Documentation for the ICCING Code

Patrick Carzon,[1,*] Mauricio Martinez,[2,†] Matthew D. Sievert,[3,‡]
Douglas E. Wertepny,[4,§] and Jacquelyn Noronha-Hostler[1,5,¶]

[1]*Illinois Center for Advanced Studies of the Universe, Department of Physics, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*

[2]*North Carolina State University, Raleigh, NC 27695, USA*

[3]*Department of Physics, New Mexico State University, Las Cruces, NM 88003, USA*

[4]*Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel*

[5]*Rutgers University, Piscataway, NJ 08854, USA*

(Dated: January 14, 2022)

*Email: pcarzon2@illinois.edu
†Email: mmarti11@ncsu.edu
‡Email: msievert@nmsu.edu
§Email: wertepny@post.bgu.ac.il
¶Email: jnorhos@illinois.edu

Contents

I. USER MANUAL

A. Requirements

Externally, ICCING only requires C++ Libraries and a compiler to run. This version uses gcc (7.2.0) and cmake (3.18.4). Other requirements are, a correctly formatted configuration file and initial conditions provided by some external source which must contain a saturation scale and a collision profile.

B. Input Parameters

A configuration file (.conf) is required to run ICCING. Below are listed the options that can be set in the configuration file with brief descriptions. The options can be entered in the configuration file in any order. Some parameters have default values that will be assigned if not specified.

External Parameters	**Description**
a_trento (Def. 119)	Conversion factor between T_R and entropy density (fm)
grid_max (Def. 12)	Absolute value of grid limits
grid_step (Def. 0.06)	Size of distance between grid points

TABLE I: Table of parameters that ICCING requires as input from initial state code. 'a_trento' is a conversion factor between the reduced thickness function and the entropy density.

C. Configuration File

In Table IV, a sample configuration file is provided. The order in which options are specified does not matter. If you mistype an input parameter keyword or specify an unknown parameter, the code will exit and throw an error message.

ICCING Parameters	**Description**
trento_input_dir	Directory for event initial condition input
quark_input_dir	File path for quark chemistry
eos_file	File path for equation of state
output_dir	Directory for output files
input_type (Def. 1)	Flag to set input type (0=full densities, 1= sparse densities)
output_type (Def. 1)	Flag to set output type (0=full densities, 1= sparse densities)
seed_ (Def. random seed)	Used to set static seed
test_ (Def. empty string)	Flag for testing code (See Tests section)
first_event (Def. 0)	First event to read from input
last_event (Def. 100)	Last event to read from input
t_a (Def. 0)	Flag to read in T_A (0=no, 1=yes)
t_b (Def. 1)	Flag to read in T_B (0=no, 1=yes)

TABLE II: General parameters for the ICCING code.

ICCING Parameters	**Description**
kappa_ (Def. 1)	Proportionality factor between Q_s and $\sqrt{T_B}$
rad_ (Def. 0.5 fm)	Radius of gluon
qrad_ (Def. 0.5 fm)	Radius of quarks
lambda_ (Def. 1)	Power of energy dependent gluon probability
dipole_model (Def. GBW)	Flag for correlation function: Unspecified/"GBW" = GBW Model "MV" = MV Model
alpha_s (Def. 0.3)	Coupling constant
alpha_min (Def. 0.01)	Cutoff on quark energy splitting fraction
r_max (Def. 1)	IR cutoff on $q\bar{q}$ seperation
lambda_bym (Def. 0.001)	Used by MV Model
e_chop (Def. 10E-20)	Tolerance to round small energy values to zero (in GeV)
tau_0 (Def. 0.6)	Initialization time for sampling (fm)
e_thresh (Def. 0.25)	Gluon threshold energy
charge_type (Def. BSQ)	Type of charge tracked: "BSQ" = Baryon, Strange, Charge "UDS" = Up, Down, Strange

TABLE III: Physical parameters for the ICCING algorithm.

```
trento_input_dir /home/trento_output/
output_dir /home/iccing_output/
eos_file /home/equation_of_state/eos.dat
quark_input_file /home/flavor_chemistry/GBW_Chemistry.dat
input_type 1
output_type 2
t_a 0
t_b 1
first_event 0
last_event 100
grid_max 12
grid_step 0.06
lambda_ 1
rad_ 0.5
qrad_ 0.5
```

TABLE IV: ICCING configuration file example. Options can be specified in any order and can be separated by white space.

D. How to Build and Run

To build ICCING code run command:

make iccing

To run ICCING use this command:

./iccing /PathToConfigurationFile.conf

E. Output

The configuration parameter output_type determines the form of the output from ICCING. Independent of output_type, eccentricities and quark counts are always printed to file. A value of 0 for output_type will print out the full density grids including filler 0s whereas a value of 1 will print out sparse density grids: x-value, y-value, and density values.

A configuration file is also output to the same location with all of the input parameters. This ensures that any data from ICCING is paired with the parameters that were used to create it.

densities.dat (sparse density)					
x	y	energy	baryon	strangeness	charge

TABLE V:

density_eccentricities.dat										
event #	entropy	ε_2	ϕ_2	ε_3	ϕ_3	ε_4	ϕ_4	ε_5	ϕ_5	radius

TABLE VI:

quark_counts.dat				
event #	up	down	strange	charm

TABLE VII:

F. Tests

The ICCING code has several built in tests that were used to check key parts of the algorithm. They have been left in the open source version for use as checks on personal changes to the code. The tests mainly focus on physical inputs to the code and may not give insight into general issues or bugs that may occur from editing. Below is a list of all tests implimented in the ICCING code along with examples of output generated by them.

1. SChop

Used to test the effect that the implemented entropy trimming has on the energy density without the interference from the splitting algorithm.

