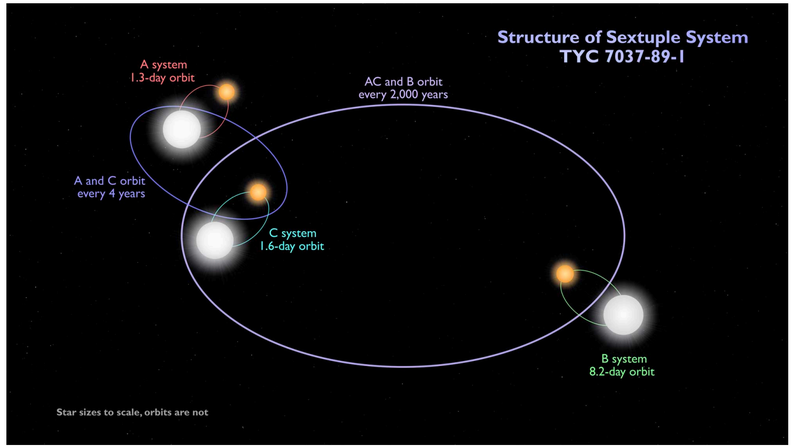
ASTR 660 / Class 4

N-Body optimisations

Challenges in the N-Body Problem



We have considered how to solve the ODE system describing motion of particles under Newtonian gravity.

Three optimisations commonly used by standard N-body codes are:

- Adaptive steps sizes;
- Gravitational softening;
- Separation of long and short-range forces.

The following is a short introduction to the keywords and general ideas. We will not go into the details.

Gadget

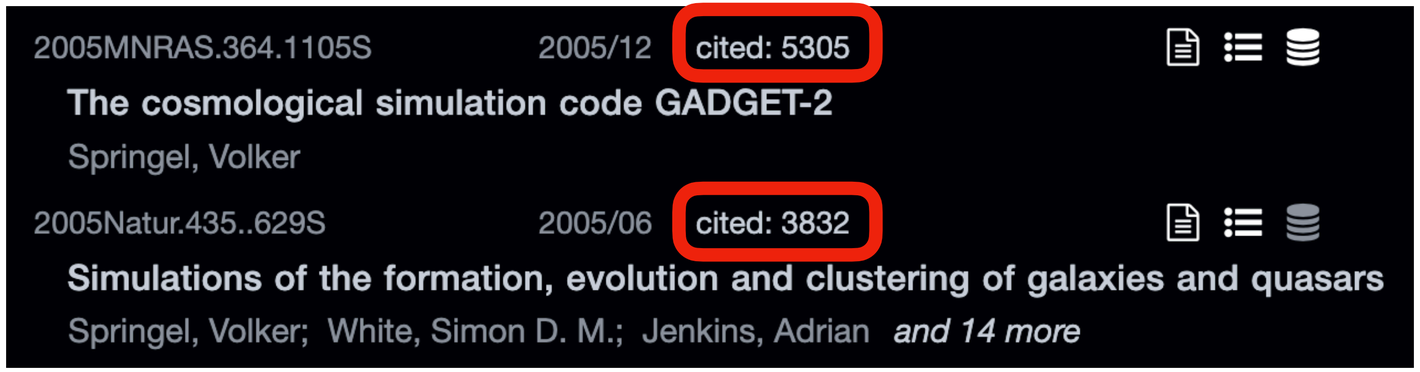
A good example of a modern collisionless N-body code is Volker Springel's Gadget-2:

Mon. Not. R. Astron. Soc. **364,** 1105–1134 (2005) doi:10.1111/j.1365-2966.2005.09655.x

The cosmological simulation code GADGET-2

Volker Springel★
Max-Planck-Institut für Astrophysik, Karl-Schwarzschild-Straße 1, 85740 Garching bei München, Germany

Accepted 2005 September 26. Received 2005 September 25; in original form 2005 May 11



Adaptive step sizes

The use of **adaptive steps** is a common feature of ODE solvers.

The accuracy of an individual step can be estimated for a particular solver. This can be analytic, but also empirical — compute the solution over a fixed time with smaller and smaller steps, and check the rate of **convergence**.

If the user can specify the level of accuracy (**tolerance**) they are satisfied with, the size of the step can be adjusted accordingly.

Adaptive step sizes

When the function being evaluated is changing quickly (e.g. when the acceleration on an N-body particle is large), smaller steps can be taken to maintain the required accuracy.

This turns out to be a very important route to speeding up most N-body simulations.

How can we decide the size of the step in this case? We certainly don't want to run the simulation many times!

The simulation has to adapt the step size on the fly, based only on the information available when the forces are being computed.

Adaptive step sizes

In GADGET-2, a time-step criterion for collisionless particles of the form

$$\Delta t_{\rm grav} = \min\left[\Delta t_{\rm max}, \left(\frac{2\,\eta\,\epsilon}{|\boldsymbol{a}|}\right)^{1/2}\right] \tag{34}$$

is adopted, where η is an accuracy parameter, ϵ gives the gravitational softening and $\boldsymbol{a}$ is the acceleration of the particle. The maximum allowed time-step is given by $\Delta t_{\rm max}$, and is usually set to a small fraction of the dynamical time of the system under study. In

$\sqrt{\epsilon/|\mathbf{a}|}$ has the dimension of time.

(In many cases we don't know the appropriate dynamical time…)

Adaptive step sizes

In the N-body case, there is an additional concern: is the adaptive timestep **global** (same for all particles, limited by the particle with the highest acceleration) or **individual**?

If different particles are being evolved with different timesteps, how can they be **synchronized** for the force calculation?

This leads to the concept of **block timesteps**. Block timestepping is a good match to the use of the leapfrog integrator (see Dehen & Read 2011).

In Gadget-2, the time steps are discretised in power-of-2 bins ($\Delta t_{\rm max}$, $\Delta t_{\rm max}/2$, $\Delta t_{\rm max}/4$ etc.). A separate force tree (described later) is built for each bin. Particles can move between timestep bins at synchronization points.

