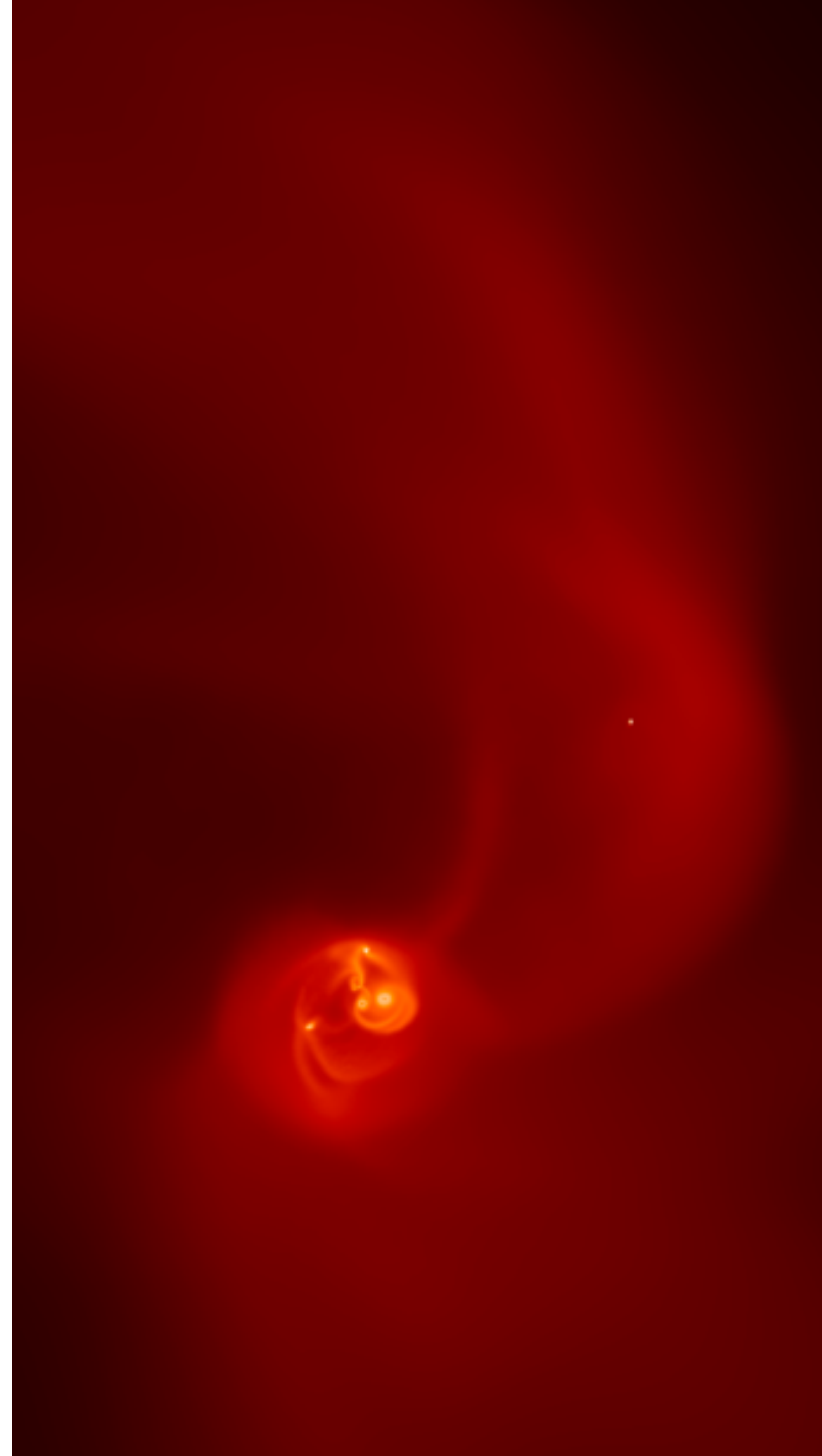
Excellence Cluster Universe

How to do units and scaling right (Or at least how to NOT do them wrong)

David Hubber

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

27th October 2015



So, what's the wrong and right way to do units and why?

- Use physical units of the completely wrong scale (e.g. cm, s)

Very wrong!

- Use physical units of a more appropriate scale (e.g. pc, Myr)

Better but still wrong

- Use arbitrary dimensionless units (i.e. rely on the user to calculate any scale factors)

Correct, but more difficult than it needs to be

- Use dimensionless units where the scale factors are calculated by the code automatically

Yes!

