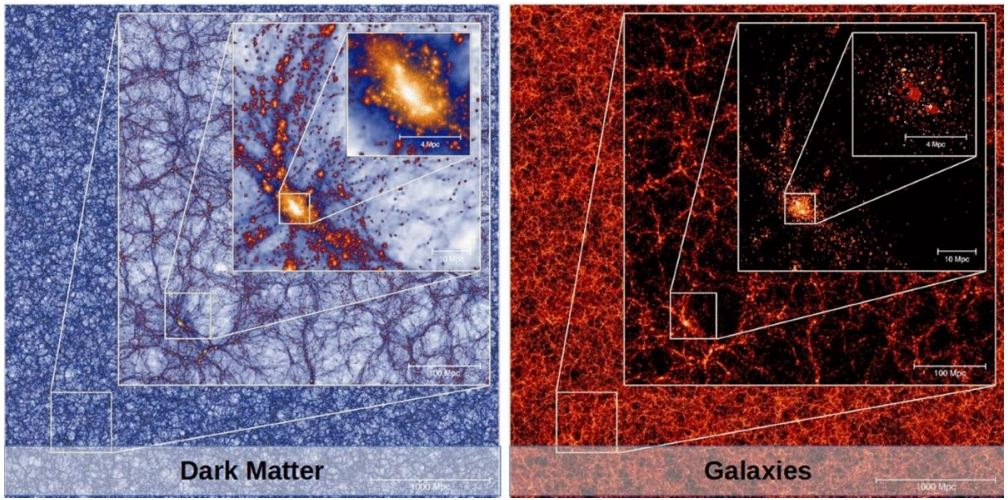
ASTR4004/8004 Computational Astronomy

ASTRONOMICAL SIMULATIONS

Yuxiang.Qin@anu.edu.au
Woolley Building W22



Australian National University

W1-6

Bash, shell, ssh

High-Performance Computing

Python

C

Parallel Computing

W7,8,9

Data Processing

Statistics

Plotting

Regression

Clustering

Model Selection

W10

Neural Network

Simulations

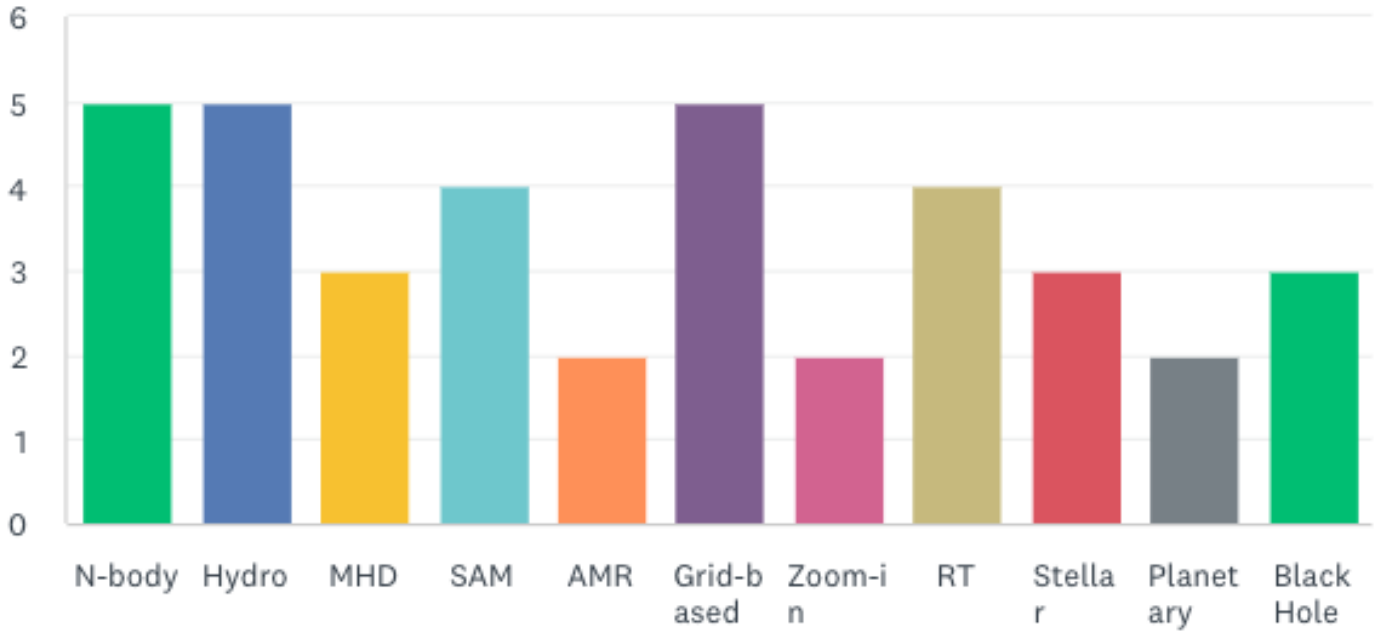
Inference

W11

W12

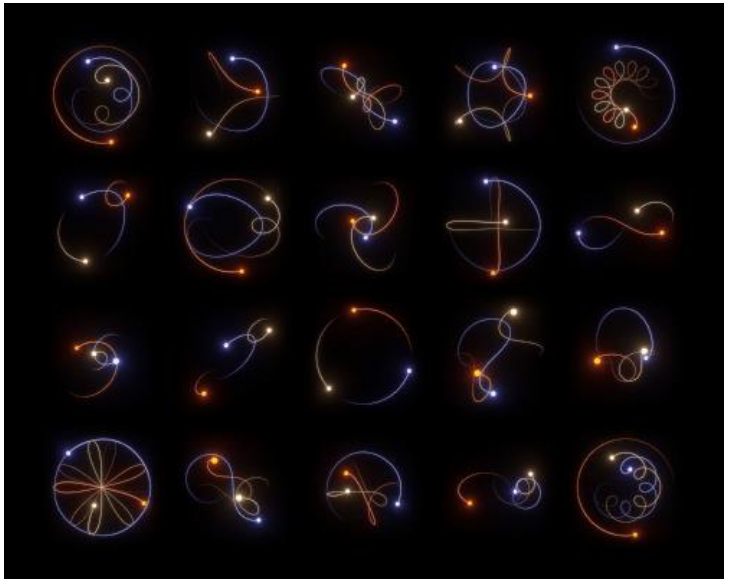
Tick all simulations you know

Answered: 8 Skipped: 5



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
```

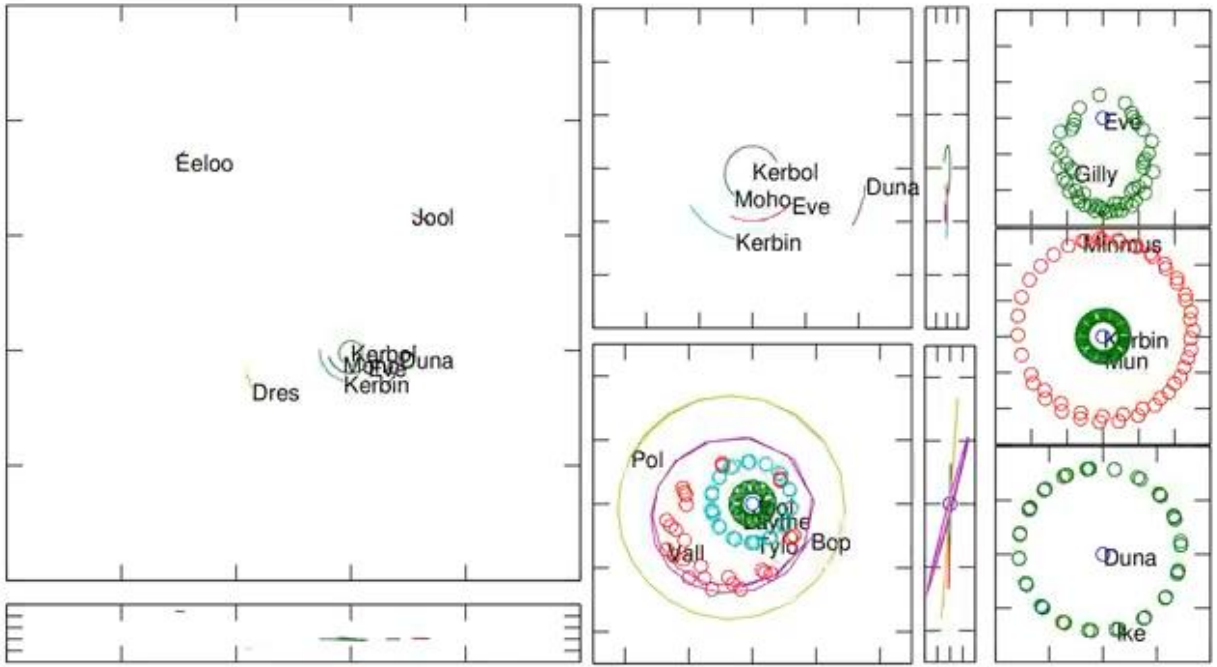
N-body simulation Pseudocode

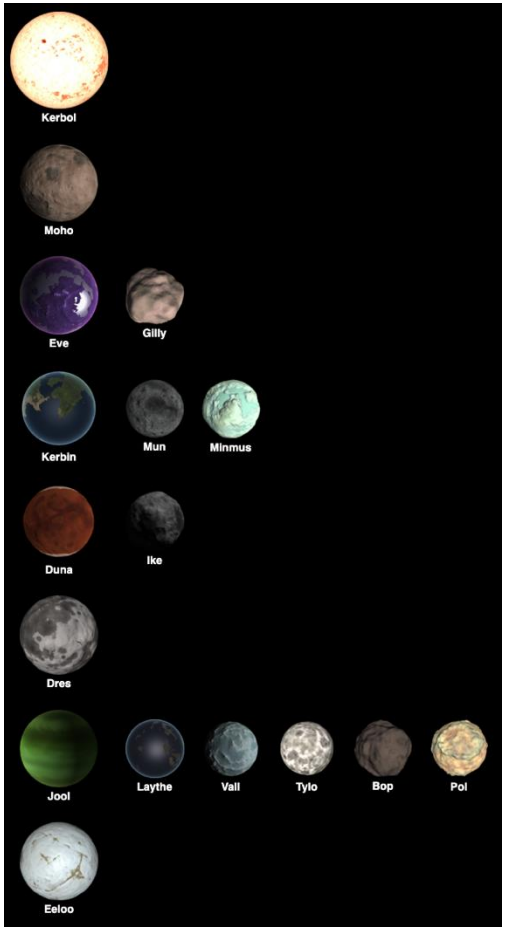
$$\ddot{\vec{r}}_j = -G \sum_{i \neq j}^{N} \frac{m_i}{\left|\vec{r}_i - \vec{r}_j\right|^3} (\vec{r}_i - \vec{r}_j)$$

