N-Body project

Computational Astrophysics, HS24

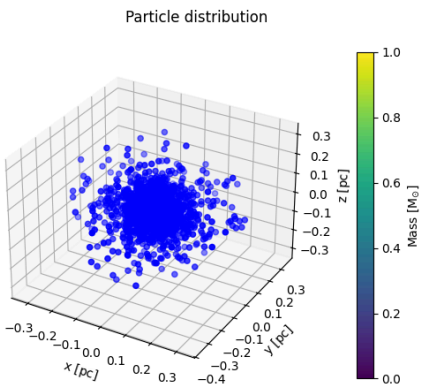
04.02.2024

Rémy Moll

1 N-body forces and analytical solutions

Overview - the system

Get a feel for the particles and their distribution



The system at hand is characterized by:

- $N \sim 10^4$ stars
- a *spherical* distribution

$\Longrightarrow$ treat the system as a **globular cluster**[1]

[1] Unit handling [code]

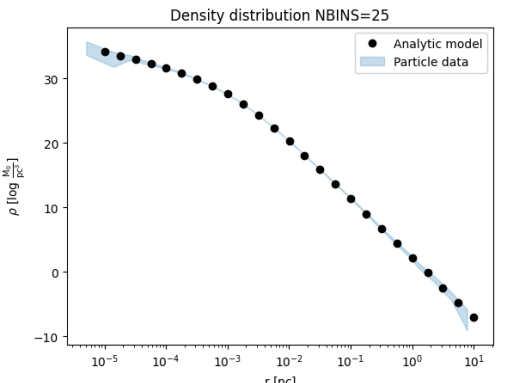
Density

Compare the computed density[2] with the analytical model provided by the *Hernquist* model:

$$\rho(r) = \frac{M}{2\pi} \frac{a}{r \cdot (r+a)^3}$$

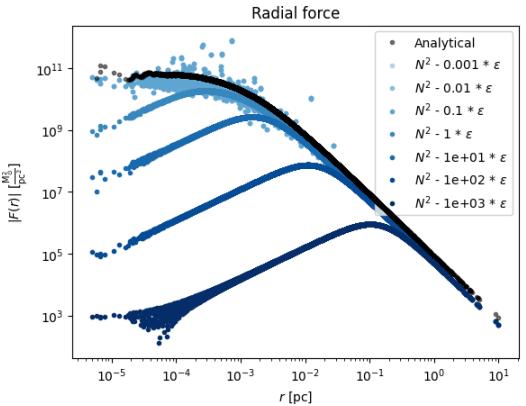
where we infer a from the half-mass radius:

$$r_{\mathrm{hm}} = \left(1 + \sqrt{2}\right) \cdot a$$



[2] Density sampling [code]

Force computation



Discussion:

- the analytical[3] method replicates the behavior accurately
- at small softenings the N^2 [4] method has noisy artifacts
- a $1 \cdot \varepsilon^5$ softening is a good compromise between accuracy and stability

[3] Analytical force [code]
[4] N^2 force [code]
[5] ε computation [code]

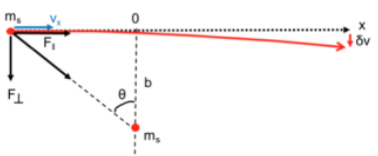
Relaxation

We express system relaxation in terms of the dynamical time of the system.

$$t_{\text{relax}} = \overbrace{\frac{N}{8 \log N}}^{n_{\text{relax}}} \cdot t_{\text{crossing}}$$

where the crossing time of the system can be estimated through the half-mass velocity $t_{\text{crossing}} = \frac{v(r_{\text{hm}})}{r_{\text{hm}}}$.

