float *f, int dir, float alpha0, float gamma, float zmin, float dz, int nz, int layer, float R)

**typedef** float (***fptr**) (float p, float D, float Sx, float Sy, float Sz, float tau, float gamma, float *gamma_up)

### Functions

unsigned int **nextPow2** (unsigned int *x*)

**__host__ __device__ bool nan_check(float a)**
check to see whether float a is a nan

**__host__ __device__ float zbrent(fptr func, const float x1, const float x2, const float to**
Using Brent's method, return the root of a function or functor func known to lie between x1 and x2. The root will be regined until its accuracy is tol.

**Parameters**

- `func`: function pointer to shallow water or compressible flux function.
- `x1x2`: limits of root
- `tol`: tolerance to which root shall be calculated to
- `DSxSySztau`: conserved variables
- `gamma`: adiabatic index

void **check_mpi_error** (int *mpi_err*)
Checks to see if the integer returned by an mpi function, mpi_err, is an MPI error. If so, it prints out some useful stuff to screen.

void **getNumKernels**(int *nx*, int *ny*, int *nz*, int *ng*, int *n_processes*, int **maxBlocks*, int **maxThreads*, dim3 **kernels*, int **cumulative_kernels*)

Return the number of kernels needed to run the problem given its size and the constraints of the GPU.

**Parameters**

- `nxnynz`: dimensions of problem
- `ng`: number of ghost cells
- `maxBlocksmaxThreads`: maximum number of blocks and threads possible for device(s)
- `n_processes`: number of MPI processes
- `kernels`: number of kernels per process
- `cumulative_kernels`: cumulative total of kernels per process

void **getNumBlocksAndThreads**(int *nx*, int *ny*, int *nz*, int *ng*, int *maxBlocks*, int *maxThreads*, int *n_processes*, dim3 **kernels*, dim3 **blocks*, dim3 **threads*)

Returns the number of blocks and threads required for each kernel given the size of the problem and the constraints of the device.

**Parameters**

- `nxnynz`: dimensions of problem
- `ng`: number of ghost cells
- `maxBlocksmaxThreads`: maximum number of blocks and threads possible for device(s)
- `n_processes`: number of MPI processes
- `kernelsblocksthreads`: number of kernels, blocks and threads per process / kernel

void **bcs_fv**(float **grid*, int *nx*, int *ny*, int *nz*, int *ng*, int *vec_dim*, bool *periodic*)

Enforce boundary conditions on section of grid.

**Parameters**

- `grid`: grid of data
- `nxnynz`: dimensions of grid
- `ng`: number of ghost cells
- `vec_dim`: dimension of state vector
- `periodic`: do we use periodic or outflow boudary conditions?

void **bcs_mpi**(float **grid*, int *nx*, int *ny*, int *nz*, int *vec_dim*, int *ng*, MPI_Comm *comm*, MPI_Status *status*, int *rank*, int *n_processes*, int *y_size*, bool *do_z*, bool *periodic*)

Enforce boundary conditions across processes / at edges of grid.

Loops have been ordered in a way so as to try and keep memory accesses as contiguous as possible.

Need to do non-blocking send, blocking receive then wait.

**Parameters**

- `grid`: grid of data
- `nxnynz`: dimensions of grid
- `vec_dim`: dimension of state vector

- `ng`: number of ghost cells
- `comm`: MPI communicator
- `status`: status of MPI processes
- `rankn_processes`: rank of MPI process and total number of MPI processes
- `y_size`: size of grid in y direction running on each process (except the last one)
- `do_z`: true if need to implement bcs in vertical direction as well
- `periodic`: do we use periodic or outflow boudary conditions?