```
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

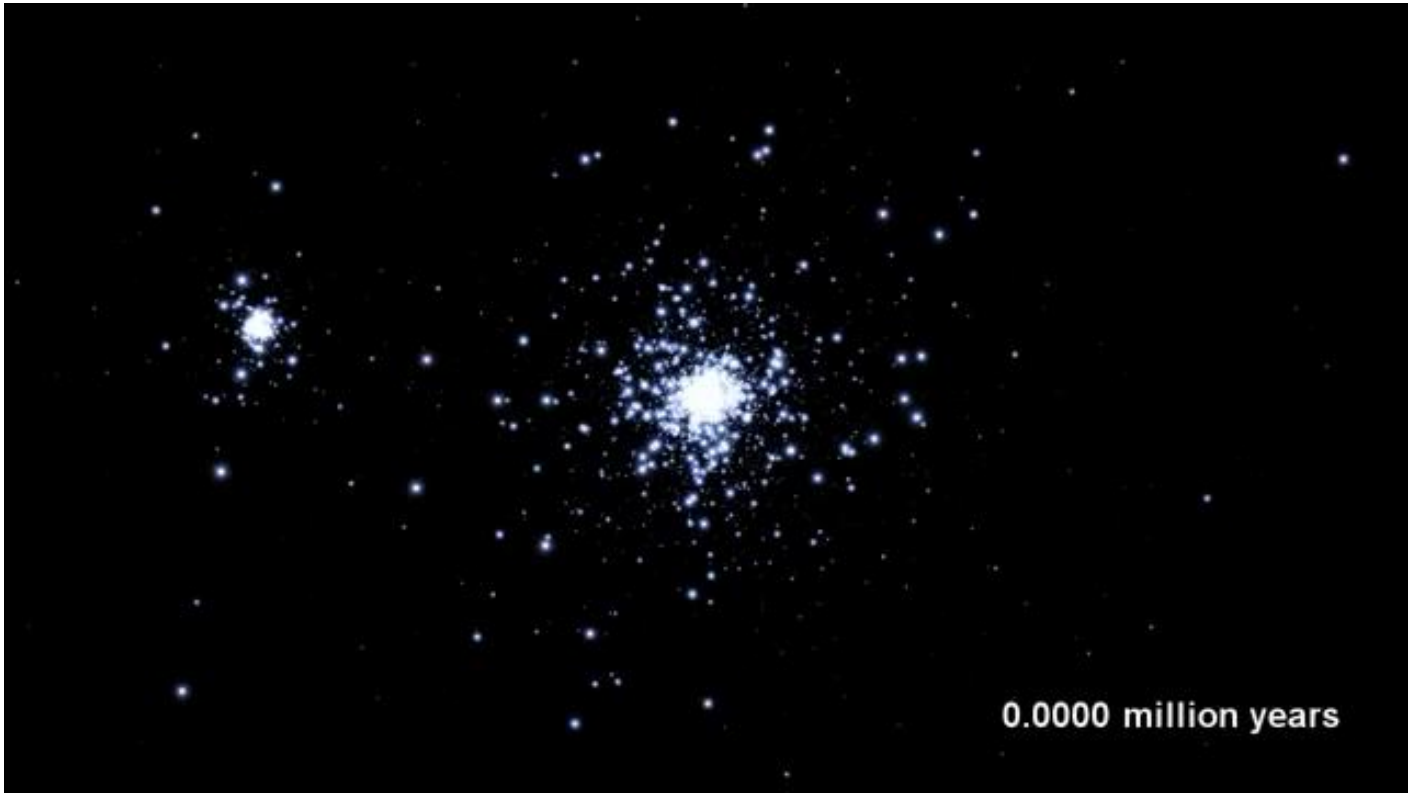
Kerbal Space Program





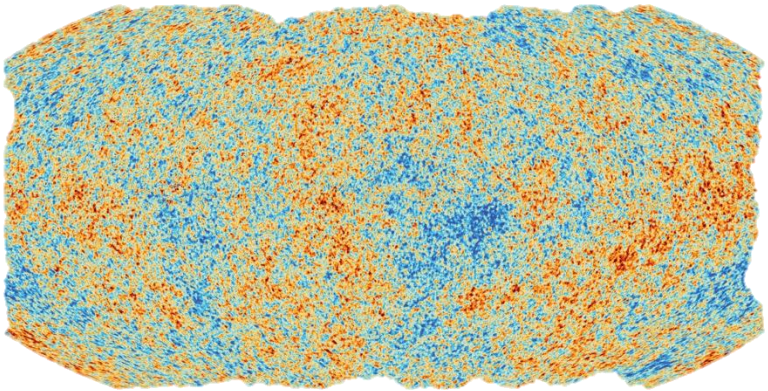
Star cluster

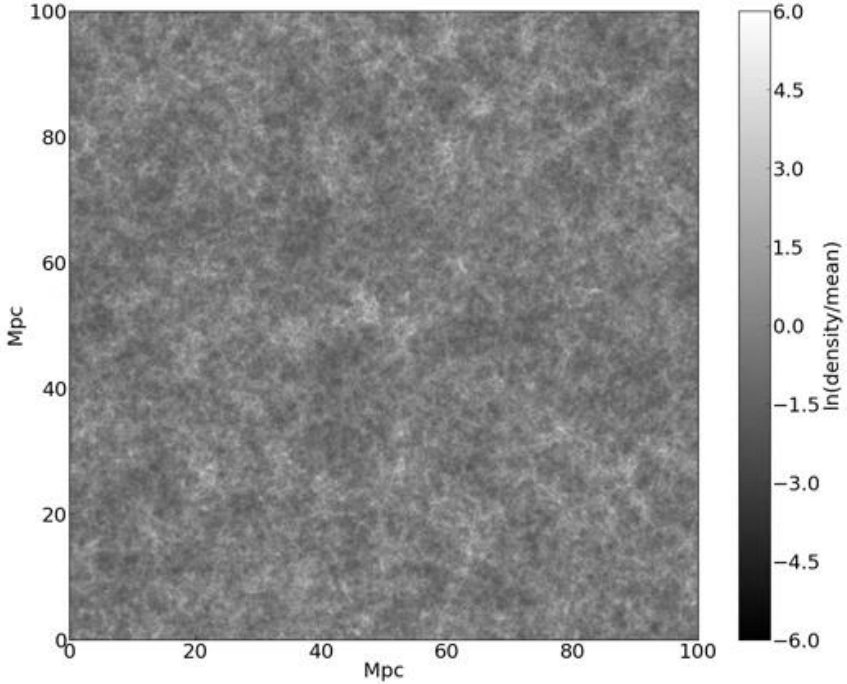
Stellar group R136 in the 30 Doradus nebula, in LMC



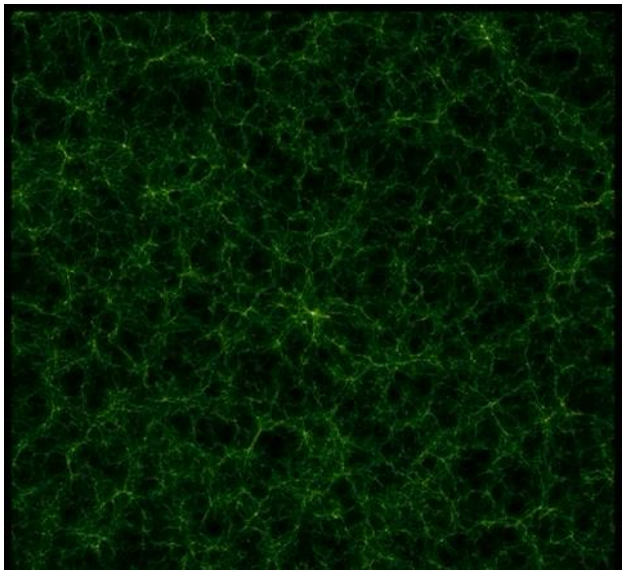
Andromeda colliding with the Milky Way

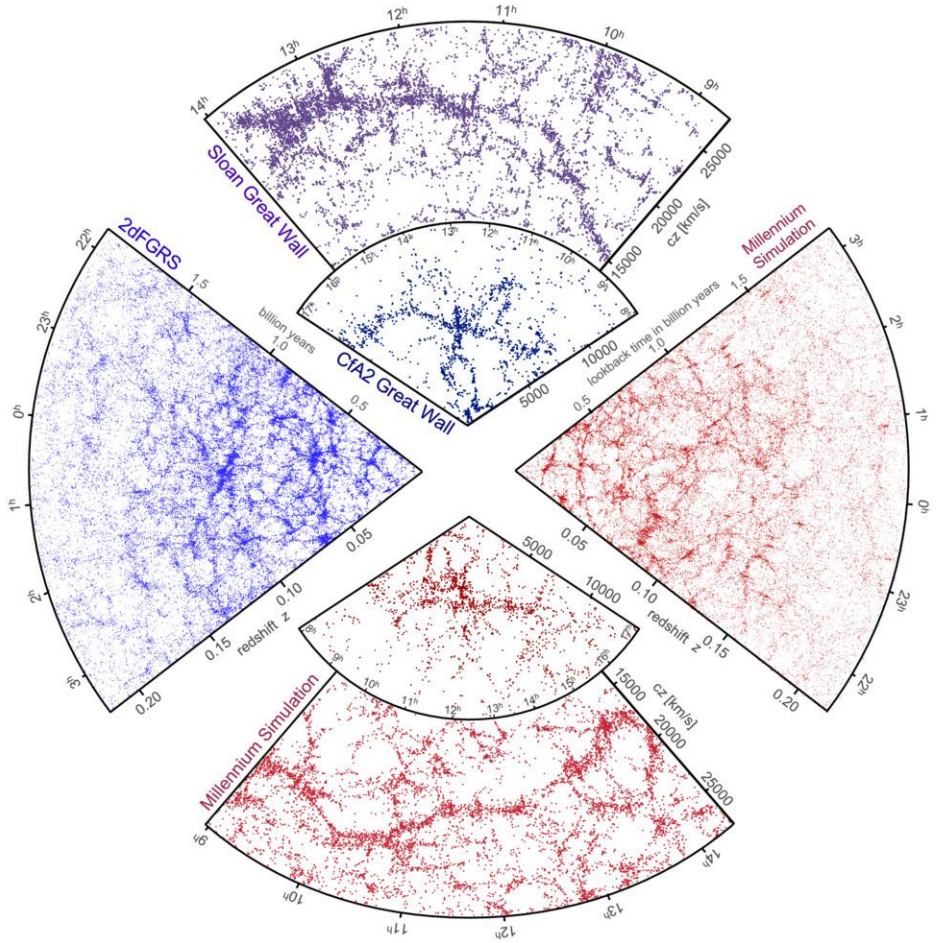
Cosmological N-body simulation





Cosmological N-body simulations





Hydrodynamic simulation Pseudocode

```
def n_body_hydrodynamics_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
    pressures = np.ones(N) * pressure_scale            # initialize pressure

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

        # 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

        # Update fluid properties
            pressures = update_pressures(positions, masses, N)

    return positions, velocities, pressures
```

Hydrodynamic simulation Pseudocode

```
def update_pressures(positions, masses, N):

    densities = np.zeros(N)  # Initialize density array
    for i in range(N):
       for j in range(N):
          if i != j:
             distance_vector = positions[j] - positions[i]
             distance = np.linalg.norm(distance_vector)
                   if distance < h:
               # Using a simple Gaussian kernel for density estimation
               kernel_value = np.exp(-distance**2 / (2 * soften **2)) / (soften * np.sqrt(2 * np.pi))
               densities[i] += masses[j] * kernel_value

    gamma = 5 / 3 # Adiabatic index for a monoatomic gas
    pressures = K * densities ** gamma

    return pressure
```

$$\rho_j = \sum_{i \neq j}^{N} m_i W(\vec{r}_i - \vec{r}_j, h)$$

$$P = K\,\rho^{\gamma}$$

Smoothed Particle Hydrodynamics (SPH)

Hydrodynamic simulation Pseudocode

$$\vec{F}_j = \sum_{i \neq j}^{N} \left(\frac{P_i - P_j}{\rho_j} \frac{\vec{r}_i - \vec{r}_j}{\left(\vec{r}_i - \vec{r}_j\right)^2} \right)$$

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

    for i in range(N):
        for j in range(i+1, N):
            # Calculate distance and direction
            distance_vector = positions[j] - positions[i]
            distance = np.linalg.norm(distance_vector)

            pressure_gradient = (pressures[j] - pressures[j]) / densities[i] # Averaged pressure
            pressure_force_magnitude = pressure_gradient / (distance**2 + soften **2)
            force_vector = pressure_force_magnitude * distance_vector
            pressure_forces[i] += force_vector # Force on particle i
            pressure_forces[j] -= force_vector # Equal and opposite force on particle j

    return pressure_ forces
```

Hydrodynamic simulation + star formation Pseudocode

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

    ...                  # initialize positions, velocities, masses, pressure
    star_particles = []       # List to store new star particles
    stellar_masses   = []     # List to store masses of new stars

    for step in range(int(total_time / dt)):  # loop through all steps
      ... # gravity,  hydrodynamics, update velocities and position, fluid properties

      for i in range(N):
        if density[i] > density_threshold:
          star_formation_rate = star_formation_efficiency * densities[i] # Modify as needed
          new_star_mass = star_formation_rate * dt
          if new_star_mass > random_number:       # Stochastically create a new star particle
            star_particles.append(positions[i])  # Store position of new star
            stellar_masses.append(new_star_mass) # Store mass of new star
            densities[i] -= new_star_mass       # Decrease gas density due to star formation

      supernova_feedback(star_particles, stellar_masses, positions, densities, dt, feedback_radius)

    return positions, velocities, pressures star_particles, stellar_masses
```

Hydrodynamic simulation + star formation Pseudocode

```
def supernova_feedback(star_particles, stellar_masses, positions, densities, dt, feedback_radius):
  for star_position, star_mass in zip(star_particles, stellar_masses):

        if stellar_mass > 0:
      feedback_energy = energy_per_supernova * fraction_supernova * star_mass / m_p

      for i in range(len(positions)):
        distance = np.linalg.norm(positions[i] - star_position)

        if distance < feedback_radius:
          # Calculate feedback impact based on distance
          influence = (1 - distance / feedback_radius)  # Influence decreases with distance
          # Update density and temperature of neighboring gas particles
          densities[i] += feedback_energy * influence / (dt * feedback_radius**3) # Energy to density conversion
          # Here, we might want to also consider temperature updates if needed:
          # temperature[i] += feedback_energy * influence / (densities[j] * specific_heat_capacity)

          # Optional: Implement a more complex model for feedback (e.g., modifying velocity)
          # velocities[i] += influence * velocity_feedback_factor  # Modify velocities based on feedback influence
```

MHD simulation Pseudocode

```
def MHD(positions, velocities, densities, pressures, magnetic_fields, dt, N):
    ... # initialize positions, velocities, masses, pressure, star particles and masses

    # Main simulation loop
    for step in range(int(total_time / dt)):  # loop through all steps
        ... # gravity,  hydrodynamics
        forces += calculate_magnetic_forces(positions, velocities, magnetic_fields, N)

        ... # update velocities and position, fluid properties
                ... # star formation and feedback

        # Update magnetic fields (could include diffusion, advection, etc.)
        magnetic_fields = update_magnetic_fields(magnetic_fields, positions, velocities, dt, N)

    return positions, velocities, star_particles, star_masses, magnetic_fields
```

MHD simulation Pseudocode

```
def calculate_magnetic_forces(positions, velocities, magnetic_fields, N):

    magnetic_forces = np.zeros((N, 3))  # Initialize magnetic force array


    for i in range(N):
        for j in range(N):
            if i != j:
                distance_vector = positions[j] - positions[i]
                distance = np.linalg.norm(distance_vector)
                if distance < h: # smoothing length for neighbor interactions
                    # Calculate magnetic field interaction
                    magnetic_force_magnitude = np.cross(velocities[i], magnetic_fields[i]) / (distance**2 + soften**2)
                    magnetic_forces[i] += magnetic_force_magnitude  # Force on particle i


    return magnetic_forces
```

$$\vec{F}_j = \sum_{i \neq j}^{N} \left(\frac{\vec{v}_j \times \vec{B}_i}{\left(\vec{r}_i - \vec{r}_j\right)^2} \right)$$

