TECHNISCHE
UNIVERSITÄT
DRESDEN

Coupling of atomic states to particle in cell simulations

Master-Arbeit
zur Erlangung des Hochschulgrades
Master of Science
im Master-Studiengang Physik

vorgelegt von

Brian Edward Marré
geboren am 06.07.1994 in Berlin

Institut für Strahlenphysik
Helmholtz Zentrum Dresden Rossendorf
2020

Eingereicht am 23. November 2020

1. Gutachter: Prof. Dr. Ulrich Schramm
2. Gutachter: Prof. Dr. Thomas Cowan

Summary

Abstract
English:

This thesis develops all components necessary for the coupling of an atomic physics solver to a particle in a cell-simulation. I develop an new memory and parallelization optimized atomic rate solver, an adaptive histogram and a new to represent atomic states, in order to make modeling atomic physics directly coupled to a PIC-simulation feasible.

Abstract
Deutsch:
Diese Arbeit entwickelt all komponenten die notwendig sind for die Kupplung der atomphysik an eine PIC-Simulation. Ich entwickle dafür einen neuen specuher und parallelisierungs optimierten atomaren Raten Gleichungs solver, ein neues adaptive histogram und eine neue rpresentation von atomaren Zuständen, um die direkte Kupplung von atom physics and eine PIC-Simulation möglich zu machen.

Contents

1 Introduction

1.1 Particle in Cell Algorithm

Particle in Cell, or short PIC, is a general algorithm used to simulate plasmas. The basic idea of PIC-algorithms is to model a plasma using two components, a discretized electromagnetic field and samples of the phase space density distribution, the so called macro-particles. Each macro particle has at least a position, a momentum, and a charge to mass ratio and may move freely in space. Macro particles are accelerated or decelerated according to the Lorentz-force, due to the electromagnetic field at their position. In pure PIC codes there are no interactions directly between macro-particles. Macro-particles instead interact indirectly over the electromagnetic field between cells and not at all in the same cell. The electromagnetic fields in contrast are discretized on grid points and updated by a Maxwell solver. A PIC-simulation time step

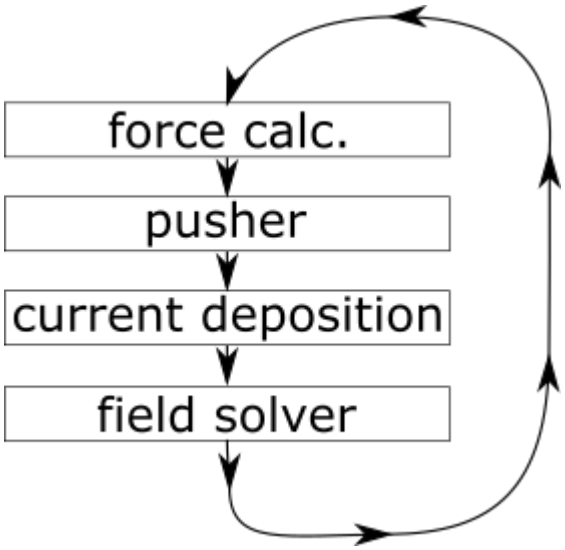

Figure 1.1: The four steps of the PIC-cycle

usually consists of four distinct steps,

1. Force calculation

   Interpolation of electromagnetic fields from the grid points to the macro particle position.

2. Particle pusher

Change of momentum and position of all macro particles due to the forces calculated in the previous step.

3. current deposition

   Deposition of the current caused by macro particle movement to grid points.

4. field solver

   Update of the $\vec{E}$ and $\vec{B}$ field values at every grid point based on neighboring grid points field values and macro particle currents.

The main advantage PIC simulation have against other plasma simulation techniques is their ability to scale with the computational power available while correctly modeling complex phase space plasmas. PIC-simulations sample the complete phase space and are able to model non linear plasma process as well as laser plasma interactions and non linear optics, while being more performant than Vlasov or particle-particle solvers. PIC-simulations are therefore the tool of choice for modeling laser generated plasmas.
The main disadvantages of PIC-simulations against other plasma simulations is the close coupling of time and space resolution due to the Courant–Friedrichs–Lewy condition. In conjunction with the comparatively small maximum cell sizes necessary to correctly model the macro particles dynamics. The resulting small time step sizes increase the length of computations significantly.

1.2 GPU Architecture

There are essentially two mayor peculiarities that must be considered when programming for GPU. Firstly, in contrast to CPUs, GPUs are optimized for working on a large number of parallel tasks. They are therefore not structured around a small number of threads, but around streaming multiprocessors, short SM. Each SM is designed to take 100s of the same task and distribute them on a set of execution units. The individual execution units of GPUs have a comparatively lower clock frequency, are less flexible and lack the extensive data flow and precaching mechanisms of standard CPU cores. This alleviated by there being many more of them many more, $\leq$ 10000 vs. $\leq$ 128, enabling a much higher overall output if all the execution units are utilized. Which brings us to one of the central challenges of utilizing GPUs, we must be able to distribute our task on as many execution units as possible to fully utilize the high throughput of GPUs.. Programs that want to use the much larger compute power of GPUs must therefore be nearly completely parallel.
The second peculiarity that must be recognized is the fact that GPUs have considerably less memory available close to the actual execution units, due to devoting more of their transistor budget to computation than a standard CPU. It is therefore imperative that the memory that is available on each memory hierarchy levels is used as efficiently as possible.

The memory hierarchy of GPUs consist of two mayor levels. Most distant from the GPU execution units itself, is the global device memory, with access latencies in the order of 600 clock cycles. This memory can be accessed by all SMs of a GPU and has maximum size of currently about 40GB, Nvidia A100 GPUs, large but not overly so, especially compared to the several 100s of GB of host memory most compute nodes possess.
The next step up in access is the on die shared memory, which is accessible from all SM and can be up to 100 times faster[21] faster to access than global device memory, but is also considerably smaller, currently a maximum of 40MB per GPU, Nvidia A100. Lower levels of memory exists with the SM bound L1 memory, which may be accessed even faster, but also is considerably smaller with a current maximum size of 192KB per SM and must be split between L1 cache and SM execution units shared memory.

1.3 The need for atomic Physics in PIC-simulations

The modeling of atomic physics in PIC-simulations has many applications. From probing plasma conditions, calculating opacities, to laser absorption, all these processes directly or indirectly depend on atomic physics process and the current atomic state distribution of a plasma. Correctly modeling the atomic states and process is therefore essential in understanding process in high density plasmas.
Unfortunately the typical implementations of atomic physics in PIC-Simulations only model atomic physics process from and to ground states[22][23]. This limits PIC-simulations significantly. For example it is to be expected that the missing transition lead to significantly changed ionization ratio[1]. This is especially problematic for modeling the laser absorption, due to absorption rate being self consistently coupled to the free electron density.
In addition many instability mechanism depend on the hydrodynamic pressure, for example RT-like or the free electron flow, e.g. Weibel-like which are directly linked to the free electron density. For high laser intensities, both of these effects have growth rates in the neighborhood of the laser pulse length and consequently may cause transient effects, that must be exactly modeled and require accurate density predictions and therefore ionization dynamic.
Another possible application of detailed atomic physics models is XFEL like probing of plasma conditions, for which not only the charge state but also the actual electron configuration becomes relevant[20] which is typically completely missing from PIC-simulations, or is only available as not self consistent look-up tables assuming some sort of equilibrium like conditions.
The lack of an accurate modeling of atomic physics in PIC is therefore becoming a hindrance in several applications of plasma simulations. Especially for methods aimed at probing the dynamics of the opacity during or short after a short laser pulse, when the plasma is far from equilibrium need an accurate model of transient atomic physics[24]. We therefore need a better model for atomic physics in PIC-Simulations.

2 Atomic States

Atomic physics relies on a set of states for its description and as such an state basis has to be chosen before a discussion of atomic dynamics is possible. Choosing this state basis to minimize memory usage is of critical importance due to the comparatively large amount of memory required to store atomic states and the already comparatively large memory usage of PIC-simulations.
We explore different atomic state basis and optimize the representation of atomic states to reduce the memory required to the absolute minimum while still representing the important physics.

