Ioffe Institute of Russian Academy of Science

FAINA

Numerical code for modeling electromagnetic radiationfrom astrophysical sources

User’s guide

Saint-Petersburg — 2023

Contents

Introduction

FAINA - is a numerical code for modeling different types of electromagnetic radiation of astrophysical source. It is written in C++ and supports parallel computations using openmp method. FAINA allows to model observable fluxes from sources with different parameters and geometries vai different emission mechanisms, and also to optimize source parameters to fit observational data.

Installation

Current version of the code is available on github https://github.com/VadimRomansky/Faina. FAINA is distributed freely under the MIT public license. Download the archive with code and extract it into preferred root directory.

Windows

With Windows OS it is recomended to use Microsoft Visual Studio and open solution Faina.sin with it. Operability was examined for Windows 10 and Visual Studio 2022 version.

Unix

There are two possible ways to run FAINA on Unix. We recommend to use IDE QtCreator and open with it file Faina.pro located in the root directory.

Other way is to use FAINA from terminal. To compile and run it you can use following commands

```
$ make
$ ./Faina
```

Operability was examined for Ubuntu 22.04.

Runnng simple problem

Let see a simple example of solving radiation problem with faina. You can find in the function evaluateSimpleSynchrotron in the file /Src/examples.cpp. Synchrotron radiation from homogenous source with the shape of cylinder with axis along line of sight and with powerlaw electron distribution is evaluated in this example. But it demonstrates a general approach to evaluation of radiation with FAINA code.

Let define values of magnetic field and electron number density in the source (code uses CGS units).

```
double B = 1.0;
double electronConcentration = 1.0;
```

Then you need to create distribution of emitting electrons. There are a different type of particle distribution implemented in the code, let use isotropic powerlaw distribution for this example. You should call the constructor of MassiveParticlePowerLawDistribution with following parameters - mass of emitting particles (electrons in this case), powerlaw index of distribution, which is defined as positive number p in $F(E)$ $1/E^p$, starting energy of powerlaw distribution, and electrons number density.

```
MassiveParticleDistribution* distribution =
new MassiveParticlePowerLawDistribution(
massElectron, 3.0, me_c2, electronConcentration);
```

After that you should create a radiation source, for example it would be homogenous flat disk with axis along line of sight. You should call the constructor of SimpleFlatSource with following parameters: electrons distribution, magnetic field, sinus of it's inclination angle to the line of sight, radous of cylinder, it's hight and distance to the observer.

```
RadiationSource* source = new SimpleFlatSource(
distribution, B, 1.0, parsec, parsec, 1000 * parsec);
```

And the last thing you need is an radiation evaluator. They are different for every specific type of radiation. Here we create a SybchrotronEvaluator with following parameters: number of grid points for integration electron distribution function over energy, lower and upper limits of electron energy that will be taken into account and boolean parameter determining include synchrotron self absorption or not.

```
RadiationEvaluator* evaluator = new
SynchrotronEvaluator(1000, me_c2, 1000 * me_c2, true);
```

Synchrotron approximation is valid only for frequencies of radiation much greater than cyclotron frequency, so let evaluate it

```
double cyclOmega =
electron_charge * B / (massElectron * speed_of_light);
```

Now you can evaluate spectrum of synchrotron radiation. Radiation evaluator has a method writeFluxFromSourceToFile which allows to calculate flux energy density and write it into the file in units energy vs power per energy per area, or erg vs $sm^{-}2s^{-}1$. This method takes following input parametes: output file name which will be created or rewritten, lower and upper limits of energy range and number of grid points, which will be distributed logarithmically in the range.

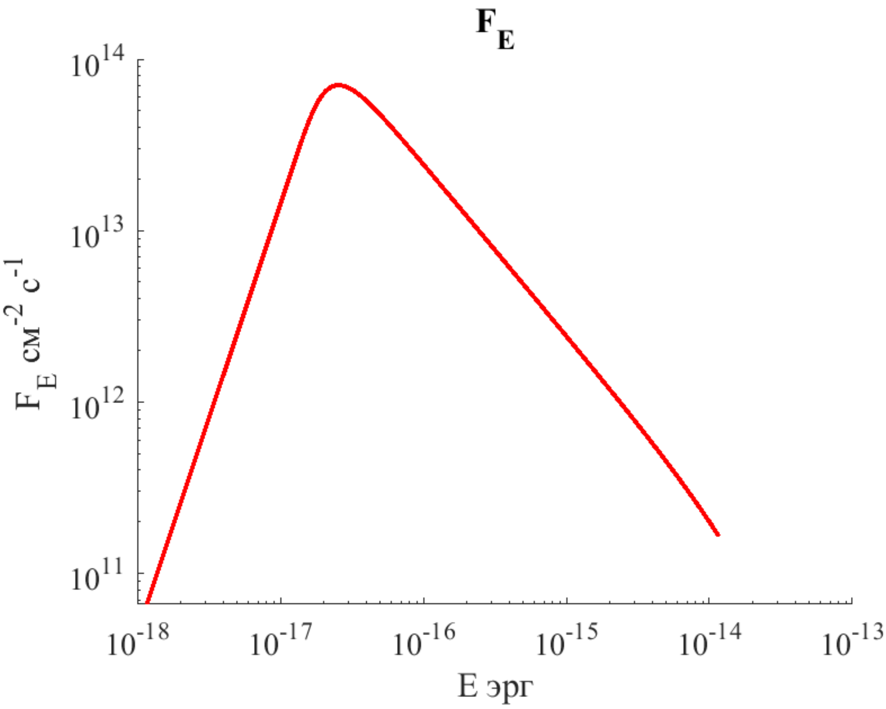

Figure 1: Synchrotron radiation flux energy density from test source

If you need other units you should use method evaluateFluxFromSource which provides a flux energy density at given energy and rewrite output.

```
evaluator->writeFluxFromSourceToFile("out.dat",source,
10*hplank*cyclOmega, 1E5*hplank*cyclOmega, 1000);
```

Evaluated spectrum of flux energy density from this source is shown in 1. Examples of plotting scripts you can find in Figure directory pyFAINA.

Chapter 1

Evaluation the radiation of the sources

FAINA allows to evaluate electromagnetic radiation from sources with various type of particle distributions and different parameters such as magnetic fiels, number density and other. In this chapter we describe creating various types of radiation sources.

1.1 Particle distributions

Crucial parameter for evaluation of any type of radiation is a distribution function of emitting particles. In the FAINA code abstract class ParticleDistribution and derived classes are used for representation of distributions. Public methods of class ParticleDistribution are listed in Table 1.1:

Table 1.1: Public methods of ParticleDistribution class

ParticleDistribution	abstract class for particle distributions
double distribution(const double& energy, const double& mu, const double& phi)	returns probability density function in polar coordinates with given energy, cosinus of polar angle and azimutal angle, normalized to the particles number density
virtual double distributionNormalized(const double& energy, const double& mu, const double& phi)	virtual method, returns probability density function in polar coordinates with given energy, cosinus of polar angle and azimutal angle, normalized to unity
virtual double getMeanEnergy()	virtual method, returns mean energy of particles in distribution
double getConcentration()	returns particles number density
void resetConcentration(const double& concentration)	changes number density to the given value

For creating a distribution object you need some inherited class. Inheritance tree of ParticleDistribution splits into two big branches - PhotonDistribution for distribution of photons, and MassiveParticleDistribution - for massive particles. Scheme of class hierarchy is shown in Figure 1.1.

It is important to note, that photons distributions are not used to represent results of evaluation of electromagnetic radiation. They are necessary only as input parameter for evaluation of inverse Compton scattering. Class PhotonDistribution is only an interface and

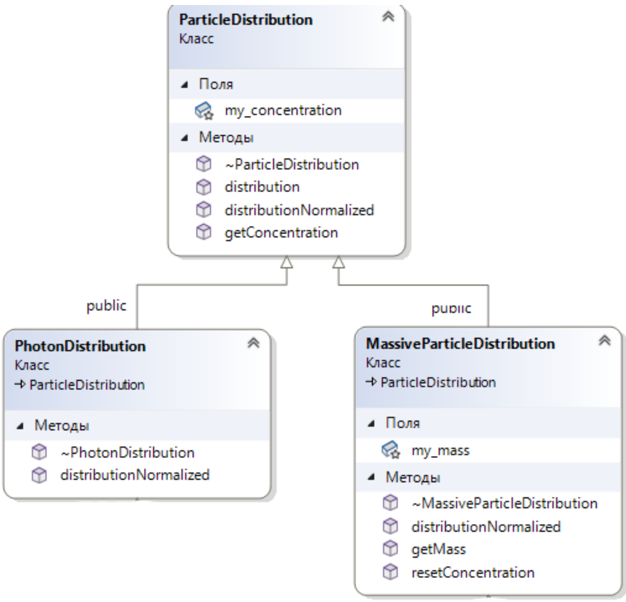

Figure 1.1: Two branches of inheritance tree of ParticleDIstribution

has not its own specific methods. Class MassiveParticleDistribution is also abstract, but his methods are listed in Table 1.2

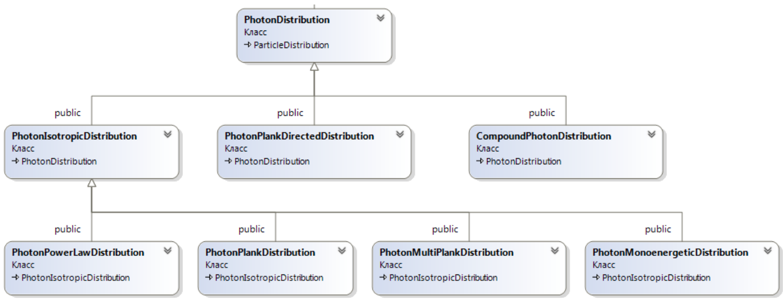

Figure 1.2: Class hierarchy of photon distributions

Table 1.2: Public methos of MassiveParticleDistribution class

MassiveParticleDistribution	abstract class for massive particles distribution
virtual double minEnergy()	virtual method, returns the lowest possible energy of particle in this distribution
virtual double maxEnergy()	virtual method, returns the upper limit of energy of particle in this distribution. NOTE that if upper limit of energy is infinite, this method returns negative number
double getMass()	returns mass of single particle

1.1.1 Photon distributions

Abstract class PhotonDistribution has following derived class: abstract PhotonIsotropicDistribution, which represented isotopic distributions and some non-abstract classes: PhotonPlankDirectedDistribution, which represent photons with Plank distribution with respect to energy, but collimated in some solid angle, and CompoundPhotonDistribution, which is usefull for sum of several arbitrary photon distributions.

Class PhotonIsotropicDistribution again has its own inherited classes. It is a PhotonPowerLawDistribution for powerlaw distribution, PhotonPlankDistribution for Plank distributions, PhotonMultiPlankDistribution for sum of several Plank distributions and PhotonMonoenergeticDistribution for isotropic photons with same energy. Class hoerarchy of photon distributions is presented in Figure 1.2.

Methods of PhotonDistribution and it's inherited classes are listed in Table 1.3. NOTE, that metods distributionNormalized(const double& energy) and distribution(const double& energy) are not distribution with respect to energy, but just full distribution with dropped angular arguments. So to obtain distribution with respect to energy one should multiply result of this functions by 4π.

Table 1.3: Public methods of PhotonDistribution class and derived classes

PhotonDistribution	abstract interface for photon distributions

PhotonIsotropicDistribution	abstract class for isotropic distributions of photons
double distribution(const double& energy)	returns probability density function in polar coordinates with dropped angular arguments (normalized to the number density divided by 4π)
virtual double distributionNormalized(const double& energy)	virtual method, returns probability density function in polar coordinates with dropped angular arguments (mormalized to the $1/4\pi$)
void writeDistribution(const char* fileName, int Ne, const double& Emin, const double& Emax)	writes distribution into given file as to columns - energy and distribution from Emin to Emax with Ne logarithmically distributed points
PhotonPowerLawDistribution	class representing powerlaw distribution of photons
PhotonPowerLawDistribution(const double& index, const double& E0, const double& concentration)	constructor, creates distribution with given power-law index p such as $F(E)\ 1/E^p$, starting energy and number density
double getIndex()	returns power-law index
double getE0()	returns starting energy of distribution
PhotonPlankDistribution	class representing Plank distribution
PhotonPlankDistribution(const double& temperature, const double& amplitude)	constructor, creates distribution with given temperature and amplitude - relation of number density to the number density of photons in equilibrium black-body radiation
static PhotonPlankDistribution* getCMBRadiation()	static method, returns object representing Cosmic Microwave Background Radiation (temperature 2.725 K, amplitude 1)
double getTemperature()	returns temperature of distribution
PhotonMultiPlankDistribution	class representing sum of several Plank distributions
PhotonMultiPlankDistribution(int N, const double* const temperatures, const double* const amplitudes)	constructor, creates distribution constisting of N plank distributions with given temperatures and amplitudes
static PhotonMultiPlankDistribution* getGalacticField()	static method, returns object representing mean Galactic photon field described in [1]. This distribution consists of five plank components with temperatures $2.725K, 20K, 3000K, 4000K, 7000K$ and amplitudes $1.0, 4\cdot 10^{-4}, 4\cdot 10^{-13}, 1.65\cdot 10^{-13}, 1.0\cdot 10^{-14}$ respectively

PhotonMonoenergeticDistribution	class representing population of isotropic photons with close energy
PhotonMonoenergeticDistribution(const double& Energy, const double& halfWidth, const double& concentration)	constructor, creates object with given mean energy, half-width of uniform distribution around mean energy and number density
CompoundPhotonDistribution	class representing sum of several arbitrary distributions
CompoundPhotonDistribution(int N, PhotonDistribution** distributions)	constructor, creates distribution consisting of N arbitrary distributions
CompoundPhotonDistribution(PhotonDistribution* dist1, PhotonDistribution* dist2)	constructor, creates distribution which is sum of two given distributions
CompoundPhotonDistribution(PhotonDistribution* dist1, PhotonDistribution* dist2, PhotonDistribution* dist3)	constructor, creats distribution which is sum of three given distributions
PhotonPlankDirectedDistribution	class representing distribution which is Plank-like with respect to energy, but collimated into given direction
PhotonPlankDirectedDistribution(const double& temperature, const double& amplitude, const double& theta0, const double& phi0, const double& deltaTheta)	constructor, creates distribution with given temperature, amplitude, angles determining mean direction of photons and half-width angle of cone in which photons propagate
double getTemperature()	return temperature of distribution

User can define other photons distribution, creating class inherited from PhotonDistribution or PhotonIsotropicDistribution and overriding virtual method distributionNormalized.