**__host__ __device__ float W_swe(float * q, float * gamma_up)**
calculate Lorentz factor for conserved swe state vector

**__host__ __device__ float phi(float r)**
calculate superbee slope limiter Phi(r)

**__host__ __device__ float find_height(float ph, float R)**
Finds r given Phi.

**__device__ float find_pot(float r, float R)**
Finds Phi given r.

**__device__ float rhoh_from_p(float p, float rho, float gamma)**
calculate rhoh using p for gamma law equation of state

**__device__ float p_from_rhoh(float rhoh, float rho, float gamma)**
calculate p using rhoh for gamma law equation of state

**__device__ __host__ float p_from_rho_eps(float rho, float eps, float gamma)**
calculate p using rho and epsilon for gamma law equation of state

**__device__ __host__ float phi_from_p(float p, float rho, float gamma, float A)**
Calculate the metric potential Phi given p for gamma law equation of state

**Parameters**

- `prho`: pressure and density
- `gamma`: adiabatic index
- `A`: constant used in Phi to p conversion

**__host__ __device__ float f_of_p(float p, float D, float Sx, float Sy, float Sz, float tau,**
Function of p whose root is to be found when doing conserved to primitive variable conversion

**Parameters**

- `p`: pressure
- `DSxSySztau`: components of conserved state vector
- `gamma`: adiabatic index

**__device__ float h_dot(float phi, float old_phi, float dt, float R)**
Calculates the time derivative of the height given the shallow water variable phi at current time and previous timestep NOTE: this is an upwinded approximation of hdot - there may be a better way to do this which will more accurately give hdot at current time.

**Parameters**

- `phi`: Phi at current timestep

- `old_phi`: Phi at previous timestep
- `dt`: timestep

**__device__ float calc_Q_swe(float rho, float p, float gamma, float Y, float Cv)**
Calculate the heating rate per unit mass from the shallow water variables

**Parameters**

- `rho`: densities of layers
- `p`: pressure
- `gamma`: adiabatic index
- `Y`: species fraction
- `Cv`: specific heat in constant volume

void **calc_Q**(float **rho*, float **q_cons*, int *nx*, int *ny*, int *nz*, float *gamma*, float **Q*, float *Cv*, float **gamma_up*)
Calculate the heating rate per unit mass.

**Parameters**

- `rho`: densities of layers
- `q_cons`: conservative state vector
- `nxnynz`: dimensions of grid
- `gamma`: adiabatic index
- `Q`: array that shall contain heating rate per unit mass
- `Cv`: specific heat in constant volume
- `gamma_up`: spatial metric

**__device__ void calc_As(float * rhos, float * phis, float * A, int nlayers, float gamma, fl**
Calculates the As used to calculate the pressure given Phi, given the pressure at the sea floor

**Parameters**

- `rhos`: densities of layers
- `phis`: Vector of Phi for different layers
- `A`: vector of As for layers
- `nlayers`: number of layers
- `gamma`: adiabatic index
- `surface_phi`: Phi at surface
- `surface_rho`: density at surface

**__device__ void find_constant_p_surfaces(float * p_const, float gamma, float * q_comp, floa**

**__device__ void enforce_hse_d(float * q_comp, float * q_swe, int kx_offset, int ky_offset,**

void **enforce_hse**(float **q_comp*, float **q_swe*, int **nxs*, int **nys*, int **nzs*, int *ng*, int *level*, int *clevel*, float *zmin*, float *dz*, int **matching_indices*, float *gamma*)

**__device__ void cons_to_prim_comp_d(float * q_cons, float * q_prim, float gamma, float * ga**
Convert compressible conserved variables to primitive variables

**Parameters**

- `q_cons`: state vector of conserved variables
- `q_prim`: state vector of primitive variables
- `gamma`: adiabatic index

void **cons_to_prim_comp**(float **q_cons*, float **q_prim*, int *nxf*, int *nyf*, int *nz*, float *gamma*, float **gamma_up*)

Convert compressible conserved variables to primitive variables

**Parameters**

- `q_cons`: grid of conserved variables
- `q_prim`: grid where shall put the primitive variables
- `nxfnyfnz`: grid dimensions
- `gamma`: adiabatic index
- `gamma_up`: spatial metric

**__device__ void shallow_water_fluxes(float * q, float * f, int dir, float alpha0, float ga**

Calculate the flux vector of the shallow water equations

**Parameters**

- `q`: state vector
- `f`: grid where fluxes shall be stored
- `dir`: 0 if calculating flux in x-direction, 1 if in y-direction
- `alpha`: lapse function
- `gamma`: adiabatic index

