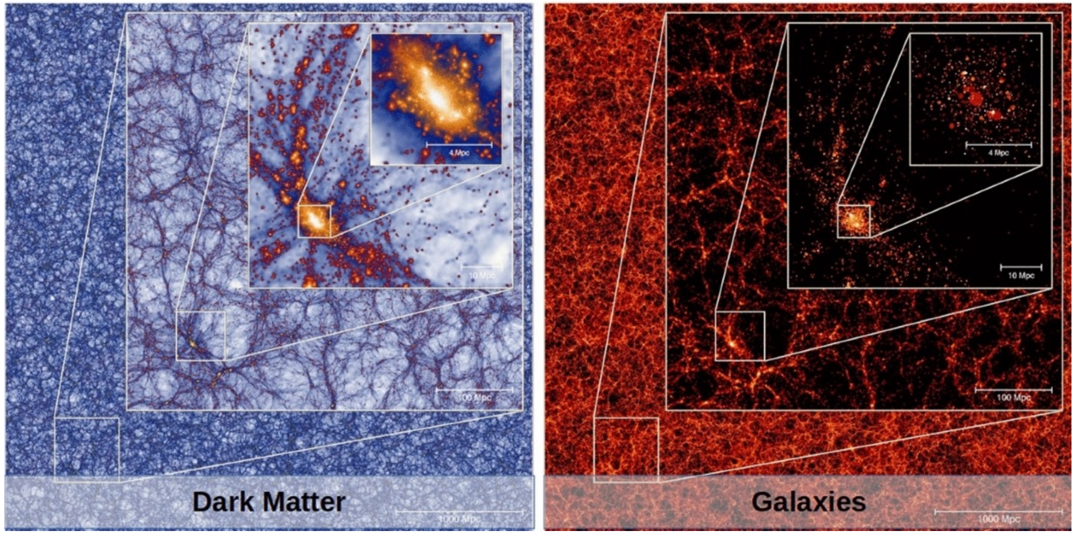
ASTR4004/8004 Computational Astronomy

ASTRONOMICAL SIMULATIONS

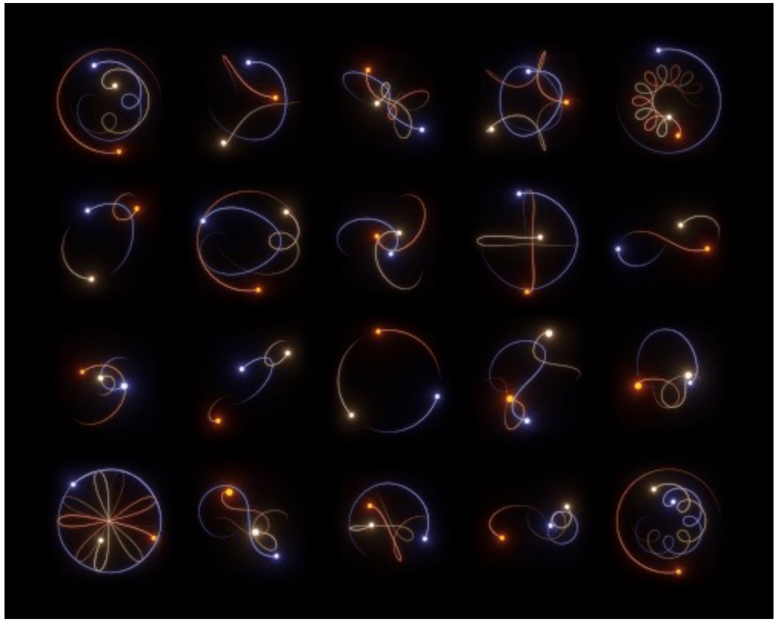
Yuxiang.Qin@anu.edu.au
Duffield Building D.115

Australian National University



N-body simulation

Isaac Newton gained renown for solving the two-body problem, showing how gravitational attraction binds the Earth and the Sun into elliptical orbits.



Moving on to the chaotic 3-body problem:

Leonhard Euler studied solutions when the three masses are in the same line.

Joseph-Louis Lagrange later discovered special solutions when the three masses form an equilateral triangle.

Then we have computers…

N-body simulation Pseudocode

```
def n_body_simulation(N, dt, total_time):

    positions = np.random.rand(N, 3) * volume_size     # initialize positions
    velocities = np.random.rand(N, 3) * velocity_scale # initialize velocities
    masses = np.ones(N) * mass_scale                   # initialize masses

    for step in range(int(total_time / dt)):  # loop through all steps
        forces = calculate_forces(positions, N) # gravity!

        # Update velocities and position based on forces
        for i in range(N):
           velocities[i] += forces[i] / masses[i] * dt     # v_new = v_old + (F/m) * dt
           positions[i]  += velocities[i] * dt             # r_new = r_old + v * dt
           positions     %= box_size                       # periodic boundary conditions

    return positions, velocities
```