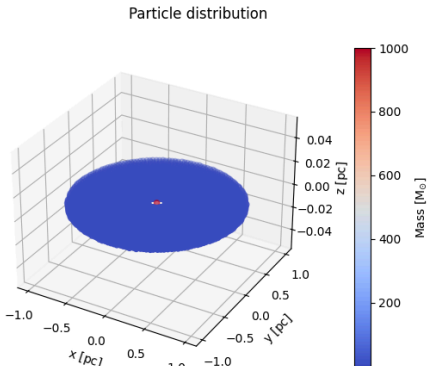
We find a relaxation of ≈ 30 Myr ([code])

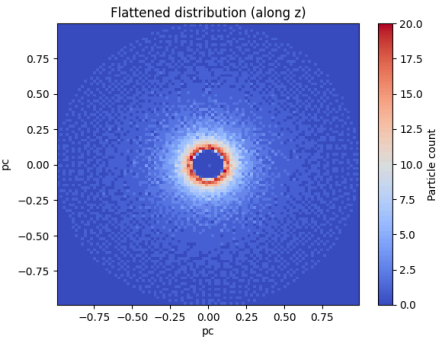


- Each star-star interaction contributes $\delta v \approx \frac{2Gm}{b}$
- Shifting by ε **dampens** each contribution
- $\Rightarrow$ relaxation time increases

2 Particle Mesh

Overview - the system





$$\Longrightarrow \text{use } M_{\text{sys}} \approx 10^4 M_{\text{sol}} + M_{\text{BH}}$$

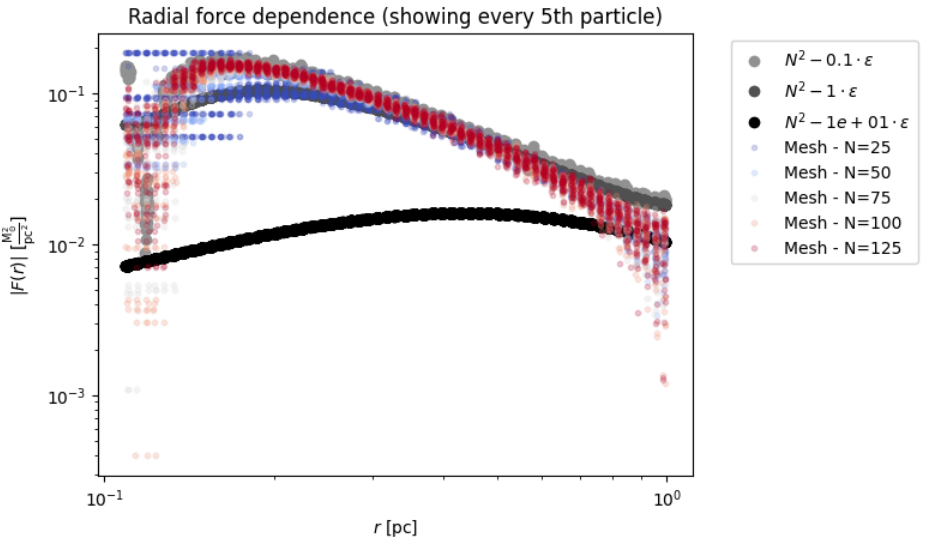
Force computation

```python
def mesh_forces(particles: np.ndarray, G: float = 1, n_grid: int = 50, mapping: callable = None) -> np.ndarray:
    """
    Computes the gravitational force acting on a set of particles using a mesh-based approach.
    Assumes that the particles array has the following columns: x, y, z, m.
    """
    max_pos = np.max(np.abs(particles[:, :3]))
    mesh, axis, spacing = create_mesh(-max_pos, max_pos, n_grid)

    fill_mesh(particles, mesh, axis, mapping)
    # we want a density mesh:
    cell_volume = spacing**3
    rho = mesh / cell_volume

    # compute the potential and its gradient
    phi = mesh_poisson(rho, G, spacing)

    # get the acceleration from finite differences of the potential
    ax, ay, az = np.gradient(phi, spacing)
    a_vec = - np.stack([ax, ay, az], axis=0)

    # compute the particle forces from the mesh potential
    forces = np.zeros_like(particles[:, :3])
    ijks = np.digitize(particles[:, :3], axis) - 1

    for i in range(particles.shape[0]):
        m = particles[i, 3]
        idx = ijks[i]
        forces[i] = m * a_vec[..., idx[0], idx[1], idx[2]]

    return forces


def mesh_poisson(mesh: np.ndarray, G: float, spacing: float) -> np.ndarray:
    """
    Solves the poisson equation for the mesh using the FFT.
```

