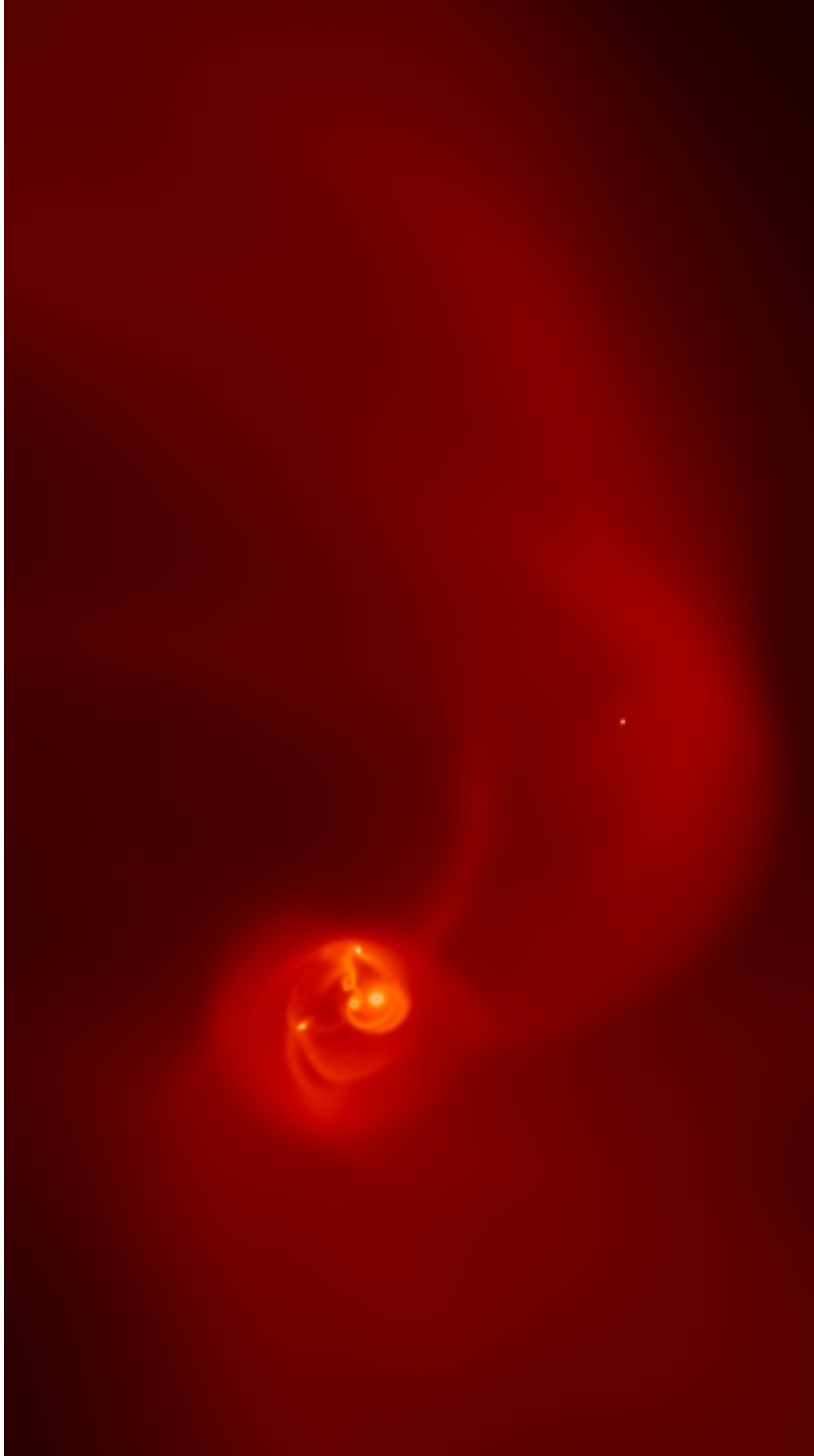
Adding new physics classes into GANDALF

David Hubber
Giovanni Rosotti

USM, LMU, München
Excellence Cluster Universe, Garching bei München, Germany;
University of Cambridge, UK