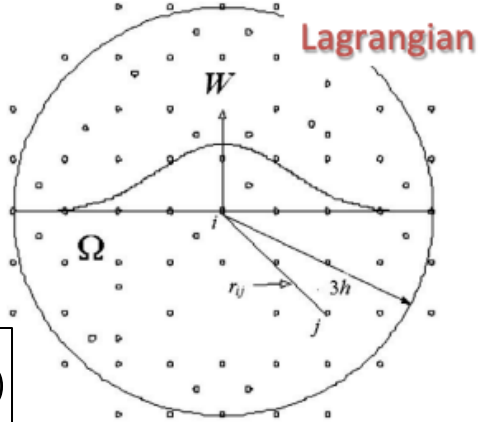
MHD simulation Pseudocode

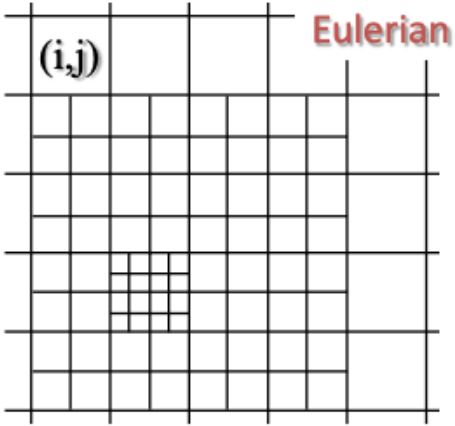
```
def update_magnetic_fields(magnetic_fields, velocities, positions, dt, N):
    updated_magnetic_fields = np.zeros_like(magnetic_fields)  # Initialize updated magnetic fields

    for i in range(N):
        # Initialize advection and diffusion components
        advection_term = np.zeros(3)
        diffusion_term = np.zeros(3)

        for j in range(N):
            if i != j:
                distance_vector = positions[j] - positions[i]
                distance = np.linalg.norm(distance_vector)

                if distance < h:
                    # Advection term (contribution from neighboring magnetic fields)
                    advection_term += velocities[i] * magnetic_fields[j] / (distance ** 2 + soften ** 2)

                    # Diffusion term (simplified Laplacian)
                    diffusion_term += (magnetic_fields[j] - magnetic_fields[i]) / (distance ** 2 + soften ** 2)

        # Update magnetic field based on advection and diffusion
        updated_magnetic_fields[i] = magnetic_fields[i] + dt * (advection_term - eta * diffusion_term)

    return updated_magnetic_fields
```

$$\frac{\partial \vec{B}}{\partial t} + \nabla \cdot (\vec{v}\vec{B}) = \nabla \cdot (\eta \nabla \vec{B})$$

Adaptive Mesh Refinement

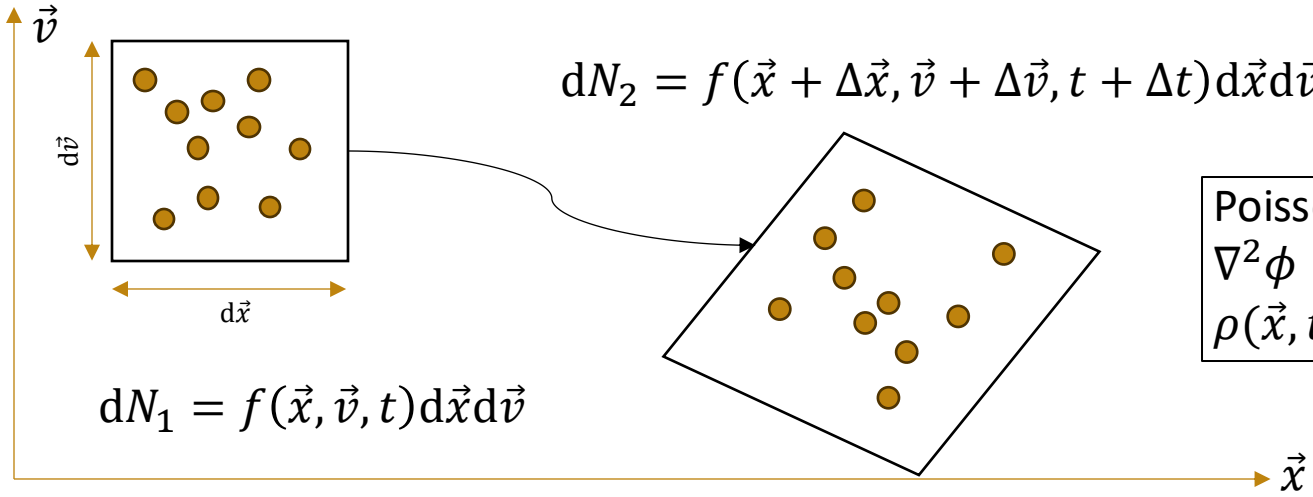
- Smoothed Particle Hydrodynamics (SPH) is particle-based, e.g., GADGET, PHANTOM.
- AMR is grid-based, with the simulation domain divided into a hierarchy of grids (with varying resolutions to dynamically refine simulation regions that need high accuracy), e.g., RAMSES, FLASH, Enzo.
  - Compared to SPH, AMR saves time by only increasing resolution in interested regions
  - deals better with sharp features like shocks (as SPH smooth out shock due to the kernel).
  - But its implementation is more complex and must be careful when dealing with boundaries.
- There are codes trying to combine both, e.g. AREPO, GIZMO.





$$\rho_j = \sum_{i \neq j}^{N} m_i W(\vec{r}_i - \vec{r}_j, h)$$

AMR vs SPH



$$\mathrm{d}N_2 = f(\vec{x} + \Delta\vec{x}, \vec{v} + \Delta\vec{v}, t + \Delta t)\mathrm{d}\vec{x}\mathrm{d}\vec{v}$$

$$\mathrm{d}N_1 = f(\vec{x}, \vec{v}, t)\mathrm{d}\vec{x}\mathrm{d}\vec{v}$$

Poisson equation:

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

$$\rho(\vec{x}, t) = \int \mathrm{d}\vec{v}\, f(\vec{x}, \vec{v}, t)$$

$$\mathrm{d}N_2 = \mathrm{d}\vec{x}\mathrm{d}\vec{v}\left[f(\vec{x}, \vec{v}, t) + \frac{\partial f}{\partial t} + \frac{\mathrm{d}\vec{x}}{\mathrm{d}t}\cdot\nabla_x f + \frac{\mathrm{d}\vec{v}}{\mathrm{d}t}\cdot\nabla_v f\right]$$

$$\mathrm{d}N_2 = \mathrm{d}N_1^* \rightarrow \text{Vlasov equation: } \frac{\partial f}{\partial t} + \vec{v}\cdot\nabla_x f + \frac{\vec{F}}{\mathrm{m}}\cdot\nabla_v f = 0$$

*assuming no particle leaving or added to the phase space (e.g., star formation or stellar death, accretion onto black holes)

AMR vs SPH

Solving Vlasov-Poisson equation:

$$\int \mathrm{d}\vec{v}\,\vec{U}_k\left[\frac{\partial f}{\partial t}+\vec{v}\cdot\nabla_x f+\frac{\vec{F}}{\mathrm{m}}\cdot\nabla_v f\right]=0 \qquad \rho(\vec{x},t)=\int \mathrm{d}\vec{v}\,f(\vec{x},\vec{v},t)$$

$\vec{U}_k = 1$ (continuity equation): $\int \mathrm{d}\vec{v}\,\frac{\partial f}{\partial t}+\int \mathrm{d}\vec{v}\,\vec{v}\cdot\nabla_x f+\int \mathrm{d}\vec{v}\,\frac{\vec{F}}{\mathrm{m}}\cdot\nabla_v f=\frac{\partial}{\partial t}\int \mathrm{d}\vec{v} f+\nabla_x\cdot\int \mathrm{d}\vec{v}\,\vec{v} f+0=$
$\frac{\partial \rho}{\partial t}+\nabla\cdot(\rho\vec{v})=0$

$\vec{U}_k = \vec{v}$(momentum equation): $\int \mathrm{d}\vec{v}\,\vec{v}\left[m\frac{\partial f}{\partial t}+m\vec{v}\cdot\nabla_x f+\vec{F}\cdot\nabla_v f\right]=\frac{\partial \int \mathrm{d}\vec{v} m\vec{v} f}{\partial t}+\nabla_x\cdot\int \mathrm{d}\vec{v}\; m\vec{v}\vec{v} f+$
$\int \mathrm{d}\vec{v}\,\vec{v}\vec{F}\cdot\nabla_v f=\frac{\partial \vec{P}}{\partial t}+\nabla_x\cdot T\;-\int \mathrm{d}\vec{v}\,\vec{F} f=\frac{\partial \vec{P}}{\partial t}+\;\nabla_x\cdot\boldsymbol{T}-\vec{F}=0$

$\vec{U}_k = v^2$ (energy equation): $\int \mathrm{d}\vec{v}\,v^2\left[0.5m\frac{\partial f}{\partial t}+0.5m\vec{v}\cdot\nabla_x f+0.5\vec{F}\cdot\nabla_v f\right]=\frac{\partial \int \mathrm{d}\vec{v} 0.5mv^2 f}{\partial t}+\nabla_x\cdot$
$\int \mathrm{d}\vec{v}\;0.5m\vec{v}v^2 f+\int \mathrm{d}\vec{v}\,0.5v^2\vec{F}\cdot\nabla_v f=\frac{\partial E}{\partial t}+\;\nabla_x\cdot\vec{Q}-\mathrm{W}=0$

*$\vec{P}$, ***T***, E, $\vec{Q}$, W are momentum and its flux tensor, energy and its flux vector and work done on the system.

AMR Pseudocode

```
class AMRSimulation:
  def __init__(self, domain_size, num_cells):
    self.domain_size = domain_size
    self.num_cells = num_cells
    self.grid = create_root_grid(self.domain_size, self. num_cells) # Create an initial coarse mesh grid
    self.initialize_variables(self.grid)

  def create_root_grid(domain_size, num_cells):
    x_min, x_max, y_min, y_max, z_min, z_max = domain_size
    grid = []

    # Loop through each spatial dimension and create cells
    for i in range(num_cells[0]):
      for j in range(num_cells[1]):
        for k in range(num_cells[2]):
          # Calculate the position of the cell in the grid
          cell_position = [x_min + i * (x_max - x_min) / num_cells[0],
                           y_min + j * (y_max - y_min) / num_cells[1],
                           z_min + k * (z_max - z_min) / num_cells[2]]
           # Define the size of each cell (constant for uniform root grid)
          cell_size = [(x_max - x_min) / num_cells[0],
                       (y_max - y_min) / num_cells[1],
                       (z_max - z_min) / num_cells[2]]
          # Initialize the cell with default physical properties (density, velocity, pressure, etc.)
          grid.append({"id": (i, j, k),  # Unique identifier for the cell
                       "position": cell_position,
                       "size": cell_size,
                       "refined": False,  # This cell is not refined yet
                       "children": []})    # Placeholder for children cells if this cell is refined
    return grid
```

AMR Pseudocode

```
class AMRSimulation:

  def initialize_variables(grid):
    for cell in grid:
      # Initialize density based on position, or set a constant value
      cell["density"] = initial_density(cell["position"])

      # Initialize momentum based on position, or set initial velocity to zero
      velocity = initial_velocity(cell["position"])  # velocity could be a vector
      cell["momentum"] = cell["density"] * velocity  # Momentum = density * velocity

      # Initialize internal energy, can depend on density or be a fixed value
      cell["energy"] = initial_energy(cell["density"], velocity)
    return grid

  def initial_density(position):
    # Example: Uniform density
    return 1.0

  def initial_velocity(position):
    # Example: Zero velocity
    return np.array([0.0, 0.0, 0.0])

  def initial_energy(density, velocity):
    # Example: Internal energy per unit mass (could be based on an equation of state)
    thermal_energy = 1.0e5  # Set an arbitrary thermal energy
    kinetic_energy = 0.5 * density * np.linalg.norm(velocity)**2  # K = 1/2 * rho * v^2
    return thermal_energy + kinetic_energy
```

AMR Pseudocode

```
class AMRSimulation:

  # Main time-stepping loop
  def run_simulation(grid, total_time, dt):
    time = 0
    while time < total_time:
      # Refine cells based on error indicators
      refine_grid(grid)

      # Compute fluxes for each cell
      fluxes = compute_fluxes(grid)

      # Update cell variables using fluxes
      update_cell_variables(grid, fluxes, dt)

      # Apply boundary conditions
      apply_boundary_conditions(grid)

      # Advance time
      time += dt

  # Adaptive mesh refinement based on error indicators
  def refine_grid(grid):
    for cell in grid:
      error_indicator = evaluate_error_indicator(cell)
      if needs_refinement(error_indicator):
        create_refined_cells(cell)  # subdivide the cell into finer cells

  # Evaluate error indicators to decide whether a cell needs refinement
  def evaluate_error_indicator(cell):
    density_gradient = np.gradient(cell["density"])
    error_indicator = np.linalg.norm(density_gradient)
    return error_indicator

  # Check if a cell needs to be refined
  def needs_refinement(error_indicator):
    if error_indicator > refinement_threshold:
      return True
    return False

  # Create finer cells by subdividing the parent cell
  def create_refined_cells(cell):
    # Subdivide cell into 2^3 = 8 smaller cells for 3D
    sub_cells = subdivide_cell(cell)
    cell.refined = True  # mark the parent cell as refined
    cell.children = sub_cells  # store refined cells as children
```

