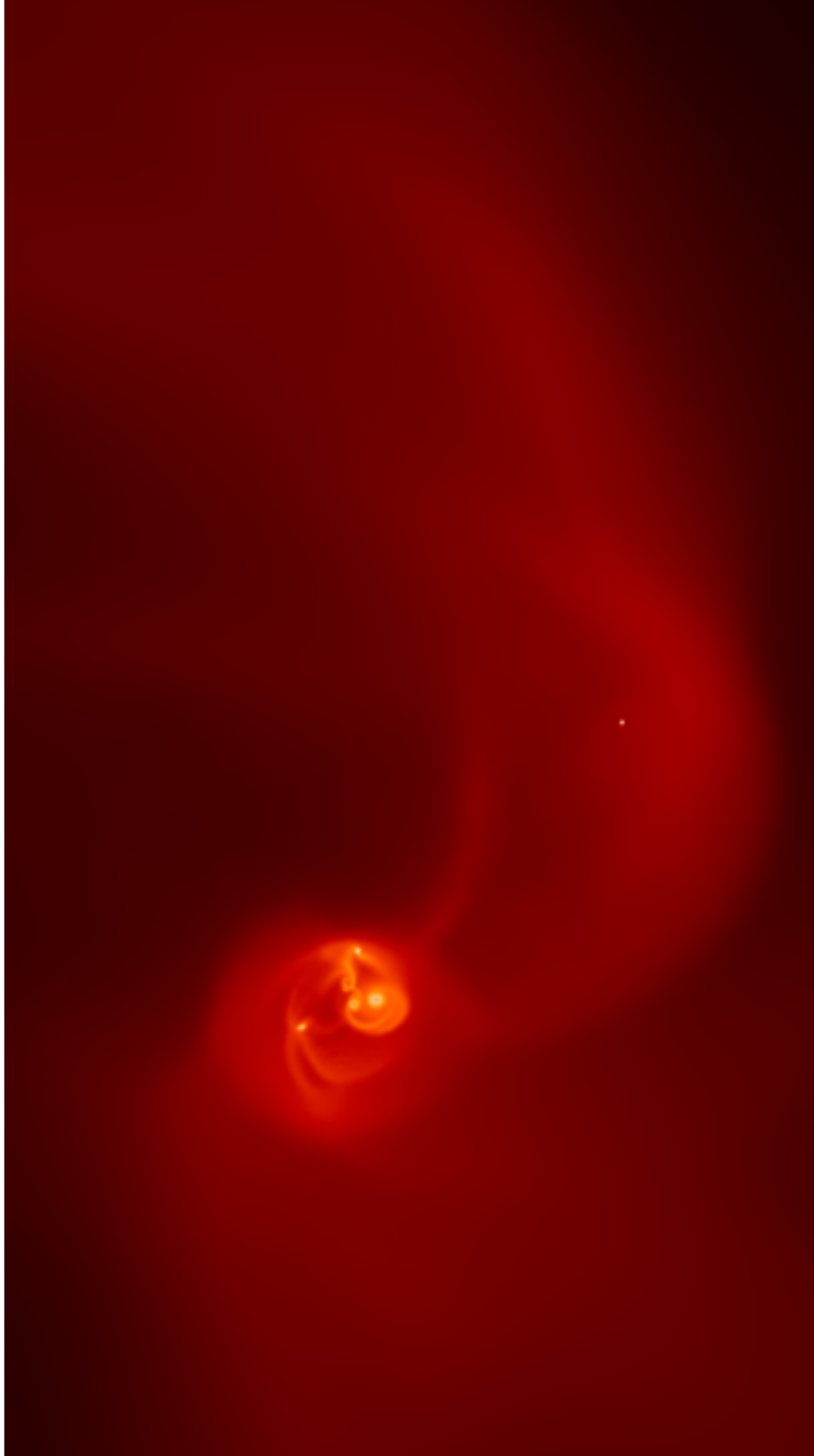
Running Simulations with GANDALF

David Hubber

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

27th October 2015



The GANDALF parameters file

- The GANDALF parameters file is used to control almost all other aspects of the simulation, the generation of initial conditions and of the algorithms used.
- The parameters file has a simple structure :
- There are **way too many** parameters to go through each in detail, so we'll just go over the broad categories of parameters available and concentrate on a few important ones