Scipy ODE solvers

Solving initial value problems for ODE systems

The solvers are implemented as individual classes, which can be used directly (low-level usage) or through a convenience function.

`solve_ivp`(fun, t_span, y0[, method, t_eval, ...])	Solve an initial value problem for a system of ODEs.
`RK23`(fun, t0, y0, t_bound[, max_step, rtol, ...])	Explicit Runge-Kutta method of order 3(2).
`RK45`(fun, t0, y0, t_bound[, max_step, rtol, ...])	Explicit Runge-Kutta method of order 5(4).
`DOP853`(fun, t0, y0, t_bound[, max_step, ...])	Explicit Runge-Kutta method of order 8.
`Radau`(fun, t0, y0, t_bound[, max_step, ...])	Implicit Runge-Kutta method of Radau IIA family of order 5.
`BDF`(fun, t0, y0, t_bound[, max_step, rtol, ...])	Implicit method based on backward-differentiation formulas.
`LSODA`(fun, t0, y0, t_bound[, first_step, ...])	Adams/BDF method with automatic stiffness detection and switching.
`OdeSolver`(fun, t0, y0, t_bound, vectorized)	Base class for ODE solvers.
`DenseOutput`(t_old, t)	Base class for local interpolant over step made by an ODE solver.
`OdeSolution`(ts, interpolants)	Continuous ODE solution.

Old API

These are the routines developed earlier for SciPy. They wrap older solvers implemented in Fortran (mostly ODEPACK). While the interface to them is not particularly convenient and certain features are missing compared to the new API, the solvers themselves are of good quality and work fast as compiled Fortran code. In some cases, it might be worth using this old API.

`odeint`(func, y0, t[, args, Dfun, col_deriv, ...])	Integrate a system of ordinary differential equations.
`ode`(f[, jac])	A generic interface class to numeric integrators.
`complex_ode`(f[, jac])	A wrapper of ode for complex systems.

There are two APIs for ODE solvers in Scipy: “new” and “old”.

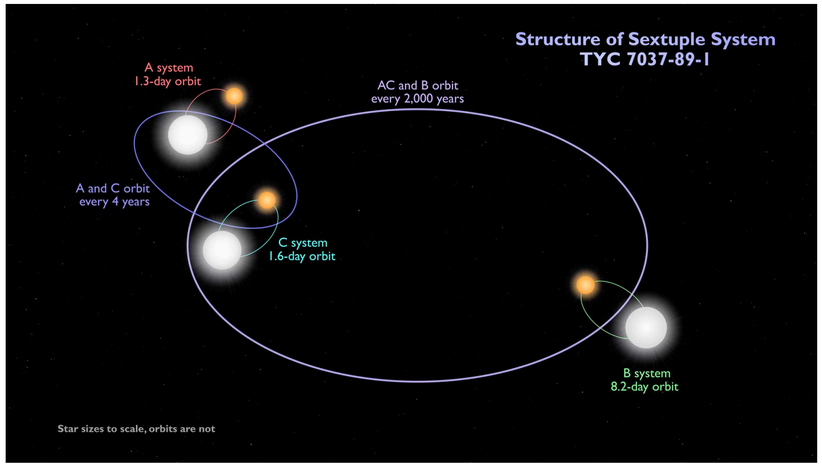
Notice parameters `atol`, `rol`, `max_step` etc.

Also notice the first argument to all these is `fun`.

Binaries

In the collisional case, **binaries** are extremely important. In globular clusters, interactions between single stars and binaries provide a 'heating' mechanism that opposes the contraction of the cluster core (Why does it contract? See Binney & Tremaine 2008 sec. 7.3.2 and/or search for **gravothermal catastrophe**).

Accurate binary orbits can take a lot of effort (very small step sizes). Production collisional N-body codes (*NBody6* etc.) evolve binaries with different techniques **(regularization)**.



Gravitational softening

The Newtonian gravitational potential diverges as $r \to 0$. Tight binaries can form.

In collisionless simulations, binary orbits are unphysical anyway. Why waste time on them?

Instead, the gravitational potential of each particles is **softened** on a scale ϵ. Below this scale, potential tends to a constant, so the force between two particles tends to zero.

The simplest example of a softened potential is the **Plummer** potential:

$$\Psi(r) = -\frac{Gm}{r} \to \tilde{\Psi}(r;\epsilon) = -\frac{Gm}{\sqrt{r^2+\epsilon^2}}$$

Gravitational softening

In practice (e.g. in Gadget) it's better to restrict the effect of the softening to small distances, by representing the **density distribution** associated with the particle using a more complicated "spline kernel" with **compact support** (Monaghan & Lattanzio 1985):

$$W(r,h) = \frac{8}{\pi h^3} \begin{cases} 1 - 6\left(\frac{r}{h}\right)^2 + 6\left(\frac{r}{h}\right)^3, & 0 \le \frac{r}{h} \le \frac{1}{2}, \\ 2\left(1 - \frac{r}{h}\right)^3, & \frac{1}{2} < \frac{r}{h} \le 1, \\ 0, & \frac{r}{h} > 1. \end{cases}$$

Gadget uses $h = 2.8\epsilon$, such that the particle potential at $r = 0$ is $-Gm/\epsilon$ (in this case ϵ is called the **Plummer equivalent softening scale**). The force is exactly Newtonian for separations larger than h.

How should we choose ϵ?

Separation of long and short range forces

The computation required for a direct N-body force calculation scales as N^2; this is much worse than the work to advance the particles, which scales as N.

The force computation can be reduced to scaling as approximately $N \ln N$ (N particles, and for each particle, $\ln N$ evaluations) by using different methods to calculate “short-" and “long-range” forces.