29th October 2015



Plan

- Adding physics classes obviously requires you to get to know the general structure of the GANDALF code a little better
- **No need to know everything about the C++ code of course**; just the classes you are changing/adding and how they interface to the relevant part of the code
- We will go over a few important parts of the GANDALF code structure
- Then **we will try a few small exercises adding in new classes into GANDALF** (and maybe even running with some basic ics)

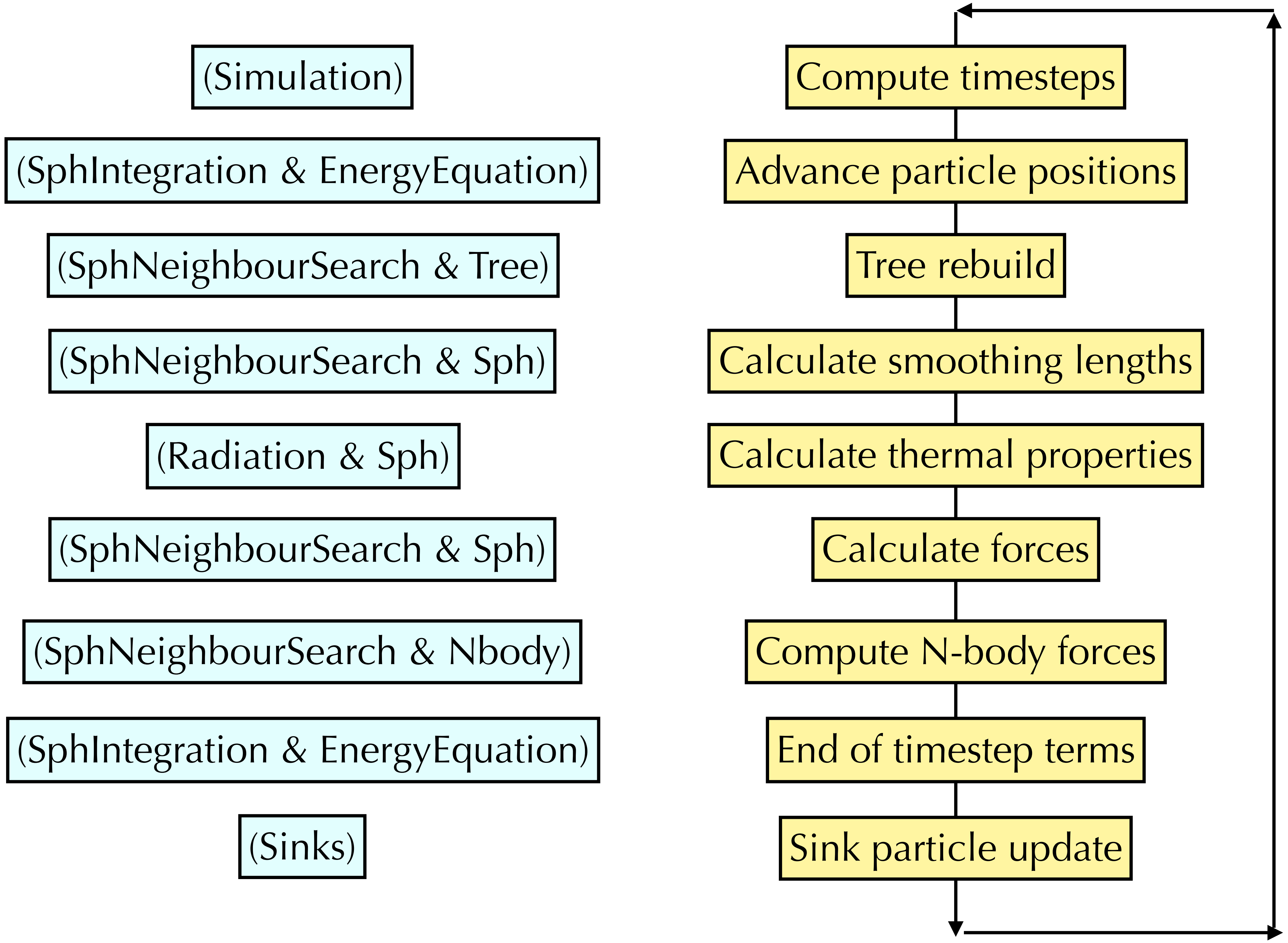
The GANDALF source directory

- The GANDALF source directory (gandalf/src) consists of several folders containing several file categories :

- Common
- GradhSph
- Headers
- Hydrodynamics
- MeshlessFV
- Mpi
- Nbody
- Radiation
- SM2013
- Thermal
- Tree
- UnitTesting

If you want to know more about the class structure in GANDALF, this is where you should look!

The SphSimulation Main Loop (simplified)


(Simulation)
Compute timesteps
(SphIntegration & EnergyEquation)
Advance particle positions
(SphNeighbourSearch & Tree)
Tree rebuild
(SphNeighbourSearch & Sph)
Calculate smoothing lengths
(Radiation & Sph)
Calculate thermal properties
(SphNeighbourSearch & Sph)
Calculate forces
(SphNeighbourSearch & Nbody)
Compute N-body forces
(SphIntegration & EnergyEquation)
End of timestep terms
(Sinks)
Sink particle update

Main classes