2.1 Nomenclature and Notation

2.1.1 Nomenclature

Before we begin I want to introduce a few basic definitions, that deviate from the common usage or might be unknown. The first of such being the term ion, in the following *ion* describes an atomic core and its bound electrons, irrespective of the overall charge. A neutral atom, a completely ionized atomic core and a partially ionized ion are therefore all described as ions. This follows the general convention in literature and shortens descriptions, as well as reflecting the identical theoretical description used for this particles.
Secondly, the term *atomic state* of an ion refers to an electron hull configuration or a number of similar configurations combined. While not strictly speaking the correct term, since we neither limit us to atoms, nor are these necessarily quantum mechanical states, this term is intended to refer to any internal *state* of an ion that is important for Atomic physics.
Thirdly, the *charge state*, how many electrons are bound to the atomic core, is not regarded as a property separate of the atomic state of an ion. Instead it is used as a derived quantity of the fundamental atomic state, important due to its influence on particle trajectory.
Lastly, the the actual quantum mechanical state of a bound electron is described as the *electron state*.

2.1.2 Notation

The symbol # is used to denote a number or count of something.

2.2 Modeling the atomic state

2.2.1 Modeling the atomic state of a single ion

To specify the electron hull configuration, the number of electrons (charge state) and electrons states occupied must be given. Describing the number of electrons is easy, describing the electron state can be difficult, as will be shown in the following.
The electron states of the hydrogen ion are analytically known, but ions with more than one electron do not have analytic eigenstates due to the electron-electron interactions. The multi-electron system electron states must therefore be approximated. Numerical solutions of these systems do exist, but computing them requires time and is generally only necessary if high spectroscopic accuracy is required[1], the lower accuracy obtainable without using exact states being sufficient for now. Instead, we use the hydrogen electron states as a basis with the implicit assumption that the electron-electron interaction and external contributions are much smaller than the central field. To somewhat account for screening effects and electron-electron interactions a semi empirical mean field approximation can be used, modifying the energy of all states based on the occupation numbers of all lower states. These approximate states are called screened hydrogen states.
These electron states provide a general description that is mostly complete, meaning they cover a wide energy range and most atomic physics initial and final states. In addition, they also are easy to compute and require relatively few memory to store. How much memory is exactly necessary can be estimated by the number # of possible states and the number of bits necessary to store an unsigned integer that large, the basic idea being that the minimal memory storage solution would be a integer index assigned to all distinct states. An upper limit of the number of super configurations is derived in the following chapter.

2.2.2 Number of electron states

A hydrogen ion electron state in a plasma can be specified by four quantum numbers:

$$\begin{aligned} 1 \leq & n \leq n_{\max} && , n \in \mathbb{N}/\{0\} \\ 0 \leq & l \leq n-1 && , l \in \mathbb{N} \\ -l \leq & m_l \leq l && , m_l \in \mathbb{Z} \\ & s \in \left\{\frac{1}{2}, -\frac{1}{2}\right\} \end{aligned}$$

These states are further grouped into levels, the level n containing $g(n)$ electron states with the principal quantum number n, $g(n)$ being the degeneracy of this level.
In contrast to the familiar atomic physics model, the principal quantum number n has an upper limit $n_{\max}$ due to potential depression effects in plasmas caused by the proximity of

other ions. Only electron states up to $n_{\max}$ remain bound, typical value $n_{\max} \approx 6$ [1], with $n_{\max}$ depending on the ionization potential depression model used, the plasma conditions and ions in question.
Based on this we can calculate the number of electron states $\#_{\text{electron states}}(n_{\max})$ as the sum over the degeneracy $g(n)$ of all existing atomic levels,

$$\#_{\text{electron states}} = \sum_{n=1}^{n_{\max}} g(n) \tag{2.1}$$

with $g(n)$ having the value,

$$g(n) = \sum_{l=0}^{n-1} \sum_{m_l=-l}^{+l} \sum_{s=-\frac{1}{2}}^{\frac{1}{2}} 1 = \sum_{l=0}^{n-1} \sum_{m_l=-l}^{+l} 2 = 2 \cdot \sum_{l=0}^{n-1} (2 \cdot l + 1) = 2 \cdot \left(\sum_{l=0}^{n-1} 2 \cdot l + \sum_{l=0}^{n-1} 1 \right) \tag{2.2}$$

$$= 2 \cdot \left(2 \cdot \left(\sum_{l=1}^{n-1} l \right) + n \right) \overset{\text{Gaus}\sum}{=} 2 \cdot \left(\cancel{2} \cdot \frac{(n-1) \cdot ((n-1)+1)}{\cancel{2}} + n \right) \tag{2.3}$$

$$\boxed{g(n) = 2 \cdot n^2} \tag{2.4}$$

substituting results in following equation

$$\#_{\text{electron states}} = \sum_{n=1}^{n_{\max}} 2 \cdot n^2 \tag{2.5}$$

which is double the square pyramidal number and can be calculated directly,

$$\#_{\text{electron states}} = 2 \cdot \left(\frac{1}{3} \cdot (2 \cdot n_{\max} + 1) \cdot \sum_{n=1}^{n_{\max}} n \right) \tag{2.6}$$

$$\overset{\text{Gaus}\sum}{=} \cancel{2} \cdot \left(\frac{1}{3} \cdot (2 \cdot n_{\max} + 1) \cdot \frac{(n_{\max} + 1) \cdot n_{\max}}{\cancel{2}} \right) \tag{2.7}$$

$$= \frac{2 \cdot n_{\max}^3 + 3 \cdot n_{\max}^2 + n_{\max}}{3} \tag{2.8}$$

For example for $n_{\max} = 6$ there are $\#_{\text{electron states}} = 182$ different electron states.

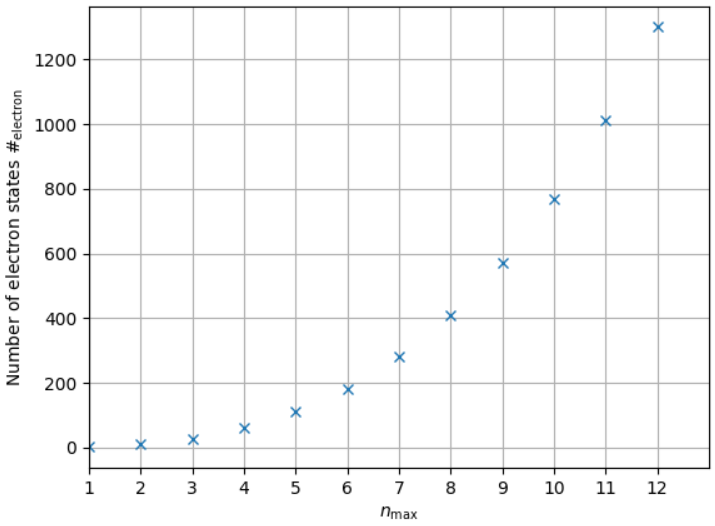

Figure 2.1: Number of electron states for different $n_{\max}$

2.2.3 Number of configurations

For a single electron system the number of atomic states is equal to the number of electron states. Following the Pauli exclusion principle and indistinguishability, in many electron system we need to distribute i electrons on $\#_{\text{electron states}}$ possible electron states without considering the sequence and without repetition. This is a bit of a mixed blessing, since it limits the number of states compared to independent electrons, but also makes counting them more difficult. The number of possible distinct configurations can be calculated with the the binomial coefficient.
Based on this we get the total number of atomic States #, for an ion of atomic number Z.

$$\# = \sum_{i=0}^{Z} \binom{\#_{\text{electron states}}}{i} \tag{2.9}$$

$$= \sum_{i=0}^{Z} \binom{\frac{2 \cdot n_{\max}^3 + 3 \cdot n_{\max}^2 + n_{\max}}{3}}{i} \tag{2.10}$$

