Modular **A**lgorithm for **R**elativistic **T**reatment of Heavy **I**o**N** **I**nteractions

Developer's Manual

Björn Schenke

McGill University, Montréal

2009

Contents

1 Installation **2**
1.1 MARTINI . . . 2
1.2 LHAPDF . . . 2
1.3 PYTHIA . . . 2
1.4 MARTINI . . . 3

2 How to write your own program **4**
2.1 Essentials . . . 4

3 Hydrodynamic background **7**
3.1 Hydro data . . . 7
3.1.1 General parameters . . . 7
3.1.2 Parameter values for the different hydro models . . . 8
3.1.3 Data structure . . . 10
3.2 Using the hydro background in the evolution . . . 14

4 Geometry **16**
4.1 Generate only one nucleon-nucleon collision per event . . . 16
4.2 Generate a full event . . . 17

5 AMY - radiative reactions **18**
5.1 Transition rates . . . 18
5.1.1 Sampling the rates . . . 20
5.1.2 Modified rates with suppressed small k_T . . . 22

6 Elastic collisions **24**
6.1 Total probabilities . . . 24
6.2 Sampling the energy transfer . . . 25
6.3 Sampling the momentum transfer . . . 26
6.4 Determining the new $\vec{p}'$ . . . 27
6.5 Conversion processes . . . 28

7 Photon production **29**
7.1 Photon conversion . . . 29
7.2 Photon radiation . . . 29

8 Fragmentation **31**

9 Post processing **33**
9.1 When using the original main.cpp . . . 33

Chapter 1

Installation

1.1 MARTINI

First just copy stuff in a directory. Details will follow once I know how it is distributed. `MARTINI` is the name of the directory with subdirs `main`, `pythia`, `hydro`, possibly `lhapdf` and `doc`.

1.2 LHAPDF

To get nuclear effects like shadowing to work MARTINI requires the Les Houches Accord Parton Distribution Function interface. LHAPDF provides a unified and easy to use interface to modern PDF sets. It is designed to work not only with individual PDF sets but also with the more recent multiple "error" sets. It can be viewed as the successor to PDFLIB, incorporating many of the older sets found in the latter, including pion and photon PDFs. Download LHAPDF from `http://www.hepforge.org/downloads/lhapdf` and install following the instructions.

After extracting the archive (to anywhere you like), in the main folder do
```
./configure --prefix=<absolute path to MARTINI>/lhapdf
```
Then
```
make
```
and
```
make install
```
`<absolute path to MARTINI>` is e.g. `/homes/nazca/schenke/pbswork/MARTINI`.

This will install LHAPDF as a sub folder into the MARTINI folder. Should you decide to install LHAPDF into a different folder you have to adjust the Makefiles. For one, change `MARTINI/main/src` so that you include the right path: change `-I ../../lhapdf/include` to your path to LHAPDF. Also adjust the Makefile in `MARTINI/main` so that `-L../lhapdf/lib -lLHAPDF` will be `-L <your path>/lib -lLHAPDF`. Finally, you need to set a global variable to locate the dynamic library. If using bash edit the file `.bashrc` in your home directory and add: `export LD_LIBRARY_PATH=<absolute path to MARTINI>/lhapdf/lib`.

Really, PYTHIA is the one that uses LHAPDF directly. But as long as MARTINI knows where it is PYTHIA will do too.

You can also install LHAPDF as root under `/usr/share`, make sure however that you adjust the Makefiles accordingly.

1.3 PYTHIA

PYTHIA is located in `MARTINI/pythia`. Since we are using a slightly modified version of PYTHIA 8.1, it is at this point not recommended (in fact, it won't work) to put another version here. We will work on allowing for easy exchange of the PYTHIA version. To get everything running on a new mashine for the first time, before compiling MARTINI go to the `pythia` folder and do:

```
make clean
./configure
make
```

Now PYTHIA is ready.

1.4 MARTINI

Finally you can compile MARTINI. Just go to `MARTINI/main` and do

```
make
```

To enforce a new compilation do

```
make clean
make
```

or do `make -B`

Chapter 2

How to write your own program

2.1 Essentials

To write your own program including `MARTINI`, you need to have a line

```
#include "MARTINI.h"
```

in your main program. Then, three commands are essential to initialize `MARTINI`. Here is a short beginning of an example main program, declaring the object `martini`, which is an instance of the class `MARTINI`:

```
int main(int argc, char* argv[])
{
  MARTINI martini;
  martini.readFile("setup.dat");
  martini.init(argc, argv);
  ...
}
```

The second command is not really mandatory, but makes the use of your own parameter sets a lot easier. If you do not change any paramter from its default you do not need this. If you only whish to change a few parameters, you can do this using `martini.readString("string")`. Note that the way parameters are set is completely analogous to PYTHIA 8.1. The third command actually initializes martini. Currently the command prompt arguments that are passed to `martini.init` have no function. Note that it is essential to set all parameters **before** calling `martini.init(argc, argv)` so that `MARTINI` really initializes with the parameter set you desire. To see what parameters `MARTINI` is using, include the line

```
martini.settings.listAll();
```

Again, note that this is just like in PYTHIA 8.1. To understand the whole settings scheme please refer to the PYTHIA 8.1 online manual at e.g. `http://home.thep.lu.se/~torbjorn/php8130/Welcome.php` under `Settings Scheme`. Just replace `pythia` with `martini` and that is how it works for `MARTINI`.

To modify PYTHIA parameters when using `MARTINI` it goes the same way. Just read in the string containing the PYTHIA parameter name and value, either within your setup file (e.g. `"setup.dat"`) or using `martini.readString("pythiaParameterString")`. There are no `MARTINI` parameters that conincide with names of PYTHIA parameters. So it will be clear what you want to change.

A main program includes the genration of an event and the subsequent evolution:

```
int runs=martini.returnRuns();

vector<Parton> ** plist;          // pointer to array of vector<Parton> objects
plist = new vector<Parton> *[2];
plist[0] = new vector<Parton>;    // plist[0] is the main list that holds the
                                  // high momentum partons that are evolved

for (int j=0; j<runs; j++)        // loop over all events
  {
    plist[0]=martini.next();
    ...
  }
```

First declare a `vector` of `Parton` objects, which is then filled in every event by using `plist[0]=martini.next()`. After that you are free to extract the event information and do binnings or whatever - that is the "..." part. Alternatively, one could write ones own `MARTINI::next` routine, e.g. when one wants to fill the initial hard parton list with something other than PYTHIA (e.g. one parton with fixed energy and momentum). This could look like in the following

```
Parton jp1;
jp1.id(1); //make the parton an up quark
jp1.p(Ejet,0.,0.); //set the parton's momentum
jp1.col(101); jp1.acol(102); //set color and anti-color index
jp1.x(0.); jp1.y(0.); jp1.z(0.); //set initial position
jp1.xini(jp1.x()); jp1.yini(jp1.y()); jp1.zini(jp1.z());

mt = static_cast<int>(maxTime/dtfm+0.0001); // compute number of time steps
for (int j=0; j<runs; j++)     // loop over all events
 {
   counter = 0;                // reset counter in the beginning of every event
   plist[0]->clear();          // clear the parton list
   if( martini.returnFixedEnergy() == 0 ) // if options ask for PYTHIA event, do this
     {
       martini.generateEvent(plist[0]);
     }
    else                       // if options want a parton with fixed energy, do this
     {
       plist[0]->push_back( jp1 ); // add the previously defined parton to the list
     }
   if (martini.returnEvolution() == 1)    // evolve in medium if settings want it
     {
       for(int i=0; i<mt; i++) // loop over all time steps
         {
           counter = martini.evolve(plist, counter, i);
           counter+=1;
         }
     }
  ...
 }
```

First, the parton object is declared and filled with information on kind of parton, momentum, position, and color. Then, the loop over runs starts and there is a choice between PYTHIA event (`martini.returnFixedEnergy() == 0`) and our own initial setup, which in this simple case just adds the one parton `jp1` to the list. Then the evolution over the `mt` time steps is started (if the user wanted an evolution (`martini.returnEvolution() == 1`). Note the use of `counter` which allows to identify every step, every parton, and by that within `MARTINI` every color index uniquely. `counter` is also increased within `MARTINI::evolve`.

Chapter 3

Hydrodynamic background

3.1 Hydro data

Hydro data is given for *tau*, x, y, and z (for the 3D case). The original Lagrangian 3D hydro data was converted from η to z. The flow velocities are still v_x, v_y, v_η though and will be converted in the main program.