1.1.2 Distributions of massive particles

Distributions of massive particles are represented by class MassiveParticleDistribution and inherited classes. Similarly to the photon distributions, isotropic distrinutions are important type, represented by class MassiveParticleIsotropicDistribution. This class also has methods distributionNormalized(const double& energy) and distribution(const double& energy), which are not distribution with respect to energy, but just full distribution with dropped angular arguments. So to obtain distribution with respect to energy one should multiply result of this functions by 4π.

Abstract class of isotropic distributions has seven inherited classes for specific distributions: MassiveParticlePowerLawDistribution - for power-law distributions, MassiveParticleBrokenPowerLawDistribution - for double power-law distibutions with knee, MassiveParticlePowerLawCutoffDistribution - for power-law distributions with exponential cutoff, MassiveParticleMaxwellDistribution - for non-relativistic maxwellian distribution (but it use full energy, including rest energy), MassiveParticleMaxwellJuttnerDistribution - for

Maxwell -Juttner distribution, MassiveParticleTabulatedIsotropicDistribution - for arbitrary distributions, described with array of values and MassiveParticleMonoenergeticDistribution - for beam of particles with close energies. Also there are six anisotropic distributions, implemented in the code. MassiveParticleTabulatedPolarDistribution - for tabulated distribution with dependence on energy and polar angle, MassiveParticleAnisotropicDistribution - for arbitrary tabulated anisotropic distributions, MassiveParticleMonoenergeticDirectedDistribution - for distributions represented narrow beam of particles with close energies, MassiveParticleMovingDistribution - for transformation the distributions from one frame to another, CompoundMassiveParticleDistribution - for sum of arbitraty distributions and CompoundWeightedMassiveParticleDistribution - for weighted sum of arbitrary distributions. In some cases operating with relative weghts of distributions is more useful than with absolute concentrations. Class hierarchy of distributions of massive particles is shown in Figure 1.3.

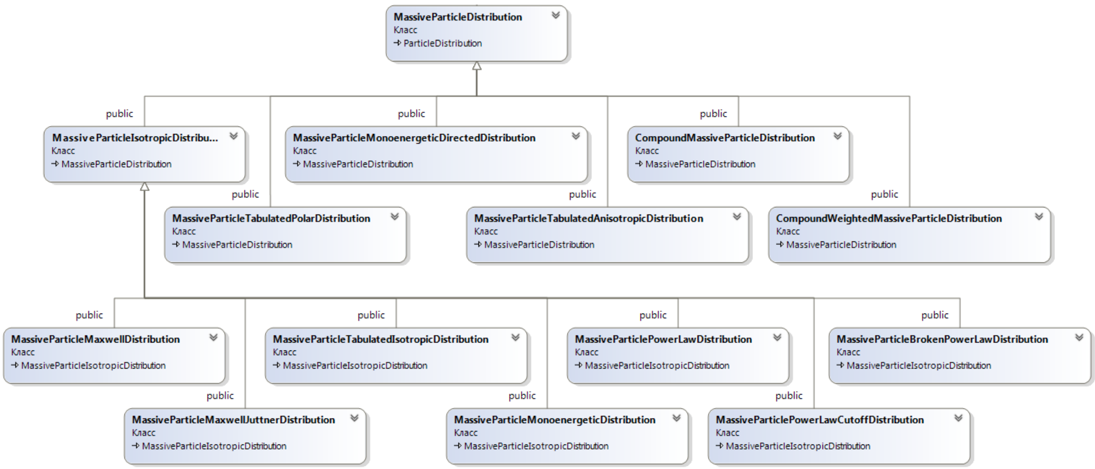

Figure 1.3: Class hierarchy of massive particles distributions

public methods of classes for massive particle distributions are listed in Table 1.4. User can define his own specific distributions, creating class inherited from MassiveParticleDistribution or MassiveParticleIsotropicDistribution.

Table 1.4: Public methods of classes derived from MassiveParticleDistribution

MassiveParticleIsotropicDistribution	Abstract class for isotropic distributions of massive particles
double distribution(const double& energy)	returns probability density function in polar coordinates with dropped angular arguments (normalized to the number density divided by 4π)

virtual double distributionNormalized(const double& energy)	virtual method, returns probability density function in polar coordinates with dropped angular arguments (mormalized to the $1/4\pi$)
void writeDistribution(const char* fileName, int Ne, const double& Emin, const double& Emax)	writes distribution into given file as to columns - energy and distribution from Emin to Emax with Ne logarithmically distributed points
double distributionNormalizedWithLosses(const double& energy, const double& lossRate, const double& time)	returns a distribution which evaluted till time t via synchrotron losses with equation $f_t(E) = f\left(\frac{E}{1-Elt}\right) \cdot \frac{1}{(1-Elt)^2}$ and loss rate $l = 4e^4B^2/9m^4c^7$
MassiveParticlePowerLawDistribution	class representibg power-law distribution
MassiveParticlePowerLawDistribution(const double& mass, const double& index, const double& E0, const double& concentration)	cobstructor, creates distribution with given particle mass, power-law index, starting energy and concentration
double getIndex()	returns power-law index
double getE0()	returns starting energy of distribution
MassiveParticleBrokenPowerLawDistribution	class representing double power-law distribution with knee
MassiveParticleBrokenPowerLawDistribution(const double& mass, const double& index1, const double& index2, const double& E0, const double& Etran, const double& concentration)	constructor, creates distribution with given particle mass, power-law indexes at low and high energies, starting energy, energy of transition from one index to another and concentration
double getIndex1()	returns power-law index at low energies
double getIndex2()	returns power-law index at high energies
double getE0()	returns starting energy of distribution
double getEtran()	returns energy of transition fron one index to another
MassiveParticlePowerLawCutoffDistribution	class representing power-law distribution with exponential cutoff
MassiveParticlePowerLawCutoffDistribution(const double& mass, const double& index, const double& E0, const double& beta, const double& Ecut, const double& concentration)	constructor, creates distribution with given particle mass, power-law index, starting energy, power of cutoff and cutoff energy and concentration $F(E) \propto (E/E_0)^{-index} \cdot \exp(-(E/E_{cut})^{\beta})$

double getIndex()	returns power-law index
double getBeta()	returns cutoff power parameter
double getE0()	returns starting energy of distribution
double getEcutoff()	returns cutoff energy
MassiveParticleMaxwellDistribution	class representing non-relativistic maxwellian distribution
MassiveParticleMaxwellDistribution(const double& mass, const double& temperature, const double& concentration)	creates distribution with given particles mass, temperature and concentration
double getTemperature()	returns temperature
MassiveParticleMaxwellJuttnerDistribution	class representing Maxwell-Juttner distribution
MassiveParticleMaxwellJuttnerDistribution(const double& mass, const double& temperature, const double& concentration)	creates distribution with given particle mass, temperature and concentration
double getTemperature()	returns temperature
MassiveParticleTabulatedIsotropicDistribution	class for tabulated isotropic distribution
MassiveParticleTabulatedIsotropicDistribution(const double& mass, const char* fileName, const int N, const double& concentration, DistributionInputType inputType)	constructor, creates distribution with given mass and concentraion, reading table with N lines from given file. inpuType - enum variable determining in which coordinates distribution is defined in file
MassiveParticleTabulatedIsotropicDistribution(const double& mass, const char* energyFileName, const char* distributionFileName, const int N, const double& concentration, DistributionInputType inputType)	constructor, creates distribution with given mass and concentraion, reading Nx2 table with from two given files. inpuType - enum variable determining in which coordinates distribution is defined in files
MassiveParticleTabulatedIsotropicDistribution(const double& mass, const double* energy, const double* distribution, const int N, const double& concentration, DistributionInputType inputType)	constructor, creates distribution with given mass and concentraion, reading two data columns from given arrays. inpuType - enum variable determining in which coordinates distribution is defined in arrays
int getN()	returns number of grid points in distribution array
double rescaleDistribution(const double& k)	rescales distribution through the energy axis using fourmula $E' = mc^2 + k \cdot (E - mc^2)$, $F(E') = F(E)/k$. It may be useful when e.g. distribution of electrons is obtained by numerical code with increased electron mass

void addPowerLaw(const double& Epower, const double& index)	replaces the tail of distribution with power-law distribution with given spectral index starting from Epower. Also renorms distribution
MassiveParticleMonoenergeticDistribution	class representing population of isotropic particles with close energy
MassiveParticleMonoenergeticDistribution(const double& mass, const double& Energy, const double& halfWidth, const double& concentration)	constructor, creates distribution with given particle mass, mean energy, half-width of uniform distribution around mean energy and number density
MassiveParticleTabulatedPolarDistribution	class for tabulated distribution with dependence on energy and polar angle
MassiveParticleTabulatedPolarDistribution(const double& mass, const char* energyFileName, const char* muFileName, const char* distributionFileName, const int Ne, const int Nmu, const double& concentration, DistributionInputType inputType)	constuctor, creates distribution with given particle mass and concentration, reading it from files with energy grid points, angular grid points and distribution. inpuType - enum variable determining in which coordinates distribution is defined in files
MassiveParticleTabulatedPolarDistribution(const double& mass, const double* energy, const double* mu, const double** distribution, const int Ne, const int Nmu, const double& concentration, DistributionInputType inputType)	constuctor, creates distribution with given particle mass and concentration, using arrays with energy grid points, angular grid points and distribution. inpuType - enum variable determining in which coordinates distribution is defined in arrays
int getNe()	returns number of energy grid points in distribution array
int getNmu()	returns number of polar angle grid points in distribution array
void double rescaleDistribution(const double& k)	rescales distribution through the energy axis using fourmula $E' = mc^2 + k \cdot (E - mc^2)$, $F(E', \mu) = F(E, \mu)/k$. It may be useful when e.g. distribution of electrons is obtained by numerical code with increased electron mass

MassiveParticleTabulatedAnisotropicDistribution	class for arbitrary tabulated distribution
MassiveParticleTabulatedAnisotropicDistribution(const double& mass, const char* energyFileName, const char* muFileName, const char* distributionFileName, const int Ne, const int Nmu, const int Nphi, const double& concentration, DistributionInputType inputType)	constuctor, creates distribution with given particle mass and concentration, reading it from files with energy grid points, angular grid points and distribution. Grid with respect to azimuthal angle considered uniform and depends only on number of drid points Nphi. inpuType - enum variable determining in which coordinates distribution is defined in files
MassiveParticleTabulatedAnisotropicDistribution(const double& mass, const double* energy, const double* mu, const double*** distribution, const int Ne, const int Nmu, const int Nphi, const double& concentration, DistributionInputType inputType)	constuctor, creates distribution with given particle mass and concentration, using arrays with energy grid points, angular grid points and distribution. Grid with respect to azimuthal angle considered uniform and depends only on number of drid points Nphi. inpuType - enum variable determining in which coordinates distribution is defined in arrays
int getNe()	returns number of energy grid points in distribution array
int getNmu()	returns number of polar angle grid points in distribution array
int getNphi()	returns number of azimuthal angle grid points in distribution array
void rescaleDistribution(const double& k)	rescales distribution through the energy axis using fourmula $E' = mc^2 + k \cdot (E - mc^2)$, $F(E', \mu, \phi) = F(E, \mu, \phi)/k$. It may be useful when e.g. distribution of electrons is obtained by numerical code with increased electron mass
MassiveParticleMonoenergeticDirectedDistribution	class representing narrow beam of particles with close energies
MassiveParticleMonoenergeticDirectedDistribution(const double& mass, const double& Energy, const double& halfWidth, const double& concentration, const double& theta0, const double& phi0, const double& deltaTheta)	constructor, creates distribution with given particle mass, mean energy, half-width of uniform distribution around the mean energy, polar and azimuthal angles determining direction of mean velocity and half-width angle of velocity cone

MassiveParticleMovingDistribution	class transforming distribution from one frame to another
MassiveParticleMovingDistribution(MassiveParticleDistribution* distribution, const double& velocity)	constructor, transforms the given distribution from the frame with givem velocity along z-axis to the lab frame
CompoundMassiveParticleDistribution	class representing distribution as sum of other distributions
CompoundMassiveParticleDistribution(int N, MassiveParticleDistribution** distributions)	constructor, creates distribution which is sum of given distributions
CompoundMassiveParticleDistribution(MassiveParticleDistribution* dist1, MassiveParticleDistribution* dist2)	constructor, creates distribution which is sum of two given distributions
CompoundMassiveParticleDistribution(MassiveParticleDistribution* dist1, MassiveParticleDistribution* dist2, MassiveParticleDistribution* dist3)	constructor, creates distribution which is sum of three given distributions
CompoundWeightedMassiveParticleDistribution	class representing distribution as weighted sum of other distributions
CompoundWeightedMassiveParticleDistribution(int N, const double* weights, MassiveParticleDistribution** distributions)	constructor, creates distribution which is sum of given distributions with given weights
CompoundWeightedMassiveParticleDistribution(MassiveParticleDistribution* dist1, const double& w1, MassiveParticleDistribution* dist2, const double& w2)	constructor, creates distribution which is sum of two given distributions with given weights
CompoundWeightedMassiveParticleDistribution(MassiveParticleDistribution* dist1, const double& w1, MassiveParticleDistribution* dist2, const double& w2, MassiveParticleDistribution* dist3, const double& w3)	constructor, creates distribution which is sum of three given distributions with given weights

