

N-Body project

Computational Astrophysics, HS24

04.02.2024

Rémy Moll

1 N-body forces and analytical solutions

Objective



Implement naive N-body force computation and get an intuition of the challenges:

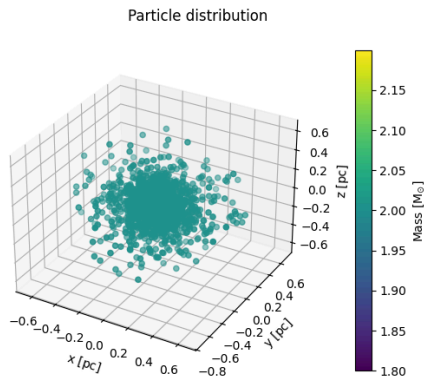
- accuracy
- computation time
- stability

⇒ still useful to compute basic quantities of the system, but too limited for large systems or the dynamical evolution of the system

Overview - the system



Get a feel for the particles and their distribution. [code]



The system at hand is characterized by:

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

⇒ treat the system as a **globular cluster**

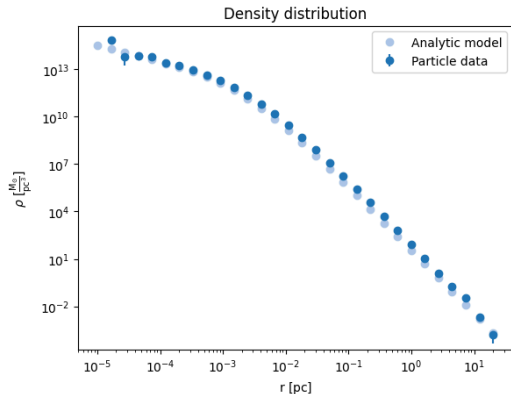
Note: for visibility the outer particles are not shown.

Density

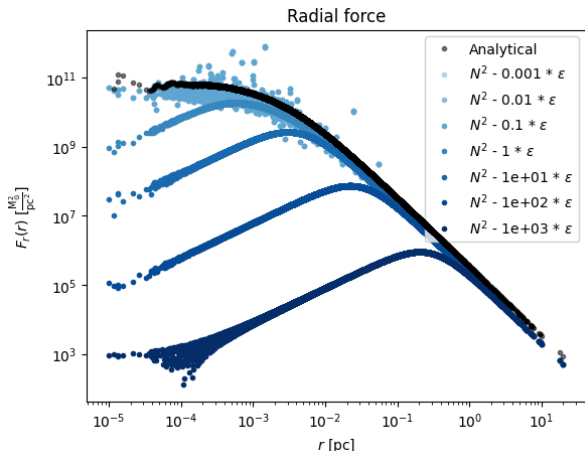


We compare the computed density with the analytical model provided by the *Hernquist* model:

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



Force computation



Discussion:

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

Analytical force [code]; N^2 force [code]; ϵ computation [code];

Relaxation



Relaxation [code]:

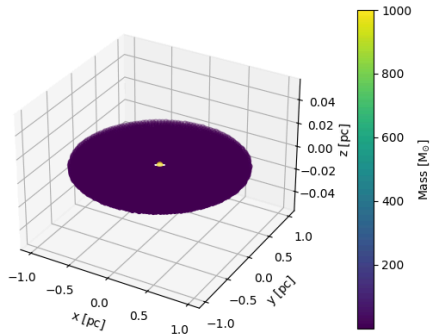
Discussion!

2 Particle Mesh

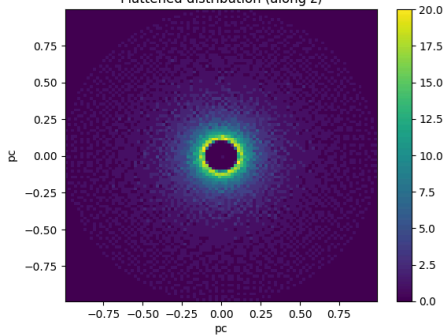
Overview - the system



Particle distribution



Flattened distribution (along z)