The standard approaches are **Particle-Mesh** methods (after Hockney & Eastwood) and **Tree** methods (also known as Barnes-Hut). Gadget uses both methods together, a “hybrid” scheme.

These methods are worth some attention, because the algorithms they use have other applications computational astrophysics.

Particle Mesh

The idea is to map the particle density onto a regular **mesh** (3D grid), then take the **Fourier transform** (using FFT, the Fast Fourier Transform algorithm).

The potential on the mesh can be computed rapidly (i.e. Poisson's equation can be solved) by manipulating (essentially multiplying) the Fourier-transformed density field, then inverse-Fourier-transforming back to coordinate space. If time permits, we'll look at this in a later class.

Once the potential on the mesh is known, the associated **force field** can be computed (e.g. by the finite difference method) and **interpolated** at the positions of the particles.

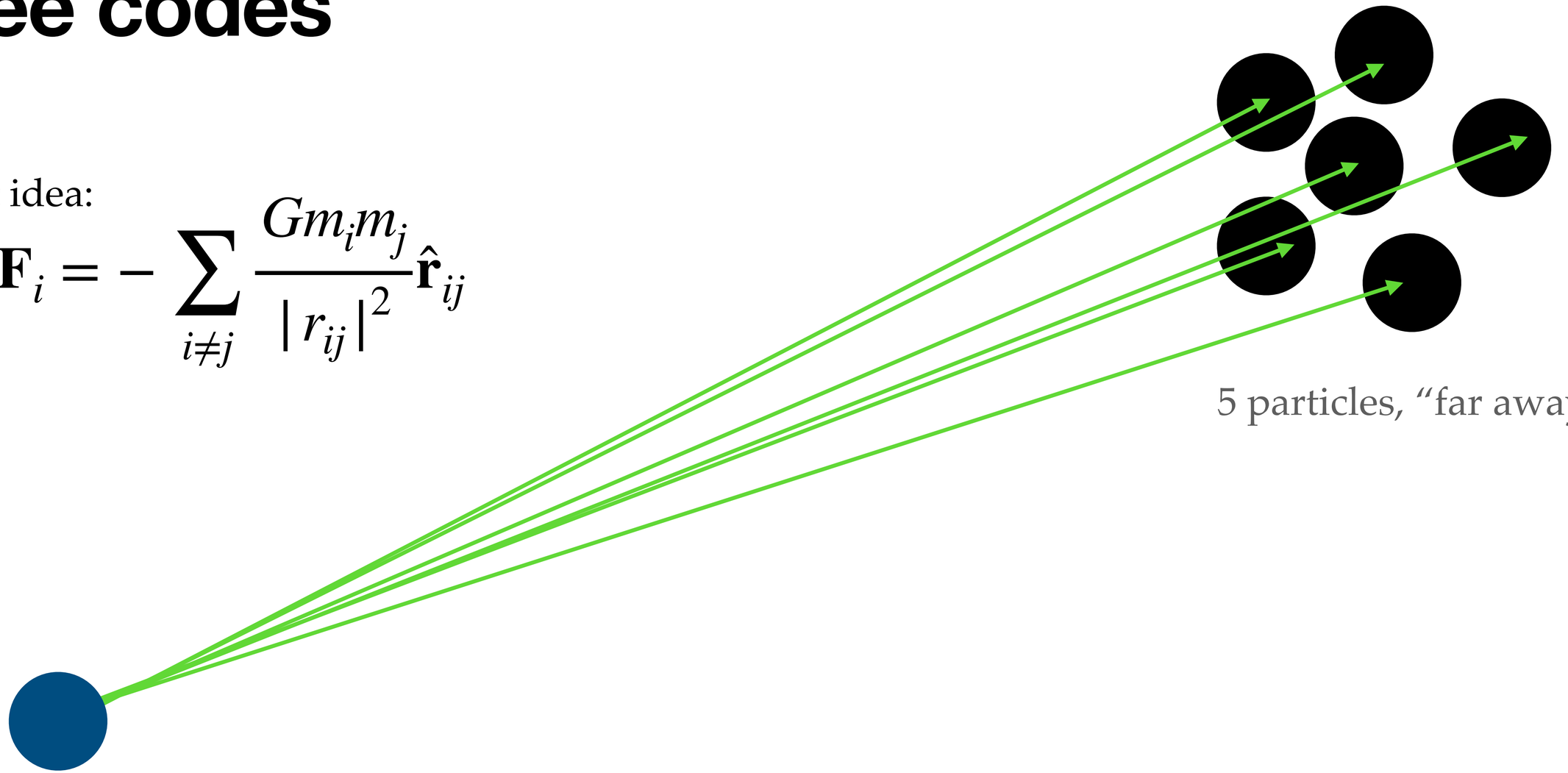
The accuracy is limited by the spacing of the mesh (somewhat analogous to softening). A finer mesh needs more memory and more time to compute.

Particle-particle / Particle Mesh (P3M) methods compute short-range forces with direct summation; care is needed on the scale of the mesh (see e.g. Gadget-2 paper or Dehnen & Read 2011).

Tree codes

Basic idea:

$$\mathbf{F}_i = -\sum_{i \neq j} \frac{G m_i m_j}{|r_{ij}|^2} \hat{\mathbf{r}}_{ij}$$