3.1.1 General parameters

Parameters in the file `./main/xmldoc/Hydro.xml`:

- `Hydro:WhichHydro`: choose a hydro evolution model:
  1. Kolb 2+1D Hydro (see page 8 for parameters and page 10 for data structure)
  2. Eskola 2+1D Hydro (see page 8 for parameters and page 11 for data structure)
  3. Nonaka 3+1D Hydro (see page 8 for parameters and page 12 for data structure)
  4. Schenke 3+1D Hydro (see page 8 for parameters and page 13 for data structure)
- `Hydro:Subset`: choose a parameter subset - only available for `WhichHydro`=4.
- `Hydro:tau0`: initial τ_0 [fm] in the data file.
- `Hydro:taumax`: final time $\tau_{\max}$ [fm] in the files, can be changed to read in less data.
- `Hydro:dtau`: step size in time $\Delta\tau$ [fm], as used in the data files
- `Hydro:xmax`: maximal x and y [fm] (same for both transverse directions). Data runs from $-x_{max}$ to $+x_{max} - \Delta x$ such that there are $(2x_{max})/\Delta x$ steps, including $x = 0$.
- `Hydro:zmax`: maximal z [fm] - only relevant in the 3D hydro case. Principally the same as `xmax` but the value can of course be different.
- `Hydro:dx`: step size in both transverse directions $\Delta x = \Delta y$ [fm], as used in the data files.
- `Hydro:dz`: step size Δz [fm] in the z direction as used in the data files - only relevant in the 3D hydro case.
- `Hydro:Tfinal`: temperature [GeV] at which to stop the evolution of the jet. Usually T_{crit}.

3.1.2 Parameter values for the different hydro models

The following parameter values are currently used in the three different cases implemented. These are due to change - also more models can be implemented. All parameter names are preceded by `Hydro:` .

Kolb 2+1D Hydro

`whichHydro`	1	
`tau0`	0.6	`fm`
`taumax`	17.16	`fm`
`dtau`	0.04	`fm`
`xmax`	10.	`fm`
`dx`	0.1	`fm`
`Tfinal`	0.16	`GeV`

Eskola 2+1D Hydro

`whichHydro`	2	
`tau0`	0.17	`fm`
`taumax`	30.171565	`fm`
`dtau`	0.1	`fm`
`xmax`	15.	`fm`
`dx`	0.1	`fm`
`Tfinal`	0.16	`GeV`

Nonaka 3+1D Hydro

`whichHydro`	3	
`tau0`	0.6	`fm`
`taumax`	17*	`fm`
`dtau`	0.1	`fm`
`xmax`	10.	`fm`
`zmax`	20.	`fm`
`dx`	0.25	`fm`
`dz`	0.5	`fm`
`Tfinal`	0.16	`GeV`

Schenke/Jeon/Gale 3+1D Hydro **

`whichHydro`	4	
`Subset`	any int	
`tau0`	0.55	`fm`
`taumax`	20	`fm`
`dtau`	0.1	`fm`
`xmax`	10.	`fm`
`zmax`	20.	`fm`
`dx`	0.5	`fm`
`dz`	0.5	`fm`
`Tfinal`	0.12	`GeV`

* this is the absolute maximum. You can choose smaller values. That would save time and memory. Maximal values may change for different impact parameters. Usually is no problem because for larger impact parameters the evolution will stop earlier such that MARTINI will not try to access data that is not there...

The Nonaka 3+1D hydro is available for four impact parameters at the moment: $b = 2.4, 4.5, 6.3$ and $7.5\,\mathrm{fm}/c$. Every data set is stored in a separate folder under `./hydro/nonaka/bx.x`. Also, note that for the three larger impact parameters available ($b = 4.5, 6.3$ and $7.5\,\mathrm{fm}/c$), the data files available only contain information up to $\tau = 20\,\mathrm{fm}/c =$`taumax`.

** these are just typical values - please check the `input` file in the correct directory and for the correct subset and adjust accordingly. The Schenke 3+1D hydro allows for including different subsets of parameters for a given impact parameter. The parameters used for the subset using `evolutionN.dat`, where `N` is the number of the subset, are stored in the same directory in the file `inputN`.

3.1.3 Data structure

Let us briefly discuss the ordering in the data files and how to read them in.

Kolb 2+1D Hydro

The data is stored in the subfolder `./hydro`. Each data file corresponds to one time step. There is one for the temperature `Txxx.dat`, one for the flow velocity in the x direction at $\eta = 0$ `VXxxx.dat`, and one for the flow velocity in the y direction at $\eta = 0$. Each file contains information for all x and y values, where x changes in the inner loop, y in the outer. When read in , the data will be ordered like this:

```
position = ix+ixmax*(iy+iymax*itau)
```

where each step is of size `dx` and we start at `-xmax`, for both x and y.

Here is an example code snippet for reading one file:

```
    // temperature
    fin.open(file[0].c_str(),ios::in);
    int ik = 0;
    while ( !fin.eof() )
      {
        ik++;
        fin >> T;
        newCell.T(T);
        if (ik<=ixmax*ixmax) lattice->push_back(newCell);
      }
    fin.close();
    // flow in x direction
    fin.open(file[1].c_str(),ios::in);
    position=ixmax*ixmax*i;
    ik = 0;
    while ( !fin.eof() )
      {
        ik++;
        fin >> vx;
        if (ik<=ixmax*ixmax)
          {
            lattice->at(position).vx(vx);
            position++;
          }
      }
   fin.close();
```

Eskola 2+1D Hydro

The data is stored in the sub-folder `./hydro/RHIC200`. There is one file for each the temperature, and the flow velocity in the radial direction (this is for imoact parameter $b = 0$ only, so the data is isotropic in the transverse plane), `TMATIn` and `VrMATin`. The files contain the whole evolution in the coordinates r and τ. r changes in the inner loop, τ in the outer loop. Also here, the data will be structured like this after read in:

```
position = ix+ixmax*(iy+iymax*itau)
```

Here is an example code to read data from the files:

```
  // temperature
  fin.open(file[0].c_str(),ios::in);
  int i = 0;
  int ir, itau;
  while ( !fin.eof() )
    {
      fin >> T;
      ir=(i)%ixmax;
      itau=floor((i)/ixmax);
      if (itau<itaumax && ir<ixmax) TofRandTau[itau][ir] = T;
      i++;
    }
  fin.close();
  for(itau=0; itau<itaumax; itau++)
  for(int ix=0; ix<ixmax; ix++)
  for(int iy=0; iy<ixmax; iy++)
     {
       x = -xmax + ix*dx;
       y = -xmax + iy*dx;
       r=sqrt(x*x+y*y);
       ir=floor(r/dx);
       if ( ir<ixmax ) T=TofRandTau[itau][ir];
       else T=0.;
       newCell.T(T);
       lattice->push_back(newCell);
     }
```

Note that we have to convert from r to x and y to get the same structure as for the other hydro data. For more code see the file `./main/src/HydroSetup.cpp`.

Nonaka 3+1D Hydro

The structure is similar to that of the Kolb data. The data is stored in the subfolder `./hydro/nonaka/bx.x`, where `bx.x` is the sub foldercontaining the data for a particular impact parameter (b =`x.x` in fm, e.g. 2.4 fm). Each data file corresponds to one time step. There is one for the temperature `Tx.dat`, and one for the flow velocities in the x, y and η direction `Vx.dat` (`x` is the number of the time step). Note: It is really $\tilde{v}_\eta$ that is stored, despite the coordinates are x, y, and z. The temperature file also includes a column indicating the fraction of QGP present in this cell - this is relevant in the mixed phase. Each file contains information for all x, y, and z values, where x changes in the inner loop, then y, then z in the outer. When read in, the data will be ordered like this:

```
position = ix + ixmax*(iy+ixmax*(iz+izmax*itau))
```

So there is just one more dimension.

Let's look at the code that reads the data:

```
  // read in temperature
  fin.open(file[0].c_str(),ios::in);
  int ik = 0;
  while ( !fin.eof() )
    {
      ik++;
      fin >> T;
      fin >> QGPfrac;
      newCell.T(T);
      newCell.QGPfrac(QGPfrac);
      if (ik<=ixmax*ixmax*izmax) lattice->push_back(newCell);
    }
  fin.close();
  // read in flow in x, y, eta direction
  fin.open(file[1].c_str(),ios::in);
  ik = 0;
  position=ixmax*ixmax*izmax*i;
  while ( !fin.eof() )
    {
      ik++;
      fin >> vx;
      fin >> vy;
      fin >> veta;
      if (ik<=ixmax*ixmax*izmax)
        {
          lattice->at(position).vx(vx);
          lattice->at(position).vy(vy);
          lattice->at(position).vz(veta);
          position++;
        }
    }
  fin.close();
```

Schenke/Jeon/Gale 3+1D Hydro

The structure is similar to that of the Nonaka data. The data is stored in the subfolder `./hydro/schenke/bx.x`, where `bx.x` is the sub folder containing the data for a particular impact parameter ($b =$`x.x` in fm, e.g. 2.4 fm). All data is stored in the file `evolution.dat` or `evolutionN.dat`, where `N` indicates the parameter subset (`N`> 1). The information on what the used parameters are is stored in the files `input` or `inputN`. The columns in the files `evolution.dat` and `evolutionN.dat` are in this order:
temperature, QGP fraction, v_x, v_y, v_z.

