

N-body methods

(*) for
the form:

$$-\sum_{\substack{j=1 \\ i \neq j}}^N \frac{\tilde{m}_j (\tilde{\mathbf{r}}_j - \tilde{\mathbf{r}}_i)}{(\tilde{r}_{ij}^2 + \epsilon^2)^{3/2}}$$

where

$$\tilde{r}_{ij}^2 = (x_i - x_j)^2$$

$\tilde{\mathbf{r}}_i$

ϵ is

We start discussion of numerical techniques with a very simple case: forces are estimated by summing up all contributions from all particles and with every particle moving with the same time-step. The computational cost is dominated by the force calculations that scale as N^2 , where N is the number of particles in the simulation. Because of the steep scaling, the computational cost of a simulation starts to be prohibitively too large for $N \gtrsim 10^6$. However, simulations with a few hundred thousand particles are fast, and there are numerous interesting cases that can be addressed with $N < 10^6$ particles. Examples include major-mergers of dark matter halos, collisions of two elliptical galaxies, and tidal stripping and destruction of a dwarf spheroidal satellite galaxy moving in the potential of the Milky Way galaxy. In these cases it is convenient to use proper, not comoving coordinates.


The problem that we try to solve numerically is the following. For given coordinates \mathbf{r}_{init} and velocities \mathbf{v}_{init} of N massive particles at moment $t = t_{\text{init}}$ find their velocities \mathbf{v} and coordinates \mathbf{r} at the next moment $t = t_{\text{next}}$ assuming that the particles interact only through the Newtonian force of gravity. If \mathbf{r}_i and m_i are the coordinates and masses of the particles, then the equations of motion are:

$$\frac{d^2 \mathbf{r}_i}{dt^2} = -G \sum_{j=1, i \neq j}^N \frac{m_j (\mathbf{r}_i - \mathbf{r}_j)}{|\mathbf{r}_i - \mathbf{r}_j|^3}, \quad (8)$$

where G is the gravitational constant. Two steps should be taken before we start solving equations (8) numerically.

$$\mathcal{W} = -\frac{1}{2} \sum_{i,j}^N \frac{G m_i m_j}{|\vec{r}_i - \vec{r}_j|}$$

$$\mathcal{K} = -\frac{1}{2} \sum_i^N m_i \left(\frac{d\vec{r}_i}{dt} \right)^2$$



First, we introduce force softening: we make the force weaker (“softer”) at small distances to avoid very large accelerations, when two particles collide or come very close to each other. This makes numerical integration schemes stable. Another reason for softening the force at small distances is that in cosmological environments, when one deals with galaxies, clusters of galaxies, or the large-scale structure, effects of close collisions between individual particles are very small and can be neglected. In other words, the force acting on a particle is dominated by the cumulative contribution of all particles, not by a few close individual companions.

There are different ways of introducing the force softening. For mesh-based codes, the softening is defined by the size of cell elements. For TREE codes the softening is introduced by assuming a particular kernel, and it is different for different implementations. The simplest and often used method is called the Plummer softening. It replaces the distance between particles $\Delta r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ in eq. (8) with the expression $(\Delta r_{ij}^2 + \epsilon^2)^{1/2}$, where ϵ is the softening parameter.

Second, we need to introduce new variables to avoid dealing with too large or too small physical units of a real problem. This can be done in a number of ways. For mesh-based codes, the size of the largest resolution element and the Hubble velocity across the element give scales of distance and velocity. Here we use more traditional scalings. Suppose M and R are scales of mass and distances. These can be defined by a particular physical problem. For example, for simulations of an isolated galaxy M and R can be the total mass and the initial radius. It really does not matter what M and R are. The scale of time t_0 is chosen as $t_0 = (GM/R^3)^{-1/2}$. Using M , R , and t_0 we can change the physical variables \mathbf{r}_i , \mathbf{v}_i , m_i into dimensionless variables using the following relations:

$$\mathbf{r}_i = \tilde{\mathbf{r}}_i \mathbf{R}, \quad \mathbf{v}_i = \tilde{\mathbf{v}}_i \frac{\mathbf{R}}{t_0}, \quad m_i = \tilde{m}_i M, \quad t = \tilde{t} t_0. \quad (9)$$

We now change the variables in eq. (8) and use the Plummer softening:

$$\tilde{\mathbf{g}}_i = - \sum_{j=1}^N \frac{\tilde{m}_j (\tilde{\mathbf{r}}_i - \tilde{\mathbf{r}}_j)}{(\Delta \tilde{r}_{ij}^2 + \tilde{\epsilon}^2)^{3/2}}, \quad \frac{d\tilde{\mathbf{v}}_i}{d\tilde{t}} = \tilde{\mathbf{g}}_i, \quad \frac{d\tilde{\mathbf{r}}_i}{d\tilde{t}} = \tilde{\mathbf{v}}_i, \quad (10)$$

All numerical algorithms for solving these equations include three steps, which are repeated many times:

- find acceleration $g(r)$
- update velocity $v = v + \Delta v(g)$
- update coordinates $r = r + \Delta r(v)$

Here is a simple fragment of a Fortran-90 code that does it using direct summation of accelerations:

```
Program Simple
.... (set parameters)
.... (read data)
Do                               ! Main loop of integration
  Call Acceleration ! find acceleration for every particle
  v = v+g*dt         ! update velocities
  X = X+v*dt         ! update coordinates
  t = t +dt         ! update time
  If(t> t_end)exit ! stop when final time is reached
End do
end Program Simple

Subroutine Acceleration ! find accelerations for each particle
  g = 0.             ! set acceleration to zero for all particles
  Do i=1,N          ! for each particle i
  Do j=1,N          ! add contributions of other particles
    g(:,i)=g(:,i)+m(j)*(X(:,j)-X(:,i))/ &
      sqrt(SUM((X(:,j)-X(:,i))**2+eps2)**3)
  EndDo
  EndDo
end Subroutine Acceleration
```

In this code we extensively use Fortran-90 feature of vector operations. For example, the statement $V = V + g \times dt$ means “do it for every element” of arrays $V(i, j)$ and $g(i, j)$. There are simple ways of speeding up the code. Particles can be assigned into groups according to their accelerations with each group having their own time step. In this case particles with large accelerations update their coordinates and accelerations more often while particles in low density (and acceleration) regions move with large time step, thus reducing the cost of their treatment. Calculations of the acceleration can be easily parallelized using OpenMp directives. These optimizations can speed up the code by hundreds of times making it a useful tool for simple simulations.

4. Moving particles: Time-stepping algorithms

Numerical integration of equations of motion are relatively simple as compared with the other part of the N -body problem – the force calculations. Still, a wrong choice of parameters or an integrator can make a substantial impact on the accuracy of the final solution and on cpu time. To make arguments more transparent, we write equations of motion in proper coordinates and assume that the gravitational acceleration can be estimated for every particle. In this case the equations of motion for each particle are simply:

$$\frac{d\mathbf{v}(\mathbf{t})}{dt} = \mathbf{g}(\mathbf{x}), \quad \frac{d\mathbf{x}(\mathbf{t})}{dt} = \mathbf{v}(\mathbf{t}). \quad (11)$$

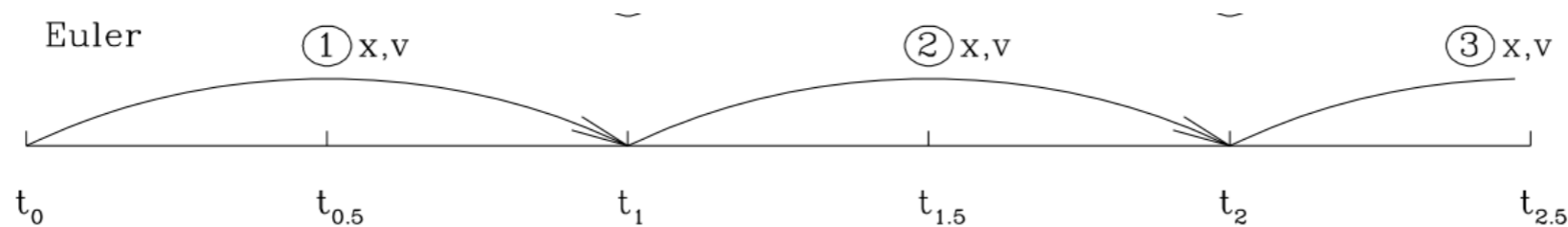
Along particle trajectory acceleration can be considered as a function of time $\mathbf{g}(\mathbf{x}(\mathbf{t}))$. If we know coordinates \mathbf{x}_0 and velocities \mathbf{v}_0 at some initial moment t_0 , then eqs. (11) can be integrated from $t = t_0$ to $t_1 = t_0 + dt$:

$$\mathbf{x}_1 = \mathbf{x}_0 + \int_{t_0}^{t_1} \mathbf{v}(\mathbf{t})d\mathbf{t}, \quad \mathbf{v}_1 = \mathbf{v}_0 + \int_{t_0}^{t_1} \mathbf{g}(\mathbf{t})d\mathbf{t}. \quad (12)$$

We now expand $\mathbf{v}(\mathbf{t})$ and $\mathbf{g}(\mathbf{t})$ in the Taylor series around t_0 and substitute those into eqs. (12) to obtain different approximations for \mathbf{x}_1 and \mathbf{v}_1 . If we keep only the first two terms, we get the first order Euler approximation: $\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{v}_0 dt + \epsilon$, where $\epsilon \approx g_0 dt^2/2 \propto O(dt^2)$ and $\mathbf{v}_1 = \mathbf{v}_0 + \mathbf{g}_0 dt + \epsilon$, $\epsilon \propto O(dt^2)$. Accuracy and convergence of the Euler integrator are low, and it is never used for real simulations. One may think that adding $g_0 dt^2/2$ term to displacements may increase the accuracy, but it really does not because velocities are still of the first order. In the next iteration the first-order velocity makes the displacement also of the first order. However, we may dramatically improve the accuracy by re-arranging terms in the Taylor expansion in order to kill some high order terms.

$$\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{v}_0 dt$$

$$\mathbf{v}_1 = \mathbf{v}_0 + \mathbf{g}_0 dt$$



Suppose initial velocity is given not at the moment t_0 , but a half timestep earlier at $t_{-1/2} = t_0 - dt/2$. Using coordinates at t_0 we find acceleration $\mathbf{g}_0(t_0)$. We now advance velocity one step forward from $t_{-1/2}$ to $t_{1/2} = t_{-1/2} + dt$. Note that when we do it, we use acceleration at the middle of the time step, not on the left boundary of the time step as in the Euler integrator. We then advance coordinates to moment $t_1 = t_0 + dt$ using the new value of velocity. As the result, the scheme of integration is:

$$\mathbf{v}_{1/2} = \mathbf{v}_{-1/2} + \mathbf{g}_0 dt, \quad \mathbf{x}_1 = \mathbf{x}_0 + \mathbf{v}_{1/2} dt. \quad (13)$$

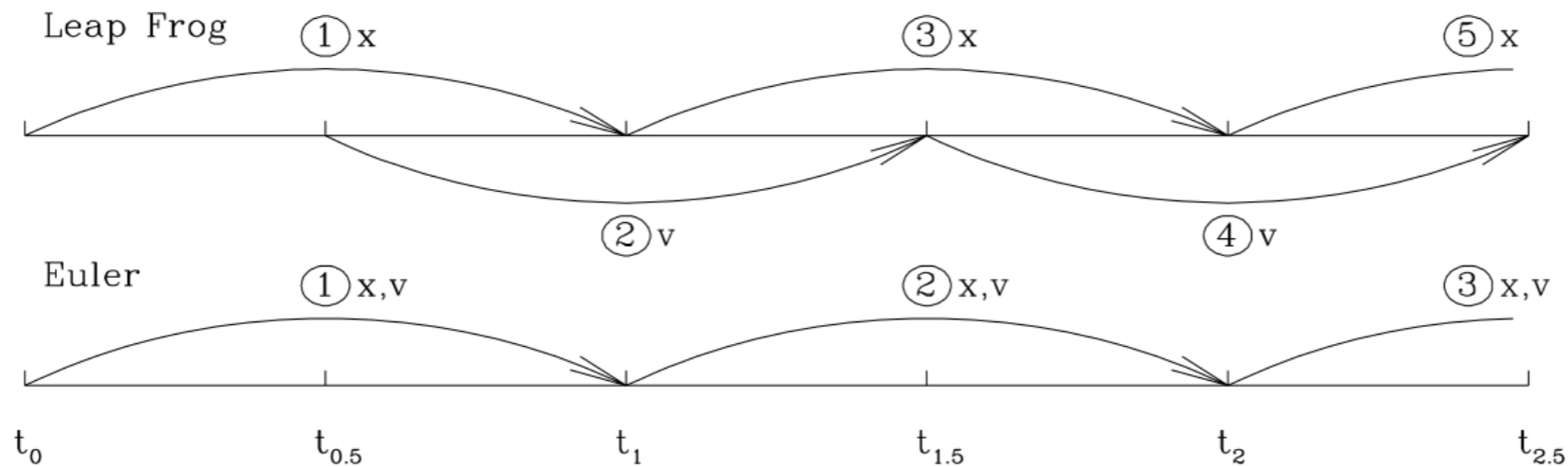
In order to find the accuracy of this approximation, we first eliminate velocities from eqs. (13): $\mathbf{x}_1 - 2\mathbf{x}_0 + \mathbf{x}_{-1} = \mathbf{g}_0 dt^2$. There is an error in this integrator, which we can find by using the Taylor expansion for $\mathbf{x}_{\pm 1}$ up to the fourth order term. This gives:

$$\mathbf{x}_1 - 2\mathbf{x}_0 + \mathbf{x}_{-1} = \mathbf{g}_0 dt^2 + \epsilon, \quad (14)$$

where the error of the approximation is

$$\epsilon = \frac{1}{12} \frac{d^2 g}{dt^2} dt^4. \quad (15)$$

Here the second time derivative of the acceleration is estimated at $t = t_0$. This is a dramatic improvement as compared with the Euler integrator: the error is proportional to dt^4 and, as a bonus, there is a small factor $1/12$.



Kick and Drift

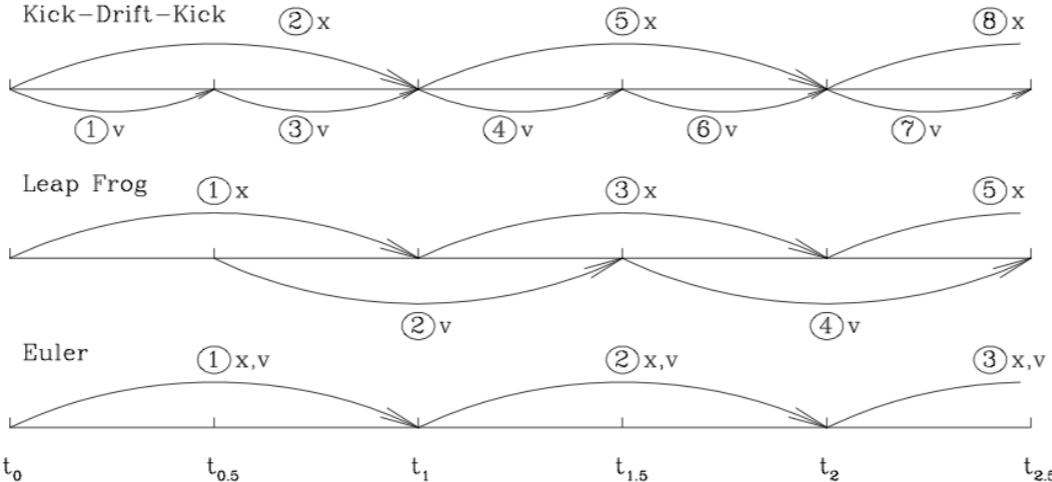
One disadvantage of the leap-frog is that velocities and coordinates are defined at different moments of time. It is convenient to split the integrator into smaller steps that allow for synchronization of time moments and are also easier to modify when the time-step changes. An algorithm of integration of trajectories can be written as a sequence of operators, which advance particle positions (called drifts) and change velocities (called kicks). Let $K(dt)$ be an operator (kick) that advances velocities by time dt . Applying the operator simply means $K(dt) : \mathbf{v} = \mathbf{v} + \mathbf{g}dt$. Similarly, the drift operator is $D(dt) : \mathbf{x} = \mathbf{x} + \mathbf{v}dt$. We also need to specify the moment when the gravitational acceleration is calculated and the moment when the decision is made to change the time-step. So, we use G and S operators to indicate those two moments. For example, a simple constant-step leap-frog integrator can be written as sequence of $GK(dt)D(dt)GK(dt)D(dt)\dots$

Using the K and D operators we can also write the leap-frog integrator which starts with \mathbf{x} and \mathbf{v} defined at the same moment of time and ends at $t + dt$ moment:

$$KDK : K(dt/2)D(dt)GK(dt/2)S. \tag{16}$$

New accelerations are estimated after advancing coordinates, and the change in the time-step dt is made at the end of each time-step. The sequence of actions for the KDK integrator is illustrated in the top panel of Figure 1.