What are dimensionless units exactly?

- **Dimensionless units** are a system of units where a physical quantity can be converted to an equivalent dimensionless quantity by way of a **scaling factor**, e.g.
- For example, take the quantity a which has a scale factor of A_0, then we can convert this to an equivalent dimensionless quantity, a′ :

$$a' = \frac{a}{A_0} \quad \Longleftrightarrow \quad a = A_0\, a'$$

- Dimensionless units themselves **can always be converted back to the real, physical units** at any time
- They are different to **dimensionless constants** (such as e, π, etc..) which are truly dimensionless and are indepedent of any external system of units.

Why use dimensionless units? Floating point precision

- One of the biggest arguments for using dimensionless units is the effect of **finite floating point precision**
- The single precision floating point range is $10^{-38} - 10^{+38}$
- Many astronomers love to **continue to use cgs units** (even in simulations) even though **they're completely inappropriate for almost any astrophysical context**
- Using cgs (or SI or similar non-astronomical units) can cause floating point precision in various situations, e.g.
- Computing volumes, e.g. a parsec size box volume,

$$1\,\mathrm{pc}^3 = 2.93 \times 10^{49}\,\mathrm{m}^3 = 2.93 \times 10^{55}\,\mathrm{cm}^{-3}$$

- Computing quadrupole (or higher-order) correction terms in the gravity tree

$$\frac{1}{r^5} = \frac{1}{pc^5} = 3.57 \times 10^{-83}\,\mathrm{m}^{-5} = 3.57 \times 10^{-93}\,\mathrm{cm}^{-5}$$

Relations between dimensionless quantities

- Although we have freedom to choose a base set of dimensionless units (e.g. mass, length and time), derived quantities that use various combinations of these units must be consistent

$$r' = \frac{r}{R_0} \qquad m' = \frac{m}{M_0} \qquad t' = \frac{t}{T_0}$$

- Velocity units, for example, would be a combination of length and time units, i.e.

$$v' = \frac{v}{V_0} = v\,\frac{T_0}{R_0} \qquad \text{where} \quad V_0 = \frac{R_0}{T_0}$$

- and acceleration units :

$$a' = \frac{a}{A_0} = a\,\frac{T_0^2}{R_0} \qquad \text{where} \quad A_0 = \frac{R_0}{T_0^2}$$

Using dimensionless units to 'eliminate' physical constants

- Many interesting physical problems involves some physical constant, such as G, c, μ_0, etc..
- We can chose a set of dimensionless units such that **the physical constant in the new set of units is unity**
- Various advantages to this
  - If chosen correctly, all physical quantities will be close to unity (better for summing floating point numbers, easier to spot 'incorrect' numbers
  - Completely factors out needing to multiply by the constant (can save a little CPU time at least)

Example : Setting G = 1

- One common example of setting physical constants to unity is in gravitational problems. N-body codes would often employ a system of units that sets G = 1
- Substituting for our dimensionless units and rearranging

$$a = \frac{G\,m}{r^2} \qquad \Rightarrow \qquad \frac{R_0}{T_0^2}\,a' = \frac{G\,M_0\,m'}{R_0^2\,r'^2} \qquad \Rightarrow \qquad a' = \underbrace{\left\{\frac{G\,M_0\,T_0^2}{R_0^3}\right\}}_{G'} \frac{m'}{r'^2}$$

- The final equation in dimensionless form is similar with the constants grouped together, which we've called G′. If we wish to effectively set G′ = 1, then this imposes a constraint on one of our quantities. Traditionally, this has been the time variable :

$$\frac{G\,M_0\,T_0^2}{R_0^3} = 1 \qquad \Rightarrow \qquad T_0 = \left(\frac{R_0^3}{G\,M_0}\right)^{1/2}$$

- Typical units (in N-body and star formation problems) would select R_0 = 1pc and M_0 = 1 solar mass. What does the time unit, T_0 , come out as?

