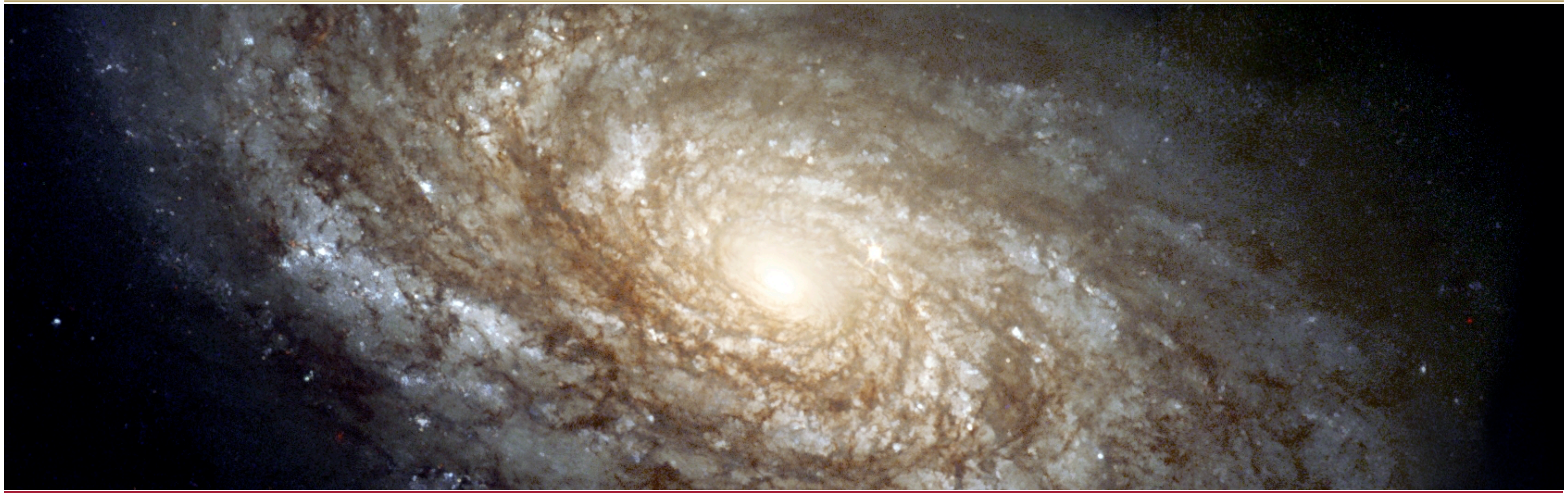
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Institute of Astronomy and Astrophysics



N-Body modelling with REBOUND

Hands-on Numerical Astrophysics School
for Exoplanetary Sciences, July 4-8, 2022

Exoplanet Diversity
DFG SPP-1992

4 July 2022 • Anna Penzlin, Christoph Schäfer

Topics

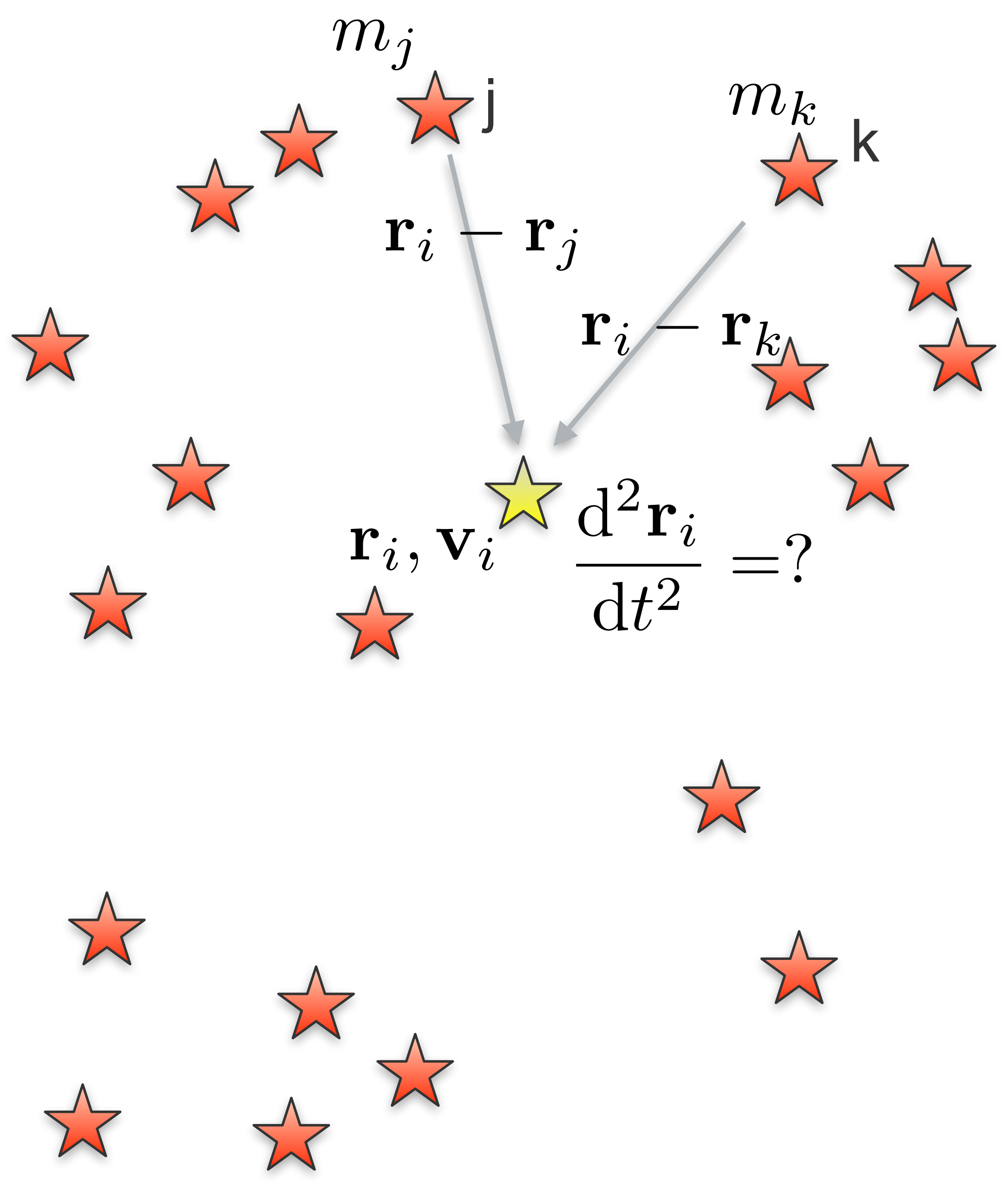
- Theory part
  - The classical astrophysical N-body problem
  - Exact N-body schemes and Tree algorithm
  - Some N-body codes
  - REBOUND integrator package by Hanno Rein
  - REBOUNDx package by Daniel Tamayo

- Hands-on exercises part
  - Two-Body problem
  - Saturn’s rings stability
  - Stability of planetary system
  - Kirkwood gaps
  - Migrating planet
  - <your project comes here> ….

The classical astrophysical N-body problem

A number of N particles interact classically through Newton’s Laws of Motion and Newton’s Law of Gravitation.

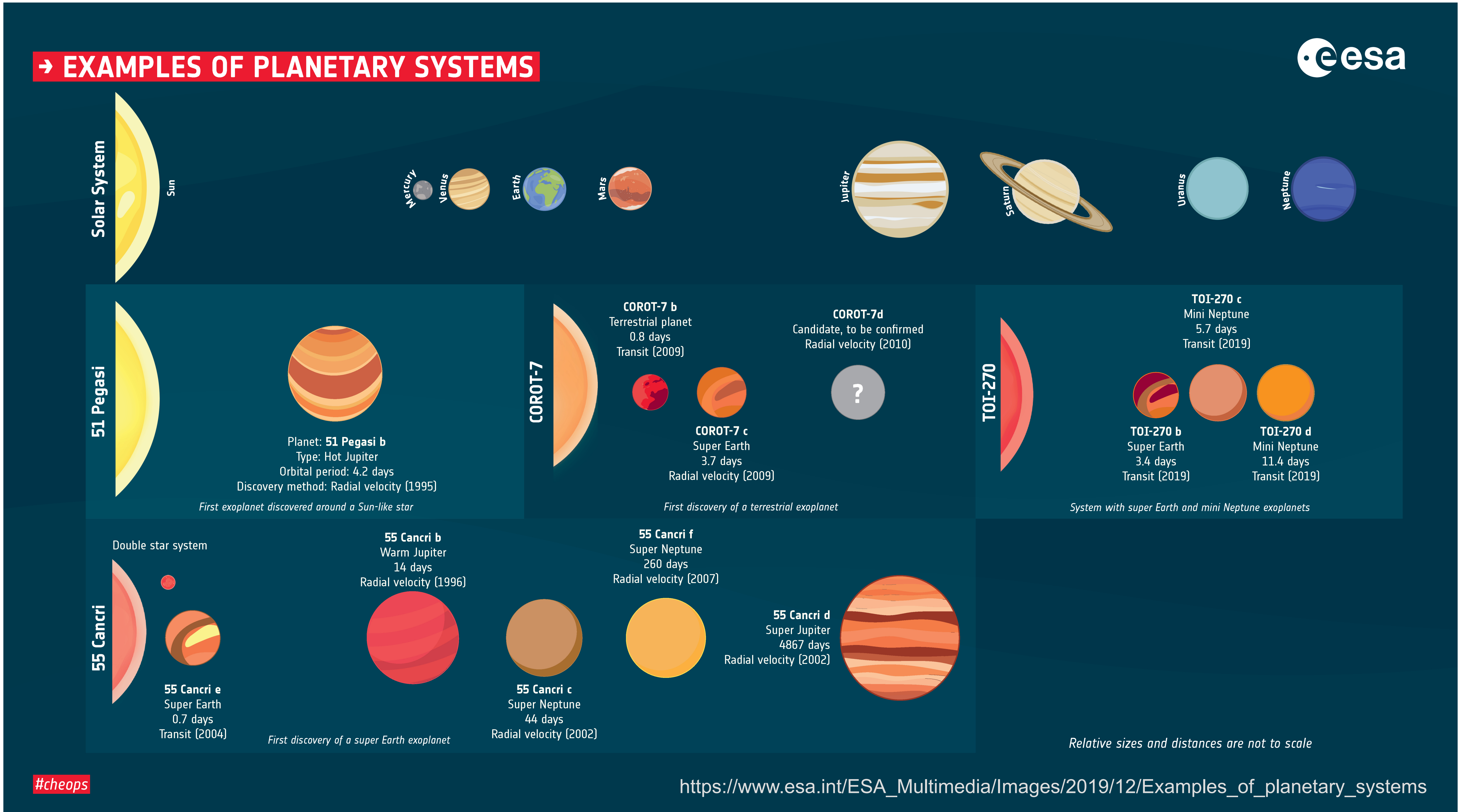
$$\frac{\mathrm{d}^2\mathbf{r}_i}{\mathrm{d}t^2} = -G\sum_{j\neq i}^{N} m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}$$


m_j
j
m_k
k
$\mathbf{r}_i - \mathbf{r}_j$
$\mathbf{r}_i - \mathbf{r}_k$
$\mathbf{r}_i, \mathbf{v}_i$
$\frac{\mathrm{d}^2\mathbf{r}_i}{\mathrm{d}t^2} = ?$

For N=1 and N=2, the equation of motion can be solved analytically.

Applications

- Few-body systems N≅ 3-10



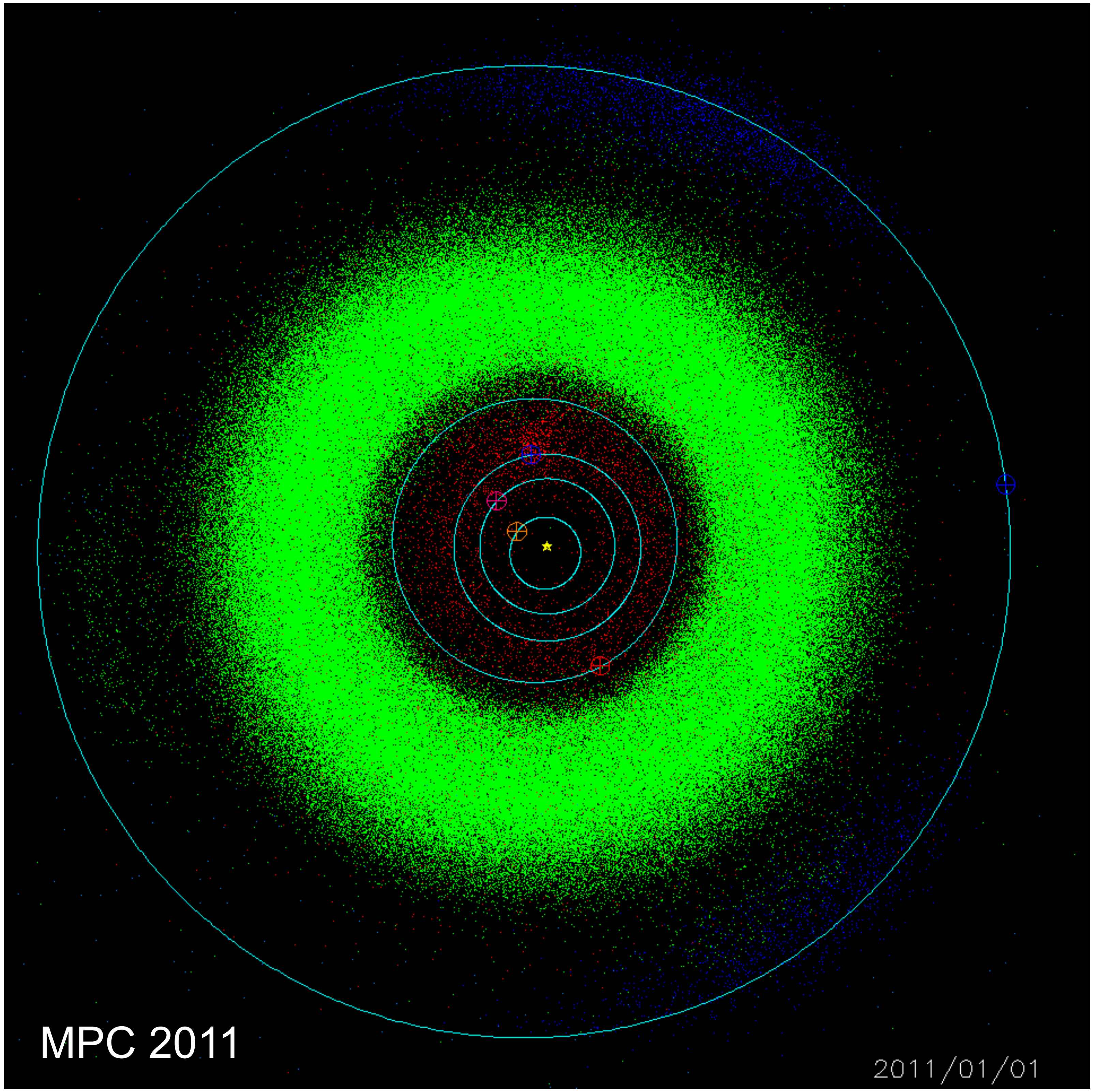
https://www.esa.int/ESA_Multimedia/Images/2019/12/Examples_of_planetary_systems