Changing the time-step may be necessary when particles experience a vast range of accelerations, which is typically the case in high-resolution cosmological simulations. However, changing the time-step results in breaking symmetries and reducing the accuracy of the leap-frog integrator. It becomes not time reversible and it loses its ability to preserve the energy. There are some ways to restore these properties, but they are complicated and never used in cosmology.



Equations of motion of particle in force field $F(\mathbf{r})$:

$$\dot{\mathbf{r}} = \mathbf{v} \quad ; \quad \dot{\mathbf{v}} = \mathbf{F}(\mathbf{r})$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n + h\mathbf{v}_n \quad ; \quad \mathbf{v}_{n+1} = \mathbf{v}_n + h\mathbf{F}(\mathbf{r}_n)$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n + h\mathbf{v}_n \quad ; \quad \mathbf{v}_{n+1} = \mathbf{v}_n + h\mathbf{F}(\mathbf{r}_{n+1})$$

$$\mathbf{r}' = \mathbf{r}_n + \frac{1}{2}h\mathbf{v}_n \quad ; \quad \mathbf{v}_{n+1} = \mathbf{v}_n + h\mathbf{F}(\mathbf{r}') \quad ; \quad \mathbf{r}_{n+1} = \mathbf{r}' + \frac{1}{2}h\mathbf{v}_{n+1}$$

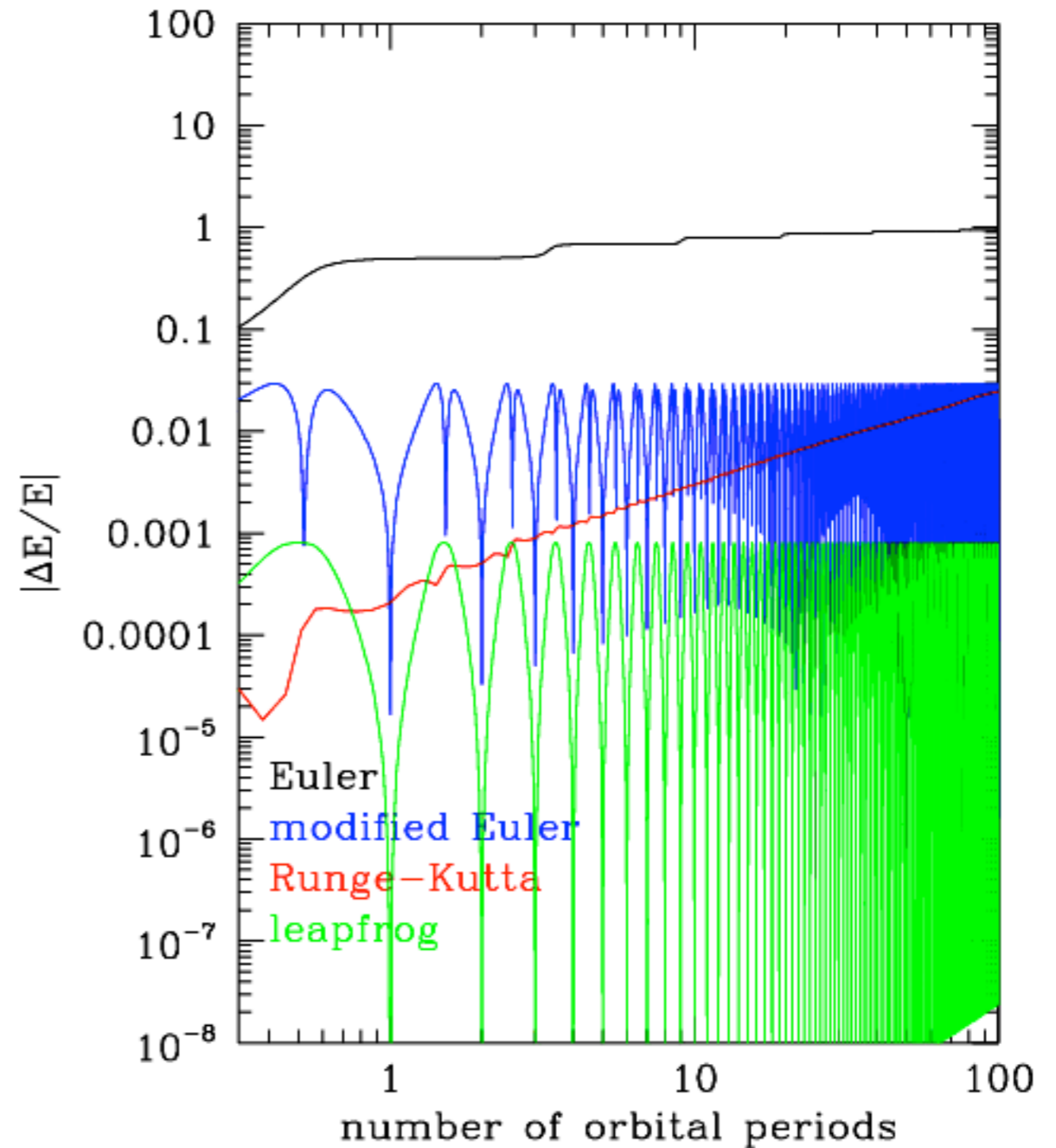
$$h = dt$$

1. Euler's method
2. modified Euler's
3. leapfrog

Consider following a particle in the force field of a point mass.
 Set $G=M=1$ for simplicity. Equations of motion read

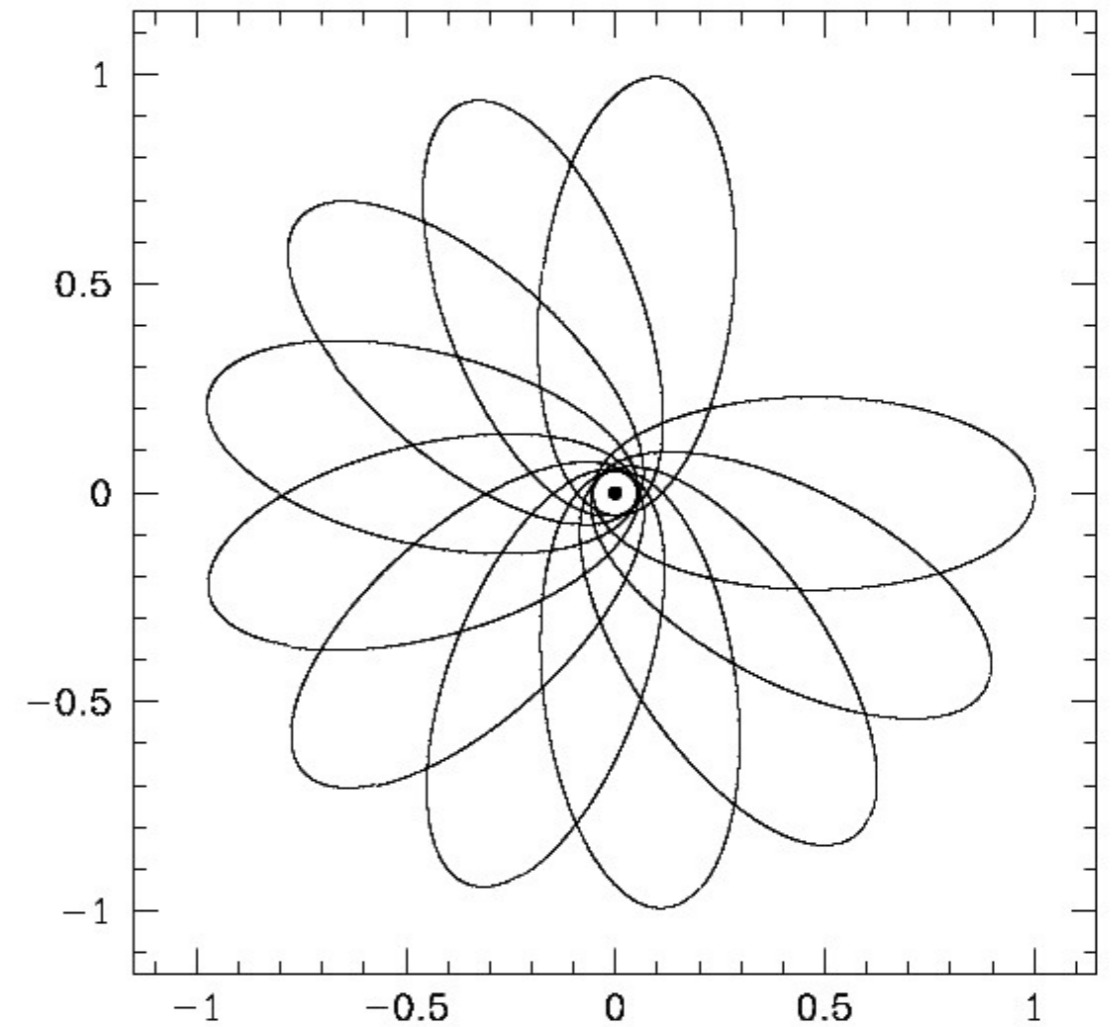
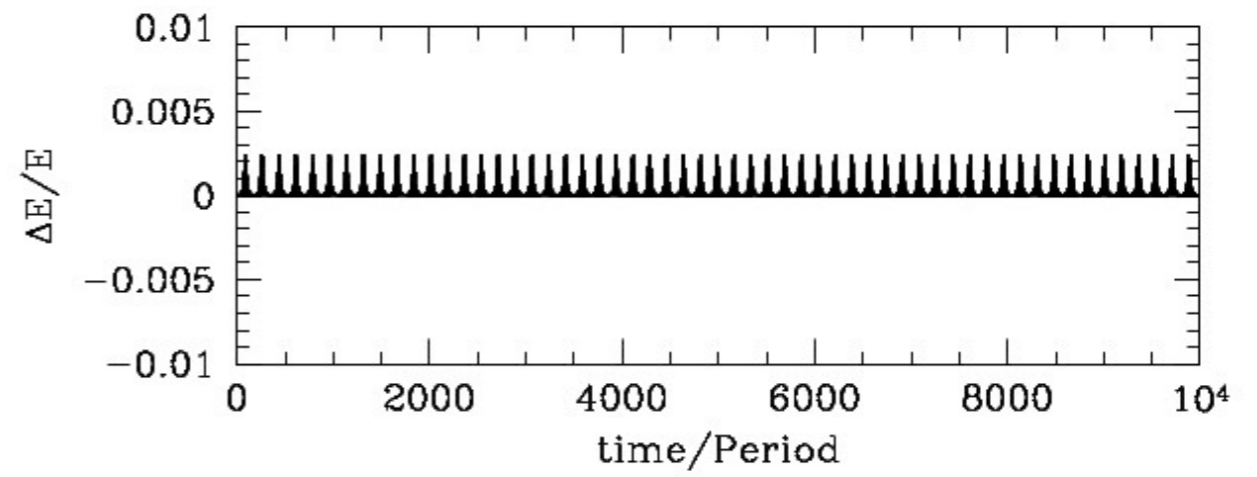
$$\dot{\mathbf{r}} = \mathbf{v} \quad ; \quad \dot{\mathbf{v}} = \mathbf{F}(\mathbf{r}) = -\frac{\hat{\mathbf{r}}}{r^2}$$

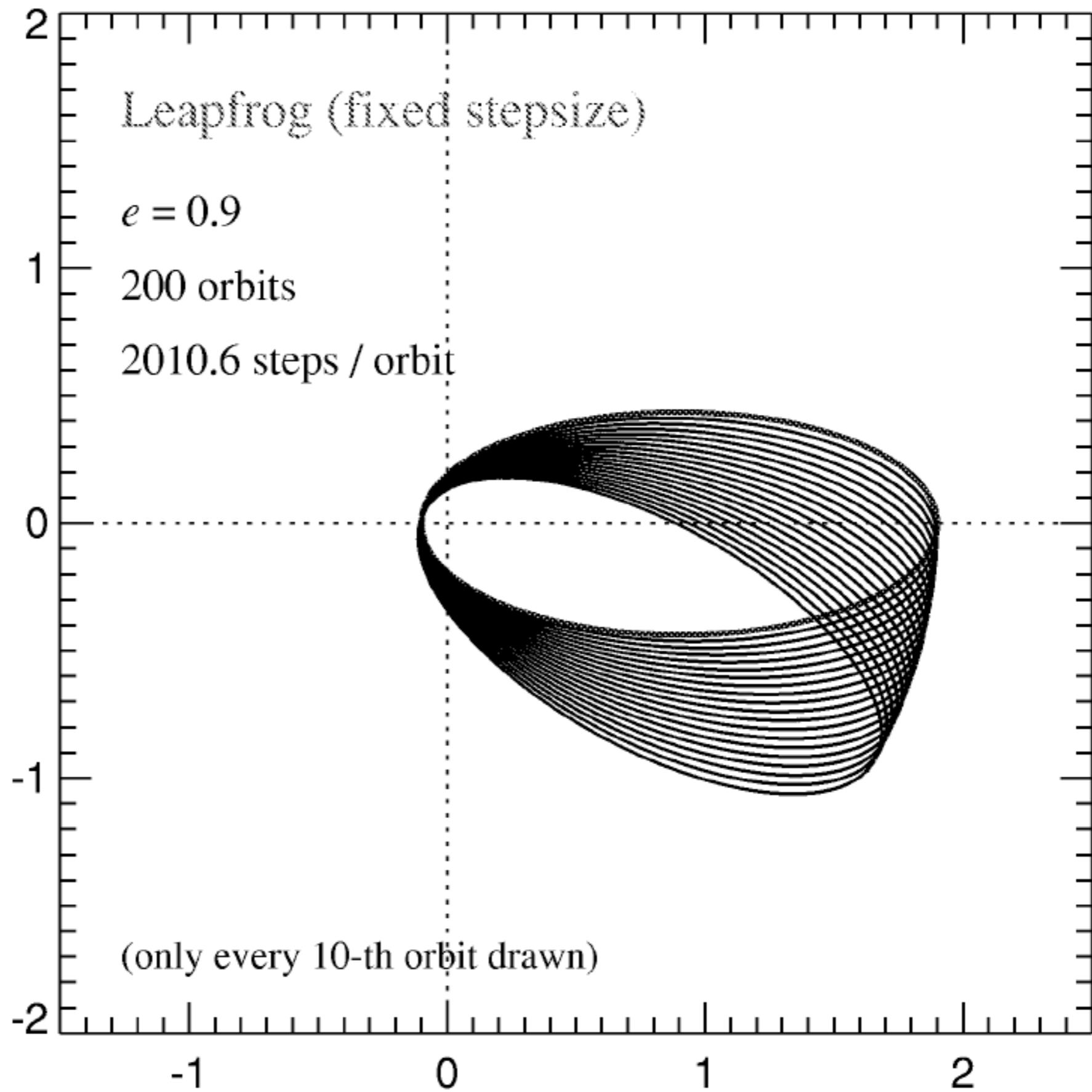
eccentricity = 0.2
 200 steps per orbit
 plot shows fractional energy error $|\Delta E/E|$