1.1.3 Reading distributions from file

Classes for tabulated distribution, such as MassiveParticleTabulatedIsotropicDistribution, have constructors allowing to read distributions from files. It should be text files with tables of data, and format of data can be different. For determining data format there is enumerable type DistributionInputType with five possible values:

- ENERGY_FE - input file contains full energy in CGS units and distribution function $F(E)$

- ENERGY_KIN_FE - input file contains kinetic energy in CGS units and distribution function $F(E_{kin})$
- GAMMA_FGAMMA - input file contains lorentz-factor and distribution function with respect to it $F(\gamma)$
- GAMMA_KIN_FGAMMA - input file contains reduced lorentz-factor $(\gamma - 1)$ and distribution function with respect to it $F(\gamma - 1)$
- MOMENTUM_FP - input file contains momentum in CGS units and distribution function with respect to it $F(p)$

Regardless of input file format, distribution function would be transformed to the units energy vs distribution $F(E)$. Example of reading distribution from file is given below

```
double electronConcentration = 1.0;
int N = 100;
MassiveParticleIsotropicDistribution* distribution = new
MassiveParticleTabulatedIsotropicDistribution(massElectron,
"energy.dat", "distribution.dat", N, electronConcentration,
DistributionInputType::ENERGY_FE);
```

Class MassiveParticleDistributionFactory is implemented for simplicity of reading distributions from files in complicated cases. It has several similar static methods allowing to read array of distribution from set of numerated files. It can be useful in cases when distribution function depends on some external parameter which varies inside the radiation source. Example of reading array of ten distributions of electrons from files, named "Fe0.dat" , "Fe1.dat" etc., consisting of two columns - loerntz-factor and distribution function, and adding power-law tail with index 3, starting from energy $10 m_e c^2$, calling one function is given below

```
double electronConcentration = 1.0;
int Nenergy = 100;
int Ndistribution = 100;
double powerLawEnergy = 100*me_c2;
double index = 3.0;
MassiveParticleIsotropicDistribution** distributions =
MassiveParticleDistributionFactory::
readTabulatedIsotropicDistributionsAddPowerLawTail(
massElectron, "./input/Fe", ".dat", Ndistribution,
DistributionInputType::GAMMA_FGAMMA, electronConcentration, Nenergy,
powerLawEnergy, index);
```

Also it is possible to create tabulated distributions not by reading them from files, but from arrays, which can be generated by user with any suitable method.

1.2 Radiation sources

To evaluate electromagnetic radiation with FAINA code user should create object, representing radiation source. It allows to take into account geomtry of the source, it's inhomogenuity and other features.

There are two types of radiation sources - sources without time dependency, represented with class RadiationSource, and cources depending on time, represented with class RadiationTimeDependentSource. This two classes are not related with each other through inheritance, but the object of class RadiationTimeDependentSource contains the object of class RadiationSource inside. Diagram of class hierarchy of radiation sources is shown in Figure 1.4.

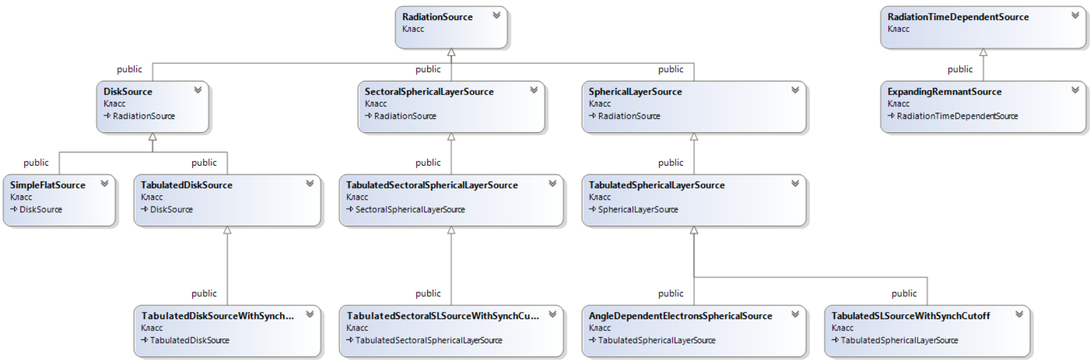

Figure 1.4: class hierarchy of radiation sources

1.2.1 Radiation sources without time dependency

Radiation sources without time dependency are represented with abstract class RadiationSourse and it's derived classes. All sources models uses cylinrical grid with z axis along line of sight. This allows easily integrate through z-axis taking into accound absorption processes. Difference of the real shape of the source from discrete shape of a grid is compensated with filling factor of every grid cell. It represent fraction of cell volume which is located inside the source. Model of the source with geometry of spherical layer is shown in Figure 1.5. Colour shows filling factors of each cell.

Radiation sources have following parameters, that can vary in different grid cells: concentration of emmitting particles, their distribution functions, magnetic field and it's orientation angles. Also distance to the observer is important parameter of the source.

Class RadiationSource has three abstract derived classes: DiskSource - for the sources with shape of a disk with axis along line of sight, SphericalLayerSource - for sources with shape of a spherical layer and SectoralSphericalLayerSource - for sources with shape of spherical layer restricted with some azimutal angle range, and also with minimum cylindrical radius limit. The last one is useful when real source is a prolonged object observed with high resolution, and some features of radiation from different regions of the source are studied.

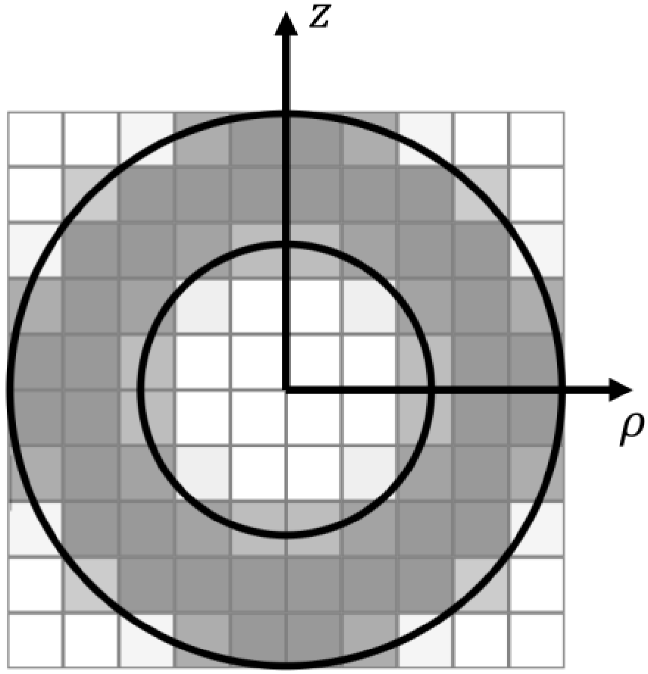

Figure 1.5: Model of the source with spherical layer geometry in the cylindrical grid. Colour shows fraction of cell volume which is located inside the source.

Sources with disk shape have three specific implementations: SimpleFlatSource - homogenous disk consisting of only one grid cell with given parameters, TabulatedDiskSource - inhomogenous disk in wich all parameters are tabulated on the spatial grid, and TabulatedDiskSourceWithSynchCutoff, which is inherited from previous one and allows to take into account synchrotron losses of emitting particles during their propagation inside the sourse. In this source model particles distribution is suppose to be generated on the upper face of disk, representing shock wave, and then particles are moving via convection inside the source. Evolution of the distribution function is described with equation:

$$f_l(E) = f\left(\frac{E}{1 - 4e^4B^2E\ l/9m^4c^7v}\right) \cdot \frac{1}{\left(1 - 4e^4B^2E\ l/9m^4c^7v\right)^2} \tag{1.1}$$

where $f(E)$ generated distribution function, E - energy of the particle, B - magnetic field, l - distance from given point to the upper face, v - velocity of convection movement, e - absolute value of particle charge, m - particle mass, c - speed of light.

Sources with shape of spherical layers have following implementations: TabulatedSphericalSource, in wich all parameters are tabulated on the spatial grid and two classes derived from it TabulatedSLSourceWithSynchCutoff and AngleDependentElectronsSphericalSource. First of them allows to take into account synchrotron losse, like it was done in TabulatedDiskSourceWithSynchCutoff, but in this case distribution is generated on the outer spherical surface, and the second is useful for important case in astrophysics when distribution function of emitting particles is dependent on inclination angle of magnetic field to the direction of shock propagation [2, 3, 4, 5]. In AngleDependentElectronsSphericalSource such parameters as number density, magnetic field and it's orintation angles are tabulated on the spatial grid, while distribution function is tabulated with respect of inclination angle of magnetic field to the shock, which is considered spherically symmetrical and propagates along radius.

Sources with shape of sector of spherical layer has following implementations: TabulatedSectoralSphericalLayerSource, in wich all parameters are tabulated on the spatial grid and derived from it TabulatedSectoralSLSourceWithSynchCutoff, taking into account synchrotron energy losses like it was done in TabulatedSLSourceWithSynchCutoff.

Public methods of classes for radiation sources are listed in the Table 1.5.

Table 1.5: Public methods of classes for time independent radiation sources

RadiationSource	abstract class for radiation sources
virtual double getMaxRho()	virtual method, returns upper boundary of the source at the cylindrical radius
virtual double getMinRho()	virtual method, returns lower boundary of the source at the cylindrical radius
virtual double getMinZ()	virtual method, returns lower boundary of the source at the z axis
virtual double getMaxZ()	virtual method, returns upper boundary of the source at the z axis
virtual double getMaxB()	virtual method, returns maximal magnetic field in the source
virtual double getAverageSigma()	virtual method, returns average magnetization $\sigma = \frac{B^2}{4\pi n m_p c^2}$
virtual double getAverageConcentration()	virtual method, returns average number density
virtual double getRho(int irho)	virtual method, returns cylindrical radius of the cell with number irho through the radial axis
virtual double getZ(int iz)	virtual method, returns z coordinate of the cell with number iz through the z axis
virtual double getPhi(int iphi)	virtual method, returns azimutal angle of the cell with iphi number through the azimutal coordinate
virtual int getRhoIndex(const double& rho)	virtual method, returns radial number of cell containing given radial coordinate
virtual bool isSource(int irho, int iphi)	virtual method, return boolean value defining are cells with given radial and azimutal numbers part of the source or not. It is useful for modeling of sources with complex geometry

int getNrho()	returns number of grid points through the radial axis
int getNz()	returns number of grid points through the z axis
int getNphi()	returns number of grid points twith respect to the azimutal angle
double getDistance()	returns distance to the source
virtual getArea(int irho)	virtual method, returns average area of cross-section of part of the cell, filled with the source matter
virtual getLength(int irho, int iz, int iphi)	virtual method, returns average thickness of the part of the source, filled with the source matter
getVolume(int irho, int iz, int iphi)	returns volume of the part of the cell, filled with source matter. This method is consistent with getArea and getLength, and returns their product
virtual getB(int irho, int iz, int iphi)	virtual method, returns magnetic filled in given cell
virtual getConcentration(int irho, int iz, int iphi)	virtual method, returns number density in given cell
virtual getSinTheta(int irho, int iz, int iphi)	virtual method, returns sinus of angle between magnetic field and line of sight in given cell
virtual void getVelocity(int irho, int iz, int iphi, double& velocity, double& theta, double& phi)	virtual method, returns velocity in given cell, and polar and azimutal angles for it's direction
virtual getTotalVolume()	virtual method, returns total volume of the source
virtual resetParameters(const double* parameters, const double* normalizationUnits)	virtual method, reseting parameters of the source. Lists of parameters are different for different types of sources. Method takes for input array of parameters in normalized units, and array of normalization conctants. This method for example is used for fitting modelled radiation to the observational data and optimization such parameters as magnetic field, number density and others. Also it is used to model time evolution of the source

virtual getParticleDistribution(int irho, int iz, int iphi)	virtual method, returns emitting particles distribution in given cell
DiskSource	abstract class for sources with shape of a disk
SimpleFlatSource	class for homogenous disk sources
SimpleFlatSource(MassiveParticleDistribution* electronDistribution, const double& B, const double& theta, const double& rho, const double& z, const double& distance, const double& velocity = 0)	constructor, creates rasiation source with given distribution of emitting particles, magnetic field, angle of it's inclination angle to the line of sight, number density, disk radius, thickness, distance to the observer and velocity of the source matter along z axis
TabulatedDiskSource	class for disk sources with tabulated parameters
TabulatedDiskSource(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, double*** B, double*** sinTheta, double*** concentration, const double& rho, const double& z, const double& distance, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and tabulated magnetic field, it's inclination angle to the line of sight, number density, disk radius, thickness, distance to the observer and velocity of the source matter along z axis
TabulatedDiskSource(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, const double& B, const double& sinTheta, const double& concentration , const double& rho, const double& z, const double& distance, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and uniform magnetic field, it's inclination angle to the line of sight, number density, disk radius, thickness, distance to the observer and velocity of the source matter along z axis
TabulatedDiskSourceWithSynchCutoff	class for disk sources with tabulated parameters and taking into account synchrotron energy losses
TabulatedDiskSourceWithSynchCutoff(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, double*** B, double*** theta, double*** concentration, const double& rho, const double& z, const double& distance, const double& downstreamVelocity, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and tabulated magnetic field, it's inclination angle to the line of sight, number density, disk radius, thickness, distance to the observer convection velocity of emitting particles and velocity of the source matter along z axis