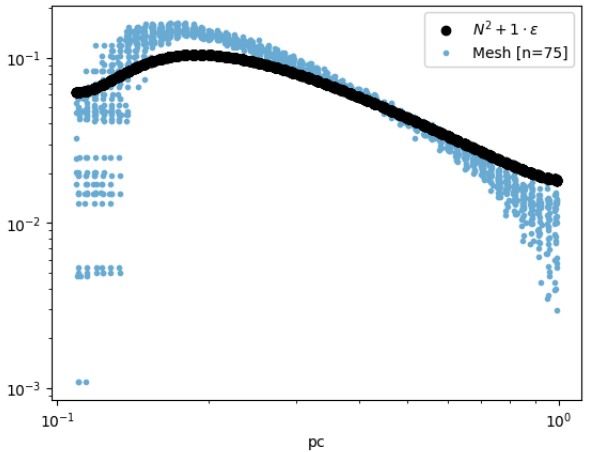
Force computation (ii)

```
    Returns the the potential - phi
    """
    rho_hat = fft.fftn(mesh)

    # we also need the wave numbers
    k = fft.fftfreq(mesh.shape[0], spacing) * (2 * np.pi)
    # assuming the grid is cubic
    kx, ky, kz = np.meshgrid(k, k, k)
    k_sr = kx**2 + ky**2 + kz**2

    k_sr[k_sr == 0] = np.inf

    phi_hat = - 4 * np.pi * G * rho_hat / k_sr
    return np.real(fft.ifftn(phi_hat))
```

Force computation (iii)



Force computation (iv)

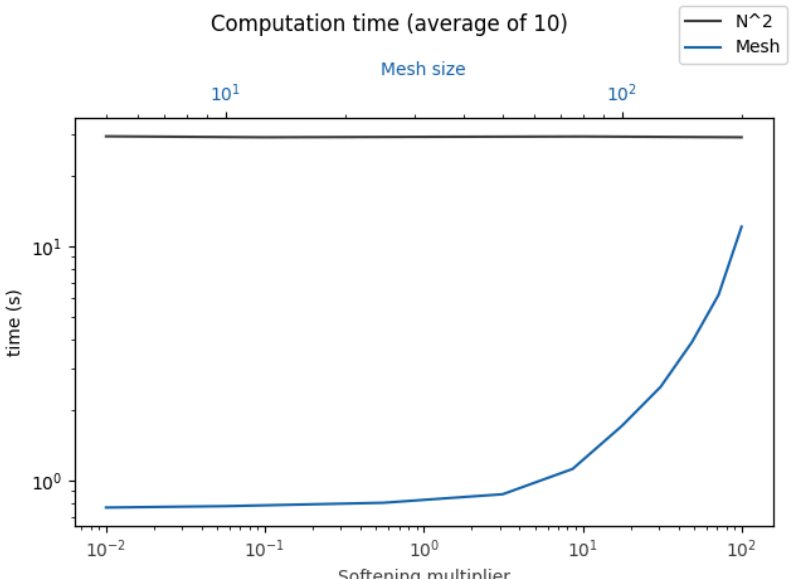


- using the (established) baseline of N^2[6] with $1 \cdot \varepsilon$[7] softening
- small grids[8] are stable but inaccurate at the center
- very large grids have issues with overdiscretization

$\Longrightarrow 75 \times 75 \times 75$ as a good compromise

[6] N^2 force [code]
[7] ε computation [code]
[8] Mesh force [code]

Force computation (v)



Time integration

Integration step

```python
def runge_kutta_4(y: np.ndarray, t: float, f: callable, dt: float):
    """
    Runge-Kutta 4th order integrator.
    """
    k1 = f(y, t)
    k2 = f(y + k1/2 * dt, t + dt/2)
    k3 = f(y + k2/2 * dt, t + dt/2)
    k4 = f(y + k3 * dt, t + dt)
    return y + (k1 + 2*k2 + 2*k3 + k4)/6 * dt
```