Core parameters

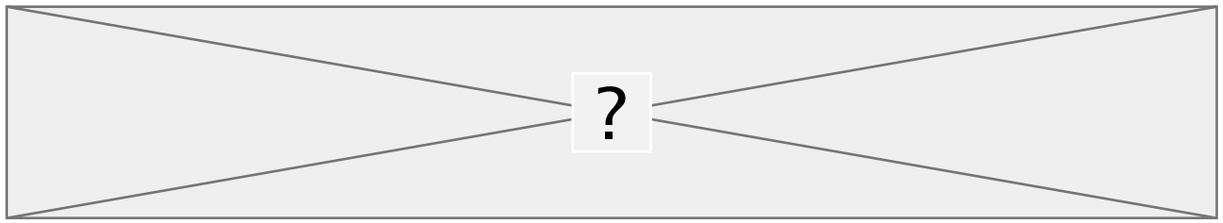
- `ndim` : Simulation dimensionality (1, 2 or 3)
- `sim` : Simulation type

sph	= SPH (+ N-body) algorithm (default : 'grad-h' SPH)
gradhsph	= 'grad-h' SPH simulation (+ N-body)
sm2012sph	= Saitoh & Makino (2012) SPH (+ N-body)
meshlessfv	= Meshless Finite-Volume algorithm (default : 'mfvmuscl')
mfvmuscl	= Meshless FV MUSCL integration simulation
mfvrk	= Meshless FV Runge-Kutta integration
nbody	= N-body only simulation

- **nbody** : Main N-body integration algorithm

lfkdk	= 2nd-order Leapfrog kick-drift-kick
lfdkd	= 2nd-order Leapfrog drift-kick-drift
hermite4	= 4th-order Hermite scheme
hermite4ts	= Time-symmetric 4th-order Hermite scheme

Core parameters



- `in_file_form` : Format of initial conditions file
  column = Simple column data format
  sf/seren_form = SEREN ASCII format
  su/seren_unform = SEREN binary format

- `out_file_form` : Format of outputted snapshot files
  column = Simple column data format
  sf/seren_form = SEREN ASCII format
  su/seren_unform = SEREN binary format

- `tend` : Termination time of the simulation (given in `tunits`)

- `dt_snap` : Snapshot time interval (given in `tunits`)

- `tsnapfirst` : Time of first snapshot (given in `tunits`)

Scaling parameters

- `dimensionless` : Are all quantities dimensionless? (0 or 1)

- `routunit` : Position unit

pc/kpc/mpc	= parsec/kiloparsec/megaparsec
au	= astronomical unit
r_sun	= Solar radius
r_earth	= Earth radius
cm/m/km	= centimetre/metre/kilometre

- `moutunit` : Mass unit

m_sun	= Solar mass
m_jup/m_earth	= Jupiter mass/Earth mass
g/kg	= gram/kilogram

- `toutunit` : Time unit

yr/myr/gyr	= year/megayear/gigayear
day	= day
sec	= second

- `voutunit` : Velocity unit

cm_s/m_s/km_s	= centimetres/metres/kilometres per second
au_yr	= astronomical units per year

