swerve Documentation

Release

Alice Harpole

Feb 24, 2017

Contents:

Shallow **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 GPU's/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

Testing

A set of tests can be compiled by 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 3

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 *_nz*, int *_nlayers*, 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 *_mu*, float *_gamma*, float *_alpha*, float *_beta*, float *_gamma_down*, bool *_periodic*, bool *_burning*, int *_dprint*)
Constructor from list of parameters.

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

void **`initial_data`**(float **D0*, float **Sx0*, float **Sy0*)
Initialise D, Sx, Sy and Q.

Parameters

- `D0`: conserved density

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

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*)
Run simulation.

Parameters

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

~Sea ()
Deconstructor

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 **nz**
number of gridpoints in z-direction of fine (compressible) grid

int **nlayers**
Number of shallow water layers

int **ng**
Number of ghost cells

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 coarse gridpoints

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

float ***U_coarse**
Conserved shallow water variables on coarse grid

float ***U_fine**
Conserved compressible variables on fine grid.

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 Members

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

int **r**
refinement ratio

int **nxf**
Number of gridpoints in x-direction on fine grid

int **nyf**
Number of gridpoints in y-direction on fine grid

int **matching_indices**[2 *2]
Location of fine grid wrt coarse grid coordinates

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

float **dy**
Gridpoint separation in y-direction on coarse 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 **mu**
Friction

float **gamma**
Adiabatic index

float **alpha**
Lapse function

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

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

mesh_cuda_kernel.h

Typedefs

typedef void (***flux_func_ptr**) (float *q, float *f, int dir, float *gamma_up, float alpha, float *beta, float gamma)

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 tol
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

- gamma_up: spatial metric

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*)
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

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*)
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

__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)

__device__ float find_height(float ph)
Finds r given Phi.

__device__ float find_pot(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

__device__ __host__ 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
- `gamma_up`: spatial metric

__device__ float h_dot(float phi, float old_phi, float dt)
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__ 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 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
- `gamma_up`: spatial metric

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`: contravariant spatial metric

__device__ void shallow_water_fluxes(float * q, float * f, int dir, float * gamma_up, float

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
- gamma_up: spatial metric
- alpha: lapse function
- beta: shift vector
- gamma: adiabatic index

__device__ void compressible_fluxes(float * q, float * f, int dir, float * gamma_up, float
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
- gamma_up: spatial metric
- alpha: lapse function
- beta: shift vector
- gamma: adiabatic index

void **p_from_swe**(float **q*, float **p*, int *nx*, int *ny*, int *nz*, float **gamma_up*, float *rho*, float *gamma*, float *A*)
Calculate p using SWE conserved variables

Parameters

- q: state vector
- p: grid where pressure shall be stored
- nxnynz: grid dimensions
- gamma_up: spatial metric
- rho: density
- gamma: adiabatic index
- A: variable required in p(Phi) calculation

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

Parameters

- q: state vector
- gamma_up: spatial metric
- 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 nx, int ny, int nz, fl
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
- nxnynz: grid dimensions
- gamma_up: spatial metric
- rhogamma: density and adiabatic index
- kx_offsetky_offset: kernel offsets in the x and y directions
- dt: timestep
- old_phi: Phi at previous timestep

__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(float * q_comp, float * q_f, float * q_c, int nx, int n
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
- nxnynlayers: coarse grid dimensions
- nxfnyfnz: fine grid dimensions
- dxdydz: coarse grid spacings
- matching_indices_d: position of fine grid wrt coarse grid
- gamma_up: spatial metric
- kx_offsetky_offset: kernel offsets in the x and y directions

void **prolong_grid**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int *nx*, int *ny*, int *nlayers*, int *nxf*, int *nyf*, int *nz*, float *dx*, float *dy*, float *dz*, float *dt*, float *zmin*, float **gamma_up_d*, float **rho*, float *gamma*, int **matching_indices_d*, int *ng*, int *rank*, float **qc_comp*, float **old_phi_d*)
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
- nxnynlayers: dimensions of coarse grid
- nxfnyfnz: dimensions of fine grid
- dxdydz: coarse grid cell spacings

- dt: timestep
- zmin: height of sea floor
- gamma_up_d: spatial metric
- 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

__global__ void swe_from_compressible(float * q, float * q_swe, int nx, int ny, int nxf, in

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
- nxfnyfnz: grid dimensions
- gamma_up: spatial metric
- 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