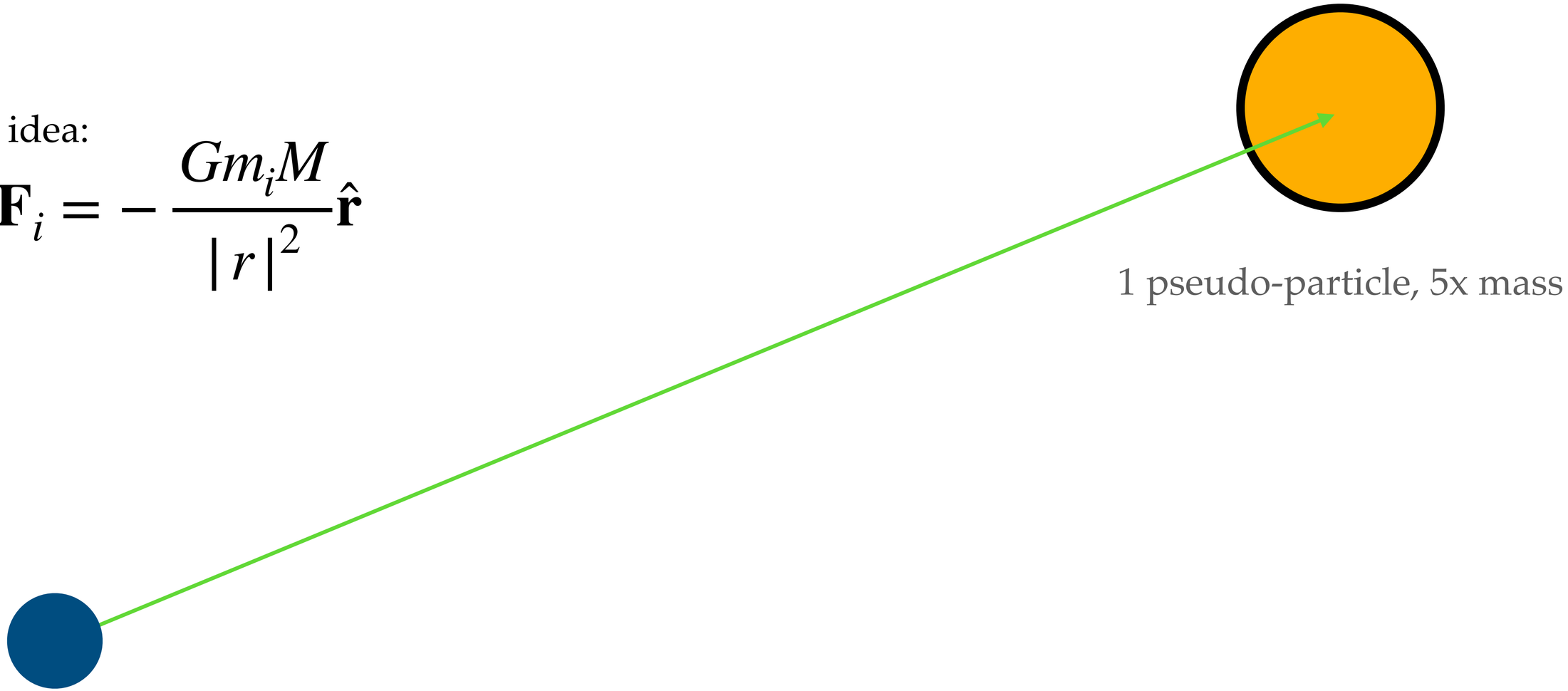


5 particles, “far away”

Tree codes

Basic idea:

$$\mathbf{F}_i = -\frac{Gm_iM}{|r|^2}\hat{\mathbf{r}}$$



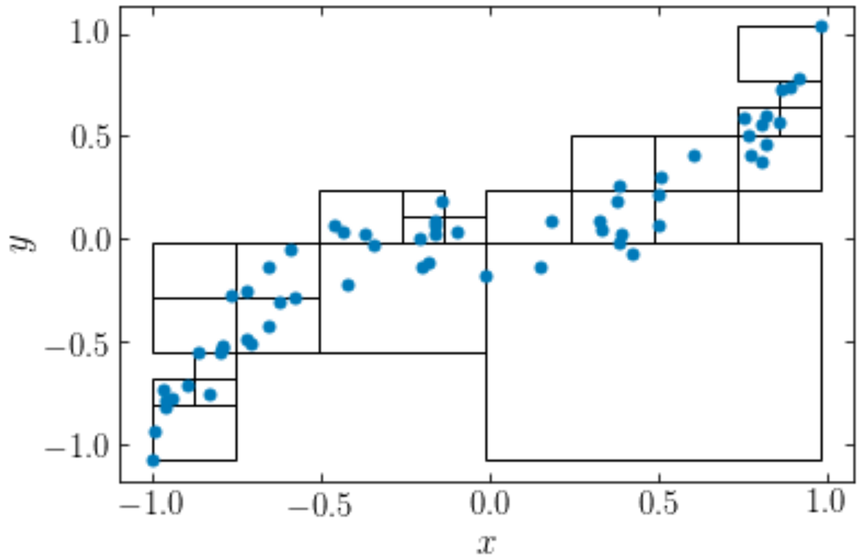
1 pseudo-particle, 5x mass

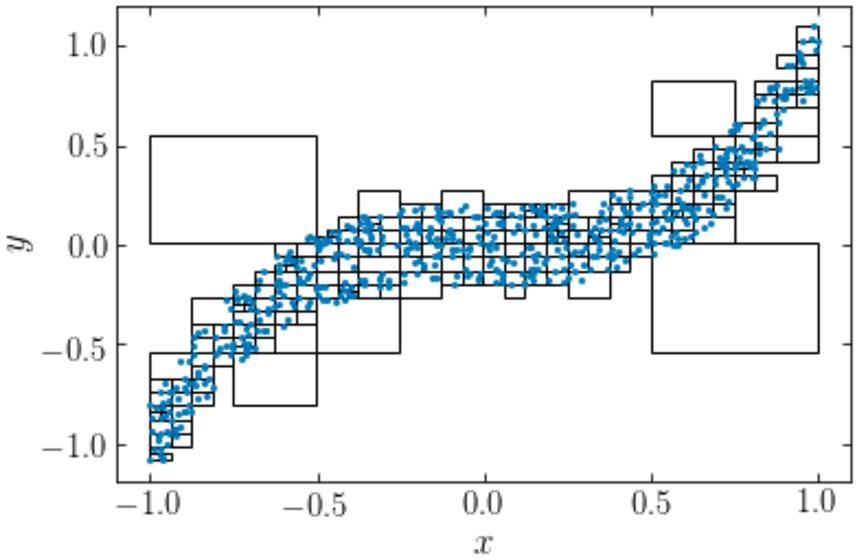
Tree codes

The trick is to construct a **hierarchical partitioning** of space by assigning particles to **unit cells**.