Applications

- Many-body systems $N \cong 10$-400 and tracers

  planetary systems with minor bodies

  formation of asteroid families

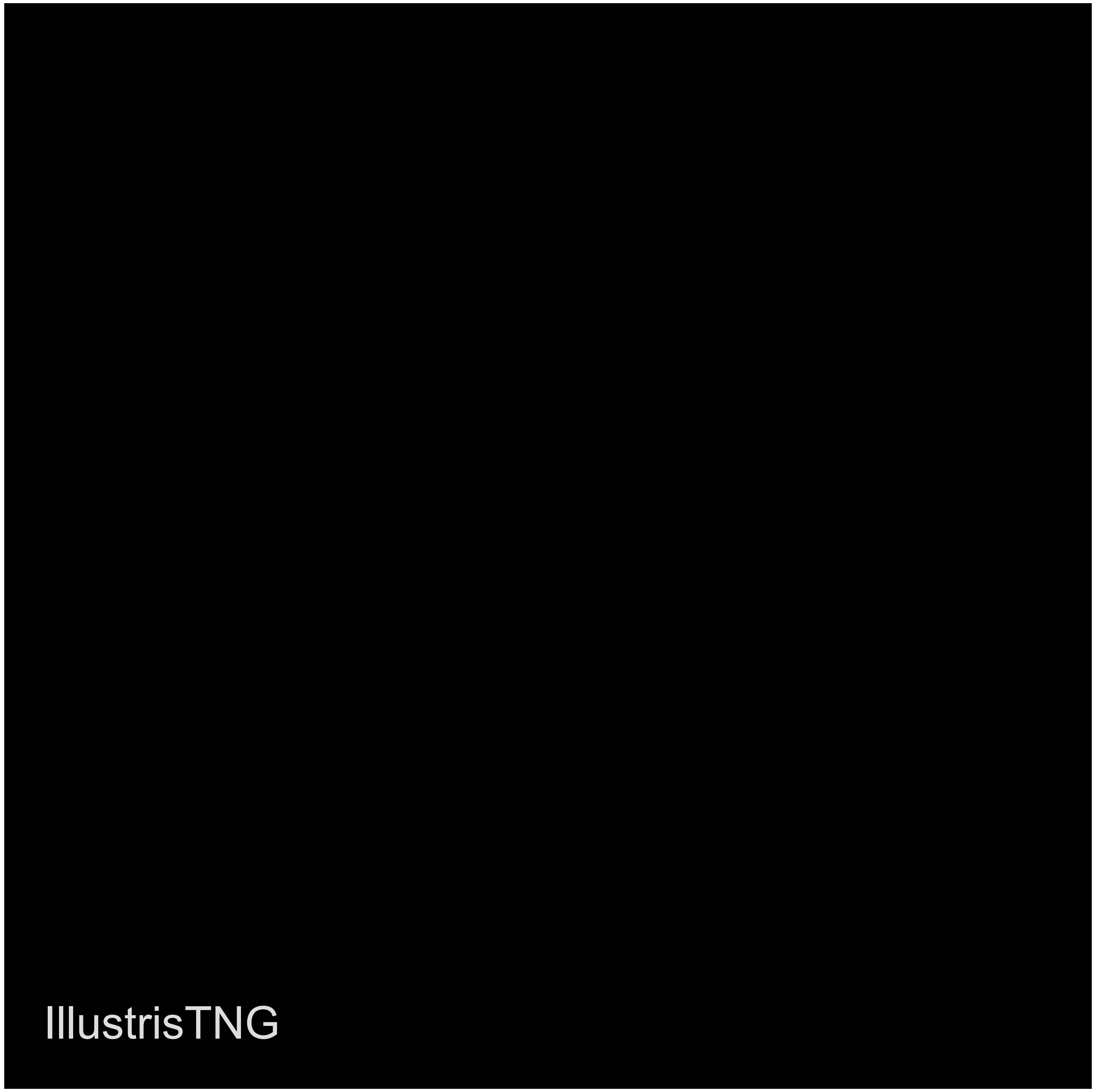


Applications

- Large N-body systems
  - globular star clusters $N > 10^3$
  - large-scale structure of the universe
  - galactic dynamics and cosmology $N > 10^6$

Applications

- Large N-body systems
  - globular star clusters N > 10^3
  - large-scale structure of the universe
  - galactic dynamics and cosmology N > 10^6
  - coupled hydro-nbody simulations, Tree-Particle-Mesh (Tree-PM)



Applications · Gravity is everywhere

- Few-body systems $N \cong 3\text{-}10$
  - planetary systems

celestial mechanics

- Many-body systems $N \cong 10\text{-}400$ and tracers
  - planetary systems with minor bodies - formation of asteroid families

- Large N-body systems
  - globular star clusters $N > 10^3$

dense stellar systems

  - large-scale structure of the universe
  - galactic dynamics and cosmology $N > 10^6$

galactic dynamics

Applications · today during the workshop

- Few-body systems N≅ 3-10
  - planetary systems

celestial mechanics

- Many-body systems N≅10-400 and tracers
  - planetary systems with minor bodies - formation of asteroid families

- Large N-body systems
  - globular star clusters $N > 10^3$

dense stellar systems

  - large-scale structure of the universe
  - galactic dynamics and cosmology $N > 10^6$

galactic dynamics

Time Integrators

- Consider the following problem

$$\begin{aligned} y'(t) &= \frac{\mathrm{d}y}{\mathrm{d}t} = f(t,(y(t)), \quad a \le t \le b, \\ y(a) &= y_0 \,. \end{aligned}$$

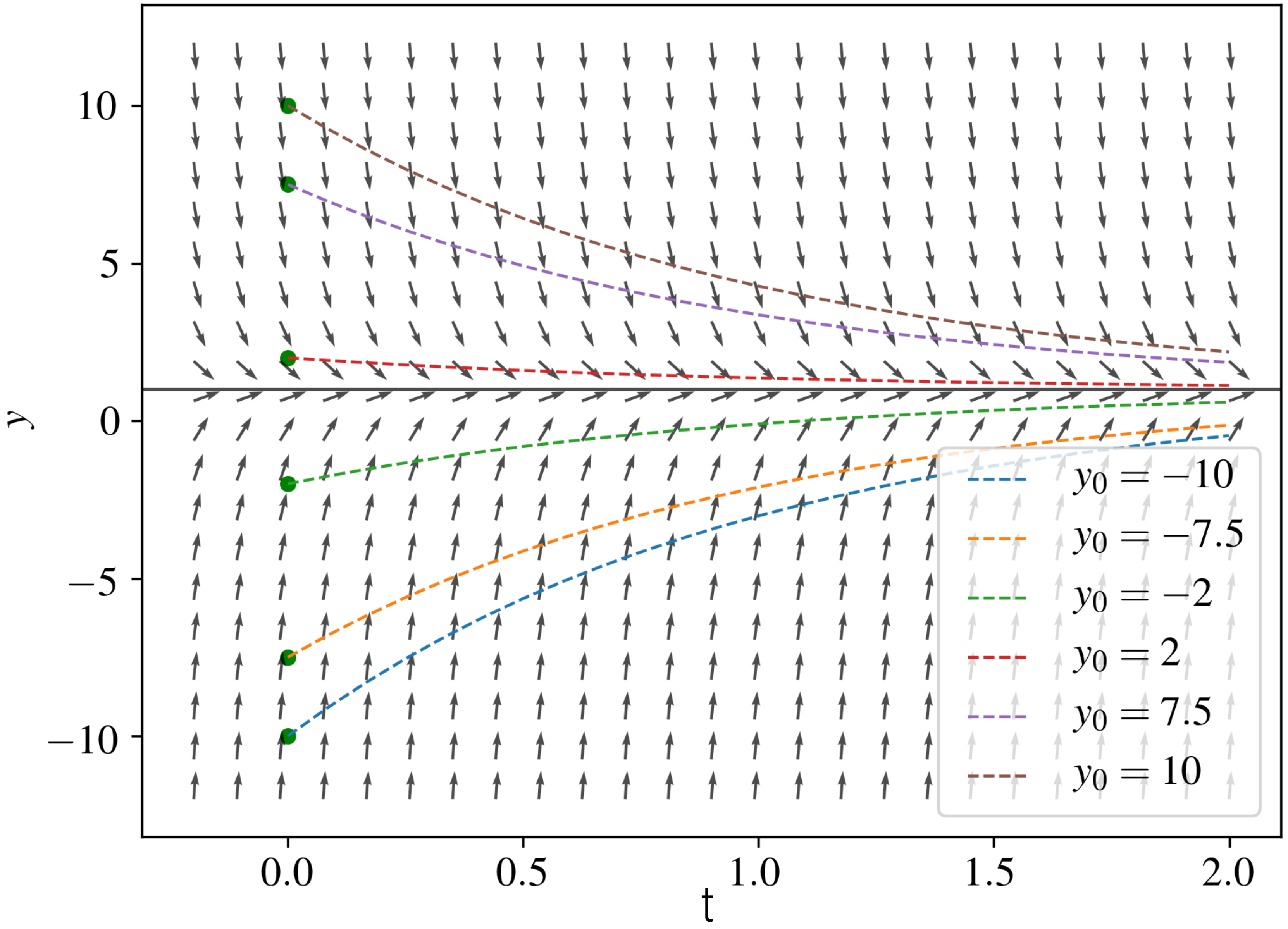
- Finite interval [a,b] and function f(t,y) are given.

- We seek a function y(t) defined on [a,b], such that $y'(t) = f(t,y(t))$ for $a \le t \le b$, and such that y(t) satisfies the initial condition $y(a) = y_0$, where y_0 is some prescribed value.

- This is called an **initial value problem**, cf. N-Body problem.

Time Integrators

- The IVP consists of two parts: the differential equation which gives the relationship between y(t) and y′(t) and the initial condition y(a) = y_0.



- Slope field, vector plot at points (t, y) showing the slope [1, f (t, y(t))]. It gives an impression on the type of possible solutions. The choice of y_0 selects the specific, corresponding solution.

Direct N-Body

set of N·dimension 2nd order differential equations

$$\frac{\mathrm{d}^2\mathbf{r}_i}{\mathrm{d}t^2} = -G\sum_{j\neq i}^{N} m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}$$

$\longrightarrow$ 2N·dimension sets of 1st order differential equations

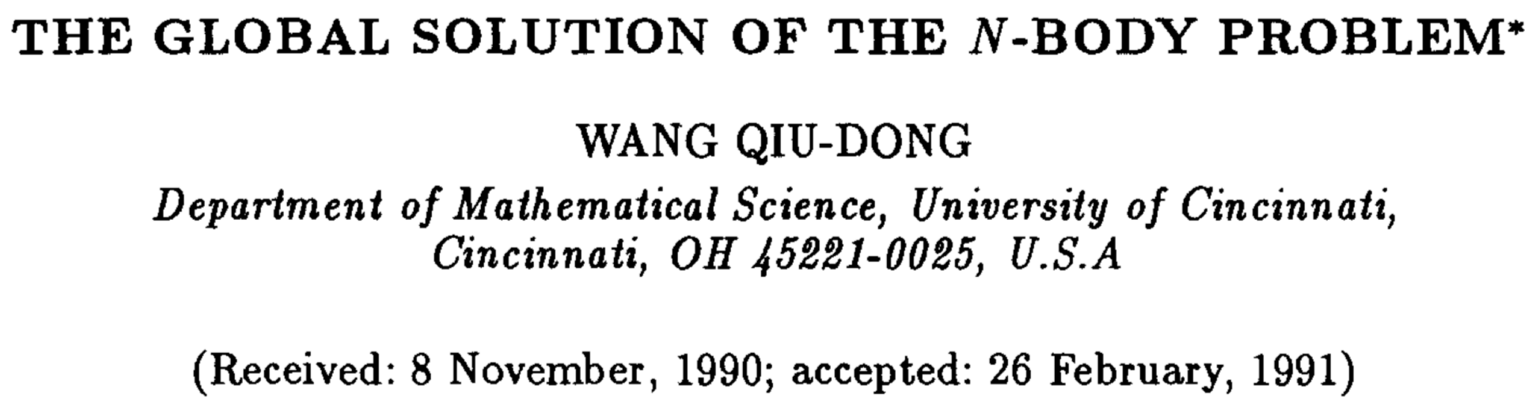
$$\frac{\mathrm{d}\mathbf{r}_i}{\mathrm{d}t} = \mathbf{v}_i \qquad \frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = \mathbf{a}_i = -G\sum_{j\neq i}^{N} m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}$$

initial value problem: 2N·dimension additional conditions needed

$$\mathbf{r}_i(t_0), \mathbf{v}_i(t_0)$$

initial positions and velocities have to be known 🤔

Direct N-Body cont’d



THE GLOBAL SOLUTION OF THE *N*-BODY PROBLEM*

WANG QIU-DONG

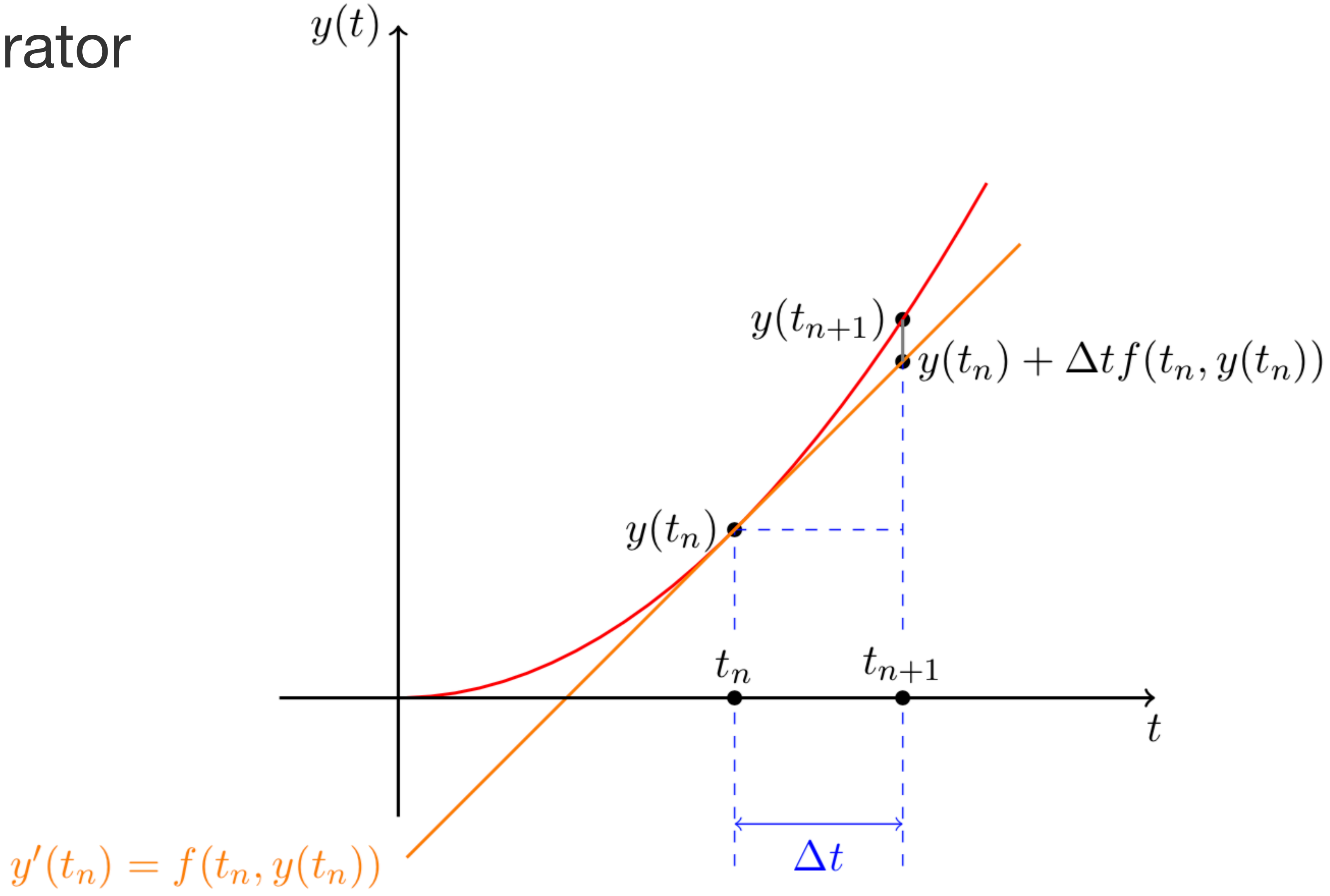
Department of Mathematical Science, University of Cincinnati, Cincinnati, OH 45221-0025, U.S.A

(Received: 8 November, 1990; accepted: 26 February, 1991)

(2) Some comments: (i) Although the conclusion given here provides a way to integrate the n-body problem, one does not obtain a useful solution in series expansion. The reason for this is because the speed of convergence of the resulting solution is terribly slow. One has to sum, for example, an incredible number of terms, even for an approximate solution of first order in q, p, t. Because we know almost nothing about the complex singular point in the 7-plane, it seems hopeless to try to improve the convergence of such a series expansion.