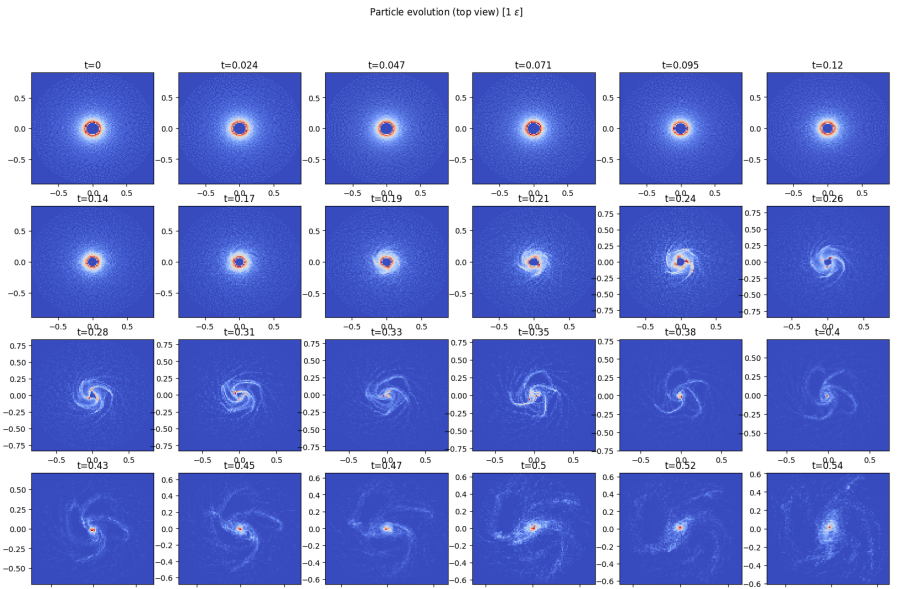
Timesteps Chosen such that displacement is small (compared to the inter-particle distance) [code]:

$$dt = 10^{-3} \cdot \frac{S}{v_{\text{part}}}$$

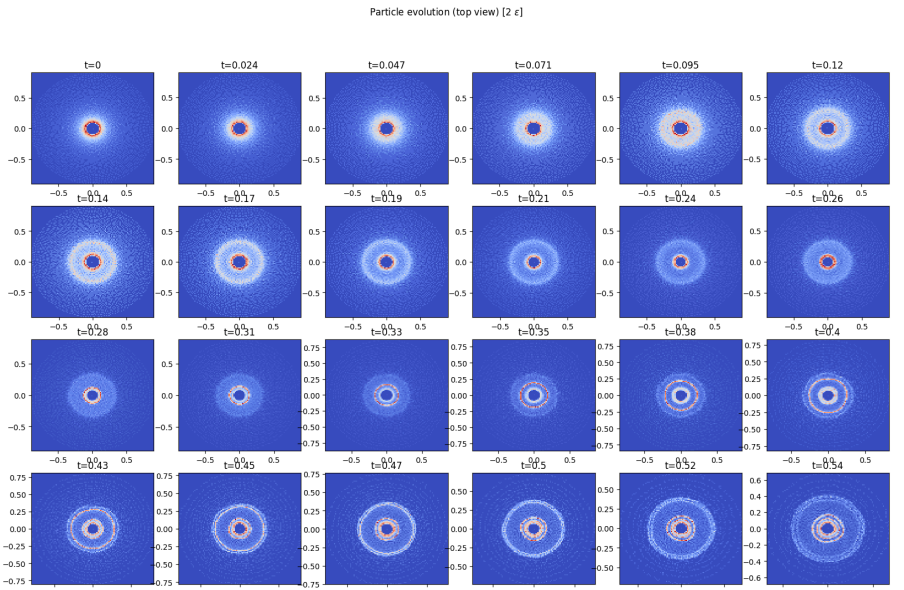
Full integration

[code]