**__device__ void compressible_fluxes(float * q, float * f, int dir, float alpha0, float gam**

Calculate the flux vector of the compressible GR hydrodynamics equations

**Parameters**

- `q`: state vector
- `f`: grid where fluxes shall be stored
- `dir`: 0 if calculating flux in x-direction, 1 if in y-direction, 2 if in z-direction
- `alpha`: lapse function
- `gamma`: adiabatic index

void **p_from_swe**(float **q*, float **p*, int *nx*, int *ny*, int *nz*, float *rho*, float *gamma*, float *A*, float **gamma_up*)

Calculate p using SWE conserved variables

**Parameters**

- `q`: state vector
- `p`: grid where pressure shall be stored
- `nxnynz`: grid dimensions
- `rho`: density

- gamma: adiabatic index
- A: variable required in p(Phi) calculation
- gamma_up: spatial metric

**__device__ float p_from_swe(float * q, float rho, float gamma, float W, float A)**
Calculates p and returns using SWE conserved variables

**Parameters**

- q: state vector
- rho: density
- gamma: adiabatic index
- W: Lorentz factor
- A: variable required in p(Phi) calculation

**__global__ void compressible_from_swe(float * q, float * q_comp, int * nxs, int * nys, int**
Calculates the compressible state vector from the SWE variables.

**Parameters**

- q: grid of SWE state vector
- q_comp: grid where compressible state vector to be stored
- nxsnysnzs: grid dimensions
- rhogamma: density and adiabatic index
- kx_offsetky_offset: kernel offsets in the x and y directions
- dt: timestep
- old_phi: Phi at previous timestep
- level: index of level

**__device__ float slope_limit(float layer_frac, float left, float middle, float right, float**
Calculates slope limited verticle gradient at layer_frac between middle and amiddle. Left, middle and right are from row n, aleft, amiddle and aright are from row above it (n-1)

**__global__ void prolong_reconstruct_comp_from_swe(float * q_comp, float * q_f, float * q_c,**
Reconstruct fine grid variables from compressible variables on coarse grid

**Parameters**

- q_comp: compressible variables on coarse grid
- q_f: fine grid state vector
- q_c: coarse grid swe state vector
- nxsnysnzs: grid dimensions
- ng: number of ghost cells
- dz: coarse grid vertical spacing
- matching_indices_d: position of fine grid wrt coarse grid
- kx_offsetky_offset: kernel offsets in the x and y directions
- coarse_level: index of coarser level

void **prolong_swe_to_comp**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int **nxs*, int **nys*, int **nzs*, float *dz*, float *dt*, float *zmin*, float **rho*, float *gamma*, int **matching_indices_d*, int *ng*, int *rank*, float **qc_comp*, float **old_phi_d*, int *coarse_level*, float *R*)

Prolong coarse grid data to fine grid

**Parameters**

- `kernelsthreadsblocks`: number of kernels, threads and blocks for each process/kernel
- `cumulative_kernels`: cumulative number of kernels in mpi processes of r < rank
- `q_cdq_fd`: coarse and fine grids of state vectors
- `nxsnysnzs`: dimensions of grids
- `ng`: number of ghost cells
- `dz`: coarse grid cell vertical spacing
- `dt`: timestep
- `zmin`: height of sea floor
- `rhogamma`: density and adiabatic index
- `matching_indices_d`: position of fine grid wrt coarse grid
- `ng`: number of ghost cells
- `rank`: rank of MPI process
- `qc_comp`: grid of compressible variables on coarse grid
- `old_phi_d`: Phi at previous timstep
- `coarse_level`: index of coarser level

**__global__ void prolong_reconstruct_comp(float * q_f, float * q_c, int * nxs, int * nys, i**

Reconstruct fine grid variables from compressible variables on coarse grid

**Parameters**

- `q_comp`: compressible variables on coarse grid
- `q_f`: fine grid state vector
- `q_c`: coarse grid swe state vector
- `nxsnysnzs`: grid dimensions
- `ng`: number of ghost cells
- `matching_indices_d`: position of fine grid wrt coarse grid
- `kx_offsetky_offset`: kernel offsets in the x and y directions
- `clevel`: index of coarser level

void **prolong_comp_to_comp**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int **nxs*, int **nys*, int **nzs*, int **matching_indices_d*, int *ng*, int *rank*, int *coarse_level*)