Direct N-Body cont'd

Numerical integration of the equations of motion
challenge: high accuracy vs. low computational cost
e.g., simple Euler integrator


$y(t)$
t
$y(t_{n+1})$
$y(t_n) + \Delta t f(t_n, y(t_n))$
$y(t_n)$
t_n
t_{n+1}
Δt
$y'(t_n) = f(t_n, y(t_n))$

Direct N-Body cont'd

Numerical integration of the equations of motion

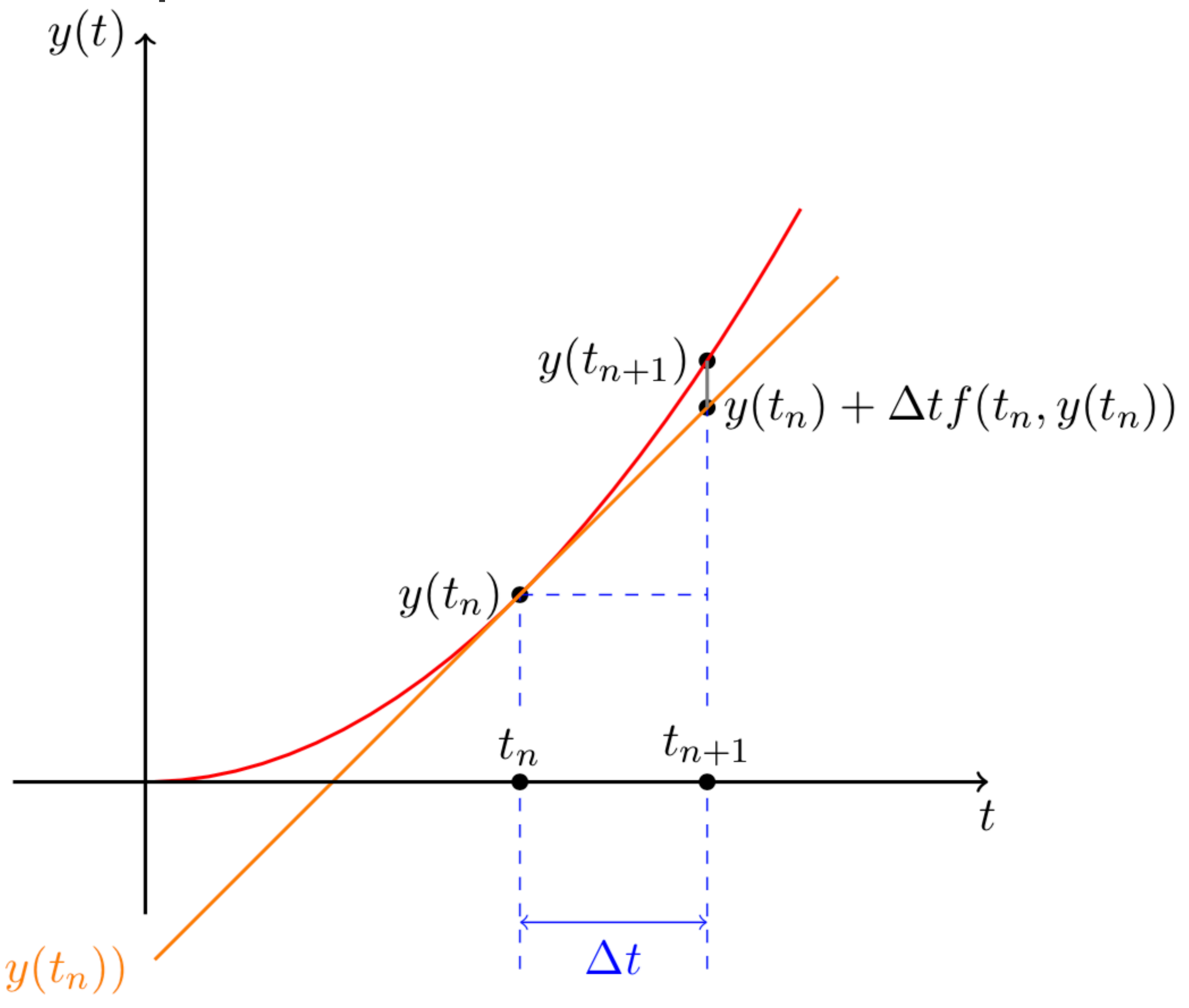
challenge: high accuracy vs. low computational cost

e.g., simple Euler integrator

$$\mathbf{v}_i(t_{n+1}) = \mathbf{v}_i(t_n) + \mathbf{a}_i(t_n)\Delta t$$

$$\mathbf{r}_i(t_{n+1}) = \mathbf{r}_i(t_n) + \mathbf{v}_i(t_n)\Delta t$$

$$\mathbf{a}_i = -G\sum_{j\neq i}^{N} m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}$$


$y(t)$
$y(t_{n+1})$
$y(t_n) + \Delta t f(t_n, y(t_n))$
$y(t_n)$
t_n
t_{n+1}
t
Δt
$y'(t_n) = f(t_n, y(t_n))$

Direct N-Body cont'd

Numerical integration of the equations of motion

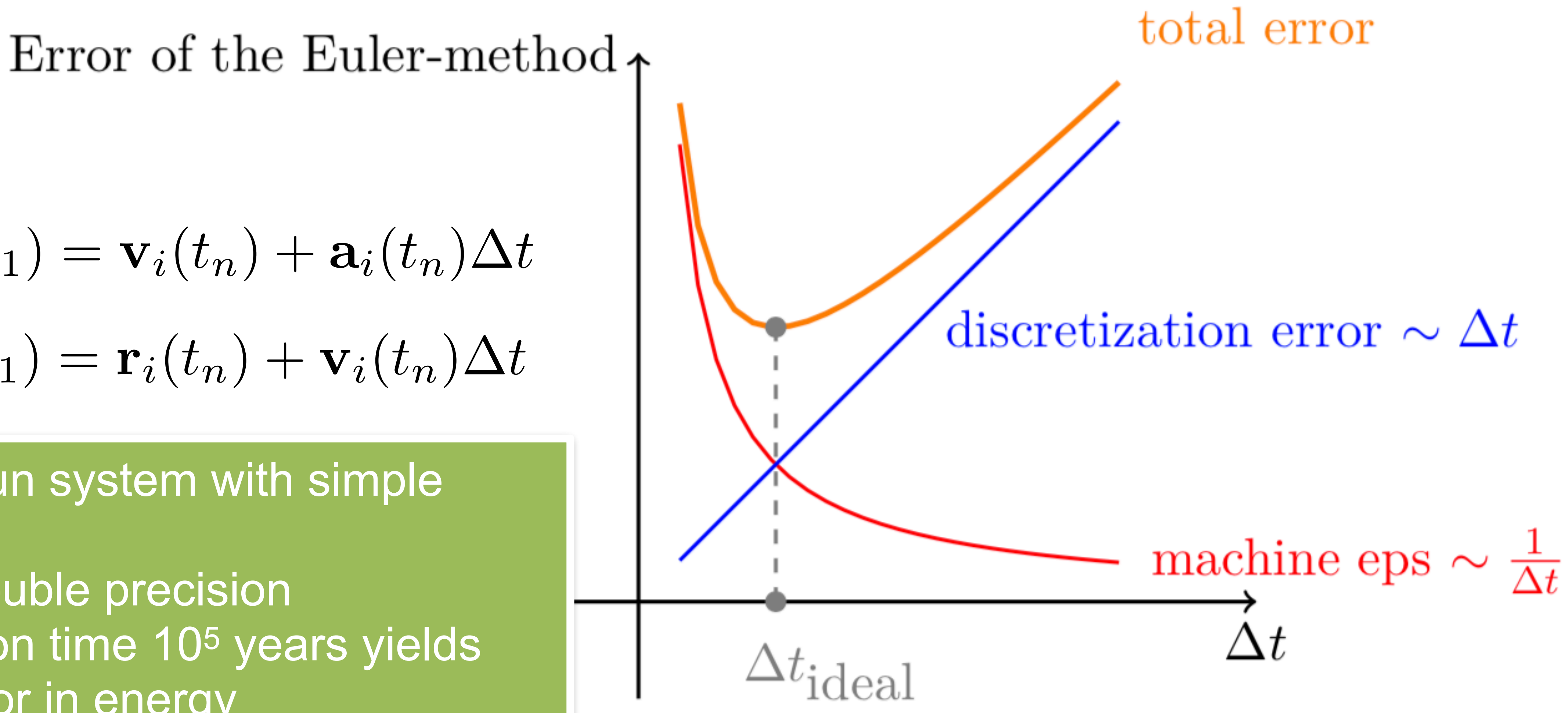
Different types of errors

e.g., simple Euler integrator

$$\mathbf{v}_i(t_{n+1}) = \mathbf{v}_i(t_n) + \mathbf{a}_i(t_n)\Delta t$$

$$\mathbf{r}_i(t_{n+1}) = \mathbf{r}_i(t_n) + \mathbf{v}_i(t_n)\Delta t$$

Earth-Sun system with simple Euler:
using double precision
simulation time 10^5 years yields
10% error in energy



Direct N-Body cont'd

Numerical integration of the equations of motion
A vast number of integrators have been developped

one step methods: Euler, Runge-Kutta,…

multi step methods: Leap-Frog, Adams–Bashforth, Nyström, Hermite,…

symplectic integrators: Leap-Frog, Wisdom-Holman, symplectic RKs,…

$$y_{n+1} = y_n + \Delta t \Phi(y, t, \Delta t)$$

$$y_{n+1} = y_n + \Delta t \sum_{i=0}^{q-1} \beta_i f(t_{n-i}, y_{n-i})$$

Hamiltonian systems

energy conservation

e.g., Euler

$$\mathbf{v}_i(t_{n+1}) = \mathbf{v}_i(t_n) + \mathbf{a}_i(t_n)\Delta t$$

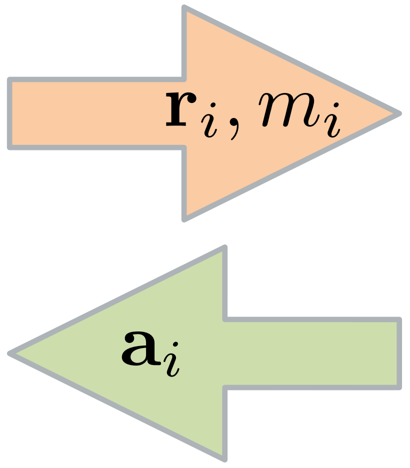
$$\mathbf{r}_i(t_{n+1}) = \mathbf{r}_i(t_n) + \mathbf{v}_i(t_n)\Delta t$$

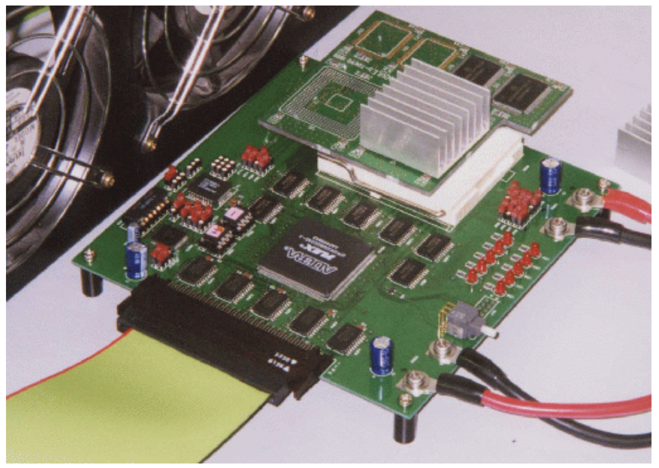
e.g., Leap-Frog for N-Body

$$\mathbf{r}_{n+1} = 2\mathbf{r}_n - \mathbf{r}_{n-1} + \mathbf{a}(\mathbf{r}_n)\Delta t^2$$

Direct N-Body cont’d

Gravitation is long range interaction
calculation of accelerations is N^2

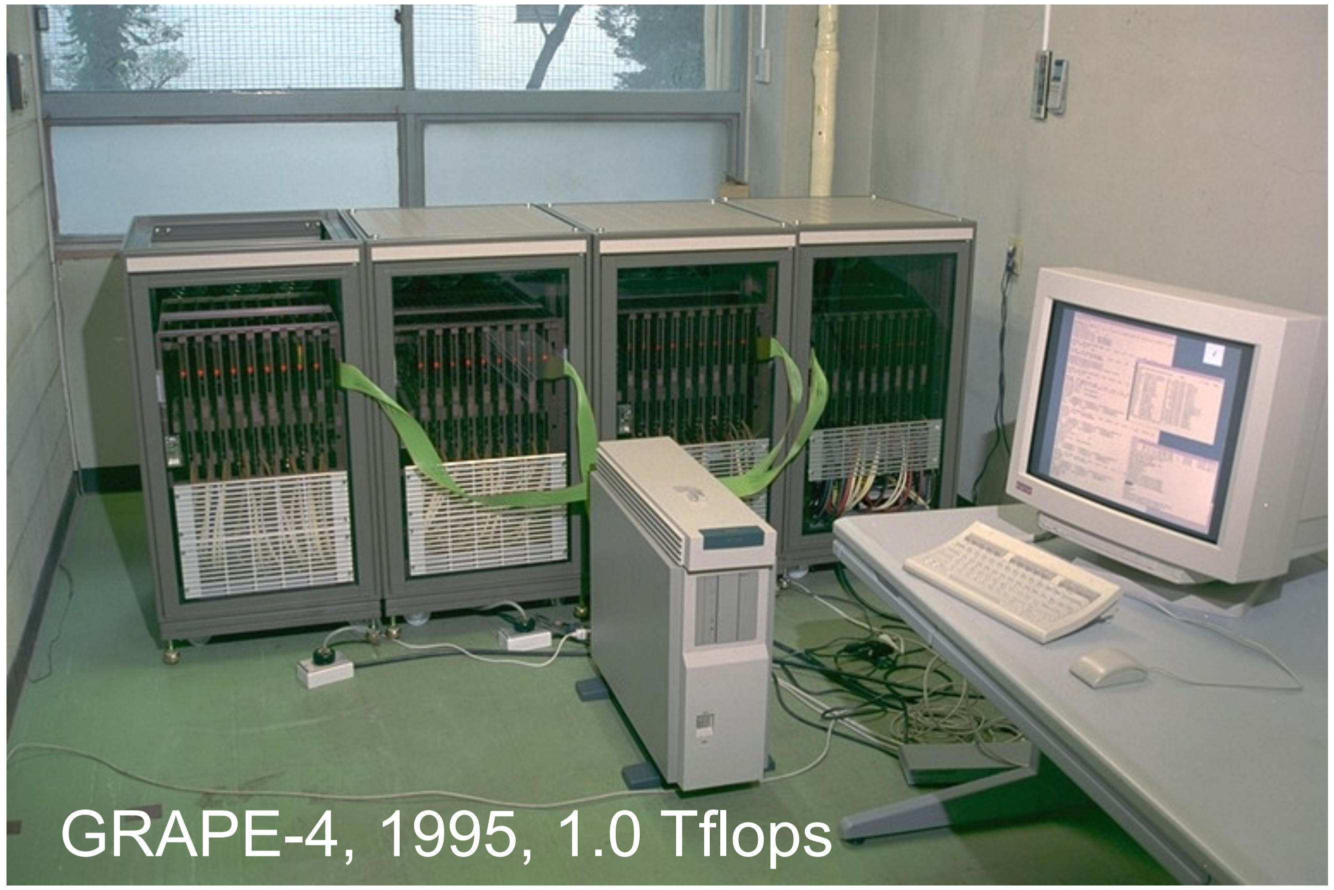




some ideas to speed up simulations

- special purpose hardware
  GRAvity PipE

last release 200x,
last supercluster 2005:
gravitySimulator, 32 GRAPE boards, 4 Tflops
128’000 particles


GRAPE-4, 1995, 1.0 Tflops

Approximate N-Body

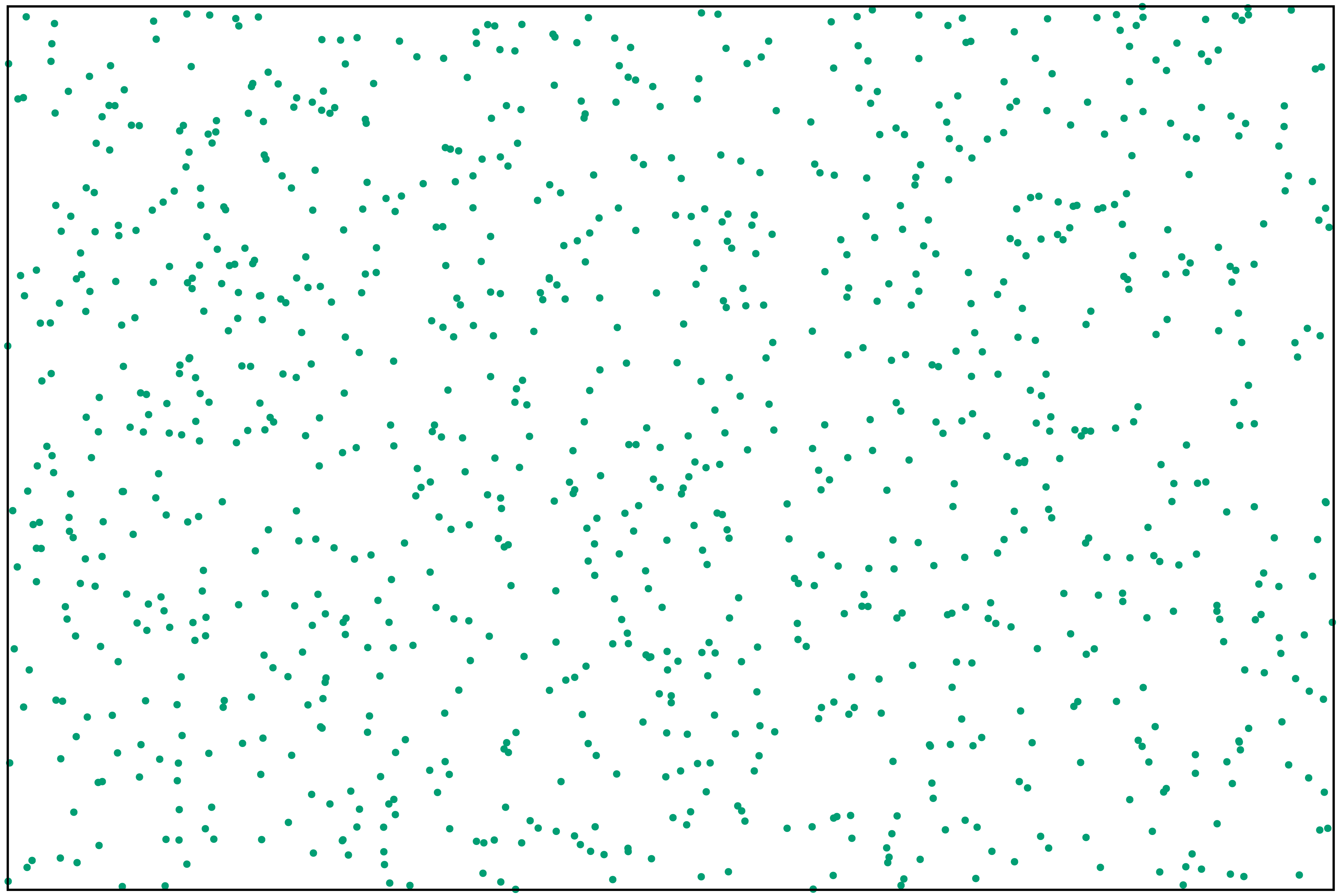
Gravitation is long range interaction
calculation of accelerations is N^2
can we decrease the computations for the cost of lower accuracy?