TabulatedDiskSourceWithSynchCutoff(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, const double& B, const double& concentration, const double& theta, const double& rho, const double& z, const double& distance, const double& downstreamVelocity, const double& velocity = 0)	given distribution of emitting particles and uniform magnetic field, it's inclination angle to the line of sight, number density, disk radius, thickness, distance to the observer, convection velocity of emitting particles and velocity of the source matter along z axis
SphericalLayerSource	abstract class for sources with the shape of spherical layer
double getInnerRho()	returns inner radius of spherrical layer
TabulatedSphericalLayerSource	class for sources with the shape of spherical layer with tabulated parameters
TabulatedSphericalLayerSource(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, double*** B, double*** sinTheta, double*** concentration, const double& rho, const double& rhoin, const double& distance, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and tabulated magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, distance to the observer and velocity of the source matter along radius
TabulatedSphericalLayerSource(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, const double& B, const double& concentration, const double& sinTheta, const double& rho, const double& rhoin, const double& distance, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and uniform magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, distance to the observer and velocity of the source matter along radius
AngleDependentElectronsSphericalSource	class for sources with the shape of spherical layer, and dependency of distribution function on inclination of magnetic field to the shock propagation velocity
AngleDependentElectronsSphericalSource(int Nrho, int Nz, int Nphi, int Ntheta, MassiveParticleDistribution** electronDistributions, double*** B, double*** sinTheta, double*** phi, double*** concentration, const double& rho, const double& rhoin, const double& distance, const double& velocity = 0)	constructor, creates radiation source with given arrays of distributions of emitting particles for different inclination angles and tabulated magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, distance to the observer and velocity of the source matter along radius

AngleDependentElectronsSphericalSource(int Nrho, int Nz, int Nphi, int Ntheta, MassiveParticleDistribution** electronDistributions, const double& B, const double& sinTheta, const double& phi, const double& concentration, const double& rho, const double& rhoin, const double& distance, const double& velocity = 0)	constructor, creates radiation source with given arrays of distributions of emitting particles for different inclination angles and uniform magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, distance to the observer and velocity of the source matter along radius
TabulatedSLSourceWithSynchCutoff	class for sources with the shape of spherical layer taking into acount synchrotron energy losses
TabulatedSLSourceWithSynchCutoff(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, double*** B, double*** theta, double*** concentration, const double& rho, const double& rhoin, const double& distance, const double& downstreamVelocity, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and tabulated magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, distance to the observer, convection velocity of emitting particles and velocity of the source matter along radius
TabulatedSLSourceWithSynchCutoff(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, const double& B, const double& concentration, const double& theta, const double& rho, const double& rhoin, const double& distance, const double& downstreamVelocity, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and uniform magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, distance to the observer, convection velocity of emitting particles and velocity of the source matter along radius
SectoralSphericalLayerSource	abstract class for sources with shape of sector of spherical layer
double getRhoin()	returns internal radius of spherical layer
TabulatedSectoralSphericalLayerSource	class for sources with the shape of spherical layer with tabulated parameters
TabulatedSectoralSphericalLayerSource(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, double*** B, double*** theta, double*** concentration, const double& rho, const double& rhoin, const double& minrho, const double& phi, const double& distance, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and tabulated magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, minimal cylindrical radius, azimutal width of the sector, distance to the observer and velocity of the source matter along radius

TabulatedSectoralSphericalLayerSource(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, const double& B, const double& concentration, const double& theta, const double& rho, const double& rhoin, const double& minrho, const double& phi, const double& distance, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and uniform magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, minimal cylindrical radius, azimutal width of the sector, distance to the observer and velocity of the source matter along radius
TabulatedSectoralSLSourceWithSynchCutoff	class for sources with the shape of sector of spherical layer taking into account synchrotron losses
TabulatedSectoralSLSourceWithSynchCutoff(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, double*** B, double*** theta, double*** concentration, const double& rho, const double& rhoin, const double& minrho, const double& phi, const double& distance, const double& downstreamVelocity, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and tabulated magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, minimal cylindrical radius, azimutal width of the sector, distance to the observer, convection velocity of emitting particles and velocity of the source matter along radius
TabulatedSectoralSLSourceWithSynchCutoff(int Nrho, int Nz, int Nphi, MassiveParticleDistribution* electronDistribution, const double& B, const double& concentration, const double& theta, const double& rho, const double& rhoin, const double& minrho, const double& phi, const double& distance, const double& downstreamVelocity, const double& velocity = 0)	constructor, creates radiation source with given distribution of emitting particles and uniform magnetic field, it's inclination angle to the line of sight, number density, external and internal radii of spherical layer, minimal cylindrical radius, azimutal width of the sector, distance to the observer, convection velocity of emitting particles and velocity of the source matter along radius

1.2.2 Radiation sources with time dependency

Radiation sources, taking into account time dependency, are represented with abstract class RadiationTimeDependentSource. This class is not derived from RadiationSource, but it contains object of this type as private field for evaluation of radiation at specific time moment. RadiationSource object with parameters corresponding to the given time moment can be obtained with virtual method getRadiationSource. This method uses resetParameters method of RadiationSource to model time evolution, so it is important to make sure that parameters of RadiationSource object are consistent with implementation of getRadiationSource method. In current version of FAINA there is only one implementation of RadiationTimeDependentSource - ExpandingRemnantSource which represents model of expanding spherical supernove remnant. In this model radius of the source is sopposed to change through time with powerlaw dependency $R \propto t^{p_t}$, and magnetic field and number density depend on time as $B \propto 1/R^{p_B}$ and $n \propto 1/R^{p_n}$.

Public methods of classes RadiationTimeDependentSource and ExpandingRemnantSource are listed in Table 1.6. User can create his own implementation of RadiationTimeDependentSource for more compliceted models.

Table 1.6: Public methods of radiation sources with time dependency

RadiationTimeDependentSource	abstract class for sources with time dependency
virtual resetParameters(const double* parameters, const double* normalizationUnits)	virtual method, reseting parameters of the source (not the same as for RadiationSource). Lists of parameters are different for different types of sources. Method takes for input array of parameters in normalized units, and array of normalization conctants. This method for example is used for fitting modelled radiation to the observational data and optimization such parameters as magnetic field, number density and others.
virtual getRadiationSource(double& time, const double* normalizationUnits)	virtual method, returns RadiationSOurce at given time moment. Also needs normaliztion units for source parameters
ExpandingRemnantSource	class representing expanding supernove remnant
ExpandingRemnantSource(const double& R0, const double& B0, const double& concentration0, const double& v, const double& widthFraction, RadiationSource* source, const double& t0, const double& radiusPower = 1.0, const double& Bpower = 1.0, const double& concentrationPower = 2.0)	constructor, creates expanding source, wight given at moment t0 radius, magnetic field, number density, velocity of expansion, fraction of radius filled with source matter, model of the source and power index of dependencies of radius, magnetic field and number density.

Chapter 2

Evaluation of radiation

In current version of the code following types of radiation are implemented: synchrotron radiation, inverse Compton scattering, gamma-ray emission due to pion decay in free-free proton interaction and also bremsstrahlung.

Abstract class RadiationEvaluator and it's inherited classes are used for evaluation of radiation. There are derved classes for every specific type of radiation and also class RadiationSumEvaluator which allows to sum several different types of radiation. Public methods of this two classes are listed in Table 2.1.

General approach to evaluation of radiation is following: create radiation source, using one of the classes, described in Section 1.2.1, or user-defined, then create object of radiation evaluator, which types are described below, and then call method evaluateFluxFromSource(const double& photonFinalEnergy, RadiationSource* source) of this object, which evaluates energy density of the energy flux from source in units $\mathrm{cm}^{-2}\mathrm{s}^{-1}$.

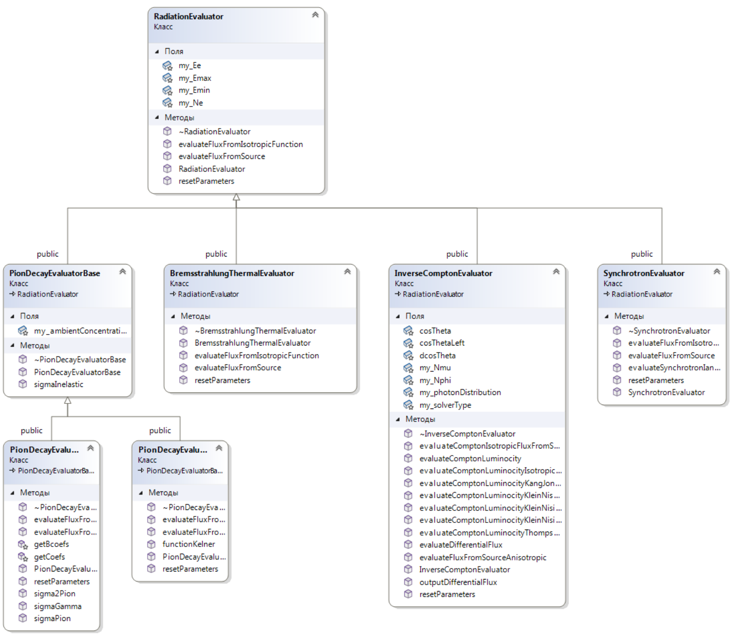

Figure 2.1: class hierarchy of radiation evaluators

Classes for every specific type of electromagnetic radiation are described below in this chapter. Class hierarchy of radiation evaluators is shown in Figure 2.1. Equations used for evaluation are discussed in Chapter 4.

Table 2.1: Public methods of RadiationEvaluator class

RadiationEvaluator	astract class for evaluation of radiation
virtual evaluateFluxFromSource(const double& photonFinalEnergy, RadiationSource* source)	virtual method, returns energy density of radiation energy flux in units $cm^{-2}s^{-1}$
virtual double evaluateFluxFromSourceAtPoint(const double& photonFinalEnergy, RadiationSource* source, int rhoi, int phi)	virtual method, returns energy density of radiatio energy flux from given grid cell on tangent plane
double evaluateTotalFluxInEnergyRange(const double& Ephmin, const double& Ephmax, int Nph, RadiationSource* source)	returns integrated energy flux in the given energy range, evaluted by Nph points distributied logarithmically, in units $ergcm^{-2}s^{-1}$
virtual resetParameters(const double* parameters, const double* normalizationUnits)	virtual method, reseting parameters of the radiation evaluator. Lists of parameters are different for different types of evaluators. Method takes for input array of parameters in normalized units, and array of normalization conctants. This method for example is used for fitting modelled radiation to the observational data and optimization.
writeFluxFromSourceToFile(const char* fileName, RadiationSource* source, const double& Ephmin, const double& Ephmax, const int Nph)	evaluates and writes into the file energy density of radiation energy flux in given energy range with Nph points distributed logarithmically. Writes two columns of data in units erg and $cm^{-2}s^{-1}$
writeImageFromSourceToFile(const char* fileName, RadiationSource* source, const double& Ephmin, const double& Ephmax, const int Nph)	evaluates and writes to file image - energy flux from every cell of the tangent plane in units $ergcm^{-2}s^{-1}$ integrated in given energy range with Nph points distributed logarithmically
writeImageFromSourceAtEToFile(const double& photonFinalEnergy, const char* fileName, RadiationSource* source)	evaluates and writes to file image - energy density of radiation energy flux from every cell of the tangent plane in units $cm^{-2}s^{-1}$
RadiationSumEvaluator	class for sum of several types of radiation
RadiationSumEvaluator(int Ne, const double& Emin, const double& Emax, RadiationEvaluator* evaluator1, RadiationEvaluator* evaluator2)	constructor, creates evaluator which sums results of two given evaluators, and takes into accaunt emmiting particles in given energy range
RadiationSumEvaluator(int Ne, const double& Emin, const double& Emax, int Nev, RadiationEvaluator** evaluators)	constructor, creates evaluator which sums results of given array of evaluators, and takes into accaunt emmiting particles in given energy range

2.1 Synchrotron radiation

Class SynchrotronEvaluator is implemented for evaluation of synchrotron radiation. It uses standard approximation of continious spectrum, described in [6, 7] and in section 4.3, - it is valid for frequencies of emitted photons much higher than gyrofrequency of emitting particles. Also it is possible to take into account synchrotron self-absorption. Cylindrical geomtry, shown in Figure 1.5 allows to integrate flux through the line of sight and take into account absorption inside the source. To create SynchrotronEvaluator object user should provide energy range of particles to be taken into account, numbers of integration points in it, and also two boolean parameters - for accounting self absorption and doppler shifting due to source matter velocity. Public methods of Synchrotron evaluator are listed in Table 2.2. Example of evaliation of synchrotron radiation is shown in section ??.

Table 2.2: Public methods of SynchrotronEvaluator

SynchrotronEvaluator	class for evaluation synchrotron radiation
SynchrotronEvaluator(int Ne, double Emin, double Emax, bool selfAbsorption = true, bool doppler = false)	constructor, creates evaluator with given energy range of particles taken into account and parameters corresponding to self-absorption and doppler effect
evaluateSynchrotronIandA(const double& photonFinalFrequency, const double& photonFinalTheta, const double& photonFinalPhi, const double& B, const double& sinhi, const double& concentration, MassiveParticleIsotropicDistribution* electronDistribution, double& I, double& A)	evaluates emissivity per unit volume and absorption coefficient for photon of given energy and direction in given magnetic field and number density and distribution of emitting particles

2.2 Inverse Compton scattering

Class InverseComptonEvaluator is implemented for evaluation of radiation produced in inverce compton scattering. Also it has one derived class InverseComptonEvaluatorWithSource. The difference between them is that in the first one distribution of seed photons is constant inside the source, and in the second one photons number density is change proportionaly inverse square of the distance to the source of seed photons.

There are four different algorithmes of evaluation IC radiation that can be used by InverseComptonEvaluator. They are listed by enum-type ComptonSolverType, having following values:

- ISOTROPIC_THOMSON - simple model of scattering in thomson regime with power-law distribution of electrons and thermal distribution of seed photons, as described in [6] ch 17, p. 466.