For the example of titanium with charge number Z = 22 and $n_{\max} = 6$, this yields:

$$\# \overset{n_{\max} = 6; Z = 22}{=} \sum_{i=0}^{22} \binom{182}{i} = \sum_{i=0}^{22} \frac{182!}{(182 - i)! \cdot i!} \tag{2.11}$$

$$\# \overset{n_{\max} = 6; Z = 22}{\approx} 1,44 \cdot 10^{28} \tag{2.12}$$

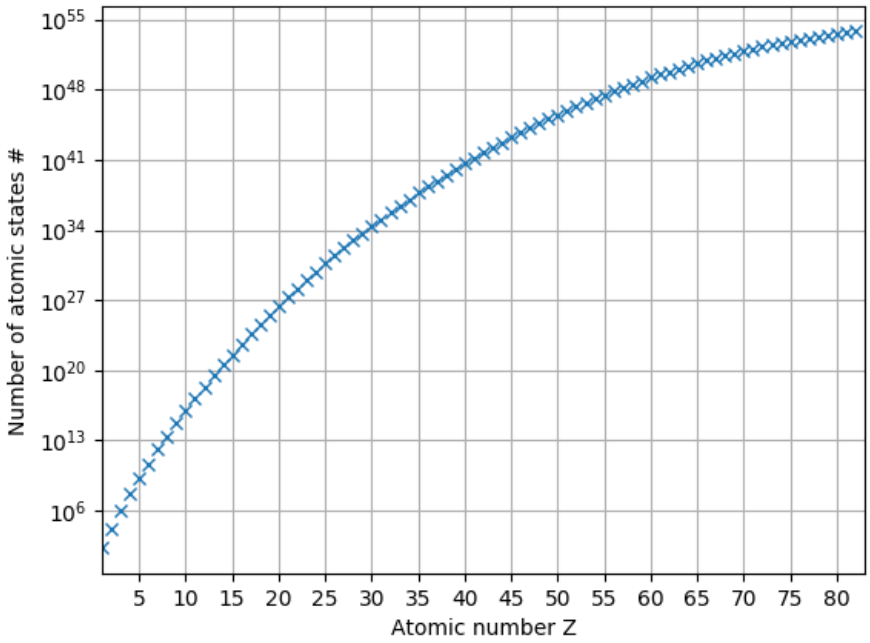

Figure 2.2: Number of atomic states of the atomic number Z, for $n_{\max} = 6$

A cusory glance on equation 2.11 and the figure 2.2 shows the enormous number of configurations that exist for every but the lowest Z ions.
This demonstrates a problem generally encountered in modeling atomic states. The number of atomic states is so large that even representing distributions over all of them becomes a challenge, never mind calculating their dynamics. Storing a vector with that many components as a 0-dimensional atomic state distribution would already exceed the available storage in even HPC-clusters by several orders of magnitudes. It is therefore necessary to reduce the number of states that we distinguish between.
Possible solutions would be:

1. sacrifice general completeness

   We want to able to model a very wide range of plasma conditions, sacrificing general completeness is therefore not an option.

2. group atomic states

   group by energy of states: The energy of a given screened hydrogen atomic state does not depend on the complete configuration, but rather only on the occupation numbers N_i of each level i, due to the assumptions and simplifications made. Grouping states by energy, into a version of so called *super configurations* [1], is therefore natural. This grouping tracks super configurations $\vec{N} = (N_1, N_2, \ldots, N_{n_{\max}})$ instead of configurations without large loss of detail, at the cost of having to assume a distribution if the angular

momentum is required.

To see whether super configurations actually reduce our problem enough such that we can represent them in GPU memory, we take a look at the number of super configurations that exist.

2.2.4 Number of super configurations

Each levels' occupation number N_n is limited by the number of electron states $g(n)$ for this level, since each electron states can only be occupied by one or zero electrons according to the Pauli principle.

$$0 \leq N_n \leq g(n) \tag{2.13}$$

Each level n therefore has $g(n) + 1$ different occupation states, resulting in $\#_{\vec{N}}$ different occupation vectors possible.

$$\#_{\vec{N}} = \prod_{n=1}^{n_{\max}} (g(n) + 1) = \prod_{n=1}^{n_{\max}} \left(2 \cdot n^2 + 1\right) \tag{2.14}$$

$$\overset{n_{\max}=6}{=} 3 \cdot 9 \cdot 19 \cdot 33 \cdot 51 \cdot 73 = 63\text{‘}026\text{‘}667 \tag{2.15}$$

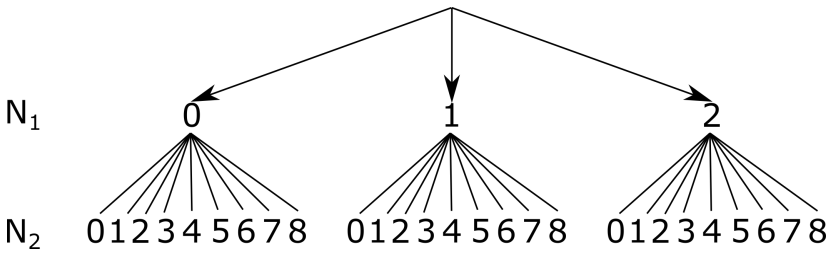

Figure 2.3: combinatorial tree of super configurations

This calculation of $\#_{\vec{N}}$ does not include the global limit on the number of electrons,

$$\sum_{n=1}^{n_{\max}} N_n \overset{!}{\leq} Z \tag{2.16}$$

Specifically we included super configurations with up to $\#_{\text{electron states}}(n)$ electrons in each level n, already possibly more electrons than are actually available, in each level. $\#_{\vec{N}}$ is therefore only an upper limit of the actual number of possible super configurations for a given Z and $n_{\max}$.

This limit can be somewhat improved by replacing $g(n)$ in equation 2.14 with the minimum of Z and $g(n)$, thereby limiting each levels occupation number to the maximum number of electrons available. This eliminates some unphysical super configurations without introducing an interdependence between occupation numbers, something that will become important later(

see 2.3.1).

$$\#_{\vec{N}} = \prod_{n=1}^{n_{\max}} (\min(g(n), Z) + 1) \tag{2.17}$$

Nevertheless this does not solve the underlying issue of containing unphysical configurations, only reduces its impact.

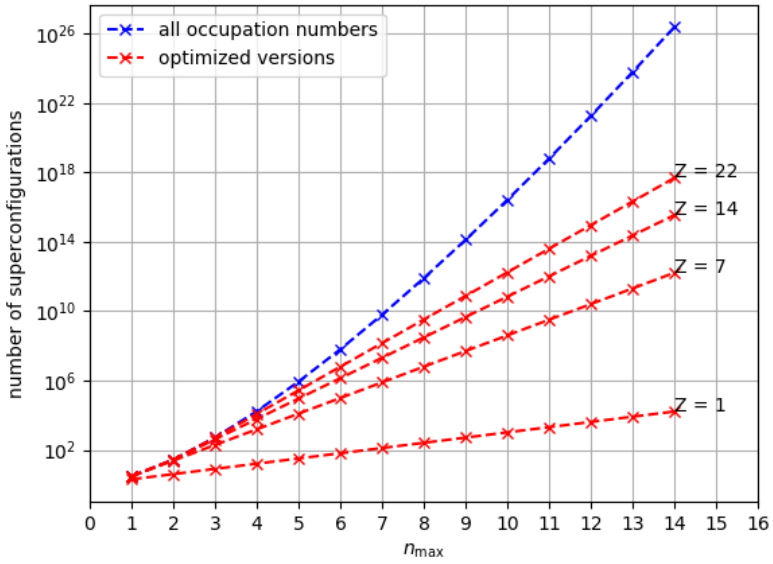

Figure 2.4: Number of super configurations based on equations 2.14 and 2.17 for $Z = 22$

The resulting upper limit of the number of physical super configurations is small enough to be represented in a 3-dimensional simulation and super configurations are therefore used as compound atomic states. How much memory is actually required for this will be discussed in sections 2.3.1 and 2.3.2

2.2.4.1 Additional ideas and dead ends

It is possible to get a better estimate of the number of physical super configurations if further external dependences and an interdependence between occupation numbers are acceptable. I will list a few ides in that direction but the solution detailed so far is good enough for our current purpose.
Ideally we would find way to count only those states that also fulfill the following condition,

$$\sum_{n=0}^{n_{\max}} N_n \leq Z \tag{2.18}$$

Unfortunately this condition introduces an interdependence between different level's occupation numbers, thus making counting the number of states much more difficult.