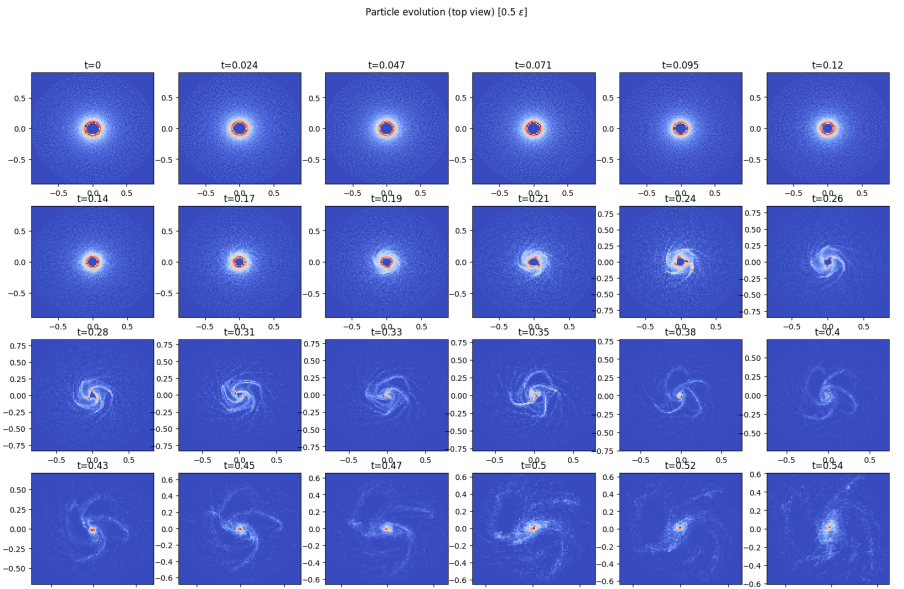
First results



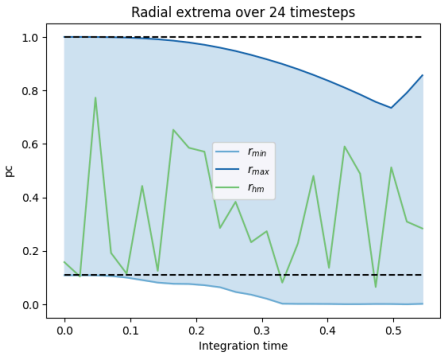
Varying the softening

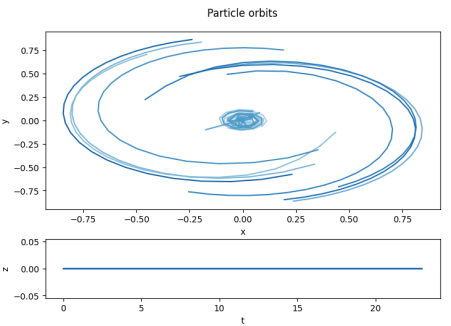


Varying the softening (ii)

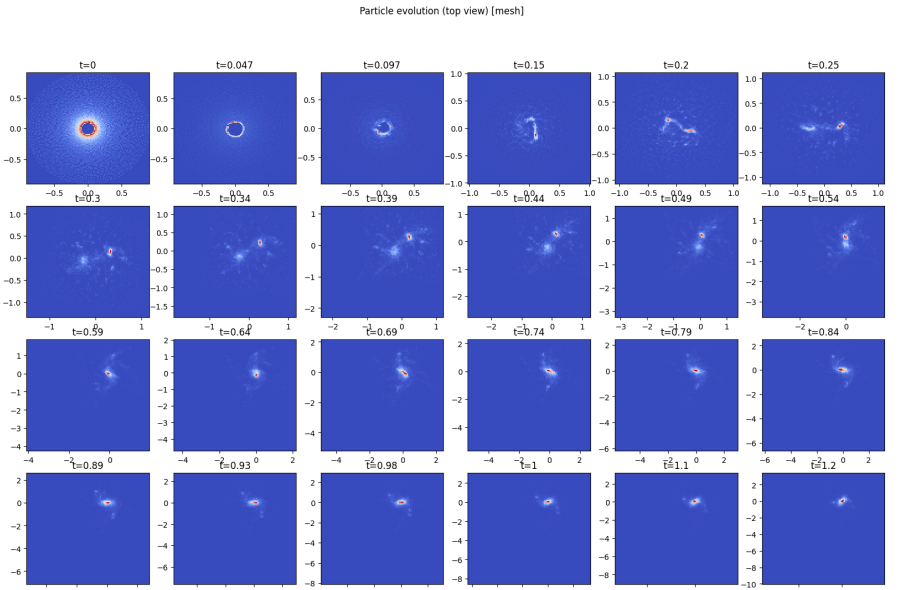


Stability [1 epsilon]