Barnes-Hut (1986) · hierarchical tree method
basic idea: **reduce** the number of terms in sum

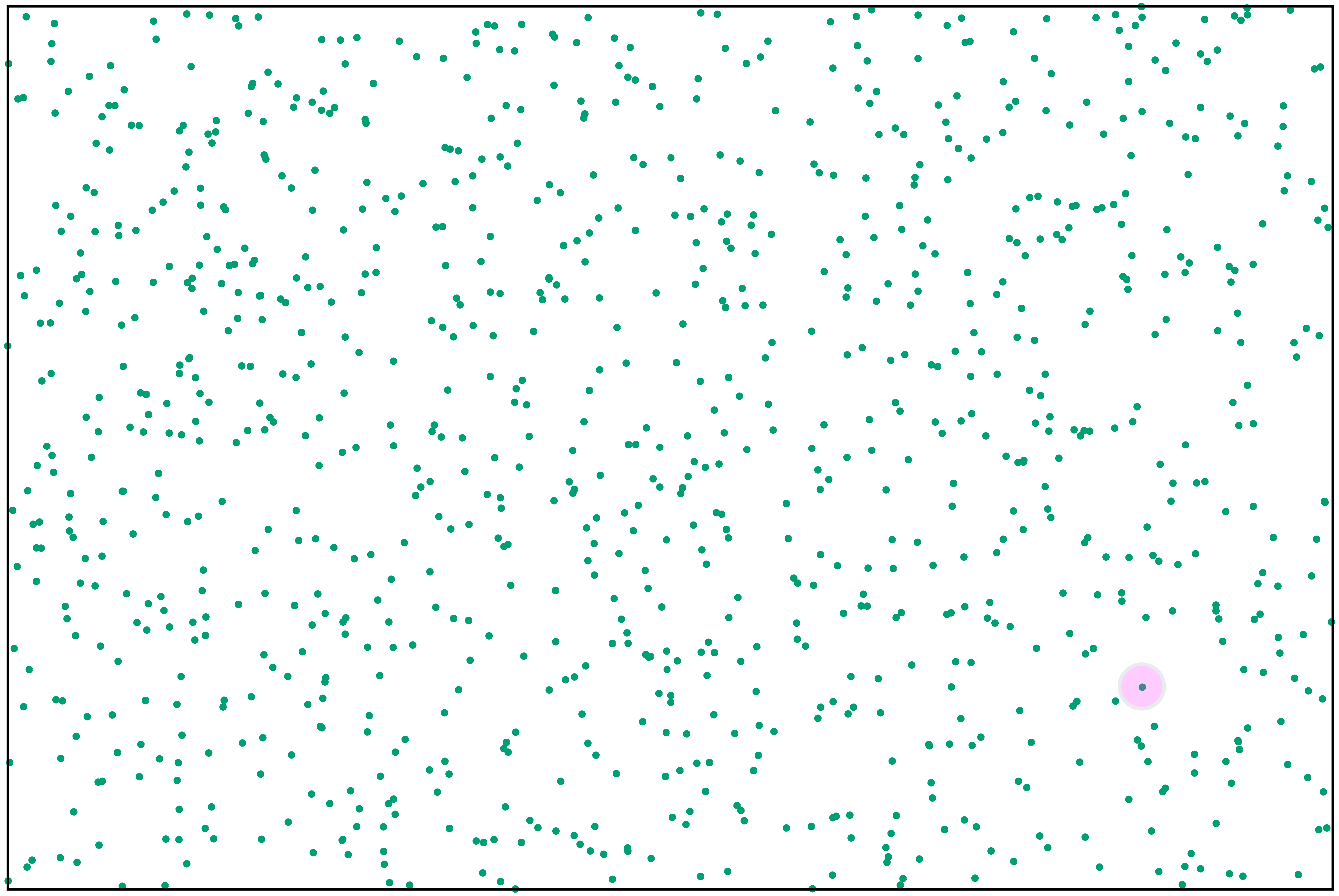
$$\mathbf{a}_i = -G \sum_{j \neq i}^{N} m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}$$

How can we achieve this in a physically correct way?