- SphIntegration: time integration (leapfrog, …)
- SphNeighbourSearch: sets-up the loops for smoothing length and force calculation
- Sph: contains the code that actually computes SPH quantities
- Nbody, sinks: self-explanatory
- Radiation: in case you are using radiative transfer
- The SPH Kernel is also a class!
- It's very important…
- …but you can't see it from the main loop
- If you want to add a new kernel in GANDALF it's very easy

Example : The Sph class

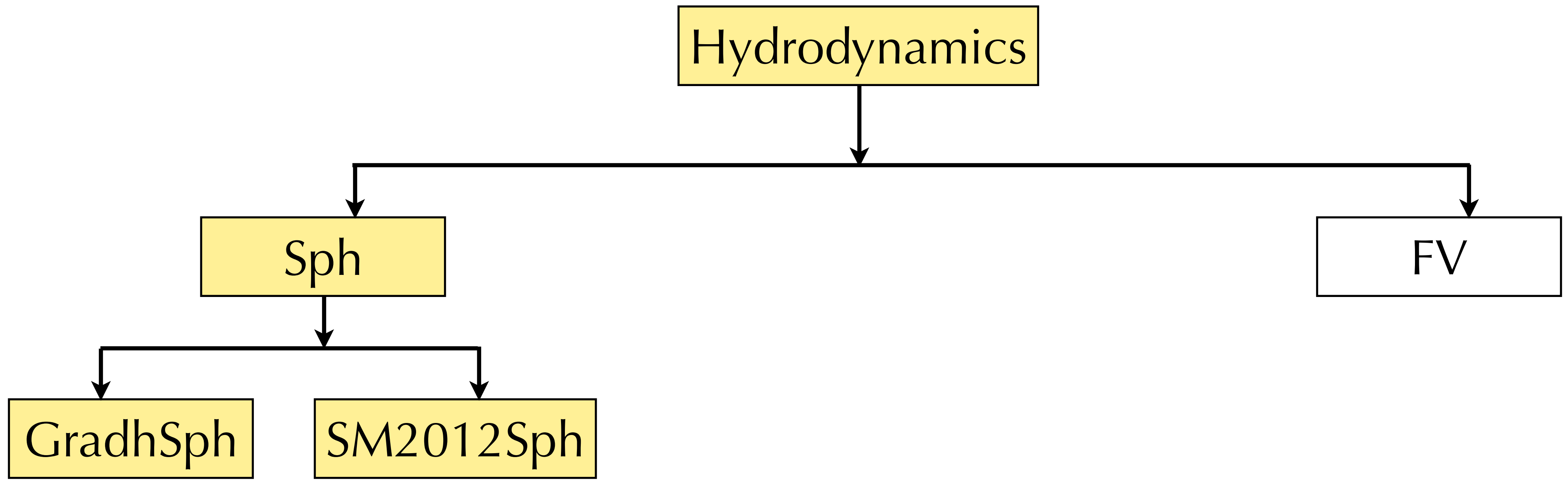
Used for example in the smoothing length calculation in GradhSphBruteForce.cpp (easier to understand than the tree version):

```
for (i=0; i<Nhydro; i++) {

    // Skip over inactive particles
    if (!sphdata[i].active || sphdata[i].itype == dead) continue;

    for (k=0; k<ndim; k++) rp[k] = sphdata[i].r[k];

    // Compute distances and the reciprical between the current particle and all neighbours here
    //---------------------------------------------------------------------------------------
    for (jj=0; jj<Nneib; jj++) {
        j = neiblist[jj];
        for (k=0; k<ndim; k++) dr[k] = sphdata[j].r[k] - rp[k];
        drsqd[jj] = DotProduct(dr,dr,ndim);
    }
    //---------------------------------------------------------------------------------------

    // Compute all SPH gather properties
    //okflag =
    sph->ComputeH(i,Nneib,big_number,m,mu,drsqd,gpot,sphdata[i],nbody);

}
```