3.2 Using the hydro background in the evolution

The data is read into the `lattice`, a `vector` of `HydroCell` objects, which are defined in `HydroCell.h`:

```
class HydroCell
{
 private:
  double itsT;
  Vec4   itsV;
  double itsQGPfrac;

 public:
  HydroCell(){};//constructor
  ~HydroCell(){};//destructor

  // flow velocity
  Vec4 v() const { return itsV; };
  void v(Vec4 value) { itsV=value; };

  double vx() const { return itsV.px(); };
  double vy() const { return itsV.py(); };
  double vz() const { return itsV.pz(); };
  void vx( double value ) { itsV.px(value); };
  void vy( double value ) { itsV.py(value); };
  void vz( double value ) { itsV.pz(value); };

  // temperature
  double T() const { return itsT; };
  void T(double value) { itsT=value; };

  // QGP fraction
  double QGPfrac() const { return itsQGPfrac; };
  void QGPfrac(double value) { itsQGPfrac = value; };
};
```

In the evolution, we access this data. In `MARTINI::evolve`, we determine the position of the parton that we want to evolve:

```
x = plist[0]->at(i).x();                  // x value of position in [fm]
y = plist[0]->at(i).y();                  // y value of position in [fm]
z = plist[0]->at(i).z();                  // z value of position in [fm]
```

Now determine the temperature in that cell (`t` is the current time), by reading and interpolating the data, and passing the result to the `hydroInfo` structure, defined in `Basics.h`. It simply stores the temperature, flow velocity, and QGP fraction values for the cell we are in.

The reading and interpolating is done in `HydroSetup::getHydroValues`.

```
hydroInfo = hydroSetup->getHydroValues(x, y, z, t, hydroXmax, hydroZmax, hydroTau0,
                                       hydroDx, hydroDz, hydroDtau, hydroWhichHydro,
                                       lattice);
```

In the 2+1D case `getHydroValues` reads the 8 values at the corners of a rectangle around the position (x, y, τ) (after converting z and t to τ). Then it interpolates linearly to get the value of T, v_x, v_y at the position (x, y, τ).

In the 3+1D case, the procedure is the same, only that we have a 4 dimensional rectangle around the point (x, y, z, τ) now, so that we have 16 values from which we interpolate.

The "front bottom left corner" of the rectangle in space is given by

```
ix = floor((hydroXmax+x)/hydroDx+0.0001);
iy = floor((hydroXmax+y)/hydroDx+0.0001);
iz = floor((hydroZmax+z)/hydroDz+0.0001);
```

Note that `x` and `y` run from `-hydroXmax` to `+hydroXmax` and `ix` and `iy` from `0` to `2*hydroXmax`, hence the `hydroXmax`+`x` or `y` for both. Analogously for `z`.

The position on the `tau` grid is

```
itau = floor((tau-hydroTau0)/hydroDtau+0.0001);
```

From these positions we reach the other corners by going one step in the different directions.

The flow velocity needs additional transformations, depending on η. In the 2+1D case we need to do the following:

$$\beta_x = v_x/\cosh(\eta) = v_x\,\tau/t \tag{3.1}$$

$$\beta_y = v_y/\cosh(\eta) = v_y\,\tau/t \tag{3.2}$$

$$\beta_z = z/t\,, \tag{3.3}$$

where $\eta = 0.5\ \ln\left(\frac{t+z}{t-z}\right)$.

In the Nonaka 3+1D case we need to transform back from the $\tilde{\mathbf{v}} = (\tilde{v}_x, \tilde{v}_y, \tilde{v}_\eta)$ that is in the data files (this is just the inverse of Eqs.(4) in [NB07]):

$$\beta_x = \tilde{v}_x\ \cosh(\tilde{y})/\cosh(\tilde{y}+\eta) \tag{3.4}$$

$$\beta_y = \tilde{v}_y\ \cosh(\tilde{y})/\cosh(\tilde{y}+\eta) \tag{3.5}$$

$$\beta_z = (\tilde{v}_\eta + \tanh(\tilde{y}))/(1 + \tilde{v}_\eta\ \tanh(\eta)) \tag{3.6}$$

where $\tilde{y} = 0.5\ \ln\left(\frac{1+\tilde{v}_\eta}{1-\tilde{v}_\eta}\right)$.

In the Schenke 3+1D case this is not necessary.

All this is done in `getHydroValues` and the flow velocity that is needed to boost to the cell's rest frame is passed to `MARTINI`.

Chapter 4

Geometry

The initial nucleon-nucleon collisions are sampled using the Glauber model. The file `Glauber.cpp` contains all the relevant routines. There are two possible ways implemented.

4.1 Generate only one nucleon-nucleon collision per event

`generateSimpleEvent` in `MARTINI` just produces one nucleon-nucleon collision in a heavy-ion background. This provides a simple way of calculating quantities like R_{AA}, because then we only have to take the ratio of the heavy-ion result over the pp result, disregarding the number of nucleon-nucleon collisions that we actually should have produced. The position in the transverse plane of the one nucleon-nucleon collision is determined by sampling the overlap function using `glauber->SamplePABRejection(random)` (here `glauber` is a `Glauber` object). It returns a `ReturnValue` object, defined in `Basics.h`:

```
struct ReturnValue
{
  double x;
  double y;
  int rejections;
  int acceptances;
};
```

all what is needed is really the `x` and `y` value here. `glauber->SamplePABRejection(random)` samples `PAB(x,y)`, also defined in `Glauber.cpp` using the rejection method. There is also a `glauber->SamplePAB(random)`, which uses Metropolis, but it is currently unused. `PAB(x,y)` is given by the following expression:

```
double Glauber::PAB(double x, double y)
{
  double s1=sqrt(pow(x+b/2.,2.)+y*y);
  double s2=sqrt(pow(x-b/2.,2.)+y*y);
  return InterNuPInSP(s1)*InterNuTInST(s2)/(currentTAB*LexusData.SigmaNN);
}/* PAB */
```

It returns

$$\mathcal{P}_{AB}(b,\mathbf{r}_\perp) = \frac{T_A(\mathbf{r}_\perp + \mathbf{b}/2)T_B(\mathbf{r}_\perp - \mathbf{b}/2)}{T_{AB}(b)}, \qquad (4.1)$$

note however that here `InterNuPInSP(s1)*InterNuTInST(s2)` is the product of particle numbers in the projectile and target, respectively ($T_A\,\sigma_{\rm inel}$). The additional factor of $\sigma_{\rm inel}^2$ is removed by dividing by `currentTAB`, which also has a factor of σ_{inel} and the explicit σ_{inel} =`LexusData.SigmaNN`. Important: because `currentTAB`, which holds the current value obtained from evaluating `double Glauber::TAB()` already contains a factor of σ_{inel}, it is not actually T_{AB} but the total number of binary collisions.

4.2 Generate a full event

To generate a full event (using `generateEvent` in `MARTINI.cpp`) with realistic fluctuations of the number of binary collisions we use a different approach than that explained above. Instead of sampling the overlap function, we sample the individual thickness functions T_A and T_B (using `void MARTINI::sampleTA()`, which calls `glauber->SampleTARejection(random)`) to determine the number of nucleons in cells in the transverse plane of size σ_{inel} =`inelasticXSec`. We then overlay the two "lattices" depending on the impact parameter. We loop over all cells and the number of nucleons on lattice A in the current cell and on lattice B in the same cell and determine whether two nucleons collide depending on the probability $\sigma_{\text{jet}}/\sigma_{\text{inel}}$. If a collision happens we set its position to lie at a random transverse position *within* the current cell.

```
for (int ix = 0; ix < ixmax; ix++) // loop over cells in x-direction
   for (int iy = 0; iy < ixmax; iy++) // loop over cells in y-direction
      {
         for (int i = 0; i < nucALat[ix][iy]; i++) //nucleons of nucleus A in the cell
            for (int j = 0; j < nucBLat[ix][iy]; j++) //nucleons of nucleus B in the cell
               { ...
```

σ_{jet} is provided by `PYTHIA` and depends on the minimum p_T we chose in the beginning. Note that in full events we always need a minimum p_T to be set, otherwise we may end up in a situation where $\sigma_{\text{jet}} > \sigma_{\text{inel}}$, since σ_{jet} is calculated perturbatively.

Then at the positions where an event happens a PYTHIA evet is called using `pythia.next()`. PYTHIA perfromes its vaccum showers up to a scale given by $1/\tau_0$, the time when the hydro evolution starts. Partons are going to be moved in the evolution routine even before that time to take care of the shift in their position. This may be improved, because partons radiated in time-like showers in PYTHIA may not get the exactly correct position at τ_0 doing this. Here we assume that all partons were generated at the initial position of the nucleon-nucleon collision and then are moved according to their final momentum (final after the PYTHIA evolution) - this is not exactly right, but probably a good approximation.