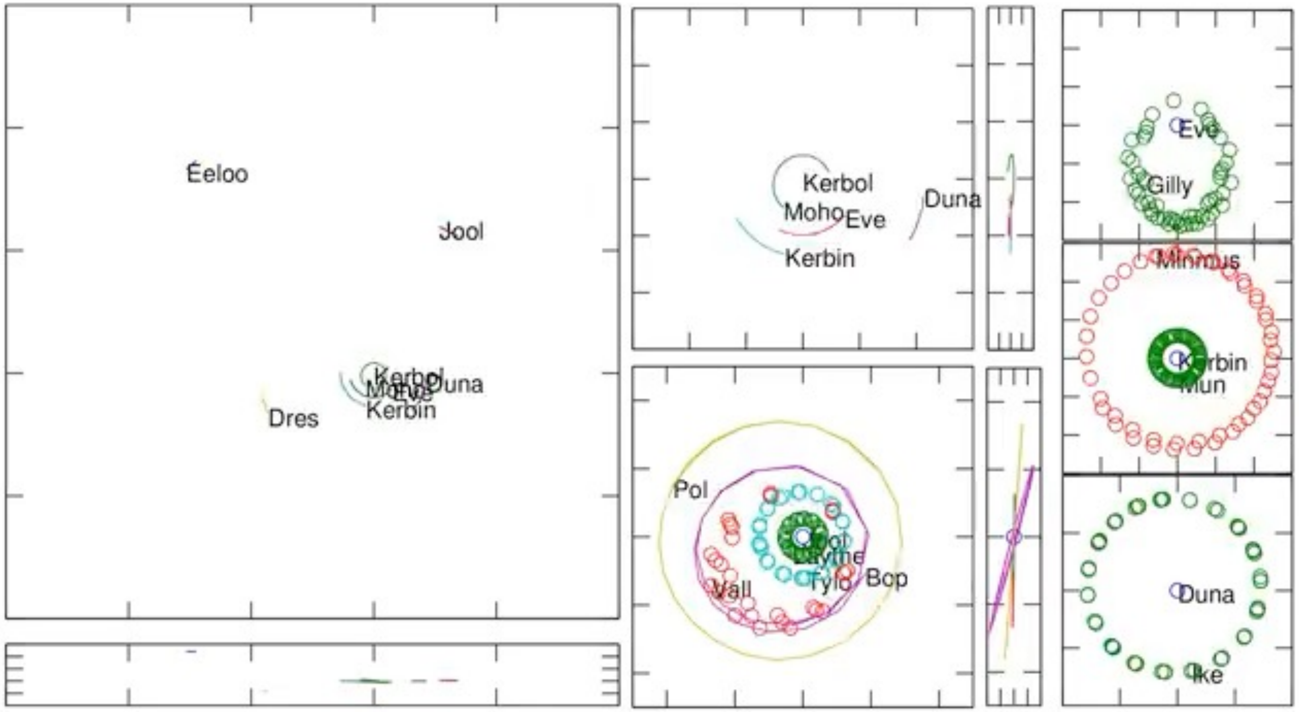
N-body simulation Pseudocode

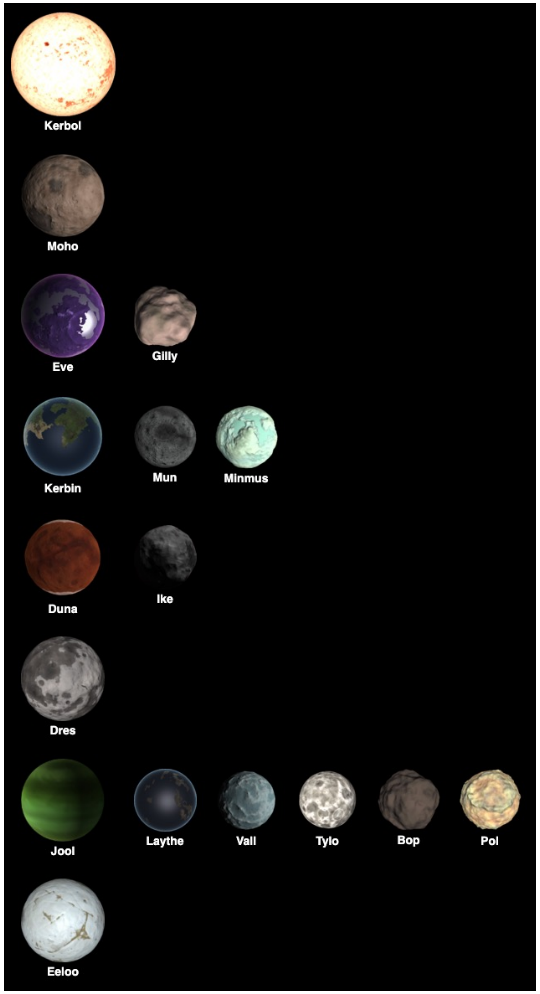
$$\ddot{\vec{r}}_j = -G \sum_{i \neq j}^{N} \frac{m_i}{\left|\vec{r_i} - \vec{r}_j\right|^3} (\vec{r_i} - \vec{r}_j)$$

```
def calculate_forces(positions, N):
    forces = np.zeros((N, 3)) # Initialize force array

    for i in range(N):
        for j in range(i + 1, N):
            # Calculate gravitational force between particles i and j
            distance_vector = positions[j] - positions[i]
            distance = np.linalg.norm(distance_vector)
            force_magnitude = G * (masses[i] * masses[j]) / (distance ** 2 + soften**2)
            force_vector = force_magnitude * distance_vector / distance
            forces[i] += force_vector  # Force on particle i
            forces[j] -= force_vector  # Equal and opposite force on particle j

    return forces
```