Here we call sph to compute H

Example : The Sph class

- The Hydrodynamics/Sph class structure :



- Most important functions/data:

```
int ComputeH(const int, const int, const FLOAT, FLOAT *, FLOAT *, FLOAT *, FLOAT *,
            SphParticle<ndim> &, Nbody<ndim> *);
void ComputeThermalProperties(SphParticle<ndim> &);
void ComputeSphGravForces(const int, const int, int *, SphParticle<ndim> &, SphParticle<ndim> *);
void ComputeSphHydroGravForces(const int, const int, int *,
                               SphParticle<ndim> &, SphParticle<ndim> *);
void ComputeSphHydroForces(const int, const int, const int *, const FLOAT *, const FLOAT *,
                           const FLOAT *, SphParticle<ndim> &, SphParticle<ndim> *);
void ComputeSphNeibDudt(const int, const int, int *, FLOAT *, FLOAT *, FLOAT *,
                        SphParticle<ndim> &, SphParticle<ndim> *) {};
void ComputeSphDerivatives(const int, const int, int *, FLOAT *, FLOAT *, FLOAT *,
                           SphParticle<ndim> &, SphParticle<ndim> *) {};
void ComputeDirectGravForces(const int, const int, int *,
                             SphParticle<ndim> &, SphParticle<ndim> *);
void ComputeStarGravForces(const int, NbodyParticle<ndim> **, SphParticle<ndim> &);

kernelclass<ndim> kern;                ///< SPH kernel
GradhSphParticle<ndim> *sphdata;       ///< Pointer to particle data
```

Constructing the right objects!

- Depending on the value of the parameters, construct the right object in ProcessParameters (see example below)
- The function is defined in SphSimulation.cpp (if you use SPH).
- Parameters specific to the SPH flavour/Nbody are defined in ProcessNbodyParameters (defined in Simulation.cpp) or ProcessSphParameters (defined in GradhSphSimulation.cpp for GradhSph)

```
if (intparams["tabulated_kernel"] == 1) {
  sph = new GradhSph<ndim, TabulatedKernel>
   (intparams["hydro_forces"], intparams[“self_gravity"],
    floatparams[“alpha_visc"], floatparams["beta_visc"],
    floatparams["h_fac"], floatparams[“h_converge"], avisc, acond,
    tdavisc, stringparams["gas_eos"], KernelName, simunits, simparams);
}
```