Finally we go through all partons that PYTHIA generated in one even, set the parton properties and add it to the list of partons (here we show a different order than in the program):

```
parton.id(pythia.event[ip].id());               // set parton id
parton.status(pythia.event[ip].status());       // set parton status
parton.mass(pythia.event[ip].m());              // set mass
parton.x(xmin+(ix)*cellLength+xPositionInCell);// set position
parton.y(xmin+(iy)*cellLength+yPositionInCell);
parton.z(0.);
parton.col(pythia.event[ip].col());             // set color
parton.acol(pythia.event[ip].acol());           // set anti-color
parton.p(pythia.event[ip].p());                 // set momentum
plist->push_back(parton);                       // add the parton to the main list
```

Chapter 5

AMY - radiative reactions

5.1 Transition rates

The transition rates have been computed earlier and are saved in the file `./data/radgamma`. They are read in using `Import.cpp`. Also, `import`, will provide the rates using its public functions:

```
double Import::getRate(double p, double k)
{
  return use_table ( p , k , Gam.dGamma , 0 );
}

double Import::getRate_gqq(double p, double k)
{
  if(k<p/2) return use_table ( p , k , Gam.dGamma_gqq , 1 );
  else return 0.;
}

double Import::getRate_ggg(double p, double k)
{
  if(k<p/2) return use_table ( p , k , Gam.dGamma_ggg , 2 );
  else return 0.;
}
```

The rest is done by the functions in `Rates.cpp` Let us follow `MARTINI:evolve`, first we determine the total probability in one time step for all the three processes and store them in `n`:

```
 // n stores the areas under the prob. distr.
n = rates->integrate(pRest/T,T,alpha_s, Nf);
```

`rates` is the pointer to the `Rates`-object, and `rates->integrate` returns the integrated transition rate. These integrals are parameterized and reperesent the numerical result to very good precision:

```
Norms Rates::integrate(double p, double T, double alpha_s, int Nf)
{
  Norms norms;
  double fac = 4*PI*alpha_s*4*PI*alpha_s*T;
  norms.Gamma = ( (0.49384-0.463155/(p*p)+0.15763/p-0.16124/sqrt(p))
               +(0.3825932+0.03217262/p))*fac;
  norms.Gamma_gqq = (Nf*(0.04392766/(p*p)-0.05353506/p+0.03259168/pow(p,0.8)
                    +0.00191753/pow(p,0.2))
                   + Nf*(0.05749489/(p*p)-0.03112226/pow(p,1.8)+0.00445603/p))*fac;
  norms.Gamma_ggg = ((1.1104368+1.433124/(p*p*p)+0.16666408/p-0.33858792/sqrt(p))
                     + (0.85739544+0.2125156/p))*fac;
  return norms;
}
```

Once these values are determined, we multiply them by the time step size Δt and get the probabilities for the three processes to occur in one time step for a parton with energy `p`.

Let us look at the evolution for two processes as an example ($q \to qg$ and $\bar{q} \to \bar{q}g$):

```
 // see if emission happens, dt/gamma is dt_rest-frame
if(random->genrand64_real1() < dt/gamma*(n.Gamma))
  {
   if(pRest/T>4.01) // do not evolve partons with momenta below this scale
     {
          // do process 1, q->qg
         f = rates->findValuesRejection(pRest/T, T, alpha_s, Nf, random, import, 1);
          // quark's new momentum in the lab frame
         p = (pRest - f.y*T)*boostBack;
          // new quark momentum
         newvecp=p/vecp.pAbs()*vecp;
          // change quark's momentum to the new one
         if (p>=0) plist[0]->at(i).p(newvecp);
          // if new parton is kept (f.y*T > threshold [in GeV])
         if (f.y*T>pCut)
           {
              // emitted gluon's momentum
             newOne.p(f.y*T*vecp.px()/vecp.pAbs()*boostBack,
             f.y*T*vecp.py()/vecp.pAbs()*boostBack,
             f.y*T*vecp.pz()/vecp.pAbs()*boostBack);
              // color of original quark
             col = plist[0]->at(i).col();
              // anti-color
             acol = plist[0]->at(i).acol();
[...]
```

`f` stores the momentum of the emitted gluon, as it was sampled from the transition rate in `rates->findValuesRejection`. In the following we take care of all the other properties of the particles - mainly color, position, and kind (`id`).

```
[...]
               // if we had a quark
              if (col!=0)
                {
                   // set color to new color
                  plist[0]->at(i).col(10000+counter);
                   // set new gluon's color to quark's original color
                  newOne.col(col);
                   // set new gluon's anti-color to quark's new color
                  newOne.acol(10000+counter);
                }
               // if we had an anti-quark
              else if (acol!=0)
                {
                   // set color to zero
                  plist[0]->at(i).col(0);
                   // set new particle's color
                  newOne.col(10000+counter);
                   // set new particle's anti-color
                  newOne.acol(acol);
                }
               // emitted parton is a gluon
              newOne.id(21);
              newOne.mass(0.);
              newOne.frozen(0);
               // set the new parton's initial position
              newOne.x(x);
              newOne.y(y);
              newOne.z(z);
                // add the gluon to the list of partons if (f.y>4.01)
              plist[0]->push_back(newOne);
            }
        }
    }
```

Now, let us see what `f = rates->findValuesRejection` does exactly.

5.1.1 Sampling the rates

In

```
ReturnValue Rates::findValuesRejection(double u, double T, double alpha_s, int Nf,
                                       Random *random, Import *import, int process)
```

`u` is the energy of the emitting jet, `T` is the temperature, `alpha_s` the strong coupling constant, `Nf` the number of flavors used, `random` is a `Random` object, providing functions for sampling random numbers, primarily uniformly distributed ones, `import` holds the tabulated rates, and `process` indicates which rate is to be sampled:

1. $q \to qg$

2. $q \to q\bar{q}$
3. $g \to gg$
4. $q \to q\gamma$

In the beginning, the values `Pos = integratePos(u,T,alpha_s, Nf);` and `Neg = integrateNeg(u,T,alpha_s, Nf);` are set. They are `Norms` objects as described above and hold the integral over the rate for positive k and negative k respectively. Using them, we first decide whether k will be positive or negative in this process. According to this, we can then use the appropriate envelope function for that region. E.g. for process 1, we have

```
posNegSwitch = 1; // first assume k is positive
// and then change it if we decide it is not:
if (random->genrand64_real1()<Neg.Gamma/(Neg.Gamma+Pos.Gamma)) posNegSwitch = 0;
```

Then, if k is positive for example, we do the sampling:

```
 if( posNegSwitch == 1 ) // if k > 0
{
  do
    {
      y = 2.5/(gsl_sf_lambert_W0(2.59235*pow(10.,23.)
                                  *exp(-100*random->genrand64_real1()
                                  *area(u+12.,u,posNegSwitch,1,import))));
      // here random->genrand64_real1()*area(u+12.,posNegSwitch)
      // is a uniform random number on [0, area under f(x)]
      x = random->genrand64_real1();
      // x is uniform on [0,1]
      //i++;
    } while( x > (function(u,y,import,1))/((0.025/(y*y))+0.01/y));
  // reject if x is larger than the ratio p(y)/f(y), f(y)=0.025/(y*y)+0.01/y
  f.y=y;
}
```

y is distributed as

$$y(x) = 2.5/W_0(2.59235\,10^{23}\,\exp(-100\,x))\,, \tag{5.1}$$

where W_0 is the Lambert W-function (or ProductLog)($W(z)$ is the principal solution to $z = W(z)\exp(W(z))$), and sampled using the random number `random->genrand64_real1()*area(u+12.,u,posNegSwitch,1,import)`, which is uniformly distributed between 0 and `area(u+12.,u,posNegSwitch,1,import)`, the total area

$$F(y) = 0.5299 - 0.025/y + 0.01\,\log(y); \tag{5.2}$$

under the envelope function, which is given by

$$f(y) = \frac{0.025}{y^2} + \frac{0.01}{y}\,. \tag{5.3}$$

Now if another random variable x on $[0,1]$ is larger than the ratio $f_{\text{actual rate}}(y)/f(y)$, then the value for y is rejected, and a new one is sampled. This goes on until a y is accepted. The close the envelope to the actual rate, the less rejections and the more effective the sampling will be. In the program we have $f_{\text{actual rate}}(y) =$ `function(u,y,import,1)`.

All other cases work analogously, including the photon emission, to which we return in Section 7.2.

5.1.2 Modified rates with suppressed small k_T

In order to study the contribution from large angle emissions which should be small by assumption of collinearity in the derivation of the AMY rates, we introduce an envelope function when integrating the DGL. This is done by the replacement

$$\int d^2k_\perp \, k_\perp \cdot F(k_\perp) \Rightarrow \int d^2k_\perp \, k_\perp \cdot F(k_\perp) \exp(-k_\perp^2/k_{\max}^2) \tag{5.4}$$

where $q_\perp$ is the transverse momentum relative to the largest momentum particle, that is, h/p where p is the incoming momentum.

The form is chosen so that the cutoff occurs when $k_\perp \simeq k_{\max}$, the exact constant is chosen because $\int d^2k_\perp \exp(-k_\perp^2/k_{\max}^2) = \pi k_{\max}^2$ is the area of the disk in $k_\perp$ space of radius $k_{\max}$.

This modification replaces the evaluation of $\Re f(b)|_{b=0}$ with an integral over small b, weighted by the Fourier transform of the envelope $\exp(-k_\perp^2/k_{\max}^2)$ defined above.

To convert units (since $k_\perp$ is in units of m_D and k, p are in units of T), we need the Debye mass:

$$m_D^2 = \frac{4\pi\alpha_s}{6}(2N_c + N_f)T^2 \tag{5.5}$$

Then the maximum value for $k_\perp$ is (converting it to being in units of m_D):

$$\begin{aligned} k_{\max}^2 &= k^2T^2/m_D^2 && \text{if } |p| > |k| \text{ and } |p-k| > k && (5.6)\\ &= (p-k)^2T^2/m_D^2 && \text{if } |p| > |k| \text{ and } |p-k| < |k| && (5.7)\\ &= (p-k)^2T^2/m_D^2 && \text{if } |p| < |k| \text{ and } |p| > |p-k| && (5.8)\\ &= p^2T^2/m_D^2 && \text{if } |p| < |k| \text{ and } |p| < |p-k| && (5.9) \end{aligned}$$

The 2D-Fourier transform of the envelope function is

$$\frac{k_{\max}^2}{2} \exp(-k_{\max}^2 b^2/4) \tag{5.10}$$

This has to be multiplied by the Jacobian b, for we are in polar coordiantes.

