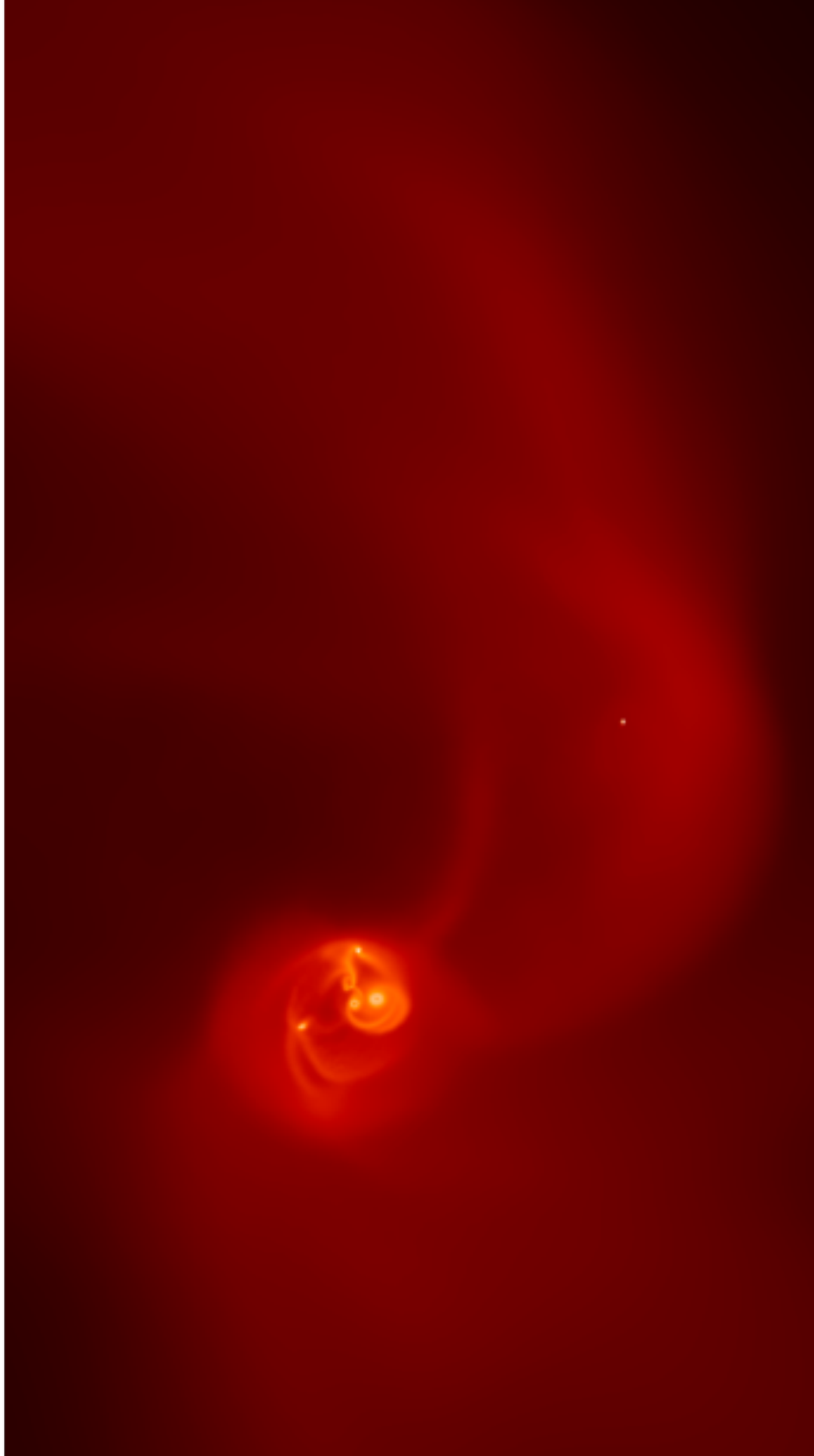
Generating initial conditions in GANDALF

David Hubber

USM, LMU, München
Excellence Cluster Universe,
Garching bei München

28th October 2015



Git update!

