Monte-Carlo Radiation Transport

Refs: arXiv:2103.16588, arXiv:2209.02538

April 17, 2025

RIT | College of Science
Center for Computational Relativity and Gravitation

Outline

Outline

Distribution Function

When evolving the equations of radiation transport in general relativistic simulations, we aim to determine the distribution function of particles

$$f(t, x^i, p^\mu) = \sum_k \delta^3(x^i - x_k{}^i(t))\delta^3(p_i - p_i{}^k(t)). \tag{1}$$

The distribution function follows Boltzmann's equation of radiation transport

$$p^\alpha \left[\frac{\partial f}{\partial x^\alpha} - \Gamma^\beta{}_{\alpha\gamma} p^\gamma \frac{\partial f}{\partial x^\beta}\right] = \left[\frac{df}{d\tau}\right]_{\text{collisions}}. \tag{2}$$

- Evolve a six-dimensional function in time

Monte Carlo Methods

Discretize the distribution function of neutrinos using Monte-Carlo packets, each representing a large number of neutrinos, i.e.

$$f(t, x^i, p^\mu) = \sum_{k=0}^{n_p} N_k \delta^3(x^i - x_k^i(t)) \delta^3(p_i - p_i^k(t)) \tag{3}$$

- Each packet represent N_k particles and has a single position and momentum

Outline

Propagation of Packets

Neglecting the finite mass of neutrinos, we evolve packets along null geodesics (Hughes et al. 1994),

$$\frac{dx^i}{dt} = \gamma^{ij}\frac{p_j}{p^t} - \beta^i, \tag{4}$$

$$\frac{dp_i}{dt} = -\alpha p^t \partial_i \alpha + p_j \partial_i \beta^i - \frac{1}{2} p_j p_k \partial_i \gamma^{jk}, \tag{5}$$

for a null vector, $p^t = \frac{\sqrt{\gamma^{ij} p_i p_j}}{\alpha}$.

Outline

Tabulated Reaction Rate

Use the **NuLib** library (O'Connor & Ott 2010) to generate tabulated values of the **emissivity** η, **absorption opacity** κ_a, and **elastic scattering opacity** κ_s experienced by Monte Carlo packets.

- Charged current reactions

$$p + e^- \leftrightarrow n + \nu_e, \quad n + e^+ \leftrightarrow p + \bar{\nu}_e. \tag{6}$$

- Pair production/annihilation

$$e^+ e^- \leftrightarrow \nu\bar{\nu}. \tag{7}$$

- Nucleon-nucleon bremsstrahlung

$$N + N \leftrightarrow N + N + \nu + \bar{\nu}. \tag{8}$$

The output of the NuLib library, is a 4D table for η, κ_a and κ_s as a function of **fluid density** ρ, **fluid temperature** T, **fluid electron fraction** Y_e, and **neutrino energy** ν.

Emission

The total energy of the emitted neutrinos of that species and in that energy bin is

$$E_{\text{tot}} = \sqrt{-g} V \Delta t \eta, \tag{9}$$

If the desired energy of neutrino packets within this cell is E_{target} and the central value of the neutrino energies in our energy bin is ν,

- we emit, on average, $E_{\text{tot}}/E_{\text{target}}$ packets, each representing E_{target}/ν neutrinos,
- the location of the packet is randomly drawn from a homogeneous distribution *in the coordinates of the simulation*,
- the 4-momentum of the neutrinos is drawn from an isotropic distribution in the fluid frame,

$$p^{\hat{\mu}}_{\text{fl}} = \nu(1, \sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta). \tag{10}$$

Absorption and Elastic Scattering

Before propagating a neutrino, we draw random numbers r_a, r_s from a homogeneous distribution in $[\epsilon, 1)$ (with $\epsilon = 10^{-70}$). The time to the next absorption/scattering event is then

$$\Delta t_{a,s} = -\frac{\kappa_{a,s} p^t}{\nu} \ln r_{a,s}, \tag{11}$$

Consider the smallest time interval between $\Delta t_{a,s}$ and the desired time step

- If Δt_a is the smallest, the packet is absorbed.
- If Δt_s is the smallest, the packet is scattered.
  - randomly draw a new 4-momentum with the same fluid-frame energy as the original packet
  - a direction of propagation drawn from an isotropic distribution

This simple process works well as long as $\kappa_{a,s}\Delta t \lesssim 1$, i.e., when individual grid cells are optically thin or semitransparent.