- Leaving the last occupation number unspecified and calculating it using the total number of electrons $(Z-q)$ for an ion of atomic number Z and charge $+q$, would in theory allow some improvements, but would introduce an external dependence on a variable value as well as *still* including unphysical super configurations.

$$\sum_{n=0}^{n_{\max}} N_n \stackrel{!}{=} Z - q \tag{2.19}$$

$$N_{n_{\max}} = (Z-q) - \sum_{n=0}^{n_{\max}-1} N_n \tag{2.20}$$

  Specifically super configurations whose sum of occupation numbers of levels up to $n_{\max}-1$ are already larger than Z would result in a negative occupation number $N_{n_{\max}}$, one would have to check for explicitly.

- One approach that might be successful is a variation of the multinomial coefficient. The multinomial coefficient itself is not directly applicable, since it does not allow the presence of not filled states.

- One might be tempted to use the familiar answer for distribution of i electrons on $n_{\max}$ different levels, without regard for sequence and without repetition and sum these numbers over all possible charge states.

$$\# = \sum_{i=0}^{Z} \binom{n_{\max} + (Z-q) - 1}{n_{\max}} \tag{2.21}$$

  Unfortunately this would include super configurations violating equation 2.13, for example the super configuration $\vec{N} = (Z-q, 0, 0, ...)^{\mathrm{T}}$.

2.3 Storing the atomic state

2.3.1 Indexing of super configurations

Atomic states are likely to add a considerable amount of required memory to already large PIC-Simulations, due to the large number of states existing, it is therefore essential to reduce the memory required to store atomic states.
The most memory efficient way to store states is always an *index*, due to it being bijective, but this often requires an index-Atomic State conversion table. This conversion table must be stored and requires memory itself, negating some of the savings, *unless* it is possible to convert the index analytically to the atomic state it represents. This is in fact possible with the counting scheme detailed in section 2.2.4.

The counting scheme we used above relied on the independence of different occupation numbers by building a large combinatorial tree with the value of each levels' occupation number as a branching point at the corresponding branching level of the tree. This counting scheme also allows us to enumerate the states easily.

Starting with the completely ionized state, $\vec{N} = (0, 0, ...)$ as state 0, we increase the lowest level occupation number by one until it reaches its maximum value, each step increasing the index by one. Upon reaching the maximum of the lowest occupation number, it wraps around and the next higher level's occupation number is increased by one, thereby reaching a new atomic state. This principle is repeated, increasing the next highest occupation number by one whenever an occupation number reaches its maximum and wraps around, until the last occupation number reaches its maximum, $N_{n_{\max}} \leq \min(g(n), Z)$.

This enumeration scheme traverses every physical atomic state and allows easy direct calculation of the index corresponding to a given occupation number vector. The index calculation is based on the fact that the number of steps it takes to increase a the occupation number of a given level by one is always equal to the number of possible occupation number vectors of current length l - 1, since there are always that many steps in between. This number can be easily calculated using equation 2.17, due to each levels occupation number being independent of all others. Since it takes N steps to reach the value of N starting from 0 and increasing x by one and lower occupation number values are simply further steps after reaching the last occupation number value, the equation 2.23 follows.

$$\# = N_1 + N_2 \cdot 3 + N_3 \cdot 3 \cdot 9 + \ldots \tag{2.22}$$

$$= \sum_{i=1}^{n_{\max}} \left(N_i \cdot \prod_{j=1}^{i-1} (\min(g(j), Z) + 1) \right) \tag{2.23}$$

This function is bijective, since it is based on a table containing no repeated atomic states, and can therefore be in-

index	occupation numbers				
	N_1	N_2	N_3	N_4	N_5
0	0	0	0	0	0
1	1	0	0	0	0
2	2	0	0	0	0
3	0	1	0	0	0
4	1	1	0	0	0
5	2	1	0	0	0
6	0	2	0	0	0
7	1	2	0	0	0
8	2	2	0	0	0
9	0	3	0	0	0
10	1	3	0	0	0
11	2	3	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮
24	0	8	0	0	0
25	1	8	0	0	0
26	2	8	0	0	0
27	0	0	1	0	0
28	1	0	1	0	0
29	2	0	1	0	0
⋮	⋮	⋮	⋮	⋮	⋮
510	0	8	18	0	0
511	1	8	18	0	0
512	2	8	18	0	0
513	0	0	0	1	0
⋮	⋮	⋮	⋮	⋮	⋮
11798	2	8	18	22	0
11799	0	0	0	0	1
⋮	⋮	⋮	⋮	⋮	⋮

Table 2.1: combinatorial table built according to enumeration scheme, for $Z = 22$ and $n_{\max} = 5$

verted using whole number division.

$$N_k = \mathrm{div}\left(\# - \sum_{i=k+1}^{n_{\max}} \left[N_i \cdot \prod_{j=1}^{i-1} (\min(g(j), Z) + 1) \right], \prod_{i=1}^{k-1} (\min(g(i), Z) + 1)\right) \tag{2.24}$$

This allows an analytic conversion between index and occupation number vector, without actually storing the table.

Equation 2.23 is consistent with the in the previous section derived equation of the total number of super configurations, equation 2.17, this is shown in section A.

The number of indices required also defines the memory required to store one index per ion. The number of bits required can be calculated as the logarithm of base 2 of the number of super configurations.
2.2

data type	length		$(n_{\max})_{\max}$, Z=			
	bit	byte	1	7	14	22
uint8_t	8	1	8	3	2	2
uint16_t	16	2	16	5	4	4
uint32_t	32	4	32	10	8	8
uint64_t	64	8	64	21	17	16

Table 2.2: short table listing common c++ data types and possible $n_{\max}$ values for different Z values

Table 2.2 gives the maximum $n_{\max}$ possible for different atomic number and integer data types. For hydrogen, $Z = 1$, an uint8_t, allows representation of all states for an $n_{\max} = 6$, while in the case of Nitrogen double the memory only allows representation of all states up to $n_{\max} = 5$. Representing all states up to $n_{\max} = 6$ for Nitrogen requires an uint32_t. An uint64_t is sufficient to represent all elements up to uranium for $n_{\max} = 6$. If very high $n_{\max}$ need to represented, typically $n_{\max} \approx 100$ for astrophysical plasma densities, even for hydrogen an uint64_t can be insufficient.

2.3.2 Atomic state representation

The previous sections described what atomic states we will be modeling, how many of them there are and how we index them, what remains is the question how to represent them in the framework of a PIC-simulation.
Fundamentally atomic states are properties of ions and as such can not be separated form their respective physical particle. This creates a co dependence of ion density distributions and atomic state distributions. To understand what to exactly this entails we will take a look at the fundamental modeling used in PIC-simulations.

Particles in PIC-simulations are modeled as a density in phase space. As such, the fundamental object representing particles in PIC-simulations is a position and momentum dependent density for a given particle species, and charge state in the case of ions. For an ion species, each such point in phase space is also associated with the ions' atomic states, forming an atomic state distribution $\vec{n}$.
This density distribution is then being sampled by macro particles, sampling both the particle density in the usual phase space as well as the associated atomic state distribution. Since macro-ions correspond to the density equivalent of a large number of physical ions, their sample typically contains a large number of atomic states, rather than a single atomic state.
The distribution associated with each macro particle sample only contains atomic states of a single charge state, since macro-ions must have a defined charge state to be able to predict their movement. The natural representation of atomic states is therefore storing a distribution of all atomic states with the ions charge state in each macro-ion. Unfortunately this is not currently practical due to the large amount of memory an atomic state distribution occupies, see below for a detailed discussion.
Based on this a practical representation of atomic states requires

- less memory per macro particle than a full distribution
- a representation of the present distribution of atomic states
- a macro particle bound atomic state

To reduce the memory per macro particle we can smear the atomic state distribution over several macro particles, up to the extreme of only storing one single atomic state in each macro particle, as we will assume from now on. This does reduce the memory per particle but does not reduce the memory per cell, since all present atomic states still need to be represented, meaning a reduction of memory per macro particle is accompanied by a increase of the number of macro ions per cell. It even increases the memory required per cell since each macro particle stores additional data. This can be offset if the PIC-cell size and atomic states distribution cell size are decoupled, only representing the complete atomic state distribution in multiples of PIC-cells. This distributes the number of macro particles required over a large number of cells, reducing the number of macro particles required per cell while still representing the complete distribution. From a physic point of view this simply averages the atomic state distribution over several PIC-cells, an acceptable compromise due to the relative small size of PIC-cells especially for solid density plasmas.
In my implementation the atomic state distribution is only represented in a super cell, an already existing tiling of constant compound cells, each consisting of 256 cells, this is configurable in PIConGPU. This covers about $\frac{1}{2}$ of the complete distribution at 20 macro particles per cell, for $n_{\max} = 4$, good enough since we expect most of these states to not be occupied. Since the

computation is parallelized on worker level in PIConGPU on such a super cell, reusing this exiting structure also allows us to retain much of the existing parallelization structures.