AMR Pseudocode

```
class AMRSimulation:

  # Main time-stepping loop
  def run_simulation(grid, total_time, dt):
    time = 0
    while time < total_time:
      # Refine cells based on error indicators
      refine_grid(grid)

      # Compute fluxes for each cell
      fluxes = compute_fluxes(grid)

      # Update cell variables using fluxes
      update_cell_variables(grid, fluxes, dt)

      # Apply boundary conditions
      apply_boundary_conditions(grid)

      # Advance time
      time += dt

  # Compute fluxes for each cell
  def compute_fluxes(grid):
    fluxes = {}
    for cell in grid:
      flux = calculate_flux(cell)
      fluxes[cell.id] = flux
    return fluxes
```

```
def calculate_flux(grid):
  fluxes = {
    "density_flux": np.zeros((len(grid), 3)),  # Assuming 3D grid
    "momentum_flux": np.zeros((len(grid), 3)),
    "energy_flux": np.zeros((len(grid), 3))
  }

  for cell in grid:
    # Get the cell properties
    den = cell["density"]
    mom = cell["momentum"]
    ene = cell["energy"]

    # Calculate velocity, potential energy and pressure
    vel, PE, P = calculate_gas_properties(den, mom, ene)

    # Example flux calculations (simplified, need flows from nearby cells)
    for dim in range(3):  # For each dimension (x, y, z)
      fluxes["density_flux"][cell["id"], dim] = den * vel[dim]
      fluxes["momentum_flux"][cell["id"], dim] = mom[dim] * vel[dim] + P
      fluxes["energy_flux"][cell["id"], dim] = (PE + 0.5 * den *\
                (np.linalg.norm(vel) ** 2) + P) * vel[dim]

  return fluxes
```

AMR Pseudocode

```
class AMRSimulation:

    # Main time-stepping loop
    def run_simulation(grid, total_time, dt):
        time = 0
        while time < total_time:
            # Refine cells based on error indicators
            refine_grid(grid)

            # Compute fluxes for each cell
            fluxes = compute_fluxes(grid)

            # Update cell variables using fluxes
            update_cell_variables(grid, fluxes, dt)

            # Apply boundary conditions
            apply_boundary_conditions(grid)

            # Advance time
            time += dt

    # Update cell variables based on fluxes
    def update_cell_variables(grid, fluxes, dt):
        for cell in grid:
            # Update cell quantities based on fluxes and the time step
            cell.density += fluxes[cell.id]["density_flux"] * dt
            cell.momentum += fluxes[cell.id]["momentum_flux"] * dt
            cell.energy += fluxes[cell.id]["energy_flux"] * dt
```

AMR Pseudocode

```
class AMRSimulation:

  # Main time-stepping loop
  def run_simulation(grid, total_time, dt):
    time = 0
    while time < total_time:
      # Refine cells based on error indicators
      refine_grid(grid)

      # Compute fluxes for each cell
      fluxes = compute_fluxes(grid)

      # Update cell variables using fluxes
      update_cell_variables(grid, fluxes, dt)

      # Apply boundary conditions
      apply_boundary_conditions(grid)

      # Advance time
      time += dt

  # Apply boundary conditions
  def apply_boundary_conditions(grid):
    for cell in grid.boundary_cells:
      enforce_boundary_condition(cell)
```

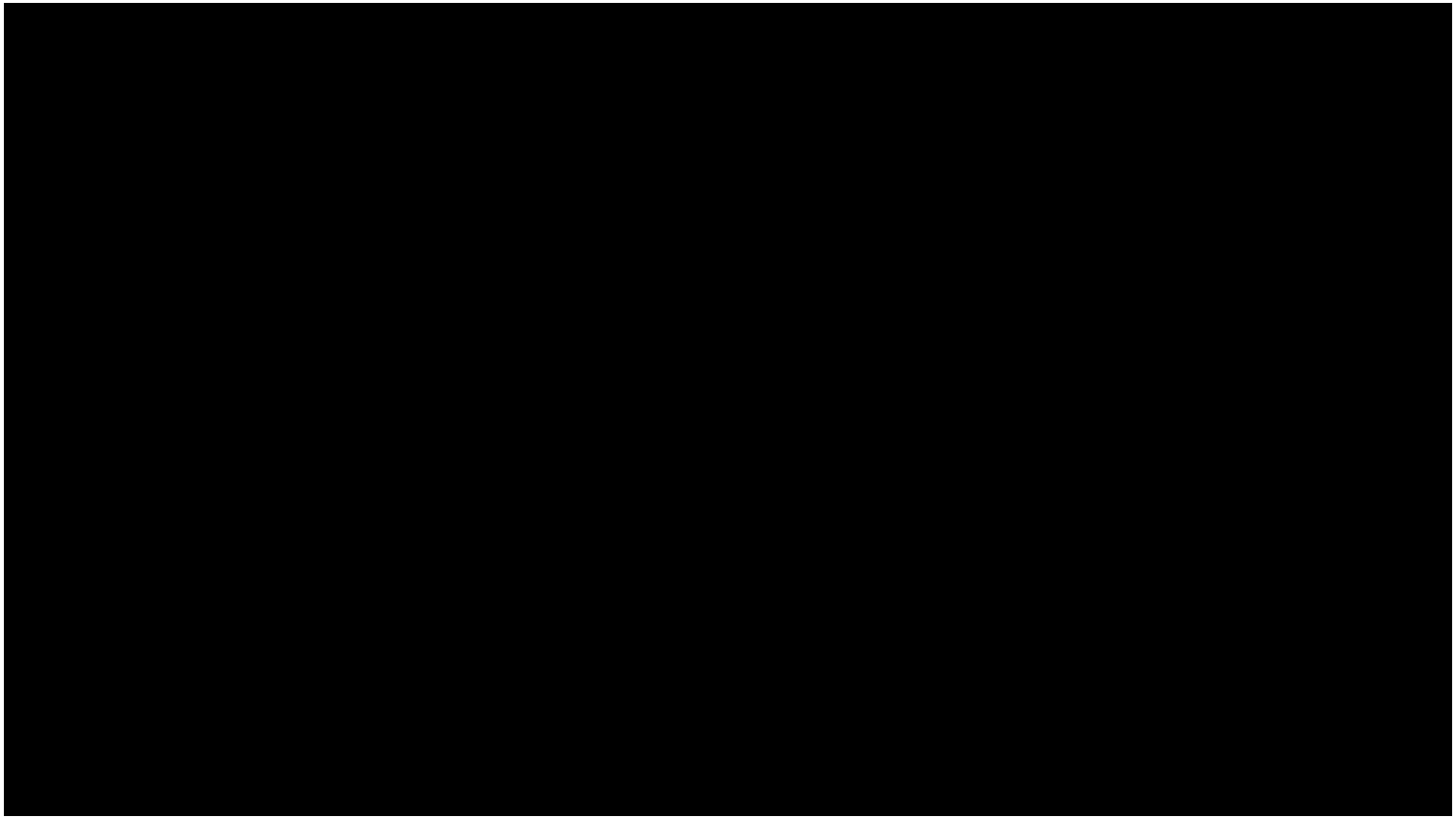
```
def enforce_boundary_condition(cell):

  # needs to first identify the boundaries, e.g. left, right, up...:
  if cell["is_boundary"]:

    # Example 1: Periodical boundary condition in cosmological v
    # find the opposite boundary, e.g., opposite(left) = right
    ghost_cell = opposite(cell)
    # pretend this ghost_cell is next to the boundary cell
    ghost_fluxes = compute_flux(ghost_cell)
    # add flux from the ghost cell to the boundary cell
    cell.density += ghost_fluxes [cell.id]["density_flux"] * dt
    cell.momentum += ghost_fluxes [cell.id]["momentum_flux"] * dt
    cell.energy += ghost_fluxes [cell.id]["energy_flux"] * dt

    # Example 2: Consider a star with a solid core surrounded
    # by a fluid outer layer. The inner solid core has a much
    # higher density and rigidity, in this case, we can set
    # velocity to zero for no movement
    if cell.distance_to_core() < self.core_radius:
      # Solid core: Dirichlet boundary condition
      cell.velocity = 0
      # update cell properties, e.g., pressure
      cell.pressure = self.calculate_pressure(cell)

  return updated_cell
```

AMR vs SPH

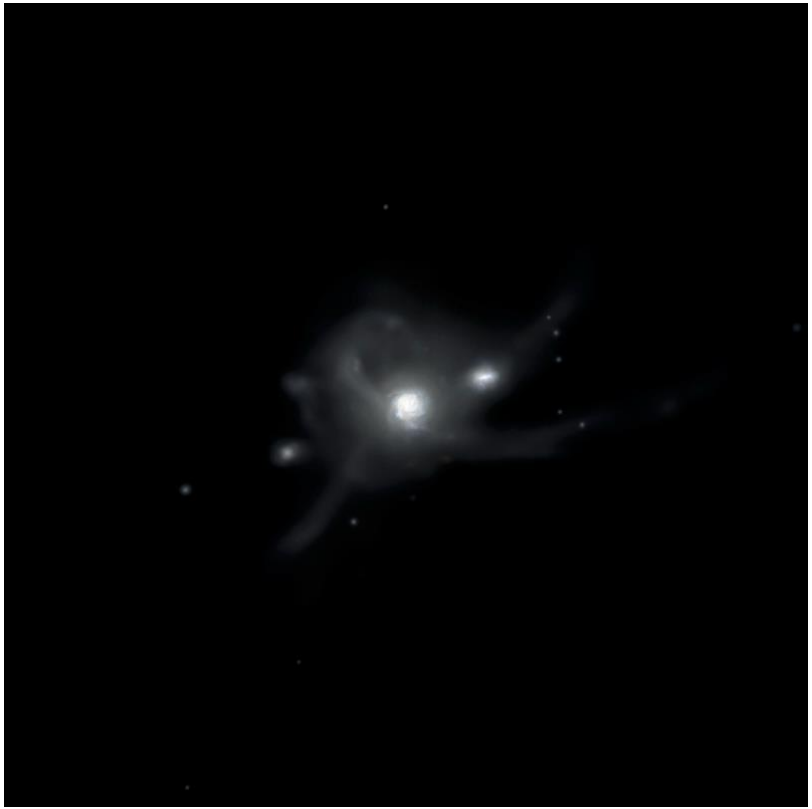
Well-known cosmological/zoom-in simulations

	SPH	AMR	Hybrid
Cosmological	EAGLE (1407.7040) Magneticum (1509.05134) FLAMINGO (2306.04024)	Horizon-AGN (1402.1165)	Illustris (1405.1418) IllustrisTNG (1707.03395)
Zoom-ins	FIRE (1608.04133)	Renaissance (1604.07842)	Illustris (1605.08205) Auriga (1601.01159)