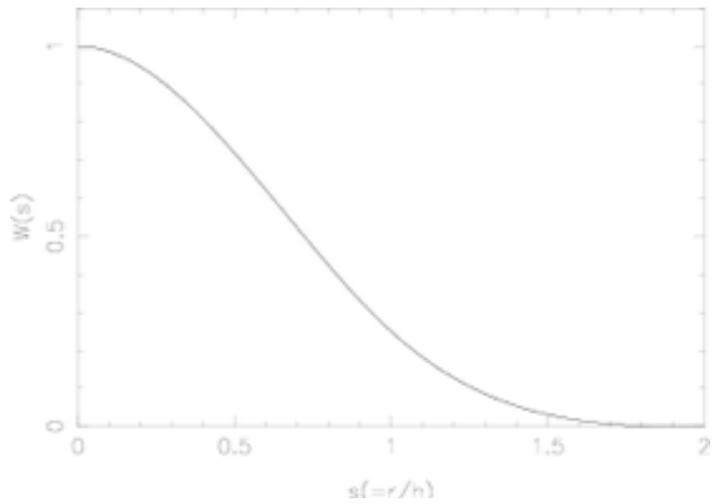
Hydrodynamical parameters

- `hydro_forces` : Compute hydro forces? (1 or 0)
- `gas_eos` : Gas particles equation-of-state
  - energy_eqn = Solve energy equation
  - isothermal = Isothermal EOS
  - barotropic = Barotropic EOS (i.e. for mimicing isothermal + adiabatic phase during protostellar collapse)
  - barotropic2 = Similar to barotropic, but using discrete power laws rather than smooth change
  - rad_ws = EOS relating to Stamatellos et al. (2007) cooling method
- `energy_integration` : Energy integration scheme (only applicable if solving the energy equation)
  - null = Energy equation not integrated separately
  - rad_ws = Integrate energy terms using Stamatellos et al. (2007) method
- `gamma_eos` : Ratio of specific heats for gas
- `temp0` : (Isothermal) temperature (isothermal or barotropic EOS)
- `mu_bar` : Mean gas particle mass (in units of hydrogen mass)

SPH parameters

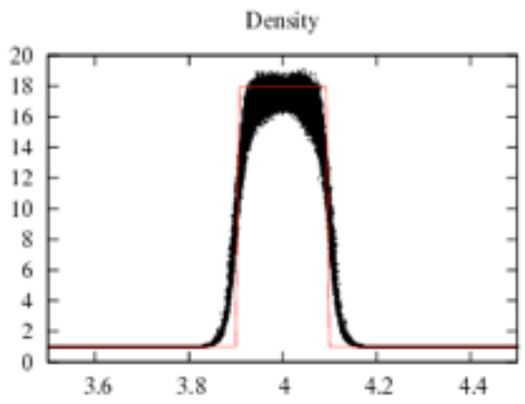
- `sph_integration` : SPH particle integration scheme
  lfkdk = 2nd-order Leapfrog kick-drift-kick
  lfdkd = 2nd-order Leapfrog drift-kick-drift

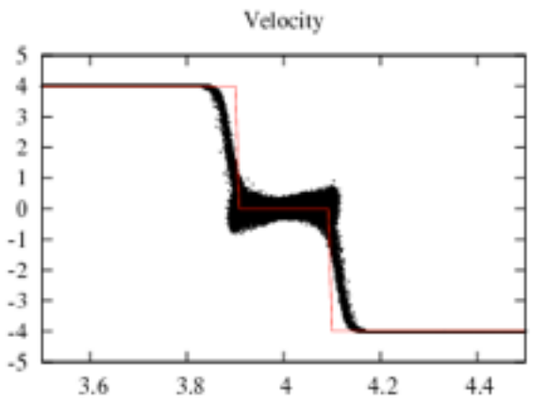
- `kernel` : SPH kernel function
  m4 = M4 Cubic spline kernel
  quintic = Quintic spline kernel
  gaussian = Gaussian kernel (truncated at 3h)



- `avisc` : Artificial viscosity options
  none = No artificial viscosity
  mon97 = Monaghan (1997) viscosity

- `acond` : Artificial conductivity options
  none = No artificial conductivity
  price2008 = Price (2008) conductivity
  wadsley2008 = Wadsley et al. (2008) conductivity