2. GluonEnergyDist

This test is used to check the gluon energy fraction probability. To run the test, use the 'test_"GluonEnergyDist"' command in the configuration file. This test is implemented in the Splitter::RollGlue function and modifies the code to run through this function 10,000 times while sampling the gluon energy fraction probability. The data from this test is output in a file called "gluon_energy_dist_test.dat" in the specified output_folder from the configuration file. This data can be used to reproduce the curves in Fig. **??** and check that this function is working correctly. After running this test, the code will exit and not complete the rest of the algorithm

3. CorrelationFunction

This test is used to check the correlation function used by the algorithm to select the separation distance of quark pairs and their relative momentum fraction. To run the test use the 'test_"CorrelationFunction"' command in the configuration file. This test is implemented in the Splitter::RollLocation function and modifies the code to run through this function 10,000 times while sampling the correlation function. The input, quark mass and saturation scale, to this function is specified by the test and must be changed and the code recompiled to run under different conditions. The data from this test is output in a file

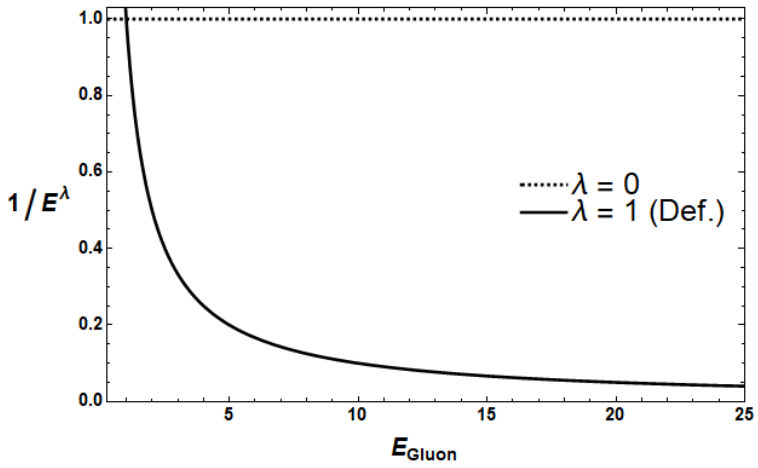

FIG. 1: Illustration of the function used to sample the gluon energy.

called "correlation_function_test.dat" in the specified output_folder from the configuration file. This data can be used to reproduce the curves in Fig. 2 and check that this function is working correctly. After running this test, the code will exit and not complete the rest of the algorithm

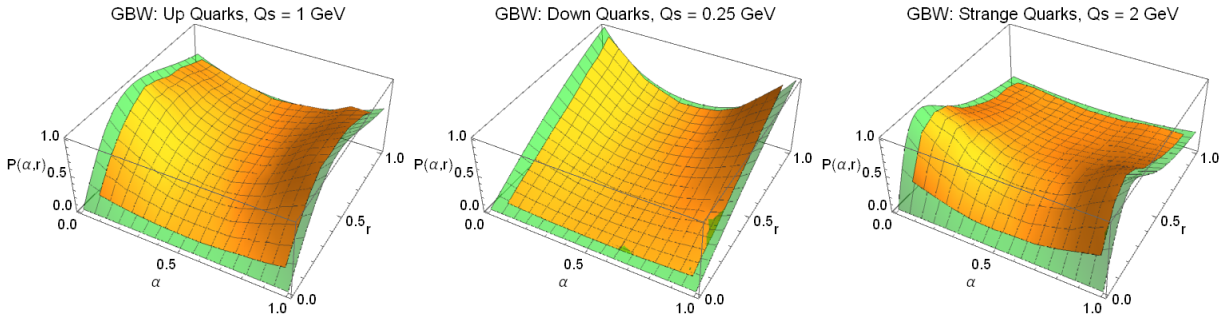

FIG. 2: Plots of the input and output of the Monte Carlo sampling, validating that it successfully reproduces the theoretical probability distribution for the GBW model. We have similarly checked that the procedure correctly reproduces the theoretical distribution for the MV model.

4. QuarkRatio

This test is used in main.cpp to output quark masses with their corresponding q_s. This data can be used to reconstruct an approximation to the probability to produce quarks of different flavors across q_s.

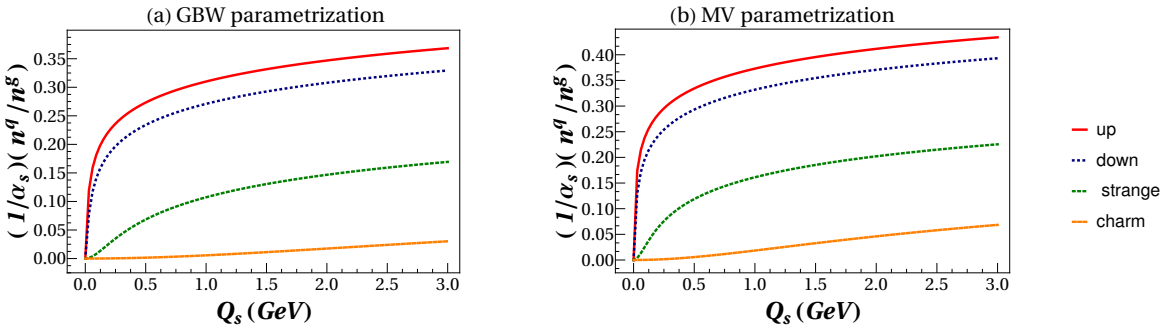

FIG. 3: Quark/gluon multiplicity ratios as a function of the target saturation scale Q_s for various flavors and different dipole amplitude approximations. In this case we used the GBW (left panel) and MV (right panel) models. Here the cutoff in the MV model has been taken to be $\Lambda/m = 0.0241$.

II. FLOW CHART

A. Global View

In Fig. 4 is a flowchart that provides a high level view of the ICCING algorithm. These are the general steps that the algorithm carries out along with the input and output.