Planetary system

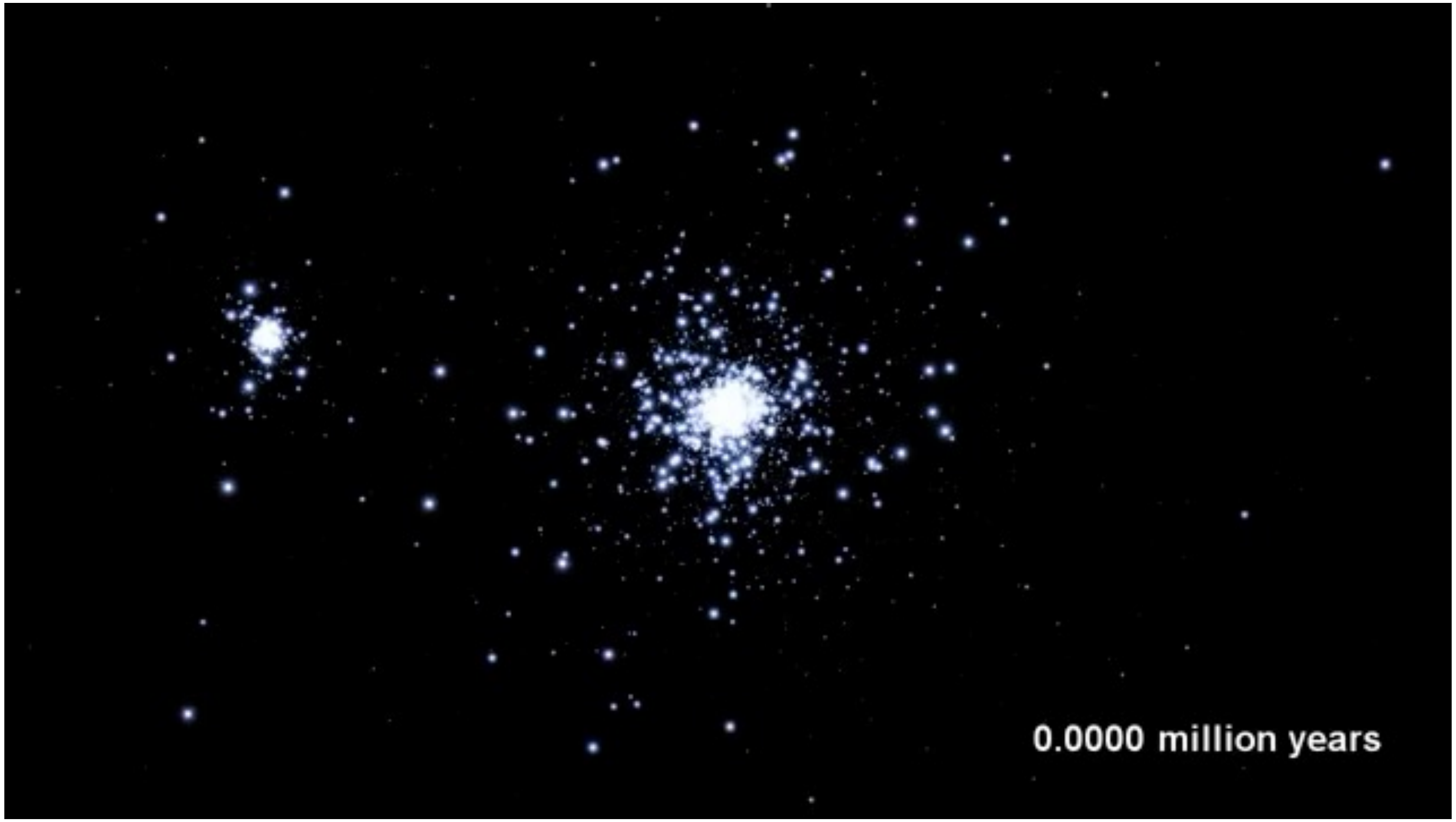
Kerbal Space Program





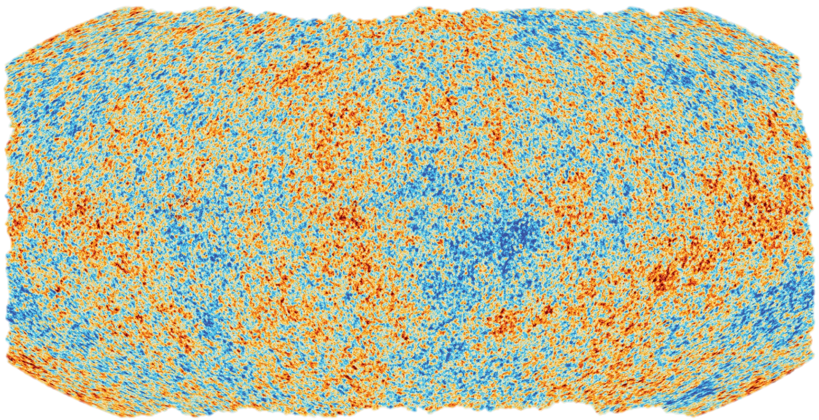
Star cluster

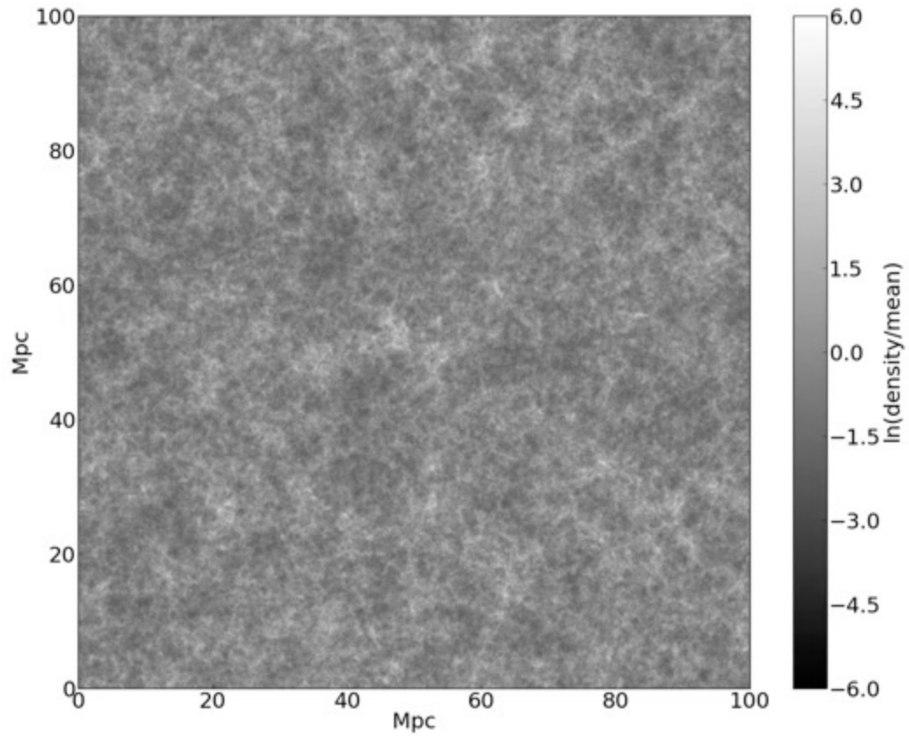
Stellar group R136 in the 30 Doradus nebula, in LMC



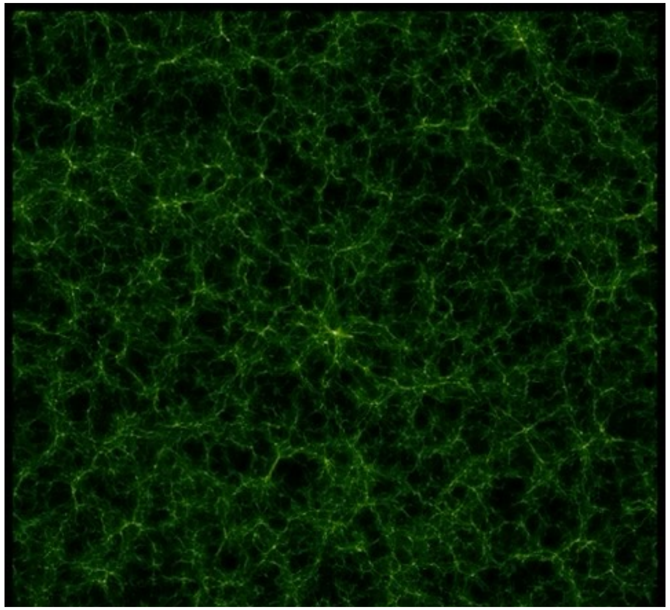
Andromeda colliding with the Milky Way

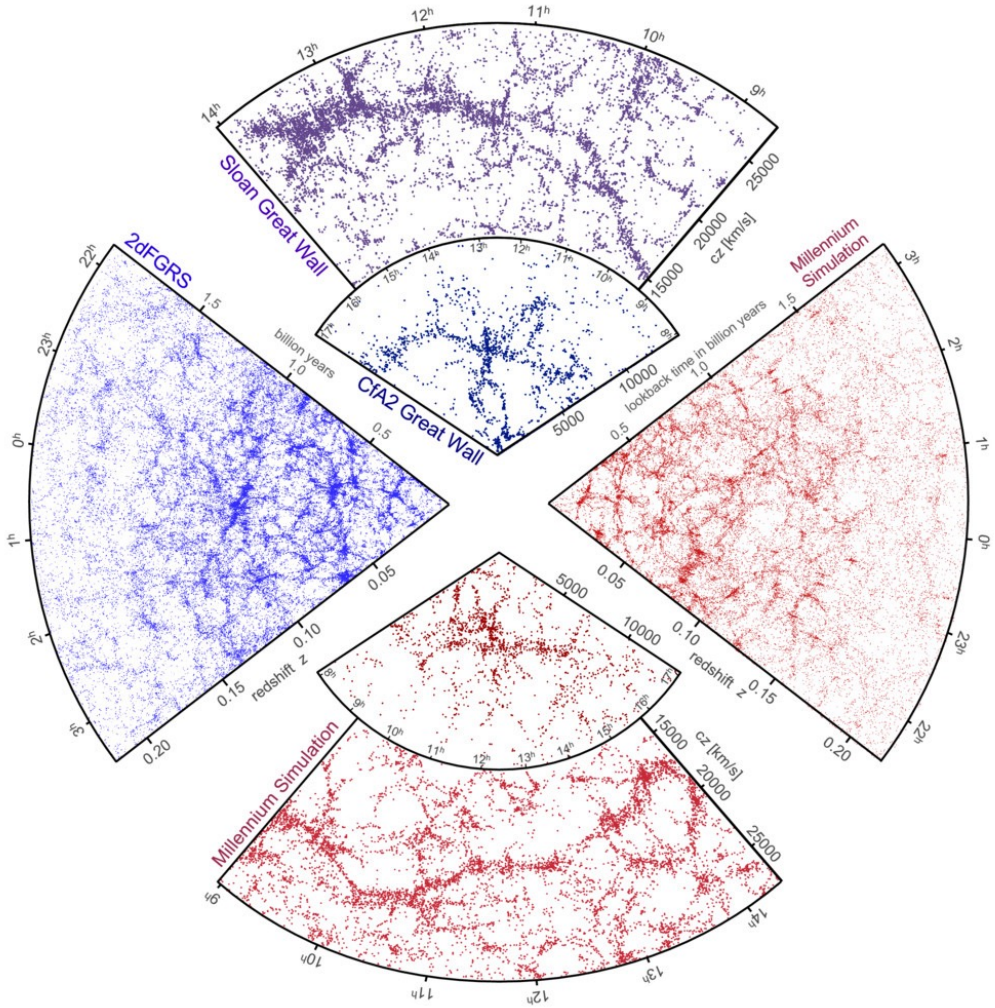
Cosmological N-body simulation





Cosmological N-body simulations



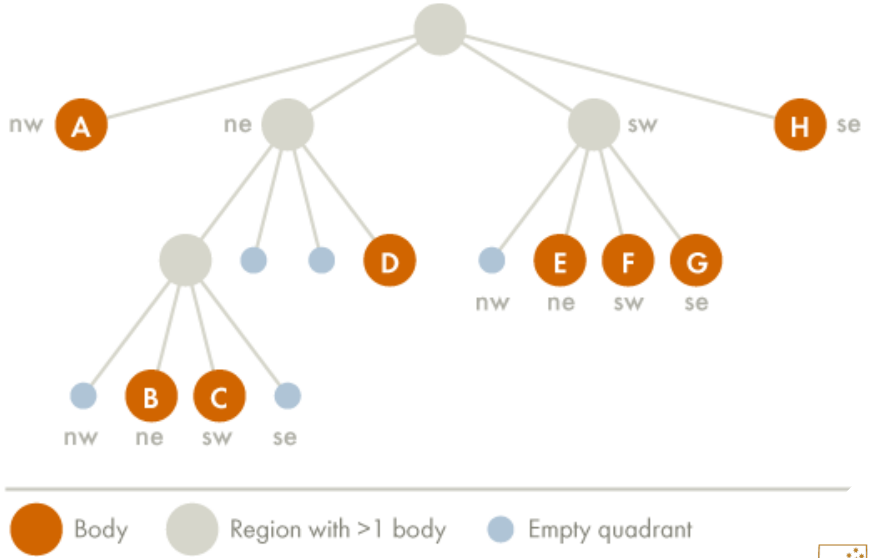


open simulations-part1.ipynb

The Barnes-Hut Algorithm

When particles are sufficiently far away, we can approximate their gravitational force as a group using their centre of *mass*.