- ANISOTROPIC_KLEIN_NISHINA - model computing radiation directly by integrating Klein-Nishina cros-section as described in [8, 9] and in section 4.2. With this model is possible to evaluate radiation produced by anisotropic distributions of initial particles

- ISOTROPIC_KLEIN_NISHINA - model similar to the previous, it uses integration of Klein-Nishina cross-section, but isotropy of distributions of initial particles is assumed, and it allows to reduce number of integrations through the azimuthal angle

- ISOTROPIC_JONES - model, using analytical integration through the all angular variables, in case of isotropic distributions of initial particles. It is described in [10, 11] and in section??

To create object of InverseComptobEvaluator type user needs to provide energy range of particles taken into account and nimber of points to integrate through it, number of points through the polar and azimutal angle, distribution function of seed photons and algorithm of computation the radiation. Public methods of InverseComptonEvaluator and InverseComptonEvaluatorWithSource are listed in Table 2.3.

Table 2.3: Public methods of inverse compton scattering evaluators

InverseComptonEvaluator	class for evaluation radiation from inverse compton scattering
InverseComptonEvaluator(int Ne, int Nmu, int Nphi, double Emin, double Emax, PhotonDistribution* photonDistribution, ComptonSolverType solverType)	constructor, creates evaluator with given energy range of particles taken into account, numbers of integration points throught the energy and angular variables, distribution function of seed photons and method of computation the radiation
evaluateFluxFromSourceAnisotropic(const double& photonFinalEnergy, const double& photonFinalTheta, const double& photonFinalPhi, PhotonDistribution* photonDistribution, RadiationSource* source)	returns energy density of radiation energy flux created by given seed photons distribution and source containing scattering particles in given direction
evaluateTotalFluxInEnergyRangeAnisotropic(const double& Ephmin, const double& Ephmax, const double& photonFinalTheta, const double& photonFinalPhi, int Nph, PhotonDistribution* photonDistribution, RadiationSource* source, ComptonSolverType solverType)	returns total energy flux of radiation created by given seed photons distribution and source containing scattering particles in given direction integrated in given energy range through Nph point distributed logarithmically.

InverseComptonEvaluatorWithSource	class for evaluation radiation from inverce comton scattering takin into account dependency of photons number density on distance to the source of photons
InverseComptonEvaluatorWithSource(int Ne, int Nmu, int Nphi, double Emin, double Emax, double Ephmin, double Ephmax, PhotonDistribution* photonDistribution, ComptonSolverType solverType, const double& sourceR, const double& sourceZ, const double& sourcePhi)	constructor, creates evaluator with given energy range of particles taken into account, numbers of integration points throught the energy and angular variables, distribution function of seed photons with number density corresponding to the origin of coordinates, method of computation the radiation and coordinates of the source of seed photons

Example of the evaluation of radiation produced by Inverse Compton scattering is shown in the function evaluateComtonWithPowerLawDistribution() in the file examples.cpp. In this function X-ray radiation from Fast Blue Optical Transient CSS161010 is evaluated. Electrons distribution is assumed power-law as in paper [12] and seed photons are taken from mean galactic photon field [1].

At first let define main parameters of the source - it's size, distance to observer, electrons number density and magnetic field. Magnetic field doesn't matter for inverse compton scattering, so assume it equal to zero. Also we define numbers of grid points for integration through the energy and angular variables

```
double electronConcentration = 150;
double sinTheta = 1.0;
double rmax = 1.3E17;
double B = 0.0;
double distance = 150*1E6*parsec;

double Emin = me_c2;
double Emax = 1000 * me_c2;
int Ne = 200;
int Nmu = 20;
int Nphi = 4;
```

Then we create distribution of seed photons, using static method of class MultiPlankDistribution getGalacticField wich returns mean galactic photon distribution. And also we create electron power-law distribution with spectral index 3.5

```
PhotonIsotropicDistribution* photonDistribution =
    PhotonMultiPlankDistribution::getGalacticField();
MassiveParticlePowerLawDistribution* electrons = new
    MassiveParticlePowerLawDistribution(massElectron, 3.5,
```

```
Emin, electronConcentration);
```

Then we create radiation source as homogenous disk and radiation evaluator for inverse Compton scattering. Let use the most universal method of comutation inverse compton radiation - ANISOTROPIC_KLEIN_NISHINA

```
RadiationSource* source = new SimpleFlatSource(
    electrons, B, sinTheta, rmax, rmax, distance);

InverseComptonEvaluator* comptonEvaluator = new
    InverseComptonEvaluator(Ne, Nmu, Nphi, Emin, Emax,
    photonDistribution, ComptonSolverType::ANISOTROPIC_KLEIN_NISHINA)
```

If user don't want to use standard method to writing radiation in to the file, in case of one need result in some other units - electron-volts for energy and write energy density flux in units $EF(E)$ - erg cm^{-2}s^{-1}, user should write result to file manually. Let create grid for energy of radiated photons

```
int Nnu = 200;
double* E = new double[Nnu];
double* F = new double[Nnu];
double Ephmin = 0.01 * kBoltzman * 2.725;
double Ephmax = 2 * Emax;
double factor = pow(Ephmax / Ephmin, 1.0 / (Nnu - 1));
E[0] = Ephmin;
F[0] = 0;
for (int i = 1; i < Nnu; ++i) {
        E[i] = E[i - 1] * factor;
        F[i] = 0;
}
```

and then compute energy density fluxes for this energies

```
for (int i = 0; i < Nnu; ++i) {
        F[i] = comptonEvaluator->evaluateFluxFromSource(
            E[i], source);
}
```

and wrtie them to file transforming to the prefered units

```
FILE* output_ev_EFE = fopen("output.dat", "w");

for (int i = 0; i < Nnu; ++i) {
        fprintf(output_ev_EFE, "%g %g\n",
```

```
            E[i] / (1.6E-12), E[i] * F[i]);
    }

    fclose(output_ev_EFE);
```

Spectrum of radiation, obtained with this code is shown in Figure 2.2

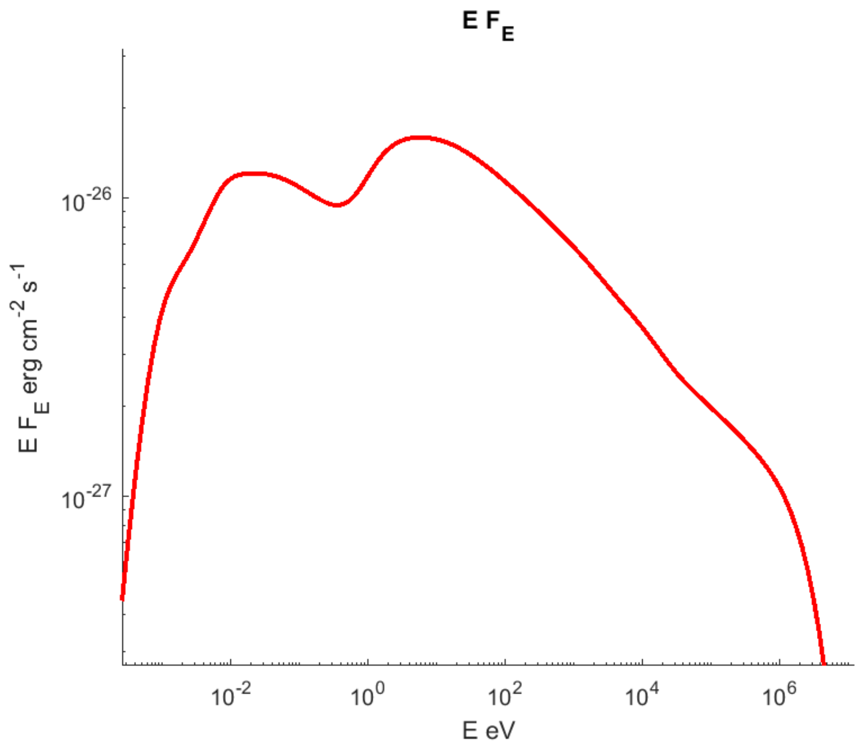

Figure 2.2: Energy density flux of inverse compton radiation

2.3 Pion decay

For evaluation of gamma-radiation, produced in proton-proton inelastic collision due to pion decay, abstract class PionDecayEvaluatorBase is implemented. Also there are two derived class for different methods of computation: PionDecayEvaluatorKelner, in wich cross-section of gamma-photon remission considered a fraction of cross-section of inelastic p-p interaction as it isdescribed in paper [13], and PionDecayEvaluator in which more accurate evaluation of cross-section at low energy is used, see [14]. In both models it is assumed, that high energy protons with isotropic distribution are scattered on cold protons of ambient medium, and that time of energy losses of fast protons due to p-p scattering is mush larger than time of particle confinment in the source, so each proton can be scattered at most one time.

To create object of pion decay evaluator user should provide energy range of particles to be taken into account, number of integration points in it and number density of ambient protons. Public methods of class PionEvaluatorBase and derived classes are listed in Table 2.4

Table 2.4: Public methods of class PionEvaluatorBase and inherited classes

PionDecayEvaluatorBase	abstract class for evaluation gamma radiation due to pion decay
sigmaInelastic(const double& energy)	returns cross-section of inelastic scattering moving proton on the resting in the lab frame. NOTE! method takes kinetic energy of moving proton
PionDecayEvaluatorKelner	class for evaluation gamma radiation assuming cross-section of emission is fraction of inelastic cross-section as it is described in [13]
PionDecayEvaluatorKelner(int Ne, double Emin, double Emax, const double& ambientConcentration)	constructor, creates evaluator with given range of protons energy taken into account, number of integration grid points in it and number density of ambient protons
PionDecayEvaluator	class for evaluation gamma radiation with method described in [14]
PionDecayEvaluator(int Ne, double Emin, double Emax, const double& ambientConcentration)	constructor, creates evaluator with given range of protons energy taken into account, number of integration grid points in it and number density of ambient protons
sigmaGamma(const double& photonEnergy, const double& protonEnergy)	returns cross-section of emission of gamma photon with given energy by scattering of proton with given kinetic energy. NOTE! method takes kinetic energy of moving proton

Example of evaluation of gamma radiation, produced by pion decay is shown in the function evaluatePionDecay() in the file examples.cpp. In this example gamma radiation from Cygnus Cocoon is evaluated. Protons are considered accelerated on the system of secondary shock as it is described in [15]. In this paper it is shown, that accelerated protons distribution is power-law with break at energy 2.2 TeV. Spectral index at low energies is 2.1 and 2.64 at high. Size of the emitting region is taken equal to the size of Sygnus Cocoon Superbubble - 55 pc.

As usual, let define general parameters of the source - number density, it's size and distance to it and magnetic field which can be set to zero in this example. Energu range of protons is from 0.01 GeV to 10 TeV, and the energy of break of the spectrum is 2.2 TeV

```
double protonConcentration = 150;
double rmax = 55 * parsec;
double B = 0;
double sinTheta = 1.0;

double distance = 1400 * parsec;
double Emin = massProton*speed_of_light2 + 0.01E9 * 1.6E-12;
```

```
double Emax = 1E13 * 1.6E-12;
double Etrans = 2.2E12 * 1.6E-12;
```

After that let create protons distribution and radiation source

```
MassiveParticleBrokenPowerLawDistribution* protons = new
        MassiveParticleBrokenPowerLawDistribution(
        massProton, 2.1, 2.64, Emin, Etrans, protonConcentration);
RadiationSource* source = new SimpleFlatSource(
        protons, B, sinTheta, rmax, rmax, distance);
```

Then one should create radiation evaluator. It is necessary to provide number density of ambient protons for it

```
double protonAmbientConcentration = 20;
PionDecayEvaluator* pionDecayEvaluator = new PionDecayEvaluator(
        200, Emin, Emax, protonAmbientConcentration);
```

Let create energy grid for radiated gamma photons, which will be used for manual output of radiation spectrum

```
int Nnu = 200;
double* E = new double[Nnu];
double* F = new double[Nnu];
double Ephmin = 0.01 * Emin;
double Ephmax = 1E16 * 1.6E-12;
double factor = pow(Ephmax / Ephmin, 1.0 / (Nnu - 1));
E[0] = Ephmin;
F[0] = 0;
for (int i = 1; i < Nnu; ++i) {
        E[i] = E[i - 1] * factor;
    F[i] = 0;
}
```

and then we can evaluate radiation energy density flux and write it into the file in preffred units

```
for (int i = 0; i < Nnu; ++i) {
        F[i] = pionDecayEvaluator->evaluateFluxFromSource(E[i], sourc
}
FILE* output_ev_dNdE = fopen("outputPionE.dat", "w");
for (int i = 0; i < Nnu; ++i) {
        double nu = E[i] / hplank;
        fprintf(output_ev_dNdE, "%g %g\n", E[i] / (1.6E-12), F[i] / E
}
fclose(output_ev_dNdE);
```

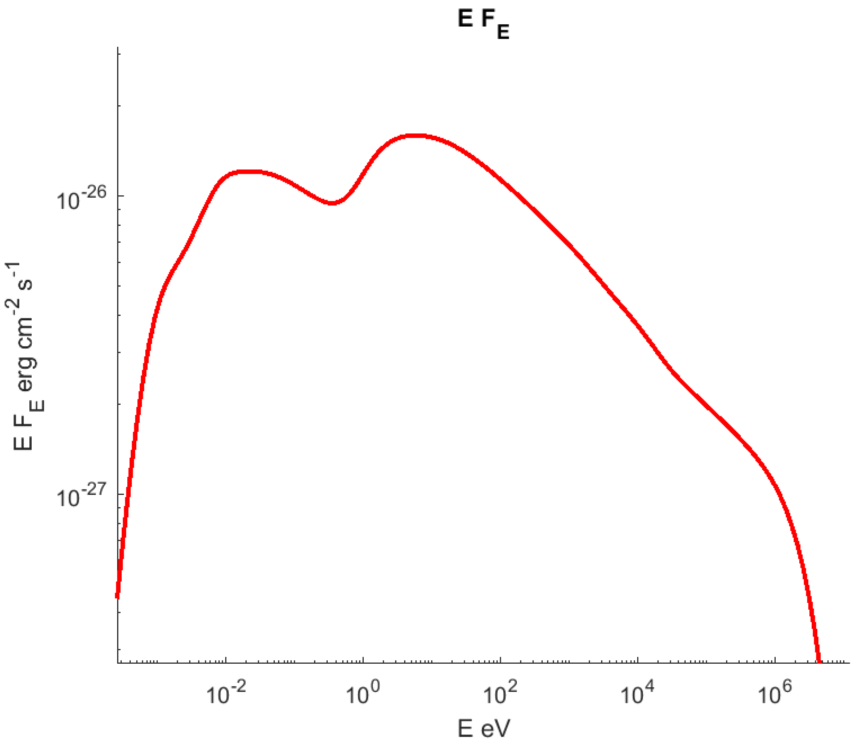

Figure 2.3: Modeled energy flux energy density from Cygnus Cocoon and observational data

Spectrum of radiation from Cygnus Cocoon, obtained with this code, and observational data by Fermi LAT, ARGO and HAWC [16, 17, 18] are shown in Figure 2.3

2.4 Bremsstrahlung

In current version of the code only simple case of thermal particle distribution is implemented with class BremsstrahlungThermalEvaluator. In this model thermal plasma with the same electron and positrons temperatures is assumed. Gaunt-factors for radiation are used as in [19]. Example of evaluation of bremsstrahlung is shown in function evaluateBremsstrahlung in the file examples.cpp.