Reading and processing parameters for your new physics class

- Initialise the variables you need in your constructor (e.g. in src/GradhSph/GradhSph.cpp)

```
template <int ndim, template<int> class kernelclass>
GradhSph<ndim, kernelclass>::GradhSph(int hydro_forces_aux, int self_gravity_aux,
                                      FLOAT alpha_visc_aux, FLOAT beta_visc_aux,
                                      FLOAT h_fac_aux, FLOAT h_converge_aux,
                                      aviscenum avisc_aux, acondenum acond_aux,
                                      tdaviscenum tdavisc_aux, string gas_eos_aux,
                                      string KernelName, SimUnits &units, Parameters *params):
Sph<ndim>(hydro_forces_aux, self_gravity_aux, alpha_visc_aux, beta_visc_aux,
          h_fac_aux, h_converge_aux, avisc_aux, acond_aux, tdavisc_aux,
          gas_eos_aux, KernelName, sizeof(GradhSphParticle<ndim>), units, params),
kern(kernelclass<ndim>(KernelName))
{
    this->kernp      = &kern;
    this->kernfac    = (FLOAT) 1.0;
    this->kernfacsqd = (FLOAT) 1.0;
    this->kernrange  = this->kernp->kernrange;
}
```

This is an initialisation list

Initialisation list

- Initialisation lists are used to :
  - Initialise ‘const’ variables in classes
  - Call the constructor of parent classes (if needed)

```
class Car {
  Car(int);
  ~Car();

  const int colour;
  bool automatic;
};
```

```
~~Car::Car(int _color) {~~
~~  color = _color;~~
~~  automatic = false;~~
~~};~~
```

```
Car::Car(int _color) : colour(_colour) {
  automatic = false;
};
```

```
class BatMobile() : public Car {
  BatMobile(bool, int);
  ~BatMobile();

  const bool flameThrower;
}
```

```
BatMobile::BatMobile(bool _flameThrower, int _color) :
  Car(_color), flameThrower(_flameThrower) {}
};
```

Creating your own classes

- Add the definition in the header files
- You probably want to inherit from one of the existing classes (remember: look in src/Headers)
- Implement your class in a cpp file
- If you add a new file, remember to add it to the makefile!
- Don't forget to initialise your object in ProcessParameters!
- Initialise all the variables you need in the constructor (don't blame me if you don't and you then have problems because of uninitialised variables)

Practical 1 : Adding a new unit into the SimUnits class

- One of the simplest classes used in GANDALF is the **SimUnit** class, which was discussed in the 'Units and scaling' talk
- Add a new SimUnit class of some new (potentially useful) quantity, e.g.
  - Kinematic viscosity
  - Specific entropy
- Remember to add a new parameter (in Parameters.cpp) in order to allow the user to change the new unit in the parameters file

Practical 2 : Adding a new external gravitational potential field

- Another simple class to add is to generate a new External Gravitational Potential field
- Open up the File src/Headers/ExternalPotential.h and read through the few implementations
- Create a new ExternalPotential class for some simple potential field, e.g.
  - Point source
  - Spiral galactic potential
  - NFW profile?
- Remember to edit the section of code that creates the ExternalPotential object to create it if the option is selected in the parameters file
- Note that this is an example of a class that exists exclusively in the header file. There is no need to create any '.cpp' file

Practical 3 : Add a new EOS class

- A slightly more complicated (but relatively simple) class to add is a new EOS (Equation of State) class
- This class contains several functions that need to be set to compute various thermal quantities, e.g. Pressure, Temperature, SoundSpeed, etc..
- Create a new EOS class for a, e.g.
    - a Polytropic Equation of State
    - Some other EOS you might need in the future
- Remember to add the new Object construction in the relevant place (the Sph constructor)