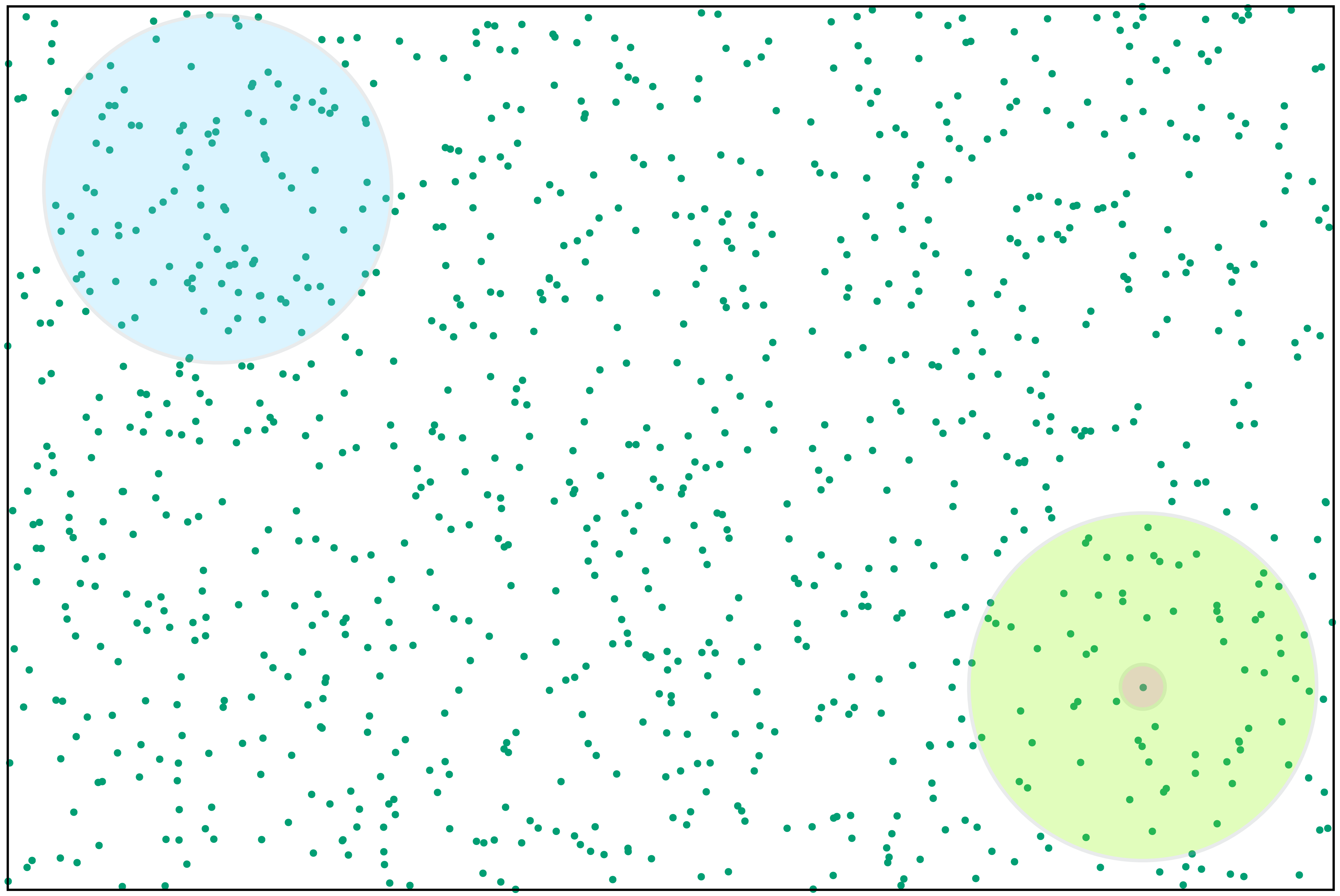
Approximate N-Body - Barnes-Hut Tree



Approximate N-Body - Barnes-Hut Tree

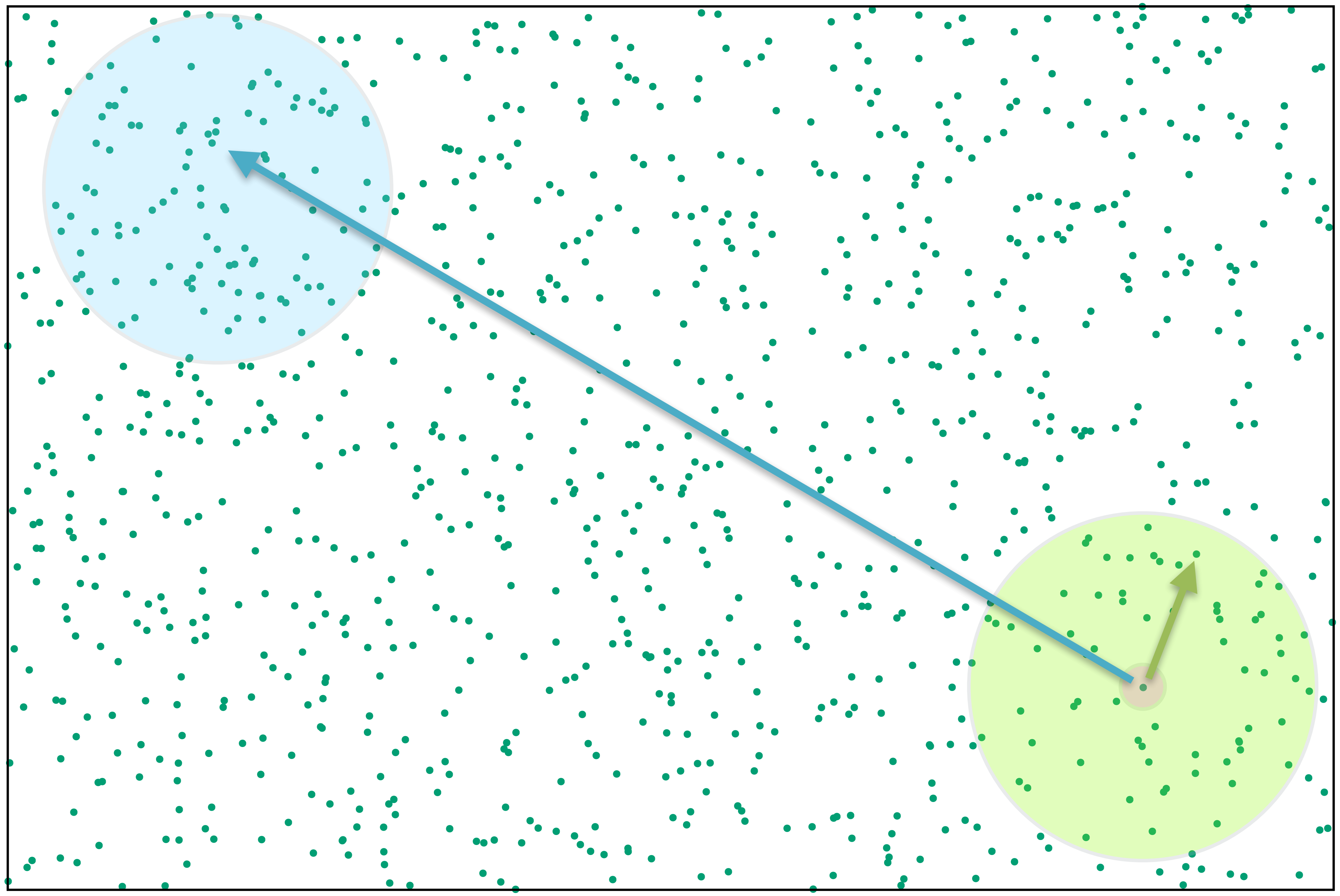


Approximate N-Body - Barnes-Hut Tree



$$\mathbf{a} \sim \frac{1}{r^2}$$

Approximate N-Body - Barnes-Hut Tree



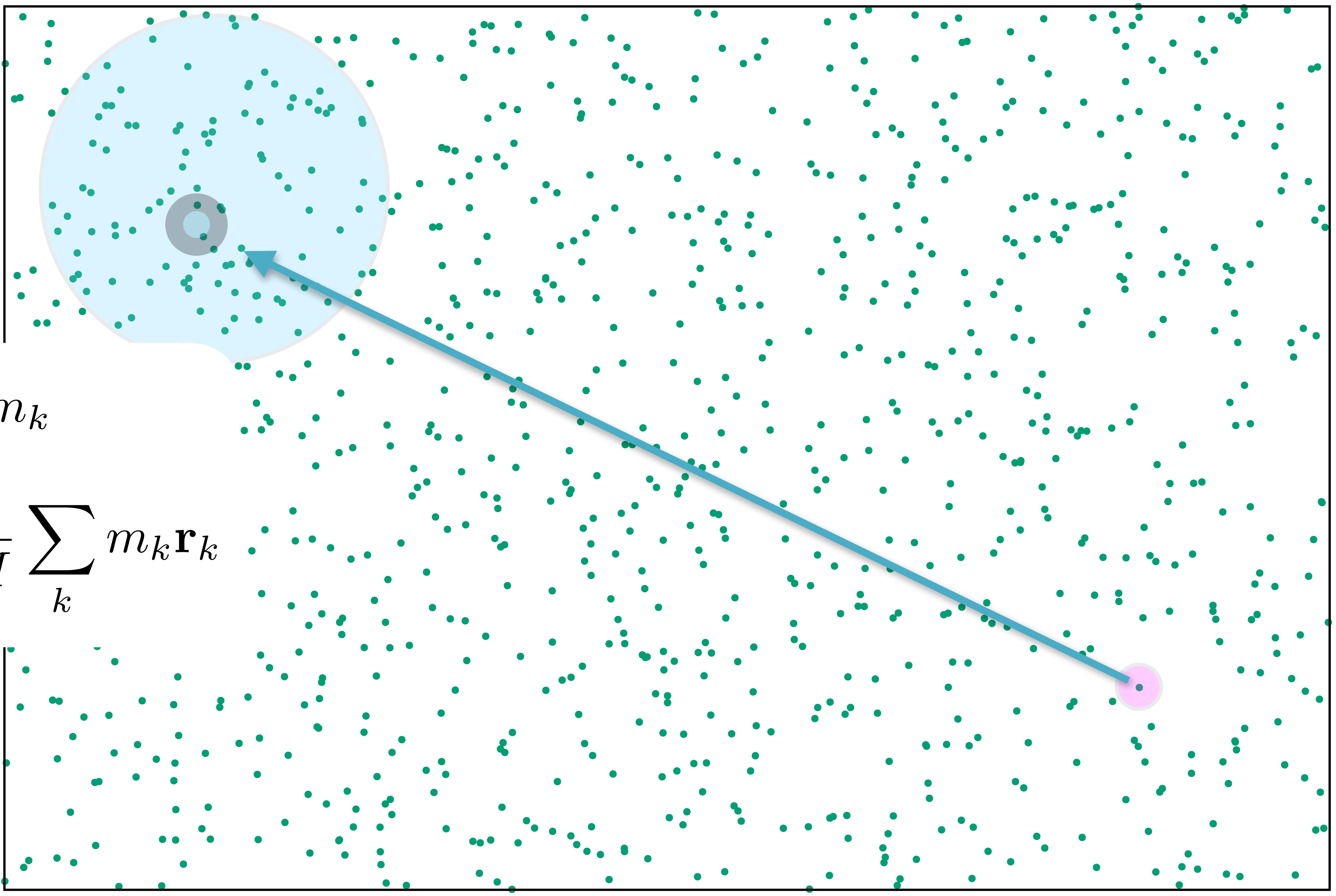
$$\mathbf{a} \sim \frac{1}{r^2}$$

Approximate N-Body - Barnes-Hut Tree

calculate center of mass and mass of blue region

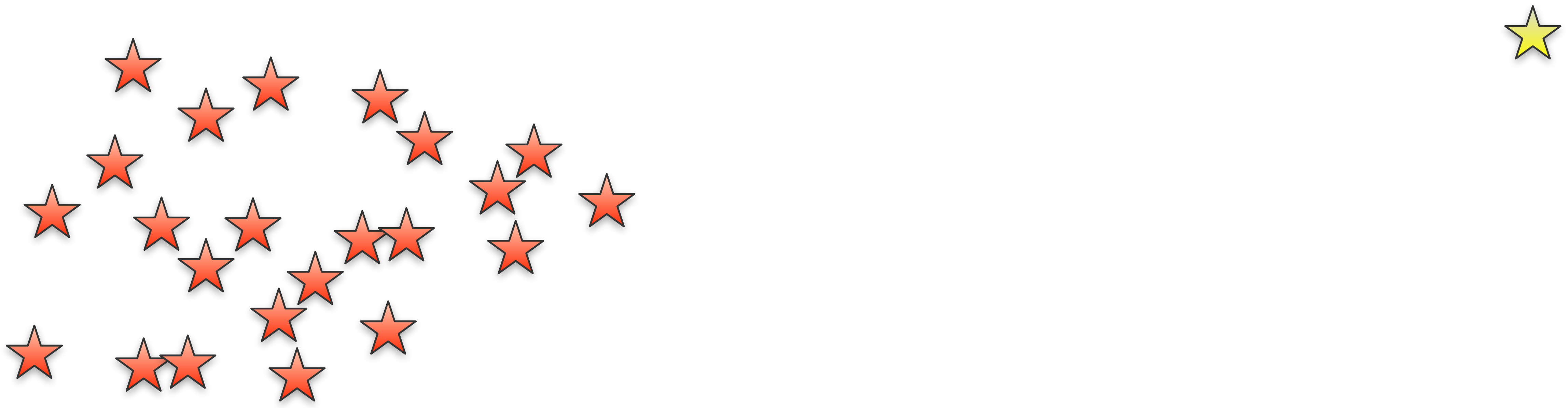
$$M = \sum_k m_k$$

$$\mathbf{r}_{\mathrm{com}} = \frac{1}{M} \sum_k m_k \mathbf{r}_k$$



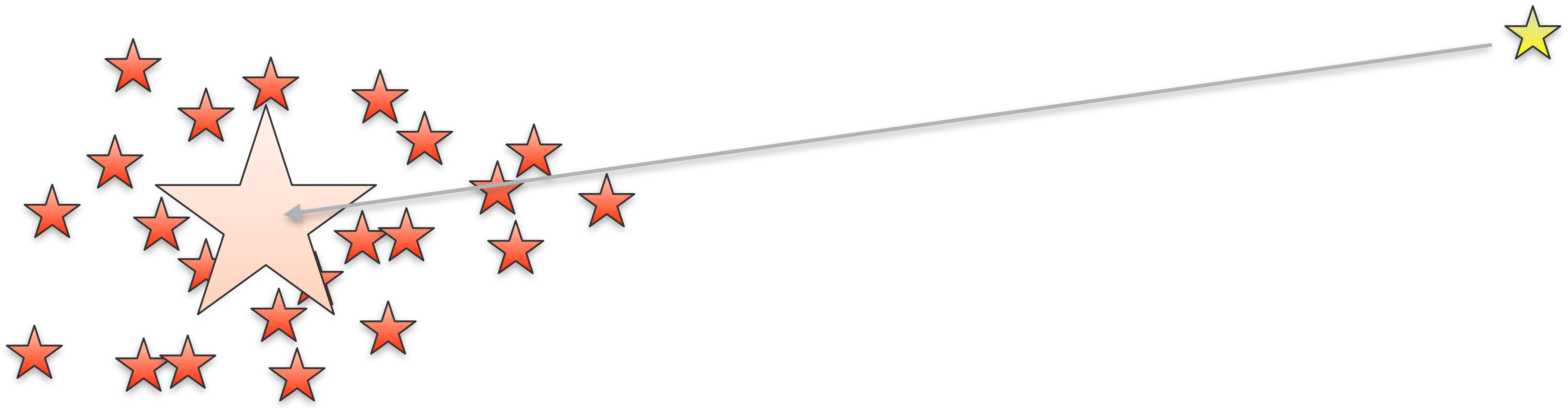
Approximate N-Body - Barnes-Hut Tree

- sort bodies into a hierarchical structure
- add condition for approximation of long range vs. short range



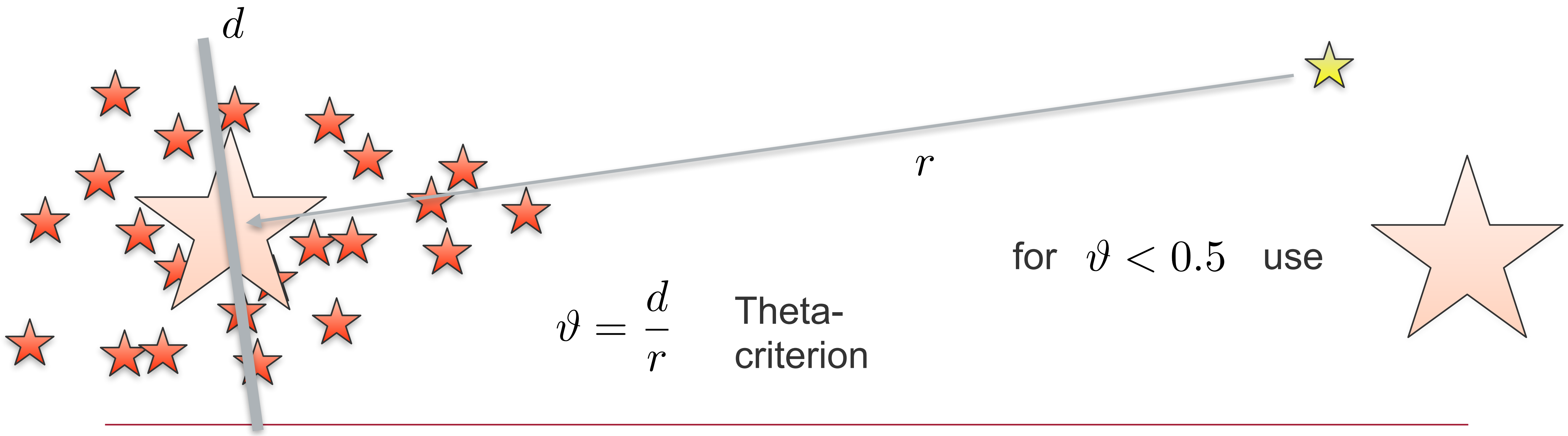
Approximate N-Body - Barnes-Hut Tree

- sort bodies into a hierarchical structure
- add condition for approximation of long range vs. short range



Approximate N-Body - Barnes-Hut Tree

- sort bodies into a hierarchical structure
- add condition for approximation of long range vs. short range
  theta criterion

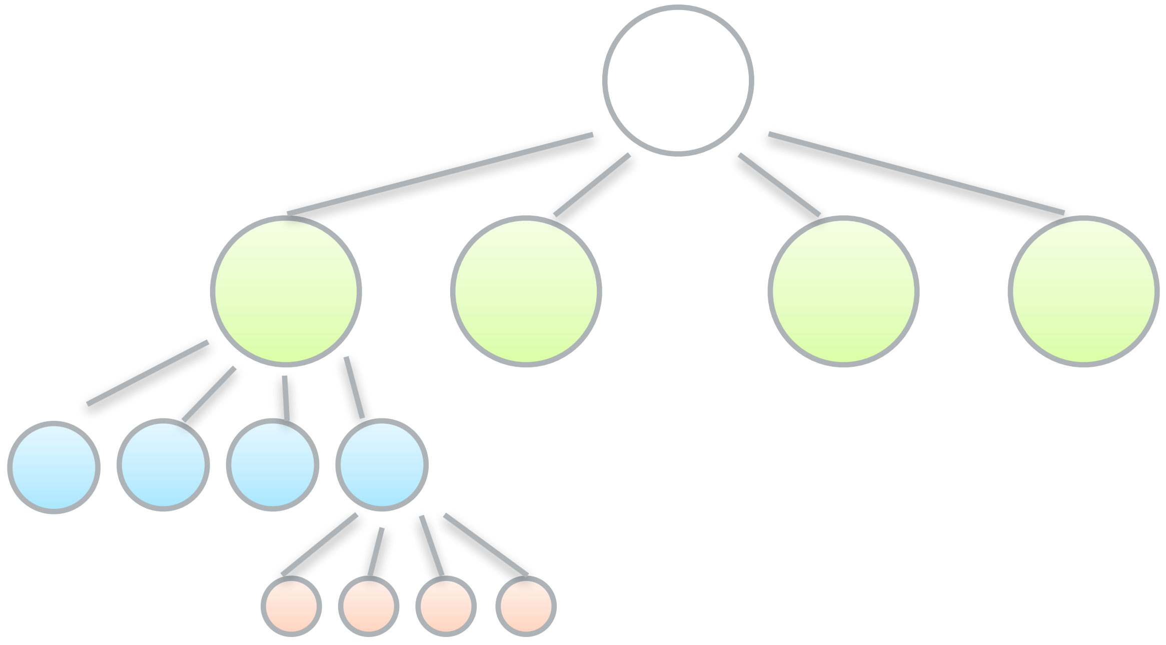


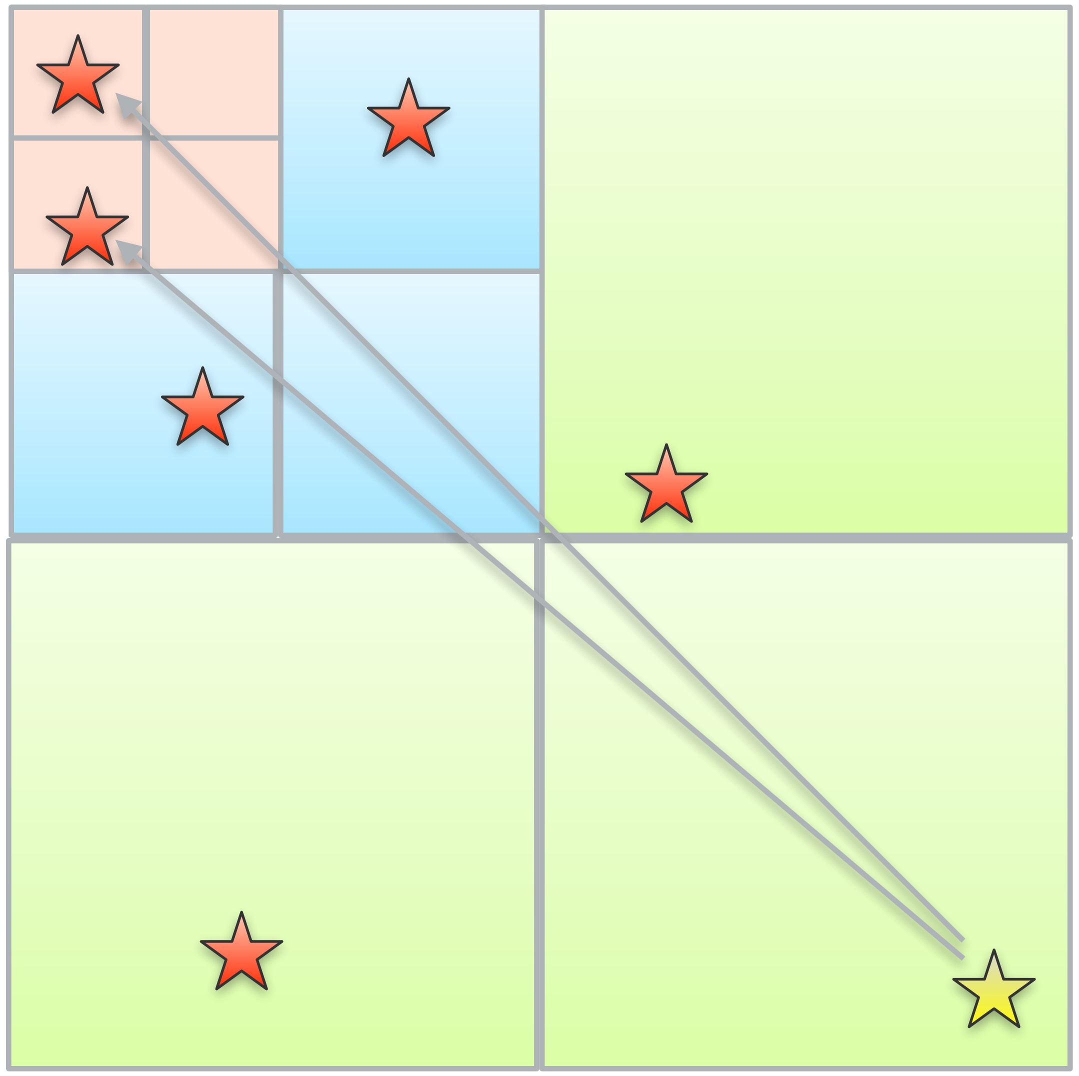
$$\vartheta = \frac{d}{r}$$ Theta-criterion

for $\vartheta < 0.5$ use

Approximate N-Body - Barnes-Hut Tree

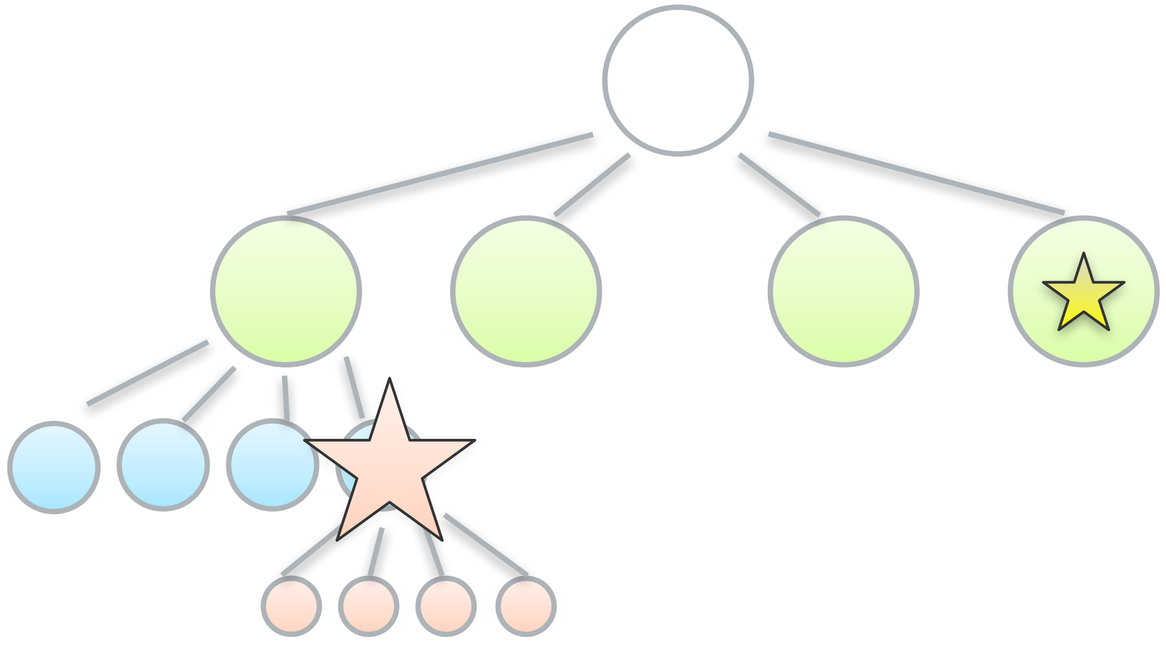
each node has $2^{\text{dimension}}$ children:
2D quadtree
3D octree