Chapter 3

Parameters optimization

Code FAINA allows not only to evaluate radiation from the source, but also to fit model data to observations and obtain parameters of the source. There are various types optimization methods and models of loss functions, implemented in the code.

3.1 Evaluators of the loss function

The first thing one should determine to start optimization process is a loss function. In FAINA code abstract class LossEvalator and derived classes are used for it. All implemented classes use quadratic loss functions, taking into account observational errors: $L = \sum \frac{(F_i - F_{obs,i})^2}{\sigma_i^2}$, where F_i - is some modeled function of radiation (e.g. energy spectral density) evaluated at point corresponding to some observation, $F_{obs,i}$ - observed value of this function, σ_i - it's uncertancy. There are following classes of loss function evaluator implemented in the cod: SpectrumLossEvaluator - for fitting energy flux spectral density in given time moment, TimeDependentSpectrumLossEvaluator - for fitting energy flux spectal density in different time moments and RadialProfileLossEvaluator - for fitting luminocity of the prolonged source in different points depending on radius in tangent plane. Public methods of this classes are listed in Table 3.1.

Table 3.1: Public methods of loss function evaluators

LossEvaluator	abstract class for evaluator of loss function
virtual double evaluate(const double* vector, const double* maxParameters, RadiationEvaluator* evaluator)	virtual method, returns value of loss function with given parameters. Also takes vector of normalization units for parameters and radiation evaluator
SpectrumLossEvaluator	class for fitting energy flux energy density. Loss function is $L = \sum \frac{(F(E_i) - F_{obs,i})^2}{\sigma_i^2}$, where $F(E_i)$ - modeled energy flux energy density evaluated at energy E_i, $F_{obs,i}$ - corresponding observed value, σ_i - it's uncertancy.

SpectrumLossEvaluator(double* energy, double* observedFlux, double* observedError, int Ne, RadiationSource* radiatiornSource)	constructor, creates loss evaluator, with given values of energies of observational points, observed fluxes and it's uncertancies, number of points and radiation source
TimeDependentSpectrumLossEvaluator	class fitting energy flux energy density at different time moments. Loss function is $L = \sum \frac{(F(E_{ij},t_j)-F_{obs,i,j})^2}{\sigma_{ij}^2}$, where $F(E_{ij},t_j)$ - modeled energy flux energy density evaluated at energy E_{ij} at time moment t_j, $F_{obs,i,j}$ - corresponding observed value, σ_{ij} - it's uncertancies. NOTE that number of energy grid point may be different at different time moments
TimeDependentSpectrumLossEvaluator(double** energy, double** observedFlux, double** observedError, int* Ne, double* times, int Ntimes, RadiationTimeDependentSource* radiationSource)	constructor, creates loss evaluator with given values of energies of observational points, observed fluxes and it's uncertancies, numbers of energy grid points, time moments, number of time grid points and radiation source
RadialProfileLossEvaluator	class for fitting luminocity of different points of the source, depending of radius in tangent plane. Loss function is $L = \sum \frac{(F(R_i)-F_{obs,i})^2}{\sigma_i^2}$, where $F(R_i)$ - energy flux surface density, at given radius R_i, $F_{obs,i}$ - corresponding observed value, σ_i - it's uncertancy
RadialProfileLossEvaluator(double energy, double* observedFlux, double* observedError, double* rhoPoints, int Nrho, RadiationSource* radiaionSource)	constructor, creates loss evaluator, wuth given value of energy to evaluate flux density, observed value of luminocity surface density, observed value and it's uncertancies, radius grid points, number of grid points and radiation source

3.2 Optimizers of loss function

For fitting modeled radiation from the source to observational data, abstract class RadiationOptimizer is implemented. It has virtual function optimize(double* vector, bool* optPar) which performs minimization of loss function. This function takes on input array of parameters of the source vector, and boolean array optPar showing for each parameter to optimize it or consider it fixed. Array of parameters must be consistent with function of used radiation source resetParameters, which was described in section 1.2.1, because this function will be used during optimization and take the same array of parameters as input.

There are three implemented classes, inherited from RadiationOptimizer: GridEnumRadiationOptimizer, which finds minimum of loss function on the fixed logarithmic grid in the parameters space, GradientDescentRadiationOptimizer, which uses gradient

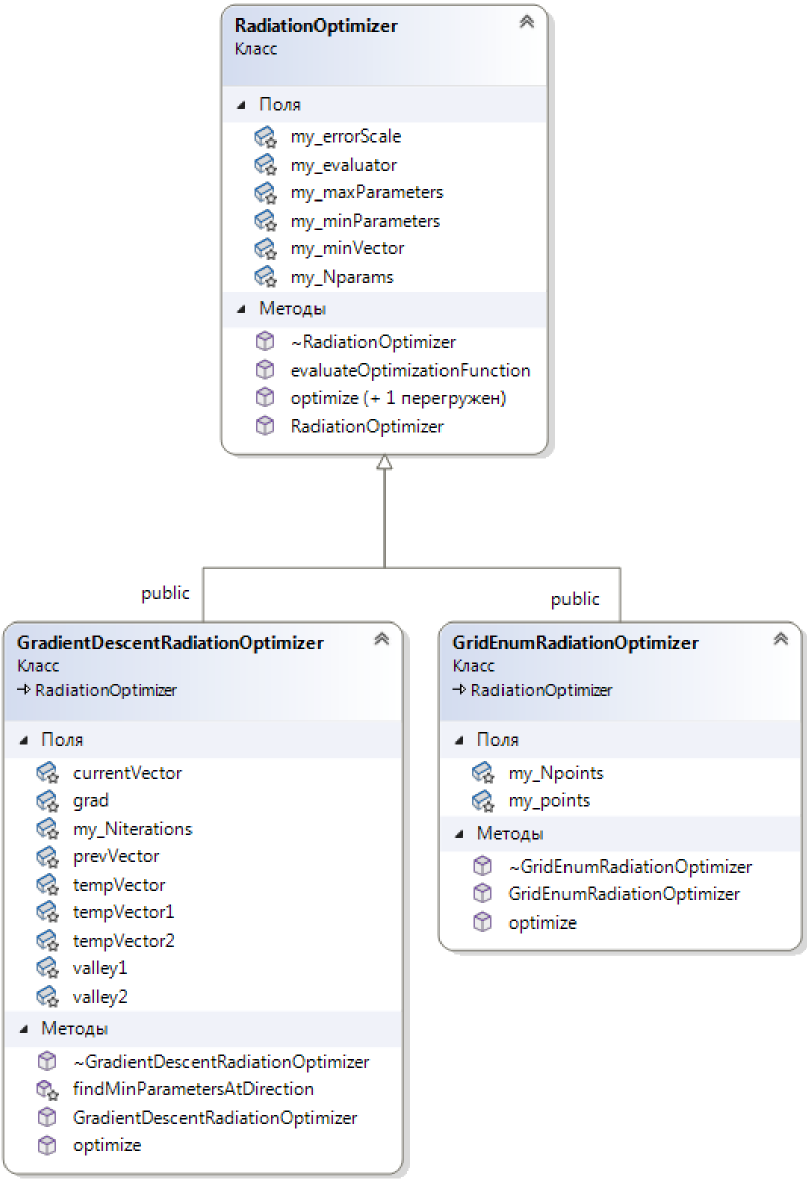

Figure 3.1: class hierarchy of radiation optimizers

descent method and CombinedRadiationOptimizer, which sequentially applies this method, using result of seqrch on the grid as starting point for gradient descent. Hierarchy scheme of optimizers classes is shown in Figure 3.1, and public methods are listed in Table 3.2. Implemented optimization methods can be applied to any types of sources, electro-magnetic radiation and loss function evaluators which were described above.

Table 3.2: Public methods of radiation optimizer classes

RadiationOptimizer	abstract class for fitting modeled radiation to observations and optimization of source parameters

double evaluateOptimizationFunction(const double* vector)	returns value of loss function with given parameters
void optimize(double* vector, bool* optPar)	performs optimization, takes array of peremeters, where final values would be writen, and array of boolean values showing to optimize corresponding parameter or consider it fixed
void outputProfileDiagrams(const double* vector, int Npoints)	evaluates and writes into files 2d profiles of loss function in parameters space which contain starting point, defined by array vector, for all possible pairs of parameters
void outputOptimizedProfileDiagram(const double* vector, bool* optPar, int Npoints, int Nparam1, int Nparam2)	evaluates and writes into file 2d profile of loss function in plane in the parameter space, defined by parameter numbers Nparam1 and Nparam2, containing starting point in it, with given nuber of grid points for this parameters, while other parameters are optimized, if it is allowed by boolean array optPar
GridEnumRadiationOptimizer	class for optimization by search of minimum value on the fixed logarithmic grid in parameters space
GridEnumRadiationOptimizer(RadiationEvaluator* evaluator, const double* minParameters, const double* maxParameters, int Nparams, const int* Npoints, LossEvaluator* lossEvaluator)	constructor, creates optimizer with given radiation evaluator, domain of parameters defined by minimum and maximum values, number of parameters, numbers of grid points for each parameters and loss function evaluator
GradientDescentRadiationOptimizer	class for optimization with gradient descent method
GradientDescentRadiationOptimizer(RadiationEvaluator* evaluator, const double* minParameters, const double* maxParameters, int Nparams, int Niterations, LossEvaluator* lossEvaluator)	constructor, creates optimizer with given radiation evaluator, domain of parameters defined by minimum and maximum values, number of parameters, number of iterations of gradient descent and loss function evaluator
CombinedRadiationOptimizer	class for optimization with sequentially use of search of minimum on the fixed grid and gradient descent method
CombinedRadiationOptimizer(RadiationEvaluator* evaluator, const double* minParameters, const double* maxParameters, int Nparams, int Niterations, const int* Npoints, LossEvaluator* lossEvaluator)	constructor, creates optimizer with given radiation evaluator, domain of parameters defined by minimum and maximum values, number of parameters, number of iterations for gradient descent, number of pointes through each axis for gread search and loss function evaluator

Example of optimizing the source parameters with fitting radiation to observationsl data is shown in the function fitCSS161010withPowerLawDistribition in file examples.cpp. Following the work [12] let evaluate synchrotron radiation from the source taking into account synchrotron self-absorption, considering power-law distribution of enitting electrons with index 3.6. But we will not fix such parameters as efractios of energy in magnetic field and accelerated electrons, instead we consider magnetic field and electrons concentration independent parameters, wich optimal values will be found with fitting.

Let optimize parameters of the source of Fast Blue Optical Transient CSS161010 at 98 day after explosion. We initialize source parameters using values from [12], they will be used as starting point of optimization.

```
double electronConcentration = 25;
double B = 0.6;
double R = 1.4E17;
double fraction = 0.5;
const double distance = 150 * 1E6 * parsec;
```

Then we create power-law distribution of emitting electrons with index 3.6, radiation source as homogenous disk and synchrotron radiation evaluator

```
double Emin = me_c2;
double Emax = 10000 * me_c2;
double index = 3.6;

SynchrotronEvaluator* synchrotronEvaluator = new
    SynchrotronEvaluator(200, Emin, Emax);

MassiveParticlePowerLawDistribution* electrons =
    new MassiveParticlePowerLawDistribution(
    massElectron, index, Emin, electronConcentration);

SimpleFlatSource* source = new
    SimpleFlatSource(electrons, B, pi/2, R, fraction * R, distance);
```

Now we define vector of parameters to be optimized - radius of the source, magnetic field, electron's number density and width fraction of the source. This parameters correspond to the resetParameters function of class SimpleFlatSource. Also one should define search domain with minimum and maximum value of each parameter. Maximum values are also used as normaliztion units.

```
const int Nparams = 4;
double minParameters[Nparams] = { 1E17, 0.01, 0.5, 0.1 };
double maxParameters[Nparams] = { 2E17, 10, 200, 1.0 };
double vector[Nparams] = { R, B, electronConcentration, fraction};
for (int i = 0; i < Nparams; ++i) {
        vector[i] = vector[i] / maxParameters[i];
}
```

Also we create arrays of observational data, which should be fitted. Note, the frequency should be transformed to energy, and flux spectral density to the flux energy density (to the units cm^{-2}s^{-1}).

```
const int Nenergy1 = 4;
double energy1[Nenergy1] = { 1.5E9*hplank, 3.0E9 * hplank,
    6.1E9 * hplank, 9.8E9 * hplank };
double observedFlux[Nenergy1] = { 1.5/(hplank*1E26),
    4.3/(hplank*1E26), 6.1/(hplank*1E26), 4.2 /(hplank*1E26)};
double observedError[Nenergy1] = { 0.1 / (hplank * 1E26),
    0.2/(hplank*1E26), 0.3/(hplank*1E26), 0.2/(hplank*1E26)};
```

Then we create evaluator of loss function and combined optimizer. We define number of grid points to search and number of gradient descent iterations. Also we create array of boolean showing that all parameters should be optimized.

```
bool optPar[Nparams] = { true, true, true, true };
int Niterations = 20;
int Npoints[Nparams] = { 10,10,10,10 };

LossEvaluator* lossEvaluator = new SpectrumLossEvaluator(energy1, observe
RadiationOptimizer* optimizer = new CombinedRadiationOptimizer(
    synchrotronEvaluator,minParameters,maxParameters,Nparams, Niterations
```

Let call function optimize and reset source parameters to obtained optimal values.

```
optimizer->optimize(vector, optPar, energy1, observedFlux,
    observedError, Nenergy1, source);
source->resetParameters(vector, maxParameters);
```

Jbtained optimal values of source parameters are: disk radius $R = 1.8 \times 10^{1}7$ см, magnetic field $B = 1.6$ Гс, electron's number density $n = 2.3$ см$^{-3}$, width fraction $fraction = 0.54$. Modeled spectrum of synchrotron radiation of source with this parameters and observational data are shown in Figure 3.2.