Prolong coarse grid data to fine grid

**Parameters**

- `kernelsthreadsblocks`: number of kernels, threads and blocks for each process/kernel

- cumulative_kernels: cumulative number of kernels in mpi processes of r < rank
- q_cdq_fd: coarse and fine grids of state vectors
- nxsnysnzs: dimensions of grids
- matching_indices_d: position of fine grid wrt coarse grid
- ng: number of ghost cells
- rank: rank of MPI process
- coarse_level: index of coarser level

**__global__ void prolong_reconstruct_swe_from_swe(float * qf, float * qc, int * nxs, int * n**

Reconstruct multilayer swe fine grid variables from single layer swe variables on coarse grid

**Parameters**

- q_f: fine grid state vector
- q_c: coarse grid swe state vector
- nxsnysnzs: grid dimensions
- ng: number of ghost cells
- matching_indices_d: position of fine grid wrt coarse grid
- kx_offsetky_offset: kernel offsets in the x and y directions
- clevel: index of coarser level

void **prolong_swe_to_swe**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int **nxs*, int **nys*, int **nzs*, int **matching_indices_d*, int *ng*, int *rank*, int *coarse_level*)

Prolong coarse grid single layer swe data to fine multilayer swe grid.

**Parameters**

- kernelsthreadsblocks: number of kernels, threads and blocks for each process/kernel
- cumulative_kernels: cumulative number of kernels in mpi processes of r < rank
- q_cdq_fd: coarse and fine grids of state vectors
- nxsnysnzs: dimensions of grids
- matching_indices_d: position of fine grid wrt coarse grid
- ng: number of ghost cells
- rank: rank of MPI process
- coarse_level: index of coarser level

**__global__ void prolong_reconstruct_multiswe_from_multiswe(float * qf, float * qc, int * nx**

Reconstruct multilayer swe fine grid variables from multilayer swe variables on coarse grid

**Parameters**

- q_f: fine grid state vector
- q_c: coarse grid swe state vector
- nxsnysnzs: grid dimensions
- ng: number of ghost cells

- matching_indices_d: position of fine grid wrt coarse grid
- kx_offsetky_offset: kernel offsets in the x and y directions
- clevel: index of coarser level

void **prolong_multiswe_to_multiswe**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int **nxs*, int **nys*, int **nzs*, int **matching_indices_d*, int *ng*, int *rank*, int *coarse_level*)

Prolong coarse grid multilayer swe data to fine multilayer swe grid.

**Parameters**

- kernelsthreadsblocks: number of kernels, threads and blocks for each process/kernel
- cumulative_kernels: cumulative number of kernels in mpi processes of r < rank
- q_cdq_fd: coarse and fine grids of state vectors
- nxsnysnzs: dimensions of grids
- matching_indices_d: position of fine grid wrt coarse grid
- ng: number of ghost cells
- rank: rank of MPI process
- coarse_level: index of coarser level

**__global__ void calc_comp_prim(float * q, int * nxs, int * nys, int * nzs, float gamma, int**

Calculates the SWE state vector from the compressible variables.

**Parameters**

- q: grid of compressible state vector
- q_swe: grid where SWE state vector to be stored
- nxsnysnzs: grid dimensions
- rhogamma: density and adiabatic index
- kx_offsetky_offset: kernel offsets in the x and y directions
- qc: coarse grid
- matching_indices: indices of fine grid wrt coarse grid
- coarse_level: index of coarser grid

**__global__ void swe_from_compressible(float * q_prim, float * q_swe, int * nxs, int * nys,**

**__global__ void restrict_interpolate_swe(float * p_const, float gamma, float * q_comp, floa**

Interpolate SWE variables on fine grid to get them on coarse grid.