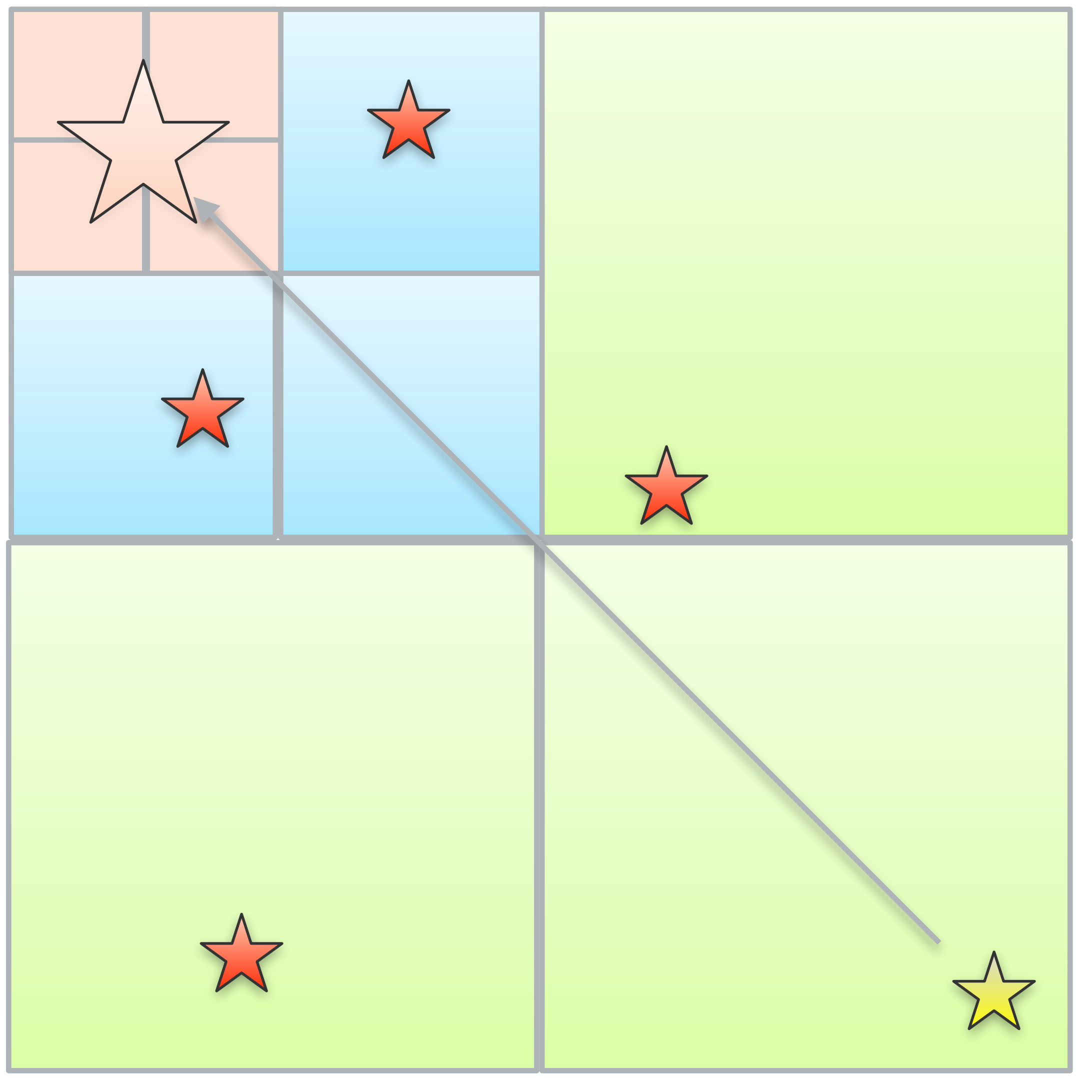




Approximate N-Body - Barnes-Hut Tree

each node has $2^{\text{dimension}}$ children:
2D quadtree
3D octree

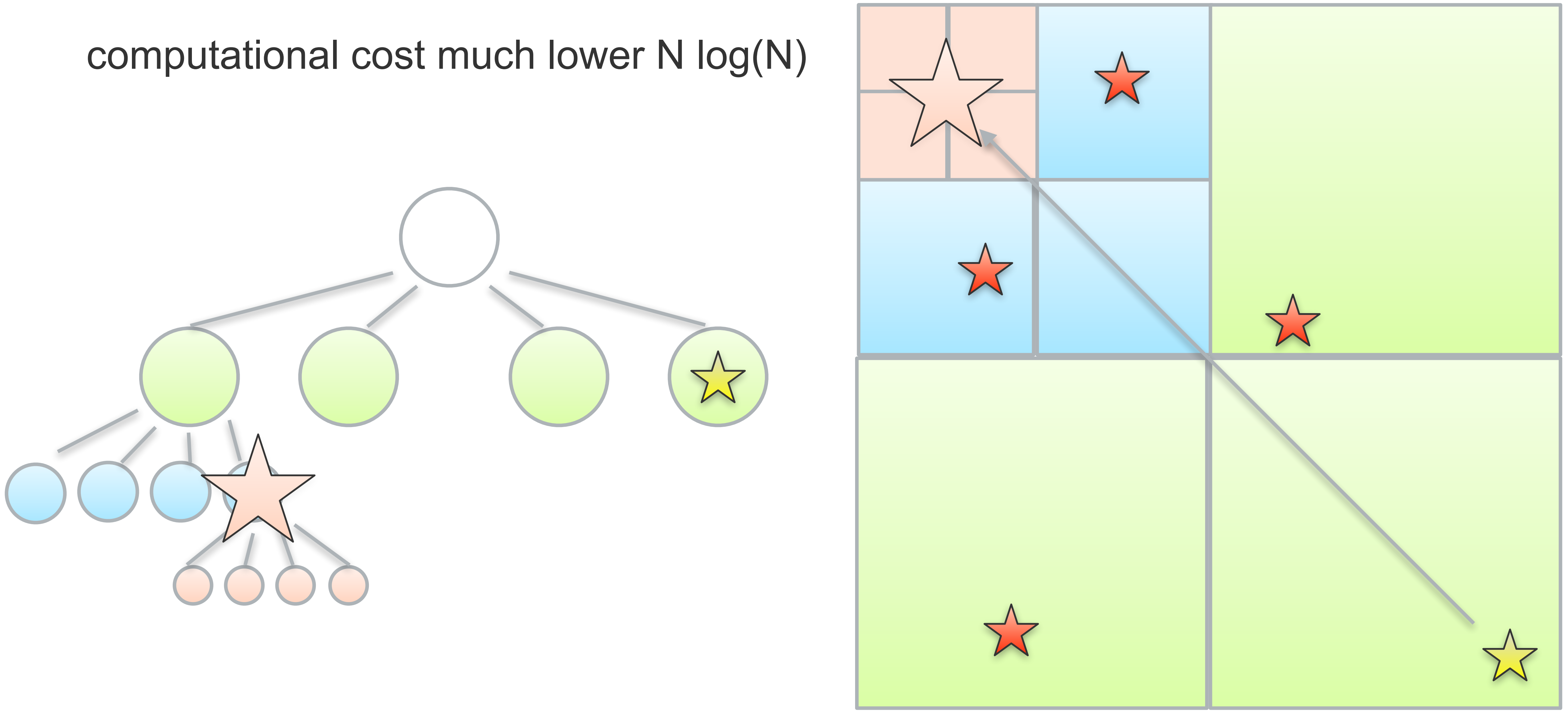




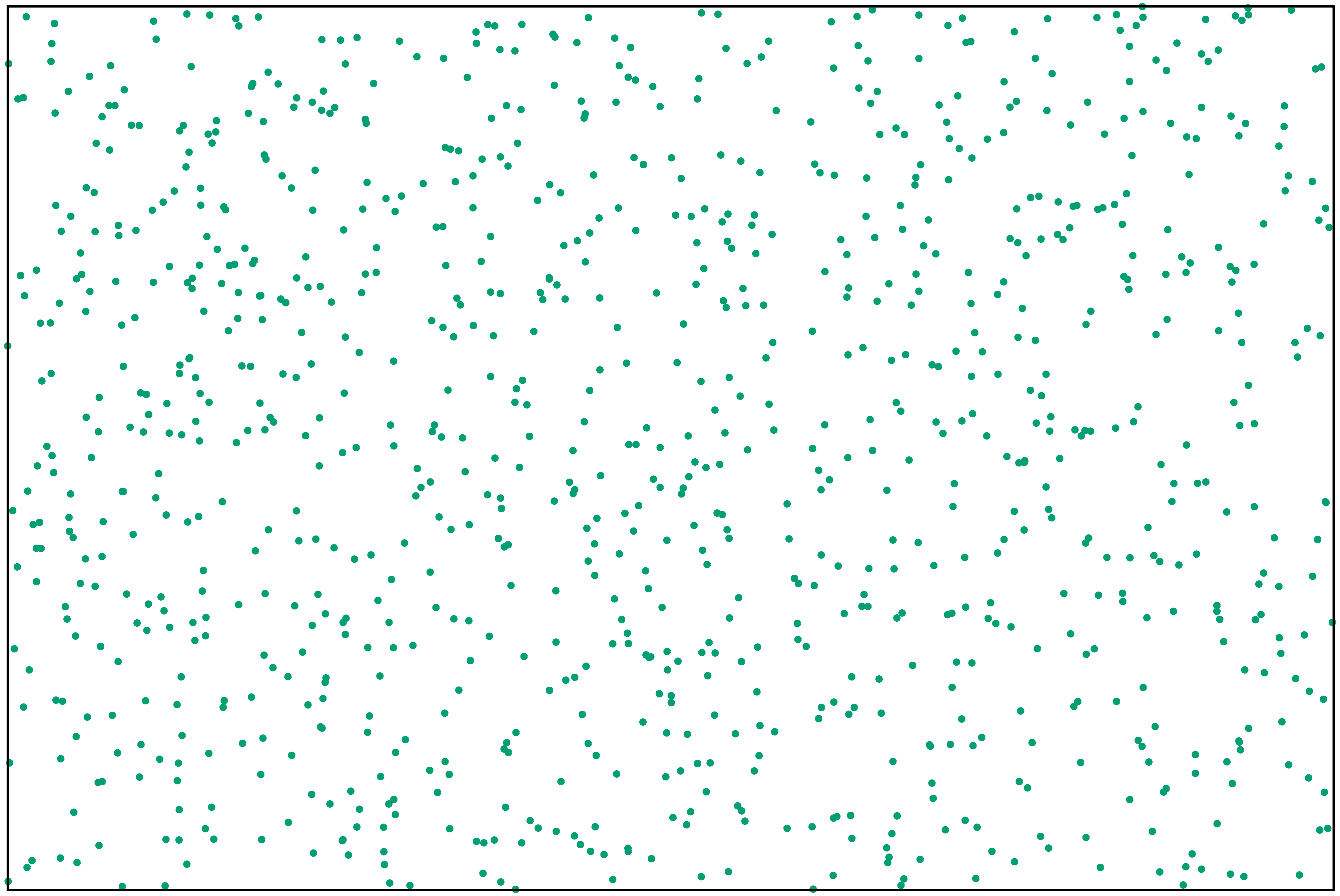
Approximate N-Body - Barnes-Hut Tree

data overhead compared to direct N-Body

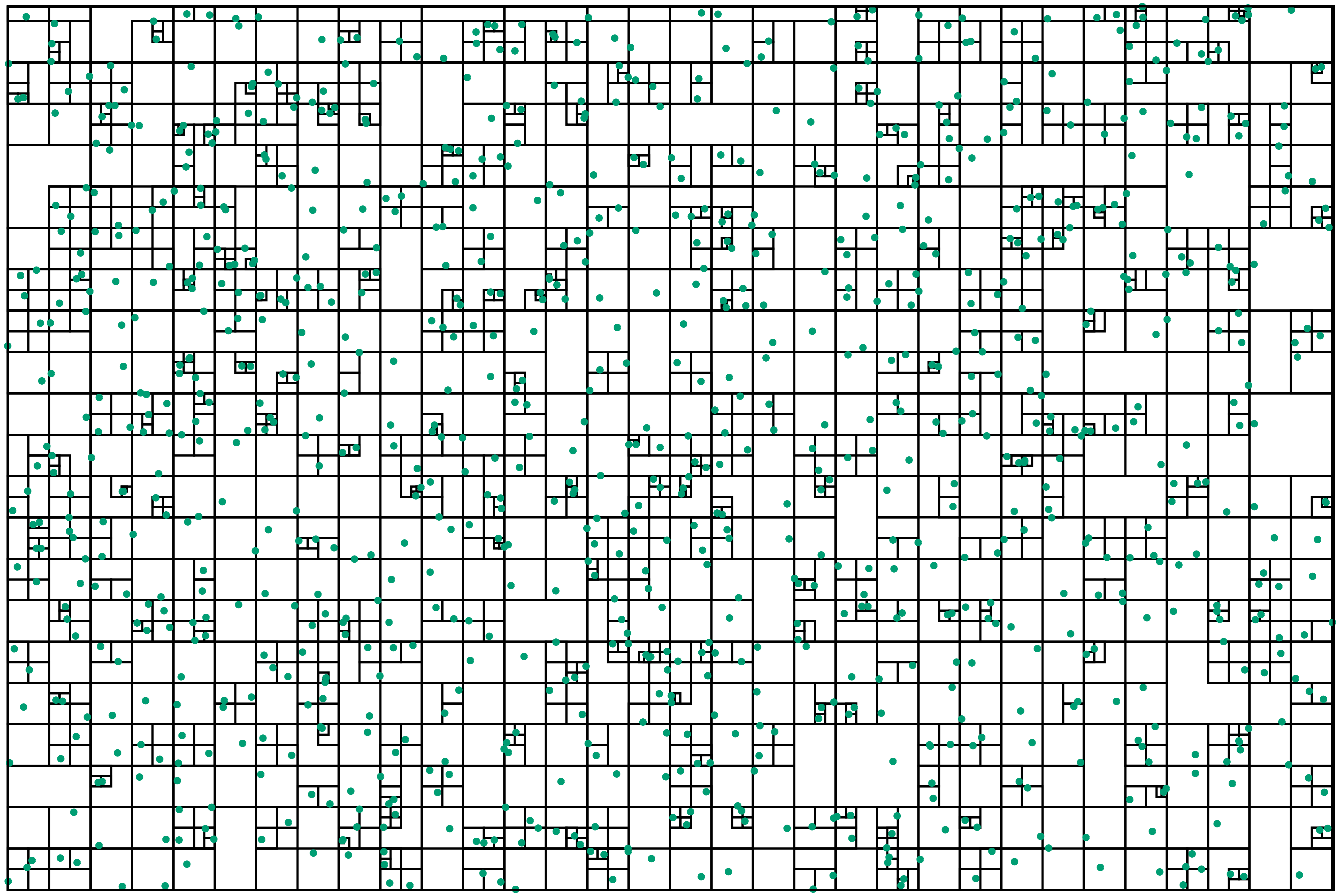
computational cost much lower N log(N)



Approximate N-Body - Barnes-Hut Tree

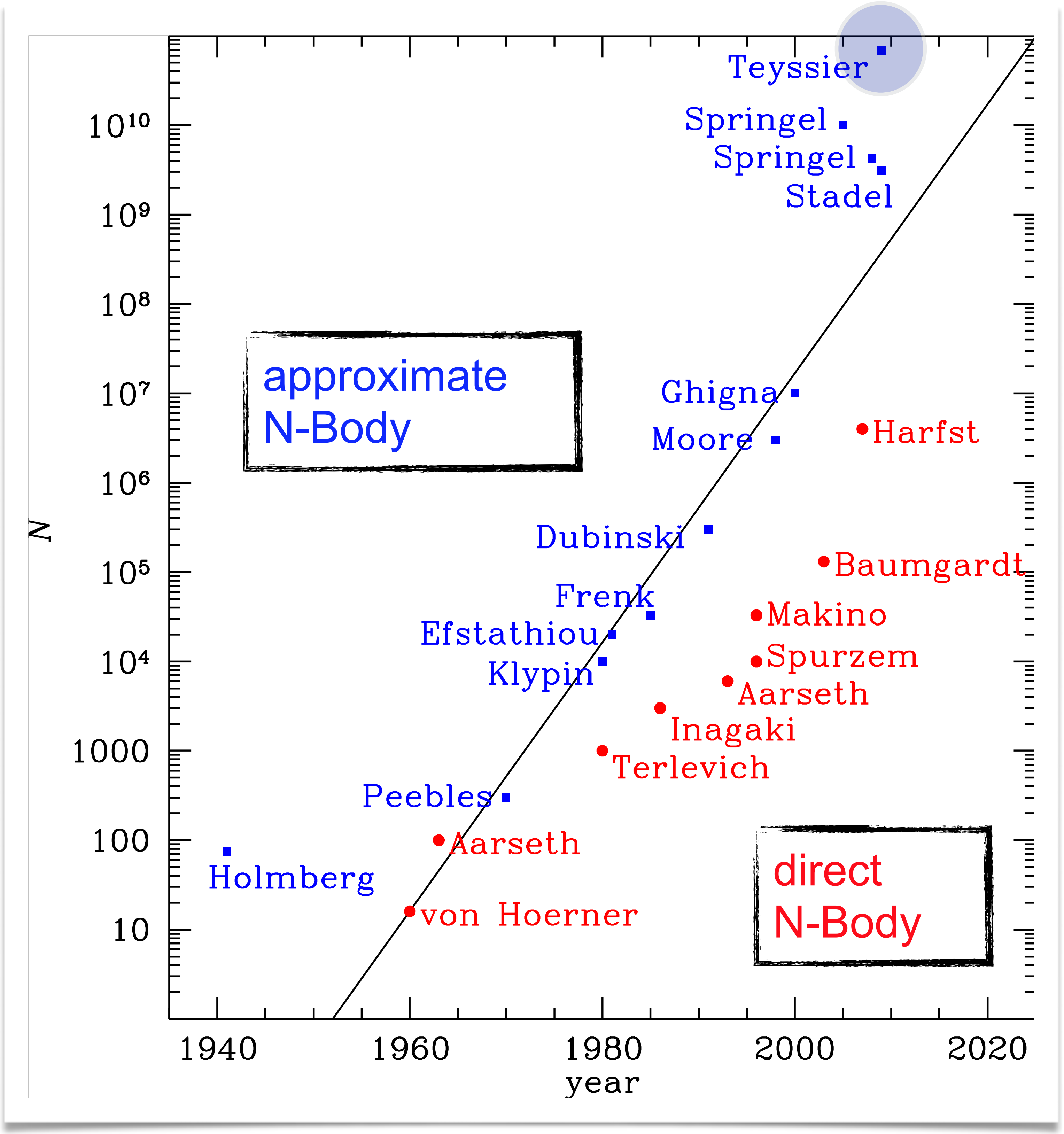


Approximate N-Body - Barnes-Hut Tree



How many particles can we compute?