In case we want to suppress angles smaller than $\pi/4$ we modify the envelope function to read

$$\exp(-k_\perp^2/(\sin(\pi/4)k_{\max})^2)\,. \tag{5.11}$$

Both envelope functions are shown in Fig. 5.1.

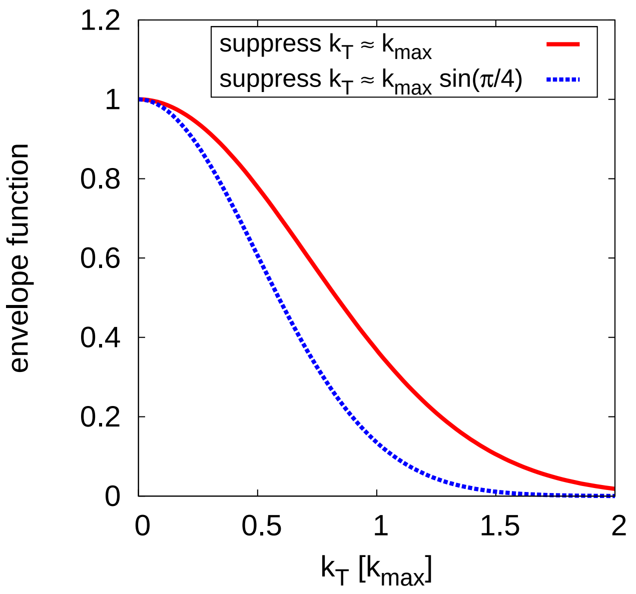

Figure 5.1: Used envelope functions to suppress $k_T \simeq k_{\max}$ (solid) and $k_T \simeq k_{\max} \sin(\pi/4)$ which corresponds to a suppression of emmission angles $\theta \gtrsim \pi/4$.

Chapter 6

Elastic collisions

Everything described here happens in the rest frame of the cell. There are four processes that can happen:

1. $qq \to qq$
2. $gq \to gq$
3. $qg \to qg$
4. $gg \to gg$

where q can be a quark or anti-quark. The second parton is always the thermal one, while the first is the jetty one. Fig. 6.1 shows the processes in terms of Feynman diagrams.

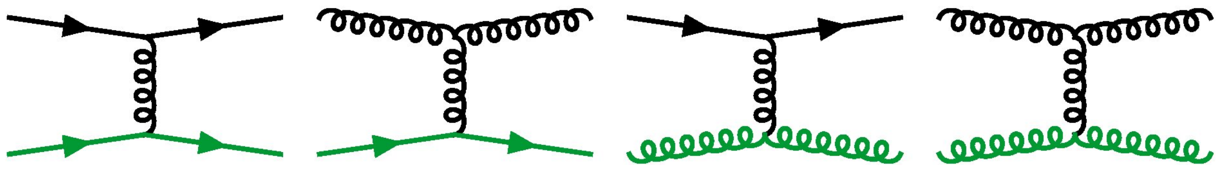

Figure 6.1: Elastic processes. Green lines indicate a thermal parton.

6.1 Total probabilities

To determine whether in one time step Δt an elastic collision occurs for a gluon or a light quark, we use the transition rate

$$\frac{d\Gamma}{d\omega}\,(E,\omega,T) = \frac{d_k}{(2\pi)^3}\frac{1}{16\,E^2}\int_0^p dq\int_{\frac{q-\omega}{2}}^{\infty} dk\,\theta(q-|\omega|)\int_0^{2\pi}\frac{d\phi_{kq|pq}}{2\pi}|\mathcal{M}|^2 f(k,T)(1\pm f(k',T))\,, \tag{6.1}$$

integrate it over ω and multiply by the size of the time step Δt. We choose a minimal $|\omega_{\min}| = 0.05\,T$, where T is the temperature. The result $\int_{-\infty}^{-\omega_{\min}} \frac{d\Gamma}{d\omega} d\omega + \int_{\omega_{\min}}^{\infty} \frac{d\Gamma}{d\omega} d\omega$ is parametrized using a fit.

We use method B as described in Ref. [SGQ09] to compute (6.1).

These total probabilities are stored in
`double Elastic::totalRate(double p, double T, double alpha_s, int Nf, int process)`
which reads `p` and `T` in units of GeV.

6.2 Sampling the energy transfer

See

```
double Elastic::findValuesRejection(double u, double T, double alpha_s, int Nf,
                                    Random *random, Import *import, int process)
```

Please not that `findValuesRejection` takes the energy `u` in units of the temperature `T` and also returns `omega` in units of the temperature.

In case `MARTINI` decides, according to the total probabilities described above, that a certain elastic process occurs, we go on and sample ω, the zeroth component of the four-momentum transfer $Q = (\omega, \mathbf{q})$. That is done using 6.1 wihout integrating it over ω. In particular, we use the rejection method to sample the tabulated and interpolated rates (see `Import` **reference here**).

For scattering with a thermal quark, we use the envelope function

$$f(\omega) = \alpha_s \frac{0.035 + 0.02\alpha_s}{0.15\sqrt{\omega^2}} \tag{6.2}$$

for both positive and negative ω. Note that it depends on α_s. For positive ω, the integral of function 6.5 from a minimal value $\omega_{\min} = 0.05\,T$ up to $\omega = y$, is given by

$$\frac{\alpha_s}{30}(7. + 4.\alpha_s)(2.99573 + \log(y))\,. \tag{6.3}$$

For negative ω, we have

$$-0.133333\alpha_s(1.75 + \alpha_s)\log(-0.0833333 * y)\,. \tag{6.4}$$

These are both saved in
`double Elastic::area (double y, double alpha_s, int posNegSwitch, int process)`,
as well as those for the processes that involve a thermal gluon and use the envelope

$$f(\omega) = 10\,\alpha_s \frac{0.035 + 0.02\alpha_s}{\sqrt{\omega^2}}\,. \tag{6.5}$$

These are for positive ω

$$0.05\,\alpha_s\,(7. + 4.\,\alpha_s)(2.99573 + \log(y)) \tag{6.6}$$

and for negative ω

$$-0.2\,\alpha_s\,(1.75 + \alpha_s)\,\log(-0.0833333\,y)\,. \tag{6.7}$$

The inverse of these integrals, e.g.

$$\tilde{\omega} = \exp\left(\frac{-1.41428\,10^9\,\alpha_s - 8.08158\,10^8\,\alpha_s^2 + 2.02327\,10^9\lambda}{\alpha_s\,(4.72097\,10^8 + 2.6977\,10^8\,\alpha_s)}\right)\,, \tag{6.8}$$

for $qq \to qq$ is the probability distribution of $\tilde{\omega}$, an auxiliary quantity. Here λ is a uniformly distributed random variable, which is maximally the full area under the envelope, here:
λ =`random->genrand64_real1()*area(u,alpha_s,posNegSwitch,process)` , where `u`= E is the energy of the jet.

Its value will be accepted, and assigned to ω, if another uniformly distributed variable, say $x \in [0, 1]$ is smaller than $\frac{d\Gamma}{d\omega}\,(E, \tilde{\omega}, T)/f(\tilde{\omega})$, where f is the used envelope function, otherwise it will be rejected and another $\tilde{\omega}$ will be chosen, and the process starts over - until an ω is accepted.

6.3 Sampling the momentum transfer

See

```
double Elastic::findValuesRejectionOmegaQ(double p, double omega, double T,
                                          double alpha_s, int Nf, Random *random,
                                          Import *import, int process)
```

Please note that `findValuesRejectionOmegaQ` takes the energy `u` and `omega` in units of the temperature `T`, and also returns `q` in units of the temperature.

To determine the transferred momentum for a given transferred energy ω, we sample the transition rate

$$\frac{d\Gamma}{d\omega\, dq}\,(E,\omega,q,T) = \frac{d_k}{(2\pi)^3}\frac{1}{16\,E^2}\int_{\frac{q-\omega}{2}}^{\infty} dk\,\theta(q-|\omega|)\int_0^{2\pi}\frac{d\phi_{kq|pq}}{2\pi}|\mathcal{M}|^2 f(k,T)(1\pm f(k',T))\,, \tag{6.9}$$

which is just (6.1) without the integral over q, at the given ω, as determined before (see Section 6.2).