- `time_dependent_avisc` : Morris & Monaghan time-dependent viscosity (1 or 0)

- `alpha_visc` : (Maximum) value of alpha viscosity parameter

- `alpha_visc_min` : Minimum value of alpha for time-dependent viscosity

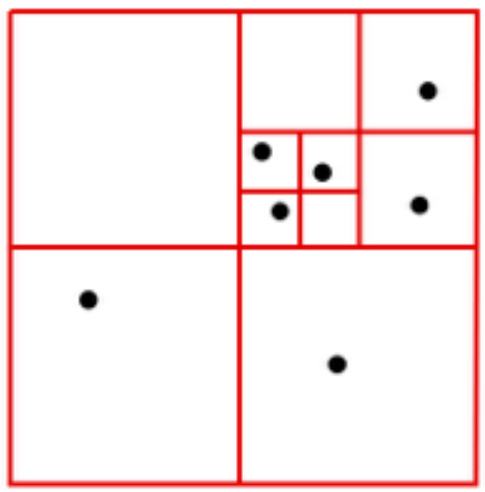
- `beta_visc` : Value of beta viscosity as a multiple of alpha

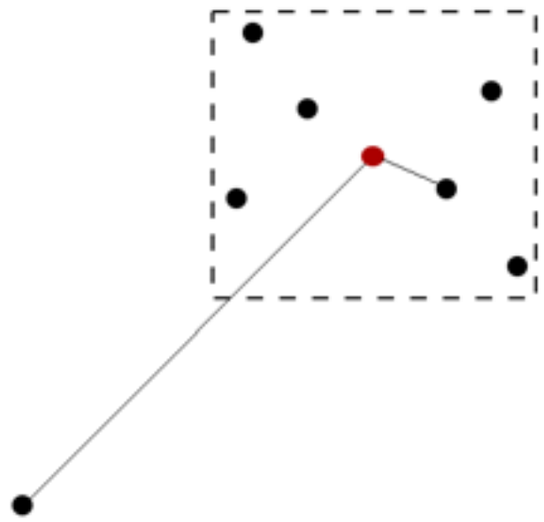
Tree parameters

- `neib_search` : Neighbour searching algorithm

  bruteforce = Brute-force (i.e. summation over all particles)
  kdtree = Balanced kd-binary tree
  octtree = Barnes-Hut octal tree

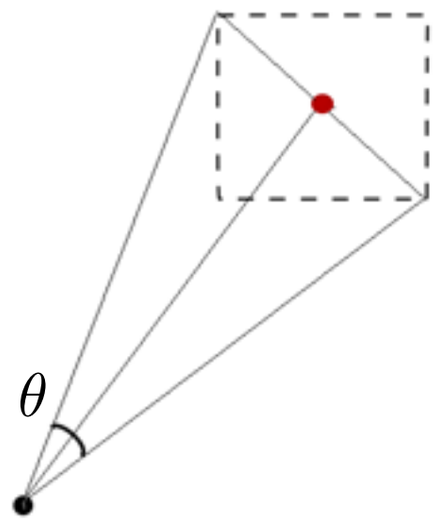
- `Nleafmax` : Maximum no. of particles allowed in tree leaf cell
- `ntreebuildstep` : Integer steps inbetween tree re-builds
- `ntreestock` : Integer steps inbetween tree re-stocks





Gravity parameters

- `self_gravity` : Compute gravitational forces? (1 or 0)
- `gravity_mac` : Gravity-tree cell-opening criteria (N.B. always defulats to geometric for now)
  - geometric = Standard Barnes-Hut geomtric opening angle criterion
  - eigenmac = Compute eigenvalues of quadrupole moment tensor for MAC (Hubber et al. 2011)
- `multipole` : Multipole expansion for tree-gravity
  - monopole = Monopole-only terms for cell gravity
  - quadrupole = Include quadrupole moment terms for cell gravity
  - fast_monopole = Compute monpoles more efficiently using Taylor expansion about cell COM
- `thetamaxsqd` : Maximum tree gravitational walk opening angle (squared)
- `macerror` : MAC error tolerance for individual cells
- `external_potential` : External gravitational potential
  - none = No external potential
  - vertical = Constant gravitational field
  - plummer = Plummer background potential