- Yesterday evening, I pushed an update to the master branch on the GANDALF github repository
- Solved several problems that some people brought to our attention
- If you'd like to, you can try to update your local version of GANDALF using git
- But if you're having no problems, then don't worry about it
- However, might be a good opportunity to try out updating while we're all here

Plan

- Quick overview of current initial condition (IC) generators in GANDALF
- Particle data structures in GANDALF
- Generic Hydrodynamical and N-body classes and how to allocate memory
- How to create a new subroutine for generating ICs in GANDALF
- How to use existing ‘helper functions’ to speed things up
- Practicals - writing your own IC functions in C++

Initial conditions in GANDALF

- There are several classes of initial conditions routines in GANDALF
- ‘Real’ Astronomy ICs
- Gravity tests
- Hydrodynamical tests
- N-body tests
- ‘Helper’ functions

IC class

```
template <int ndim>
class Ic
{
private:

  Simulation<ndim>* const sim;              ///< Simulation class pointer
  Hydrodynamics<ndim>* const hydro;         ///< Hydrodynamics algorithm pointer
  const FLOAT invndim;                      ///< 1/ndim
  const SimUnits& simunits;                 ///< Reference to main simunits object
  const DomainBox<ndim>& simbox;            ///< Reference to simulation bounding box object
  Parameters* simparams;                    ///< Pointer to parameters object
  RandomNumber *randnumb;                   ///< Random number object pointer

  // Helper routines
  //-----------------------------------------------------------------------------------------
  void AddAzimuthalDensityPerturbation(const int, const int, const FLOAT, const FLOAT *, FLOAT *);
  void AddBinaryStar(FLOAT, FLOAT, FLOAT, FLOAT, FLOAT, FLOAT, FLOAT, FLOAT, FLOAT, FLOAT,
                     FLOAT *, FLOAT *, NbodyParticle<ndim> &, NbodyParticle<ndim> &);
  etc…

public:

  Ic(Simulation<ndim>* sim_aux, Hydrodynamics<ndim>* hydro_aux, FLOAT invndim_aux) :
    sim(sim_aux), hydro(hydro_aux), invndim(invndim_aux),
    simunits(sim_aux->simunits), simbox(sim_aux->simbox),
    simparams(sim_aux->simparams), randnumb(sim_aux->randnumb)
  {
  };

  // Initial conditions routines
  //-----------------------------------------------------------------------------------------
  void BinaryAccretion(void);
  void BinaryStar(void);
  void BlastWave(void);
  void BondiAccretion(void);
  etc…

};
```

SimulationIC.hpp

```
template <int ndim>
void Simulation<ndim>::GenerateIC(void)
{
  string ic = simparams->stringparams["ic"];   // Local copy of initial conditions string

  // If not a restart, generate initial conditions either from external file or created on the fly.
  Ic<ndim> icGenerator(this, hydro, invndim);

  if (ic == "file") {
    ReadSnapshotFile(simparams->stringparams["in_file"], simparams->stringparams["in_file_form"]);
    rescale_particle_data = true;
    this->initial_h_provided = false;
  }
  else if (ic == "bb") {
    icGenerator.BossBodenheimer();
  }
  else if (ic == "binary") {
    icGenerator.BinaryStar();
  }
  else if (ic == "binaryacc") {
    icGenerator.BinaryAccretion();
  }
  etc…

  else if (ic == "python") {
    return;
  }
  else {
    string message = "Unrecognised parameter : ic = " + ic;
    ExceptionHandler::getIstance().raise(message);
  }

  // Check that the initial conditions are valid
  icGenerator.CheckInitialConditions();

 return;
}
```

How to add a new IC function

- Add your own function definition/prototype in the Ic.h file
- Add the function code itself in the Ic.cpp file
- Add an extra ‘if’ option to call your new IC function in the Simulation::GenerateIC function

Practical 1 : Add your own IC function

- **Create a new branch of GANDALF via git** for you to experiment in (you might need to commit to your original master branch if you have made changes there)
- Change to the new branch
- Open the **Ic.hpp**, **Ic.cpp** and **SimulationIc.hpp** files
- Create an empty IC function in the Ic.cpp file (perhaps add some lines to print to screen and then exit the program for now)
- Add the correct subroutine prototypes and calls to Ic.hpp and SimulationIc.hpp
- Check that the code compiles
- Generate a small parameters file calling your minimalistic IC routine and run the code to confirm it is called correctly