Example : Setting G = 1

$$T_0 = 4.7 \times 10^{14}\,\mathrm{s} = 14.91\,\mathrm{Myr}$$

- This has a knock-on effect on any other unit that has time as a dimension
- e.g. Velocity unit

$$V_0 = \frac{R_0}{T_0} = 0.065\,\mathrm{km\,s^{-1}} = 0.067\,\mathrm{pc\,Myr^{-1}}$$

Another example : Setting $k_b/m_h = 1$

- In hydrodynamics, if we wish to convert from internal energy to temperature, we must use the Boltzmann constant, k_b, and the mass of a hydrogen atom, m_h.
- Similar to setting G = 1, we can set the collection of constants in the sound speed equation to unity to set an appropriate unit for temperature, i.e.

$$c^2 = \gamma \frac{k_b\, T}{\bar{m}} \quad \Rightarrow \quad V_0^2\, {c'}^2 = \gamma\, \frac{k_b\, \theta_0\, T'}{m_h\, \bar{\mu}} \quad \Rightarrow \quad {c'}^2 = \gamma\, \underbrace{\left\{ \frac{k_b\, \theta_0}{m_h\, V_0^2} \right\}}_{1} \frac{T'}{\bar{\mu}}$$

$$\frac{k_b\, \theta_0}{m_h\, V_0^2} = 1 \quad \Rightarrow \quad \theta_0 = V_0^2\, \frac{m_h}{k_b}$$

- Using the same typical star formation units we employed for the G = 1 example, we fine that

$$\theta_0 = 0.51\,\mathrm{K}$$

Units in GANDALF

- We created a 'SimUnit' class in GANDALF to hold and compute all required scaling factors
- Each required unit class inherits from this base class

```
class SimUnit
{
 public:

  SimUnit();
  virtual ~SimUnit() {};

  virtual DOUBLE SIUnit(string) = 0;
  virtual string LatexLabel(string) = 0;
  DOUBLE OutputScale(string);

  DOUBLE inscale;
  DOUBLE inSI;
  DOUBLE outcgs;
  DOUBLE outscale;
  DOUBLE outSI;
  string inunit;
  string outunit;
};
```

```
class LengthUnit: public SimUnit
{
 public:
  LengthUnit() : SimUnit() {};
  DOUBLE SIUnit(string);
  string LatexLabel(string);

};

class MassUnit: public SimUnit
{
 public:
  MassUnit() : SimUnit() {};
  DOUBLE SIUnit(string);
  string LatexLabel(string);

};

etc..
```

Units in GANDALF

- An all-encompasing class called 'SimUnits' (there's an extra 's') which then holds everything in one place :

```
class SimUnits
{
 public:

  SimUnits();
  ~SimUnits();

  void SetupUnits(Parameters *);
  void OutputScalingFactors(Parameters *);

  int dimensionless;                    ///< Are we using dimensionless units?
  bool ReadInputUnits;                  ///< Are input units read from snapshot?


  // Instances of all unit classes
  //-----------------------------------------------------------------------
  LengthUnit r;                         ///< Length unit
  MassUnit m;                           ///< Mass unit
  TimeUnit t;                           ///< Time unit
  VelocityUnit v;                       ///< Velocity unit
  AccelerationUnit a;                   ///< Acceleration unit
  DensityUnit rho;                      ///< Density unit
  etc..
};
```

Input units vs Output units

- GANDALF is designed to handle simulataneously an input and output set of units
- For example, maybe you are reading in initial conditions in one set of units (e.g. pcs, Myr) but want to output in a different set (e.g. au, yr)
- However, in most cases, you will generate initial conditions with the same set of units OR create initial conditions on the fly
- Your choice of output units will often be set in the parameters file

```
#---------------------------
# Simulation units variables
#---------------------------
Use physical units                             : dimensionless = 0
Length units                                   : routunit = pc
Mass units                                     : moutunit = m_sun
Time units                                     : toutunit = myr
Velocity units                                 : voutunit = km_s
Density units                                  : rhooutunit = g_cm3
Temperature units                              : tempoutunit = K
Specific internal energy units                 : uoutunit = J_kg
Angular velocity unit                          : angveloutunit = rad_s
```

Computing the scaling factors

- In order to simpify the calculation of scaling factors, **we calculate everything in the same set of units internally**. We use **SI units** (although we could have chosen cgs or another set if we wished)

$$R_0 = \underbrace{R_{\mathrm{outscale}}}_{\text{Converts code units to requested output units}} \times \overbrace{R_{\mathrm{outSI}}}^{\text{Converts requested output units to SI units}}$$

- If we have a different set of input and output units, then both of these should be consistent with each other, i.e.