Pair Annihilation in Low-density Regions

- The absorption opacity for pair annihilation

$$\kappa_p = \frac{C_{\text{pair}} c C_F^2}{3\pi} \frac{p^\alpha p^\beta}{\nu} \bar{T}_{\alpha\beta}. \tag{12}$$

$$\bar{\kappa}_p = \frac{C_{\text{pair}} c C_F^2}{3\pi} \frac{\bar{p}^\alpha \bar{p}^\beta}{\bar{\nu}} T_{\alpha\beta}. \tag{13}$$

- This estimate of κ_p is only valid in low-density regions, we choose to suppress κ_p in dense regions,

$$\kappa_p \to \kappa_p e^{-\rho/\rho_{\text{crit}}}. \tag{14}$$

Pair Annihilation in Low-density Regions

- We take κ_p into account by correcting the number of neutrinos N represented by any given packet at the end of a time step

$$\frac{dN}{dt} = -\kappa_p N \rightarrow N(t) = N_0 e^{-\kappa_p (t - t_0)}. \tag{15}$$

  After a time step Δt, we thus have $N(t_0 + \Delta t) = N(t_0) e^{-\kappa_p \Delta t}$.

- Pair annihilation is treated differently from other absorption processes: it never destroys Monte Carlo packets, but **only changes the number of neutrinos in each packet**.

Outline

Stress-energy Tensor and Moments

The general relativistic stress-energy tensor is,

$$T_{\mu\nu}(t, x^i) = \sum_{k=1}^{n_p} N_k \frac{p^k{}_\mu p^k{}_\nu}{\sqrt{-g} p_k{}^t} \delta^3(x^i - x_k{}^i(t)). \tag{16}$$

An observer with 4-velocity u^μ measures the corresponding energy density

$$J = T_{\mu\nu} u^\mu u^\nu = \sum_{k=1}^{n_p} N_k \frac{\nu_k^2}{\sqrt{-g} p_k{}^t} \delta^3(x^i - x_k{}^i(t)), \tag{17}$$

with $\nu_k = -p^k{}_\mu u^\mu$ the energy of neutrinos in packet k as measured by our observer.

Stress-energy Tensor and Moments

The average energy density within a region of coordinate volume V is then

$$J = \sum_{k \in V} N_k \frac{\nu_k^2}{\sqrt{-g} V {p_k}^t}. \tag{18}$$

Similarly, the average linear momentum $H_\alpha = -T^{\mu\nu} u_\mu (g_{\nu\alpha} + u_\nu u_\alpha)$ measured by an observer with u^μ is

$$H_\alpha = \sum_{k \in V} N_k \frac{\nu_k ({p^k}_\alpha - \nu_k u_\alpha)}{\sqrt{-g} V {p_k}^t}. \tag{19}$$

Stress-energy Tensor and Moments

To couple particles with the fluid, we will also need to compute terms of the form

$$\int dt\kappa J = \sum_{k=1}^{n_p} \kappa_k N_k \frac{\nu_k^2}{\sqrt{-g}{p_k}^t} \Delta t_k \delta^3(x^i - {x_k}^i(t)), \tag{20}$$

with κ an opacity that depends on position and momentum of a particle. The average of this integral within a grid cell of volume V,

$$\int dt\kappa J = \sum_{k \in V} \kappa_k N_k \frac{\nu_k^2}{\sqrt{-g} V {p_k}^t} \Delta t_k. \tag{21}$$

Outline

Source Terms for Fluid Evolution

The source terms appearing in the evolution of these variables due to neutrino-matter interactions are

$$\partial_t \tilde{\tau} = ... + \alpha\sqrt{\gamma} S^\alpha n_\alpha, \tag{22}$$

$$\partial_t \tilde{S}_i = ... - \alpha\sqrt{\gamma} S^\alpha \gamma_{\alpha i}, \tag{23}$$

$$\partial_t (\rho_* Y_e) = ... - \sum s_i m_p \alpha\sqrt{\gamma} \frac{\eta - \kappa_a J}{\nu}. \tag{24}$$

The sourece term is,

$$S^\alpha = \sum \eta' u^\alpha - \sum \left(\kappa'_a J u^\alpha + (\kappa'_a + \kappa'_s) H^\alpha \right). \tag{25}$$

Calculating the changes in the evolved fluid variables over one time step Δt thus requires calculations of

$$\int \kappa'_a J dt; \quad \int (\kappa'_a + \kappa'_s) H^\alpha dt; \quad \int \kappa'_a J dt/\nu, \tag{26}$$

Outline

Main Feature

- Particle Packets: (x^i, p^μ, N_k, species, ...),
- Propagation of Packets,
- Emission, Absorption, Scattering, ...,
- Interaction with Fluid.

Particle and ParticleContainter

```
Particle<2, 2> p;

p.pos(0) = 1.0;
p.pos(1) = 2.0;
p.pos(2) = 3.0;
p.id() = 1;
p.cpu()  = 0;

// p.rdata(0) is the first extra real component, not the
// first real component overall
p.rdata(0) = 5.0;
p.rdata(1) = 5.0;

// and likewise for p.idata(0);
p.idata(0) = 17;
p.idata(1) = -64;
```