Inherited particle data

- In GANDALF, we define a base particle class that contains the most basic particle data that is used in all hydrodynamical classes (defined in **src/Headers/Particle.h**)

```
template <int ndim>
struct Particle
{
  bool active;
  bool potmin;
  int iorig;
  int itype;
  int sinkid;
  int levelneib;
  int nstep;
  int nlast;
  int level;
  FLOAT r[ndim];
  FLOAT v[ndim];
  FLOAT a[ndim];
  FLOAT r0[ndim];
  FLOAT v0[ndim];
  FLOAT a0[ndim];
  FLOAT agrav[ndim];
  FLOAT m;
  FLOAT h;
  …
};
```

```
template <int ndim>
struct SphParticle : public Particle<ndim>
{
  FLOAT pfactor;
  FLOAT div_v;
  FLOAT alpha;
  FLOAT dalphadt;
};
```

```
template <int ndim>
struct MeshlessFVParticle : public Particle<ndim>
{
  FLOAT invh
  FLOAT hfactor;
  FLOAT invrho;
  FLOAT invomega;
  FLOAT zeta;
  …
  FLOAT B[ndim][ndim];
  FLOAT Wprim[ndim+2];
  FLOAT Wmin[ndim+2];
  FLOAT Wmax[ndim+2];
  …
};
```

Storing particle data in GANDALF

- Before working with the actual particle data in GANDALF, **we have to allocate the memory** that we need :

```
hydro->Nhydro = 10000;
nbody->Nstar  = 0;
sim->AllocateParticleMemory();
```

- GANDALF stores particle data in **a simple array which can ONLY be accessed directly from within the hydrodynamics class that uses it**.
  - The GradhSph class can access ONLY the GradhSphParticle array
  - The MeshlessFV class can access ONLY the MeshlessFVParticle array
- This presents a small problem because we don't want to have write the SAME initial conditions code for every hydrodynamics case

‘GetParticlePointer’ function

- To **access a particle from ANY function outside the Hydrodynamics class**, you can use the GetParticle Pointer function :

```
Particle<ndim>& GetParticlePointer(const int i)
```

- This can be used simply to, for example, loop over all particles :

```
for (i=0; i<Nbox; i++) {
  Particle<ndim>& part = hydro->GetParticlePointer(i);
  for (k=0; k<ndim; k++) part.r[k] = r[ndim*i + k];
  for (k=0; k<ndim; k++) part.v[k] = (FLOAT) 0.0;
  part.m    = (FLOAT) 1.0/(FLOAT) Nbox;
  part.u    = press1/rhofluid1/gammaone;
}
```

Practical 2 : Add some particles

- Add some particles to your new IC function
- Use input parameters to set important variables (e.g. number of particles, size of the domain)
- Chose a simple distribution (e.g. a circular or cubic distribution of particles) and asign them basic properties (e.g. position, velocity, internal energy)
- Remember to allocate memory for the new particles
- Finally prepare a short parameters file to test that you can ‘run’ your initial conditions
- Plot the initial snapshot to check the initial conditions are what you intended

Helper functions

- The so-called ‘Helper functions’ are simple functions included in the IC class that can be used to generate common particle configurations or compute add

```
//=================================================================================
//  Ic::AddRandomBox
/// Populate given bounding box with random particles.
//=================================================================================
template <int ndim>
void Ic<ndim>::AddRandomBox
 (const int Npart,                          ///< [in] No. of particles
  const DomainBox<ndim> box,                ///< [in] Bounding box containing particles
  FLOAT *r)                                 ///< [out] Positions of particles
{
  debug2("[Ic::AddRandomBox]");
  assert(r);

  for (int i=0; i<Npart; i++) {
    for (int k=0; k<ndim; k++) {
      r[ndim*i + k] = box.boxmin[k] +
        (box.boxmax[k] - box.boxmin[k])*sim->randnumb->floatrand();
    }
  }

  return;
}
```

Practical 3 : Using the helper functions

- Create some initial conditions using the helper functions
- e.g. generate your circular or cubic particle distribution using **BOTH** a **random** distribution (e.g. **AddRandomSphere**, **AddRandomBox** functions) and a **lattice** (e.g. **AddLatticeSphere**, **AddCubicLattice** functions)
- Look at the difference in the density distributions between the random and lattice initial particle placements
- Run a simulation containing a random box of particles for several sound crossing times
- Look at the density distribution at various timesteps

Practical 4 : Using physical units

- Generate initial conditions of a spherical cloud of radius 1 pc and total mass 10 solar masses
- Give the particles a random velocity component in each direction of maximum magnitude 1km/s
- Run the simulation for a short time
- Check that the initial conditions are consistent in the first snapshot file