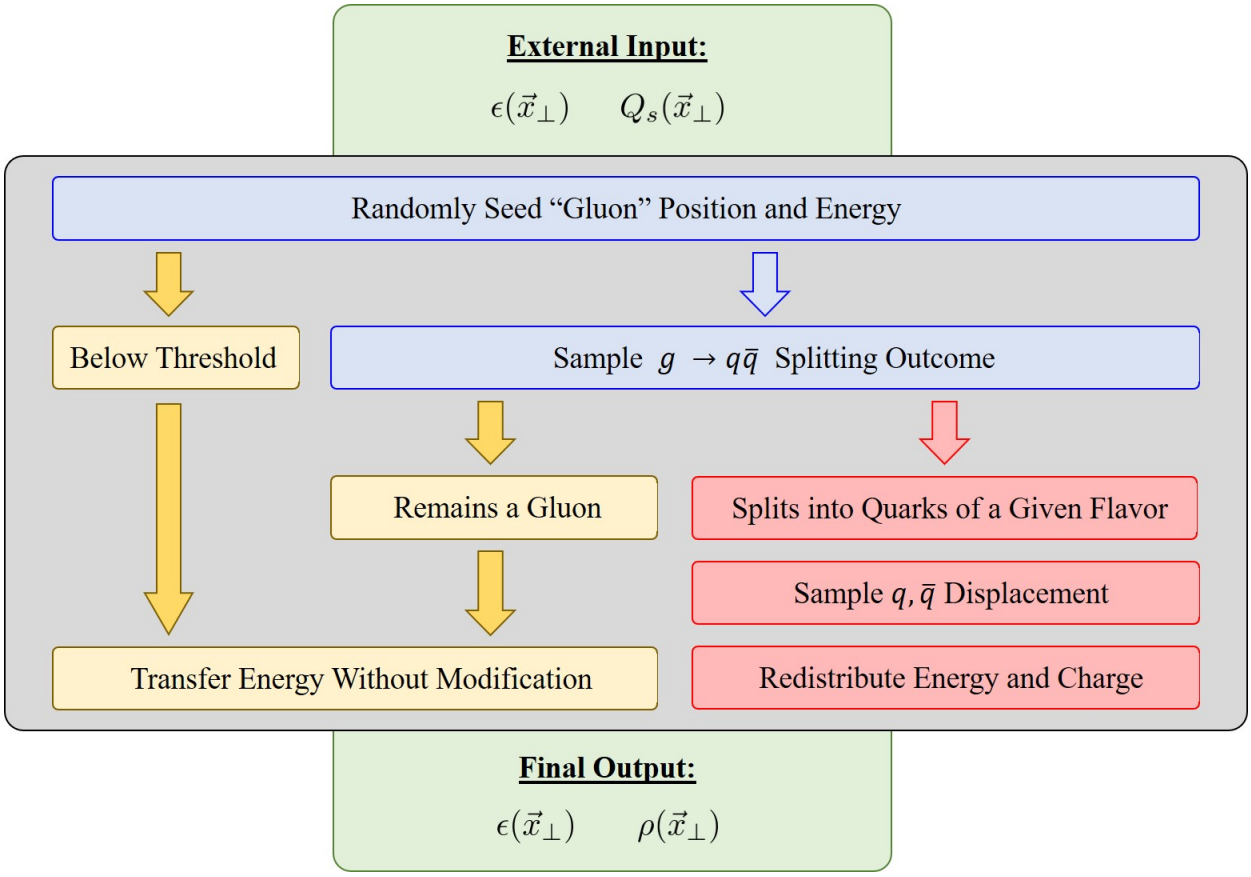

FIG. 4: Decision tree of the ICCING algorithm.

B. Main Loop

In Figs. 5 and 6, a more detailed flowchart of the ICCING program is provided with specific reference to function calls. The different steps are color coded to where they are implemented in the code. Dark blue cells are implemented in main.cpp, light blue in io.h, green in event.h, and red in splitter.h. More information about each of the functions used here can be found in Section III. Expanded flowcharts for functions specifically used for splitting are provided in Section II C.

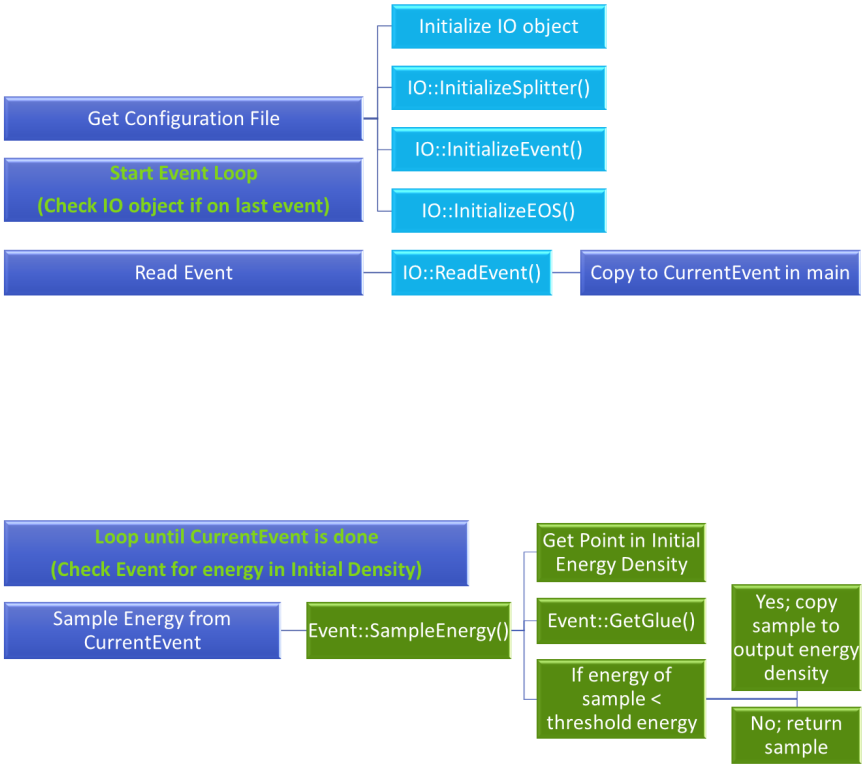

FIG. 5: Detailed flowchart for ICCING algorithm (Part 1). Details the initialization of the code, event input, and beginning of main loop which includes sampling of energy from input.

FIG. 6: Detailed flowchart for ICCING algorithm (Part 2). Details the splitting of gluons and the completion of the main loop.

C. Splitting Functions

Figs. 7 and 8, display flowcharts detailing the splitting functions used by ICCING. Cells are color coded to where they appear in the code with red cells being in splitter.h, brown in functions.h, and purple in correlation.h. Further details about functions mentioned here can be found in Section III.