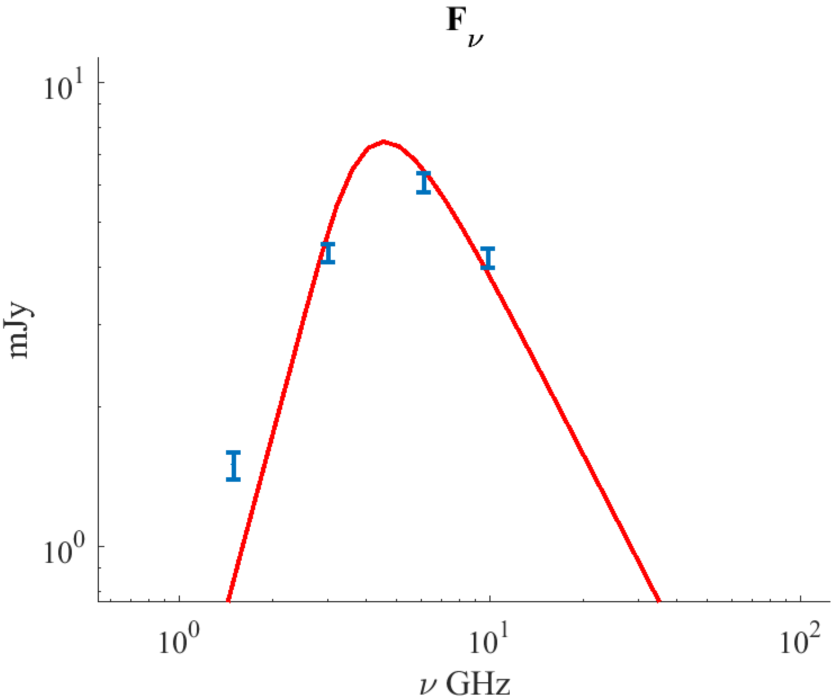

Figure 3.2: Modeled spectrum of synchrotron radiation and observational data for CSS161010 at 98 day after explosion

Chapter 4

Physics behind numerical methods

4.1 Distribution function transform

Distribution function of particles in phase space is presented in code in spherical coordinates $n(\epsilon, \mu, \phi)$. Let consider transform to the frame moving along z-axis with lorentz-factor $\gamma = 1/\sqrt{1-\beta^2}$. Number of particles in the corresponding phase volumes N is invariant.

$$N = n(\epsilon, \mu, \phi) d\epsilon d\mu d\phi dV = n'(\epsilon', \mu', \phi') d\epsilon' d\mu' d\phi' dV' \tag{4.1}$$

So, to obtain n' we need to evaluate determinant of Jacobi transformation matrix. Note, that azimuthal angle phi does not changes in transformation to the moving frame $phi' = phi$, and energy and polar angle does not depend on space volume, so in general Jacobi matrix has following non-zero terms

$$J = \begin{pmatrix} \frac{d\epsilon'}{d\epsilon} & \frac{d\epsilon'}{d\mu} & 0 & 0 \\ \frac{d\mu'}{d\epsilon} & \frac{d\mu'}{d\mu} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{dV'}{d\epsilon} & \frac{dV'}{d\mu} & 0 & \frac{dV'}{dV} \end{pmatrix} \tag{4.2}$$

Note, that determinant of this matrix can be decomposed by the last column and we get

$$|J| = \frac{dV'}{dV} \left(\frac{d\epsilon'}{d\epsilon} \frac{d\mu'}{d\mu} - \frac{d\epsilon'}{d\mu} \frac{d\mu'}{d\epsilon} \right) \tag{4.3}$$

Let start with transforming space volume V. First approach, following Landau-Lifshitz 2, paragraph 10 [20], is to transform volume to the rest-frame of moving beam of particle with given momentum, and then derive that $dV'/dV = \epsilon/\epsilon'$. It is correct result, but proof is not valid for massles particles, which have not physical rest frame.

So we evaluate transformation of volume, containing chosen particles, directly from Lorentz transformations. Let assume flux of particles, aligned with z axis, with uniform interval L between them, moving with same velocity v with angle θ to the z axis, and $\mu = cos(\theta)$, as it is shown in Figure 4.1.

So in the lab frame, at the moment t i-th particle is placed at $z_i = i \cdot L + \mu v t$. Let evaluate coordinates of particles in the moving frame.

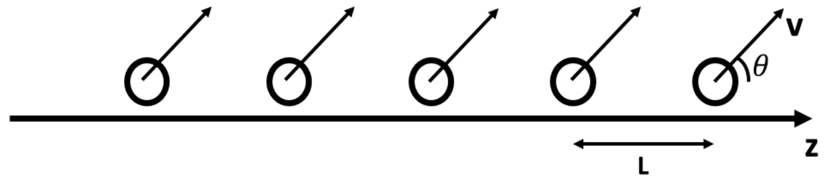

Figure 4.1: Beam of evenly distributed particles

$$\begin{pmatrix} ct' \\ z'_i \end{pmatrix} = \begin{pmatrix} \gamma & -\beta\gamma \\ -\beta\gamma & \gamma \end{pmatrix} \times \begin{pmatrix} ct \\ z_i \end{pmatrix} \tag{4.4}$$

From this we can obtain values $z'_i = \gamma z_i + (\gamma\mu v - c\beta\gamma)t$, but this values are measured at the different time moments in the moving frame, if t is the same for all particles in lab frame. To evaluate volume or number density we should evaluate them at the same moment t'. So let express t in terms of z_i and t', and put into the equation for z'_i.

$$t = \frac{t' + \gamma\beta z_i/c}{\gamma - \beta\mu v/c} \tag{4.5}$$

and

$$z'_i = \gamma z_i + (\gamma\mu v - c\beta\gamma)\frac{t' + \gamma\beta z_i/c}{\gamma - \beta\mu v/c} = z_i \frac{1}{\gamma(1 - \beta\mu v/c)} + t' \frac{\mu v/c - \beta}{1 - \beta\mu v/c} \tag{4.6}$$

Second term, containing t' gives us standard relativistic velocity-addition formula. And the first one gives a desired expression to the compression of the distance between particles $L' = z'_{i+1} - z'_i = L/(\gamma(1 - \beta\mu v/c))$. The distances in the transversal directions does not comress with Lorentz transformations, so volume also transforms as

$$V' = V/(\gamma(1 - \beta\mu v/c)) \tag{4.7}$$

This result is the same, as given by [20].

Next, we need to find expressions for ϵ' and μ', but it is better to deal with them in to separate cases - for massless and massive particles.

4.1.1 Photons

For massless photons, we consider transformation of energy-momentum vector, taking into account that z component of momentum is $p_z = \mu\epsilon/c$, and transversal components stay constant.

$$\begin{pmatrix} \epsilon' \\ \mu'\epsilon' \end{pmatrix} = \begin{pmatrix} \gamma & -\beta\gamma \\ -\beta\gamma & \gamma \end{pmatrix} \times \begin{pmatrix} \epsilon \\ \mu\epsilon \end{pmatrix} \tag{4.8}$$

From the first line we get equation for Doppler shift of photon's energy

$$\epsilon' = \gamma(1-\mu\beta)\epsilon \tag{4.9}$$

Derivatives of ϵ' with respect to ϵ and μ are

$$\frac{d\epsilon'}{d\epsilon} = \gamma(1-\mu\beta) \tag{4.10}$$

$$\frac{d\epsilon'}{d\mu} = -\gamma\beta\epsilon \tag{4.11}$$

From the second line of 4.8 we get $\mu'\epsilon' = -\beta\gamma\epsilon + \gamma\mu\epsilon$. Then we plug in expression for ϵ' from 4.9 and obtain equation for aberration of light

$$\mu' = \frac{\mu-\beta}{1-\mu\beta} \tag{4.12}$$

Angle of photon's velocity to the z-axis does not depend on their energy. And derivative of μ' with respect to μ is

$$\frac{d\mu'}{d\mu} = \frac{d\mu'}{d\mu} = \frac{d}{d\mu}\frac{1}{\beta}\frac{\beta\mu-1+1-\beta^2}{1-\mu\beta} = \frac{d}{d\mu}\frac{1}{\beta}\frac{1-\beta^2}{1-\mu\beta} = \frac{1-\beta^2}{(1-\mu\beta)^2} = \frac{1}{\gamma^2(1-\mu\beta)^2} \tag{4.13}$$

And Jacobi matrix of coordinate transformation in case of photons is

$$J = \begin{pmatrix} \frac{d\epsilon'}{d\epsilon} & \frac{d\epsilon'}{d\mu} & 0 & 0 \\ 0 & \frac{d\mu'}{d\mu} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{dV'}{d\mu} & 0 & \frac{dV'}{dV} \end{pmatrix} \tag{4.14}$$

Determinant of this matrix, fortunately, equals to the multiple of diagonal terms

$$\frac{D(\epsilon',\mu',\phi',V')}{D(\epsilon,\mu,\phi,V)} = \frac{d\epsilon'}{d\epsilon}\frac{d\mu'}{d\mu}\frac{dV'}{dV} = \gamma(1-\mu\beta)\frac{1}{\gamma^2(1-\mu\beta)^2}\frac{1}{\gamma(1-\mu\beta)} = \frac{1}{\gamma^2(1-\mu\beta)^2} \tag{4.15}$$

And finally, photons distribution function in spherical coordinates transforms as

$$n'_{ph}(\epsilon',\mu',\phi') = \frac{n_{ph}(\epsilon,\mu,\phi)}{\frac{D(\epsilon',\mu',\phi',V')}{D(\epsilon,\mu,\phi,V)}} = \gamma^2(1-\mu\beta)^2 n_{ph}(\epsilon,\mu,\phi) \tag{4.16}$$

4.1.2 Massive particles

In the case of massive particles, expression for ϵ' and μ' are more complicated. Now $p_z = \mu\sqrt{\epsilon^2 - m^2c^4}/c$, where m is particle mass, and Lorentz transformation of energy-momentum vector is expressed as

$$\begin{pmatrix} \epsilon' \\ \mu'\sqrt{\epsilon'^2 - m^2c^4} \end{pmatrix} = \begin{pmatrix} \gamma & -\beta\gamma \\ -\beta\gamma & \gamma \end{pmatrix} \times \begin{pmatrix} \epsilon \\ \mu\sqrt{\epsilon^2 - m^2c^4} \end{pmatrix} \tag{4.17}$$

So expressions for ϵ' and μ' are

$$\epsilon' = \gamma\epsilon - \beta\gamma\mu\sqrt{\epsilon^2 - m^2c^4} \tag{4.18}$$

$$\mu' = \frac{-\beta\gamma\epsilon + \gamma\mu\sqrt{\epsilon^2 - m^2c^4}}{\sqrt{\epsilon^2 - m^2c^4}} \tag{4.19}$$

And expression for volume transformation 4.7 in terms of ϵ and μ is

$$V' = \frac{V}{\gamma(1 - \mu\beta\sqrt{\epsilon^2 - m^2c^4}/\epsilon)} \tag{4.20}$$

Expressions for partial derivatives of ϵ', μ', V', and especially for Jacobian are really terrible, so here we present only final result for distribution function in units $c = 1$

$$\frac{n'_m(\epsilon',\mu',\phi')}{n_m(\epsilon,\mu,\phi)} = \frac{\gamma(\epsilon - \mu\sqrt{\epsilon^2 - m^2}\beta)(\gamma^2\epsilon^2 - m^2 + \mu^2((\epsilon^2 - m^2)(\gamma^2 - 1)) - 2\mu\epsilon\gamma^2\beta\sqrt{\epsilon^2 - m^2})^{3/2}}{\epsilon(((\gamma^2 - 1)(\epsilon^2 - m^2)\mu^2 + \gamma^2\epsilon^2 - m^2)\sqrt{\epsilon^2 - m^2} - 2\mu\epsilon\gamma(\epsilon^2 - m^2)\sqrt{\gamma^2 - 1})} \tag{4.21}$$

4.2 Inverse Compton scattering

Consider scattering of photons on the one electron, moving along z axis, following [9]. Klein-Nishina cross-section in electron's rest frame is

$$\frac{d\sigma}{d\epsilon'_1 d\Omega'_1} = \frac{{r_e}^2}{2}\left(\frac{\epsilon'_1}{\epsilon'_0}\right)^2\left(\frac{\epsilon'_1}{\epsilon'_0} + \frac{\epsilon'_0}{\epsilon'_1} - \sin^2\Theta'\right)\delta\left(\epsilon'_1 - \frac{\epsilon'_0}{1 + \frac{\epsilon'_0}{m_ec^2}(1 - \cos\Theta')}\right) \tag{4.22}$$