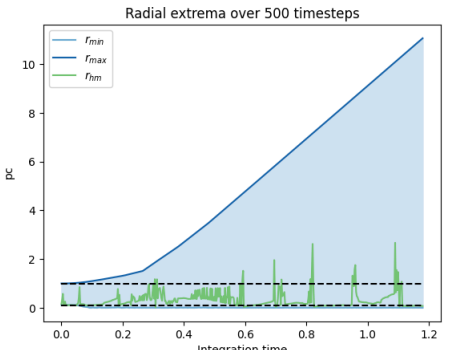


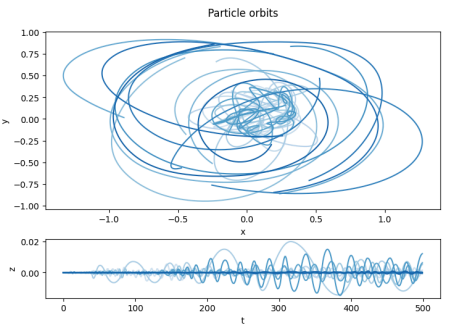


Particle mesh solver



Particle mesh solver (ii)





3 Appendix - Code

Code

```python
def apply_units(columns: np.array, quantity: str):
    if quantity == "mass":
        return columns * M_SCALE
    elif quantity == "position":
        return columns * R_SCALE
    elif quantity == "volume":
        return columns * R_SCALE**3
    elif quantity == "density":
        return columns * M_SCALE / R_SCALE**3

    ## Derived quantities
    elif quantity == "force":
        # using F = GMm/R^2 => F = M_SCALE**2 / R_SCALE**2 (G = 1)
        return columns * M_SCALE**2 / R_SCALE**2
    elif quantity == "velocity":
        # using the Virial theorem: v^2 = GM/R => v = sqrt(GM/R) => v = sqrt(M_SCALE / R_SCALE) (G = 1)
        return columns * np.sqrt(M_SCALE / R_SCALE)
    elif quantity == "time":
        # using the dynamical time: t_dyn = 1/sqrt(G*rho) => t_dyn = sqrt(4/3 * pi * R_SCALE**3 / M_SCALE) (G = 1)
        return columns * np.sqrt(4/3 * np.pi * R_SCALE**3 / M_SCALE)
    else:
        raise ValueError(f"Unknown quantity: {quantity}")
```

Code (ii)

```python
def density_distribution(r_bins: np.ndarray, particles: np.ndarray, ret_error: bool = False):
    """
    Computes the radial density distribution of a set of particles.
    Assumes that the particles array has the following columns: x, y, z, m.
    If ret_error is True, it will return the absolute error of the density.
    """
    if particles.shape[1] != 4:
        raise ValueError("Particles array must have 4 columns: x, y, z, m")

    m = particles[:, 3]
    r = np.linalg.norm(particles[:, :3], axis=1)

    m_shells = np.zeros_like(r_bins)
    v_shells = np.zeros_like(r_bins)
    error_relative = np.zeros_like(r_bins)
    r_bins = np.insert(r_bins, 0, 0)

    for i in range(len(r_bins) - 1):
        mask = (r >= r_bins[i]) & (r < r_bins[i + 1])
        m_shells[i] = np.sum(m[mask])
        v_shells[i] = 4/3 * np.pi * (r_bins[i + 1]**3 - r_bins[i]**3)
        if ret_error:
            count = np.count_nonzero(mask)
            if count > 0:
                # the absolute error is the square root of the number of particles
                error_relative[i] = 1 / np.sqrt(count)
            else:
                error_relative[i] = 0

    density = m_shells / v_shells

    if ret_error:
        return density, density * error_relative
    else:
```