Sink particle parameters

- `sink_particles` : Do stars/sinks accrete? (0 or 1)
- `create_sinks` : Create new sink particles? (0 or 1)
- `smooth_accretion` : Use smooth accretion? (0 or 1)
- `rho_sink` : Sink particle creation density (in cgs units)
- `alpha_ss` : Sunyaev-Shakura alpha for smooth disc accretion
- `sink_radius` : Sink particle radius (in units of smoothing length)
- `sink_radius_mode` : How to calculate new sink radius
  hmult = sink radius a multiple of SPH particle smoothing length
  fixed = sink radius is fixed for all new sinks

Initial conditions generators in GANDALF

- GANDALF contains a variety of in-built initial conditions generators for :
  - 1D Hydrodynamical tests (e.g. shock-tubes, blast waves)
  - Multi-dimensional hydrodynamical tests (e.g. Sedov-Taylor explosion, Kelvin-Helmholtz instability)
  - Simple gravitational tests (e.g. free-fall collapse)
  - Simple N-body tests (e.g. binary stars, triple stars, Plummer sphere)
  - Simplified astrophysical test cases (e.g. Boss-Bodenheimer test)
  - Complete astrophysical initial conditions (e.g. turbulent prestellar core)
- Feel free to try other test problems (although only some of them you'll be able to plot with a simple plotting program like gnuplot)
- If you have splash successfully installed, then try changing the output format to 'sf' and then plotting them in splash

Compiling the code

- We wil compile GANDALF with the simplest possible set of options

```
CPP                 = g++
PYTHON              = python
COMPILER_MODE       = FAST
PRECISION           = DOUBLE
OPENMP              = 0
OUTPUT_LEVEL        = 1
DEBUG_LEVEL         = 0


# FFTW libary flags and paths.
#-----------------------------------------------------
FFTW                = 0
FFTW_INCLUDE        =
FFTW_LIBRARY        =


# GNU Scientific library flags and paths.
#-----------------------------------------------------
GSL                 = 0
GSL_INCLUDE         =
GSL_LIBRARY         =
```

- To compile the full C++ code AND the python library : `make -j`
- To just compile the C++ executable : `make -j executable`

Running simulations on the command-line

- Once compiled, the gandalf executable will be placed in the ‘bin’ sub-directory located in the main gandalf directory :
- You can either :
  - Run it with the absolute path (e.g. bin/gandalf), or
  - Set your PATH directory to include the gandalf bin subdirectory
- To run a simulation using the parameters file ‘params.dat’, type :

```
bin/gandalf params.dat
```

Practical 1 : Run shocktube simulation in GANDALF

- Let's run some simple test problems with GANDALF
- From the 'tests' sub-directory, open the **adshock.dat** parameters file
- Run the simulation with

```
bin/gandalf tests/adsod.dat
```

- The simulation should produce a series of output dumps of the form ADSHOCK1.su.00001, ADSHOCK.su.00002, etc…OR ADSHOCK1.column.00001, ADSHOCK1.column.00002 (if you selected column format)
- Plot the results with a simple plot program (e.g. gnuplot)

Practical 2 : Modify parameters in adsod.dat

- Try experimenting with the parameters in the file, e.g.
  - **double the output frequency** of snapshots
  - **double the number of particles** in the simulation
  - What happens if you **reduce or even switch off artificial viscosity**?
  - **change the SPH kernel**

Restarting simulations

- To restart a simulation using the last snapshot file generated, just run gandalf as usual but **with the '-r' option added**, i.e.

```
bin/gandalf -r params.dat
```