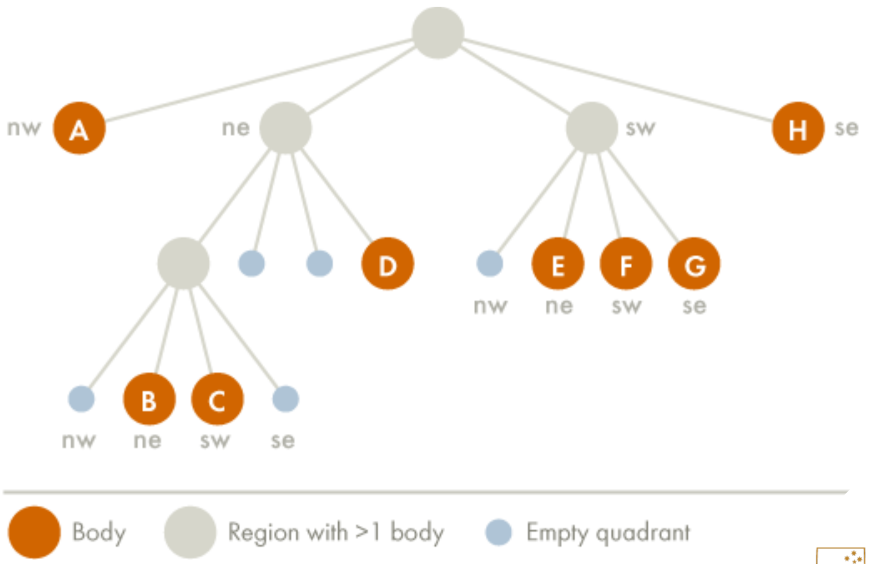
Octree structure: the computational domain is hierarchically partitioned into a sequence of cubes, where each cube contains 8 siblings & each with half the side-length of the parent cube.



The Barnes-Hut Algorithm

-- Domain Decomposition

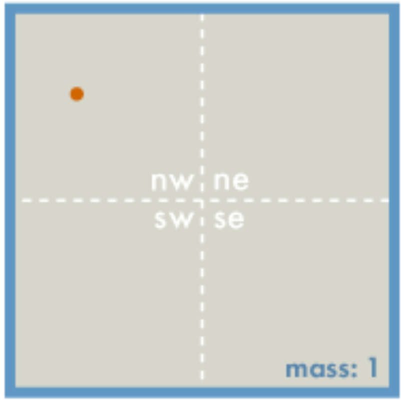
```
class Node:
    box, mass, com, children, particles, is_leaf

def octant(center, pos):
    # which of 8 sub-cubes (x,y,z) belongs to
    return (x>=cx) + 2*(y>=cy) + 4*(z>=cz)

def build_tree(positions, masses):
    root = Node(bounding_box_of_all)
    for each particle p:
        insert(root, p)
    compute_mass(root)
    return root

def insert(node, p):
    if node.is_leaf and few_particles:
        add p to node
    else:
        if node.is_leaf: subdivide(node into 8)
        child = node.children[ octant(node.center, p.pos) ]
        insert(child, p)

def compute_mass(node):
    if node.is_leaf:
        node.mass = Σ masses
        node.com  = weighted_average(positions)
    else:
        for each child: compute_mass(child)
        node.mass = Σ child.mass
        node.com  = Σ(child.mass*child.com) / node.mass
```

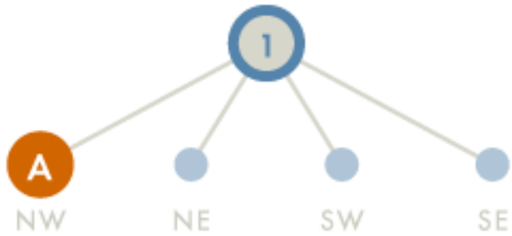


The Barnes-Hut Algorithm

-- Domain Decomposition

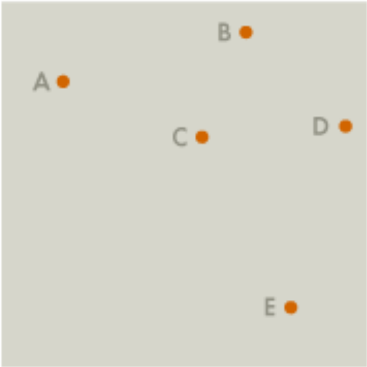

```
class Node:
    box, mass, com, children, particles, is_leaf

def octant(center, pos):
    # which of 8 sub-cubes (x,y,z) belongs to
    return (x>=cx) + 2*(y>=cy) + 4*(z>=cz)

def build_tree(positions, masses):
    root = Node(bounding_box_of_all)
    for each particle p:
        insert(root, p)
    compute_mass(root)
    return root

def insert(node, p):
    if node.is_leaf and few_particles:
        add p to node
    else:
        if node.is_leaf: subdivide(node into 8)
        child = node.children[ octant(node.center, p.pos) ]
        insert(child, p)

def compute_mass(node):
    if node.is_leaf:
        node.mass = Σ masses
        node.com  = weighted_average(positions)
    else:
        for each child: compute_mass(child)
        node.mass = Σ child.mass
        node.com  = Σ(child.mass*child.com) / node.mass
```