Here we use two different methods to sample q. For $\omega < 6\,T$, we use the rejection method with the envelope function

$$f(q) = A\,\frac{q}{q^4+B}\,, \tag{6.10}$$

where

$$A = (0.7+\alpha_s)\,0.0012\,(1000. + 40./\sqrt{\omega^2} + 10.\omega^4)\,\alpha_s \tag{6.11}$$

$$B = 2.\sqrt{\omega^2} + 0.01 \tag{6.12}$$

for processes 1 and 2 ($qq \to qq$ and $gq \to gq$), and

$$A = (0.7+\alpha_s)\,0.0022\,(1000. + 40./\sqrt{\omega^2} + 10.\omega^4)\,\alpha_s \tag{6.13}$$

$$B = 2.\sqrt{\omega^2} + 0.002 \tag{6.14}$$

for processes 3 and 4 ($qg \to qg$ and $gg \to gg$). Note again, that here the overall normalization of (6.9) does not matter. It is only important for the total transition rate, when we determine the total probability for the process to happen. Hence, to sample q (or ω), we can neglect the factor of 9/4 for the processes involving a thermal gluon.

The area under the envelope function up to $q = y$ is given by

$$F(y) = 0.5\,A\,\frac{\arctan\left(\frac{y^2}{\sqrt{B}}\right) - \arctan(\frac{\omega^2}{\sqrt{B}})}{\sqrt{B}} \tag{6.15}$$

with A and B as above. Note that we limited the integration to $q > \omega$.

Finally, we determine invert (6.15) and get an equation for $\tilde{q}$

$$B^{0.25}\sqrt{\tan\left(\frac{2.\sqrt{B}\,x + A\,\arctan(\omega^2/\sqrt{B})}{A}\right)}\,, \tag{6.16}$$

where x = `random->genrand64_real1()*areaOmegaQ(p,omega,alpha_s,process)` is a uniformly distributed random number on [0, `areaOmegaQ(p,omega,alpha_s,process)`] $= [0, F(\omega)]$ (see Eq.6.15), where $p = E$ (the parton's energy) is the maximal value for $\tilde{q}$. Then we compare the ratio

$$r = \frac{d\Gamma}{d\omega\, dq}(E,\omega,\tilde{q},T)/f(E,\omega,\tilde{q},T)$$

to a uniformly distributed random number x and reject the value of $\tilde{q}$ if $x > r$, otherwise we accept and the final result $q = \tilde{q}$ is passed to MARTINI.

For $6 < \omega < 60\,T$ we use a different envelope

$$f(q) = A\,\exp(-0.5\,q)\,,$$

but the same principle method.

For $\omega > 60\,T$ we use Metropolis sampling.

6.4 Determining the new $\vec{p}'$

After we have sampled ω, the transferred energy, and $q = |\mathbf{q}|$, we can determine the momentum vector $\mathbf{p}'$, the final momentum of the parton after the elastic collision. This is done in

```
Vec4 Elastic::getNewMomentum(Vec4 vecpRest, double omega, double q, Random *random)
```

This function takes the old momentum vector $\mathbf{p} =$ `vecpRest` (well, the three-vector contained in `vecpRest`), ω, and q as arguments and from them finds a new $\mathbf{p}'$.

Using the angle between $\mathbf{p}$ and $\mathbf{q}$, θ_{pq}, we can separate $\mathbf{q}$ into $q_{\|}$ and $q_{\perp}$, which are the components parallel and perpendicular to $\mathbf{p}$, respectively. With

$$\cos(\theta_{pq}) = \frac{p^2 + q^2 - (p-\omega)^2}{2\,p\,q}\,, \tag{6.17}$$

we find

$$q_{\perp} = q\,\sin(\theta_{pq}) = q\,\sqrt{1-\cos^2(\theta_{pq})} \tag{6.18}$$

$$q_{\|} = q\,\cos(\theta_{pq}) \tag{6.19}$$

Then, using the cross product of $\mathbf{p}$ with $\mathbf{e}_x$ or $\mathbf{e}_y$, depending whether $\mathbf{p}$ points more into the y or the x direction, we find a vector perpendicular to $\mathbf{p}$. This vector, normalized to 1, is taken time $q_{\perp}$ and then subtracted from $\mathbf{p}$. Finally, take $\mathbf{e}_{\|} = \mathbf{p}/p$, multiply it by $q_{\|}$ and subtract it from $\mathbf{p}$ as well.

This way we find

$$\tilde{\mathbf{p}} = \mathbf{p} - \mathbf{q} = \mathbf{p} - q_{\perp}\,\mathbf{e}_{\perp} - q_{\|}\,\mathbf{e}_{\|}\,. \tag{6.20}$$

Since we want the new vector $\mathbf{p}'$ to point in a direction with arbitrary azimuthal angle around $\mathbf{p}$, we rotate $\tilde{\mathbf{p}}$ around $\mathbf{p}$ by a random angle $\phi = 2\,\pi$ `random->genrand64_real1()`. This is done using the rotation matrix

$$\mathcal{M} = \begin{pmatrix} x^2\,(1-\cos(\phi)) + \cos(\phi) & x\,y\,(1-\cos(\phi)) + z\,\sin(\phi) & x\,z\,(1-\cos(\phi)) - y\,\sin(\phi) \\ y\,x\,(1-\cos(\phi)) - z\,\sin(\phi) & y^2\,(1-\cos(\phi)) + \cos(\phi) & y\,z\,(1-\cos(\phi)) + x\,\sin(\phi) \\ z\,x\,(1-\cos(\phi)) + y\,\sin(\phi) & z\,y\,(1-\cos(\phi)) - x\,\sin(\phi) & z^2\,(1-\cos(\phi)) + \cos(\phi) \end{pmatrix}\,,$$

where $x = p_x/p$, $y = p_y/p$, and $z = p_z/p$. So finally we have

$$\mathbf{p}' = \mathcal{M}\cdot\tilde{\mathbf{p}}\,, \tag{6.21}$$

which concludes this section.

6.5 Conversion processes

We include conversion via Compton and annihilation processes. Special care has to be taken about the color indices of the partons. E.g., when a quark is converted to a gluon we have to add an aditional color line which has to be connected to another gluon (a thermal one), which again has to have its other color index connected to another thermal parton (we always use a quark for that one to end adding more connected thermal partons - that is fine because it will only change the low momentum part of the hadronization). Fig. 6.2 shows how the color lines are connected and the grey blobs are the thermal partons that are sampled from Fermi or Bose distributions at the present temperature. The momentum of the hard parton is not changed, as the momentum transfer is typically very small.

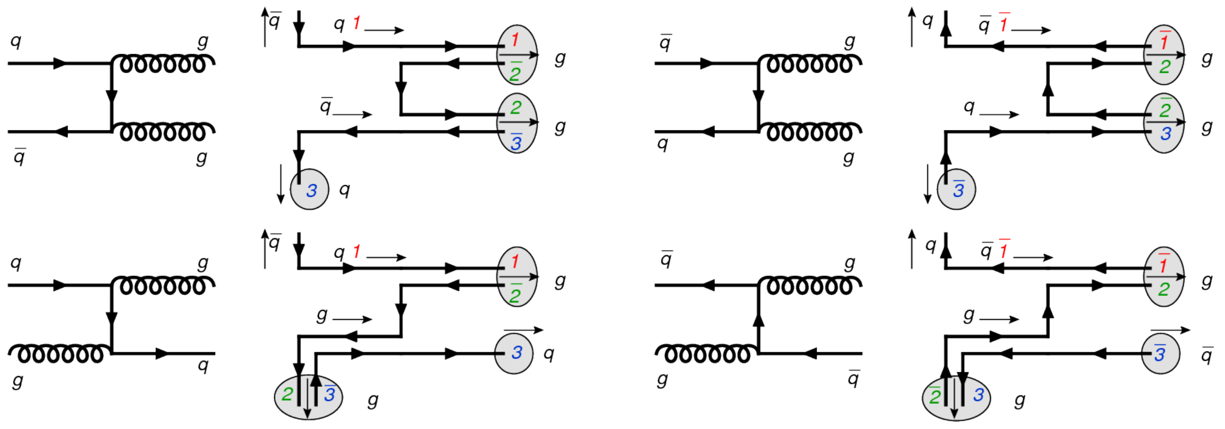

Figure 6.2: $q \to g$ and $\bar{q} \to g$ Feynman diagrams on the left and corresponding color lines on the right. The upper part of each diagram represents the hard partons, the lower part the thermal ones.

Processes that convert a gluon into a quark or anti-quark are treated analogously.