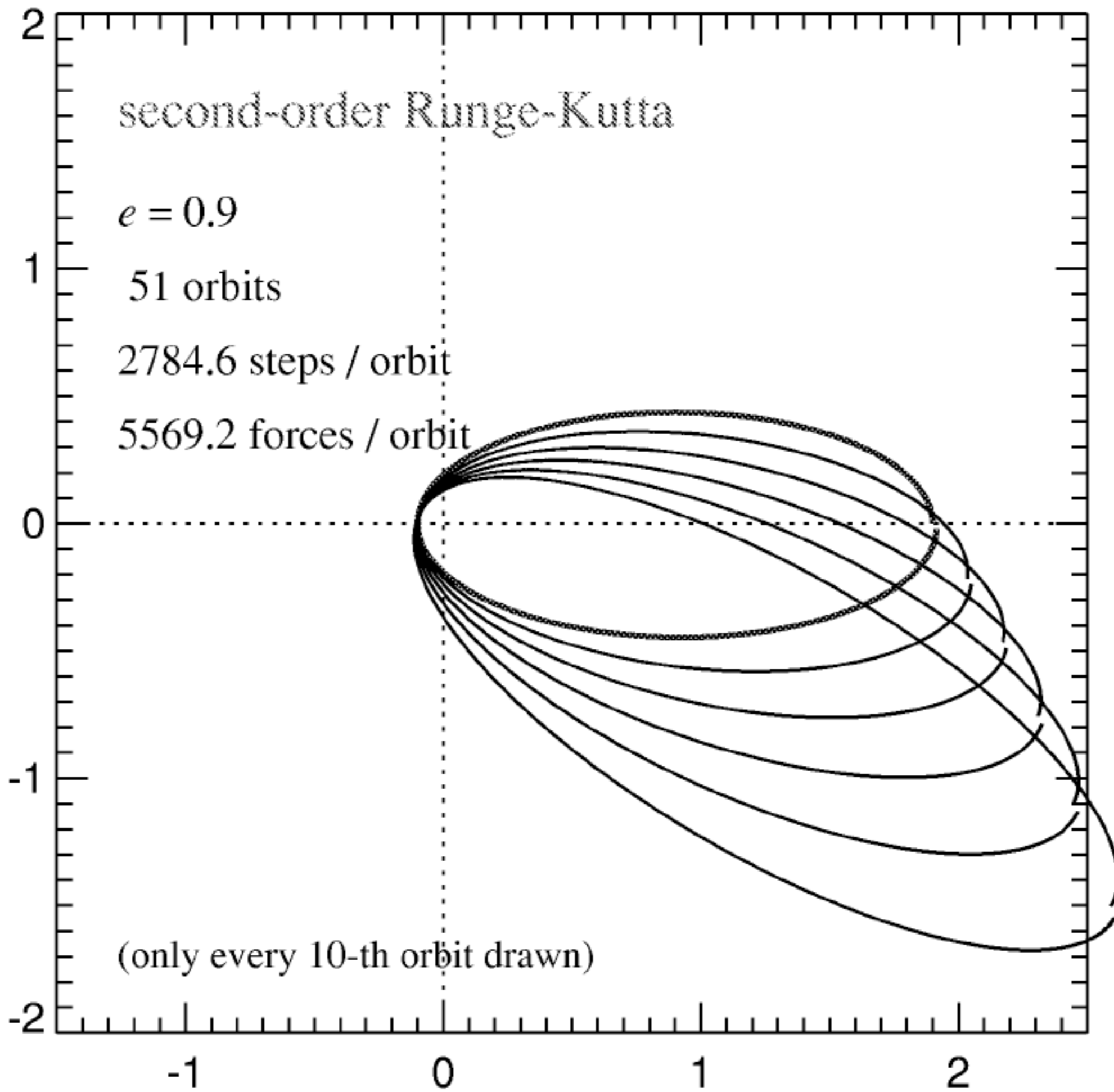


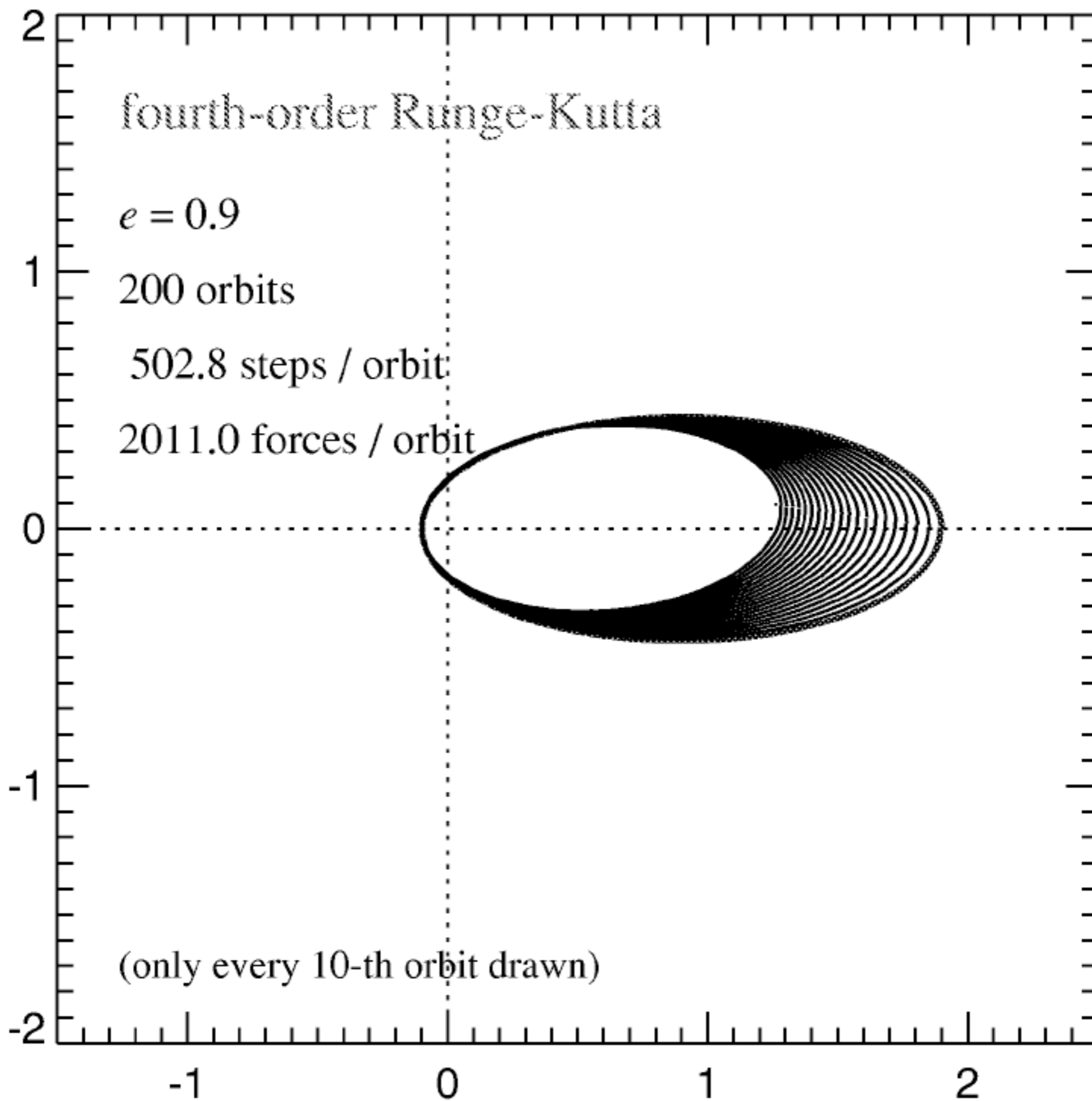
Energy conservation (top panel) and drift of orbit orientation (bottom panel) for integration with constant-step leapfrog scheme. Initial orbit is shown with the full curve.

1500 accelerations/period.









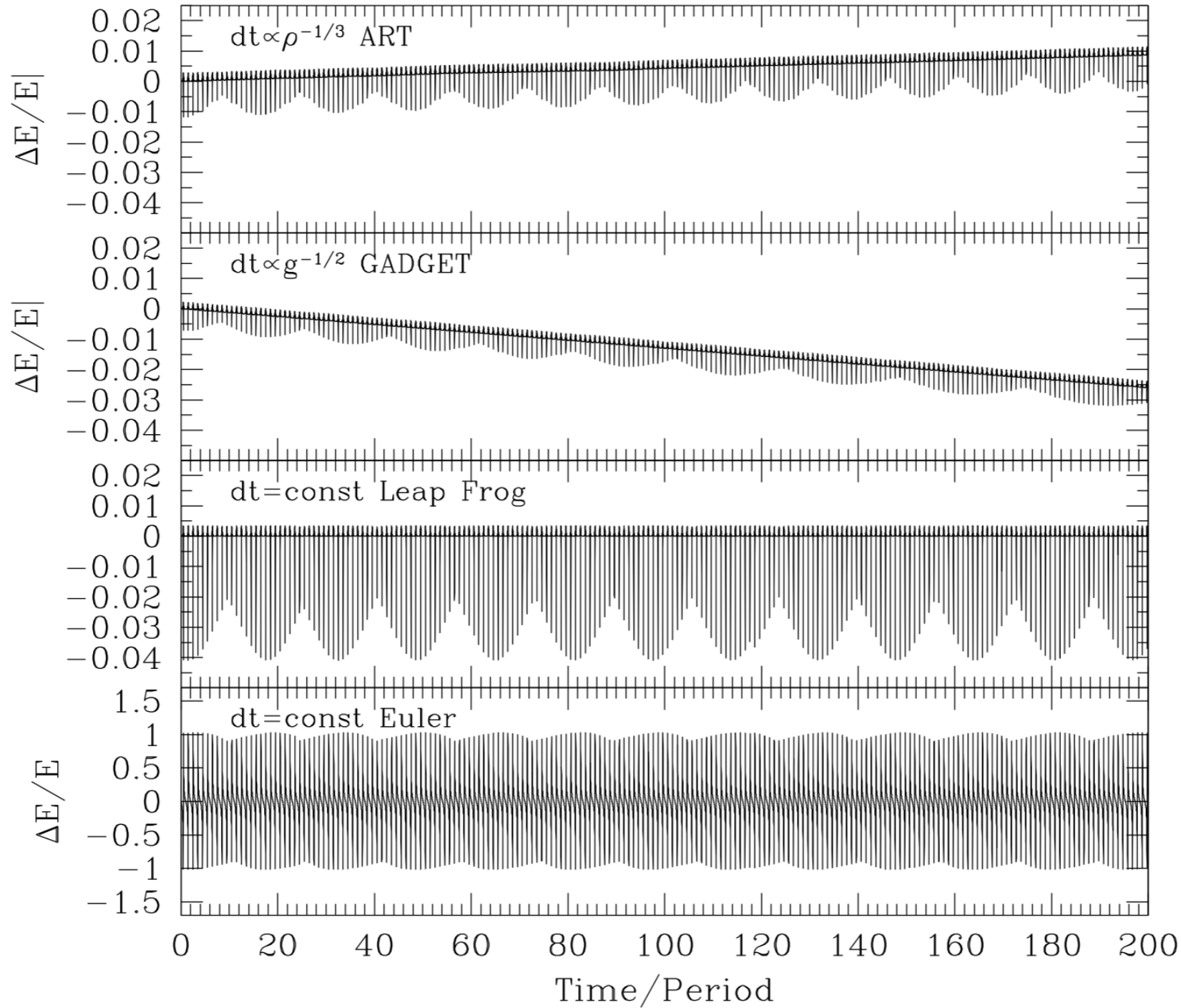
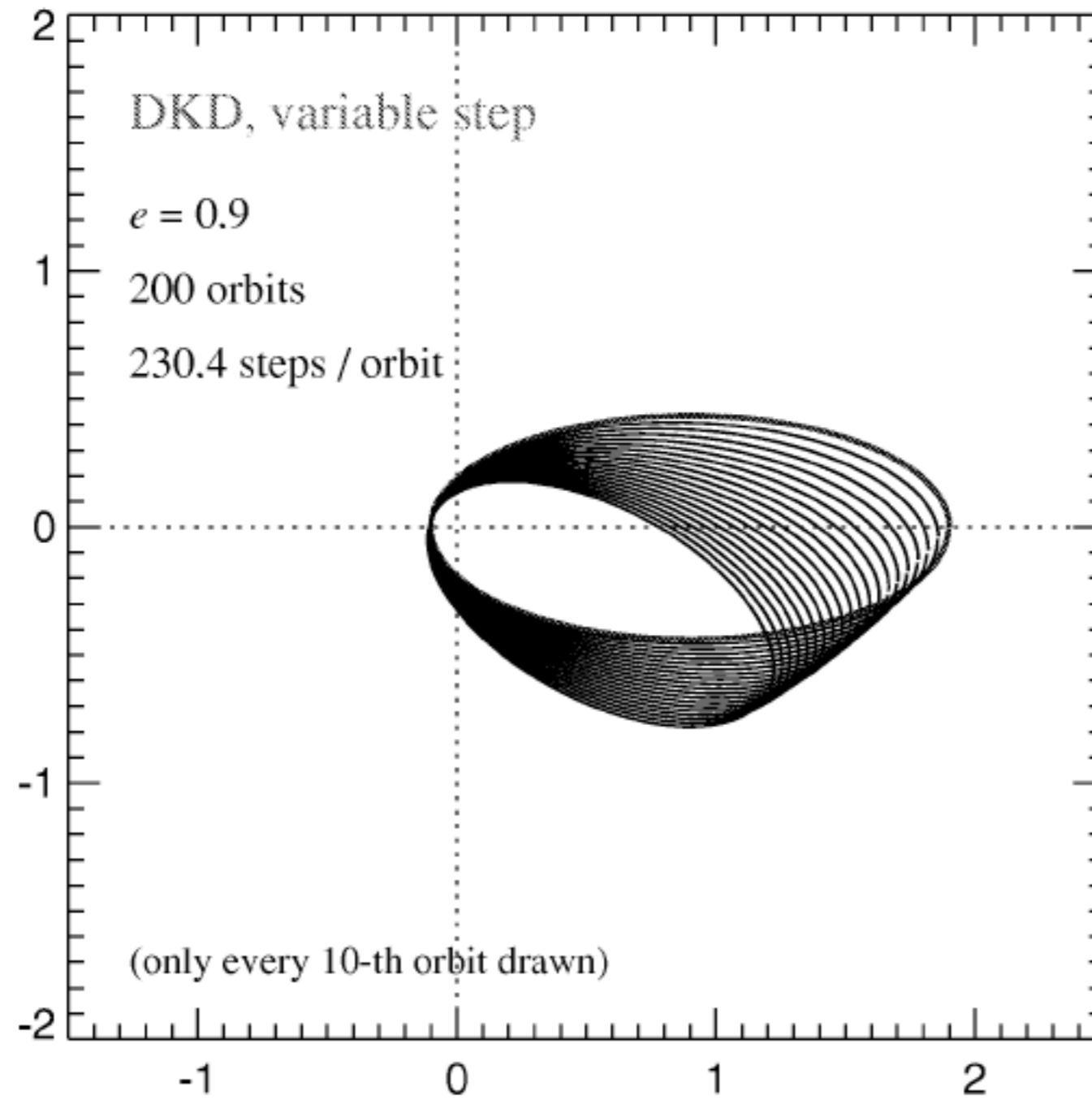
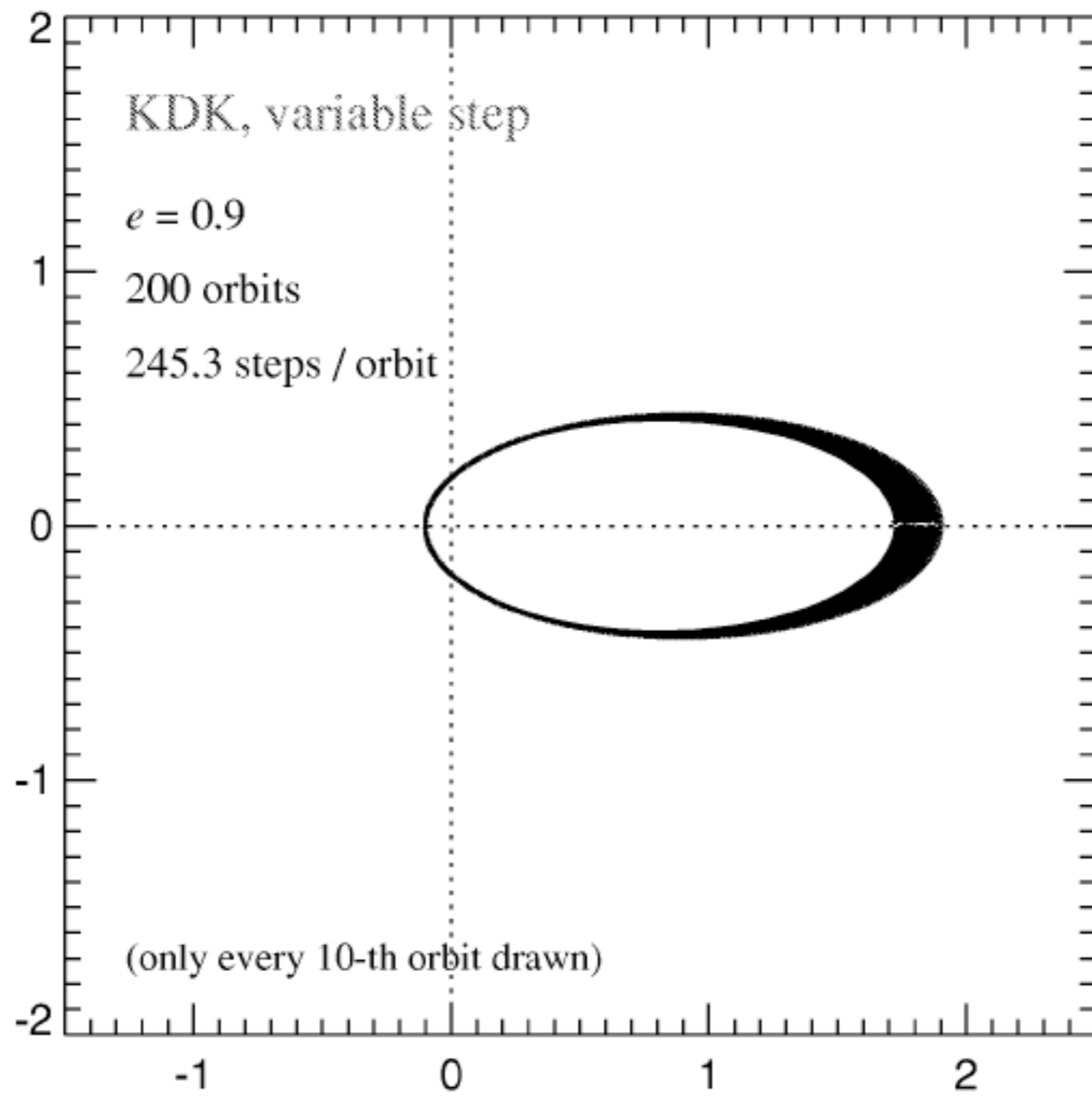