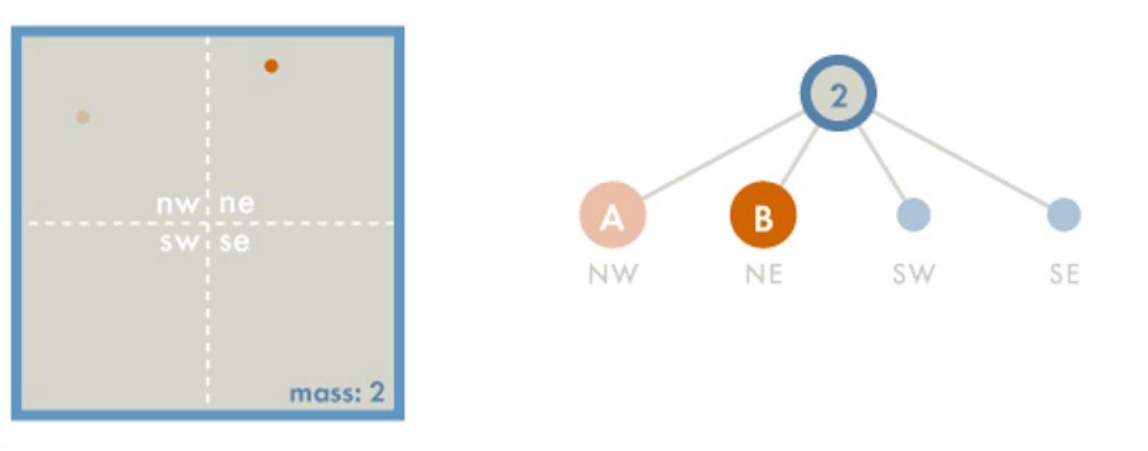


The Barnes-Hut Algorithm

-- Domain Decomposition



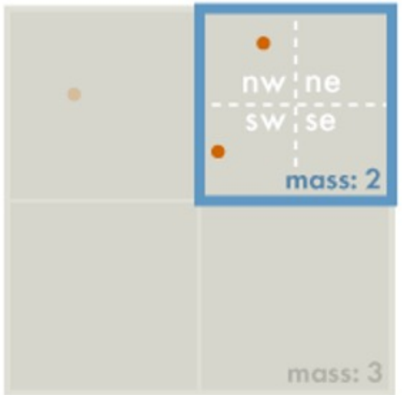

```
class Node:
    box, mass, com, children, particles, is_leaf

def octant(center, pos):
    # which of 8 sub-cubes (x,y,z) belongs to
    return (x>=cx) + 2*(y>=cy) + 4*(z>=cz)

def build_tree(positions, masses):
    root = Node(bounding_box_of_all)
    for each particle p:
        insert(root, p)
    compute_mass(root)
    return root

def insert(node, p):
    if node.is_leaf and few_particles:
        add p to node
    else:
        if node.is_leaf: subdivide(node into 8)
        child = node.children[ octant(node.center, p.pos) ]
        insert(child, p)

def compute_mass(node):
    if node.is_leaf:
        node.mass = Σ masses
        node.com  = weighted_average(positions)
    else:
        for each child: compute_mass(child)
        node.mass = Σ child.mass
        node.com  = Σ(child.mass*child.com) / node.mass
```

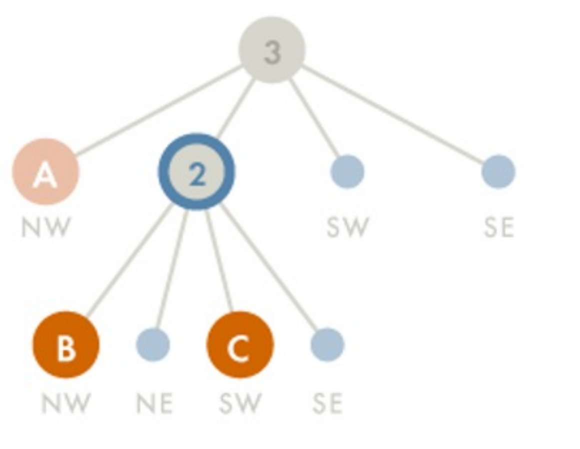


The Barnes-Hut Algorithm

-- Domain Decomposition

```
class Node:
    box, mass, com, children, particles, is_leaf

def octant(center, pos):
    # which of 8 sub-cubes (x,y,z) belongs to
    return (x>=cx) + 2*(y>=cy) + 4*(z>=cz)

def build_tree(positions, masses):
    root = Node(bounding_box_of_all)
    for each particle p:
        insert(root, p)
    compute_mass(root)
    return root

def insert(node, p):
    if node.is_leaf and few_particles:
        add p to node
    else:
        if node.is_leaf: subdivide(node into 8)
        child = node.children[ octant(node.center, p.pos) ]
        insert(child, p)

def compute_mass(node):
    if node.is_leaf:
        node.mass = Σ masses
        node.com  = weighted_average(positions)
    else:
        for each child: compute_mass(child)
        node.mass = Σ child.mass
        node.com  = Σ(child.mass*child.com) / node.mass
```

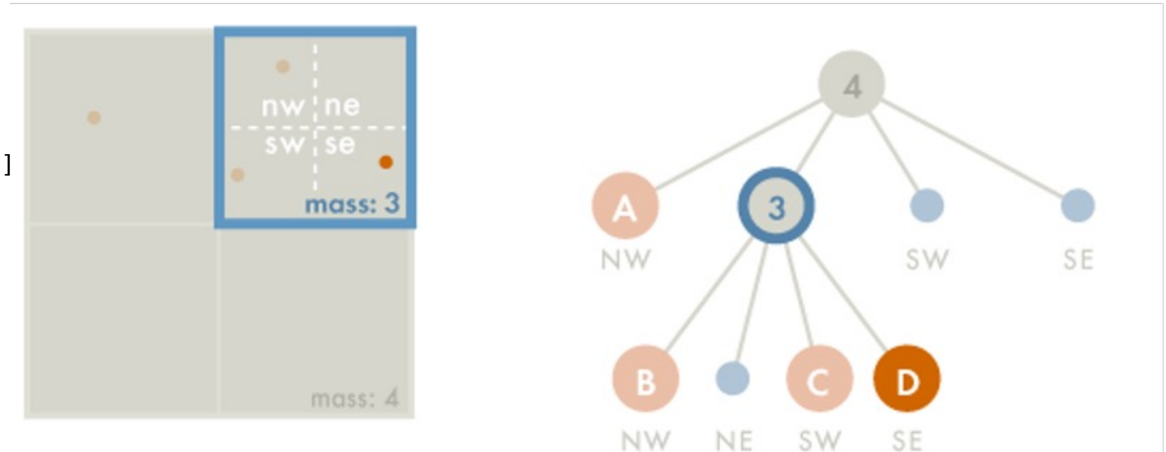




The Barnes-Hut Algorithm

-- Domain Decomposition

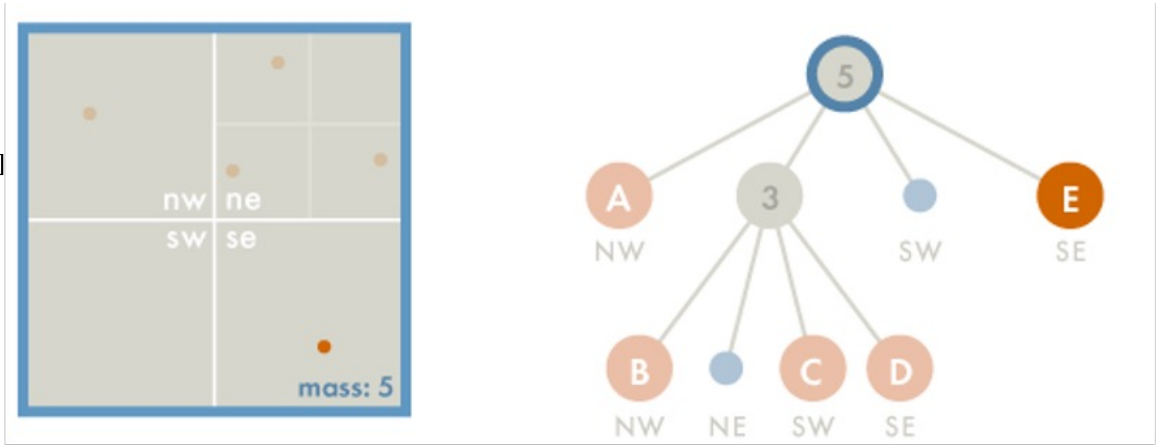

```
class Node:
    box, mass, com, children, particles, is_leaf

def octant(center, pos):
    # which of 8 sub-cubes (x,y,z) belongs to
    return (x>=cx) + 2*(y>=cy) + 4*(z>=cz)

def build_tree(positions, masses):
    root = Node(bounding_box_of_all)
    for each particle p:
        insert(root, p)
    compute_mass(root)
    return root

def insert(node, p):
    if node.is_leaf and few_particles:
        add p to node
    else:
        if node.is_leaf: subdivide(node into 8)
        child = node.children[ octant(node.center, p.pos) ]
        insert(child, p)

def compute_mass(node):
    if node.is_leaf:
        node.mass = Σ masses
        node.com  = weighted_average(positions)
    else:
        for each child: compute_mass(child)
        node.mass = Σ child.mass
        node.com  = Σ(child.mass*child.com) / node.mass
```