Chapter 7

Photon production

7.1 Photon conversion

Photon conversion $q \to \gamma$ is similar to the conversion processes discussed in Section 6.5. A quark or anti-quark can convert into a high energy photon via Compton or annihilation processes. All these processes are shown in Fig. 7.1. The partons in the gray blobs are sampled from Fermi or Bose distributions,

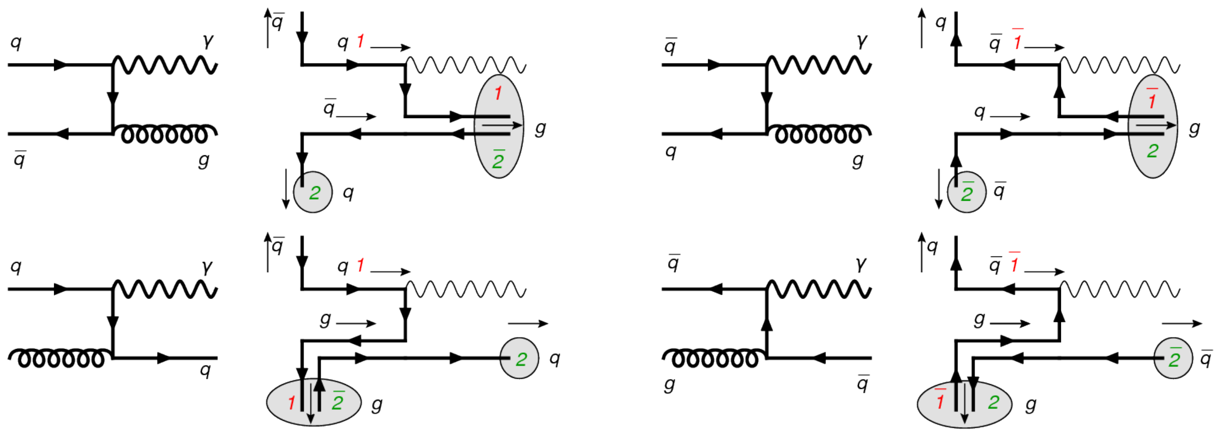

Figure 7.1: $q \to \gamma$ and $\bar{q} \to \gamma$ Feynman diagrams on the left and corresponding color lines on the right. The upper part of each diagram represents the hard partons and photons, the lower part the thermal partons.

depending on their kind.

7.2 Photon radiation

The other included process is a quark emitting a photon $q \to q\gamma$ analogous to the previously described process of gluon emission (see Section 5). Here, we do not have a complicated color structure to take care of. The quark of course keeps its color when emitting a photon. Please note that the transition rates do not include a factor of $(e_f/e)^2$, which is 4/9 for up and 1/9 for down and strange quarks. These factors are included in the main routine, when we actually know what kind of quark we have. Everything else, such as the sampling of the rate using the rejection method, works just like in the case of gluon emission.

The actual implementation is rather short because of he lack of need for color indeces. Here is what is done when photon emission occurs:

```
else if( radiatePhoton == 1 ) // see if photon emission happens
{
    // do process 4, q->qgamma
    f = rates->findValuesRejection(pRest/T, T, alpha_s, Nf, random, import, 4);

    newOne.p(f.y*T*vecp.px()/vecp.pAbs()*boostBack,  // emitted gamma's momentum
    f.y*T*vecp.py()/vecp.pAbs()*boostBack,    // since the direction does not change
    f.y*T*vecp.pz()/vecp.pAbs()*boostBack);   // I can use vecp here already

    newOne.id(22);                            // emitted parton is a photon
    newOne.mass(0.);
    newOne.frozen(1);
    newOne.x(x);                              // set the new parton's initial position
    newOne.y(y);
    newOne.z(z);
    plist[0]->push_back(newOne);              // add the photon to the list of partons
}
```

In `ReturnValue Rates::findValuesRejection(double u, double T, double alpha_s, int Nf, Random *random, Import *import, int process)` process number 4 takes care of photon emission. See Section 5.1.1 for details.

Chapter 8

Fragmentation

The alternative fragmentation mechanism does not need MARTINI to keep track of the color structure of the shower. Instead, we connect color strings in the end, when a parton exits the QGP phase or its evolution stops. It is more likely that a high momentum parton connects to a close-by medium parton than a jet parton which went in the other direction initially. Assuming that the elastic collisions in the plasma randomize the color structure anyway, this should be a more realistic description.

So once a parton is done with its evolution we go into its cell and determine all its neighbours, be it jets or medium partons. Of course, since the medium is described by hydrodynamics there are no individual medium partons in the code yet. So we add them to the cell. How many do we add? That is determined solely by the temperature. The density of gluons and quarks per degree of freedom are given by

$$n_g^1(T) = \frac{4\pi}{(2\pi)^3}\int_0^\infty p^2 \frac{1}{\exp(p/T)-1}dp = \frac{\zeta(3)}{\pi^2}T^3\,, \tag{8.1}$$

$$n_{u,d,s,\bar{u},\bar{d},\bar{s}}^1(T) = \frac{4\pi}{(2\pi)^3}\int_0^\infty p^2 \frac{1}{\exp(p/T)+1}dp = \frac{3\zeta(3)}{4\pi^2}T^3\,. \tag{8.2}$$

Now, quarks and anti-quarks have $2\,N_c$ degrees of freedom per flavor, gluons have $2\,(N_c^2-1)$, so that the densities are

$$n_g(T) = \frac{16\,\zeta(3)}{\pi^2}T^3\,, \tag{8.3}$$

$$n_{u,d,s,\bar{u},\bar{d},\bar{s}}(T) = \frac{9\,\zeta(3)}{2\pi^2}T^3\,, \tag{8.4}$$

for $N_c = 3$. The number of medium partons that we add per cell is then given by

$$N_g(T) = V\,n_g(T)\,, \tag{8.5}$$

$$N_{u,d,s,\bar{u},\bar{d},\bar{s}}(T) = V\,n_{u,d,s,\bar{u},\bar{d},\bar{s}}(T)\,, \tag{8.6}$$

where V is the volume of that cell. We should take the size of a hydro cell with constant temperature. We could also allow partons from neighboring cells to connect to the jet parton - maybe that is a good test.

We can now sample the number from some probability distribution with average N or just add the exact number per cell. I'd say for now it is sufficient not to sample. Then we sample the position within the cell (from a uniform distribution) and the momentum from the corresponding thermal distribution. Note that the momentum of the medium parton has then to be boosted into the 'lab-frame'.

Once the medium partons are added we can start connecting strings. we have to make sure that the group of partons that we submit to the PYTHIA fragmentation routine is overall color neutral, i.e. that all strings are connected.

So we start with one of the jet partons in the cell and randomly select a partner for it (all other partons in the cell, medium or jet get the same probability).

If the jet parton is a quark, we can connect its string to either a gluon or an anti-quark. If we happen to choose an anti-quark, we are done with this jet parton. If we choose a gluon, then we also have to connect the gluon to either another gluon or an anti-quark. So, choosing gluons demands continuing more strings.

The same happens if the jet parton was an anti-quark. Just that the chain has to end with a quark now.

If the first jet parton is a gluon, we need to connect it to both a quark and anti-quark and possibly intermediate gluons.

When this chain is terminated, we move on to the next jet parton that hasn't been connected and do the same. If we are left with a jet parton but no possible partners, we sample one (or two if it is a gluon) more medium parton(s) (randomly u,d, or s (anti-)quark), connect it/them to it and finish.

To do all this effecitvely, we should first sort all jet partons into groups determined by their position, i.e., by the cell they are in. Then we have N_{cells} sub-lists for which we can do the above procedure.

Chapter 9

Post processing

9.1 When using the original main.cpp

At this point in time (svn revision 18, 2010/02/12), the output files of one set of runs are:

distances.dat	distribution of travelled distances [fm] of quarks and gluons with errors
distancesGluonPhi.dat	as above for gluons, colums are phi bins=0-15 15-30 30-45 45-60 60-75 75-90
distancesQuarkPhi.dat	same for quarks
gluonphi.dat	distribution of gluons vs. p_T [GeV] in different phi bins with errors
quarkphi.dat	same for quarks
partons.dat	distributions of quarks, gluons vs. p_T (no error)
pi0.dat	p_T distribution of pi0's with error
pi0phi.dat	p_T distribution of pi0's in phi bins with error
pi+.dat	p_T distribution of pi+'s with error
pi-.dat	p_T distribution of pi-'s with error
hplus.dat	p_T distribution of positive hadrons with error
hminus.dat	p_T distribution of negative hadrons with error
protons.dat	p_T distribution of protons with error
anti-protons.dat	p_T distribution of anti-protons with error
photons.dat	p_T distribution of all photons with error
K0.dat	p_T distribution of K_0 with error
K0S.dat	p_T distribution of K_0 short with error
K0L.dat	p_T distribution of K_0 long with error
K+.dat	p_T distribution of K_+ with error
K-.dat	p_T distribution of K_- with error
electrons.dat	p_T distribution of e- with error
positrons.dat	p_T distribution of e+ with error
mu.dat	p_T distribution of mu- with error
muplus.dat	p_T distribution of mu+ with error
rho0.dat	p_T distribution of rho_0 with error
omega.dat	p_T distribution of omega with error
eta.dat	p_T distribution of eta meson with error
phi.dat	p_T distribution of phi meson with error
JPsi.dat	p_T distribution of J/Psi with error
theta.dat	binning of quarks in theta=atan(pt/pl) bins
x.dat	initial positions of all partons in x [fm]
y.dat	initial positions of all partons in y [fm]
xy.dat	same in y,x (in that order) and the initial momentum [GeV] in x-direction
r.dat	initial distance from x=y=0 [fm]
test.dat	shows random seed and progress during the run (not very important)

There are several c-shell scripts to extract important information from these files. Note that what I called error above is actually just the sum of squares of entries from individual runs. It is used to compute the standard deviation in post-processing.