__global__ void restrict_interpolate(float * qf_sw, float * q_c, int nx, int ny, int nlayer

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

Parameters

- qf_swe: SWE variables on fine grid
- q_c: coarse grid state vector
- nxnynlayers: coarse grid dimensions
- nxfnyfnz: fine grid dimensions
- matching_indices: position of fine grid wrt coarse grid
- gamma_up: spatial metric
- kx_offsetky_offset: kernel offsets in the x and y directions

void **restrict_grid**(dim3 **kernels*, dim3 **threads*, dim3 **blocks*, int **cumulative_kernels*, float **q_cd*, float **q_fd*, int *nx*, int *ny*, int *nlayers*, int *nxf*, int *nyf*, int *nz*, float *dz*, float *zmin*, int **matching_indices*, float **rho*, float *gamma*, float **gamma_up*, int *ng*, int *rank*, float **qf_swe*)

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
- `nxnynlayers`: dimensions of coarse grid
- `nxfnyfnz`: dimensions of fine grid
- `matching_indices`: position of fine grid wrt coarse grid
- `rhogamma`: density and adiabatic index
- `gamma_up`: spatial metric
- `ng`: number of ghost cells
- `rank`: rank of MPI process
- `qf_swe`: grid of SWE variables on fine grid

`__global__ void evolve_fv(float * beta_d, float * gamma_up_d, float * Un_d, flux_func_ptr f`

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

- `beta_d`: shift vector at each grid point.
- `gamma_up_d`: gamma matrix at each grid point
- `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
- `dxdydt`: grid dimensions and timestep
- `kx_offsetky_offset`: x, y offset for current kernel

`__global__ void evolve_z(float * beta_d, float * gamma_up_d, float * Un_d, flux_func_ptr fl`

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

- `beta_d`: shift vector at each grid point.
- `gamma_up_d`: gamma matrix at each grid point
- `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
- dzdt: vertical grid spacing and timestep
- 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 evolve_fv_heating(float * gamma_up_d, float * Up, float * U_half, float * c

Does the heating part of the evolution.

Parameters

- gamma_up_d: gamma matrix at each grid point
- 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
- mu: friction
- nxnynlayers: dimensions of grid
- alpha: lapse function
- dxdydt: gridpoint spacing and timestep spacing
- burning: is burning present in this system?
- 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 **beta_d*, float **gamma_up_d*, 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 *alpha*, float *gamma*, float *dx*, float *dy*, float *dz*, float *dt*, int *rank*, *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
- beta_d: shift vector at each grid point
- gamma_up_d: gamma matrix at each grid point
- 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 *beta_d, float *gamma_up_d, 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 *nx*, int *ny*, int *nz*, int *vec_dim*, int *ng*, float *alpha*, 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 h_flux_func*, bool *do_z*)

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
- beta_d: shift vector at each grid point
- gamma_up_d: gamma matrix at each grid point
- 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?

void **cuda_run**(float **beta*, float **gamma_up*, float **Uc_h*, float **Uf_h*, float **rho*, float **Q*, float *mu*, int *nx*, int *ny*, int *nlayers*, int *nxf*, int *nyf*, int *nz*, int *ng*, int *nt*, float *alpha*, float *gamma*, float *zmin*, float *dx*, float *dy*, float *dz*, float *dt*, bool *burning*, int *dprint*, char **filename*, MPI_Comm *comm*, MPI_Status *status*, int *rank*, int *n_processes*, int **matching_indices*)

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
- `Uc_h`: state vector at each grid point in each layer at current timestep on host in coarse grid
- `Uf_h`: state vector at each grid point in each layer at current timestep on host in fine grid
- `rho`: densities in each layer
- `Q`: heating rate at each point and in each layer
- `mu`: friction
- `nxnynlayers`: dimensions of coarse grid
- `nxfnyfnz`: dimensions of fine grid
- `ng`: number of ghost cells
- `nt`: total number of timesteps
- `alpha`: lapse function
- `gamma`: adiabatic index
- `zmin`: height of sea floor
- `dxdydzdt`: gridpoint spacing and timestep spacing
- `burning`: is burning included in this system?
- `dprint`: number of timesteps between each printout
- `filename`: name of file to which output is printed
- `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

__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)
```

run_mesh_cuda.cpp

Functions

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

namespace std

Includes main function to run mesh cuda simulation.

Compile with 'make mesh'.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index