Cosmological hydrodynamic simulations



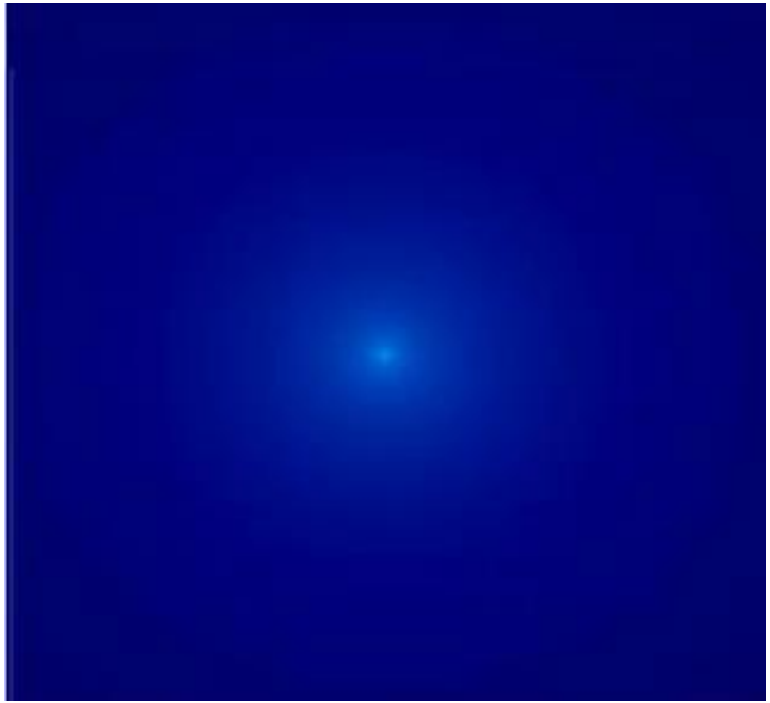
Zoom in simulations



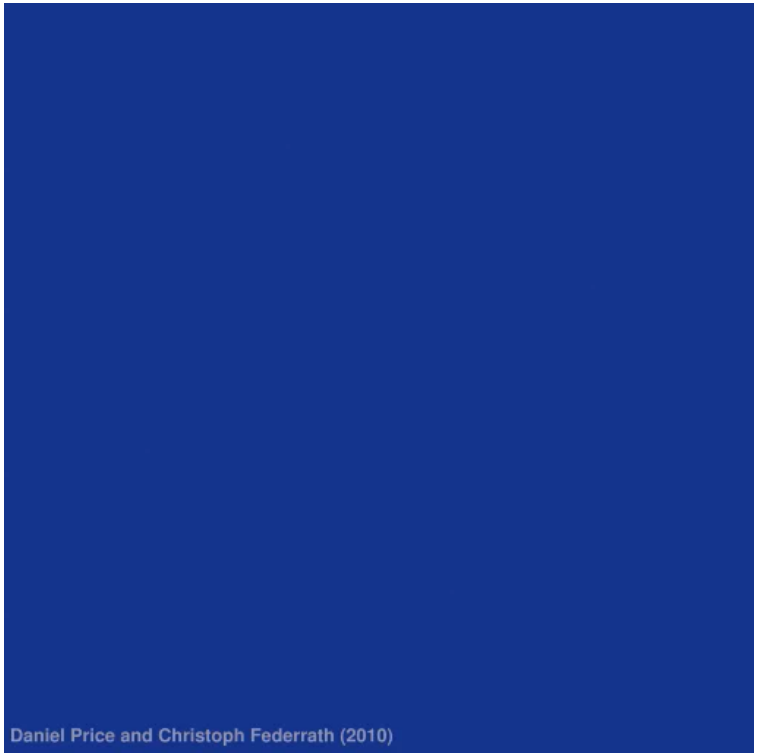
Galaxy formation

MACS1149-JD1

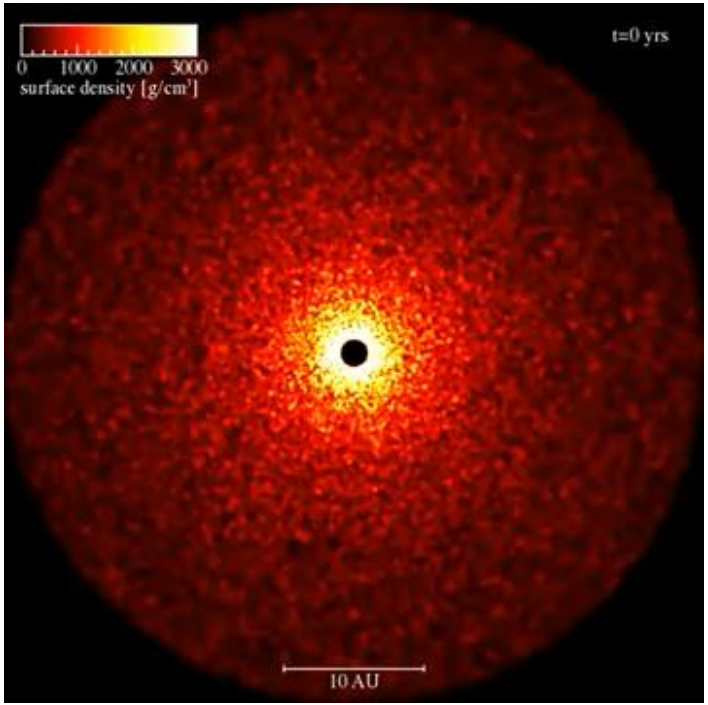
Star formation



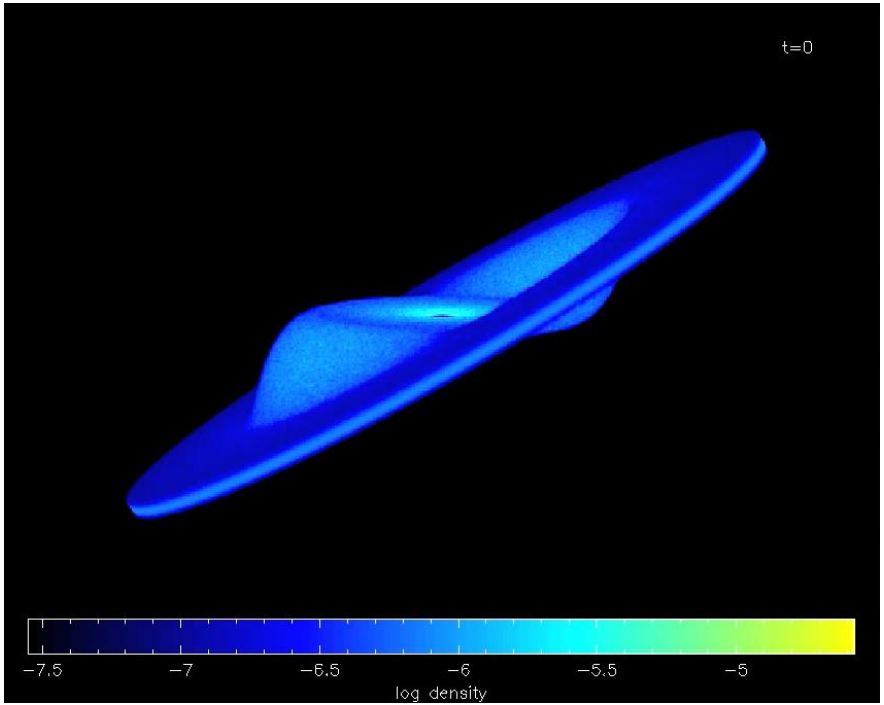
Molecular cloud



Planet formation



Blackhole

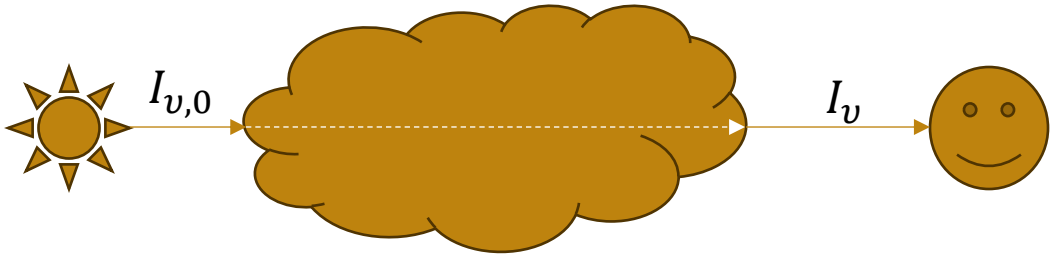


Radiative Transfer

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

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

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

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

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

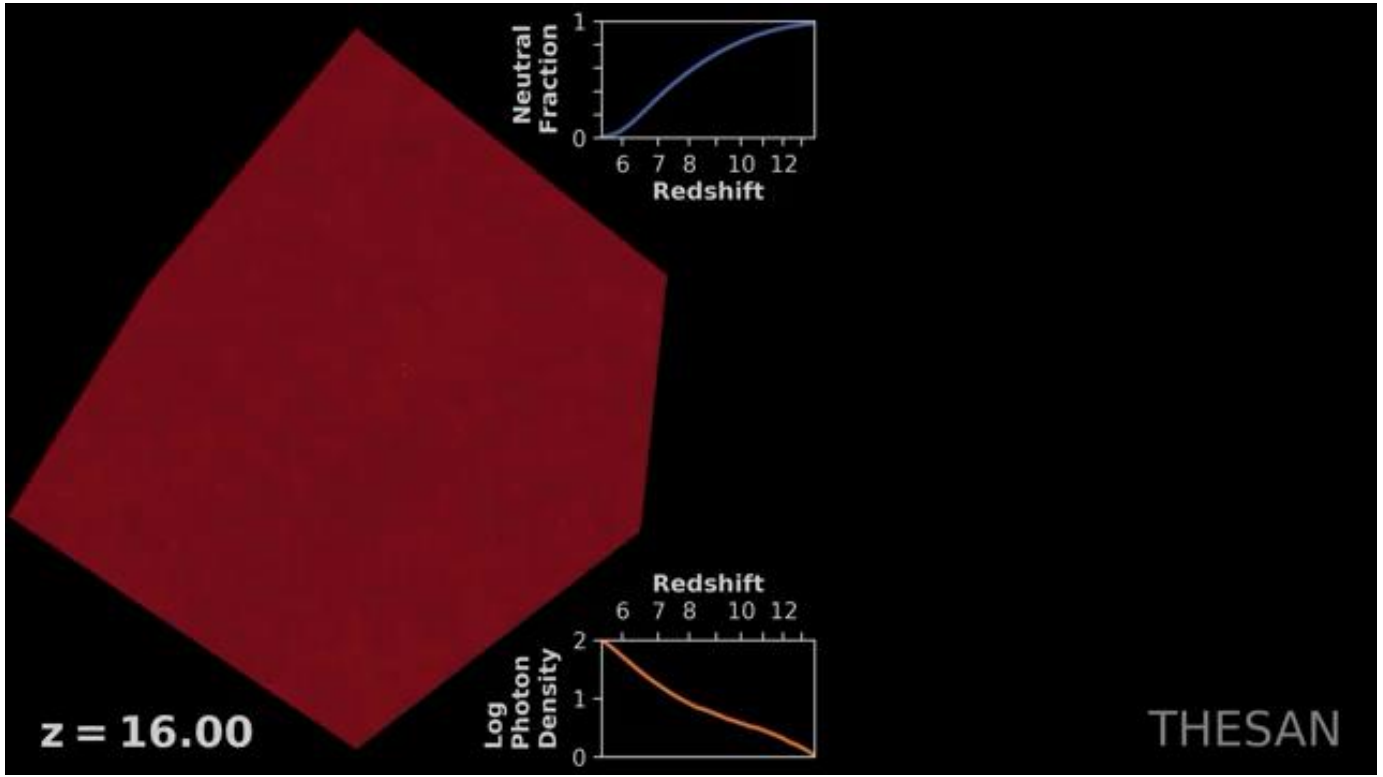
```
grid = initialize_grid(GRID_SIZE)

# 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

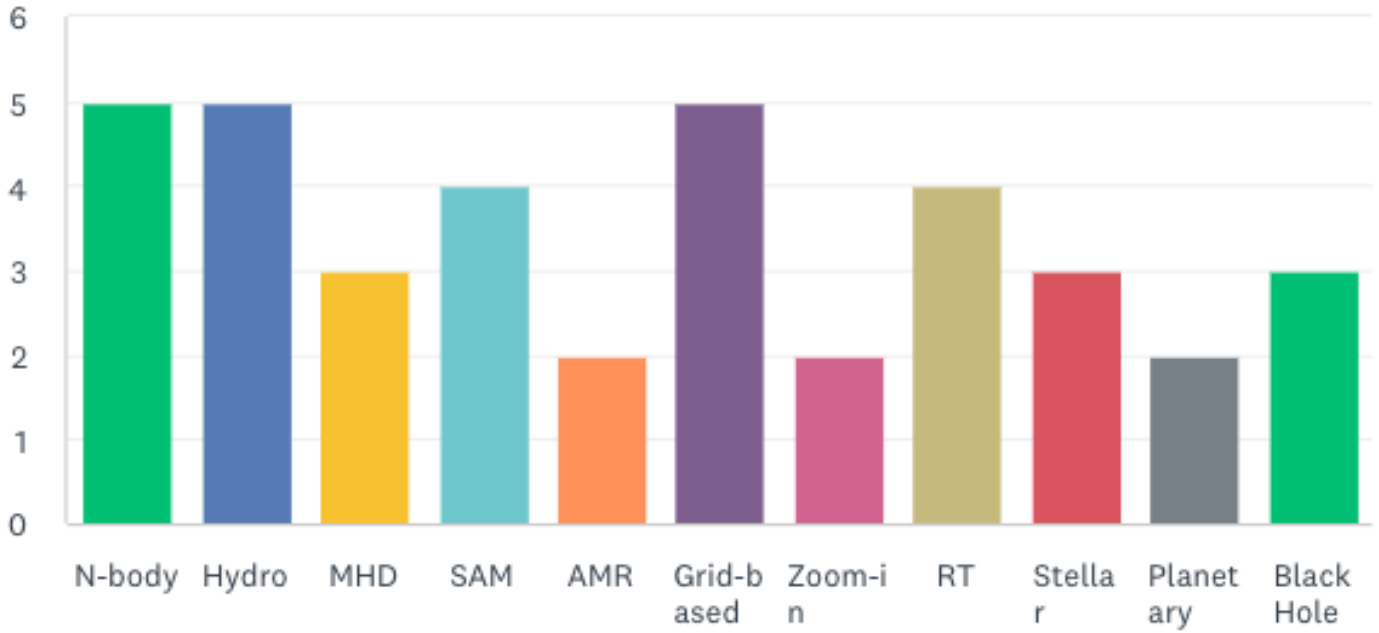
```
def trace_ray(ray, grid):
    while ray.distance_traveled < MAX_DISTANCE:
        # Update ray position based on direction
        ray.position += ray.direction * TIME_STEP  # Move ray forward
        ray.distance_traveled += TIME_STEP