Force computation



```
def mesh_forces(particles: np.ndarray, G: float, n_grid: int, mapping: callable) -> 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.
    """
    if particles.shape[1] != 4:
        raise ValueError("Particles array must have 4 columns: x, y, z, m")

    logger.debug(f"Computing forces for {particles.shape[0]} particles using mesh
[mapping={mapping.__name__}, {n_grid=}]")

    mesh, axis = to_mesh(particles, n_grid, mapping)
    spacing = np.abs(axis[1] - axis[0])
    logger.debug(f"Using mesh spacing: {spacing}")

    # we want a density mesh:
    cell_volume = spacing**3
    rho = mesh / cell_volume

    if logger.isEnabledFor(logging.DEBUG):
        show_mesh_information(mesh, "Density mesh")

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

    if logger.isEnabledFor(logging.DEBUG):
        logger.debug(f"Got phi_grad with: {phi_grad.shape}, {np.max(phi_grad)}")
        show_mesh_information(phi_grad[0], "Potential gradient (x-direction)")
```

Force computation (ii)



```
# compute the particle forces from the mesh potential
forces = np.zeros_like(particles[:, :3])
for i, p in enumerate(particles):
    ijk = np.digitize(p, axis) - 1
    logger.debug(f"Particle {p} maps to cell {ijk}")
    # this gives 4 entries since p[3] the mass is digitized as well -> this is meaningless and we
discard it
    # logger.debug(f"Particle {p} maps to cell {ijk}")
    forces[i] = - p[3] * phi_grad[..., ijk[0], ijk[1], ijk[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.
    Returns the derivative of the potential - grad phi
    """
    rho_hat = fft.fftn(mesh)
    k = fft.fftfreq(mesh.shape[0], spacing)
    # shift the zero frequency to the center
    k = fft.fftshift(k)

    kx, ky, kz = np.meshgrid(k, k, k)
    k_vec = np.stack([kx, ky, kz], axis=0)
    k_sr = kx**2 + ky**2 + kz**2
    if logger.isEnabledFor(logging.DEBUG):
        logger.debug(f"Got k_square with: {k_sr.shape}, {np.max(k_sr)} {np.min(k_sr)}")
        logger.debug(f"Count of ksquare zeros: {np.sum(k_sr == 0)}")
    show_mesh_information(np.abs(k_sr), "k_square")
```

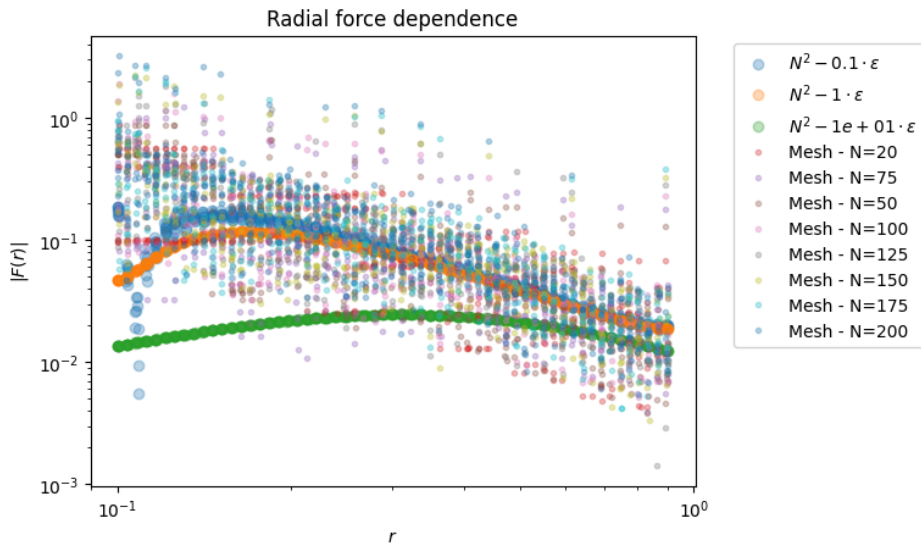
Force computation (iii)