2.3.2.1 Memory required for an atomic state distribution

In principle an atomic state distribution is simply a histogram, with each entry corresponding to one atomic state, and the histogram value being the relative density of this atomic state. The representation of an histogram requiring the least memory is an array of the relative densities, the corresponding atomic state being accessible via the index an entry corresponds to. The total amount of memory required is dependent on both the memory required for each entry of the vector and the number atomic states, simple multiply both to get the memory required.
As a rough estimate let us assume that we want to represent at least 256 different density values. To do so we need at least 8 bit of memory or 1 byte, this value was choosen since 1 byte is the typical minimum memory size a variable not boolean. Thus the total amount of memory required for a atomic state distribution is simply how many atomic states there are in this distribution in bytes.
Modeling the bare minimum of 4 atomic levels, $n_{\max} = 4$, there are about 10‘000 different atomic states, depending on atomic number in question, resulting in about 10‘000 bytes of memory required to represent the complete distribution.
A bare bone macro particle requires about 25 bytes of memory in comparison.

Typical minimal memory required per macro particle:

$$\begin{aligned} \text{1 macro particle} &\approx \underbrace{3 \cdot 4 \text{ byte}}_{\text{(position, 32 bit float)}} + \underbrace{3 \cdot 4 \text{ byte}}_{\text{(momentum, 32 bit float)}} \\ &+ \underbrace{1 \text{ byte}}_{\text{(charge, 8 bit integer)}} \\ \text{1 macro particle} &\approx 25 \text{ byte} \end{aligned}$$

Additional memory is required for ionization and particle tracking.

While the complete atomic state distribution is distributed, according to charge state, over several macro particles, each macro particle would have to store at least the partial atomic state distribution corresponding to its charge state, rough estimate $\frac{1}{Z}$, with $Z \leq 82$, even this reduced distribution would increase substantially the memory required per macro particle, for all but the highest charge states. Direct native representation of the atomic state distribution in each macro particle is therefore not currently feasible, due to the large amount of memory required.

2.3.2.2 Additional ideas and dead ends

- Smart histograms

  The memory required to store an atomic distribution can in principle be reduced by only storing relative densities of atomic states actually present, but doing so requires us to also store which atomic states are present. One possible solution to this is storing two arrays on containing the data itself and a second containing the index each array element corresponds to.

  As was detailed in chapter 2.3.1 an index variable will require more memory than the actual value being stored for every ion above lithium, $Z = 3$, $n_{\max} = 4$ see table 2.2. Significant memory saving are therefore unlikely for atomic state distributions containing more than a handful of atomic states. The expected typical number of atomic states present changes with charge state and excitation of ions, but is expected to be relatively large, making such an approach less useful.

- Description as a field

  The approximate 10s of kilobytes of memory required for an atomic state distribution is comparable to the memory required for all macro particles in a cell, typical expected maximum of about ≈13 kilobytes of memory per cell in macro particles without atomic physics.

  Assuming titanium, $Z = 22$, and corresponding electrons only.

  $$\begin{aligned} 1 \text{ cell} &\leq 22 \cdot 1 \text{ macro-ion} + 22 \cdot 22 \text{ macro-electrons} \leq 506 \cdot 25 \text{ byte} \\ 1 \text{ cell} &= 12650 \text{ byte} \end{aligned}$$

  It therefore would in principle be possible to store one atomic state distribution per cell as a vector field $\vec{n}$, each vector component holding the local density of an atomic state, as has been done before, for example in scfly [3].

  Based on this one might be tempted to bind the atomic state storage to cells instead of macro particles. While seeming equivalent at first, this approach creates problems on closer inspection, due to the atomic state not being particle bound.

  While a convenient macroscopic quantity, this vector field description is not fundamental in PIC. This becomes especially apparent when ion flows inside plasmas are considered in combination with a discretized not moving vector field description.

  Every flow of ions is coupled to an equivalent flow of atomic states, due to the atomic states bound to these. To represent this flow it becomes necessary to move a portion of the density of the atomic state distribution of one cell to another as ions move between these cells. Since we did not attach the atomic state to specific particles we can not move the exact part of the atomic state distribution these particles would have and are

instead forced to use an average of the atomic states of all ions with one specific charge state at a time. Therefore we essentially average every time step over all ions of one charge state atomic states. We can not describe a history of atomic states following the ions these states are actually physically attached to.

But especially in laser acceleration simulations distinguishing between the atomic states of slow moving bulk ions and fast moving accelerated ions this is important. It is therefore not a good idea to use a vector field approach if actual trajectories of particles are available, an insight this author also has needed quite a bit of time to realize. In plasma simulations where trajectories are not directly available such as MHD-simulations, gradient flows can be used to mitigate this, but they would likely introduce further complexity and should therefore be avoided in coupling atomic states to PIC.

3 Atomic State Dynamics

While there are several existing implementations of algorithms for solving for the atomic state distribution, such as flyCHK[3], scFly[3] and the Los Alamos Code Suite[1], most of these are not suitable for our purpose since they require assumptions not valid in our case, require too much memory or are not easily parallelizable. The latter two being especially important if PIC-Simulations are not to become too computationally demanding with the addition of modeling atomic physics.
We therefore need to derive a new atomic rate equation solver, better suited for the coupling to PIC-simulations. To be suitable such an algorithm has to fulfill two conditions.

- The memory an algorithm requires has to fit into the limited GPU memory to avoid very long access times
- The algorithm must scale very well with parallelization, since GPU architecture are optimized for massive parallelism

In addition, the coupling of an algorithm to PIC-Simulations should not increase the time per PIC time step required massively, my goal being that a single simulation can be calculated in a few days on HPC Clusters.
I will start with the analysis of several already existing algorithms solving for the atomic state population and discuss in this context

- Which plasma conditions they assume.
- How much additional memory they require over just representing the atomic population vector.
- How they can be parallelized.

Based on this analysis we will then derive an optimized new suitable algorithm.

3.1 Atomic Rate Equation

The time evolution of atomic states distributions is described by the rate equation, which, for the benefit of the reader, will briefly be derived here.

As the basis of our description assume some finite set of atomic states $\{i\}$, which specific set is not important here. For each of these we define a local number density n_i of ions with the atomic state i.

Abstracting from the specific set of atomic states used, we group the the atomic state number densities n_i into a vector $\vec{n} = (n_1, n_2, \dots)^{\mathrm{T}}$, with each vector component corresponding to one specific atomic state. This is the atomic state population density vector or short atomic population vector [1], a basic quantity of atomic physics in plasmas.

Ions may change their atomic state, transitioning from its previous state i to a new atomic state j. This process is probabilistic in nature, with each individual ion having a probability per time to change to a given new atomic state. Averaging over a large number of ions, the rate at which the average state shifts from i to j can be described by the average rate $R_{i\to j}$, which is the equivalent to the average rate with which an ion of atomic state i changes to the atomic state j [1]. This rate of change is transition specific, but we are interested in the general change of the atomic population vector, $\frac{d}{dt}\vec{n}$, due to transitions, since this is our fundamental quantity.