**Parameters**

- p_const: pressure on SWE surfaces
- adiabatic: index
- q_comp: primitive compressible state vector on grid
- q_swe: conserved SWE state vector on grid
- zmin: height of bottom layer

- dz: compressible grid separation
- nxsnysnzs: grid dimensions
- matching_indices: position of fine grid wrt coarse grid
- kx_offsetky_offset: kernel offsets in the x and y directions
- coarse_level: index of coarser level

void **restrict_comp_to_swe**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int **nxs*, int **nys*, int **nzs*, float *dz*, float *zmin*, int **matching_indices*, float **rho*, float *gamma*, int *ng*, int *rank*, float **qf_swe*, int *coarse_level*, float **p_const*, float *R*, float *alpha0*)

Restrict fine grid data to coarse grid

**Parameters**

- kernelsthreadsblocks: number of kernels, threads and blocks for each process/kernel
- cumulative_kernels: cumulative number of kernels in mpi processes of r < rank
- q_cdq_fd: coarse and fine grids of state vectors
- nxsnysnzs: dimensions of grids
- matching_indices: position of fine grid wrt coarse grid
- rhogamma: density and adiabatic index
- ng: number of ghost cells
- rank: rank of MPI process
- qf_swe: grid of SWE variables on fine grid
- coarse_level: index of coarser level

**__global__ void restrict_interpolate_comp(float * qf, float * qc, int * nxs, int * nys, int**

Interpolate fine grid compressible variables to get them on coarser compressible grid.

**Parameters**

- qf: variables on fine grid
- qc: coarse grid state vector
- nxsnysnzs: grid dimensions
- ng: number of ghost cells
- matching_indices: position of fine grid wrt coarse grid
- kx_offsetky_offset: kernel offsets in the x and y directions
- clevel: index of coarser level

void **restrict_comp_to_comp**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int **nxs*, int **nys*, int **nzs*, int **matching_indices*, int *ng*, int *rank*, int *coarse_level*)

Restrict fine compressible grid data to coarse compressible grid.

**Parameters**

- kernelsthreadsblocks: number of kernels, threads and blocks for each process/kernel

- `cumulative_kernels`: cumulative number of kernels in mpi processes of r < rank
- `q_cdq_fd`: coarse and fine grids of state vectors
- `nxsnysnzs`: dimensions of grids
- `matching_indices`: position of fine grid wrt coarse grid
- `ng`: number of ghost cells
- `rank`: rank of MPI process
- `coarse_level`: index of coarser level

**__global__ void restrict_interpolate_swe_to_swe(float * qf, float * qc, int * nxs, int * ny**

Interpolate multilayer SWE variables on fine grid to get them on single layer SWE coarse grid.

**Parameters**

- `qf`: variables on fine grid
- `qc`: coarse grid state vector
- `nxsnysnzs`: grid dimensions
- `ng`: number of ghost cells
- `matching_indices`: position of fine grid wrt coarse grid
- `kx_offsetky_offset`: kernel offsets in the x and y directions
- `clevel`: index of coarser level

void **restrict_swe_to_swe**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int **nxs*, int **nys*, int **nzs*, int **matching_indices*, int *ng*, int *rank*, int *coarse_level*)

Restrict fine multilayer swe grid data to coarse single layer swe grid.

**Parameters**

- `kernelsthreadsblocks`: number of kernels, threads and blocks for each process/kernel
- `cumulative_kernels`: cumulative number of kernels in mpi processes of r < rank
- `q_cdq_fd`: coarse and fine grids of state vectors
- `nxsnysnzs`: dimensions of grids
- `matching_indices`: position of fine grid wrt coarse grid
- `ng`: number of ghost cells
- `rank`: rank of MPI process
- `coarse_level`: index of coarser level

**__global__ void restrict_interpolate_multiswe_to_multiswe(float * qf, float * qc, int * nxs**

Interpolate multilayer SWE variables on fine grid to get them on multilayer SWE coarse grid.

**Parameters**

- `qf`: variables on fine grid
- `qc`: coarse grid state vector
- `nxsnysnzs`: grid dimensions
- `ng`: number of ghost cells

- matching_indices: position of fine grid wrt coarse grid
- kx_offsetky_offset: kernel offsets in the x and y directions
- clevel: index of coarser level

void **restrict_multiswe_to_multiswe**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int **nxs*, int **nys*, int **nzs*, int **matching_indices*, int *ng*, int *rank*, int *coarse_level*)

Restrict fine multilayer swe grid data to coarse multilayer swe grid.