A cell can be substituted for the position (and mass) of all the particles inside it, when computing the force on far-away particles.

Bovy Ch. 19





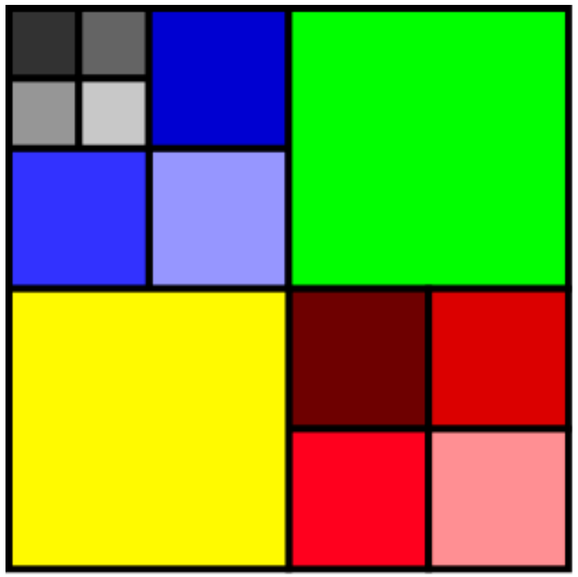
Tree codes

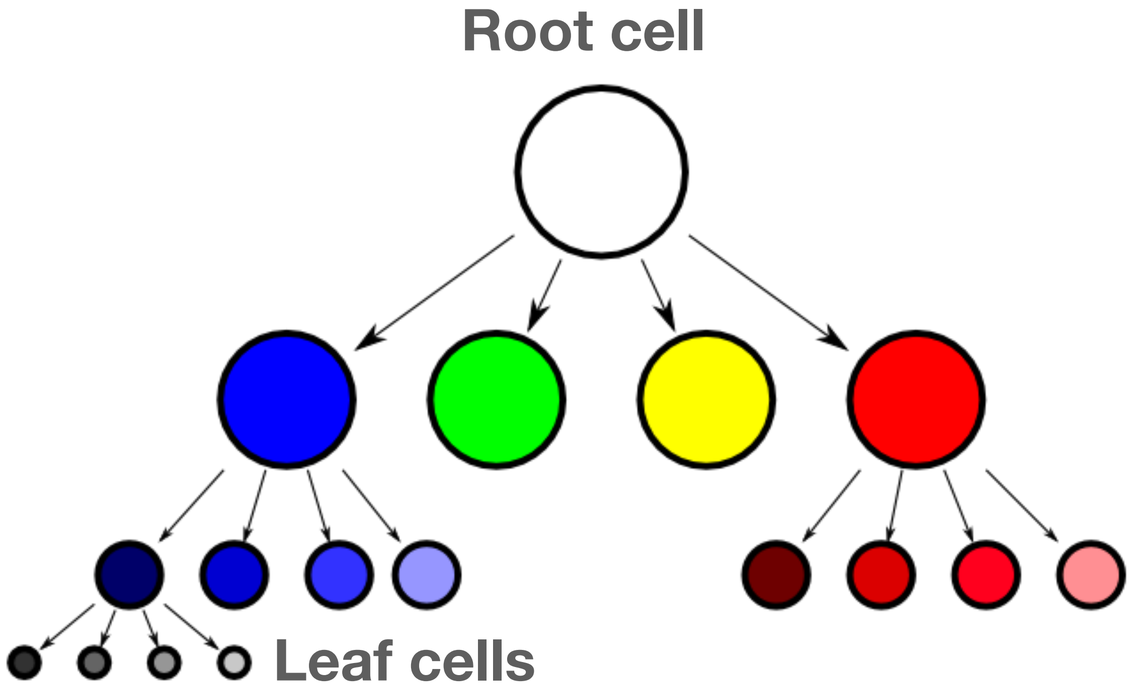
Requirements:

- Assigning particles to cells (building the tree) must be quick (needs to be done again each time the particles move!);
- Finding which cell a particle is in must be quick;
- The resolution of the tree should adapt to the density of the particle distribution;
- The tree must not take too much memory (could be used for more particles!).

An **octree** (in 2-d, **quadtree**) is regular, hierarchical partitioning of the simulation volume into equal-sized cubes (squares).

Octrees





Regular: the volume spanned by each cell is predetermined. Particle positions can be mapped quickly to the corresponding leaf nodes.

Hierarchical: simple relationship between geometry of smaller (child) and larger (parent) cells.

Adapted from https://commons.wikimedia.org/wiki/File:Quadtree,_graphical_and_tree_representation.png

Quad/Oct-Trees

Dehnen & Read (2011)