Code (iii)

```
        return density
```

Code (iv)

```python
def analytical_forces(particles: np.ndarray):
    """
    Computes the interparticle forces without computing the n^2 interactions.
    This is done by using newton's second theorem for a spherical mass distribution.
    The force on a particle at radius r is simply the force exerted by a point mass with the enclosed mass.
    Assumes that the particles array has the following columns: x, y, z, m.
    """

    n = particles.shape[0]
    forces = np.zeros((n, 3))

    logger.debug(f"Computing forces for {n} particles using spherical approximation")

    r_particles = np.linalg.norm(particles[:, :3], axis=1)
    for i in range(n):
        r_current = np.linalg.norm(particles[i, 0:3])
        m_current = particles[i, 3]

        m_enclosed = np.sum(particles[r_particles < r_current, 3])

        # the force is the same as the force exerted by a point mass at the center
        f = - m_current * m_enclosed / r_current**2
        forces[i] = f

        if i % 5000 == 0:
            logger.debug(f"Particle {i} done")

    return forces
```

Code (v)

```python
def n_body_forces(particles: np.ndarray, G: float = 1, softening: float = 0):
    """
    Computes the gravitational forces between a set of particles.
    Assumes that the particles array has the following columns: x, y, z, m.
    """
    if particles.shape[1] != 4:
        raise ValueError("Particles array must have 4 columns: x, y, z, m")

    x_vec = particles[:, 0:3]
    masses = particles[:, 3]

    n = particles.shape[0]
    forces = np.zeros((n, 3))
    logger.debug(f"Computing forces for {n} particles using n^2 algorithm (using {softening=:.2g})")

    for i in range(n):
        # the current particle is at x_current
        x_current = x_vec[i, :]
        m_current = masses[i]

        # first compute the displacement to all other particles (and its magnitude)
        r_vec = x_vec - x_current
        r = np.linalg.norm(r_vec, axis=1)

        # add softening to the denominator
        r_adjusted = r**2 + softening**2
        # usually with a square root: r' = sqrt(r^2 + softening^2)
        # and then cubed, but we combine that below

        # the numerator is tricky:
        # m is a list of scalars and r_vec is a list of vectors (2D array)
        # we only want row_wise multiplication
        num = G * (masses * r_vec.T).T
        # a zero value is expected where we have the same particle
        r_adjusted[i] = 1
```

Code (vi)

```
        num[i] = 0

        f = - np.sum((num.T / r_adjusted**1.5).T, axis=0) * m_current
        forces[i] = f

        if i!= 0 and i % 5000 == 0:
            logger.debug(f"Particle {i} done")

    return forces
```

Code (vii)

```python
def mean_interparticle_distance(particles: np.ndarray):
    """
    Computes the mean interparticle distance of a set of particles.
    Assumes that the particles array has the following columns: x, y, z ...
    """
    if particles.shape[1] < 3:
        raise ValueError("Particles array must have at least 3 columns: x, y, z")


    r_half_mass = half_mass_radius(particles)
    r = np.linalg.norm(particles[:, :3], axis=1)

    n_half_mass = np.sum(r < r_half_mass)
    logger.debug(f"Number of particles within half mass radius: {n_half_mass} of {particles.shape[0]}")

    rho = n_half_mass / (4/3 * np.pi * r_half_mass**3)
    # the mean distance between particles is the inverse of the density

    epsilon = (1 / rho)**(1/3)
    logger.info(f"Found mean interparticle distance: {epsilon}")
    return epsilon
```

Code (viii)

```python
## Relaxation timescale (using the velocity at the half mass radius)

# enclosed mass at half mass radius
m_half = np.sum(particles[:, 3]) / 2
r_half = utils.half_mass_radius(particles)

# set the units
m_half = utils.apply_units(m_half, "mass")
r_half = utils.apply_units(r_half, "position")
# finally add a unit to G
true_G = 4.3e-3 * (u.pc / u.M_sun) * (u.km / u.s)**2

# crossing time
v_c = np.sqrt(true_G * m_half / r_half)
logger.info(f"Circular velocity @ HM {v_c}")
r_tot = np.max(np.linalg.norm(particles[:, :3], axis=1))
r_tot = utils.apply_units(r_tot, "position")
t_c = r_tot / v_c
t_c = t_c.to(u.Myr)
logger.info(f"Crossing time for half mass system: {t_c:.2g}")

## Using the relaxation timescale formula
n = particles.shape[0]
n_relax = n / (8 * np.log(n))
t_rel = t_c * n_relax
t_rel = t_rel.to(u.Myr)

logger.info(f"Relaxation timescale: {t_rel:.2g}")
```