        # 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)
```

IGM Reionization Simulation



Tick all simulations you know

Answered: 8 Skipped: 5



Semi-analytic model

Semi-analytic models balance between using theoretical models and numerical simulations, widely used to study galaxy formation and evolution

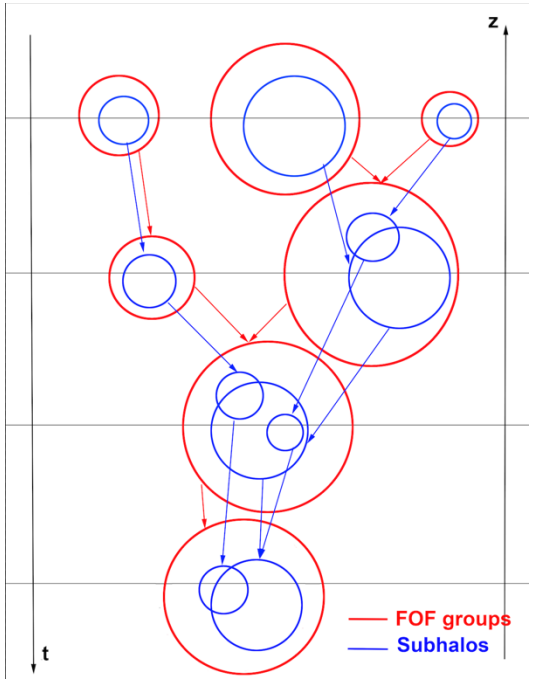
Features	Semi-Analytic Models (SAM)	Hydrodynamic Simulations
Complexity	Simplified physical models with parametrized equations	Full treatment of physical processes
Computational Cost	Low; O(#halos)	High; O(#particles)
Input	Halo merger trees from N-body simulations	N-body + hydrodynamics
Spatial resolution	Coarse, not resolving halos	High, resolving galaxy substructure
Mass resolution	High	Low

Semi-analytic model

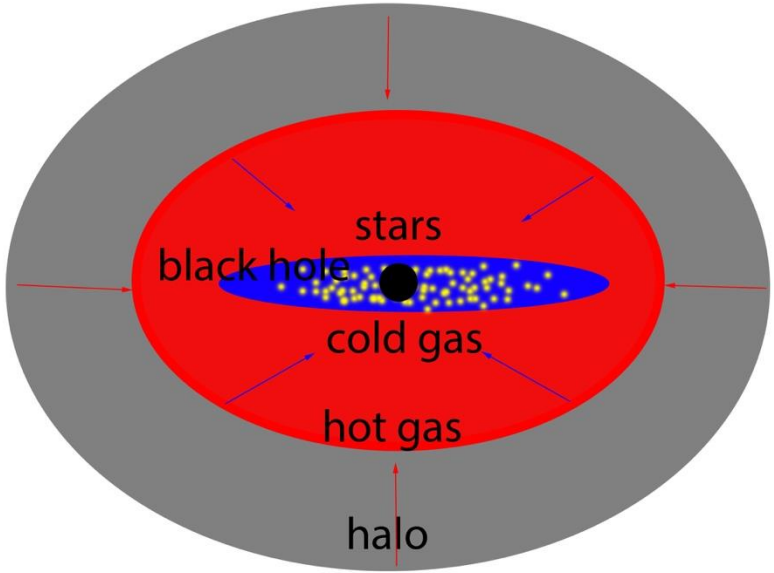
Features	Semi-Analytic Models (SAM)	Hydrodynamic Simulations
Flexibility	Highly flexible due to the simplicity of models, allowing exploration of a wide range of parameters.	Less flexible, constrained by physical laws and computational limitations.
Precision	Less precise, as it relies on parametrizations and approximations for complex processes.	High precision due to detailed physical modeling, including non-linear effects.
Scalability	Easily scalable, allowing for simulations of large cosmological volumes	Less scalable, as computational costs increase steeply with volume and resolution.
Applications	Ideal for statistical studies of galaxy populations, galaxy mergers, reionization.	Ideal for studying individual galaxies, gas accretion, star formation, and feedback in detail.

Semi-analytic model

Halo Merger Trees



Semi-Analytic Model



Baryonic Physics

Gas infall
Gas cooling
Star Formation
Supernovae
Metal Enrichment
Black Hole Growth
Reionization Heating
AGN Feedback
Star Burst
…

Semi-analytic model

$\Omega_{\rm m}, \Omega_{\rm b} = 0.308, 0.0484$ (Planck15)

$f_{\rm b} = \Omega_{\rm b}/\Omega_{\rm m} = 0.157$

$M_{\rm vir} = 10^{15}{\rm M}_\odot$

$M_{\rm hot} = 10^{14}{\rm M}_\odot$

$M_{\rm cold} = 10^{13}{\rm M}_\odot$

$M_* = 10^{12}{\rm M}_\odot$

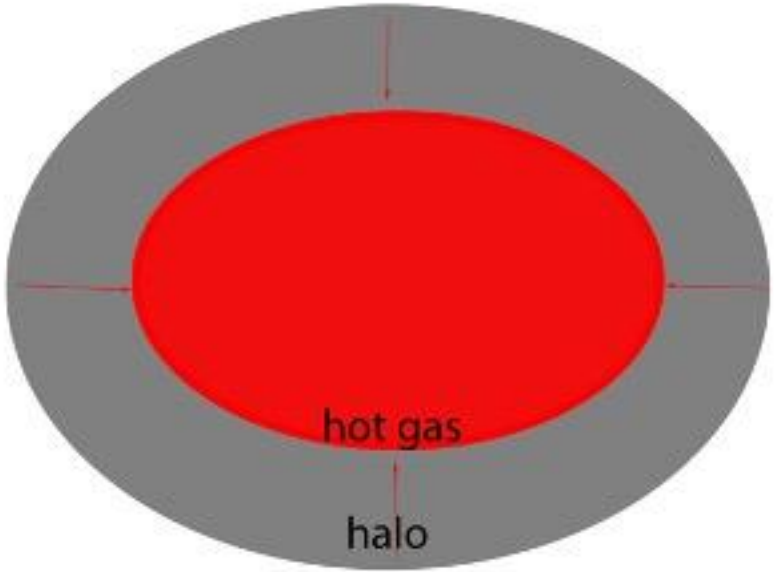
$M_{\rm eject} = 10^{11}{\rm M}_\odot$

$M_{\rm bh} = 10^{9}{\rm M}_\odot$

$M_{\rm bh,acc} = 10^{9}{\rm M}_\odot$

$f_{\rm mod} = 1$

Semi-Analytic Model



Baryonic Physics

Gas infall
Gas cooling
Star Formation
Supernovae
Metal Enrichment
Black Hole Growth
Reionization Heating
AGN Feedback
Star Burst
...

$$\Delta M_{\rm hot} = f_{\rm mod} f_{\rm b} M_{\rm vir} - \left(M_{\rm hot} + M_{\rm cold} + M_* + M_{\rm eject} + M_{\rm bh} + M_{\rm bh,acc}\right) = 10^{13.66}{\rm M}_\odot$$

Semi-analytic model

$\Omega_{\rm m}, \Omega_{\rm b} = 0.308, 0.0484$ (Planck15)

$f_{\rm b} = \Omega_{\rm b}/\Omega_{\rm m} = 0.157$

$M_{\rm vir} = 10^{15}{\rm M}_\odot$

$M_{\rm hot} = 10^{14.16}{\rm M}_\odot$

$M_{\rm cold} = 10^{13}{\rm M}_\odot$

$M_* = 10^{12}{\rm M}_\odot$

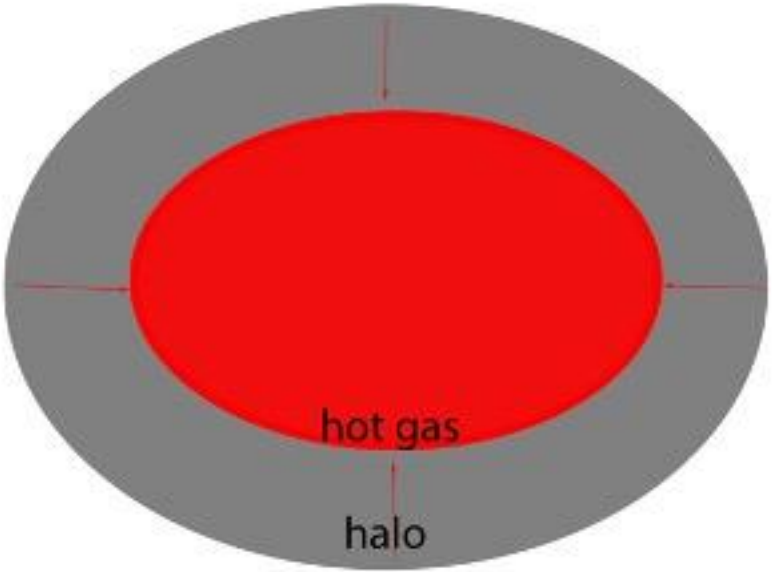
$M_{\rm eject} = 10^{11}{\rm M}_\odot$

$M_{\rm bh} = 10^{9}{\rm M}_\odot$

$M_{\rm bh,acc} = 10^{9}{\rm M}_\odot$

$f_{\rm mod} = 1$

Semi-Analytic Model

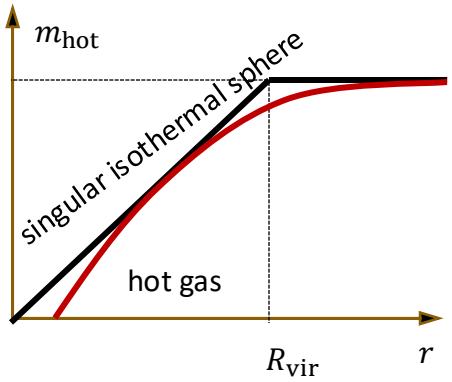


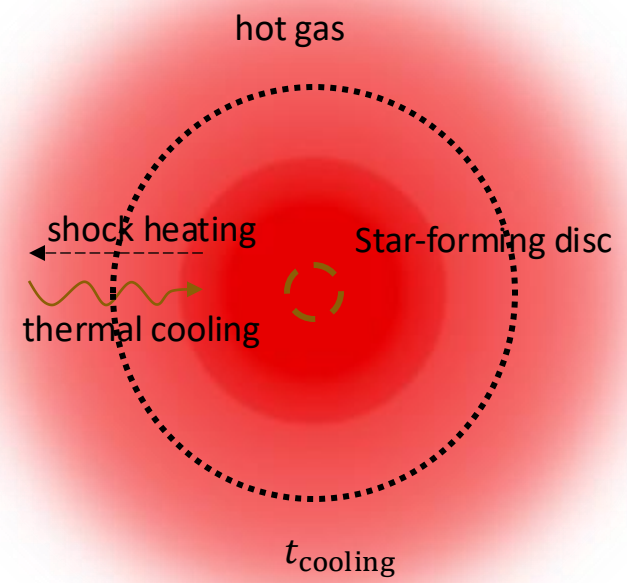
Baryonic Physics

Gas infall
Gas cooling
Star Formation
Supernovae
Metal Enrichment
Black Hole Growth
Reionization Heating
AGN Feedback
Star Burst
...

$$\Delta M_{\rm hot} = f_{\rm mod} f_{\rm b} M_{\rm vir} - \left(M_{\rm hot} + M_{\rm cold} + M_* + M_{\rm eject} + M_{\rm bh} + M_{\rm bh,acc}\right) = 10^{13.66}{\rm M}_\odot$$