Fig. 2. Accuracy of energy conservation for a particle orbiting the center of isothermal density profile $\rho \propto r^{-2}$ in an eccentric orbit with apo- to pericenter ratio 10:1. Trajectories were followed with different integrators, each integrator using 500 time-steps per orbital period. The Euler scheme gives the worst accuracy (note the change in the y-axis). The leap-frog with a constant time-step shows no long-term energy drift, but errors are large as compared with codes with variable time-steps. Errors are smaller for variable time-step integrators, but they also show a linear trend with time.





Different types of N-body codes

Finding grav. acceleration is the most time consuming part of any large N-body code. This is why codes are classified by methods used to find accelerations:

1) Direct summation $\sim N^2$

2) Solve the Poisson equation on a mesh:

$$\nabla^2 \sigma = 4\pi G \rho$$

a) need to design mesh

b) find density on the mesh

c) solve Poisson equation

d) differentiate σ to get \vec{g}

3) Tree codes:

a) generate a hierarchical mesh

b) multipole expansion for mass distribution inside each cell

c) sum contribution using only large-enough cells

The simplest code is called Particle-Mesh (PM):

a constant-size cubic mesh is overlaid on computational volume.

- (1) Calculate density in each cell.
- (2) Solve the Poisson equation using FFT.
- (3) Numerically differentiate grav.potential to find acceleration for each particle.
- (4) Move particles by one small time-step. Repeat the procedure.

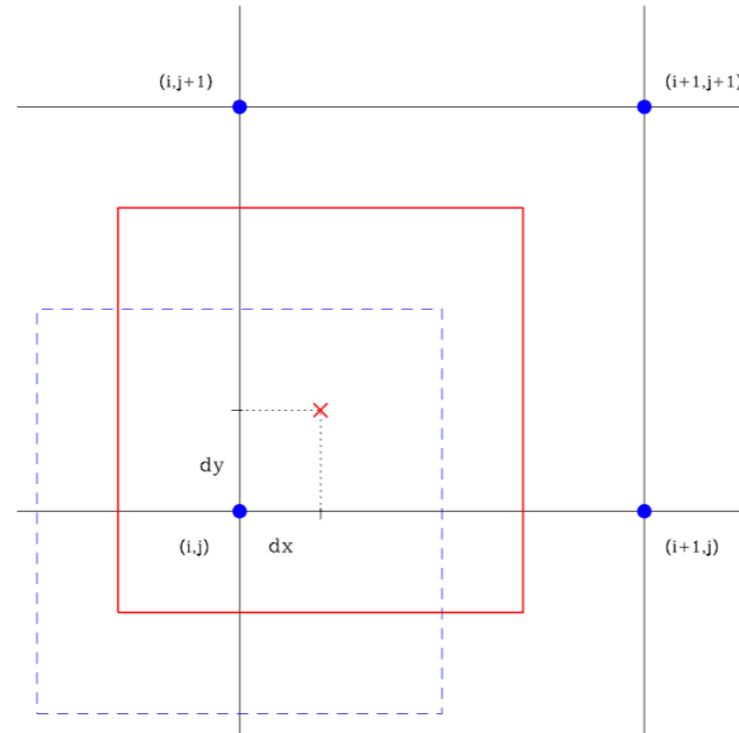


Fig. 4. Example of the Cloud-In-Cell density assignment in two dimensions. Centers of mesh cells are shown with large blue circles. Blue dashed square presents boundaries of the cell with coordinates (i, j) . Particle center shown with red cross has coordinates (dx, dy) and its boundaries are shown as red box. Area of intersection of the red and blue boxes is the mass that the particle contributes to the cell (i, j) . All four cells indicated in the plot receive a contribution from the particle.

particle to density ρ are:

$$\begin{cases} \rho_{i,j,k} & = \rho_{i,j,k} & + (1 - dx)(1 - dy)(1 - dz) \\ \rho_{i+1,j,k} & = \rho_{i+1,j,k} & + dx(1 - dy)(1 - dz) \\ & \dots & \\ \rho_{i+1,j+1,k+1} & = \rho_{i+1,j+1,k+1} & + dx dy dz \end{cases} \quad (20)$$

Poisson equation:

$$\nabla^2 \phi = 4\pi G \rho$$

Expand density and potential into Fourier series:

$$\phi(\mathbf{x}) = \sum \tilde{\phi}_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{x}}, \quad \rho(\mathbf{x}) = \sum \tilde{\rho}_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{x}}$$

Find Fourier components of the potential: $\tilde{\phi}_{\mathbf{k}} = -4\pi G \tilde{\rho}_{\mathbf{k}} |\mathbf{k}|^{-2}$

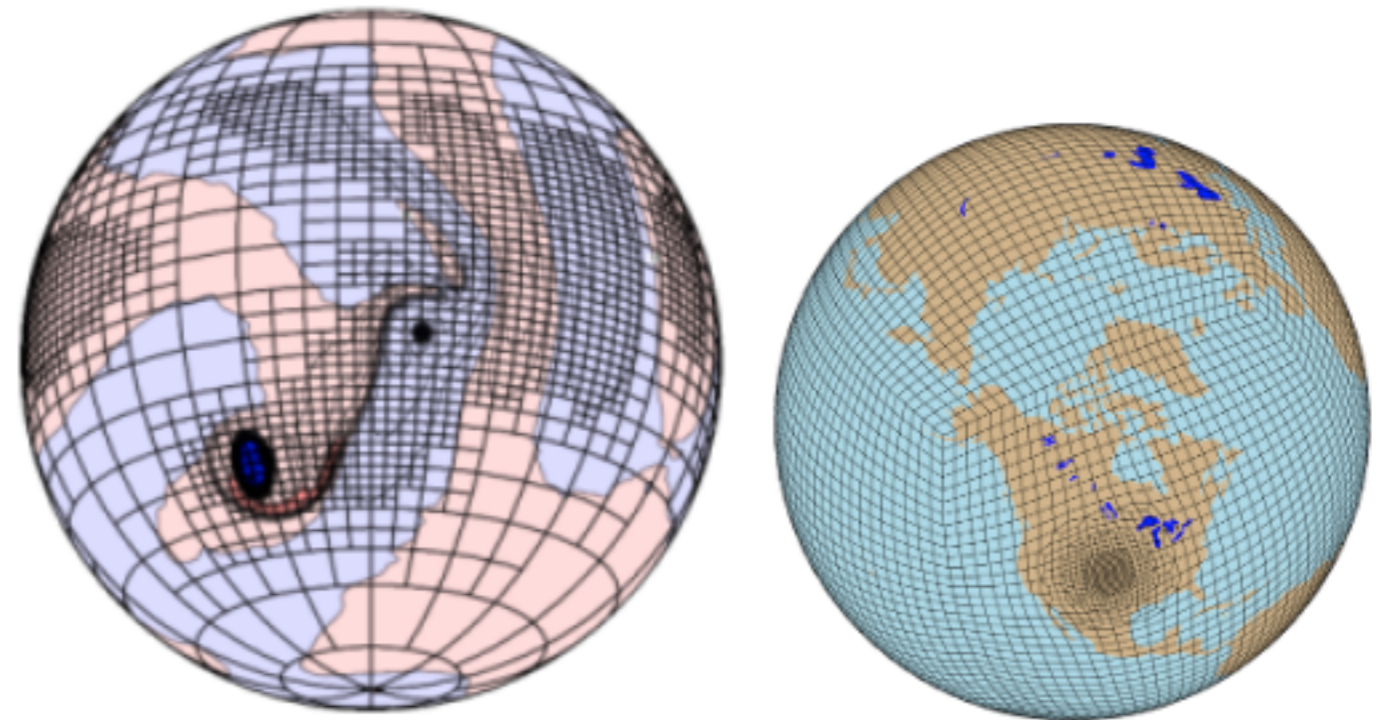
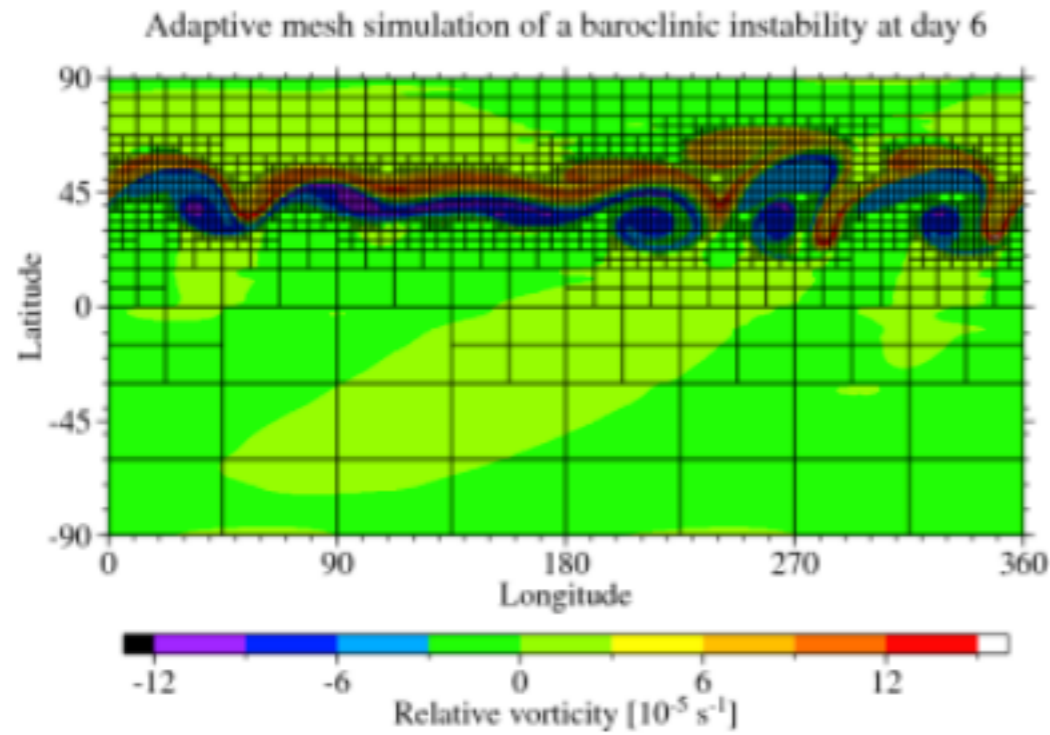
(1) Use direct FFT to find $\tilde{\rho}_{\mathbf{k}}$

(2) Find $\tilde{\phi}_{\mathbf{k}}$

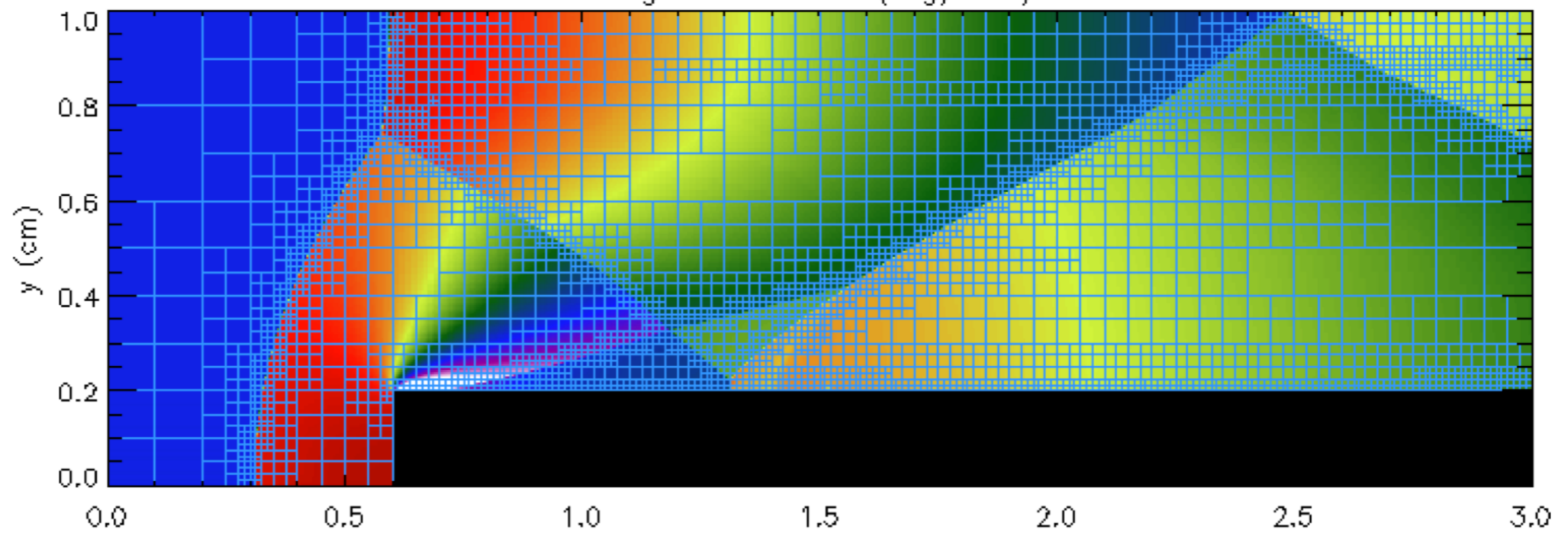
(3) Make inverse FFT to find $\phi(\mathbf{x})$