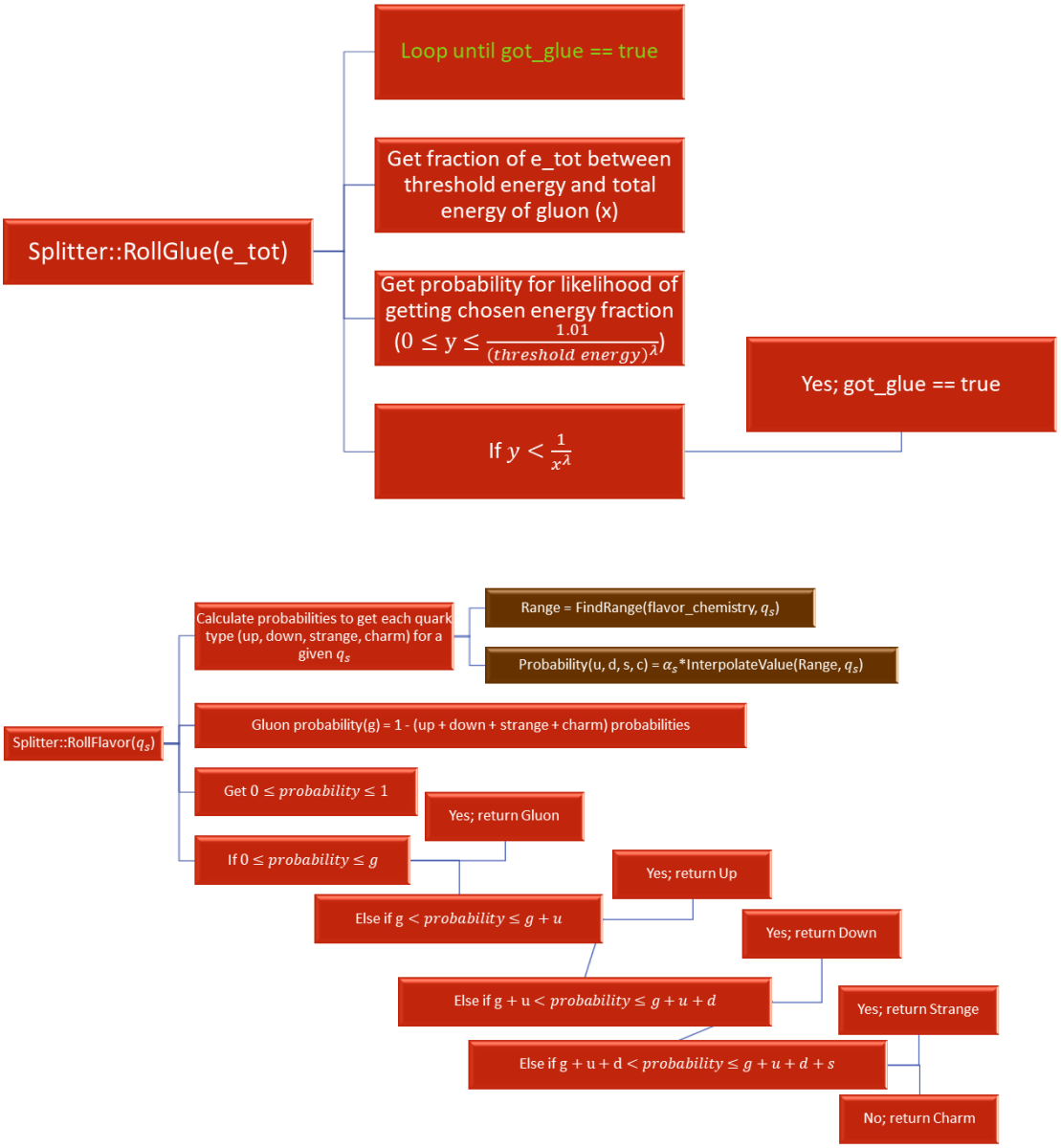

FIG. 7: Detailed flowchart for ICCING algorithm (Part 3). Details the selection of gluon energy and of quark flavor.

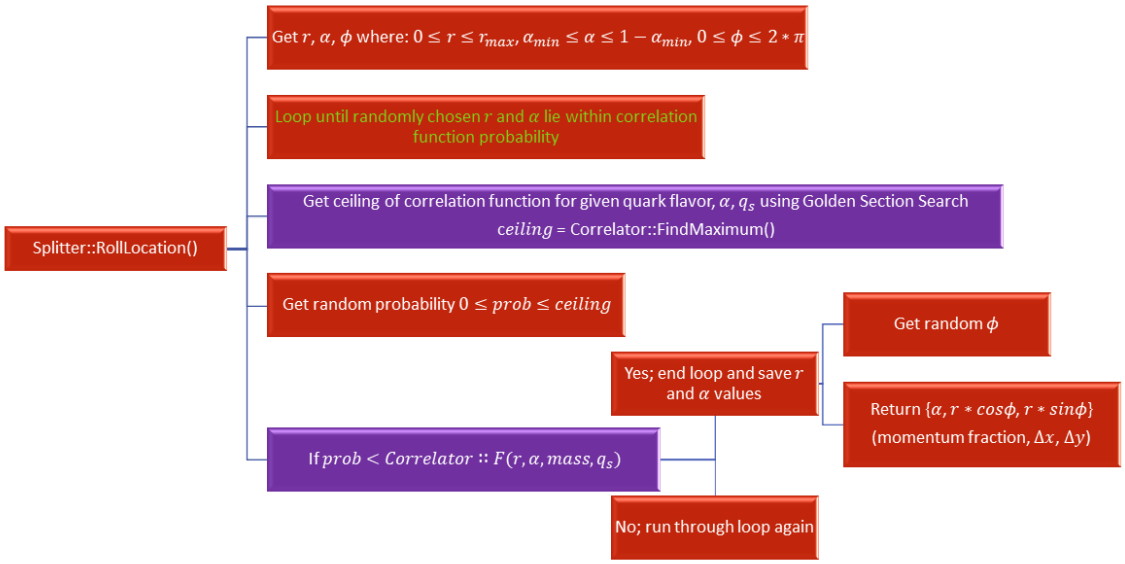

FIG. 8: Detailed flowchart for ICCING algorithm (Part 4). Details the sampling of the correlation function to get position and momentum fraction for quark pair.