$$R_0 = R_{\mathrm{outscale}} \times R_{\mathrm{outSI}} = R_{\mathrm{inscale}} \times R_{\mathrm{inSI}}$$

Length and mass units

- As discussed earlier, we select the length and mass units directly so these are trivial and are set as 1.0 each. We also need to compute the SI and cgs factors

```
// Length units
//-----------------------------------------------------------
r.inunit   = params->stringparams["rinunit"];
r.outunit  = params->stringparams["routunit"];
r.inSI     = r.SIUnit(params->stringparams["rinunit"]);
r.outSI    = r.SIUnit(params->stringparams["routunit"]);
r.outcgs   = 100.0*r.outSI;
r.outscale = 1.0;
r.inscale  = r.outscale*r.outSI/r.inSI;
```

$$R_{\text{inscale}} = \frac{R_{\text{outscale}} \times R_{\text{outSI}}}{R_{\text{inSI}}}$$

Mass units

- As discussed earlier, we select the length and mass units directly so these are trivial and are set as 1.0 each

```
// Mass units
//-------------------------------------------------------
m.inunit   = params->stringparams["minunit"];
m.outunit  = params->stringparams["moutunit"];
m.inSI     = m.SIUnit(params->stringparams["minunit"]);
m.outSI    = m.SIUnit(params->stringparams["moutunit"]);
m.outcgs   = 1000.0*m.outSI;
m.outscale = 1.0;
m.inscale  = m.outscale*m.outSI/m.inSI;
```

Time units

- The time units are computed to ensure G = 1 as described earlier

```
// Time units
//-----------------------------------------------------------------------
t.inunit   = params->stringparams["tinunit"];
t.outunit  = params->stringparams["toutunit"];
t.inSI     = t.SIUnit(params->stringparams["tinunit"]);
t.outSI    = t.SIUnit(params->stringparams["toutunit"]);
t.inscale  = pow(r.inscale*r.inSI,1.5)/sqrt(m.inscale*m.inSI*G_const);
t.inscale  /= t.inSI;
t.outscale = pow(r.outscale*r.outSI,1.5)/sqrt(m.outscale*m.outSI*G_const);
t.outscale /= t.outSI;
t.outcgs   = t.outSI;
```

$$T_0 = \left(\frac{R_0^3}{G\,M_0}\right)^{1/2} \quad \Rightarrow \quad T_{\rm outscale} \times T_{\rm outSI} = \left(\frac{R^3_{\rm outscale} \times R^3_{\rm outSI}}{G_{\rm SI}\,M_{\rm outscale} \times M_{\rm outSI}}\right)^{1/2}$$

$$T_{\rm outscale} = \left(\frac{R^3_{\rm outscale} \times R^3_{\rm outSI}}{G_{\rm SI}\,M_{\rm outscale} \times M_{\rm outSI}}\right)^{1/2} \frac{1}{T_{\rm outSI}}$$

Converting initial conditions and input parameters to code units

- Converting initial conditions and/or parameters is simply a case of using the inscale/outscale variables.
- e.g. if you input velocities in km/s, then

```
Velocity units                                  : voutunit = km_s
```

- Then all velocity quantities are scaled using

$$v' = \frac{v(\mathrm{km\_s})}{V_{\mathrm{outscale}}}$$

- If you have a parameter that is in either SI or cgs, then also divide by the SI/cgs factor

$$rho' = \frac{rho(\mathrm{g\,cm^{-3}})}{rho_{\mathrm{outscale}} \times rho_{\mathrm{outcgs}}}$$

Code units vs output units

- GANDALF only uses these dimensionless units internally for computing quantities internally
- When generating output, such as snapshot files, all quantities are converted back to physical units (specified by the outscale selections)

$$v(\mathrm{km\_s}) = V_{\mathrm{outscale}} \times v'$$

- For example, when outputting to file :

```
outfile << part.r[0]*simunits.r.outscale << "   "
        << part.v[0]*simunits.v.outscale << "   "
        << part.m*simunits.m.outscale << "   "
        << part.h*simunits.r.outscale << "   "
        << part.rho*simunits.rho.outscale << "   "
        << part.u*simunits.u.outscale << "   "
        << endl;
```

So, some basic rules about chosing units for your code

- Obviously, choose some set of units that is representive of your problem. Having a nice dimensionless framework means **nothing if you still chose strange units**
- **And that's about it!**