Semi-analytic model





$$\dot{M}_{\rm cold} = \rho_{\rm hot}(r_{\rm cool}) 4\pi r_{\rm cool}^2 \dot{r}_{\rm cool}$$

Baryonic Physics

Gas infall
Gas cooling
Star Formation
Supernovae
Metal Enrichment
Black Hole Growth
Reionization Heating
AGN Feedback
Star Burst
...

Semi-analytic model

$\Omega_{\rm m}, \Omega_{\rm b} = 0.308, 0.0484$ (Planck15)

$f_{\rm b} = \Omega_{\rm b}/\Omega_{\rm m} = 0.157$

$M_{\rm vir} = 10^{15}{\rm M}_\odot$

$M_{\rm hot} = 10^{14.16}{\rm M}_\odot$

$M_{\rm cold} = 10^{13}{\rm M}_\odot$

$M_* = 10^{12}{\rm M}_\odot$

$M_{\rm eject} = 10^{11}{\rm M}_\odot$

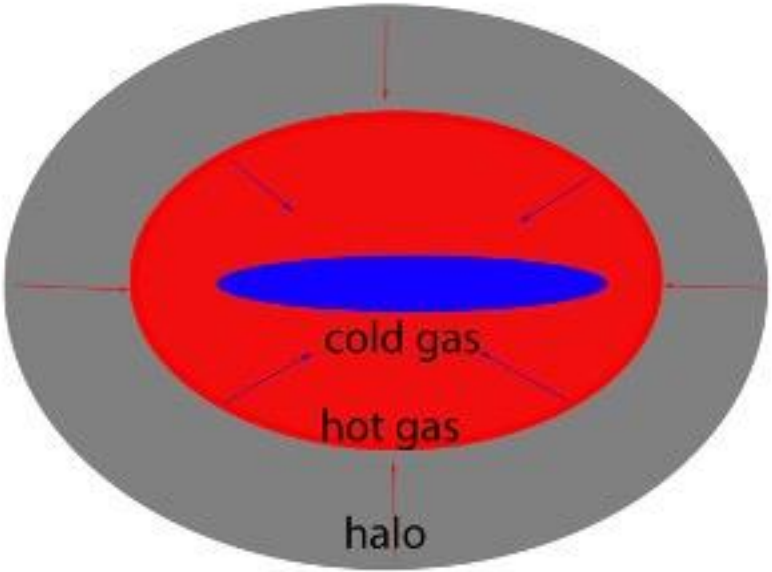
$M_{\rm bh} = 10^{9}{\rm M}_\odot$

$M_{\rm bh,acc} = 10^{9}{\rm M}_\odot$

$f_{\rm mod} = 1$

$\Delta M_{\rm cold} = \dot{M}_{\rm cold}\Delta t = 10^{6}{\rm M}_\odot{\rm yr}^{-1} \times 11{\rm Myr} = 1.1 \times 10^{13}\ {\rm M}_\odot$

Semi-Analytic Model



Baryonic Physics

Gas infall
Gas cooling
Star Formation
Supernovae
Metal Enrichment
Black Hole Growth
Reionization Heating
AGN Feedback
Star Burst
...

Semi-analytic model

$\Omega_{\rm m}, \Omega_{\rm b} = 0.308, 0.0484$ (Planck15)

$f_{\rm b} = \Omega_{\rm b}/\Omega_{\rm m} = 0.157$

$M_{\rm vir} = 10^{15}{\rm M}_\odot$

$M_{\rm hot} = 10^{14.13}{\rm M}_\odot$

$M_{\rm cold} = 10^{13.3222}{\rm M}_\odot$

$M_* = 10^{12}{\rm M}_\odot$

$M_{\rm eject} = 10^{11}{\rm M}_\odot$

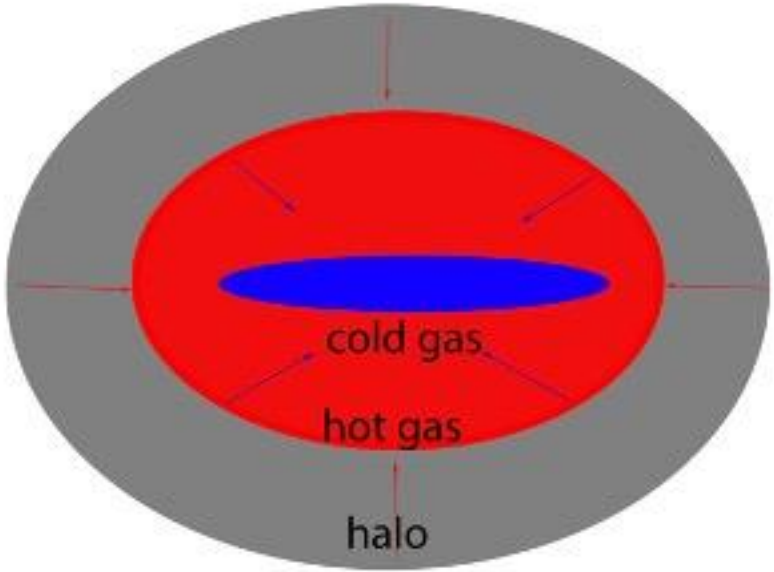
$M_{\rm bh} = 10^{9}{\rm M}_\odot$

$M_{\rm bh,acc} = 10^{9}{\rm M}_\odot$

$f_{\rm mod} = 1$

$\Delta M_{\rm cold} = \dot{M}_{\rm cold}\Delta t = 10^{6}{\rm M}_\odot{\rm yr}^{-1} \times 11{\rm Myr} = 1.1 \times 10^{13}\ {\rm M}_\odot$

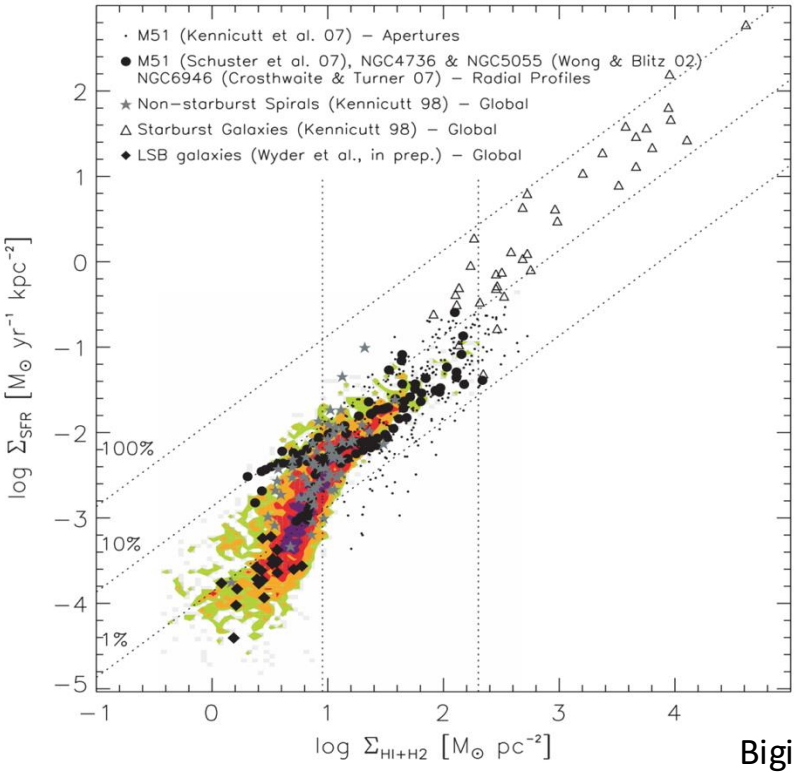
Semi-Analytic Model



Baryonic Physics

Gas infall
Gas cooling
Star Formation
Supernovae
Metal Enrichment
Black Hole Growth
Reionization Heating
AGN Feedback
Star Burst
...

Semi-analytic model



Bigiel+08

Convert surface density to mass assuming exponential disc profile.

$$\dot{M}_* = \frac{M_{\rm cold} - M_{\rm crit}}{t_{\rm H}}$$
$$= \alpha_{\rm sf} \frac{M_{\rm cold} - M_{\rm crit}}{t_{\rm dyn,dics}}$$

Semi-analytic model

$\Omega_{\rm m}, \Omega_{\rm b} = 0.308, 0.0484$ (Planck15)

$f_{\rm b} = \Omega_{\rm b}/\Omega_{\rm m} = 0.157$

$M_{\rm vir} = 10^{15}{\rm M}_\odot$

$M_{\rm hot} = 10^{14.13}{\rm M}_\odot$

$M_{\rm cold} = 10^{13.3222}{\rm M}_\odot$

$M_* = 10^{12}{\rm M}_\odot$

$M_{\rm eject} = 10^{11}{\rm M}_\odot$

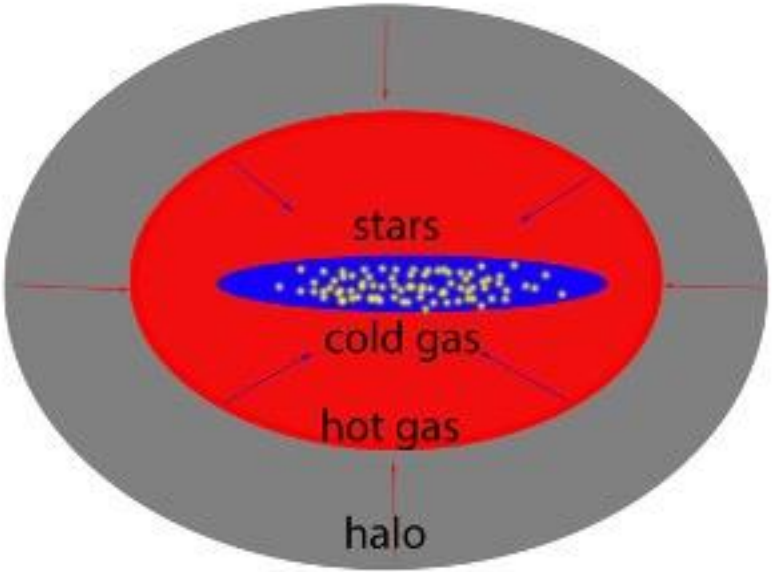
$M_{\rm bh} = 10^{9}{\rm M}_\odot$

$M_{\rm bh,acc} = 10^{9}{\rm M}_\odot$

$f_{\rm mod} = 1$

$\Delta M_* = \dot{M}_* \Delta t = 2000{\rm M}_\odot{\rm yr}^{-1} \times 11{\rm Myr} = 2.2 \times 10^{10}\ {\rm M}_\odot$

Semi-Analytic Model



Baryonic Physics

Gas infall
Gas cooling
Star Formation
Supernovae
Metal Enrichment
Black Hole Growth
Reionization Heating
AGN Feedback
Star Burst
…

Semi-analytic model

$\Omega_{\rm m}, \Omega_{\rm b} = 0.308, 0.0484$ (Planck15)

$f_{\rm b} = \Omega_{\rm b}/\Omega_{\rm m} = 0.157$

$M_{\rm vir} = 10^{15}{\rm M}_\odot$

$M_{\rm hot} = 10^{14.13}{\rm M}_\odot$

$M_{\rm cold} = 10^{13.3218}{\rm M}_\odot$

$M_* = 10^{12.01}{\rm M}_\odot$

$M_{\rm eject} = 10^{11}{\rm M}_\odot$

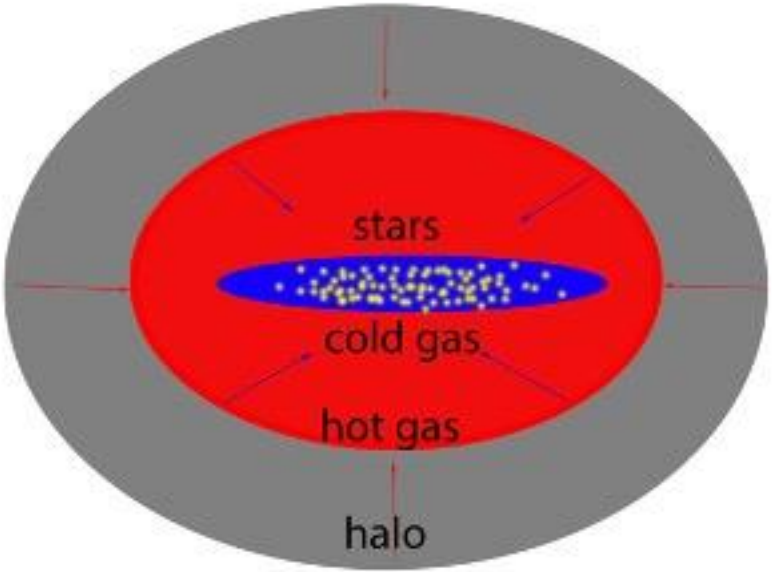
$M_{\rm bh} = 10^{9}{\rm M}_\odot$

$M_{\rm bh,acc} = 10^{9}{\rm M}_\odot$

$f_{\rm mod} = 1$

$\Delta M_* = \dot{M}_* \Delta t = 2000{\rm M}_\odot{\rm yr}^{-1} \times 11{\rm Myr} = 2.2 \times 10^{10}\ {\rm M}_\odot$