III. CLASS STRUCTURE AND FUNCTION DESCRIPTION

A. Event

- Sample GetGlue()
  - Select energy of gluon. Find the total energy available within a circle of the gluon radius centered around the Event global variables x_center and y_center, and calculate the Q_s within that same circle. Return a Sample object containing the available energy and Q_s.
  - Called by Event::SampleEnergy()
  - Calls Event::GetIntegrationBounds()
- vector ⟨int⟩ GetIntegrationBounds(int size, double raduis, int x, int y)
  - Gets integration bounds for density grid manipulations. This function checks the size of a mask against the full density grid to ensure points out of bounds are not accessed. This is used, specifically, to ensure that when depositing quarks they do not go out of bounds. If quarks do go out of bounds, program exits and prints error message.
  - Called by Event::GetGlue(), Event::UpdateEnergy(), and Event::UpdateDensity
- void UpdateEnergy(double ratio)
  - Subtracts energy from initial_energy and adds it to density[0]. Move energy in circle with gluon radius from initial energy density to output energy density. This is done proportionally by multiplying the gluon mask by the provided ratio. This ensures that all geometry from the initial state is carried through. This is visualized in Fig. 9.
  - Calls Event::GetIntegrationBounds()
- Sample SampleEnergy()
  - Sample Initial Energy for ICCING algorithm. Select random valued point from initial energy density to act as the center of a new gluon. If the energy available

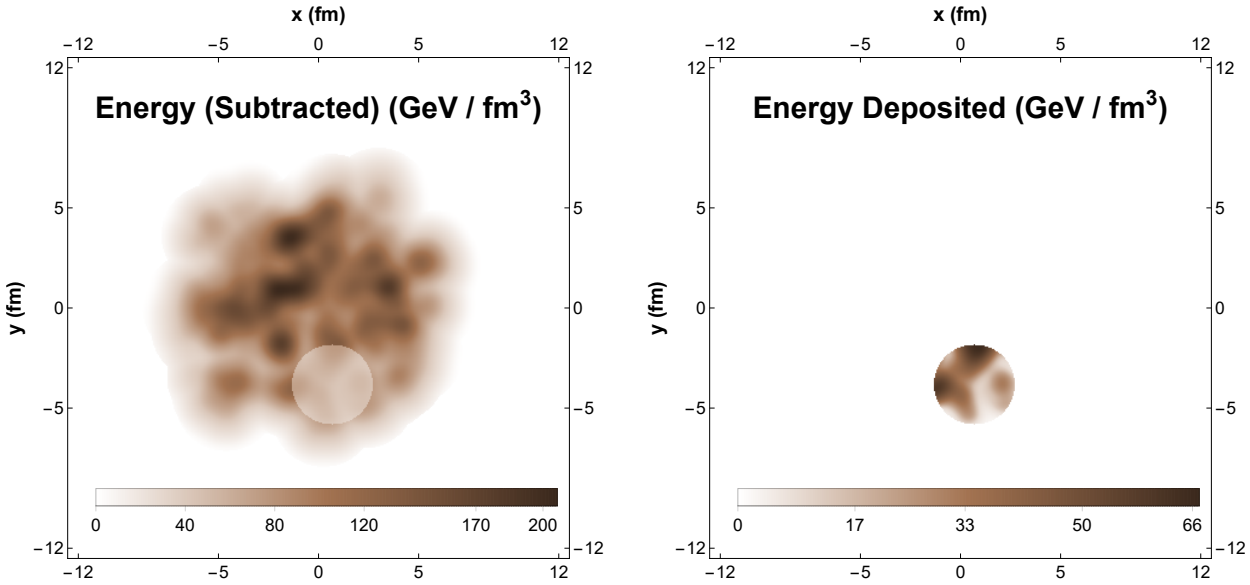

FIG. 9: Illustration of how the ICCING algorithm transfers energy when the gluon *does not* split into a $q\bar{q}$ pair. The energy is deducted from the input grid (left) and deposited in the output grid (right) as shown. The energy transfer is done point by point and proportionately to the total enclosed energy. As a result, the transferred energy retains the underlying geometric structure of the original energy density, as seen in both the input energy grid after subtraction and the output energy grid after deposition. The gluon radius here has been greatly increased to clearly show these details.

for this gluon is less than the threshold energy. e_thresh, then copy all of the energy over to the output density since there is not enough to split into a quark pair. Further check to see if the total energy left in the initial energy density is below e_thresh, if it is then copy all of the remaining energy over to the output since no new quarks can be made. In these cases, signal the program to finish this sample loop and search for a new quark splitting.

- Calls Event::GetGlue(), Event::UpdateEnergy()

- bool UpdateDensity(Quarks quark_density)

  - Propagates Results of Splitter. This function takes a quark object and subtracts its energy from the initial energy density and adds it and the appropriate charge

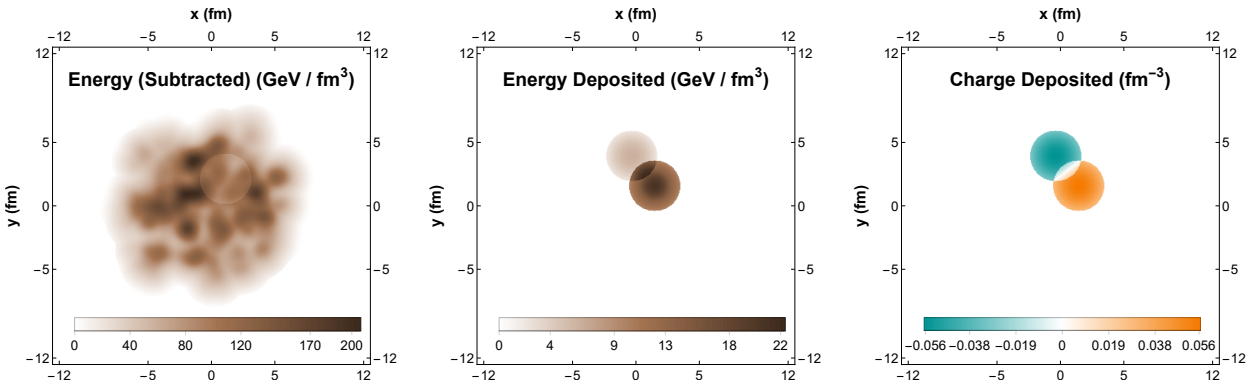

FIG. 10: Illustration of how the ICCING algorithm transfers energy when the gluon *does* split into a $q\bar{q}$ pair. The energy is deducted from the input grid (left plot) proportionately, preserving the underlying geometry in the input grid. But it is deposited in Gaussian blobs for the $q\bar{q}$ pair, which are displaced relative to the original gluon position. This modifies the energy density distribution (center plot) and also leads to a net displacement of positive and negative charge (here baryon density, right plot). Note that the energy is in general shared unequally between the quark and antiquark; in this case, the quark carried about 75% of the original gluon energy, as visible in the center plot. Here both the radii and overall $q\bar{q}$ displacement have been greatly increased to clearly show these details.

distributions to the output density grids. If the quark object is just a gluon then the energy is moved to the output. If the quark object is a quark pair, then the centers of the quark and anti-quark are calculated, a test is made to ensure that they lie in the bounds of the output grid, and the appropriate densities are propagated to the output density grids using the quark mask. This is visualized in Fig. 11.