Where r_e - classical electron radius, ϵ'_0 and ϵ'_1 - photon initial and final energies respectively Θ' angle between initial and final photon directions, defined by expression $\cos\Theta' = \cos\theta'_0\cos\theta'_1 + \sin\theta'_0\sin\theta'_1\cos(\phi'_1 - \phi'_0)$. Primed values are measured in the electron's rest frame. Final and initial photon's energies are related as follow:

$$\epsilon'_1 = \frac{\epsilon'_0}{1 + \frac{\epsilon'_0}{m_ec^2}(1 - \cos\Theta')} \tag{4.23}$$

$$\epsilon'_0 = \frac{\epsilon'_1}{1 - \frac{\epsilon'_1}{m_ec^2}(1 - \cos\Theta')} \tag{4.24}$$

Number of photons, which are scattered in unit solid angle in unit energy range in time unit in electron's rest frame is

$$\frac{dN'}{dt'd\epsilon'_1 d\Omega'_1} = \int c\frac{d\sigma}{d\epsilon'_1 d\Omega'_1}\frac{dn'}{d\epsilon'_0 d\Omega'_0}d\Omega'_0 d\epsilon'_0 \tag{4.25}$$

Let rewrite delta-function in 4.22 with photon initial energy using relation

$$\delta(f(x)) = \sum \frac{\delta(x-x_k)}{|f'(x_k)|} \tag{4.26}$$

where x_k - are roots of $f(x)$. Derivative of expression inside delta-function is

$$\frac{d\epsilon'_1}{d\epsilon'_0} = \frac{1}{(1+\frac{\epsilon'_0}{m_e c^2}(1-\cos\Theta'))^2} \tag{4.27}$$

and it will cancel out with $(\epsilon'_1/\epsilon'_0)^2$ in 4.22. Initial photons distribution function in the lab frame is $\frac{dn}{d\epsilon_0 d\Omega_0}$, and we transform it to the electron frame using 4.16.

$$\frac{dN'}{dt'd\epsilon'_1 d\Omega'_1} = \int \frac{r_e^2 c}{2}\gamma_e^2(1-\mu_0\beta_e)^2\left(\frac{\epsilon'_1}{\epsilon'_0}+\frac{\epsilon'_0}{\epsilon'_1}-\sin^2\Theta'\right)\frac{dn}{d\epsilon_0 d\Omega_0}\delta\left(\epsilon'_0-\frac{\epsilon'_1}{1-\frac{\epsilon'_1}{m_e c^2}(1-\cos\Theta')}\right)d\epsilon'_0 d\mu'_0 d\phi'_0 \tag{4.28}$$

Now we get rid of delta-function by integrating over ϵ'_0

$$\frac{dN'}{dt'd\epsilon'_1 d\Omega'_1} = \int \frac{r_e^2 c}{2}\gamma_e^2(1-\mu_0\beta_e)^2\left(1+\cos^2\Theta'+\left(\frac{\epsilon'_1}{m_e c^2}\right)^2\frac{(1-\cos\Theta')^2}{1-\frac{\epsilon'_1}{m_e c^2}(1-\cos\Theta')}\right)\frac{dn}{d\epsilon_0 d\Omega_0}d\mu'_0 d\phi'_0 \tag{4.29}$$

Now we need to transform number of scattered photons per time unit per energy unit per solid angle into the lab frame $\frac{dN}{dtd\epsilon_1 d\Omega_1} = \frac{dN'}{dt'd\epsilon'_1 d\Omega'_1}\frac{dt'}{dt}\frac{d\epsilon'_1}{d\epsilon_1}\frac{d\Omega'_1}{d\Omega_1}$. Using $dt = \gamma_e dt'$, $\epsilon = \frac{1}{\gamma_e(1-\mu_1\beta_e)}\epsilon'$ and $\mu'_1 = \frac{\mu_1-\beta_e}{1-\mu_1\beta_e}$ we finally get

$$\frac{dN}{dtd\epsilon_1 d\Omega_1} = \int \frac{r_e^2 c}{2}\frac{(1-\mu_0\beta_e)^2}{1-\mu_1\beta_e}\left(1+\cos^2\Theta'+\left(\frac{\epsilon'_1}{m_e c^2}\right)^2\frac{(1-\cos\Theta')^2}{1-\frac{\epsilon'_1}{m_e c^2}(1-\cos\Theta')}\right)\frac{dn}{d\epsilon_0 d\Omega_0}d\mu'_0 d\phi'_0 \tag{4.30}$$

Also it may be useful to integrate in terms of lab frame, and expression for number of scattered photons would be following

$$\frac{dN}{dtd\epsilon_1 d\Omega_1} = \int \frac{r_e^2 c}{2}\frac{1}{\gamma_e^2(1-\mu_1\beta_e)}\left(1+\cos^2\Theta'+\left(\frac{\epsilon'_1}{m_e c^2}\right)^2\frac{(1-\cos\Theta')^2}{1-\frac{\epsilon'_1}{m_e c^2}(1-\cos\Theta')}\right)\frac{dn}{d\epsilon_0 d\Omega_0}d\mu_0 d\phi_0 \tag{4.31}$$

For integration one should express all angles in terms of integration variables. For evaluating photon flux from scattering on electron distribution, one should integrate formula 4.30 or 4.31 with electron distribution, normalized to the number density of particles. Note, that it is necessary to take into account different directions of electron movement, and perform corresponding rotations of coordinates.

In code compton evaluators return energy density of energy flux from the source in units of $\mathrm{cm}^{-2}\mathrm{s}^{-1}$. To obtain this value we perform integration of $\frac{dN}{dtd\epsilon_1 d\Omega_1}$ through the volume of the source, multiply it by energy of photon and divide by square of the distance to the source D.

$$F(\epsilon_1) = \frac{\epsilon_1}{D^2}\int \frac{dN}{dtd\epsilon_1 d\Omega_1}\frac{dn_e}{d\epsilon_e d\Omega_e} dV d\epsilon_e d\Omega_e \tag{4.32}$$

While considering processes including very high energy electrons ($\gamma_e \approx 10^8$) numerical errors can became very large, because such parameters as β_e и $\mu_0, \mu_1, \cos\Theta'$ are very close to 1 in important energy and angle ranges, and standard type double has not enough resolution to deal with such values. So in code we introduce following auxilary variables to reduce numerical errors :

$$\delta_e = 1 - \beta_e \tag{4.33}$$

$$\text{versin } \theta = 1 - \cos\theta \tag{4.34}$$

Then such expression as $1 - \mu\beta_e$ with this variables can be presented as

$$1 - \mu\beta_e = \text{versin } \theta + \delta_e - \text{versin } \theta \ \delta_e \tag{4.35}$$

and expression for angle between final and initial photons as

$$1 - \cos\Theta' = \text{versin } \theta_0' + \text{versin } \theta_1' - \text{versin } \theta_0' \text{versin } \theta_1' - \sin\theta_0' \sin\theta_1' \cos(\phi_1' - \phi_0') \tag{4.36}$$

Use of this expressions significantly increase accuracy of numerical integration in case of high energy photons and electrons.

In case of isotropic distributionfunctions of electrons and initial photons, it is possible to integrate cross-section analytically over the angle variables [10, 11], and in the equation for enrgy flux there are only integrations by photons and electrons energy

$$F(\epsilon_1) = \frac{\epsilon_1}{D^2}\int \frac{2\pi r_e^2 \beta_e c}{\epsilon_0 \gamma_e^2}\frac{dn_{ph}}{d\epsilon_0}\frac{dn_e}{d\epsilon_e}(2q\ \ln(q) + 1 + q - 2q^2 + \frac{q^2(1-q)\Gamma^2}{2(1+q\Gamma)}) d\epsilon_0 d\epsilon_e dV \tag{4.37}$$

where $\Gamma = 4\epsilon_0\gamma_e/m_e c^2$, $q = \epsilon_1/((\gamma_e m_e c^2 - \epsilon_1)\Gamma)$.

4.3 Синхротронное излучение

Process of synchrotron radiation of relativistic electrons is well-known and described in classical works as for example [6]. But synchrotron absorption is also possible. It's cross section was obtained in [21]. In code we will use spectral emissivity per unit volume and absorption coefficient, described in[7]. Emissivity (spectral density of energy radiated per unit time) per unit volume is

$$I(\nu) = \int_{E_{min}}^{E_{max}} dE \frac{\sqrt{3}e^3 nF(E)B\sin(\phi)}{m_e c^2}\frac{\nu}{\nu_c}\int_{\frac{\nu}{\nu_c}}^{\infty} K_{5/3}(x)dx, \tag{4.38}$$

where ϕ is angle between magnetic field and line of sight, ν_c is critical frequency defined as $\nu_c = 3e^2 B \sin(\phi) E^2 / 4\pi {m_e}^3 c^5$, and $K_{5/3}$ - Macdonald function. Absorption coefficient for photons, propagating along line of sight is

$$k(\nu) = \int_{E_{min}}^{E_{max}} dE \frac{\sqrt{3}e^3}{8\pi m_e \nu^2} \frac{nB\sin(\phi)}{E^2} \frac{d}{dE} E^2 F(E) \frac{\nu}{\nu_c} \int_{\frac{\nu}{\nu_c}}^{\infty} K_{5/3}(x) dx. \tag{4.39}$$

4.4 Pion decay

4.5 Bremsstahlung

References

1. Mathis J. S., Mezger P. G., Panagia N. Interstellar radiation field and dust temperatures in the diffuse interstellar medium and in giant molecular clouds // Astron. Astrophys.. — 1983. — Vol. 128. — P. 212–229.

2. Sironi Lorenzo, Spitkovsky Anatoly. Particle Acceleration in Relativistic Magnetized Collisionless Pair Shocks: Dependence of Shock Acceleration on Magnetic Obliquity // ApJ. — 2009. — Vol. 698, no. 2. — P. 1523–1549.

3. Guo Xinyi, Sironi Lorenzo, Narayan Ramesh. Non-thermal Electron Acceleration in Low Mach Number Collisionless Shocks. I. Particle Energy Spectra and Acceleration Mechanism // ApJ. — 2014. — Vol. 794, no. 2. — P. 153.

4. Crumley P., Caprioli D., Markoff S., Spitkovsky A. Kinetic simulations of mildly relativistic shocks - I. Particle acceleration in high Mach number shocks // MNRAS. — 2019. — Vol. 485, no. 4. — P. 5105–5119.

5. Romansky V. I., Bykov A. M., Osipov S. M. Electron and ion acceleration by relativistic shocks: particle-in-cell simulations // Journal of Physics Conference Series. — Vol. 1038 of Journal of Physics Conference Series. — 2018. — P. 012022.

6. Ginzburg V. L. Theoretical physics and astrophysics. Additional chapters. — 1975.

7. Ghisellini Gabriele. Radiative Processes in High Energy Astrophysics. — 2013. — Vol. 873.

8. Klein O., Nishina Y. The Scattering of Light by Free Electrons according to Dirac's New Relativistic Dynamics // Nature. — 1928. — Vol. 122, no. 3072. — P. 398–399.

9. Dubus G., Cerutti B., Henri G. The modulation of the gamma-ray emission from the binary LS 5039 // Astron. Astrophys.. — 2008. — Vol. 477, no. 3. — P. 691–700.

10. Jones Frank C. Calculated Spectrum of Inverse-Compton-Scattered Photons // Physical Review. — 1968. — Vol. 167, no. 5. — P. 1159–1169.

11. Bykov A. M., Chevalier R. A., Ellison D. C., Uvarov Yu. A. Nonthermal Emission from a Supernova Remnant in a Molecular Cloud // ApJ. — 2000. — Vol. 538, no. 1. — P. 203–216.

12. Coppejans D. L., Margutti R., Terreran G. et al. A Mildly Relativistic Outflow from the Energetic, Fast-rising Blue Optical Transient CSS161010 in a Dwarf Galaxy // ApJ Lett.. — 2020. — may. — Vol. 895, no. 1. — P. L23.

13. Kelner S. R., Aharonian F. A., Bugayov V. V. Energy spectra of gamma rays, electrons, and neutrinos produced at proton-proton interactions in the very high energy regime // Phys. Rev. D. — 2006. — Vol. 74, no. 3. — P. 034018.

14. Kafexhiu Ervin, Aharonian Felix, Taylor Andrew M., Vila Gabriela S. Parametrization of gamma-ray production cross sections for p p interactions in a broad proton energy range from the kinematic threshold to PeV energies // Phys. Rev. D. — 2014. — Vol. 90, no. 12. — P. 123014.

15. Bykov A. M., Kalyashova M. E. Modeling of GeV-TeV gamma-ray emission of Cygnus Cocoon // Advances in Space Research. — 2022. — Vol. 70, no. 9. — P. 2685–2695.

16. Ackermann M., Ajello M., Allafort A. et al. A Cocoon of Freshly Accelerated Cosmic Rays Detected by Fermi in the Cygnus Superbubble // Science. — 2011. — Vol. 334, no. 6059. — P. 1103.

17. Bartoli B., Bernardini P., Bi X. J. et al. Identification of the TeV Gamma-Ray Source ARGO J2031+4157 with the Cygnus Cocoon // ApJ. — 2014. — Vol. 790, no. 2. — P. 152.

18. Abeysekara A. U., Albert A., Alfaro R. et al. HAWC observations of the acceleration of very-high-energy cosmic rays in the Cygnus Cocoon // Nature Astronomy. — 2021. — Vol. 5. — P. 465–471.

19. Rybicki George B., Lightman Alan P. Radiative Processes in Astrophysics. — 1986.

20. Landau L. D., Lifshitz E. M. The classical theory of fields. — 1975.

21. Ghisellini Gabriele, Svensson Roland. The synchrotron and cyclo-synchrotron absorption cross-section // MNRAS. — 1991. — Vol. 252. — P. 313–318.