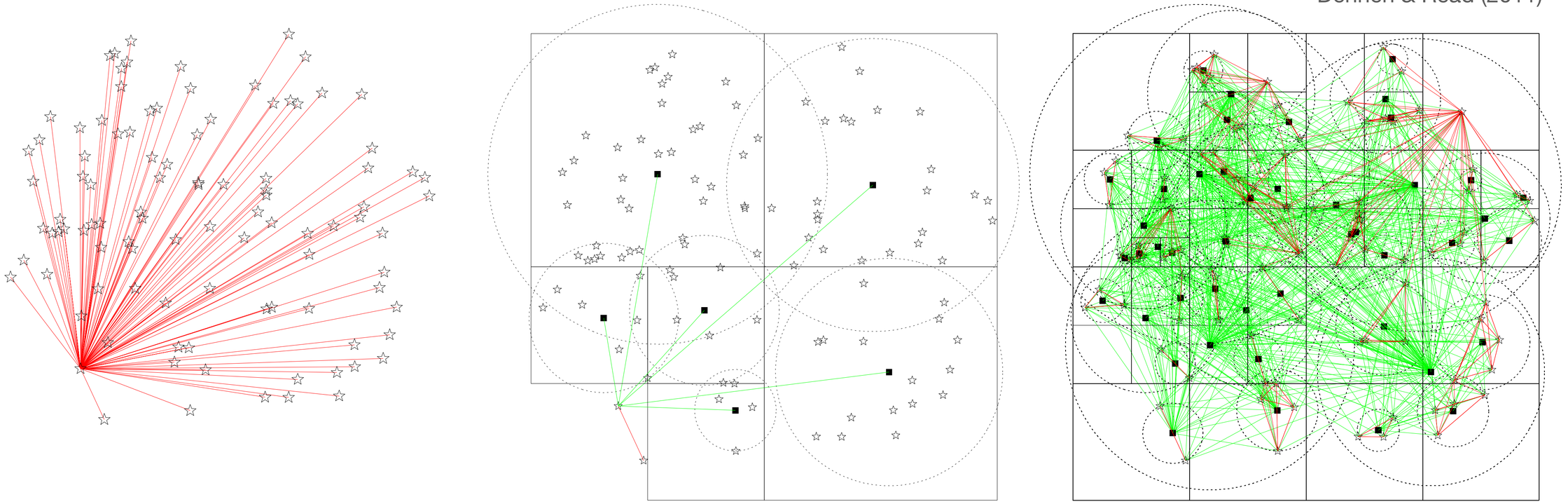
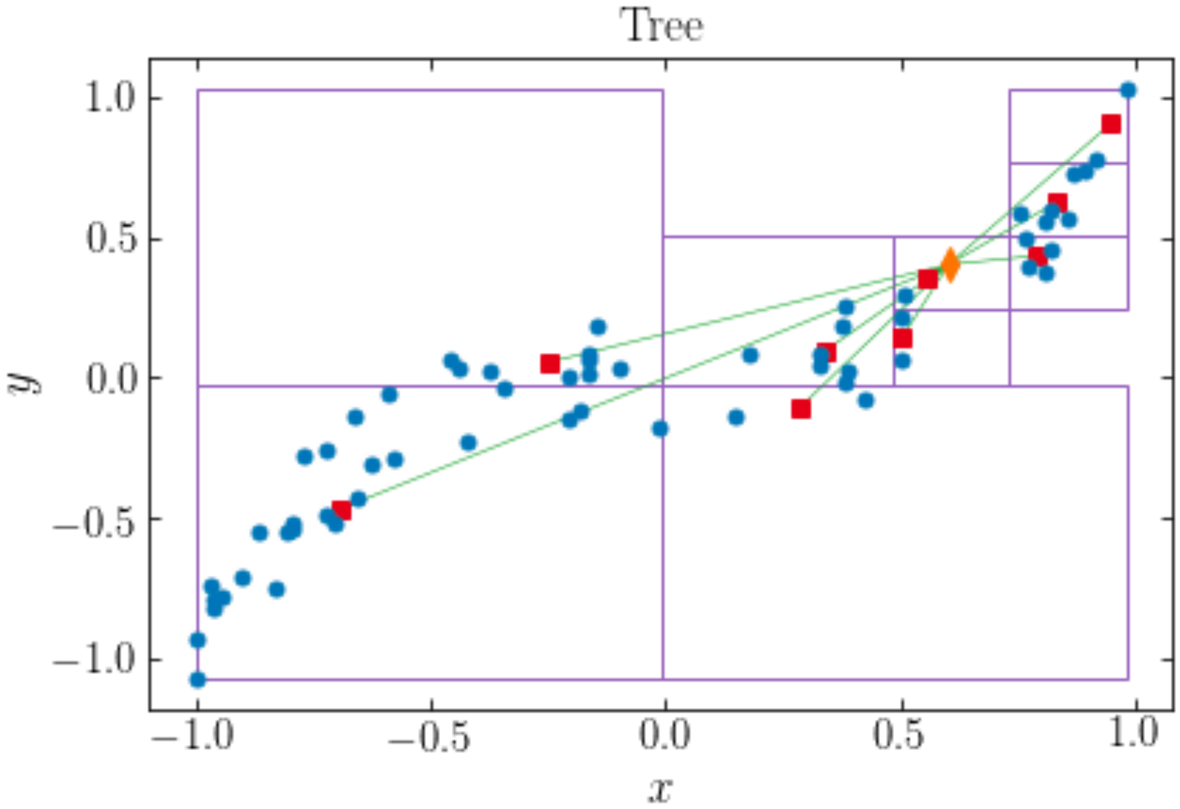


Fig. 5. Left: computation of the force for one of 100 particles (asterisks) in two dimensions (for graphical simplicity) using direct summation: every line corresponds to a single particle-particle force calculation. **Middle:** approximate calculation of the force for the same particle using the tree code. Cells opened are shown as black squares with their centres $\boldsymbol{z}$ indicated by solid squares and their sizes w by dotted circles. Every green line corresponds to a cell-particle interaction. **Right:** approximate calculation of the force for all 100 particles using the tree code, requiring 902 cell-particle and 306 particle-particle interactions ($\theta = 1$ and $n_{\rm max} = 1$), instead of 4950 particle-particle interactions with direct summation.