The Barnes-Hut Algorithm

-- Domain Decomposition

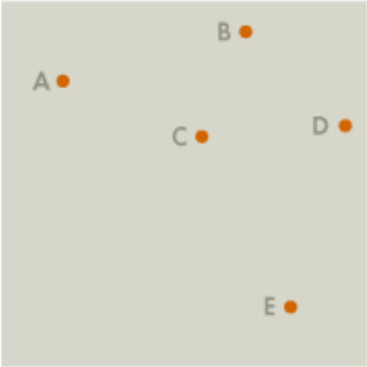

```
class Node:
    box, mass, com, children, particles, is_leaf

def octant(center, pos):
    # which of 8 sub-cubes (x,y,z) belongs to
    return (x>=cx) + 2*(y>=cy) + 4*(z>=cz)

def build_tree(positions, masses):
    root = Node(bounding_box_of_all)
    for each particle p:
        insert(root, p)
    compute_mass(root)
    return root

def insert(node, p):
    if node.is_leaf and few_particles:
        add p to node
    else:
        if node.is_leaf: subdivide(node into 8)
        child = node.children[ octant(node.center, p.pos) ]
        insert(child, p)

def compute_mass(node):
    if node.is_leaf:
        node.mass = Σ masses
        node.com  = weighted_average(positions)
    else:
        for each child: compute_mass(child)
        node.mass = Σ child.mass
        node.com  = Σ(child.mass*child.com) / node.mass
```

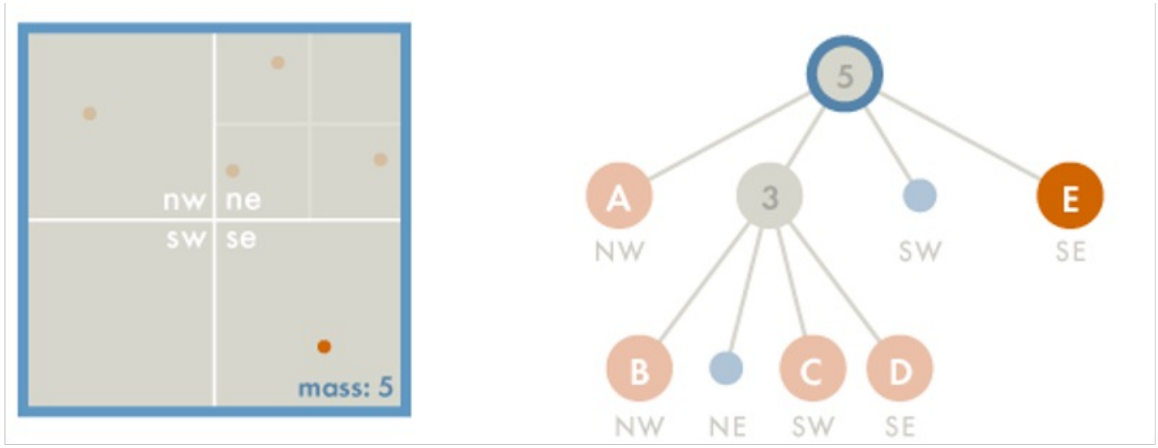


The Barnes-Hut Algorithm

-- Calculate Forces

```python
def calculate_forces(pos, mass, G, softening):
    acc  = zeros_like(pos)
    root = build_octree(pos, mass)
    soft2 = softening**2

    def add_from_node(i, node):
        if node.mass == 0: return

        if node.is_leaf:
            for j in node.idx:
                if j == i: continue
                dr = pos[j] - pos[i]
                dist = norm(r)
                inv = 1 / (dist**2 + soft2)**1.5
                acc[i] += G * mass[j] * dr * inv
        else:

            dr = node.com - pos[i]
            dist = norm(r)

            if node.size()/dist < theta:
                inv = 1 / (dist**3 + soft2**1.5)
                acc[i] += G * node.mass * dr * inv
            else:
                for ch in node.children:
                    add_from_node(i, ch)

    for i in range(len(pos)):
        add_from_node(i, root)

    return acc
```

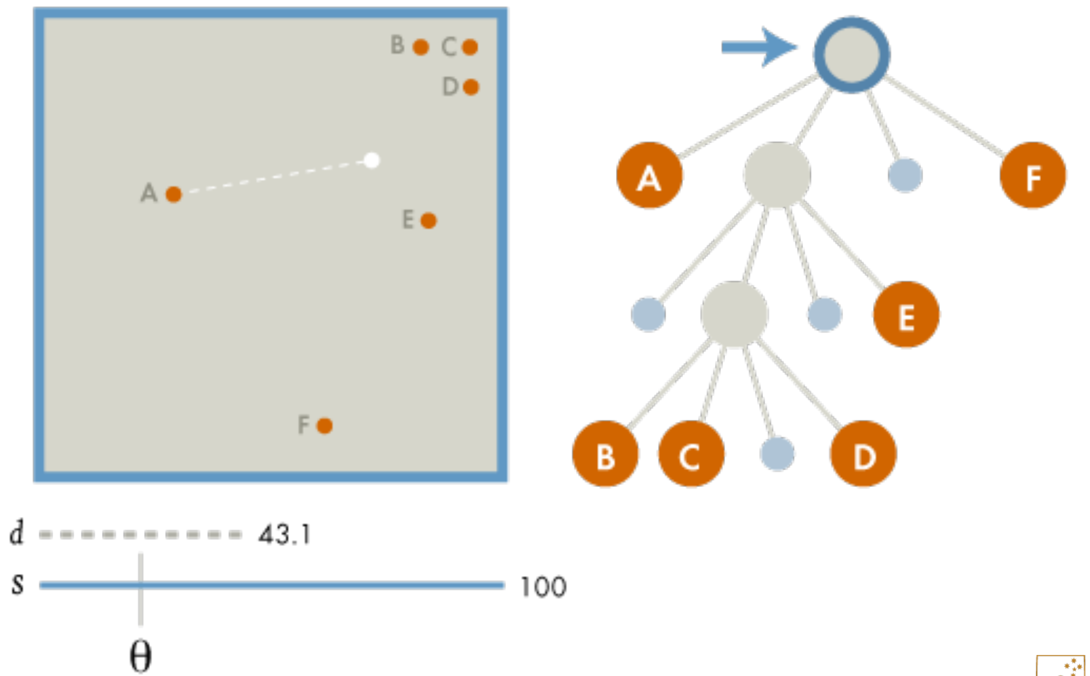




The Barnes-Hut Algorithm

-- Calculate Forces

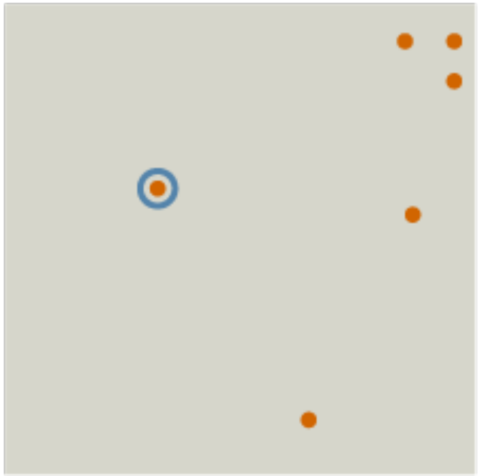
```
def calculate_forces(pos, mass, G, softening):
    acc  = zeros_like(pos)
    root = build_octree(pos, mass)
    soft2 = softening**2

    def add_from_node(i, node):
        if node.mass == 0: return

        if node.is_leaf:
            for j in node.idx:
                if j == i: continue
                dr = pos[j] - pos[i]
                dist = norm(r)
                inv = 1 / (dist**2 + soft2)**1.5
                acc[i] += G * mass[j] * dr * inv
        else:

            dr = node.com - pos[i]
            dist = norm(r)

            if node.size()/dist < theta:
                inv = 1 / (dist**3 + soft2**1.5)
                acc[i] += G * node.mass * dr * inv
            else:
                for ch in node.children:
                    add_from_node(i, ch)

    for i in range(len(pos)):
        add_from_node(i, root)

    return acc
```