The increase of number density of a given atomic state j due to the transition $i \to j$ can be calculated as,

$$\left(\frac{d}{dt}n_j\right)_{i\to j} = R_{i\to j} \cdot n_i \tag{3.1}$$

We of course also must include the reverse transition reducing the number density of the atomic state j.

$$\left(\frac{d}{dt}n_j\right)_{j\to i} = -R_{j\to i} \cdot n_j \tag{3.2}$$

To get the general rate of change $\frac{d}{dt}n_j$ due to atomic transitions we must consider all possible transitions leading to and from a given atomic state j, resulting in,

$$\frac{d}{dt}n_j = \sum_{i\in\{i\}/\{j\}} (R_{i\to j} \cdot n_i) + \left(-\left(\sum_{i\in n/\{j\}} R_{j\to i}\right) \cdot n_j\right) \tag{3.3}$$

This equation has the structure of matrix multiplication and such can be rewritten using $R_{jj} = -\sum_{i\in\{n\}/\{j\}} R_{j\to i}$ and $R_{ji} = R_{i\to j}$to,

$$\frac{d}{dt}n_j = \sum_{i\in\{i\}/\{j\}} (R_{i\to j} \cdot n_i) + R_{jj} \cdot n_j = \sum_{i\in\{i\}} (R_{ji} \cdot n_i) \tag{3.4}$$

$$\boxed{\frac{d}{dt}\vec{n} = R \cdot \vec{n}} \tag{3.5}$$

This is the so called rate equation of atomic physics[1], or atomic rate equation in short.

3.2 Rate matrix

The rate matrix describes the total rate of transition from one atomic state to another. Its elements do not describe monolithic transitions, but rather a compound rate built from several separate physical processes rates starting with the same atomic state and ending with the same state[1].
The fundamental process that make up the rate matrix elements can be grouped into two major groups,

- Spontaneous processes

  Spontaneous process are not directly caused by a singular trigger, instead they are purely probabilistic and may happen at every point of time.

  This does not mean that they are not be caused by an external influence, only that the cause must be practical constant on the atomic physics process time scale. Examples of such are process are deexcitations due to slow external fields perturbing the atomic potential, auto ionization of electrons due to coulomb wall tunneling and deexcitations due to atomic state mixture.

  Since these process feature no interaction partner whose energy may be reduced and energy is conserved, they never increase the total energy of an ion. The rate of spontaneous process by definition, only depends on the atomic structure of an ion and/or continuum interactions, allowing us easily tabulate them, due to the low number of parameters and correspondingly small memory footprint of these tables[15].

- Interaction processes

  Interaction process in contrast are triggered by a singular external interaction, causing an ion to change its atomic state. Since interaction process by definition only happen if an interaction takes place, they are dependent on the interaction frequency ω_I.

  With the interaction frequency of a stationary ion and a flow of uniformly distributed interaction partner of density f_I, all moving with the velocity $\vec{v}_\mathrm{rel}$ being,

$$\dot{V}_\mathrm{I} = \sigma \cdot \vec{v}_\mathrm{rel} \tag{3.6}$$
$$\Rightarrow \omega_\mathrm{I} = f_\mathrm{I} \cdot \dot{V}_\mathrm{I} \tag{3.7}$$

  $\dot{V}_\mathrm{I}$ ... interaction volume per time
  σ ... cross section
  $\vec{v}_\mathrm{rel}$ ... relative velocity of interaction partners
  f_I ... interaction partner density

Based on this we can derive the local rate for a given interaction process p, using the interaction partner's and local spectrum[1].

$$\boxed{R_{i\to j,p}(\vec{r},t) = \int_E \sigma_{i\to j,p}(E) \cdot |\vec{v}_{\text{rel}}(E,\vec{r},t)| \cdot f(E,\vec{r},t)dE} \tag{3.8}$$

$R_{i\to j,p}(\vec{r})$	...	local average rate of the atomic process p, from the atomic stat i to state j
E	...	energy of the interaction partner
$\sigma_{i\to j,p}(E)$	...	cross section of the process p from the atomic state i to the state j
$\vec{v}_{\text{rel}}$	...	relative velocity of the interaction partner and the ion
$f(E)$	...	spectral density of the interaction partner
$\vec{r}$	...	position
t	...	time

For electrons and using the assumption of ions to moving much slower than electrons, this takes the form[1],

$$\boxed{R_{i\to j,p}(\vec{r},t) = \int_0^{E_{\max}} \sigma_{i\to j,p}(E) \cdot |\vec{v}_e(E)| \cdot f(E,\vec{r},t)dE} \tag{3.9}$$

$\vec{v}_e$ ... electron velocity

If the interaction partner is a photon instead, equation 3.8 takes the form[1],

$$\boxed{R_{i\to j,p}(\vec{r},t) = \int \sigma_{i\to j,p}(\nu) \cdot I(\nu,\vec{r},t) \cdot \frac{1}{h\nu}d\nu} \tag{3.10}$$

ν	...	frequency of radiation
h	...	Planck constant
I	...	spectral radiation intensity

The dependence of interaction rates on the interaction partners spectrum, makes tabulation of interaction rates for general spectra unfeasible, due to the high number of parameters necessary to describe everything but the easiest spectra. In contrast to spontaneous process, interaction process may increase as well as decrease an ions energy, since energy conservation may be satisfied by changing the interaction partner's energy.

3.2.1 Rate calculation in PIC simulations

- Spontaneous processes

  For spontaneous processes the rate may be provided directly as an input file to PIC-simulation. This input file must be stored in device memory, due to its large size of up

to several MB, but small subsection may be loaded into shared memory one at a time to reduce access latencies.

- Electron interaction

  For electron interactions instead of rates, only the oscillator strength of each transition ($i \rightarrow j$) can be provided, due to their dependence on local plasma conditions. These should be stored in device memory, again due to the relatively large size, with subsection being loaded into shared memory. The cross sections can then be calculated for each energy bin of the histogram using the semi empirical cross section algorithms of FlyCHK and scFly[3], based on the transition's oscillator strength.

  This is done only on the super cell level since resolution of the rate matrix should be the same as the atomic state distribution resolution, which is limited to one super cell due to memory constrains, see chapter 2.3 for more information.

- Radiation interactions

  Radiation interactions can not currently be modeled, since the spectral intensity required is not practically available in PIC-simulations.

  PIC-simulations do not resolve the frequency of radiation fields, only current overall field strengths, which means that the local spectral intensity is not directly available. Spectral information would in theory be available using Fourier transformations, but Fourier transformations are computational very expensive and therefore unfeasible.

  In the future high frequency radiation, x-ray and up, process may be added relatively easily, modeled using binary collision of macro-photons and macro-ions, with binary collision algorithms currently being implemented in PIConGPU and macro-photons already implemented. Lower frequencies process are more problematic, due to the presence of the driving laser in this frequency band. This laser is modeled in PIC as a time dependent field, preventing us from adding macro-photons representing the same radiation without closely coupling them to the radiation field itself. How such a coupling would work, or if it even is possible, is an open question. Alternatively a continuum description might be possible, but this too is an area of ongoing research.

3.3 Solving the atomic rate equation

At a glance the atomic rate equation(3.5) seems to be rather simple to solve, appearing to be a first order linear differential equation only, but in this case appearances are deceiving. Neither is the rate Matrix R in general constant in time, which would make the equation analytically solvable, nor is it strictly linear, since R does in fact indirectly depend on the

atomic population vector $\vec{n}$ as will be discussed in section 4. As such, solving the general atomic rate equation is not trivial.

There are different general approaches for solvers of the rate equation, each based on different assumptions of plasma conditions, we will be discussing these approaches in the following section.

3.3.1 Solver Approaches

This section is intended to give the reader an overview of some standard approaches and introduce several important techniques. While many of the different approaches will turn out to *not* be applicable to the general case, they are useful for validation, comparisons and optimization and are therefore included here. Experienced readers can skip directly to the section detailing time dependent solvers, section 3.3.1.6.

We will start with the most restrictive plasma condition assumptions and piecewise relax them until we reach assumptions compatible with our goals.

3.3.1.1 Global thermal equilibrium, TE

While PIC-Simulation are rarely done under the assumption of TE, TE conditions allow to solve for the atomic population vector analytically, making it very useful as an analytical solution to compare to and often used as the first step of validation of solvers. In addition TE solvers require comparatively few computational resources, making them the least expensive option, both in memory and compute time, to model atomic physics, even if severely constrained by its assumptions. Since standard implementations of this algorithm are sequential we will also derive a parallelized version to be used in validations of the later derived more general solvers.