**Parameters**

- kernelsthreadsblocks: number of kernels, threads and blocks for each process/kernel
- cumulative_kernels: cumulative number of kernels in mpi processes of r < rank
- q_cdq_fd: coarse and fine grids of state vectors
- nxsnysnzs: dimensions of grids
- matching_indices: position of fine grid wrt coarse grid
- ng: number of ghost cells
- rank: rank of MPI process
- coarse_level: index of coarser level

void **interpolate_rhos**(float **rho_column*, float **rho_grid*, float *zmin*, float *zmax*, float *dz*, float **phs*, int *nx*, int *ny*, int *nz*)

**__global__ void evolve_fv(float * Un_d, flux_func_ptr flux_func, float * qx_plus_half, floa**

First part of evolution through one timestep using finite volume methods. Reconstructs state vector to cell boundaries using slope limiter and calculates fluxes there.

NOTE: we assume that beta is smooth so can get value at cell boundaries with simple averaging

**Parameters**

- Un_d: state vector at each grid point in each layer
- flux_func: pointer to function to be used to calulate fluxes
- qx_plus_halfqx_minus_half: state vector reconstructed at right and left boundaries
- qy_plus_halfqy_minus_half: state vector reconstructed at top and bottom boundaries
- fx_plus_halffx_minus_half: flux vector at right and left boundaries
- fy_plus_halffy_minus_half: flux vector at top and bottom boundaries
- nxnynz: dimensions of grid
- alphagamma: lapse function and adiabatic index
- kx_offsetky_offset: x, y offset for current kernel

**__global__ void evolve_z(float * Un_d, flux_func_ptr flux_func, float * qz_plus_half, float**

First part of evolution through one timestep using finite volume methods. Reconstructs state vector to cell boundaries using slope limiter and calculates fluxes there.

NOTE: we assume that beta is smooth so can get value at cell boundaries with simple averaging

**Parameters**

- Un_d: state vector at each grid point in each layer
- flux_func: pointer to function to be used to calculate fluxes
- qz_plus_halfqz_minus_half: state vector reconstructed at top and bottom boundaries
- fz_plus_halffz_minus_half: flux vector at top and bottom boundaries
- nxnynz: dimensions of grid
- vec_dim: dimension of state vector
- alphagamma: lapse function and adiabatic index
- kx_offsetky_offset: x, y offset for current kernel

**__global__ void evolve_fv_fluxes(float * F, float * qx_plus_half, float * qx_minus_half, fl**
Calculates fluxes in finite volume evolution by solving the Riemann problem at the cell boundaries.

**Parameters**

- F: flux vector at each point in grid and each layer
- qx_plus_halfqx_minus_half: state vector reconstructed at right and left boundaries
- qy_plus_halfqy_minus_half: state vector reconstructed at top and bottom boundaries
- fx_plus_halffx_minus_half: flux vector at right and left boundaries
- fy_plus_halffy_minus_half: flux vector at top and bottom boundaries
- nxnynz: dimensions of grid
- vec_dim: dimension of state vector
- alpha: lapse function
- dxdydt: gridpoint spacing and timestep spacing
- kx_offsetky_offset: x, y offset for current kernel

**__global__ void evolve_z_fluxes(float * F, float * qz_plus_half, float * qz_minus_half, flo**
Calculates fluxes in finite volume evolution by solving the Riemann problem at the cell boundaries in z direction.

**Parameters**

- F: flux vector at each point in grid and each layer
- qz_plus_halfqz_minus_half: state vector reconstructed at right and left boundaries
- fz_plus_halffz_minus_half: flux vector at top and bottom boundaries
- nxnynz: dimensions of grid
- vec_dim: dimension of state vector
- alpha: lapse function
- dzdt: gridpoint spacing and timestep spacing
- kx_offsetky_offset: x, y offset for current kernel

**__global__ void grav_sources(float * q, float gamma, int nx, int ny, int nz, int vec_dim, i**
Calculate gravitational source terms

**__global__ void evolve_fv_heating(float * Up, float * U_half, float * qx_plus_half, float ***
Does the heating part of the evolution.