- The code produces a file called 'runid.restart' which contains the filename (and format) of the last snapshot produced by the code
- This could be used when :
- the simulation has crashed (or the computer has crashed)
- the simulation endtime has been reached and you wish to extend the simulation

Practical 3 : Restarting simulations

- Run the simulation and kill it before it reaches the end (N.B. you might need to increase the number of particles so it doesn't run too fast).
- Restart the simulation using the '-r' flag to verify it will successfully continue until the end
- Try changing the 'tend' parameter and restarting

Practical 4 : Create a new simulation from a parameters file

- **Create a new parameters file** (or copy an old one) to generate the following set of initial conditions for a shock problem :
  - Isothermal equation of state, temp = 0.5
  - LHS, rho = 1.0, vx = 0.0
  - RHS, rho = 0.5, vx = -0.2
  - tend = 0.2
- Choose appropriate values for other important parameters
- Plot the results at the end

Compiling and running GANDALF for debugging

- GANDALF has a number of different options for helping with debugging

```
CPP                     = g++
PYTHON                  = python
COMPILER_MODE           = FAST
PRECISION               = DOUBLE
OPENMP                  = 0
OUTPUT_LEVEL            = 1
DEBUG_LEVEL             = 0
```

- COMPILER_MODE = DEBUG
  - Disables some optimisations and enables the '-g' flag (needed for debuggers)
- OUTPUT_LEVEL = 2
  - Prints to screen more fine-grained information about where the code currently is running at
- DEBUG_LEVEL = 1
  - Enables asserts in the code to help spot clear and quantifyable errors
- DEBUG_LEVEL = 2
  - Enables more detailed (but very expensive) checking of individual algorithms

Running GANDALF with gdb (and other debuggers)

- When COMPILER_MODE = DEBUG, then you can run the code through the debugger (e.g. gdb, lldb)
- To start the debugger (assuming gdb), type :

```
gdb bin/gandalf
```

- To run a simulation with a given parameters file, type :

```
run params.dat
```

- If the code crashes before the end, you can try various commands:
  - to look at the subroutine call try to find out where the code crashed
  - to print the values of the variables

Quick reference list

- The debugger will stop automatically if you have a segmentation fault
- Once it's stopped you can **print** the value of local variables
- **bt** prints the full stack (i.e. tells you in which function you are, and which function called it, all the way up to main)
- **list** prints a few lines of code around the point where you stopped
- You can even execute your code one line at time using **step** (enters inside function calls) and **next** (stops once the function has finished). Useful when you want to see what the code is actually doing
- But true power comes with **breakpoints** -> tell gdb to stop at a particular line
- Use the following:

```
break file.cpp:15
```

- and with **watchpoints** -> tell gdb to stop when a variable changes

```
watch variable
watch *0x12345678
```

Practical 5 : Using the debugger 1

- Type in the following code in the file ‘debugtest.cpp’

```
#include <stdio.h>

void main()
{
  char *temp = "Paras";
  int i;
  i = 0;

  temp[3] = 'F';

  for (i=0; i<5; i++) printf(“%c\n", temp[i]);

  return 0;
}
```

- Compile with

```
gcc –o –g debugtest debugtest.cpp
```

- Now run with and without the debugger. See why a debugger is useful now?

Practical 5 : Using the debugger 2

```
#include <iostream>

int ComputeFactorial(int number) {
  int fact = 0;

  for (int j = 1; j < number; j++) {
    fact = fact * j;
  }

  return fact;
}

int main() {
  int input;
  std::cout<< "Enter a number to compute its factorial" << std::endl;
  std::cin >> input;

  int fac = ComputeFactorial(input);
  std::cout << "The result is " << fac << std::endl;
}
```

- Locate and fix all problems with the debugger (even if your eagle-eye spots the problem by looking at the code) by stepping in the code line by line
- Remember to restart the debugger each time you re-compile