Adaptive Mesh Refinement algorithm

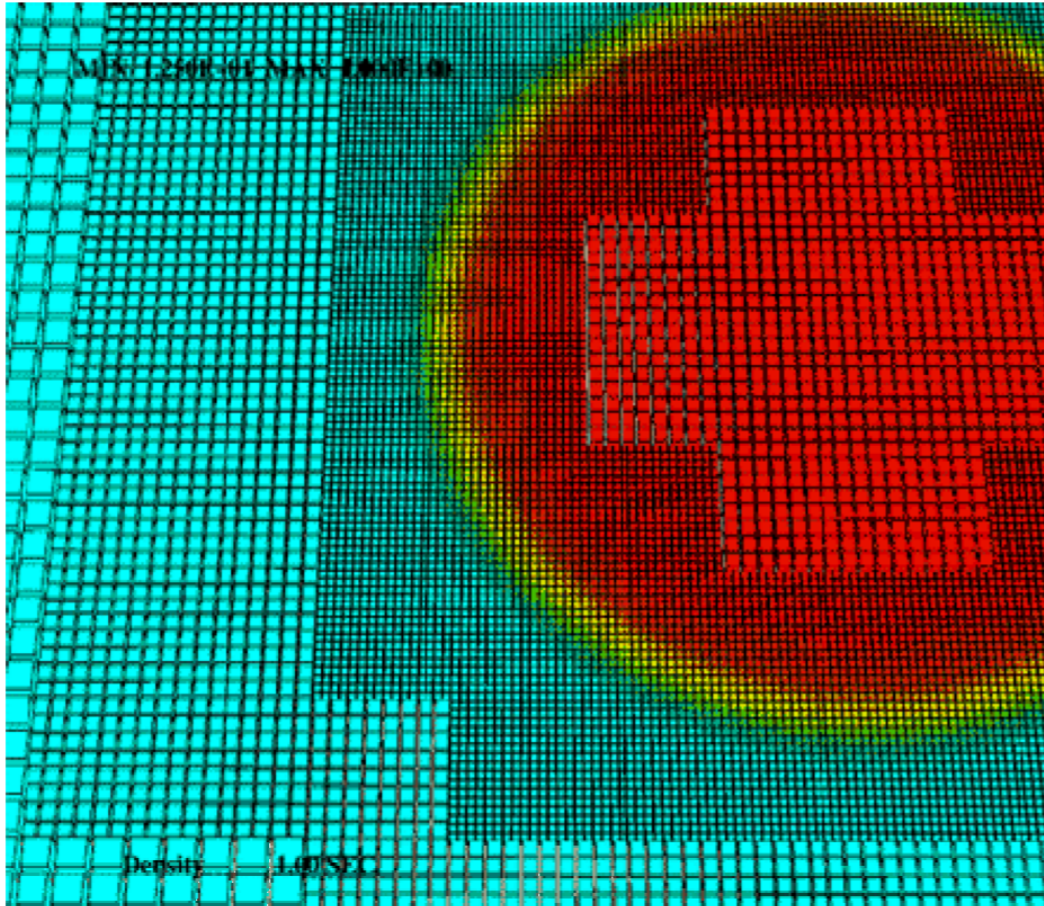
AMR cyclone simulations



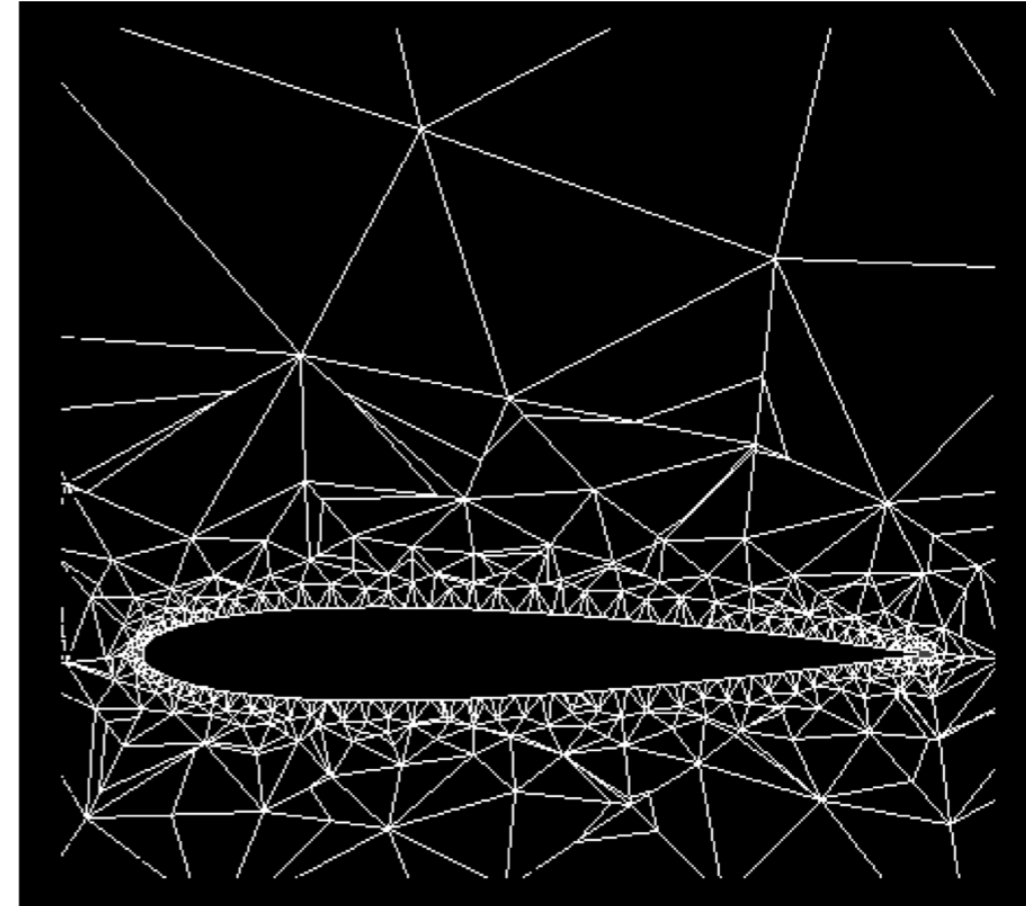
Log10 Pressure (erg/cm^3)



Structured vs unstructured AMR



Structured: hierarchy of rectangular grids or irregularly shaped meshes of cubic cells



Unstructured: highly flexible refinement meshes, efficient for cases of complicated region geometry and boundaries; more sophisticated data structures and algorithms

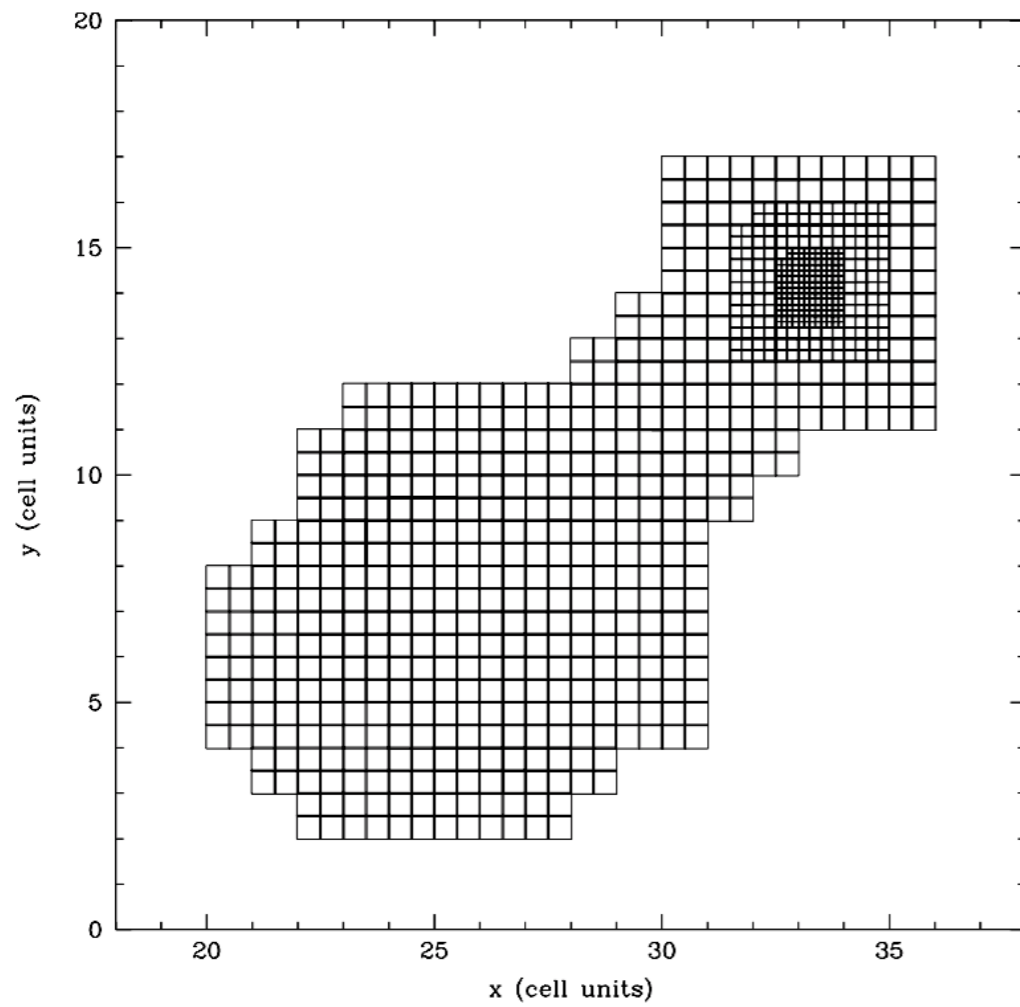
Adaptive Mesh Refinement algorithm: N-body simulations

We can improve the PM method by increasing the resolution only where it is needed: by placing additional small-size elements – cubic cells – only in regions where there are many particles and where the resolution should be larger. Codes that use this idea are called the Adaptive Mesh Refinement (AMR) codes because they recursively increase the resolution constructing a hierarchy of cubic cells with smaller and smaller elements in dense regions while keeping only large cells in regions that do not require high resolution.

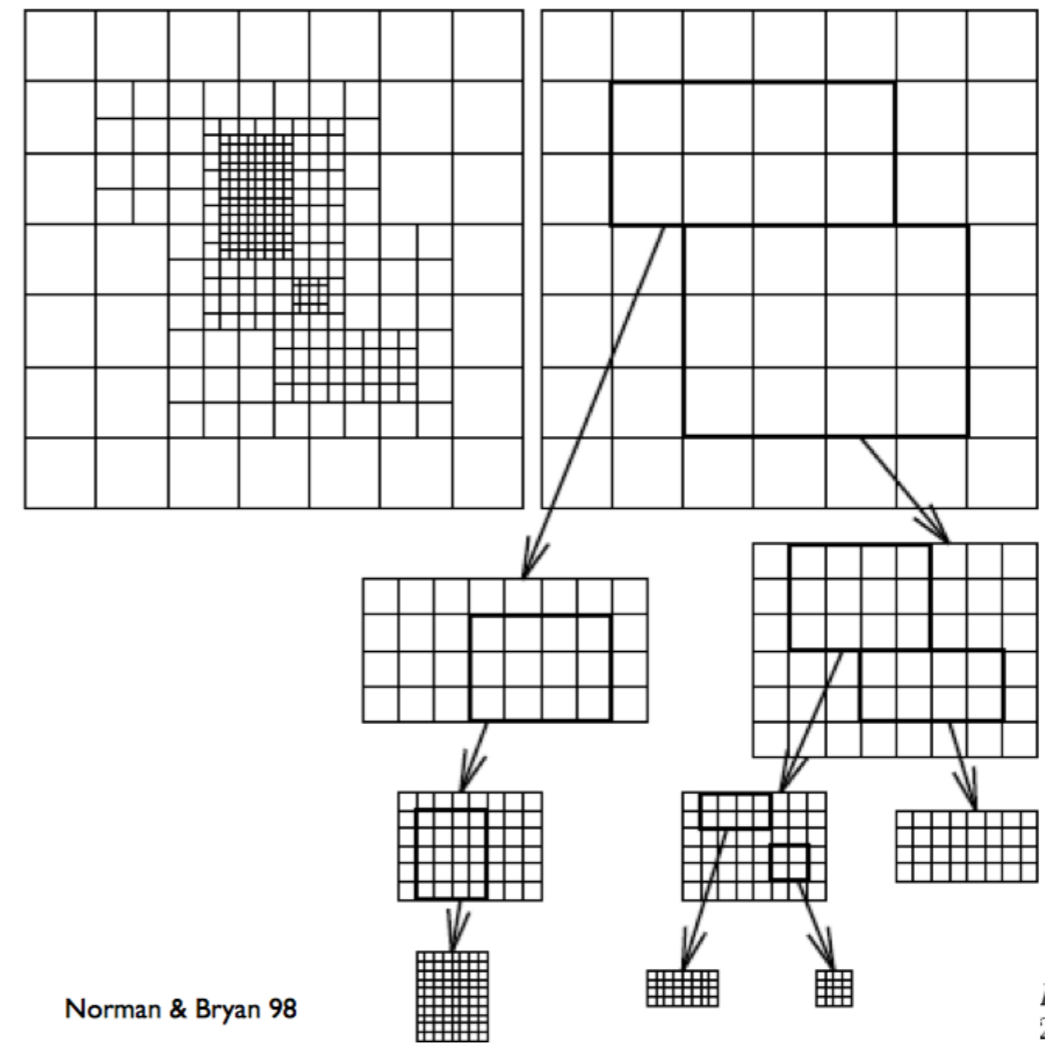
There are two ways of doing this:

- by splitting every element of the mesh, that has many particles, into 8 twice smaller boxes (Khokhlov, 1998)
- by placing a new rectangular block of cells to cover the whole high density region (Berger and Colella, 1989).