TE solver may also be generalized to local thermal equilibrium conditions and as such I am using this section to introduce the basic algorithm first, before generalizing it in the next section.

In the case of a global equilibrium, it is possible calculate the atomic population vector without using the atomic rate equation (3.5). Instead we can describe the plasma's atomic states as an ensemble of internal states of a system coupled to thermal bath of known temperature T and use the atomic states' energy and multiplicity to solve for the stable state atomic population vector using the canonical ensemble. Doing so [1] gives us the relative abundance, ratio between states abundances, between neighboring *charge* states using the Saha equation[1]

$$\frac{N_{i+1}}{N_i} \cdot n_e = 2 \cdot \left(\frac{2\pi m_e k_\mathrm{B} T}{h^2} \right)^{\frac{3}{2}} \cdot \frac{Z_{i+1}(T)}{Z_i(T)} \cdot e^{-\Delta E_i/(k_\mathrm{B}T)} \tag{3.11}$$

N_i ... Number of ions with the charge state i
n_e ... Electron number density
T ... Temperature of the plasma
Z_i ... partition function of the charge state i
ΔE_i ... Ionization energy of charge state i
k_B ... Boltzman constant

with [1]

$$Z_i = \sum_{j \in \{b\}_i} g_j \cdot e^{-(E_j - E_i^0)/(k_\mathrm{B} T)} \tag{3.12}$$

$\{b\}_i$ ... set of bound state of the charge state i
g_j ... statistical weight of the bound state j
E_i^0 ... ground state energy of the charge state i
E_j ... energy of bound state j

and the relative abundance of bound *atomic* states of a given *charge* state[1],

$$\frac{n_i}{n_j} = \frac{g_i}{g_j} e^{-\frac{\left(E_i - E_j\right)}{k_\mathrm{B} \cdot T}} \tag{3.13}$$

g_i ... statistical weight of the bound state i
E_i ... energy of bound state i

These equations only specify the relative abundance, meaning that for a given solution of the atomic population vector $\vec{n}$, multiples of it $\vec{n}' = \lambda \cdot \vec{n}$ are also solutions. This ambiguity can be resolved using the conservation of the number of ions, by specifying the total ion density n_I. Due to this conservation, the sum of all, atomic- or charge-, state densities is always equal to the total ion density[1], with only one scaling factor λ satisfying this condition, proof by contradiction.

$$\sum_i n_i =: n_I \tag{3.14}$$

Instead of directly specifying the total ion density, it is also possible to use the electron density n_e and the charge neutrality of an equilibrium plasma and the charge of each atomic state to substitute the ion density with the electron density[1].

$$n_e \stackrel{!}{=} \sum_i \left(q_i \cdot n_i\right) \tag{3.15}$$

q_i ... charge of atomic state i

Using these equations, we can calculate the atomic population vector if,

- a plasma temperature T is given

- a set of bound atomic states $i \in \{b\}$ is assumed, with for each atomic state known
  - energy E_i
  - statistical weights g_i
- and the electron density n_e is known.

The general idea is to use the above equations to the get the relative abundance $\frac{n_i}{n_j}$ of all atomic states i relative to some reference atomic states density $n_{j=\mathrm{r}}$. We can use the above relations to fill the atomic population vector relative to the density of the reference atomic state,

$$\begin{pmatrix} \vdots \\ \frac{n_i}{n_\mathrm{r}} \\ \vdots \end{pmatrix} \tag{3.16}$$

and then scale the resulting vector vector such that it fulfills equation 3.15, to get the atomic population vector $\vec{n}$.

$$\vec{n} = \lambda \cdot \begin{pmatrix} \vdots \\ \frac{n_i}{n_\mathrm{r}} \\ \vdots \end{pmatrix} \tag{3.17}$$

This algorithm is described in more detail in the following section. This is done to provide complete description of this algorithm ready for implementation and provide the necessary fundamentals to discuss its characteristics.

3.3.1.2 Solving for the atomic population vector in TE conditions

To be able to follow the approach previously described we need the relative abundance of all atomic states relative to some arbitrary chosen reference atomic state. These are unfortunately not directly available.

The equations 3.11 and 3.13 allow us to calculate the abundance of a given, either charge- or atomic-, state in relation to the abundance of one other reference state.

The equation 3.13 is more flexible, since it allows us to choose the reference state, but it has to be noted that equation 3.13 is only valid for comparison between atomic states with identical charge state, due to ionization also changing the electron partition function, and thus limiting the possible reference states.

Equation 3.11 in contrast, always uses the next lower charge state for reference.

As such we can not get the relative abundance between every pair of atomic states directly. Instead, we must first calculate the relative abundance of all *charge* states $\frac{N_i}{N_j}$ and then use the charge state's internal relative abundance of its *atomic* states from equation 3.13, to calculate the global relative abundance of each *atomic* state.

To do so we choose without loss of generality a reference *charge* state N_r. We can calculate the relative abundance of the next higher and/or lower charge state using equation 3.11 directly,

$$3.11 \Rightarrow \frac{N_{\mathrm{r}+1}}{N_\mathrm{r}} \tag{3.18}$$

$$3.11 \Rightarrow \frac{N_\mathrm{r}}{N_{\mathrm{r}-1}} = \frac{1}{\frac{N_{\mathrm{r}-1}}{N_\mathrm{r}}} \Rightarrow \frac{N_{\mathrm{r}-1}}{N_\mathrm{r}} \tag{3.19}$$

Using these values as a known previous value in addition to equation 3.11 we can calculate the abundance of the next higher and lower charge state relative to the reference state .

$$\frac{N_{\mathrm{r}+2}}{N_\mathrm{r}} = \frac{N_{\mathrm{r}+2}}{N_{\mathrm{r}+1}} \cdot \frac{N_{\mathrm{r}+1}}{N_\mathrm{r}} \tag{3.20}$$

$$\frac{N_{\mathrm{r}-2}}{N_\mathrm{r}} = \frac{N_{\mathrm{r}-2}}{N_{\mathrm{r}-1}} \cdot \frac{N_{\mathrm{r}-1}}{N_\mathrm{r}} = \frac{1}{\frac{N_i}{N_{i-1}}} \cdot \frac{N_i}{N_R} \tag{3.21}$$

This can be generalized to all other charge states, allowing us to calculate the abundance of every charge state relative to the reference state, $\frac{N_{i+1}}{N_\mathrm{r}}$, using the known previous states abundance relative to the reference state, $\frac{N_i}{N_\mathrm{r}}$.

$$\frac{N_{i+1}}{N_\mathrm{r}} = \frac{N_{i+1}}{N_i} \cdot \frac{N_i}{N_\mathrm{r}} \tag{3.22}$$

$$\frac{N_{i-1}}{N_\mathrm{r}} = \frac{N_{i-1}}{N_i} \cdot \frac{N_i}{N_\mathrm{r}} = \frac{1}{\frac{N_i}{N_{i-1}}} \cdot \frac{N_i}{N_R} \tag{3.23}$$

Resulting in all charge states' abundances relative to the reference state being known.

$$\begin{pmatrix} \vdots \\ \frac{N_{\mathrm{r}+1}}{N_\mathrm{r}} \\ \frac{N_\mathrm{r}}{N_\mathrm{r}} \\ \frac{N_{\mathrm{r}-1}}{N_\mathrm{r}} \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ \frac{N_{\mathrm{r}+1}}{N_\mathrm{r}} \\ 1 \\ \frac{N_{\mathrm{r}-1}}{N_\mathrm{r}} \\ \vdots \end{pmatrix} =: \frac{1}{N_\mathrm{r}} \cdot \vec{N} \tag{3.24}$$

$\vec{N}$ ... charge state population vector
N_i ... density of charge state i
N_r ... density of reference charge state

Which we then rescale to match a given electron density according to 3.15.

$$\sum_i \left(q_i \cdot \frac{N_i}{N_\mathrm{r}} \right) = \frac{1}{N_\mathrm{r}} \cdot \left(\sum_i \left(q_i \cdot N_i \right) \right) \overset{3.15}{=} \frac{1}{N_\mathrm{r}} \cdot n_e \tag{3.25}$$

$$\Rightarrow N_\mathrm{r} = \frac{n_e}{\sum_i \left(q_i \cdot \frac{N_i}{N_\mathrm{r}} \right)} \tag{3.26}$$

Giving us the charge state population vector $\vec{N}$ using equations 3.26 and 3.24.

$$\vec{N} = \frac{N_\mathrm{r}}{N_\mathrm{r}} \cdot \vec{N} = N_\mathrm{r} \cdot \left(\frac{1}{N_\mathrm{r}} \cdot \vec{N} \right) \tag{3.27}$$

$$\boxed{\vec{N} = \frac{n_e}{\sum_i \left(q_i \cdot \frac{N_i}{N_\mathrm{r}} \right)} \cdot \begin{pmatrix} \vdots \\ \frac{N_{\mathrm{r}+1}}{N_\mathrm{r}} \\ 1 \\ \frac{N_{\mathrm{r}-1}}{N_\mathrm{r}} \\ \vdots \end{pmatrix}} \tag{3.28}$$

To get the atomic population vector we basically repeat this procedure for all atomic states of a each charge state.