- Calls Event::UpdateEnergy(), Event::GetIntegrationBounds(), and All functions of Quarks struct

- void CalculateEccentricities()
  - Calculate Eccentricities from density grids. This function calls the CalculateEccentricities function from the Eccentricity class and passes it the full density grids for the calculation of eccentricities for the energy and charge distributions.

- Calls Eccentricity::CalculateEccentricities()

- void CleanEvent()

  - Clears Event variables as a cautionary measure. Set important event specific variables to zero and clear grids.

- bool IsEventDone()

  - Check list of valued points to see if all energy has been processed and copied to the output. If there is no energy left in the initial energy density then return true and finish processing the event. This function contains the SChop test which copies all energy immediately to the output and is used to test the entropy trimming procedure.

B. Splitter

- double RollGlue(double e_tot)

  - Select energy of gluon. Sample power law distribution from Fig. 1 to get the proportion of the total energy that will be the gluon's energy. Contains GluonEnergyDist test which runs through this process 10,000 times and outputs the results to a file for use in testing this function and reproducing Fig. 1.

  - Called by Splitter::SplitSample()

- Charge RollFlavor(double Qs)

  - Select flavor of gluon. For a given Q_s, find the splitting probability for each quark flavor using the interpolation function created using the quark chemistry file. Sample a random number between 0 and 1 then compare to the different probabilities to determine the flavor of the quark splitting. Return the relevant charge structure.

  - Called by Splitter::SplitSample()

- vector⟨double⟩ RollLocation(double mass, double Qs)

- Sample location and momentum fraction of quark pair. Using quark mass and Q_s, sample correlation function to find relative location and momentum fraction of quarks. Start by finding the ceiling of the correlation function at these parameters. Then sample random values for r, alpha, and the probability. Compare the correlation function using the random r and alpha and the specified mass and Q_s to the sampled probability and if the probability is lower than the correlation function use the r value as the separation distance between the quark and anti-quark and alpha as the momentum fraction. This function also contains a test that sets the mass and Q_s and runs the loop over the sampling 100,000 times to test the correlation function and reproduce the plot in Fig. 2.
- Called by Splitter::SplitSample()

• Quarks SplitSample(Sample sampled_energy)

- Process energy from Event and create gluon or quark pairs. Receive sample object from Event class and run it through the splitting algorithm. Get the energy fraction using RollGlue and the charges using RollFlavor. Check to see if there is enough energy to produce two quarks of the mass selected in RollFlavor and if not exit function and signal program to try another splitting. Also check to see if the RollFlavor returned a gluon, if so then do not run the RollLocation function and save processing time, otherwise run RollLocation and get momentum fraction and seperation of quarks. Create Quark object and return this to the Event class.
- Calls Splitter::RollGlue(), Splitter::RollFlavor(), and Splitter::RollLocation()

C. IO

• IO()

- IO class constructor. This constructor calls the Initialize() function and then reads in parameters from specified configuration file. This is done through a large switch that utilizes the map "mapConfigParams" to identify the variable and assign it the correct value. Parameters can be specified in any order in

the config file. If you specify a parameter in the configuration file that is not recognized by the program, an error message will be printed to the standard output and the program will exit. If no seed is specified, a random one will be selected. The final task here is to print out all parameters used by the run to the output directory with the file name "run_parameters#.dat".

  - Calls Initialize() and OutputConfig()

- void Initialize()

  - Initialize variables and map. Set default values for all variables (file paths need to be specified in configuration file or else program won't run). Also initialize map of parameter names as strings to parameter identifiers for use in reading configuration file.

  - Called by class constructor

- void ConvertEvent(vector⟨vector⟨double⟩⟩ &input, double &total)

  - Convert event input to energy. Uses cubic spline created by InitializeEOS() to convert event input into energy. ICCING requires energy density so, as in the case of Trento, when initializing event with entropy or something proportional this needs to be converted. Trento outputs something proportional to entropy, so the a_trento parameter is used to correct this before feeding the value through the EOS. This can be changed and updated for different initial state models. This function also uses an entropy cuttoff value "s_chop" to trim out points that won't be caught by hydro simulations and are thus unneeded for the ICCING algorithm. This is also used to decrease run time since there will be fewer negligible points to process.

  - Called by IO::ReadEvent()

  - Calls FindRange(), InterpolateValue(), and uses SplinSet all defined in functions.h.

- void OutputConfig(string file_name)

  - Print configuration parameters to ICCING output folder for future reference. The parameters are printed to a file named "run_parameters#.dat".

- Called by class constructor

- void OutputFullDensityGrids(vector⟨vector⟨double⟩⟩ &density_grid, string file_name)
  - Print Density Grids with filler 0s. Only prints out a single 2-d grid, so each density must be printed individually.
  - Called by IO::WriteEvent()

- void OutputSparseDensityGrids(vector⟨vector⟨double⟩⟩ &density_grid, string file_name)
  - Print Density Grids without filler 0s. Only prints out a single 2-d grid, so each density must be printed individually. Converts grid points to their x, y coordinates and prints out these values along with the density value at that point.
  - Called by IO::WriteEvent()

- void OutputSparseDensityGrids(vector⟨vector⟨vector⟨double⟩⟩⟩ &density_grid, double tot_energy, string file_name)
  - Print Density Grids without filler 0s (Overload for output densities). Prints out all of the densities to a single file. Converts grid points to their x, y coordinates and prints out these values along with the density values at that point. A header line is printed in this file that specifies "current_event", "grid_step","grid_step","tot_energy", "-grid_max", and "-grid_max" respectively.
  - Called by IO::WriteEvent()

- void OutputEccentricities(double total_entropy, vector⟨vector⟨double⟩⟩ eccentricities, string density_type, string file_name)
  - Print Eccentricities from event. Only prints one type of density at a time, if charge density then print out positive and negative to different files. See Sec. I E for more information on the output format.
  - Called by IO::WriteEvent()