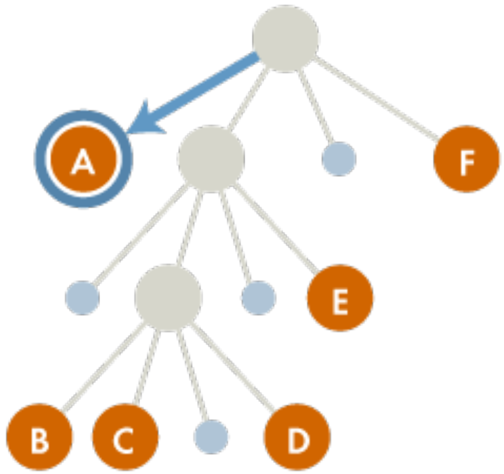


The Barnes-Hut Algorithm

-- Calculate Forces


```
def calculate_forces(pos, mass, G, softening):
    acc  = zeros_like(pos)
    root = build_octree(pos, mass)
    soft2 = softening**2

    def add_from_node(i, node):
        if node.mass == 0: return

        if node.is_leaf:
            for j in node.idx:
                if j == i: continue
                dr = pos[j] - pos[i]
                dist = norm(r)
                inv = 1 / (dist**2 + soft2)**1.5
                acc[i] += G * mass[j] * dr * inv
        else:

            dr = node.com - pos[i]
            dist = norm(r)

            if node.size()/dist < theta:
                inv = 1 / (dist**3 + soft2**1.5)
                acc[i] += G * node.mass * dr * inv
            else:
                for ch in node.children:
                    add_from_node(i, ch)

    for i in range(len(pos)):
        add_from_node(i, root)

    return acc
```

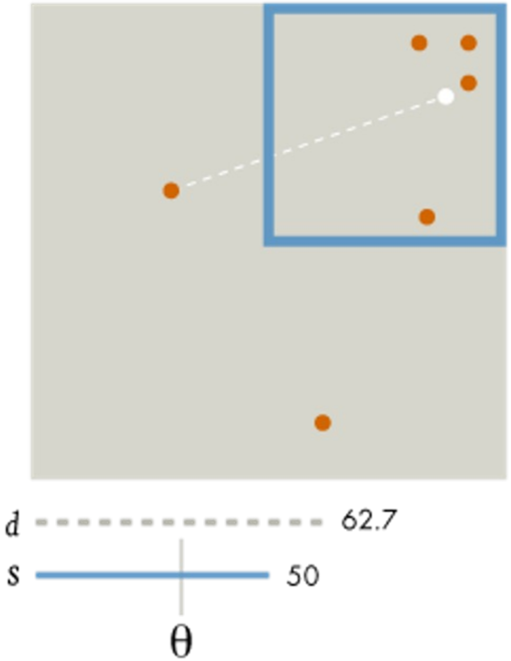


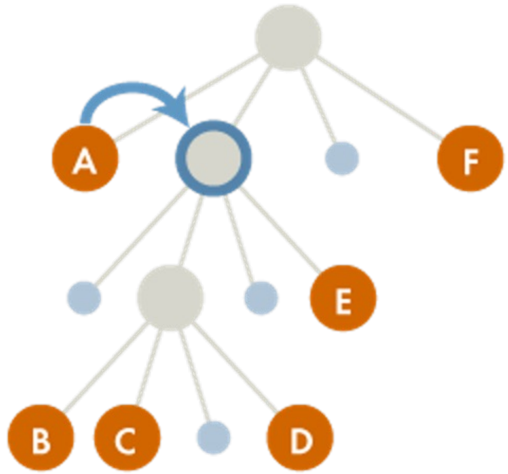


The Barnes-Hut Algorithm

-- Calculate Forces

```
def calculate_forces(pos, mass, G, softening):
    acc  = zeros_like(pos)
    root = build_octree(pos, mass)
    soft2 = softening**2

    def add_from_node(i, node):
        if node.mass == 0: return

        if node.is_leaf:
            for j in node.idx:
                if j == i: continue
                dr = pos[j] - pos[i]
                dist = norm(r)
                inv = 1 / (dist**2 + soft2)**1.5
                acc[i] += G * mass[j] * dr * inv
        else:

            dr = node.com - pos[i]
            dist = norm(r)

            if node.size()/dist < theta:
                inv = 1 / (dist**3 + soft2**1.5)
                acc[i] += G * node.mass * dr * inv
            else:
                for ch in node.children:
                    add_from_node(i, ch)

    for i in range(len(pos)):
        add_from_node(i, root)

    return acc
```

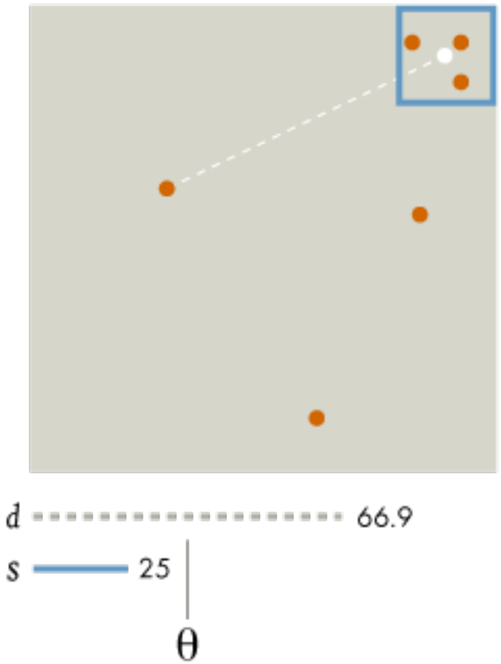


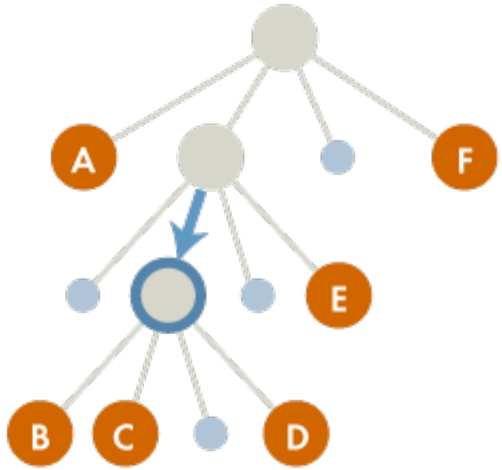


The Barnes-Hut Algorithm

-- Calculate Forces


```
def calculate_forces(pos, mass, G, softening):
    acc  = zeros_like(pos)
    root = build_octree(pos, mass)
    soft2 = softening**2

    def add_from_node(i, node):
        if node.mass == 0: return

        if node.is_leaf:
            for j in node.idx:
                if j == i: continue
                dr = pos[j] - pos[i]
                dist = norm(r)
                inv = 1 / (dist**2 + soft2)**1.5
                acc[i] += G * mass[j] * dr * inv
        else:

            dr = node.com - pos[i]
            dist = norm(r)

            if node.size()/dist < theta:
                inv = 1 / (dist**3 + soft2**1.5)
                acc[i] += G * node.mass * dr * inv
            else:
                for ch in node.children:
                    add_from_node(i, ch)

    for i in range(len(pos)):
        add_from_node(i, root)

    return acc
```