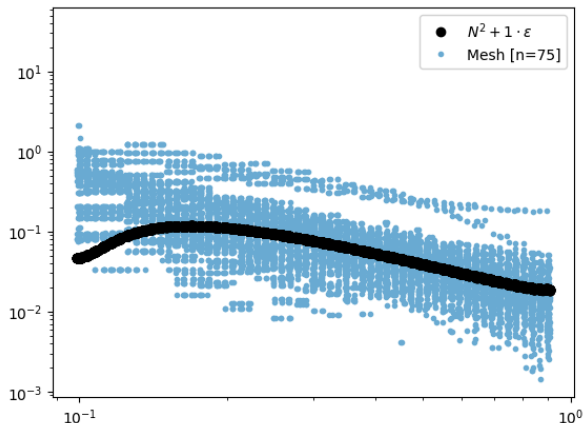
```
k_sr[k_sr == 0] = np.inf
k_inv = k_vec / k_sr # allows for element-wise division

logger.debug(f"Proceeding to poisson equation with {rho_hat.shape=}, {k_inv.shape=}")
grad_phi_hat = - 4 * np.pi * G * rho_hat * k_inv * 1j
# nabla^2 phi => -i * k * nabla phi = 4 pi G rho => nabla phi = - i * rho * k / k^2
# todo: check minus
grad_phi = np.real(fft.ifftn(grad_phi_hat))
return grad_phi
```

Force computation (iv)



Force computation (v)



Discussion:

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

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

N^2 force [code]; ε computation [code]; Mesh force [code];

Time integration



2.c.a Runge-Kutta

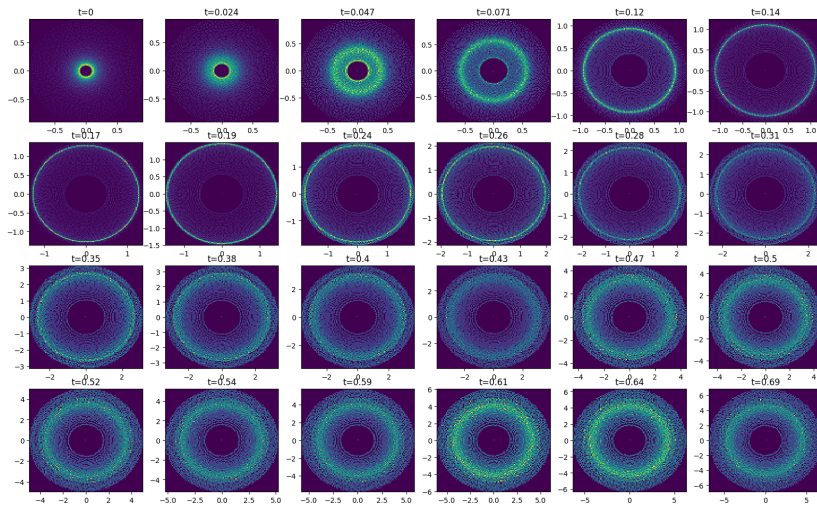
```
def runge_kutta_4(y: np.ndarray, t: float, f: callable, dt: float):  
    """  
    Runge-Kutta 4th order integrator.  
    """  
    k1 = f(y, t)  
    k2 = f(y + k1/2 * dt, t + dt/2)  
    k3 = f(y + k2/2 * dt, t + dt/2)  
    k4 = f(y + k3 * dt, t + dt)  
    return y + (k1 + 2*k2 + 2*k3 + k4)/6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

Time integration (ii)

2.c.b Results



Particle evolution (top view)



Particle mesh solver

sdlsd



3 Appendix - Code

Code



```
# Set G = 1
G = 1

# Since we have a globular cluster, we can use typical values
M_TOT = 1e5 * u.M_sun
R_TOT = 20 * u.pc

# Rescale the units of the particles
M_particles = particles[:,3].sum()
R_particles = np.max(np.linalg.norm(particles[:, :3], axis=1))
logger.info(f"Considering a globular cluster - total mass of particles: {M_particles}, maximum radius of
particles: {R_particles}")
m_scale = M_TOT / M_particles
r_scale = R_TOT / R_particles
utils.seed_scales(r_scale, m_scale)