- void OutputQuarkCounts(int up, int down, int strange, int charm, string file_name)
  - Prints quark counts for event. Print out event number, total entropy, and gluon, up, down, strange, and charm counts.
  - Called by IO::WriteEvent()
- Event InitializeEvent()
  - Initialize the Event Object. Pass all parameters to event object that are needed. Initialize density grids with zeros. Create gluon mask, grid that contains a circle with radius "gluon_rad" where points in the circle are 1 and outside are 0. Using this mask to transfer energy decreases run time since there is no need to calculate the circle more than once. Create quark mask, grid that contains a circle with radius "quark_rad" where points in the circle are set to a normalized Gaussian profile and outside are 0. Using this mask to distribute quarks decreases run time since there is no need to calculate the Gaussian more than once.
  - Called in main.cpp at beginning of run
- Splitter InitializeSplitter()
  - Initialize the Splitter Object. Pass all parameters to splitter object that are needed. Initialize correlation function. Read in quark chemistry file and create a cubic spline set for each flavor for use in Splitter::RollFlavor().
  - Called in main.cpp at beginning of run
- void InitializeEOS()
  - Initialize equation of state. Read in EOS from file and create a cubic spline set for interpolation between entropy and energy.
  - Called in main.cpp at beginning of run
- Event ReadEvent(Event event_in)
  - Read Event specific Energy density. This function has code implemented for full grid input and sparse array input. Read in event profile and calculate total

entropy. Convert entropy to energy. Read in T_A and T_B, these can be selected for reading by specifing their appropriate flag in the configuration file. The reason for these profiles is to calculate Q_s, the calculation of this can be modified here.

  - Calls IO::ConvertEvent()

- void WriteEvent(Event event)
  - Write Energy densities, eccentricities, and quark counts. It is possible to specify the output as full density grids or sparse density grids, this only effects the density profiles. Regardless of choice, eccentricities and quark counts are always printed. This is where the current_event number is incremented.
  - Calls IO::OutputFullDensityGrids(), IO::OutputSparseDensityGrids(), IO::OutputEccentricities(), and IO::OutputQuarkCounts()
- bool LastEvent()
  - Return true if on last event
  - Called in main.cpp

D. Functions

- Used by IO::InitializeEOS and IO::ConvertEvent
- struct SplineSet
  - Data Structure for spline functions on interval x - x+1
- vector⟨SplineSet⟩ CubicSpline(vector⟨double⟩ &x, vector⟨double⟩ &y)
  - Natural Cubic Spline (c++ implimentation: https://bit.ly/3hP8dUd, algorithm: https://bit.ly/3gdXZfK)
- double InterpolateValue(SplineSet range, double value)
  - Interpolate Value in given SplineSet range
- SplineSet FindRange(vector⟨SplineSet⟩ function, double value)
  - Find the range that contains the value

E. Global

- All structs used by Event and Splitter
- struct Charge
  - Data Structure Used to keep track of particles and their charges. The input parameter "charge_type" can be set to "BSQ" or "UDS" to specify the charges tracked. The values of these charges can be seen in Table VIII. Contains functions that are used to set the charge as gluon, up, down, strange, or charm and a function that returns the current charge.
- struct Quarks
  - Data Structure that contains the relevant data to specify a pair of quarks ie. their charge, energy fraction, momentum fraction, and center of mass position. The initialization function is CreateQuarks and requires input for all parameters listed above. There are also functions for accessing all of the parameters of this object.
- struct Sample
  - Data Structure for sample of initial energy density. This is initialized by the Event class and passed to the Splitter for processing. This contains the total energy of the sample and the associated Qs.

Flavor	B	S	Q	U	D	S
u	$\frac{1}{3}$	0	$\frac{2}{3}$	1	0	0
d	$\frac{1}{3}$	0	$-\frac{1}{3}$	0	1	0
s	$\frac{1}{3}$	-1	$-\frac{1}{3}$	0	0	1
c	$\frac{1}{3}$	0	$\frac{2}{3}$	0	0	0

TABLE VIII: The conserved charges for the relevant quark flavors: BSQ [baryon number (B), strangeness (S), and electric charge (Q)], and UDS [up (U), down (D), and strange (S)].

F. Eccentricity

- vector⟨double⟩ StandardCalculation(string density_type, int m, int n)
  - Calculate standard energy eccentricity. Takes as input a density type (only allowed value is "Energy"), radial (m) weight, and angular (n) weight. This is a standard geometric eccentricity calculation. See Sec. IV for more information.
  - Called by Eccentricity::CalculateEccentricities()
- vector⟨double⟩ NewCalculation(string density_type, int m, int n)
  - Calculate standard eccentricities, separating positive and negative density values. Allowed density types are "Baryon", "Strange", and "Charge". Aside from the separation of positive and negative values, this function is identical to the StandardCalculation.
  - Called by Eccentricity::CalculateEccentricities()
- vector⟨vector⟨vector⟨double⟩⟩⟩ CalculateEccentricities(int grid_max, double grid_step, vector⟨vector⟨vector⟨double⟩⟩⟩ density)
  - Calculate all eccentricities for given event. Takes as input a 3-d vector object containing full grid densities and converts this to a sparse array of energy valued points for ease of processing. At the same time, the energy center of mass is calculated for use in the calculation of eccentricities. The return type used here contains all of the calculations needed for analysis and is a little complicated, in Table IX this is defined.
  - Called by Event::CalculateEccentricities()
  - Calls Eccentricity::StandardCalculation() and Eccentricity::NewCalculation()

G. Correlator

- Correlator(string model, double lambda, double alphas)

$\{\varepsilon_{22}^{energy}, \psi_{22}, r\}$	$\{\varepsilon_{33}^{energy}, \psi_{33}, r\}$	$\{\varepsilon_{44}^{energy}, \psi_{44}, r\}$	$\{\varepsilon_{55}^{energy}, \psi_{55}, r\}$
$\{\varepsilon_{22}^{B-}, \psi_{22}^{B-}, r^{B-}, \varepsilon_{22}^{B+}, \psi_{22}^{B+}, r^{B+}\}$	$\{\varepsilon_{33}^{B\mp}, \psi_{33}^{B\mp}, r^{B\mp}\}$	$\{\varepsilon_{44}^{B\mp}, \psi_{44}^{B\mp}, r^{B\mp}\}$	$\{\varepsilon_{55}^{B\mp}, \psi_{55}^{B\mp}, r^{B\mp}\}$
$\{\varepsilon_{22}^{S-}, \psi_{22}^{S-}, r^{S-}, \varepsilon_{22}^{S+}, \psi_{22}^{S+}, r^{S+}\}$	$\{\varepsilon_{33}^{S\mp}, \psi_{33}^{S\mp}, r^{S\mp}\}$	$\{\varepsilon_{44}^{S\mp}, \psi_{44}^{S\mp}, r^{S\mp}\}$	$\{\varepsilon_{55}^{S\mp}, \psi_{55}^{S\mp}, r^{S\mp}\}$
$\{\varepsilon_{22}^{C-}, \psi_{22}^{C-}, r^{C-}, \varepsilon_{22}^{C+}, \psi_{22}^{C+}, r^{C+}\}$	$\{\varepsilon_{33}^{C\mp}, \psi_{33}^{C\mp}, r^{C\mp}\}$	$\{\varepsilon_{44}^{C\mp}, \psi_{44}^{C\mp}, r^{C\mp}\}$	$\{\varepsilon_{55}^{C\mp}, \psi_{55}^{C\mp}, r^{C\mp}\}$