**Parameters**

- `Up`: state vector at next timestep
- `U_half`: state vector at half timestep
- `qx_plus_halfqx_minus_half`: state vector reconstructed at right and left boundaries
- `qy_plus_halfqy_minus_half`: state vector reconstructed at top and bottom boundaries
- `fx_plus_halffx_minus_half`: flux vector at right and left boundaries
- `fy_plus_halffy_minus_half`: flux vector at top and bottom boundaries
- `sum_phs`: sum of Phi in different layers
- `rho_d`: list of densities in different layers
- `Q_d`: heating rate in each layer
- `nxnynlayers`: dimensions of grid
- `alphagamma`: lapse function and adibatic index
- `dxdydt`: gridpoint spacing and timestep spacing
- `burning`: is burning present in this system?
- `Cv`: specific heat in constant volume
- `E_He`: energy release per unit mass of helium
- `kx_offsetky_offset`: x, y offset for current kernel

**__global__ void evolve2(float * Un_d, float * Up, float * U_half, float * sum_phs, int nx,**

Adds buoyancy terms.

**Parameters**

- `Un_d`: state vector at each grid point in each layer at current timestep
- `Up`: state vector at next timestep
- `U_half`: state vector at half timestep
- `sum_phs`: sum of Phi in different layers
- `nxnynlayers`: dimensions of grid
- `ng`: number of ghost cells
- `alpha`: lapse function
- `dxdydt`: gridpoint spacing and timestep spacing
- `kx_offsetky_offset`: x, y offset for current kernel

void **homogeneuous_fv**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **Un_d*, float **F_d*, float **qx_p_d*, float **qx_m_d*, float **qy_p_d*, float **qy_m_d*, float **qz_p_d*, float **qz_m_d*, float **fx_p_d*, float **fx_m_d*, float **fy_p_d*, float **fy_m_d*, float **fz_p_d*, float **fz_m_d*, int *nx*, int *ny*, int *nz*, int *vec_dim*, int *ng*, float *alpha0*, float *gamma*, float *dx*, float *dy*, float *dz*, float *dt*, int *rank*, float *zmin*, float *R*, *flux_func_ptr* *h_flux_func*, bool *do_z*)

Solves the homogeneous part of the equation (ie the bit without source terms).

**Parameters**

- `kernelsthreadsblocks`: number of kernels, threads and blocks for each process/kernel

- `cumulative_kernels`: Cumulative total of kernels in ranks < rank of current MPI process
- `Un_d`: state vector at each grid point in each layer at current timestep
- `F_d`: flux vector
- `qx_p_dqx_m_d`: state vector reconstructed at right and left boundaries
- `qy_p_dqy_m_d`: state vector reconstructed at top and bottom boundaries
- `fx_p_dfx_m_d`: flux vector at right and left boundaries
- `fy_p_dfy_m_d`: flux vector at top and bottom boundaries
- `nxnynz`: dimensions of grid
- `alphagamma`: lapse function and adiabatic index
- `dxdydzdt`: gridpoint spacing and timestep spacing
- `rank`: rank of MPI process
- `flux_func`: pointer to function to be used to calculate fluxes
- `do_z`: should we evolve in the z direction?

void **rk3**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **Un_d*, float **F_d*, float **Up_d*, float **qx_p_d*, float **qx_m_d*, float **qy_p_d*, float **qy_m_d*, float **qz_p_d*, float **qz_m_d*, float **fx_p_d*, float **fx_m_d*, float **fy_p_d*, float **fy_m_d*, float **fz_p_d*, float **fz_m_d*, int *level*, int **nxs*, int **nys*, int **nzs*, int **vec_dims*, int *ng*, float *alpha0*, float *R*, float *gamma*, float *dx*, float *dy*, float *dz*, float *dt*, float **Up_h*, float **F_h*, float **Un_h*, MPI_Comm *comm*, MPI_Status *status*, int *rank*, int *n_processes*, *flux_func_ptr flux_func*, bool *do_z*, bool *periodic*, int *m_in*, float **U_swe*, int **matching_indices*, float *zmin*)

Integrates the homogeneous part of the ODE in time using RK3.