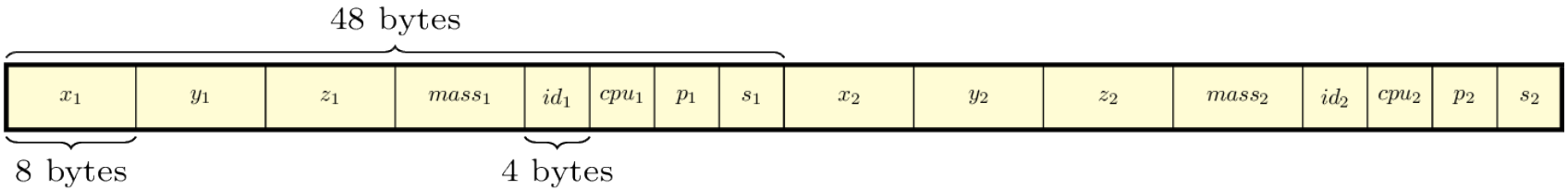
Particle and ParticleContainter

```
using MyParticleContainer = ParticleContainer<3, 2, 4, 4>;
MyParticleContainer mypc;
```

- For the first two template parameters, particles added to the container are stored in the Array-of-Structs style.
- There are two more optional template parameters that allows the user to specify additional particle variables that will be stored in Struct-of-Array form.

Arrays-of-Structs and Structs-of-Arrays

Array-of-Structs



Struct-of-Arrays

Constructing ParticleContainers

```
ParticleContainer (const Geometry             & geom,
                   const DistributionMapping & dmap,
                   const BoxArray             & ba);
```

```
ParticleContainer (const Vector<Geometry>             & geom,
                   const Vector<DistributionMapping> & dmap,
                   const Vector<BoxArray>             & ba,
                   const Vector<int>                  & rr);
```

Redistribute

After calling `Redistribute()`

- all the particles will be moved to their proper places in the container,
- all invalid particle (particles with id set to -1) will be removed.

Initializing Particle Data

```
for (MFIter mfi = MakeMFIter(lev); mfi.isValid(); ++mfi) {

    // ``particles'' starts off empty
    auto& particles = GetParticles(lev)[std::make_pair(mfi.index(),
                                                       mfi.LocalTileIndex())];

    ParticleType p;
    p.id()   = ParticleType::NextID();
    p.cpu()  = ParallelDescriptor::MyProc();
    p.pos(0) = ...
    etc...

    // AoS real data
    p.rdata(0) = ...
    p.rdata(1)  = ...

    // AoS int data
    p.idata(0) = ...
    p.idata(1) = ...
```

Initializing Particle Data

```
    // Particle real attributes (SoA)
    std::array<double, 2> real_attribs;
    real_attribs[0] = ...
    real_attribs[1] = ...

    // Particle int attributes (SoA)
    std::array<int, 2> int_attribs;
    int_attribs[0] = ...
    int_attribs[1]  = ...

    particles.push_back(p);
    particles.push_back_real(real_attribs);
    particles.push_back_int(int_attribs);

    // ... add more particles if desired ...
}
```

Iterating over Particles

- Iterate over all the AoS data

```
using MyParConstIter = MyParticleContainer::ParConstIterType;
for (MyParConstIter pti(pc, lev); pti.isValid(); ++pti) {
    const auto& particles = pti.GetArrayOfStructs();
    for (const auto& p : particles) {
        // do stuff with p...
    }
}
```

- Access the SoA data

```
using MyParIter = MyParticleContainer::ParIterType;
for (MyParIter pti(pc, lev); pti.isValid(); ++pti) {
    auto& particle_attributes = pti.GetStructOfArrays();
    RealVector& real_comp0 = particle_attributes.GetRealData(0);
    IntVector&  int_comp1  = particle_attributes.GetIntData(1);
    for (int i = 0; i < pti.numParticles; ++i) {
        // do stuff with your SoA data...
    }
}
```

Interacting with Mesh Data

```
Ex.FillBoundary(gm.periodicity());
Ey.FillBoundary(gm.periodicity());
Ez.FillBoundary(gm.periodicity());
for (MyParIter pti(MyPC, lev); pti.isValid(); ++pti) {
    const Box& box = pti.validbox();

    const auto& particles = pti.GetArrayOfStructs();
    int nstride = particles.dataShape().first;
    const long np  = pti.numParticles();

    const FArrayBox& exfab = Ex[pti];
    const FArrayBox& eyfab = Ey[pti];
    const FArrayBox& ezfab = Ex[pti];

    interpolate_cic(particles.data(), nstride, np,
                    exfab.dataPtr(), eyfab.dataPtr(), ezfab.dataPtr(),
                    box.loVect(), box.hiVect(), plo, dx, &ng);
}
```

Interacting with Mesh Data

```
rho.setVal(0.0, ng);
for (MyParIter pti(*this, lev); pti.isValid(); ++pti) {
    const Box& box = pti.validbox();

    const auto& particles = pti.GetArrayOfStructs();
    int nstride = particles.dataShape().first;
    const long np  = pti.numParticles();

    FArrayBox& rhofab = (*rho[lev])[pti];

    deposit_cic(particles.data(), nstride, np, rhofab.dataPtr(),
                box.loVect(), box.hiVect(), plo, dx);
}

rho.SumBoundary(gm.periodicity());
```