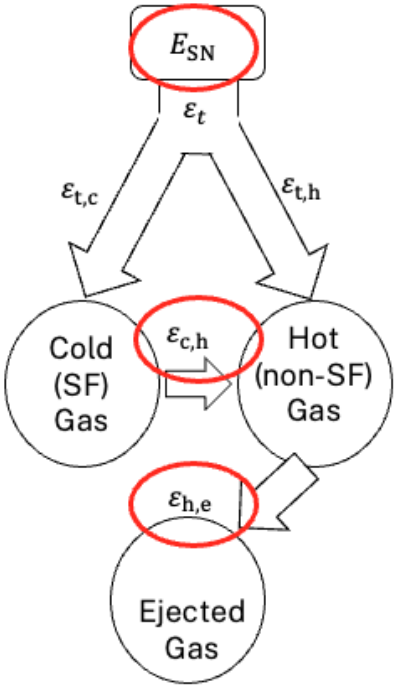
Semi-Analytic Model



Baryonic Physics

Gas infall
Gas cooling
Star Formation
Supernovae
Metal Enrichment
Black Hole Growth
Reionization Heating
AGN Feedback
Star Burst
...

Semi-analytic model



$$\mathrm{E_{SN}} = \eta_{\mathrm{SN}} \Delta M_* \times 10^{51} \mathrm{erg}$$

$$\varepsilon_{\mathrm{energy}} \mathrm{E_{SN}} = \frac{1}{2} \Delta M_{\mathrm{hot}} V_{\mathrm{vir}}^2 + \frac{1}{2} \Delta M_{\mathrm{eject}} \left(V_{\mathrm{esc}}^2 - V_{\mathrm{vir}}^2 \right)$$

$$= \frac{1}{2} \varepsilon_{\mathrm{mass}} \Delta M_* V_{\mathrm{vir}}^2 + \frac{1}{2} \Delta M_{\mathrm{eject}} V_{\mathrm{vir}}^2$$

Stellar recycling:

$$\Delta M_* = -0.01 \, \Delta M_* = -2.2 \times 10^8 \, \mathrm{M}_\odot$$

$$\Delta M_{\mathrm{cold}} = 0.01 \, \Delta M_* = 2.2 \times 10^8 \, \mathrm{M}_\odot$$

SN heating:

$$\Delta M_{\mathrm{hot}} = \varepsilon_{\mathrm{mass}} \Delta M_* = 6 \Delta M_* = 1.32 \times 10^{11} \, \mathrm{M}_\odot$$

$$\Delta M_{\mathrm{cold}} = -6 \, \Delta M_* = -1.32 \times 10^{11} \, \mathrm{M}_\odot$$

SN ejecting (given $\varepsilon_{\mathrm{energy}}$):

$$\Delta M_{\mathrm{eject}} = 10^{11} \mathrm{M}_\odot$$

$$\Delta M_{\mathrm{hot}} = 10^{11} \mathrm{M}_\odot$$

Semi-analytic model

$\Omega_{\rm m}, \Omega_{\rm b} = 0.308, 0.0484$ (Planck15)

$f_{\rm b} = \Omega_{\rm b}/\Omega_{\rm m} = 0.157$

$M_{\rm vir} = 10^{15}{\rm M}_\odot$

$M_{\rm hot} = 10^{14.1301}{\rm M}_\odot$

$M_{\rm cold} = 10^{13.3191}{\rm M}_\odot$

$M_* = 10^{12.0099}{\rm M}_\odot$

$M_{\rm eject} = 10^{11.3}{\rm M}_\odot$

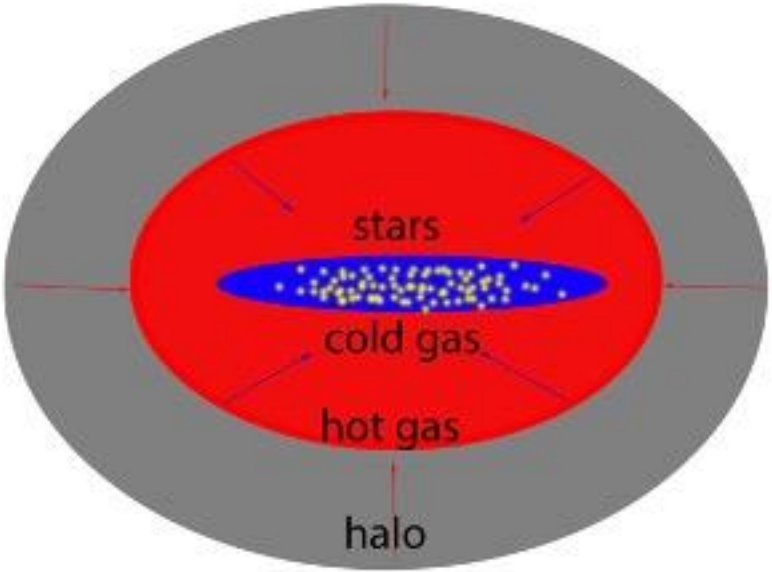
$M_{\rm bh} = 10^{9}{\rm M}_\odot$

$M_{\rm bh,acc} = 10^{9}{\rm M}_\odot$

$f_{\rm mod} = 1$

$\Delta M_* = \dot{M}_* \Delta t = 2000{\rm M}_\odot{\rm yr}^{-1} \times 11{\rm Myr} = 2.2 \times 10^{10}\ {\rm M}_\odot$

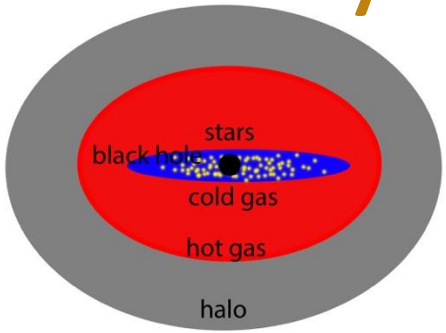
Semi-Analytic Model

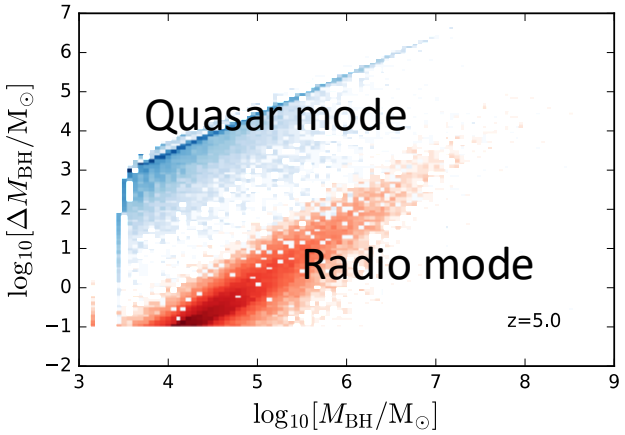


Baryonic Physics

Gas infall
Gas cooling
Star Formation
Supernovae
Metal Enrichment
Black Hole Growth
Reionization Heating
AGN Feedback
Star Burst
...

Semi-analytic model





Radio mode (accretion from hot gas)

- Bondi-Hoyle accretion rate:

$$\dot{m_{\rm bh}} \approx \kappa_{\rm h} 5.9 G m_{\rm bh} \mu m_{\rm p} k T_{\rm hot} \Lambda^{-1}$$

- AGN heating

$$m_{\rm heat} = \kappa_{\rm r} \eta \dot{m_{\rm bh}} \Delta t \; c^2 \times 2 V_{\rm vir}^{-2}$$

- Black hole growth

$$\Delta m_{\rm bh} = (1 - \eta) \; \dot{m_{\rm bh}} \Delta t$$

Quasar mode (accretion from cold gas)

- Triggered by merger:

$$\Delta m_{\rm bh,acc} = \kappa_{\rm c} \gamma m_{\rm cold} \left(1 + 280 {\rm km \; s^{-1}} V_{\rm vir}^{-1}\right)^{-2}$$

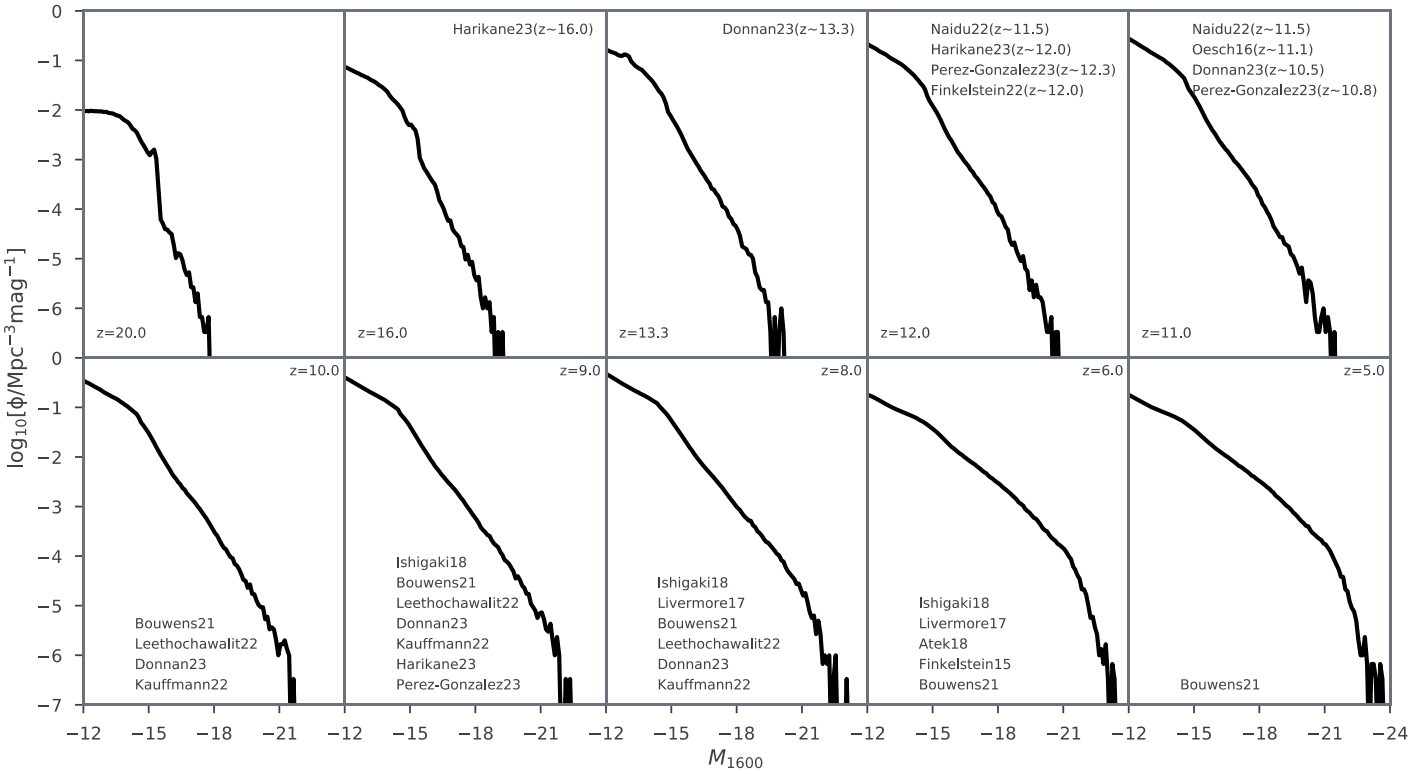
- Limited by Eddington luminosity:

$$\eta \dot{m}_{\rm bh} \, c^2 = \epsilon m_{\rm bh} c^2 t_{\rm Edd}^{-1}$$

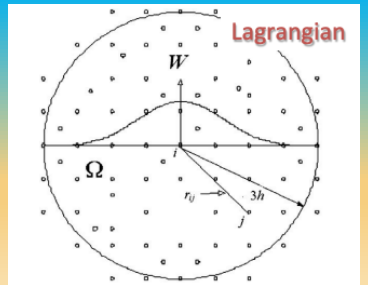
- Observable:

$$L_{\rm bol} = \epsilon m_{\rm bh} c^2 t_{\rm Edd}^{-1}$$

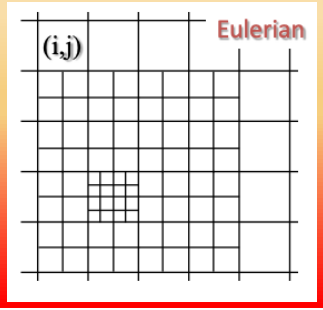
Population study using SAM



Summary



SPH
vs
AMR



magneto-hydrodynamic simulation

hydrodynamic simulation

magnetic field

N-body simulation

Gravity

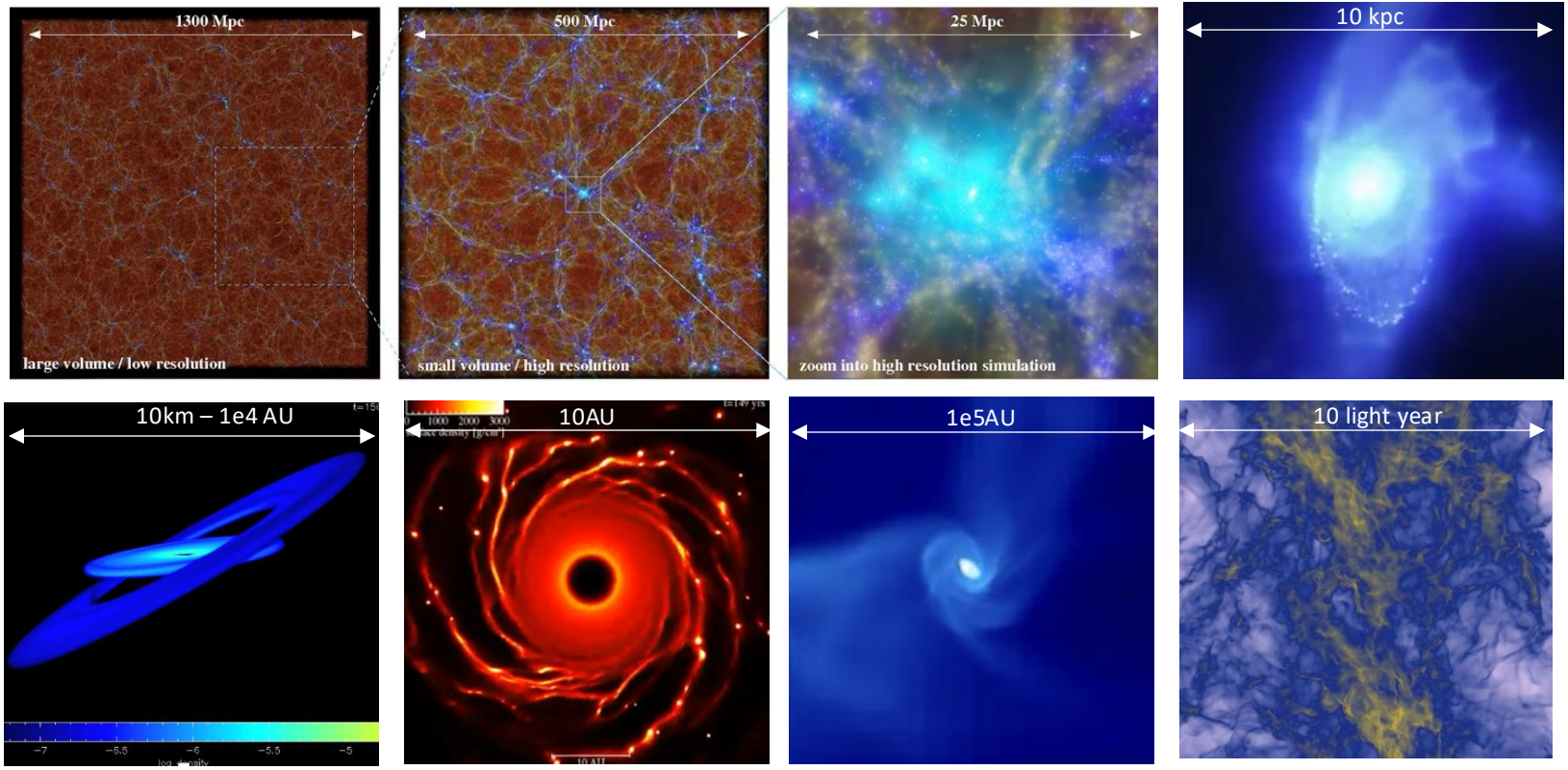
hydrody namics

radiative transfer

Semi-analytic model

radiation-magneto-hydrodynamic simulation

Summary



prerequisite

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

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

Install ipympl, matplotlib, numpy, astropy, hmf, scipy, commah