TABLE IX: Eccentricities as stored by the Event class.

  - Correlation class constructor. This initializes important variables for the calculation of the correlation function; lambda which is used by the MV model and alphas used by GBW and MV models. The "model" variable is used to bind the specified dipole model correlation function to the function data type "corr". This is a simple way to select the correlation function at the beginning of a run and not have to continually run an if-else statement to check which dipole model to use. "GBW" is the default correlation function but this could be changed depending on your preference.

- double GBWModel(double r, double alpha, double m, double Qs)
  - Correlation function in GBWModel. This function splits the correlation function into three terms that makes it easy to read and returns the product of them.
- double MVModel(double r, double alpha, double m, double Qs)
  - Correlation function in MV Model. This function splits the correlation function into three terms that makes it easy to read and returns the product of them.
- double FindMaximum(double alpha, double m, double Qs, double lower, double upper, double tolerance)
  - Find Maximum Probability given quark flavor, random alpha, and Qs (Golden Section Search). Uses a specified tolerance value that is hard coded as 0.001 which is small enough to produce the correct result.
- double F(double r = 0., double alpha = 0., double m = 0., double Qs = 0.)

– Given quark pair data, determine distance and momentum fraction. This function calls the general function data type "corr" which has been previously set to point to the correct dipole model correlation function.

IV. ECCENTRICITY CALCULATION

To calculate the eccentricities of energy and different charge distributions, a method is used where the eccentricity vector is defined in the imaginary plane in order to disentangle the magnitude from the angular information. Since this process is a bit involved, this section provides a derivation of this method and a description of its implementation in the ICCING code.

A. Initial Condition

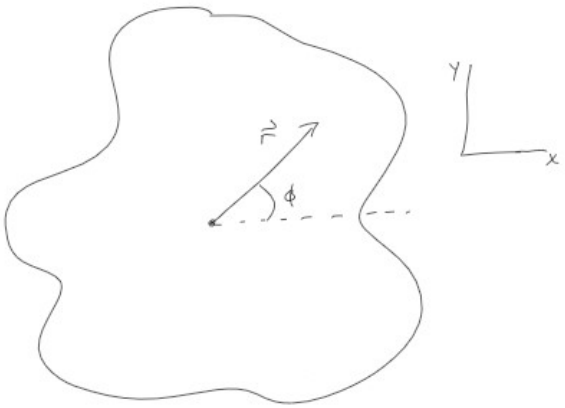

FIG. 11:

In order to describe an initial condition with some density geometry, there needs to be a consistent reference for coordinates which is taken to be the center of mass. From this reference point, a vector can be drawn to any point in the density and represented by a magnitude, $r^2 = (\Delta x)^2 + (\Delta y)^2$, and angle, $\phi = arctan(\frac{\Delta y}{\Delta x})$. In order to calculate different orders of eccentricities, the radius vector needs to be rescalable:

$$r^m = \rho * (r^2)^{m/2} \tag{1}$$

where ρ is the local density and m is the radius weight.

B. Definition of Eccentricity

The eccentricity vector can be defined as an average over all points in the initial condition where each point can be described with separately weighted magnitude and angular components. The convention is to use m as the radial weight and n as the angular weight.

$$\vec{\epsilon}_{mn} = \langle r^m e^{in\phi} \rangle \tag{2}$$

Using Euler's formula, we can separate this into its real and imaginary components:

$$\vec{\epsilon}_{mn} = \langle r^m cos(n\phi) \rangle + i \langle r^m sin(n\phi) \rangle \tag{3}$$

The representation of this vector, can be viewed in the imaginary plane as having a magnitude and angle as seen in Fig. 12.

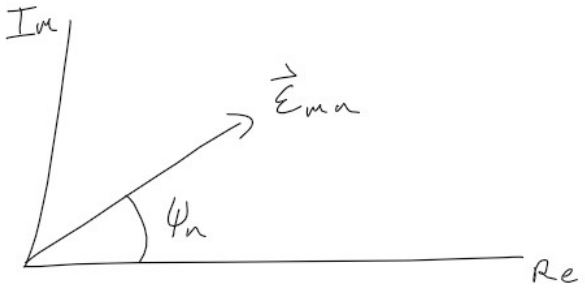

FIG. 12:

This allows the magnitude and angular components to be calculated separately :

$$\vec{\epsilon}_{mn} = \begin{cases} \psi_n, & \langle arctan \frac{Im(\vec{\epsilon}_{mn})}{Re(\vec{\epsilon}_{mn})} \rangle \\ \epsilon_{mn}, & \frac{1}{Norm} \sqrt{Re(\vec{\epsilon}_{mn})^2 + Im(\vec{\epsilon}_{mn})^2} \end{cases} . \tag{4}$$

Here ψ_n is the relative event plane angle that is used to define the eccentricity. The average is over the number of points used in the calculation.

C. Final Equations in Code

We can make this easier to implement in the code by inserting the real and imaginary parts of the vector back into the angle and magnitude equations:

$$\psi_n = \frac{1}{N} \sum_i^N tan^{-1}(\frac{r_i^m sin(n\phi_i)}{r_i^m cos(n\phi_i)}); \tag{5}$$

$$\epsilon_{mn} = \frac{1}{r_{tot}^m} \sum_i^N r_i^m cos(n[\phi_i - \psi_n]). \tag{6}$$

N is the number of points used in the calculation and

$$r_{tot}^m = \sum_i^N r_i^m. \tag{7}$$