Opening cells



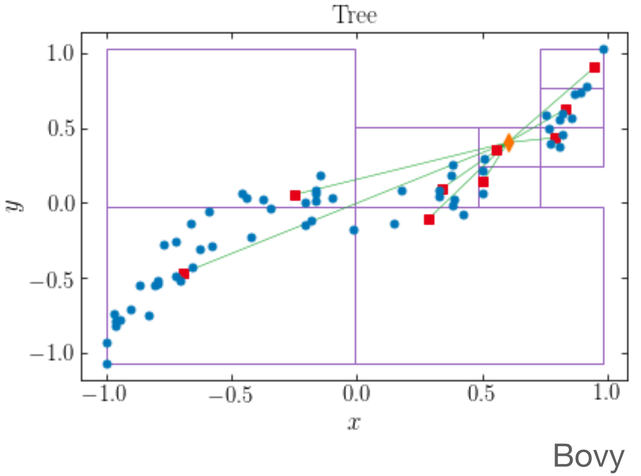
Bovy Ch. 19

Forces from “nearby” particles are calculated by direct summation. Forces from particles in “far away” cells are calculated using the total mass and centre-of-mass position* of their leaf cell.

Very far away cells can themselves be grouped together into their parent cells (i.e. we can use lower and lower levels of the tree). We talk about **opening** cells to different levels, based on their distance and the required level of accuracy.

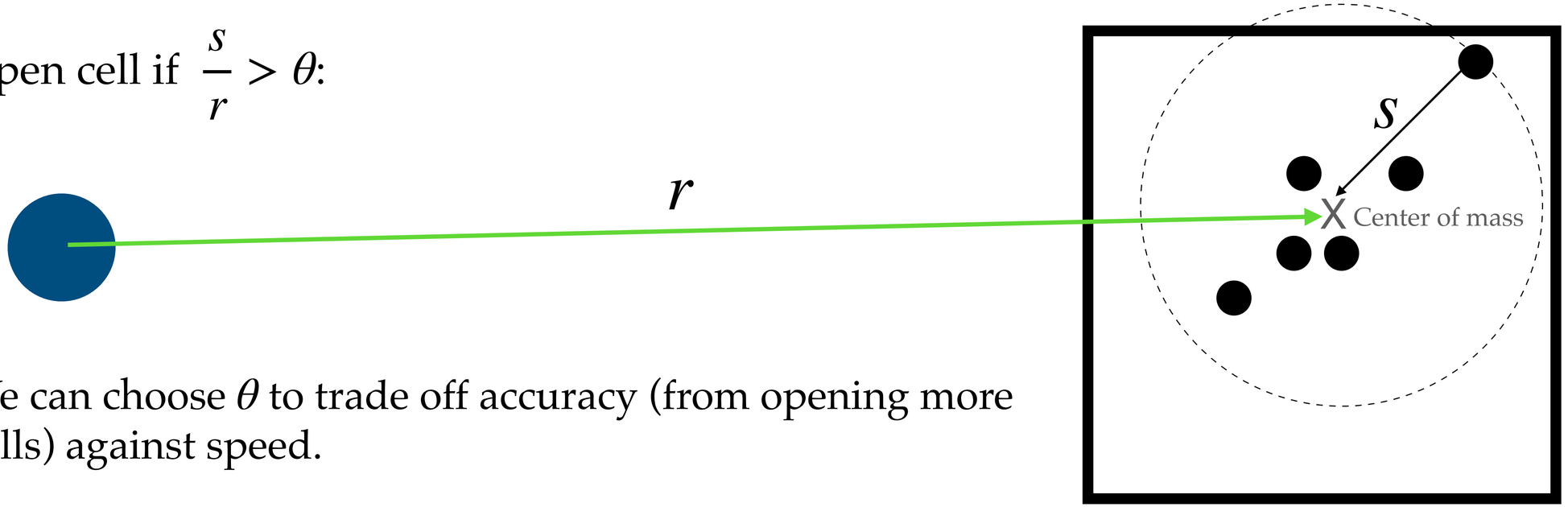
* This is the lowest-order **monopole** approximation; higher-order **multipole expansion** is sometimes used to improve the approximation of the mass distribution in the cell (but not in Gadget!).

Opening angle



An **opening angle** criterion is used to determine when to stop walking down the tree for each cell, given a target particle. This is effectively calculating the apparent size of a cell, as seen by the target particle.

Open cell if $\frac{s}{r} > \theta$:



We can choose θ to trade off accuracy (from opening more cells) against speed.

When a leaf node is opened, we use direct summation.

Tree codes in practice

See Section 3.1 in the Gadget-2 paper for a nice summary of the advantages and issues involved in using trees for large N-body simulations.

For example, notice how the author defines an adaptive opening angle in equation 18, and how he makes use of the tree for the neighbour-search step of the SPH algorithm.

Gadget uses trees for short/intermediate range forces and (optionally) a PM method for long range forces; this is a similar idea to P3M.

A major breakthrough in Gadget-2 was to develop efficient parallel versions of this hybrid method (we may discuss these in a later class).

Tree codes for neighbour finding

Trees trade off a one-time cost in building the tree (and the memory needed to store the tree) for much faster **lookups** afterwards.

Tree algorithms are generically useful for problems that involve computing distances in large multidimensional datasets. A common application is searching for **neighbouring points**.