70 x 10^9



Dehnen & Read 2011

Some N-Body codes you may know

Code	Author	Application	Features
MERCURY	Chambers et al. (2000)	celestial mechanics	symplectic, newer versions from different groups, direct N-Body
NBODY	Aarseth et al. (1985)	star clusters, celestial mechanics	various versions, GPU version, direct N-Body
SyMBA	Duncan et al. (1998)	celestial mechanics	symplectic, direct N-Body
GADGET-2.0	Springel et al. (2005)	galactic dynamics	SPH tree code
AREPO	Springel et al. (2010)	galactic dynamics	moving mesh code
FLASH	Flash consortium	galactic dynamics, accretion discs, star formation	different solvers for gravity: multigrid, tree, …
REBOUND	Rein et al. (2012)	celestial mechanics	😍

and many more…

REBOUND

open source multi-purpose N-body code for collisional dynamics
written by Hanno Rein (University of Toronto at Scarborough)

- features
    - C (iso C99 standard) programming language
    - python frontend
    - modular, use it as an external library
    - OpenMP and MPI (only tree) parallelized
    - various integrators
    - active particles and tracer particles
    - Support for collisional/granular dynamics, various collision detection routines
- **very good documentation**
  https://rebound.readthedocs.io/en/latest
- soon(ish): GPU support

REBOUND

- modular, use it as an external library
  - your nbody simulation is an independent source file which includes rebound.h
  - link your binary to librebound.so
  - run the code

- basic structure

```
#include “rebound.h”
int main(int argc, char* argv[])
{
      struct reb_simulation *r = reb_create_simulation();
      /* choose integrator */
      r->integrator = REB_INTEGRATOR_LEAPFROG;
      /* here you would add some bodies to the simulation */
      /* start integration */
      reb_integrate(r, INFINITY);
}
```

REBOUND

- modular, install via pip (pip install rebound)

- basic structure

```
import rebound

sim = rebound.Simulation()
sim.integrator = “leapfrog”
# here you would add some bodies to the simulation

# now integrate for 100 time units
sim.integrate(100)
```

REBOUND

- different types of particles
  - active particles and testparticles

    set by reb_simulation.testparticle_type

    Type of the particles with an index>=N_active. 0 means particle does not influence any other particle (default), 1 means particles with index < N_active feel testparticles. Testparticles never feel each other.

```
int main(int argc, char* argv[])
{
	struct reb_simulation *r = reb_create_simulation();
	r->N_active = 2;
	(…)
	for (int i = 0; i < 1000; i++) {
		struct reb_particle testparticle = {0};
		testparticle.x = (double) i;
		reb_add(r, testparticle);
	}
	reb_integrate(r, INFINITY);
}
```

REBOUND

- different types of particles
  - active particles and testparticles

    set by sim.testparticle_type

    Type of the particles with an index>=N_active. 0 means particle does not influence any other particle (default), 1 means particles with index < N_active feel testparticles. Testparticles never feel each other.

```
import rebound
sim = rebound.Simulation()

sim.add(m=1)
sim.add(m=0.5, a=1)
# add 1000 testparticles between 2 and 3
a_ini = np.linspace(2, 3, 1000)
for a in a_ini:
    sim.add(m=planetesimal_mass, a=a, f=np.random.rand()*2.*np.pi) # mass is set to 0 by
default, random true anomaly
sim.N_active = 2
# sim.testparticle_type = 0 # default value anyways
```

REBOUND

- each particle can be identified exactly by a particle hash
  - add and remove particles

    ```
    void reb_add(struct reb_simulation *const r, struct reb_particle pt)
    int reb_remove(struct reb_simulation *const r, int index, int keepSorted)
    int reb_remove_by_hash(struct reb_simulation *const r, uint32_t hash, int keepSorted)
    ```
  - identify particles

    ```
    struct reb_particle *reb_get_particle_by_hash(struct reb_simulation *const r, uint32_t hash)
    ```

REBOUND

- each particle can be identified exactly by a particle hash
    - add and remove particles

```
sim.add(m=1)  # Star, index 0
sim.add(m=1e-3, a=52.0) # Jupiter, index 1
sim.add(m=3e-6, a=1.0) # some earth

# remove by index
sim.remove(1) # removes Jupiter, earth is now index 1

sim.add(m=1e-3, a=5.2, hash="Jupiter")
sim.remove("Jupiter")

# test particles
for i in range(1,100):
    sim.add(m=0., a=i, hash=i)
```

- best practice: if you need to track your particles, hash all of them

REBOUND available modules

- integrators
    - REB_INTEGRATOR_IAS15
    - REB_INTEGRATOR_WHFAST
    - REB_INTEGRATOR_JANUS
    - REB_INTEGRATOR_LEAPFROG
    - REB_INTEGRATOR_SEI
    - REB_INTEGRATOR_MERCURIUS
    - REB_INTEGRATOR_HERMES

REBOUND available modules

- integrators

```
#include “rebound.h”
int main(int argc, char* argv[])
{
    struct reb_simulation *r = reb_create_simulation();
    /* choose integrator */
    r->integrator = REB_INTEGRATOR_LEAPFROG;
    r->dt = 1e-3;
}
```

```
import rebound
sim = rebound.Simulation()
sim.integrator = “leapfrog”
sim.dt = 1e-3
```

REBOUND available modules

- integrators
  - `REB_INTEGRATOR_IAS15`
  - `REB_INTEGRATOR_WHFAST`
  - `REB_INTEGRATOR_JANUS`
  - `REB_INTEGRATOR_LEAPFROG`
  - `REB_INTEGRATOR_SEI`
  - `REB_INTEGRATOR_MERCURIUS`
  - `REB_INTEGRATOR_HERMES`

IAS15 stands for Integrator with Adaptive Step-size control, 15th order. It is a vey high order, non-symplectic integrator which can handle arbitrary (velocity dependent) forces and is in most cases accurate down to machine precision, see Rein & Spiegel 2015. default

REBOUND available modules

- integrators
  - `REB_INTEGRATOR_IAS15`
  - `REB_INTEGRATOR_WHFAST`
  - `REB_INTEGRATOR_JANUS`
  - `REB_INTEGRATOR_LEAPFROG`
  - `REB_INTEGRATOR_SEI`
  - `REB_INTEGRATOR_MERCURIUS`
  - `REB_INTEGRATOR_HERMES`

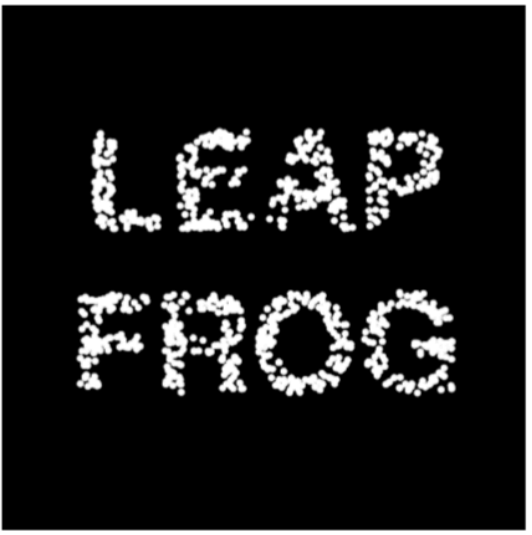
WHFast is the integrator described in Rein & Tamayo 2015, it’s a second order symplectic Wisdom Holman integrator with 11th order symplectic correctors. Good for planetary systems without collisions.

REBOUND available modules

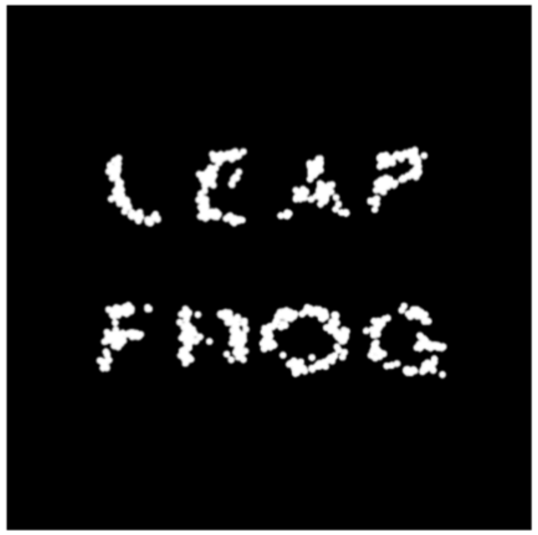
- integrators
  - REB_INTEGRATOR_IAS15
  - REB_INTEGRATOR_WHFAST
  - REB_INTEGRATOR_JANUS
  - REB_INTEGRATOR_LEAPFROG
  - REB_INTEGRATOR_SEI
  - REB_INTEGRATOR_MERCURIUS
  - REB_INTEGRATOR_HERMES

Janus is a bit-wise time-reversible high-order symplectic integrator using a mix of floating point and integer arithmetic, see Rein & Tamayo 2017. JANUS is explicit, formally symplectic and satisfies Liouville's theorem exactly. Its order is even and can be adjusted between two and ten.

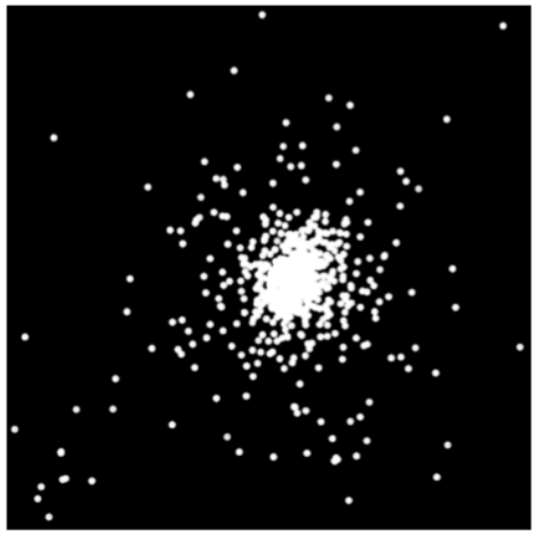
REBOUND available modules



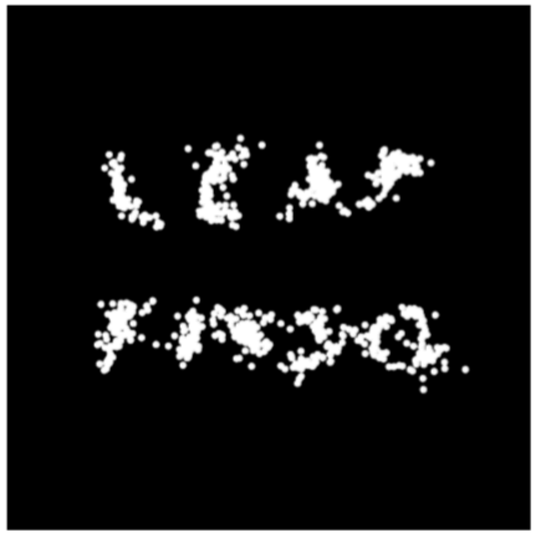
(a) leap-frog, $t = 0$



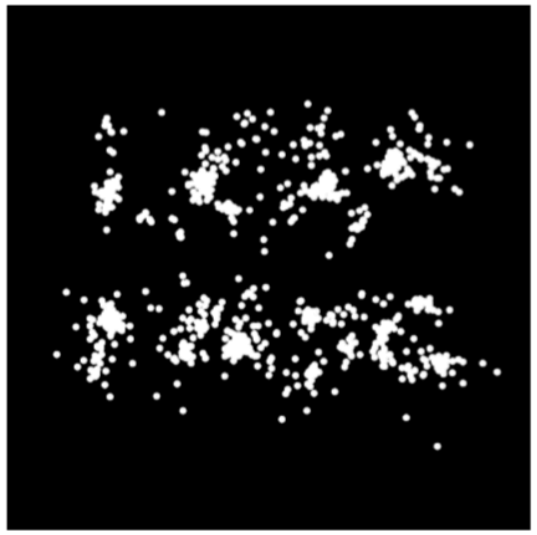
(b) leap-frog, $t = 35$



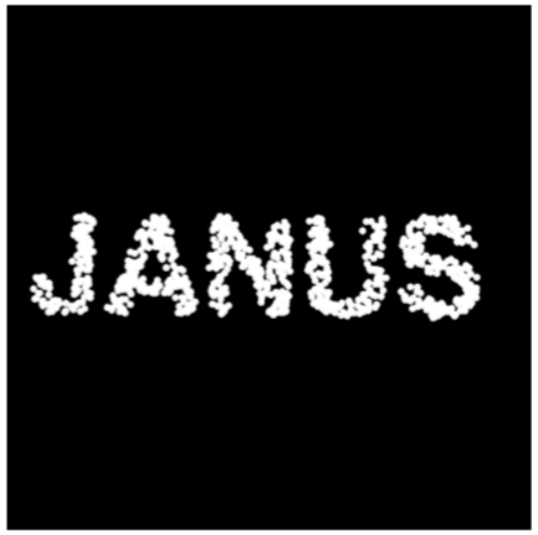
(c) leap-frog, $t = 500$



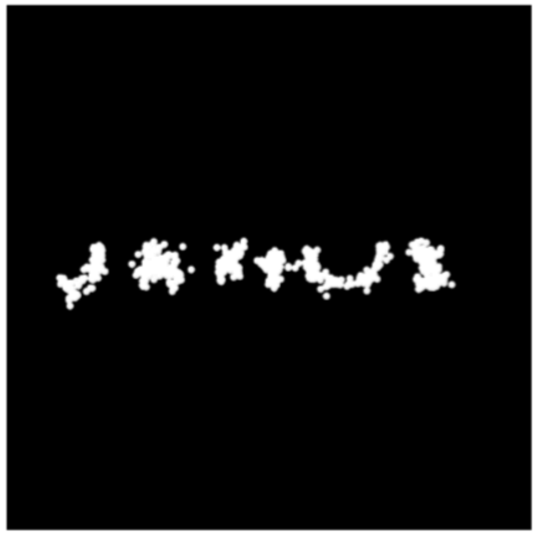
(d) leap-frog, $t = 965$



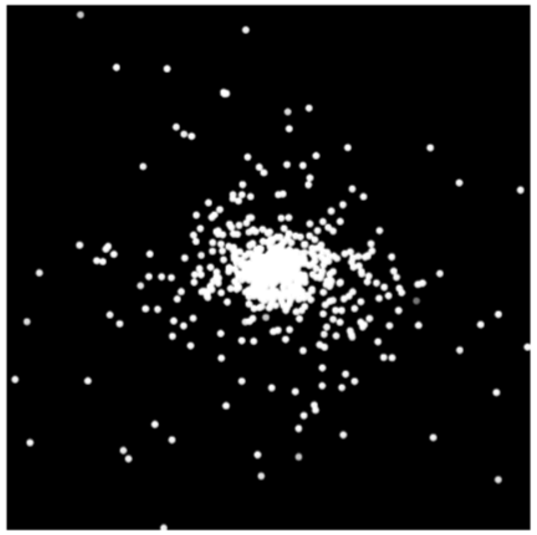
(e) leap-frog, $t = 1000$



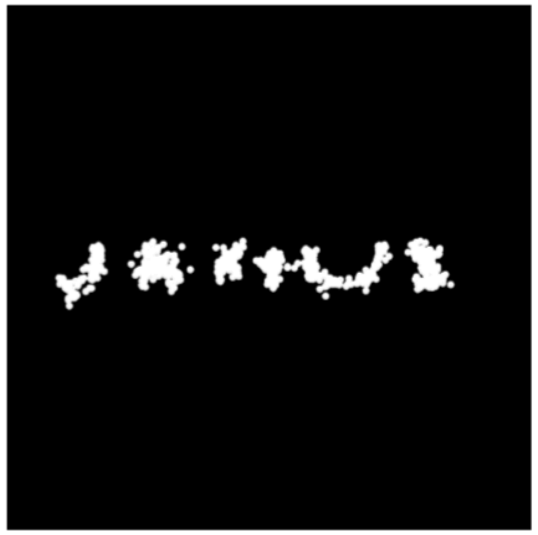
(f) JANUS, $t = 0$



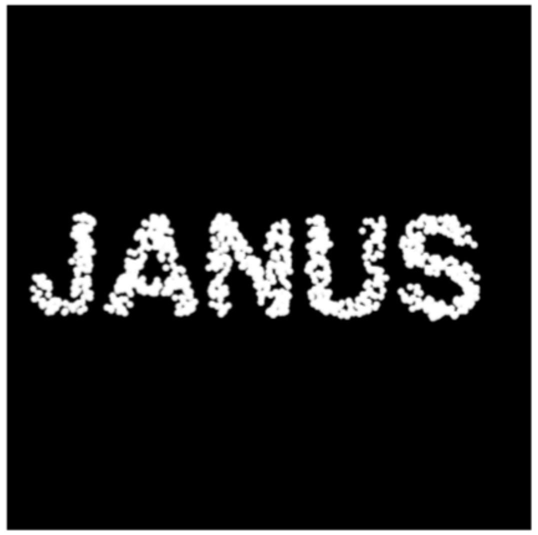
(g) JANUS, $t = 35$



(h) JANUS, $t = 500$



(i) JANUS, $t = 965$



(j) JANUS, $t = 1000$

Rein & Tamayo 2017

REBOUND available modules

- integrators
  - REB_INTEGRATOR_IAS15
  - REB_INTEGRATOR_WHFAST
  - REB_INTEGRATOR_JANUS
  - REB_INTEGRATOR_LEAPFROG
  - REB_INTEGRATOR_SEI
  - REB_INTEGRATOR_MERCURIUS
  - REB_INTEGRATOR_HERMES

Leap frog, second order, symplectic.

REBOUND available modules

- integrators
  - REB_INTEGRATOR_IAS15
  - REB_INTEGRATOR_WHFAST
  - REB_INTEGRATOR_JANUS
  - REB_INTEGRATOR_LEAPFROG
  - REB_INTEGRATOR_SEI
  - REB_INTEGRATOR_MERCURIUS
  - REB_INTEGRATOR_HERMES

A hybrid integrator very similar to the one found in MERCURY. It uses WHFast for long term integrations but switches over smoothly to IAS15 for close encounters.