### Plot again with scales
reduced = utils.remove_outliers(particles.copy())
positions = utils.apply_units(reduced[:, :3], "position")
masses = utils.apply_units(reduced[:, 3], "mass")
utils.plot_particles_3d(positions, masses, title="Particle distribution")
```

Code (ii)



```
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 (iii)



```
def n_body_forces(particles: np.ndarray, G: float, 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
```

Code (iv)



```
# 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
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 (v)



```
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
# TODO: check if this is correct
```

Code (vi)



```
## Computes the relaxation timescale of a set of particles 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")

v_c = np.sqrt(G * m_half / r_half)
logger.info(f"Central velocity @ HM {v_c}")

t_c = r_half / v_c
logger.info(f"Crossing time for half mass system: {t_c:.2g}")
# TODO: how to treat this unit?

## Using the derived formula for the relaxation timescale
# Compute the relaxation timescale through the estimate
# t_relax = t_c * n_relax = t_c * N / (10 * log(N))
n = particles.shape[0]
n_relax = n / (10 * np.log(n))
t_rel = t_c * n_relax
logger.info(f"Direct estimate of the relaxation timescale: {t_rel:.2g}")
```


Code (vii)



```
def mesh_forces(particles: np.ndarray, G: float, n_grid: int, mapping: callable) -> 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.
    """

    if particles.shape[1] != 4:
        raise ValueError("Particles array must have 4 columns: x, y, z, m")

    logger.debug(f"Computing forces for {particles.shape[0]} particles using mesh
[mapping={mapping.__name__}, {n_grid=}]")

    mesh, axis = to_mesh(particles, n_grid, mapping)
    spacing = np.abs(axis[1] - axis[0])
    logger.debug(f"Using mesh spacing: {spacing}")

    # we want a density mesh:
    cell_volume = spacing**3
    rho = mesh / cell_volume

    if logger.isEnabledFor(logging.DEBUG):
        show_mesh_information(mesh, "Density mesh")

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

    if logger.isEnabledFor(logging.DEBUG):
        logger.debug(f"Got phi_grad with: {phi_grad.shape}, {np.max(phi_grad)}")
        show_mesh_information(phi_grad[0], "Potential gradient (x-direction)")
```

Code (viii)



```
# compute the particle forces from the mesh potential
forces = np.zeros_like(particles[:, :3])
for i, p in enumerate(particles):
    ijk = np.digitize(p, axis) - 1
    logger.debug(f"Particle {p} maps to cell {ijk}")
    # this gives 4 entries since p[3] the mass is digitized as well -> this is meaningless and we
discard it
    # logger.debug(f"Particle {p} maps to cell {ijk}")
    forces[i] = - p[3] * phi_grad[..., ijk[0], ijk[1], ijk[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.
    Returns the derivative of the potential - grad phi
    """
    rho_hat = fft.fftn(mesh)
    k = fft.fftfreq(mesh.shape[0], spacing)
    # shift the zero frequency to the center
    k = fft.fftshift(k)

    kx, ky, kz = np.meshgrid(k, k, k)
    k_vec = np.stack([kx, ky, kz], axis=0)
    k_sr = kx**2 + ky**2 + kz**2
    if logger.isEnabledFor(logging.DEBUG):
        logger.debug(f"Got k_square with: {k_sr.shape}, {np.max(k_sr)} {np.min(k_sr)}")
        logger.debug(f"Count of ksquare zeros: {np.sum(k_sr == 0)}")
    show_mesh_information(np.abs(k_sr), "k_square")
```

Code (ix)



```
k_sr[k_sr == 0] = np.inf
k_inv = k_vec / k_sr # allows for element-wise division

logger.debug(f"Proceeding to poisson equation with {rho_hat.shape=}, {k_inv.shape=}")
grad_phi_hat = - 4 * np.pi * G * rho_hat * k_inv * 1j
# nabla^2 phi => -i * k * nabla phi = 4 pi G rho => nabla phi = - i * rho * k / k^2
# todo: check minus
grad_phi = np.real(fft.ifftn(grad_phi_hat))
return grad_phi
```