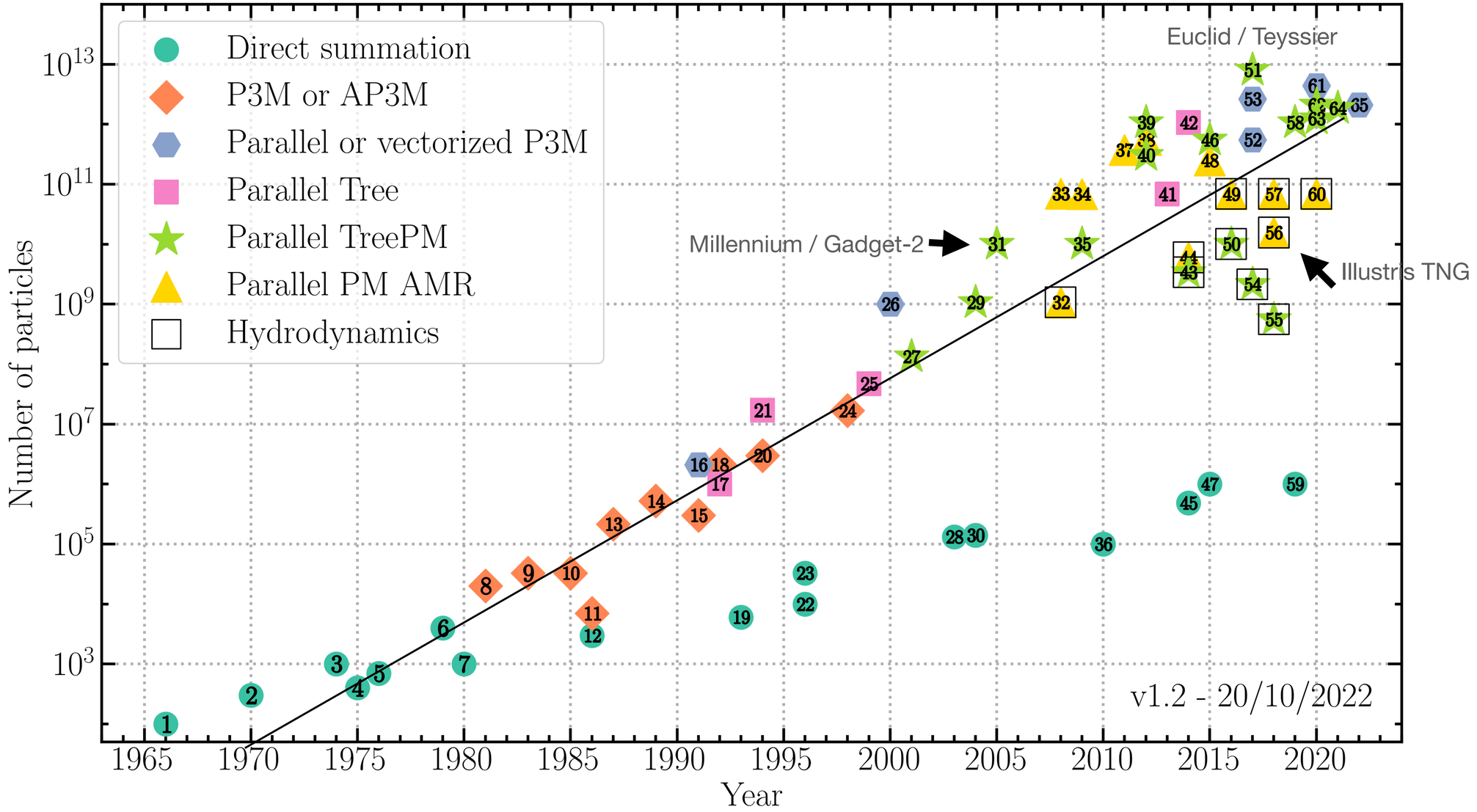
Outside N-body (and perhaps other astrophysical problems) you are more likely to use a KDTree, which is optimal for the common task of finding (distances to) near neighbours but not as useful for N-body force calculations (see NR, and discussion in Gadget-2 paper).

Scipy has a fast and very useful KDTree method (but no octree): scipy.spatial.KDTree

I strongly recommend learning how this works.

N-body progress

Florent Leclercq: https://www.florent-leclercq.eu/blog.php?page=2



Extra material

Code design

Aside: code design

Different ODE solvers have features in common.

The input is some (parameterised) function (or system of functions), $\hat{f}(t; a, b, c)$, with a shared independent variable t (maybe this is 'time', maybe not).

The solver evaluates that function a bunch of times in order to advance $t \to t'$.

We could say different solvers are *subclasses* of a general *class* of methods for solving ODEs.

Common function interfaces

The ODE solvers in Scipy are all written in this way:

```
solve_ivp(fun, t_span, y0[, method, t_eval, ...])
```

They all have the same sequence of arguments. They take the **function** to be integrated their first argument.

In Python, functions are objects and so can be manipulated in the same way as other objects.

Functions are objects

```
def add_together(a,b):
    """
    Adds argument a to argument b.
    """
    return a+b

print(add_together.__class__)
<class 'function'>
```

Passing functions as arguments

```
def call_some_function(f,a,b):
    """
    Calls the function f with arguments a,b
    """
    print(f(a,b))
    return

call_some_function(add_together,2,3)
 > 5
```

The purpose of this is to **improve the readability** of your code, mainly by **avoiding repetition**.

Of course, power comes with responsibility. Mis-using this power can also make your code **less** readable…

Lambdas

Functions can be created on one line with **lambda** statements, as an alternative to **def** blocks. The purpose of this is to **improve the readability** of your code.

```
negative_square = lambda x: -x**2
print(negative_square(2))
```

Partial evaluation

[22]:
```
from functools import partial
```

[26]:
```
partial?
```

```
Init signature: partial(self, /, *args, **kwargs)
Docstring:
partial(func, *args, **keywords) - new function with partial application
of the given arguments and keywords.
File:           /opt/miniconda3/envs/astr660/lib/python3.11/functools.py
Type:           type
Subclasses:
```

[30]:
```
def projectile_range(u,theta,g=g):
    """
    Calculates the range of a projectile launched with velocity
    u (m/s) and angle theta (degrees) given a constant vertical
    acceleration due to gravity g (m/s/s).
    """
    return (u**2)*np.sin(2*np.deg2rad(theta))/g
```

[47]:
```
projectile_range(100,45,9.8)
```

[47]:
```
1020.408163265306
```

[49]:
```
projectile_range_earth = partial(projectile_range,g=9.8)
projectile_range_moon  = partial(projectile_range,g=9.8/10)
```

[51]:
```
projectile_range_earth(100,45)
```

[51]:
```
1020.408163265306
```

[52]:
```
projectile_range_moon(100,45)
```

[52]:
```
10204.08163265306
```