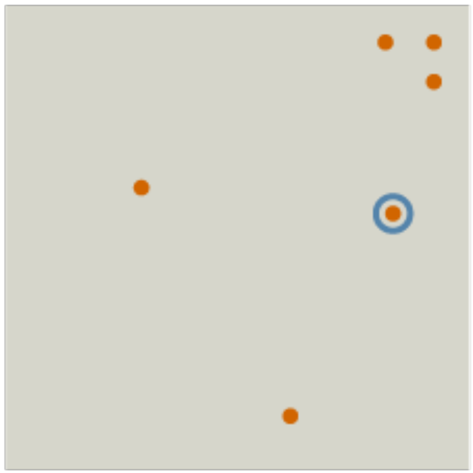


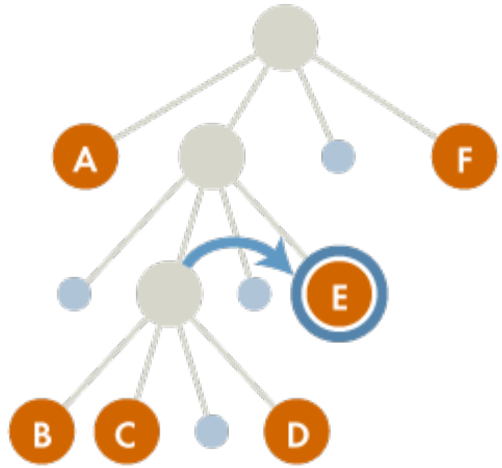


The Barnes-Hut Algorithm

-- Calculate Forces


```
def calculate_forces(pos, mass, G, softening):
    acc  = zeros_like(pos)
    root = build_octree(pos, mass)
    soft2 = softening**2

    def add_from_node(i, node):
        if node.mass == 0: return

        if node.is_leaf:
            for j in node.idx:
                if j == i: continue
                dr = pos[j] - pos[i]
                dist = norm(r)
                inv = 1 / (dist**2 + soft2)**1.5
                acc[i] += G * mass[j] * dr * inv
        else:

            dr = node.com - pos[i]
            dist = norm(r)

            if node.size()/dist < theta:
                inv = 1 / (dist**3 + soft2**1.5)
                acc[i] += G * node.mass * dr * inv
            else:
                for ch in node.children:
                    add_from_node(i, ch)

    for i in range(len(pos)):
        add_from_node(i, root)

    return acc
```

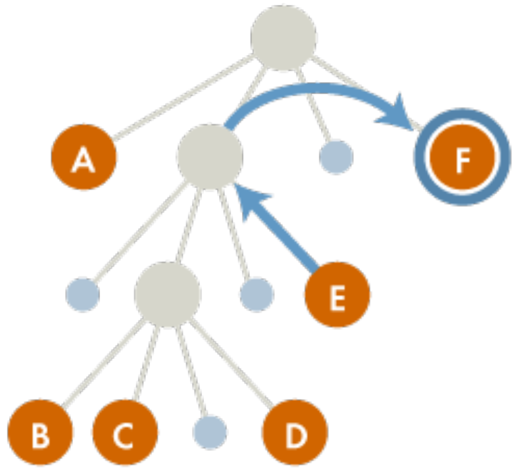




The Barnes-Hut Algorithm

-- Calculate Forces


```python
def calculate_forces(pos, mass, G, softening):
    acc  = zeros_like(pos)
    root = build_octree(pos, mass)
    soft2 = softening**2

    def add_from_node(i, node):
        if node.mass == 0: return

        if node.is_leaf:
            for j in node.idx:
                if j == i: continue
                dr = pos[j] - pos[i]
                dist = norm(r)
                inv = 1 / (dist**2 + soft2)**1.5
                acc[i] += G * mass[j] * dr * inv
        else:

            dr = node.com - pos[i]
            dist = norm(r)

            if node.size()/dist < theta:
                inv = 1 / (dist**3 + soft2**1.5)
                acc[i] += G * node.mass * dr * inv
            else:
                for ch in node.children:
                    add_from_node(i, ch)

    for i in range(len(pos)):
        add_from_node(i, root)

    return acc
```



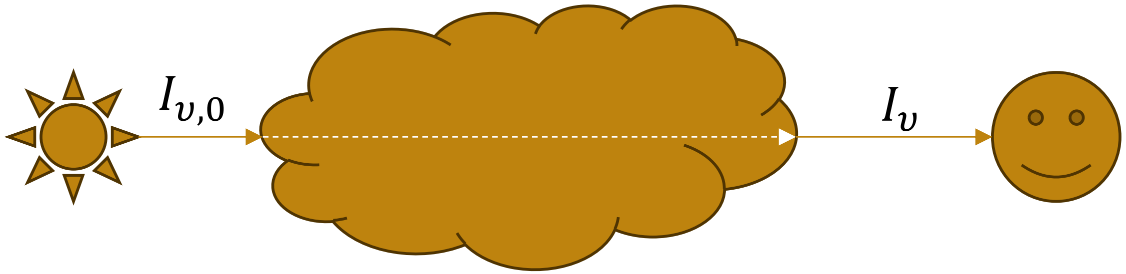
open simulations-part2.ipynb

Radiative Transfer

$$\frac{\mathrm{d}I_\upsilon}{\mathrm{d}s} = j_\upsilon - \alpha_\upsilon I_\upsilon$$

Optical depth: $\tau_\upsilon = \int_0^s \mathrm{d}s' \, \alpha_\upsilon$

$$I_\upsilon = I_{\upsilon,0} e^{-\tau_\upsilon} + \int_0^s \mathrm{d}s' j_\upsilon e^{-\tau_\upsilon(s')}$$



Radiative Transfer Pseudocode

$$\frac{\mathrm{d}I_v}{\mathrm{d}s} = j_v - \alpha_v I_v$$

Optical depth: $\tau_v = \int_0^s \mathrm{d}s' \, \alpha_v$

$$I_v = I_{v,0} e^{-\tau_v} + \int_0^s \mathrm{d}s' j_v e^{-\tau_v(s')}$$

```
grid = initialize_grid(GRID_SIZE)

# Main ray tracing loop
for i in range(NUM_RAYS):
    ray = initialize_ray()
    trace_ray(ray, grid)

# Apply recombination in the grid after processing all photons
for cell in grid:
    cell["ionization_fraction"] *= (1 - RECOMBINATION_RATE)
```

```
def initialize_ray():
    # Initialize ray properties (starting position, direction)
    ray = {
        "position": SOURCE_POSITION,
        "direction": random_direction(),
        "distance_traveled": 0.0,
        "absorbed": False
    }
    return ray

def random_direction():
    # Generate random spherical coordinates
    theta = np.random.uniform(0, 2 * np.pi)  # azimuthal angle
    phi = np.random.uniform(0, np.pi)        # polar angle

    # Convert spherical coordinates to Cartesian coordinates
    x = np.sin(phi) * np.cos(theta)
    y = np.sin(phi) * np.sin(theta)
    z = np.cos(phi)

    # Normalize the vector to ensure it has a length of 1
    direction = np.array([x, y, z])
    return direction
```

Radiative Transfer Pseudocode

```
def trace_ray(ray, grid):
    while ray.distance_traveled < MAX_DISTANCE:
        # Update ray position based on direction
        ray.position += ray.direction * TIME_STEP  # Move ray forward
        ray.distance_traveled += TIME_STEP

        # Check for grid boundaries
        cell = get_cell_at_position(grid, ray.position)
        if cell:
            # Check for absorption based on grid properties
            if check_for_absorption(ray, cell):
                ray.absorbed = True
                cell["ionization_fraction"] += (CELL_VOLUME * IONIZATION_EFFICIENCY) # update ionization fraction
                break  # Stop tracing this ray if absorbed

def get_cell_at_position(grid, position):
    for cell in grid:
        if (cell["x_min"] <= position[0] <= cell["x_max"] and
            cell["y_min"] <= position[1] <= cell["y_max"] and
            cell["z_min"] <= position[2] <= cell["z_max"]):
            return cell
    return None

def check_for_absorption(ray, cell):
    # Determine if the ray is absorbed based on the cell properties
    tau = cell["optical_depth"] * cell["cross_section"] * distance
    absorption_probability = 1 - np.exp(-tau)
    return random.uniform(0, 1) < absorption_probability(cell)
```

open simulations-part3.ipynb

prerequisite

We will start at 1:05pm. Before that, please

git clone/pull from https://github.com/qyx268/astr4004-8004-2025.git

Install ipympl