Now about the scripts:

- `makeplot.csh`: run `./makeplot.csh N`, where `N` is the number of "sub-runs" to combine into the final result. So if you had started 10 actual runs with say 10000 events each, you will find sub-folders `1` to `10` have been created. Now running `./makeplot.csh 10` will combine these 10, compute the standard deviation and write the output (p_T spectrum of π^0s with standard deviation) into the file `pi0.dat`. This will be the average over all 100000 events. Using a number smaller than ten will only include the folders up to that number. Will also generate `pijoined` using the result for pp collisions stored in pi0-pp.dat that you have to copy into the current folder before. This is just the `pi0.dat` that you obtained the same way in a pp run and renamed and copied it here. `pijoined` has p_T in GeV in the first column, then the pp result and the error in the second and third respectively, then p_T in GeV again (column 4 = column 1 - make sure that that is true, otherwise the binning in the pp and the AA calculation was different), and then the columns AA result as in `pi0.dat`.

  Needs `template_plot.csh` and `add.awk` and will create `iplot.csh` and execute it.

- `makeDistanceplot.csh`: same principle. no errors. output: `distances.dat`. columns: distance in [fm], quark distribution, gluon distribution

  Needs `template_DistancePlot.csh` and `addDistances.awk`, creates `iDistancePlot.csh` and runs it.

- `makeDistancephiplot.csh`: same principle. no errors. output: `distancesQuarkPhi.dat`. columns: distance in [fm], quark distribution in the different phi bins.

  Needs `template_Distancephiplot.csh` and `addDistancePhi.awk`, creates `iDistancephiplot.csh` and runs it.

- `makeDistancephiplotGluon.csh`: same principle. no errors. output: `distancesGluonPhi.dat`. columns: distance in [fm], gluon distribution in the different phi bins.

  Needs `template_DistancephiplotGluon.csh` and `addDistancePhi.awk`, creates `iDistancephiplotGluon.csh` and runs it.

- `makeKaonPlot.csh`: like `makeplot.csh` but for K^+. creates `K+.dat`, and `kaonjoined` if `K+-pp.dat` is there (same as with `makeplot.csh`).

  Needs `template_kaon_plot.csh` and `add.awk` and will create `ikaonplot.csh` and execute it.

- `makeK-Plot.csh`: like `makeplot.csh` but for K^-. creates `K-.dat`, and `K-joined` if `K--pp.dat` is there (same as with `makeplot.csh`).

  Needs `template_K-_plot.csh` and `add.awk` and will create `iK-plot.csh` and execute it.

- `makephiplot.csh`: same principle. Will combine the results for π^0 in phi bins. creates `pi0phi.dat`. Will also generate `piphijoined` using the result for pp collisions stored in pi0-pp.dat that you have to copy into the current folder before. `piphijoined` has p_T in GeV in the first column, then the pp result and the error in the second and third respectively, then p_T in GeV again (column 4 = column 1 - make sure that that is true, otherwise the binning in the pp and the AA calculation was different), and then the columns with the different phi-bins as in `pi0phi.dat`.

  Needs `template_phiplot.csh` and `addphi.awk` and will create `iphiplot.csh` and execute it. Includes columns with standard deviation.

- `makephotonplot.csh`: like `makeplot.csh` but for photons. creates `photons.dat`. Will also generate `photonsjoined` using the result for pp collisions stored in photons-pp.dat that you have to copy into the current folder before. `photonsjoined` has p_T in GeV in the first column, then the pp result and the error in the second and third respectively, then p_T in GeV again (column 4 = column 1 - make sure that that is true, otherwise the binning in the pp and the AA calculation was different), and then the columns AA result as in `photons.dat`.

  Needs `template_photon_plot.csh` and `add.awk` and will create `iphotonplot.csh` and execute it.

- `makePi+Plot.csh`: like `makeplot.csh` but for π^+. creates `pi+.dat`, and `pi+joined` if `pi+-pp.dat` is there (same as with `makeplot.csh`).

  Needs `template_pi+_plot.csh` and `add.awk` and will create `ipi+plot.csh` and execute it.

These are all routines so far. Now let me give an example on how to plot the results: This will plot the R_{AA} for π^0 for two different impact parameters using `gnuplot`

```
set term post eps enhanced color font "Helvetica,26"
set output 'raa.eps'

set key top left Left
set key reverse
set key box
set key width -13
set key spacing 1.2
set xlabel 'p_T [GeV]'
set xrange [0:14]
set yrange [0.0000000001:1]
set style fill transparent solid 0.3 noborder

set size 1,1.3
set multiplot
set size 1,0.7
set ytics (0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9)
plot\
 '../data/phenix-prl-pi0-raa-central.txt' u 1:2:5 w e t
 '{/Helvetica=22 PHENIX Au+Au 200 GeV, 0-10 % central}' pt 7 ps 2 lw 3 lt -1 lc -1,\
'pijoinednonaka17r039ptmin4.dat' every ::7::34 u ($1):($5/$2):(0.25):(($6/$2+$3*$5/($2*$2)))
notitle w boxxyerrorbars lw 3 lt -1 lc rgbcolor "#44AA44",\
 'pijoinednonaka17r039ptmin4.dat'  every ::7::35 u ($1-0.25):($5/$2)
 t '{/Helvetica=22 AMY + 3+1D Hydro {/Symbol a}_s=0.39 b=2.4 fm}'
 w steps lw 6 lt 2 lc rgbcolor "#117711",\
'pijoinednonaka17el03ptmin2.dat'  every ::7::11 u ($1):($5/$2):(0.25):(($6/$2+$3*$5/($2*$2)))
notitle w boxxyerrorbars lw 3 lt -1 lc rgbcolor "#BB4444",\
 'pijoinednonaka17el03ptmin2.dat'  every ::7::12 u ($1-0.25):($5/$2)
 t '{/Helvetica=22 AMY + elastic + 3+1D Hydro {/Symbol a}_s=0.3 b=2.4 fm}'
 w steps lw 3 lt -1 lc rgbcolor "#BB1111",\
'pijoinednonaka17el03ptmin4.dat'  every ::12::34 u ($1):($5/$2):(0.25):(($6/$2+$3*$5/($2*$2)))
notitle w boxxyerrorbars lw 3 lt -1 lc rgbcolor "#BB4444",\
 'pijoinednonaka17el03ptmin4.dat'  every ::12::35 u ($1-0.25):($5/$2)
 t '' w steps lw 3 lt -1 lc rgbcolor "#BB1111",\
 'pijoined03ptmin4nucb2.4-av'  every ::12::35 u ($1-0.25):($5/$2)
 t '' w steps lw 3 lt -1 lc rgbcolor "#BB11BB",\
 '../data/phenix-prl-pi0-raa-central.txt' u 1:2:5 w e notitle pt 7 ps 2 lw 3 lt -1 lc -1

unset xlabel
set label '{/Symbol p}^0 R_{AA}' at graph -0.13, graph -0.15 rotate left
unset xtics
set size 1,0.6
set origin 0,0.648
set ytics (0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1)
plot\
 '../data/phenix-prl-pi0-raa-20-30.txt' u 1:2:5 w e
 t '{/Helvetica=22 PHENIX Au+Au 200 GeV, 20-30% central}' pt 7 ps 2 lw 3 lt -1 lc -1,\
'pijoinednonaka17r039b7.5ptmin4.dat'  every ::7::34 u ($1):($5/$2):(0.25):(($6/$2+$3*$5/($2*$2)))
notitle w boxxyerrorbars lw 3 lt -1 lc rgbcolor "#44AA44",\
'pijoinednonaka17r039b7.5ptmin4.dat'  every ::7::35 u ($1-0.25):($5/$2)
t '{/Helvetica=22 AMY + 3+1D Hydro {/Symbol a}_s=0.39 b=7.5 fm}'
w steps lw 6 lt 2 lc rgbcolor "#117711",\
'pijoinednonaka17el03b7.5ptmin4.dat'  every ::7::34 u ($1):($5/$2):(0.25):(($6/$2+$3*$5/($2*$2)))
notitle w boxxyerrorbars lw 3 lt -1 lc rgbcolor "#BB4444",\
 'pijoinednonaka17el03b7.5ptmin4.dat'  every ::7::35 u ($1-0.25):($5/$2)
 t '{/Helvetica=22 AMY + elastic + 3+1D Hydro {/Symbol a}_s=0.3 b=7.5 fm}'
w steps lw 3 lt -1 lc rgbcolor "#BB1111",\
 '../data/phenix-prl-pi0-raa-20-30.txt' u 1:2:5 w e notitle pt 7 ps 2 lw 3 lt -1 lc -1
unset multiplot
pause -1
```

Note that all lines should be in one up to the "`,\`". I only write it this way to fit on a page. The files beginning with `pijoined` are just the `pijoined` that we got as an output and then renamed to indicate which run they are from. File beginning with `phenix` are experimental data.

Appendix

Bibliography

[NB07] Chiho Nonaka and Steffen A. Bass. Space-time evolution of bulk QCD matter. *Phys. Rev.*, C75:014902, 2007.

[SGQ09] Bjorn Schenke, Charles Gale, and Guang-You Qin. The evolving distribution of hard partons traversing a hot strongly interacting plasma. *arXiv:0901.3498*, 2009.

Index

AMY, 9

background, 2, 7

Elastic collisions, 12
    energy transfer, 13
    momentum transfer, 14

Hydro data structure, 4
Hydrodynamics, 2
    2+1D Hydro
        Eskola, 5
        Kolb, 4
    3+1D Hydro
        Nonaka, 6

Radiative loss, 9