REBOUND available modules

- Gravity solvers
    - REB_GRAVITY_COMPENSATED
    - REB_GRAVITY_NONE
    - REB_GRAVITY_BASIC
    - REB_GRAVITY_TREE

```
#include "rebound.h"
int main(int argc, char* argv[])
{
    struct reb_simulation *r = reb_create_simulation();
    r->gravity = REB_GRAVITY_BASIC;
}
```

```
import rebound
sim = rebound.Simulation()
sim.gravity = "basic"
```

REBOUND available modules

- Gravity solvers
  - REB_GRAVITY_COMPENSATED
  - REB_GRAVITY_NONE
  - REB_GRAVITY_BASIC
  - REB_GRAVITY_TREE

Direct summation with compensated summation, $O(N^2)$, default

On your computer

$$\sum_{i=1}^{10000} \frac{1}{i^2} \neq \sum_{i=10000}^{1} \frac{1}{i^2}$$

1.6448340718480518
1.6448340718480596

REBOUND available modules

- Gravity solvers
  - REB_GRAVITY_COMPENSATED
  - REB_GRAVITY_NONE
  - REB_GRAVITY_BASIC
  - REB_GRAVITY_TREE

No self-gravity.

REBOUND available modules

- Gravity solvers
  - REB_GRAVITY_COMPENSATED
  - REB_GRAVITY_NONE
  - REB_GRAVITY_BASIC
  - REB_GRAVITY_TREE

Direct summation, $O(N^2)$.

REBOUND available modules

- Gravity solvers
  - REB_GRAVITY_COMPENSATED
  - REB_GRAVITY_NONE
  - REB_GRAVITY_BASIC
  - REB_GRAVITY_TREE

Oct tree, Barnes & Hut 1986, O(N log(N).

REBOUND available modules

- Gravity solvers
  - REB_GRAVITY_COMPENSATED
  - REB_GRAVITY_NONE
  - REB_GRAVITY_BASIC
  - REB_GRAVITY_TREE

```
#include “rebound.h”
int main(int argc, char* argv[])
{
      struct reb_simulation *r = reb_create_simulation();
      r->gravity = REB_GRAVITY_BASIC;
}
```

```
import rebound
sim = rebound.Simulation()
sim.gravity = “basic”
```

REBOUND available modules

- Boundary conditions
  - `REB_BOUNDARY_NONE`
  - `REB_BOUNDARY_OPEN`
  - `REB_BOUNDARY_PERIODIC`
  - `REB_BOUNDARY_SHEAR`

REBOUND available modules

- Boundary conditions
    - `REB_BOUNDARY_NONE`
    - `REB_BOUNDARY_OPEN`
    - `REB_BOUNDARY_PERIODIC`
    - `REB_BOUNDARY_SHEAR`

Particles are not affected by boundary conditions, default.

REBOUND available modules

- Boundary conditions
  - `REB_BOUNDARY_NONE`
  - `REB_BOUNDARY_OPEN`
  - `REB_BOUNDARY_PERIODIC`
  - `REB_BOUNDARY_SHEAR`

Particles are removed from the simulation if they leave the box.

REBOUND available modules

- Boundary conditions
  - REB_BOUNDARY_NONE
  - REB_BOUNDARY_OPEN
  - REB_BOUNDARY_PERIODIC
  - REB_BOUNDARY_SHEAR

Periodic boundary conditions. Particles are reinserted on the other side if they cross the box boundaries.

REBOUND available modules

- Boundary conditions
  - REB_BOUNDARY_NONE
  - REB_BOUNDARY_OPEN
  - REB_BOUNDARY_PERIODIC
  - REB_BOUNDARY_SHEAR

Shear periodic boundary conditions. Similar to periodic boundary conditions, but ghost-boxes are moving with constant speed, set by the shear. see `example/shearing_sheet`.

REBOUND available modules

- Boundary conditions
  - REB_BOUNDARY_NONE
  - REB_BOUNDARY_OPEN
  - REB_BOUNDARY_PERIODIC
  - REB_BOUNDARY_SHEAR

Choose via r->boundary

```
int main(int argc, char* argv[])
{
    struct reb_simulation *r = reb_create_simulation();
    /* choose open boundaries */
    r->boundary     = REB_BOUNDARY_OPEN;
    /* define a box for the open boundary, one box with edge length 10 */
    reb_configure_box(r, 10., 1, 1, 1)
    reb_integrate(r, INFINITY);
}
```

REBOUND available modules

- Boundary conditions
  - REB_BOUNDARY_NONE
  - REB_BOUNDARY_OPEN
  - REB_BOUNDARY_PERIODIC
  - REB_BOUNDARY_SHEAR

Choose via sim.boundary

```
import rebound

sim.boundary = “none" # the default

sim.configure_box(100.)   # confine the simulation to a box of size 100
sim.boundary = “open”  # simple outflow, particles are removed
```

REBOUND available modules

- collision detection, assign radius to particles, choose between
  - `REB_COLLISION_NONE`
  - `REB_COLLISION_DIRECT`
  - `REB_COLLISION_LINE`
  - `REB_COLLISION_TREE`

REBOUND available modules

- collision detection
  - REB_COLLISION_NONE
  - REB_COLLISION_DIRECT
  - REB_COLLISION_LINE
  - REB_COLLISION_TREE

no collisions detection. default.

REBOUND available modules

- collision detection
  - `REB_COLLISION_NONE`
  - `REB_COLLISION_DIRECT`
  - `REB_COLLISION_LINE`
  - `REB_COLLISION_TREE`

brute force collision search, $O(N^2)$, checks for instantaneous overlaps only.

REBOUND available modules

- collision detection
  - REB_COLLISION_NONE
  - REB_COLLISION_DIRECT
  - REB_COLLISION_LINE
  - REB_COLLISION_TREE

brute force collision search, $O(N^2)$, checks for overlaps that occured during the last timestep assuming particles travelled along straight lines.

REBOUND available modules

- collision detection
  - REB_COLLISION_NONE
  - REB_COLLISION_DIRECT
  - REB_COLLISION_LINE
  - REB_COLLISION_TREE

uses the Oct tree for particle overlapping, O(N log(N)).

REBOUND available modules

- collision detection
  - REB_COLLISION_NONE
  - REB_COLLISION_DIRECT
  - REB_COLLISION_LINE
  - REB_COLLISION_TREE

rebound tracks all collisions and the user can choose between different kinds of collisional outcome:

fully elastic
inelastic
merging (conserves mass, momentum and volume)
user-defined function

REBOUND available modules

- collision detection
    - REB_COLLISION_NONE
    - REB_COLLISION_DIRECT
    - REB_COLLISION_LINE
    - REB_COLLISION_TREE

```
int main(int argc, char* argv[])
{
    struct reb_simulation *r = reb_create_simulation();
    r->collision = REB_COLLISION_DIRECT;
    r->collision_resolve = reb_collision_resolve_merge;
    // or use your own function here
    // see https://rebound.readthedocs.io/en/latest/c_examples/
    closeencounter_record/
    // r->collision_resolve = <your_function_pointer_goes_here>;
    reb_integrate(r, INFINITY);
}
```

REBOUND available modules

- collision detection
    - REB_COLLISION_NONE
    - REB_COLLISION_DIRECT
    - REB_COLLISION_LINE
    - REB_COLLISION_TREE

```
import rebound

sim.collision = "direct"
sim.collision_resolve = “merge”
#sim.collision_resolve = <your_function_goes_here> # see exercises…

# you need to add a collision radius to the particles now
sim.add(m=1.)
sim.add(m=1e-3, a=1., r=np.sqrt(1e-3/3.))
sim.add(m=5e-3, a=1.25, r=1.25*np.sqrt(5e-3/3.))
```

REBOUND functions

- some useful functions

REBOUND functions

- heartbeat function, set via `reb_simulation.heartbeat` pointer
    - is called after each time step

```
int main(int argc, char* argv[])
{
     struct reb_simulation *r = reb_create_simulation();
     /* set heartbeat function */
     r->heartbeat = heartbeat;
     (…)
     reb_integrate(r, INFINITY);
}


void heartbeat(struct reb_simulation* r) {
    if (reb_output_check(r, 1000)) { // <- checks for interval of 1000 time units
        fprintf(stdout, “current time is %e\n", r->t);
    }
}
```

REBOUND functions

- use a simple array in python for the same functionality

```
import rebound

sim = rebound.Simulation()

simulationtime = 100*2*np.pi
Nsteps = 100
time = np.linspace(0, simulationtime, Nsteps)
x = np.zeros(N)
y = np.zeros(N)
for i, t in enumerate(times):
    print(t/simulationtime, end="\r")
    sim.integrate(t, exact_finish_time=1)
```

REBOUND functions

- functions for i/o
  - void reb_output_ascii(struct reb_simulation *r, char *filename)
    Append the positions and velocities of all particles to an ASCII file.
  - void reb_output_orbits(struct reb_simulation *r, char *filename)
    Append an ASCII file with orbital paramters of all particles.

```
void heartbeat(struct reb_simulation* r) {
    if (reb_output_check(r, 1000)) { // <- checks for interval of 1000 time units
        reb_output_orbits(r, “cool_orbital_data.txt”);
    }
}
```

REBOUND functions

- i/o for particles is easier in python using numpy….

```
import rebound

sim = rebound.Simulation()
simulationtime = 100*2*np.pi
Nsteps = 100
time = np.linspace(0, simulationtime, Nsteps)
# (…)  add N particles here….
x = np.zeros(N)
y = np.zeros(N)
for i, t in enumerate(times):
    print(t/simulationtime, end="\r")
    sim.integrate(t, exact_finish_time=1)
    for j, p in enumerate(sim.particles):
        x[j] = p.x
        y[j] = p.y
    np.savetxt("orbit"+ ("%05d" % i) + ".txt", np.transpose([x, y]))
```

REBOUND functions

- but it is also possible to save and load a complete simulation

```
import rebound

sim = rebound.Simulation()
simulationtime = 100*2*np.pi
Nsteps = 100
time = np.linspace(0, simulationtime, Nsteps)
# (…)  add N particles here…. and integrate
# save current status
sim.save(“mysim.bin”)
```

```
import rebound

# load old simulation
sim = rebound.Simulation(“mysim.bin”)
print(sim.status())
```

REBOUND functions

- many more functions
    - calculate energy
    - calculate angular momentum
    - convert between coordinate system variables and more celestial mechanics related functions
    - print information about time and timestep
    - print information about energy
    - create initial particle distributions, e.g., plummer spheres
    - …

see the documentation

https://rebound.readthedocs.io/en/latest/

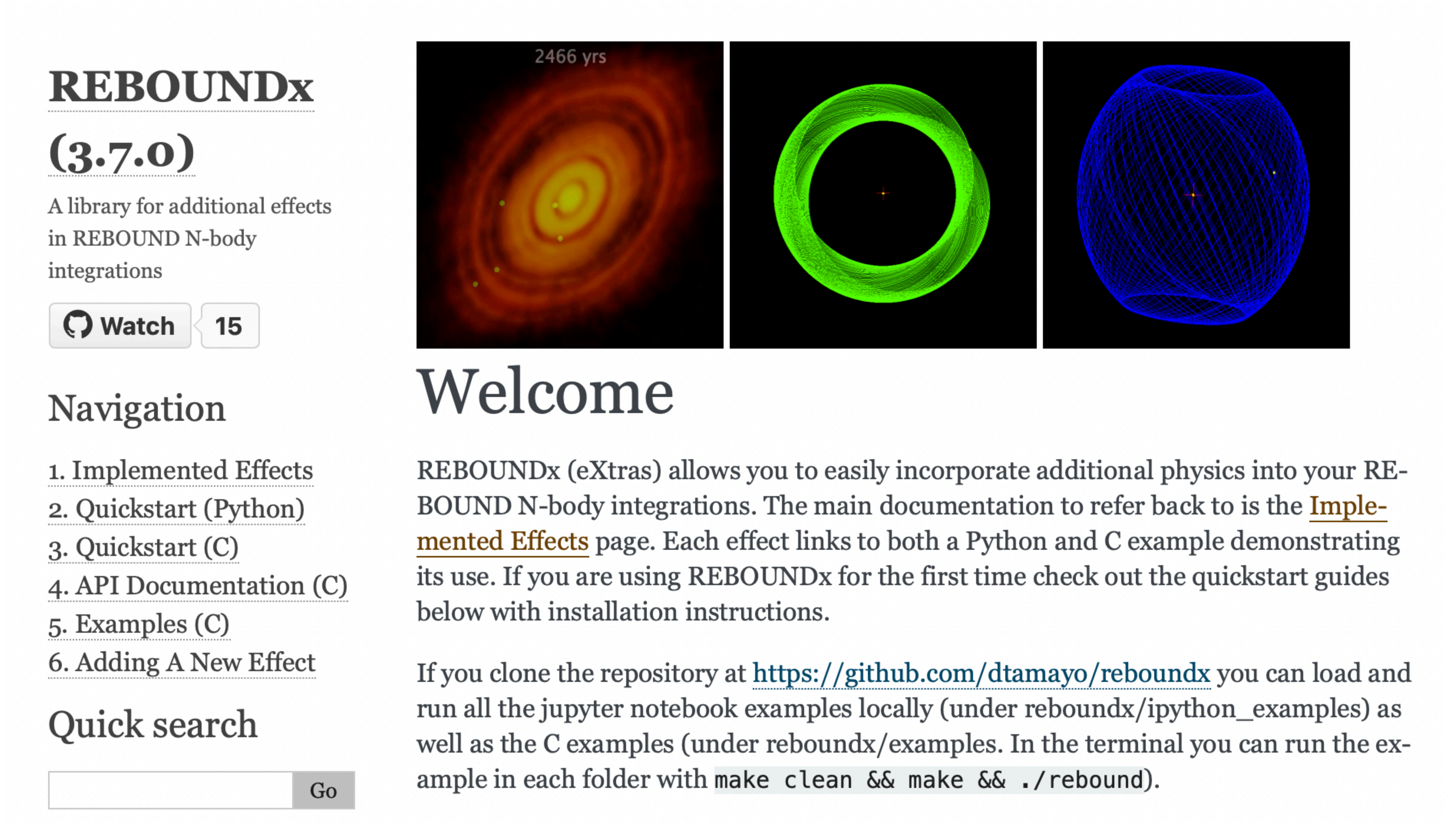
REBOUND adding more physics

- hook function for additional forces (examples/prdrag.c)

```
int main(int argc, char* argv[])
{
        struct reb_simulation *r = reb_create_simulation()
        r->force_is_velocity_dependent     = 1;
        r->additional_forces          = radiation_forces;
        (…)
        reb_integrate(r, INFINITY);
}
void radiation_forces(struct reb_simulation* r){
    struct reb_particle* particles = r->particles;
    const int N = r->N;
    for (int i=0;i<N;i++){
        const struct reb_particle p = particles[i];            // cache
        if (p.m!=0.) continue;                           // only dust particles feel radiation forces
        (…)
        // Equation (5) of Burns, Lamy, Soter (1979), Poynting-Robertson effect
        particles[i].ax += F_r*((1.-rdot/c)*prx/pr - prvx/c);
        particles[i].ay += F_r*((1.-rdot/c)*pry/pr - prvy/c);
        particles[i].az += F_r*((1.-rdot/c)*prz/pr - prvz/c);
    }
}
```

REBOUNDx adds more physics

- before adding effects, take a look at REBOUNDx



REBOUNDx (3.7.0)

A library for additional effects in REBOUND N-body integrations

Navigation

1. Implemented Effects
2. Quickstart (Python)
3. Quickstart (C)
4. API Documentation (C)
5. Examples (C)
6. Adding A New Effect

Quick search

Welcome

REBOUNDx (eXtras) allows you to easily incorporate additional physics into your REBOUND N-body integrations. The main documentation to refer back to is the Implemented Effects page. Each effect links to both a Python and C example demonstrating its use. If you are using REBOUNDx for the first time check out the quickstart guides below with installation instructions.

If you clone the repository at https://github.com/dtamayo/reboundx you can load and run all the jupyter notebook examples locally (under reboundx/ipython_examples) as well as the C examples (under reboundx/examples. In the terminal you can run the example in each folder with `make clean && make && ./rebound`).

REBOUNDx adds more physics

- implemented effects
  - orbit modifications (e.g., planet migration…)
  - General Relativity (e.g., first-order post-newtonian effects from all bodies in the system)
  - radiation forces (radiation pressure and Poynting-Robertson drag, Yarkovsky effect)
  - mass modifications (growth, loss)
  - gravitational harmonics (add azimuthally symmetric gravitational harmonics (J2, J4))
  - (…)

REBOUNDx adds more physics

- built for addon module for REBOUND

```
import rebound
import reboundx

sim = rebound.Simulation()
# (…) setup your rebound simulation

# activate reboundx for your sim
rebx = reboundx.Extras(sim)
# load the effect you want, in this case orbital modifications, see examples
mof = rebx.load_force("modify_orbits_forces")
rebx.add_force(mof)
# (…) set some parameters depending on the effect
# integrate
sim.integrate(1000)
```

REBOUND exercises

👩‍💻👨‍💻

http://www.tat.physik.uni-tuebingen.de/~schaefer/rebound_exercises.pdf

- Hands-on exercises part
  - Two-Body problem
  - Saturn’s rings stability
  - Stability of planetary system
  - Kirkwood gaps
  - Migrating planet
  - <your project comes here> ….

Thank you.

Contact:

Christoph Schäfer
Computational Physics
Auf der Morgenstelle 10
72076 Tübingen

ch.schaefer@uni-tuebingen.de
www.tat.physik.uni-tuebingen.de/~schaefer