Cell refinement

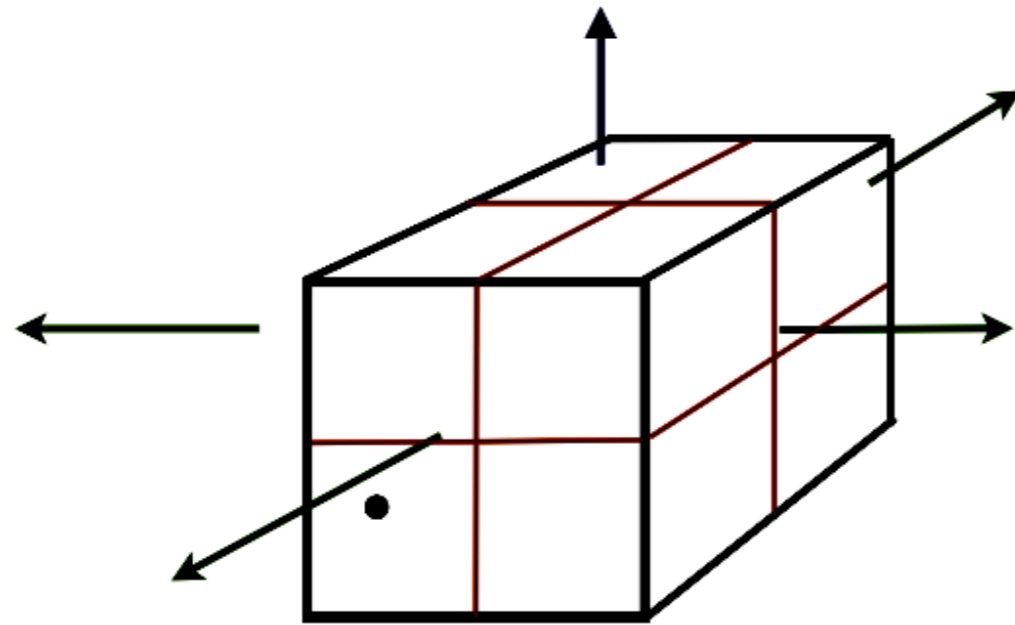


Block refinement

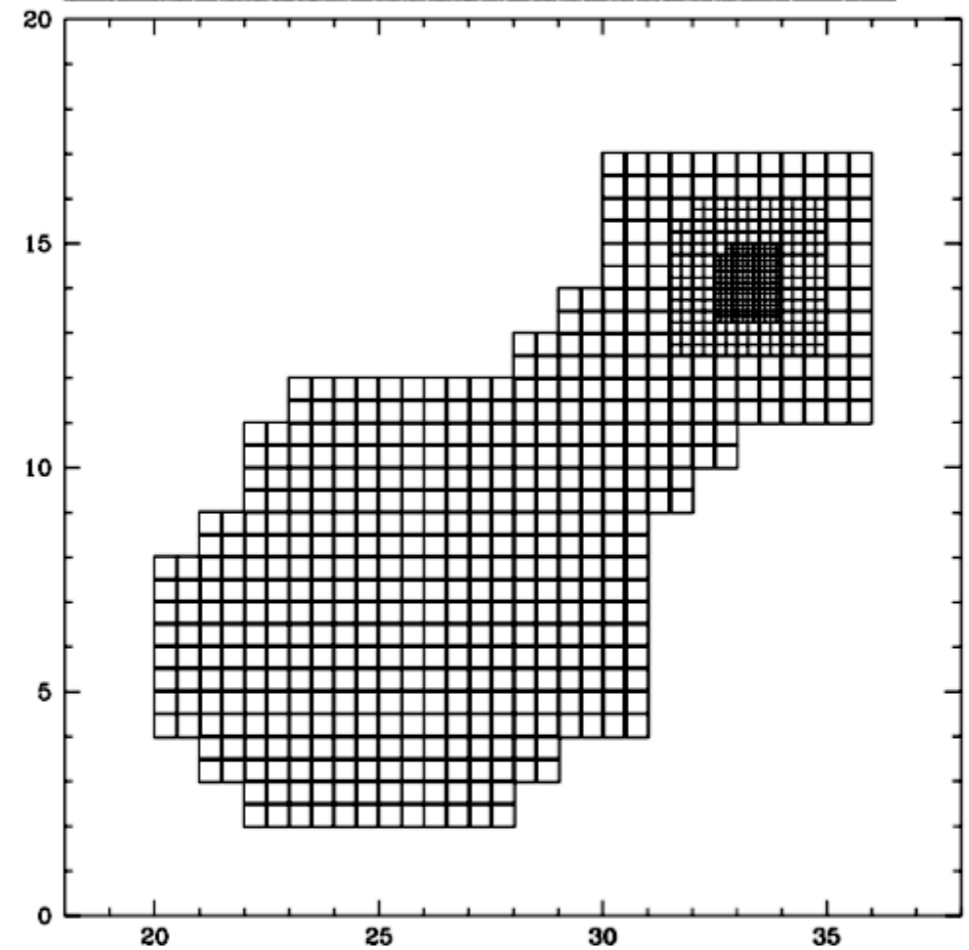
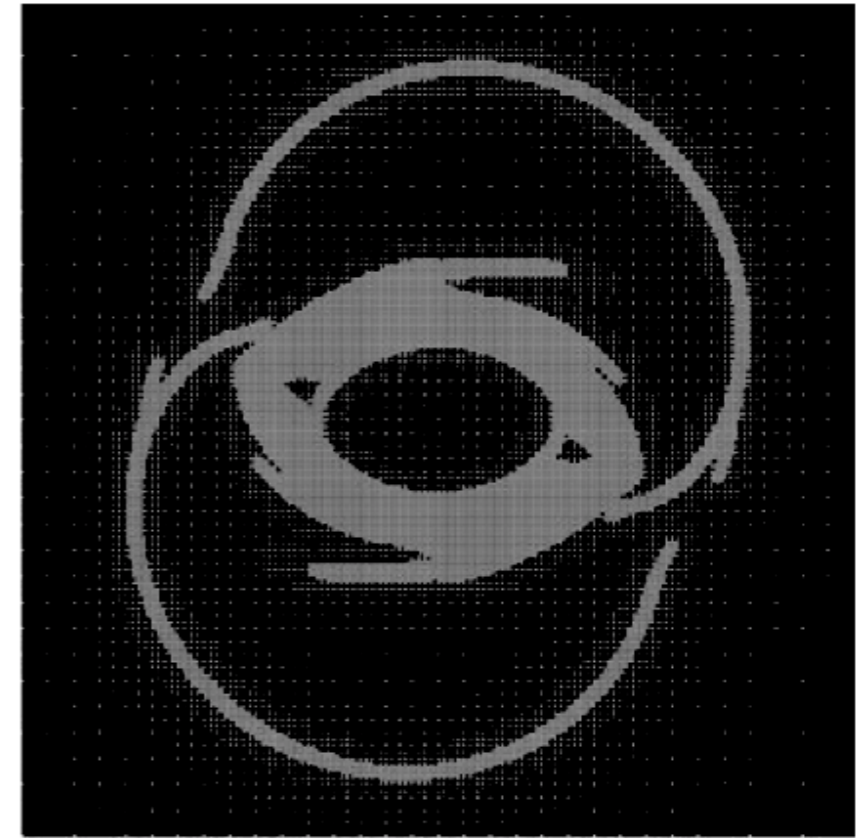


Norman & Bryan 98

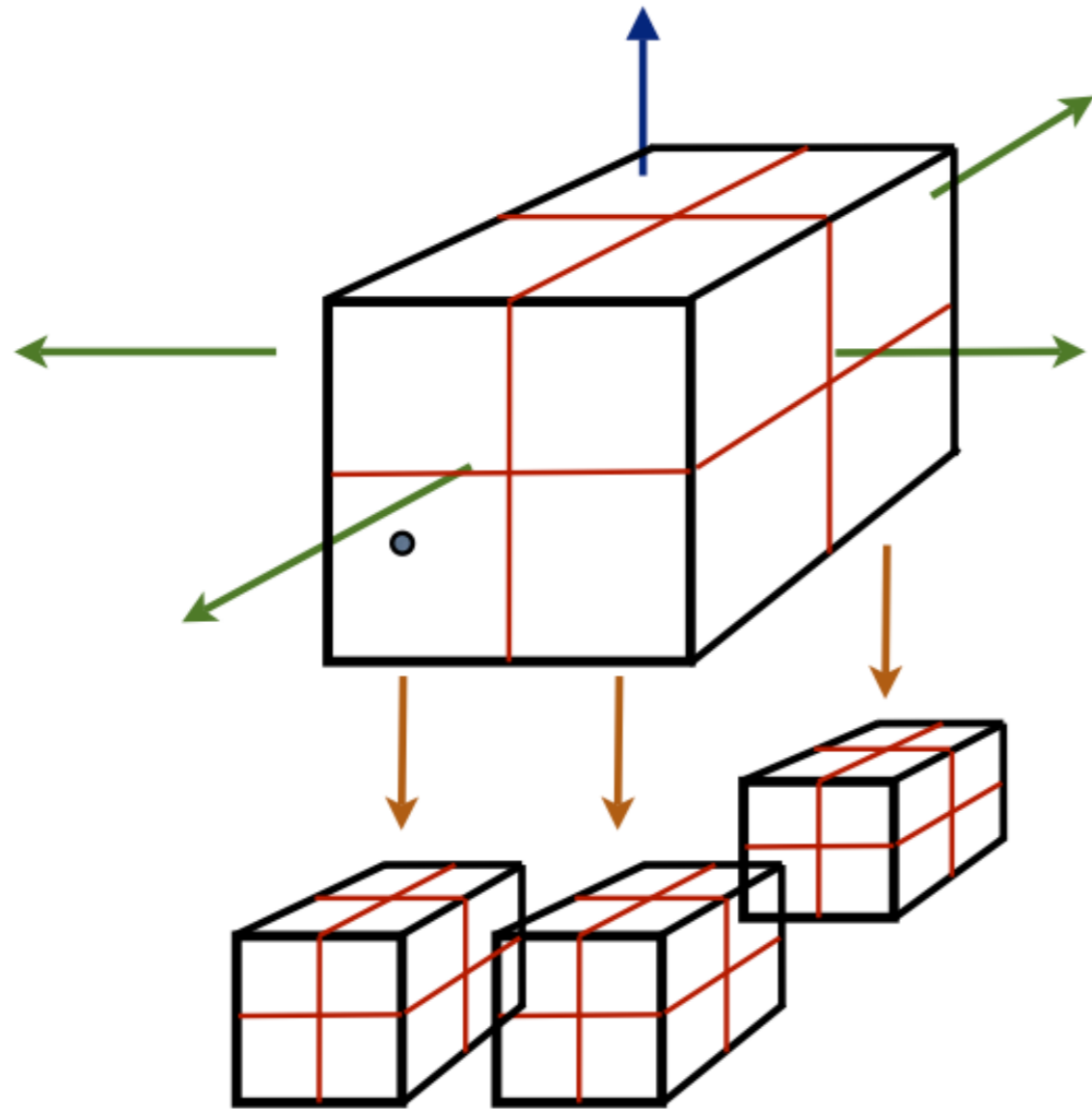
ART/RAMSES: cell splitting



Oct:
- pointers to 6 neighbors
- pointer to parent
- pointer to first child

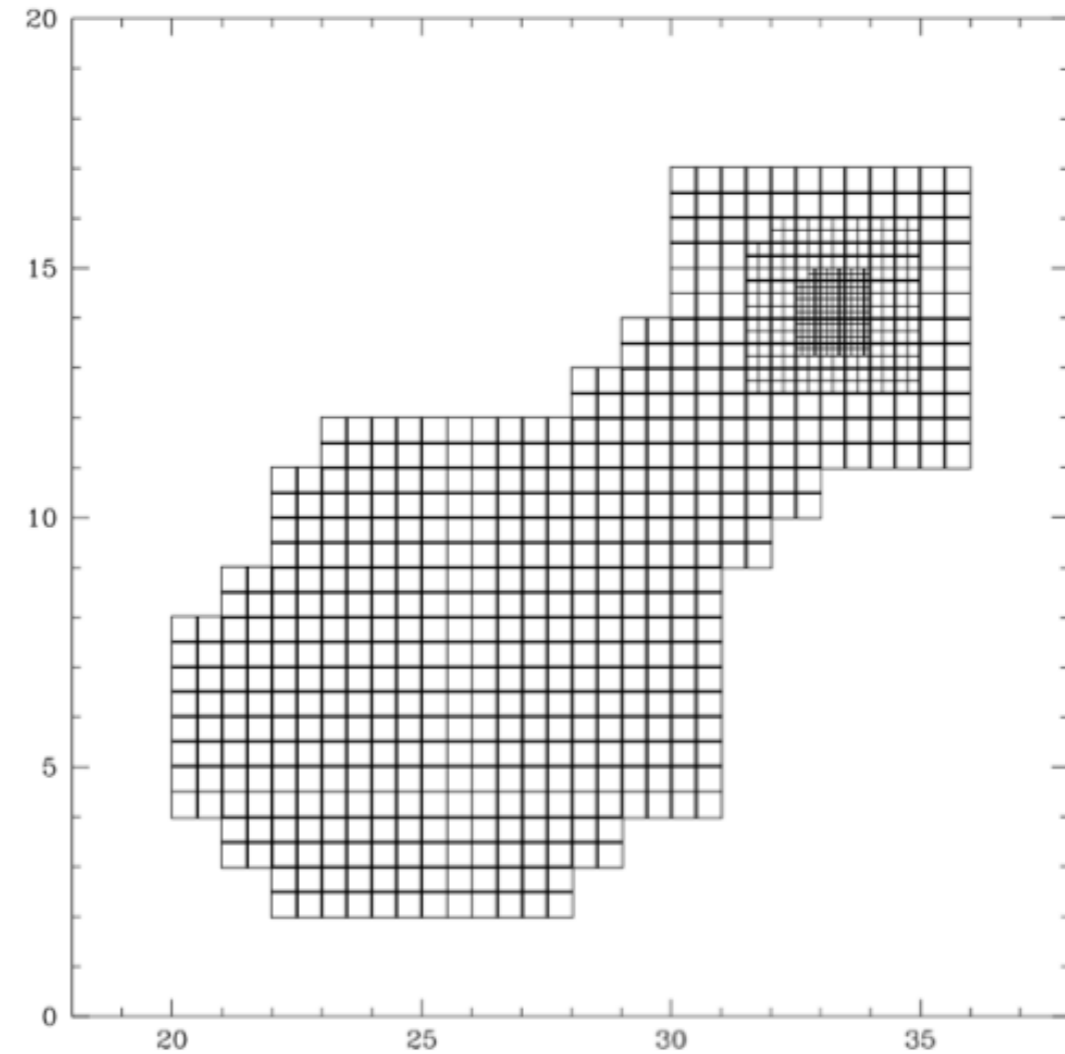
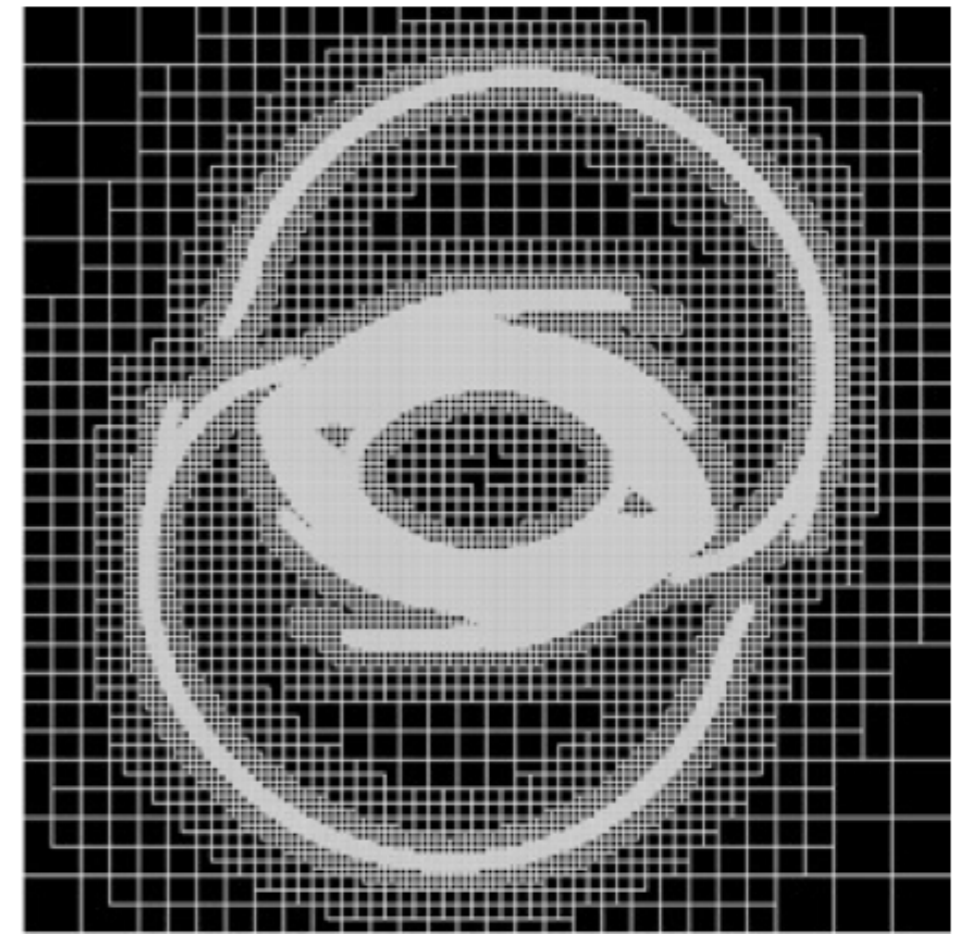