**Parameters**

- `kernelsthreadsblocks`: number of kernels, threads and blocks for each process/kernel
- `cumulative_kernels`: Cumulative total of kernels in ranks < rank of current MPI process
- `Un_d`: state vector at each grid point in each layer at current timestep on device
- `F_d`: flux vector on device
- `Up_d`: state vector at next timestep on device
- `qx_p_dqx_m_d`: state vector reconstructed at right and left boundaries
- `qy_p_dqy_m_d`: state vector reconstructed at top and bottom boundaries
- `fx_p_dfx_m_d`: flux vector at right and left boundaries
- `fy_p_dfy_m_d`: flux vector at top and bottom boundaries
- `nxnynz`: dimensions of grid
- `vec_dim`: dimension of state vector
- `ng`: number of ghost cells
- `alphagamma`: lapse function and adiabatic index
- `dxdydzdt`: gridpoint spacing and timestep spacing
- `Up_hF_hUn_h`: state vector at next timestep, flux vector and state vector at current timestep on host
- `comm`: MPI communicator

- `status`: status of MPI processes
- `rankn_processes`: rank of current MPI process and total number of MPI processes
- `flux_func`: pointer to function to be used to calculate fluxes
- `do_z`: should we evolve in the z direction?
- `periodic`: do we use periodic or outflow boundary conditions?

void **cuda_run**(float **beta*, float ***Us_h*, float **rho*, float **Q*, int **nxs*, int **nys*, int **nzs*, int *nlevels*, char **models*, int **vec_dims*, int *ng*, int *nt*, float *alpha0*, float *R*, float *gamma*, float *E_He*, float *Cv*, float *zmin*, float *dx*, float *dy*, float *dz*, float *dt*, bool *burning*, bool *periodic*, int *dprint*, char **filename*, char **param_filename*, MPI_Comm *comm*, MPI_Status *status*, int *rank*, int *n_processes*, int **matching_indices*, int *r*, int *n_print_levels*, int **print_levels*, int *tstart*, float **p_const*)

Evolve system through nt timesteps, saving data to filename every dprint timesteps.

**Parameters**

- `beta`: shift vector at each grid point
- `gamma_up`: gamma matrix at each grid point
- `rho`: densities in each layer
- `Q`: heating rate at each point and in each layer
- `nxsnysnzs`: dimensions of grids
- `nlevels`: number of levels of mesh refinement
- `models`: Array describing the physical model to use on each level. S = single layer SWE, M = multilayer SWE, C = compressible, L = Low Mach
- `vec_dims`: Dimensions of state vectors on each grid Us_h Array of pointers to grids.
- `ng`: number of ghost cells
- `nt`: total number of timesteps
- `alpha0`: lapse function at sea floor
- `R`: radius of star
- `gamma`: adiabatic index
- `E_He`: energy release per unit mass of helium burning
- `Cv`: specific heat per unit volume
- `zmin`: height of sea floor
- `dxdydzdt`: gridpoint spacing and timestep spacing
- `periodic`: do we use periodic or outflow boudary conditions?
- `burning`: is burning included in this system?
- `dprint`: number of timesteps between each printout
- `filename`: name of file to which output is printed
- `param_filename`: name of parameter file
- `comm`: MPI communicator
- `status`: status of MPI processes

- `rankn_processes`: rank of current MPI process and total number of MPI processes
- `matching_indices`: position of fine grid wrt coarse grid
- `r`: ratio of grid resolutions
- `n_print_levels`: number of levels to be output to file
- `print_levels`: numbers of the levels to be output to file
- `tstart`: start timestep
- `p_const`: pressures on multilayer SWE grids

```
__global__ void test_find_height(bool * passed)
__global__ void test_find_pot(bool * passed)
__global__ void test_rhoh_from_p(bool * passed)
__global__ void test_p_from_rhoh(bool * passed)
__global__ void test_p_from_rho_eps(bool * passed)
__global__ void test_hdot(bool * passed)
__global__ void test_calc_As(bool * passed)
__global__ void test_cons_to_prim_comp_d(bool * passed, float * q_prims)
__global__ void test_shallow_water_fluxes(bool * passed)
__global__ void test_compressible_fluxes(bool * passed)
__global__ void test_p_from_swe(bool * passed)
```

### Variables

```
__constant__ float beta_d[3]
```

## run_mesh_cuda.cpp

### Functions

void **multiscale_test**(*Sea* *sea)

void **acoustic_wave**(*Sea* *sea)

int **main**(int *argc*, char *argv[])

## mesh_output.h

**Warning:** doxygenfile: Cannot find file "mesh_output.h

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Index