Code (ix)

```python
def mesh_forces(particles: np.ndarray, G: float = 1, n_grid: int = 50, mapping: callable = None) -> np.ndarray:
    """
    Computes the gravitational force acting on a set of particles using a mesh-based approach.
    Assumes that the particles array has the following columns: x, y, z, m.
    """
    max_pos = np.max(np.abs(particles[:, :3]))
    mesh, axis, spacing = create_mesh(-max_pos, max_pos, n_grid)

    fill_mesh(particles, mesh, axis, mapping)
    # we want a density mesh:
    cell_volume = spacing**3
    rho = mesh / cell_volume

    # compute the potential and its gradient
    phi = mesh_poisson(rho, G, spacing)

    # get the acceleration from finite differences of the potential
    ax, ay, az = np.gradient(phi, spacing)
    a_vec = - np.stack([ax, ay, az], axis=0)

    # compute the particle forces from the mesh potential
    forces = np.zeros_like(particles[:, :3])
    ijks = np.digitize(particles[:, :3], axis) - 1

    for i in range(particles.shape[0]):
        m = particles[i, 3]
        idx = ijks[i]
        forces[i] = m * a_vec[..., idx[0], idx[1], idx[2]]

    return forces


def mesh_poisson(mesh: np.ndarray, G: float, spacing: float) -> np.ndarray:
    """
    Solves the poisson equation for the mesh using the FFT.
```

Code (x)

```
    Returns the the potential - phi
    """
    rho_hat = fft.fftn(mesh)

    # we also need the wave numbers
    k = fft.fftfreq(mesh.shape[0], spacing) * (2 * np.pi)
    # assuming the grid is cubic
    kx, ky, kz = np.meshgrid(k, k, k)
    k_sr = kx**2 + ky**2 + kz**2

    k_sr[k_sr == 0] = np.inf

    phi_hat = - 4 * np.pi * G * rho_hat / k_sr
    return np.real(fft.ifftn(phi_hat))
```

Code (xi)

```
## Integration setup

# Determine the integration timesteps
v = np.linalg.norm(particles[:, 3:6], axis=1)
v_mean = np.mean(v)
# a timestep should result in a small displacement, wrt. to the mean interparticle distance
r_inter = utils.mean_interparticle_distance(particles)

dt = r_inter / v_mean * 1e-3
logger.info(f"Mean velocity: {v_mean}, timestep: {dt}")
if np.isnan(dt):
    raise ValueError("Invalid timestep")

# Determine the integration range
t_orbit = 2 * np.pi * r_inter / v_mean
n_steps = int(t_orbit / dt * 5)
# in theory we should integrate for a few orbits, but we only do a few steps as a POC
n_steps = 24
t_range = np.arange(0, n_steps*dt, dt)
logger.info(f"Integration range: {t_range[0]} -> {t_range[-1]}, n_steps: {n_steps}")
```

Code (xii)

```python
def integrate(method: str, force_function: callable, p0: np.ndarray, t_range: np.ndarray) -> np.ndarray:
    """
    Integrate the gravitational movement of the particles, using the specified method
    - method: the integration method to use ("scipy" or "rk4")
    - force_function: the function that computes the forces acting on the particles
    - p0: the initial conditions of the particles (n, 7) array, unflattened
    - t_range: the time range to integrate over
    Returns: the integrated positions and velocities of the particles in a 'flattened' array (time_steps, nx7)
    """
    y0, y_prime = utils.ode_setup(p0, force_function)

    if method == "scipy":
        sol = spi.odeint(y_prime, y0, t_range, rtol=1e-2)
    elif method == "rk4":
        sol = np.zeros((t_range.shape[0], y0.shape[0]))
        sol[0] = y0
        dt = t_range[1] - t_range[0]
        for i in range(1, t_range.shape[0]):
            t = t_range[i]
            sol[i,...] = utils.runge_kutta_4(sol[i-1], t, y_prime, dt)

            if i % 100 == 0:
                logger.info(f"Integration step {i} done")

    logger.info(f"Integration done, shape: {sol.shape}")
    return sol
```