ART/RAMSES: cell splitting

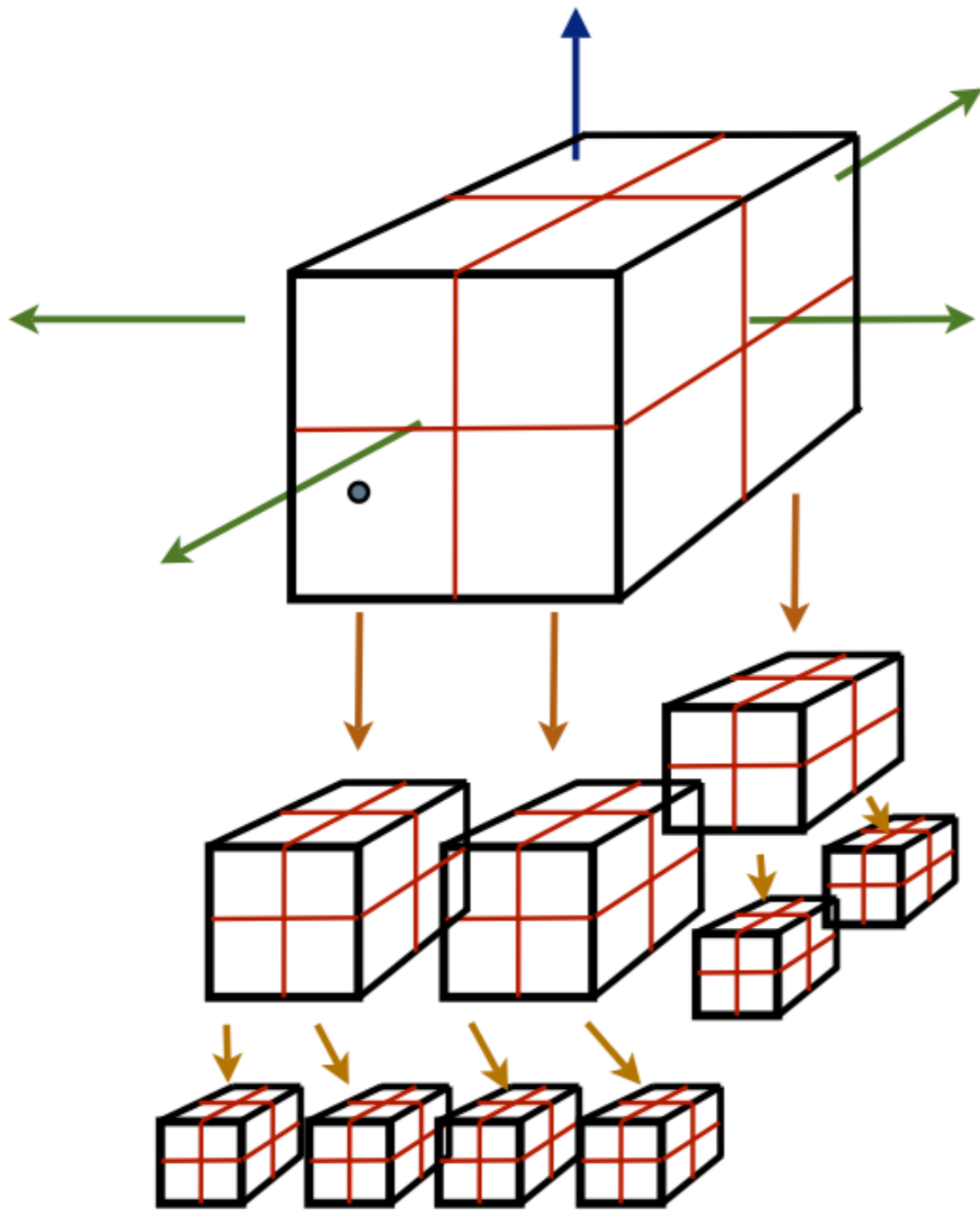


Oct:

- pointers to 6 neighbors
- pointer to parent
- pointer to first child

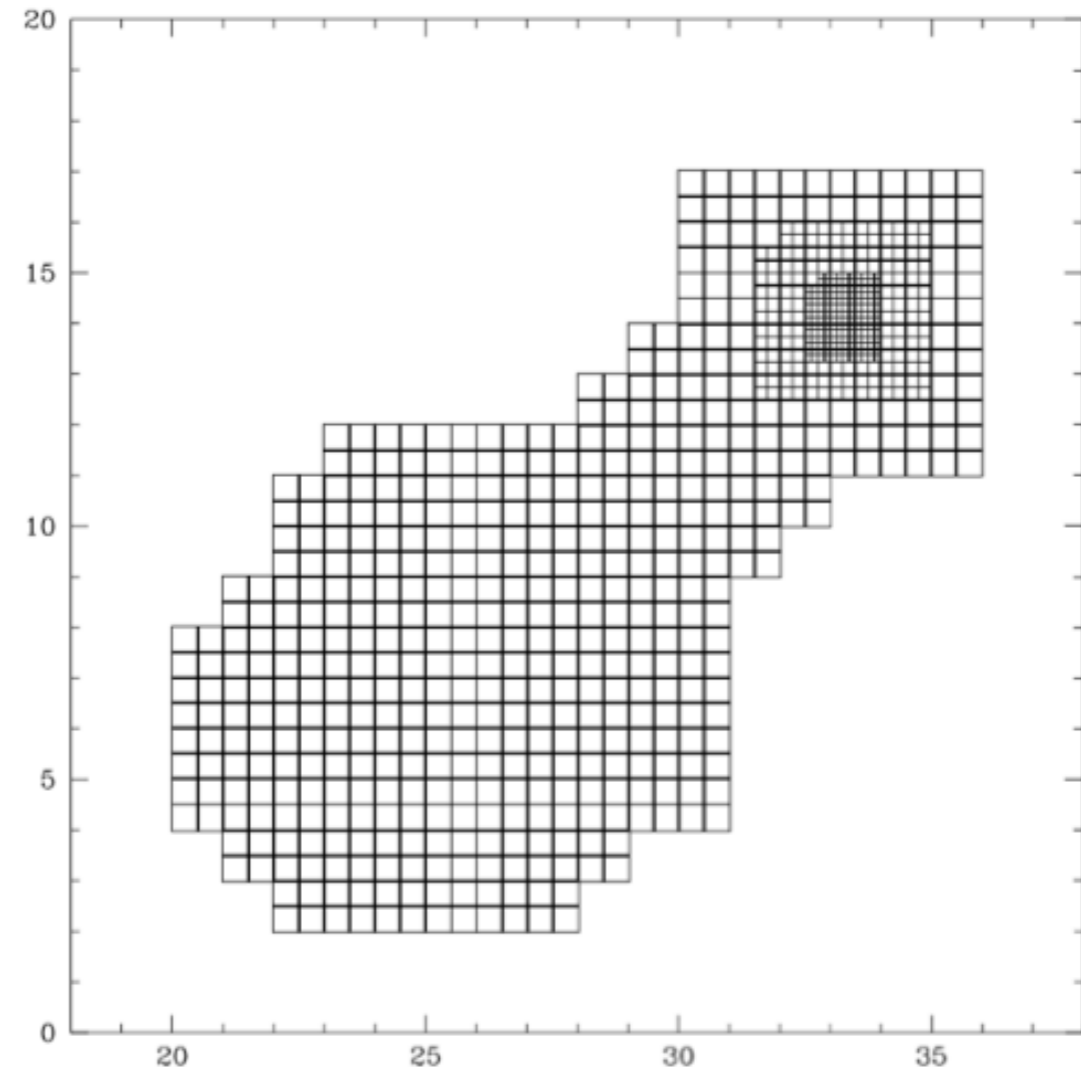
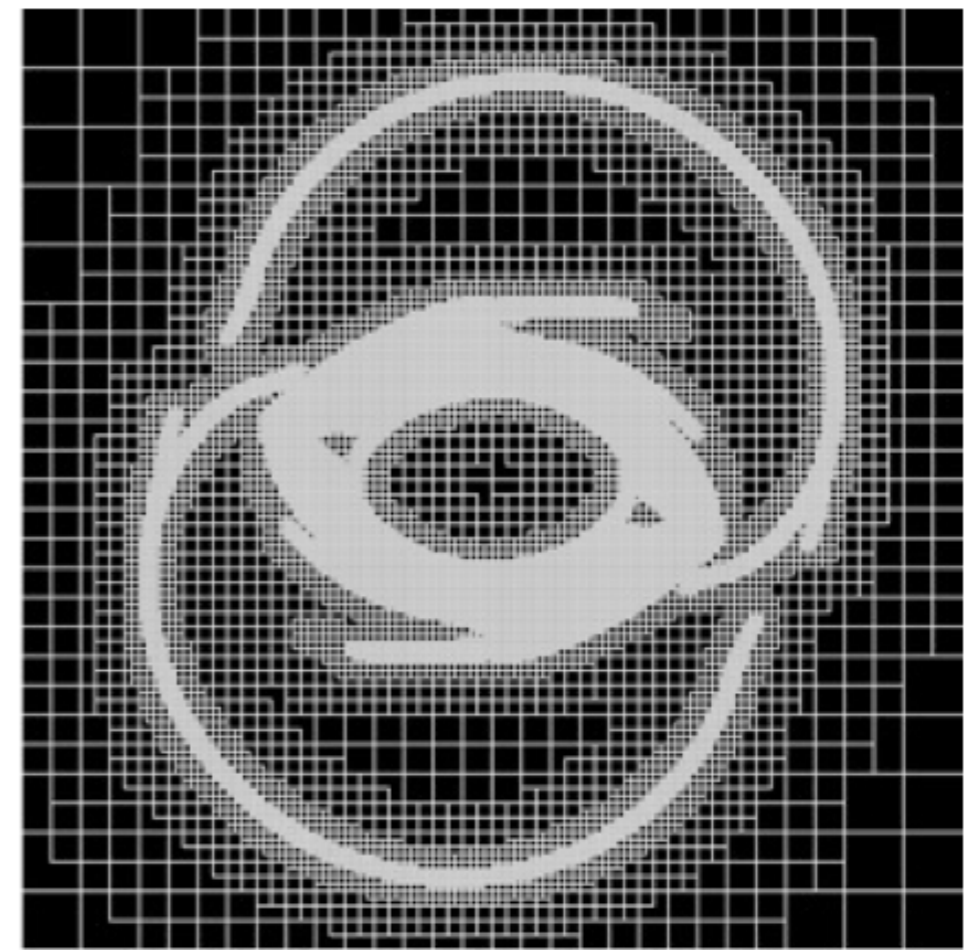


ART/RAMSES: cell splitting

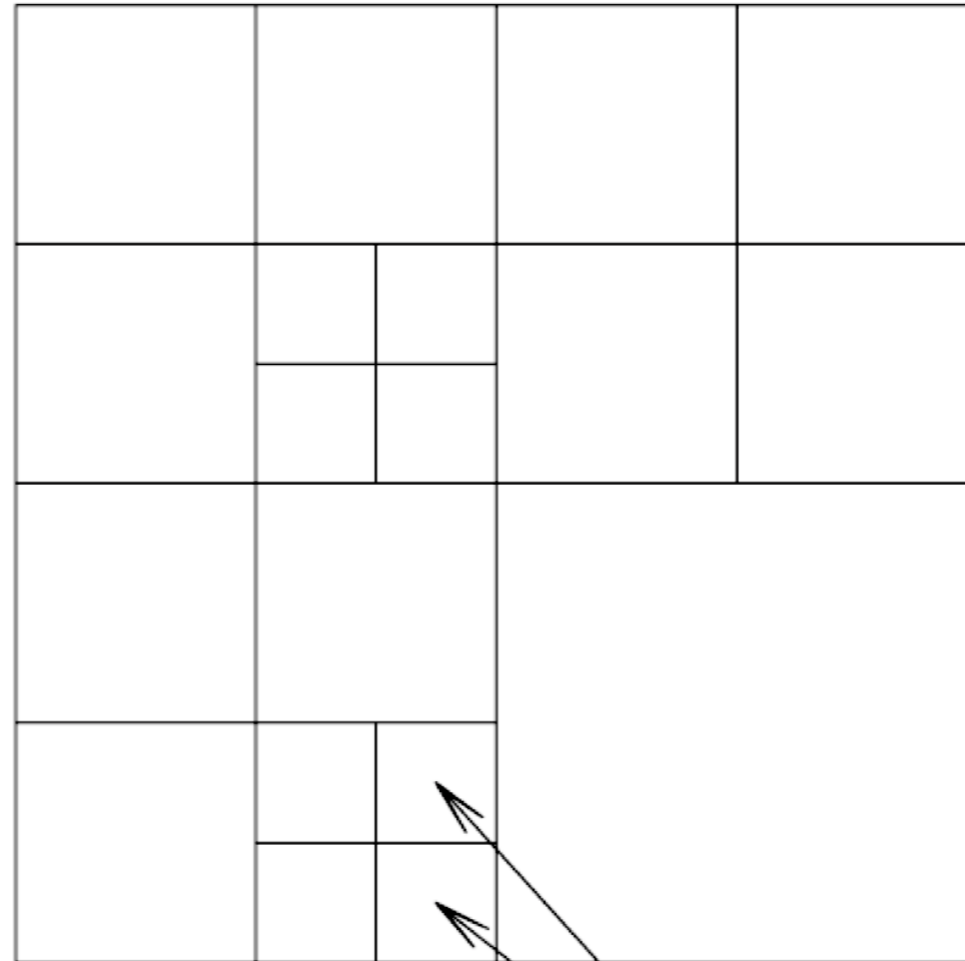


Oct:

- pointers to 6 neighbors
- pointer to parent
- pointer to first child

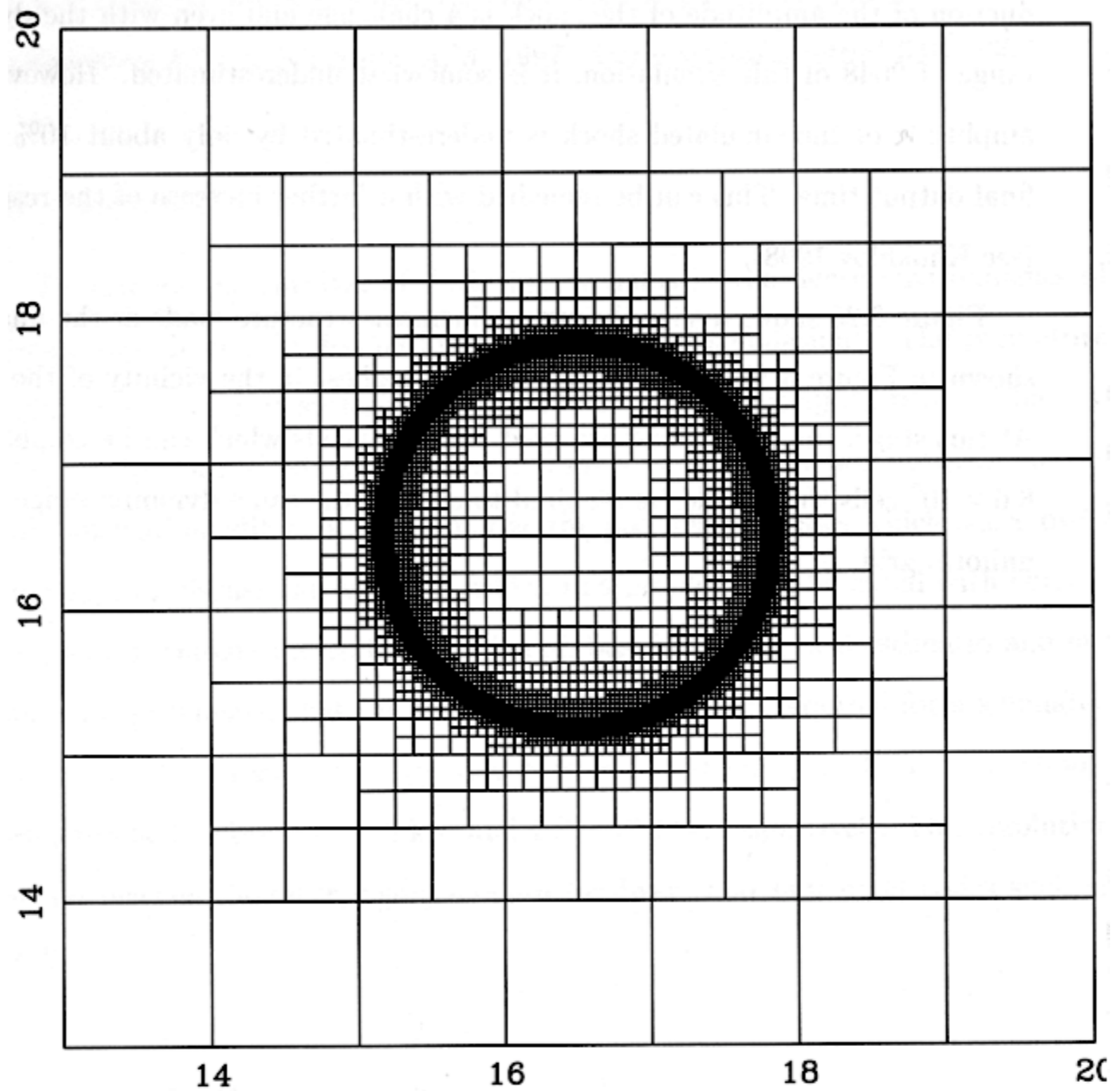


No sharp changes (more than a factor of 2) in resolution are allowed

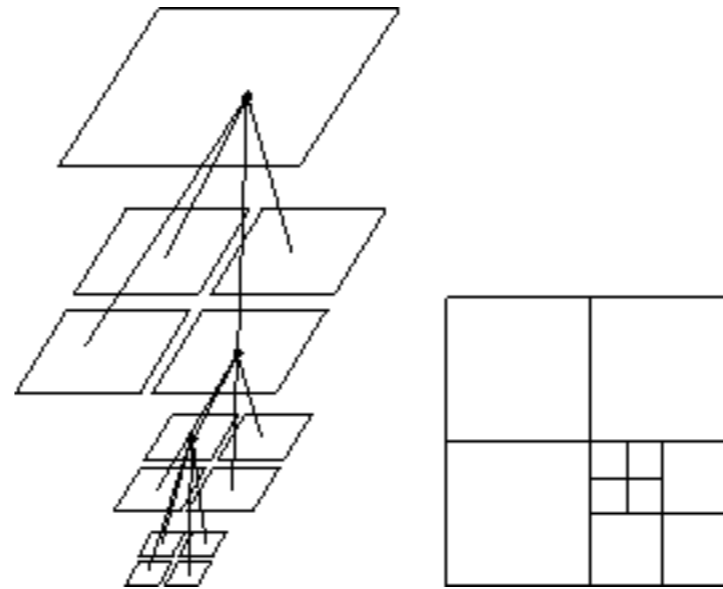


not allowed

Cell refinement: example for spherical shock wave



Oct-Tree



This is a 2D example of a quad-tree. Every square, which has too many particles is split into 4 squares, which have $1/2$ of original size. In 3D every cubic cell is split into 8 smaller cells. If any of new cells still have too many particles, they are also split into 8 even smaller cells.

The structure is adaptive: the level of the tree depends on local density. When particles move, density changes and so does the structure.

Oct Trees are used for TREE codes and for some types of Adaptive-Mesh-Refinement (AMR) codes.

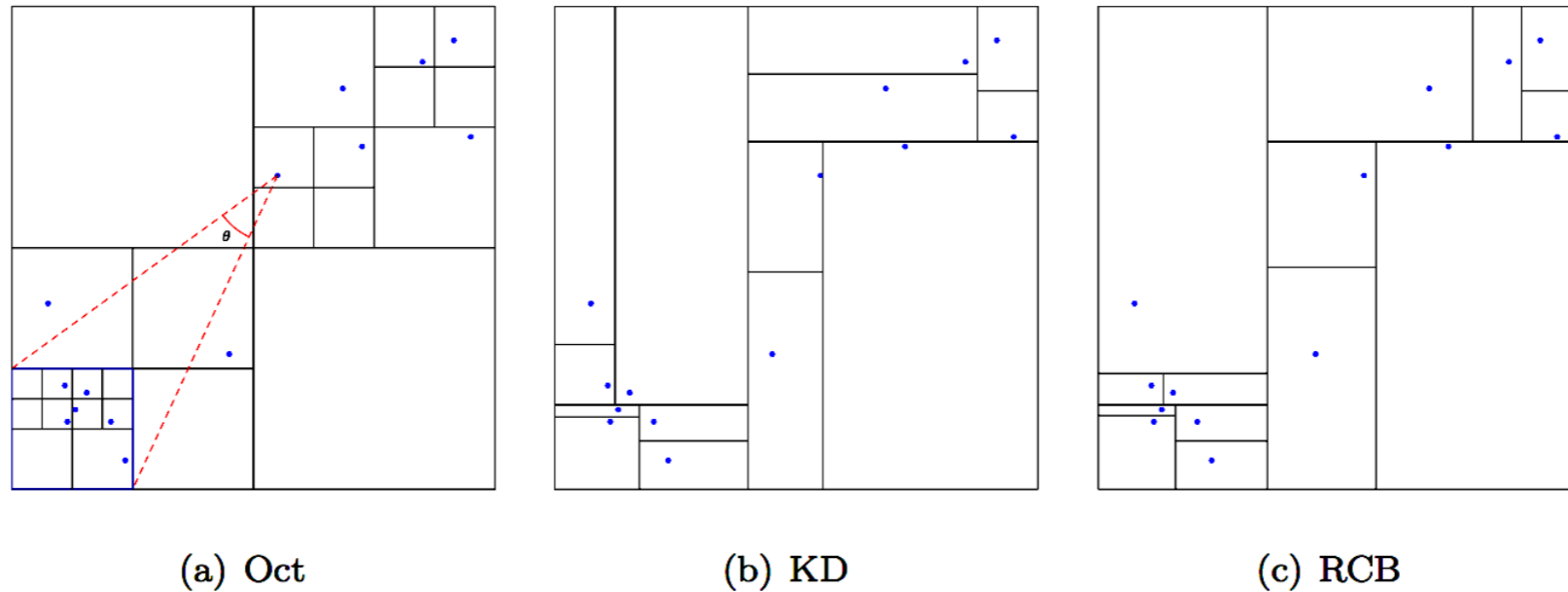
Once the structure is created, we can find grav.potential or grav acceleration using different techniques:

- Sum contributions from different nodes. Use 'opening angle' criterion to select appropriate level of refinement. This is for TREE codes.
- Solve the Poisson equation by different iterative schemes. This is for AMR codes.

Modern N-body codes are typically combinations of Particle-Mesh code (very fast) with either TREE or AMR additions for high resolution in dense environments.

TREE algorithms

- split particles into groups of different size and replace force from individual particles with a single multipole force of the whole group. The larger is the distance from a particle, the bigger is the allowed size of the particle group.

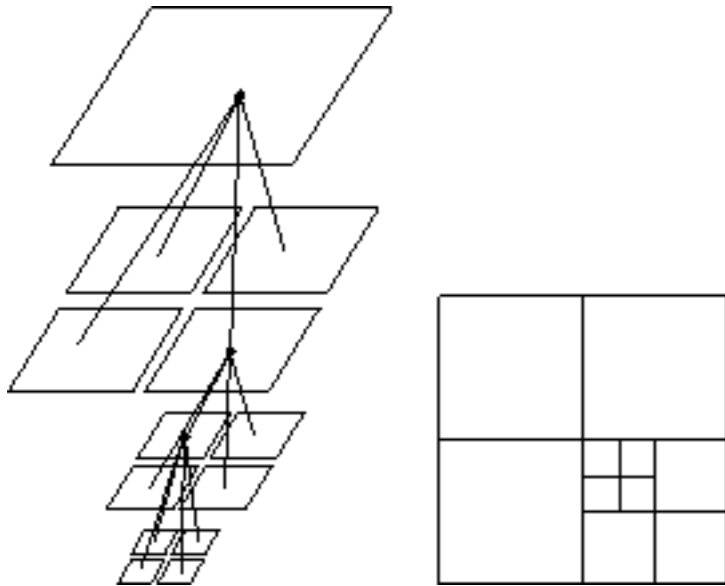


Examples of particle grouping algorithms for TREE codes. Left panel: the oct tree for 14 particles presented by blue circles. If the number of particles in a cell exceeds a specified threshold (in this case one particle), it is split into 8 small cubic cells (4 cells in 2D). Red dashed lines show opening angle θ for a particles close to the centre and for a cell indicated by a thick blue square. Middle panel: binary KD tree for the same set of particles. Boundaries of rectangular cells are defined by position of medians along each alternating direction. In some cases cells are quite elongated. Right panel: Recursive Coordinate Bisection tree. Cells are split at the center of mass with the direction of the bisecting plane being perpendicular to the direction of the maximum cell size. Cells are less elongated than in the case of KD trees.

Tree is truncated once a cell reaches a specified minimum number of particles. In this case the cell called a leaf. The number of particles in a leaf can be as low as one. However, it can be significantly larger. If a leaf has more than one particle, then forces between particles in the cell are estimated using pair-wise summation. This can be faster than building more levels of the TREE hierarchy.

Multipole expansion. A number of physical quantities is collected for each cell that are used for force estimates. GADGET-2 code stores the mass and the center of mass of all particles in a given cell.

Multipole expansion up to hexadecapole is used in PKDGRAV. Quadrupole expansion was also used. There is no rule what order of expansion to select. Low orders are faster to calculate and less memory is needed to store the information. At the same time, higher orders of expansion may allow one to use larger opening angles resulting in faster overall calculations. Grouping algorithm may also affect the selection of the expansion. The bisection trees can produce elongated cells implying that a higher order of mass expansion may be needed to maintain force accuracy.



Cell opening condition. Once the TREE is constructed and all information regarding mass distribution in each cell is stored, we start to find the forces by looping through all leaves and for each leaf by walking along the TREE down from the largest cells.

Each cell of size l is tested whether angle $\theta \approx l/d$ at which it is seen by particles in the leaf at distance d is too large. If θ exceeds a specified threshold, the force contributions are not taken from the cell itself. We “open” the cell meaning that we descend to children of the cell and test them regarding their opening angles. Once the opening angle is small enough, the force contribution from the cell is accepted, and the algorithm proceeds to the next top-level cell.

Particular implementation of the cell-opening condition changes from code to code. In GADGET-2 the force is accepted if

$$\theta = \frac{l}{d} \leq \sqrt{\alpha g / [GM/d^2]}, \quad (29)$$

where g is the particle acceleration from the previous time-step, d is the distance from the particle to the cell of mass M and linear size l . Here α is a tolerance parameter defining the error of the force. There is an additional condition that each coordinate distance of the particle and geometrical cell center should be small:

Moving mesh hydrodynamics (MMH) (Pen 1998)

Lagrangian mesh with expansion limiter to prevent zones from becoming too distorted

Exploits fact that rotational mode is decaying in cosmology

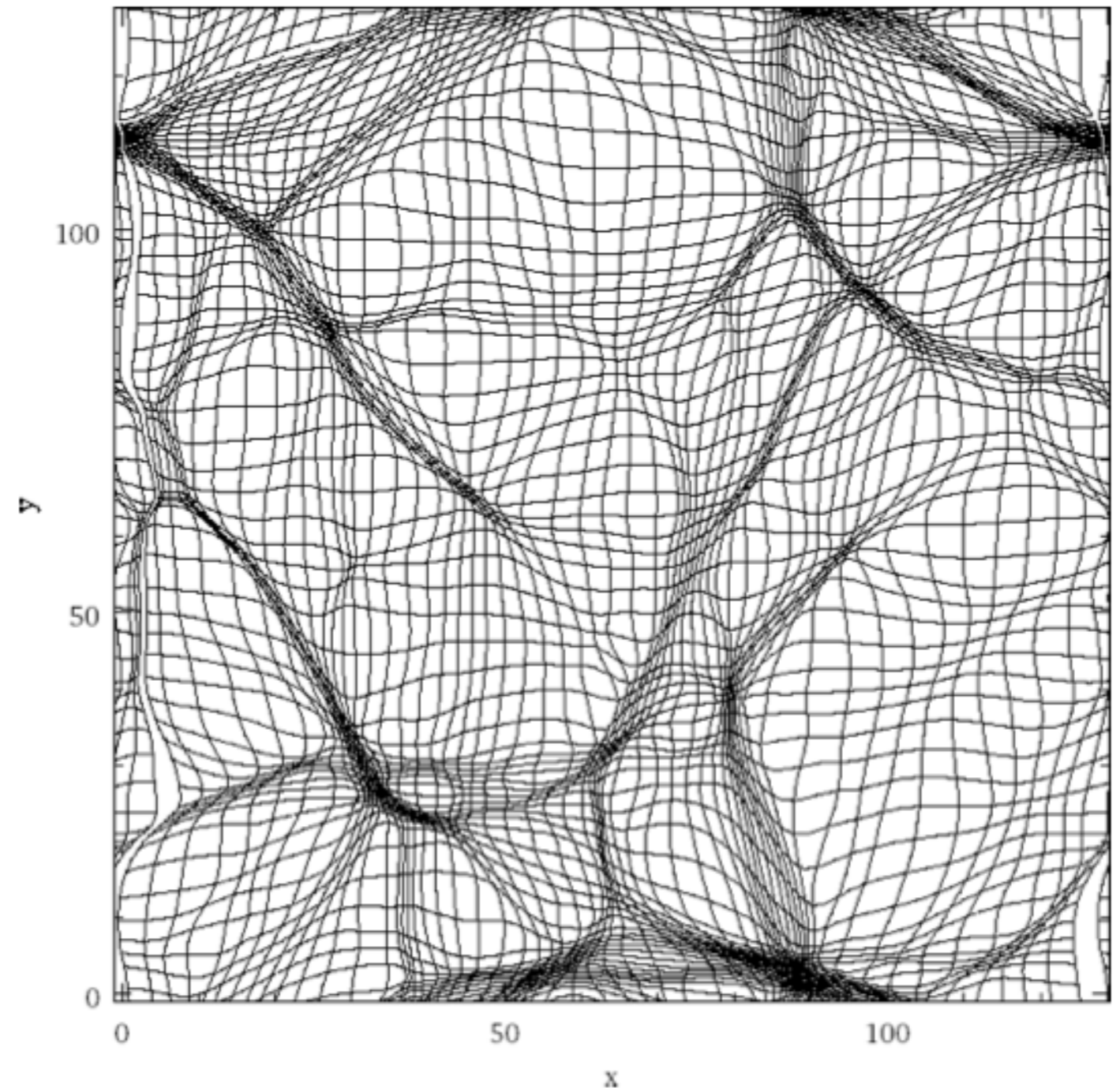
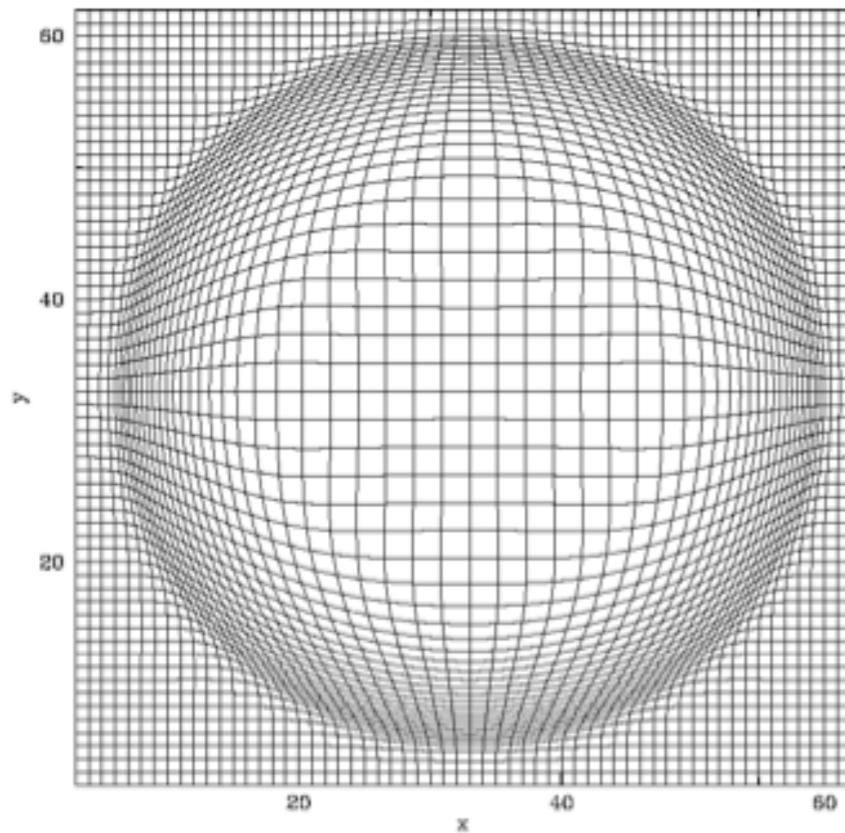


FIG. 5.—Mesh geometry at midplane for Sedov Taylor simulation. The expansion limiter prevents the cells from expanding more than a factor of 10 in volume.

Nbody:

cpu requirements: $\text{cpu-time} = 5 \text{sec} \times N_{\text{steps}} \times \left(\frac{N_{\text{particles}}}{10^4} \right)^2$

energy conservation: 2000 particles; $\left. \frac{E_{\text{kin}}}{E_{\text{pot}}} \right|_{\text{init}} = -0.15$
 $\epsilon = 3e-3$
 $R = 1$

Δt	Error in Energy _{tot}	$t_{\text{dyn}} \approx 1.3$
$3e-3$	2%	
$1.5e-3$	1%	
$0.75e-3$	0.5%	

$\epsilon = 1.5e-3$

Δt	Error	
$1.5e-3$	1%	$\Rightarrow 1000 \text{ steps} / t_{\text{dynamical}}$

20,000 particles $\text{cpu-time/step} = 18 \text{sec}$
30,000 particles =