1. choose an arbitrary reference atomic state r out of all atomic states of the current charge state j, $\mathrm{r} \in \{b\}_j$.

2. use equation 3.13 to calculate the relative abundance of all other atomic states of this charge state, $i \in \{b\}_j$, relative to the reference atomic state.

$$3.13 \Rightarrow \frac{n_i}{n_\mathrm{r}} \tag{3.29}$$

3. compile them into a sub vector, chosen sub set of components arranged into a vector, of the actual atomic population vector

$$\begin{pmatrix} \vdots \\ \frac{n_i}{n_\mathrm{r}} \\ \vdots \end{pmatrix} = \frac{1}{n_\mathrm{r}} \cdot \vec{n}_j \tag{3.30}$$

$\vec{n}_j$ ... charge state j sub vector of the atomic population vector,
vector containing the densities of all atomic states with the charge state j

4. scale them such that the following equation is fulfilled.

$$\sum_{i \in \{b\}_j} n_i := N_j \tag{3.31}$$

$$\sum_{i \in \{b\}_j} \frac{n_i}{n_\mathrm{r}} = \frac{1}{n_\mathrm{r}} \cdot \left(\sum_{i \in \{b\}_j} n_i \right) = \frac{1}{n_\mathrm{r}} \cdot N_j \tag{3.32}$$

$$\Rightarrow n_\mathrm{r} = \frac{N_j}{\sum_{i \in \{b\}_j} \frac{n_i}{n_\mathrm{r}}} \tag{3.33}$$

Once all atomic population sub vectors are known the atomic population vector can be assembled from them relatively easily.
Solving for the atomic population vector this way allows analytically calculating the atomic population vector and requires only one memory location for each charge state in addition to the storage of the atomic population vector itself.

3.3.1.3 Parallelizing the Solver

So far we used a sequential algorithm to calculate the atomic population vector $\vec{n}$. For large numbers of atomic states, $\geq 10^6$, a sequential algorithm is no longer feasible as will be briefly estimated below.

One sequential operation takes at least one cycle to execute, resulting in the lower limit of approximately 0.25 ns per multiplication on modern CPUs with a 4 GHz clock frequency.

$$1 \text{ multplication} \stackrel{\wedge}{=} 1 \text{ cycle} = \frac{1}{f_{\text{CPU}}} = \frac{1}{4\text{GHz}} = \frac{1}{4} \cdot 10^{-9} \text{ s} \tag{3.34}$$

Calculating the relative abundance requires at least one operation per atomic state. A CPU can execute at most one sequential operation per clock cycle, which for 4 million atomic states would take at least 1 ms.

$$\Rightarrow 4 \cdot 10^6 \cdot \frac{1}{4} \cdot 10^{-9}\text{s} = 10^{-3}\text{s} \tag{3.35}$$

A time step of a PIConGPU simulations usually takes about 1 ms for standard non demanding simulations, which is comparable to our previous lower limit for $4 \cdot 10^6$ atomic states.
In reality some calculations in complex instruction sets, CISC architectures, require more than one cycle to complete and performance is also highly influenced by data locality and utilization of fetch-decode-compute pipelines[7]. Even if perfect utilization is assumed how many cycles are required for a given instruction is highly dependent on the architecture of a given CPU or GPU, with modern, ca. 2019, Intel CPUs typically requiring 1-2 cycles for multiplication operations with a latency of 3 cycles, and about 6 cycles for a division operation with a latency of 23 cycles[7]. Power functions or exponential functions require even more cycles, with the exact time required highly dependent on library implementation. Since we need to complete at least one exponential function calculation and one multiplication per atomic state, we can safely assume that sequentially calculating the atomic population vector for more than approximately 10^6 different states is going to at least double the total time required.

It is therefore necessary to parallelize the algorithm. The algorithm presented above, can be divided into 4 different sub steps.

1. calculation of charge state relative abundances
2. scaling to get charge state densities
3. calculating the relative abundances of all atomic states of all charge states
4. scaling to get the atomic state densities

The steps 1 and 3 can be fully parallelized, the steps 2 and 4 at least partially, as will be discussed in the following.

- Step 1:

  Step 1 can be parallelized on the level of charge states, since equation 3.11 is independent for all charge states. For every charge state higher than the reference we can chain equation 3.22, until only a multiplication of equation 3.11 terms remains, which can be directly calculated.

$$\frac{N_i}{N_\mathrm{r}} \overset{3.22}{=} \frac{N_i}{N_{i-1}} \cdot \frac{N_{i-1}}{N_\mathrm{r}} \overset{3.22}{=} \frac{N_i}{N_{i-1}} \cdot \frac{N_{i-1}}{N_{i-2}} \cdot \frac{N_{i-2}}{N_\mathrm{r}} = \frac{N_i}{N_{i-1}} \cdot \frac{N_{i-1}}{N_{i-2}} \cdot \ \ldots \ \cdot \frac{N_\mathrm{r}}{N_\mathrm{r}} \tag{3.36}$$

$$\boxed{\frac{N_i}{N_\mathrm{r}} = \prod_{j=\mathrm{r}+1}^{i} \frac{N_j}{N_{j-1}}} \tag{3.37}$$

  This can be improved by re substituting the original equation 3.11

$$\begin{aligned} \frac{N_i}{N_{i-1}} \cdot \frac{N_{i-1}}{N_{i-2}} \overset{3.11}{=} & \left(\frac{1}{n_e} \cdot 2 \left(\frac{2\pi m_e k_\mathrm{B} T}{h^2} \right)^{\frac{3}{2}} \cdot \frac{Z_i(T)}{\cancel{Z_{i-1}(T)}} \cdot e^{-\Delta E_i/(k_\mathrm{B}T)} \right) \\ & \cdot \left(\frac{1}{n_e} \cdot 2 \left(\frac{2\pi m_e k_\mathrm{B} T}{h^2} \right)^{\frac{3}{2}} \cdot \frac{\cancel{Z_{i-1}(T)}}{Z_{i-2}(T)} \cdot e^{-\Delta E_{i-1}/(k_\mathrm{B}T)} \right) \end{aligned} \tag{3.38}$$

$$= \left(\frac{1}{n_e} \cdot 2 \left(\frac{2\pi m_e k_\mathrm{B} T}{h^2} \right)^{\frac{3}{2}} \right)^2 \cdot \frac{Z_{i+1}(T)}{Z_{i-1}(T)} \cdot e^{-(\Delta E_i + \Delta E_{i-1})/(k_\mathrm{B}T)} \tag{3.39}$$

$$\Rightarrow \boxed{\frac{N_i}{N_\mathrm{r}} \overset{3.37}{=} \left(\frac{1}{n_e} \cdot 2 \left(\frac{2\pi m_e k_\mathrm{B} T}{h^2} \right)^{\frac{3}{2}} \right)^{i-\mathrm{r}} \cdot \frac{Z_i(T)}{Z_\mathrm{r}(T)} \cdot e^{-\left(\sum_{j=\mathrm{r}}^{i-1} \Delta E_j\right)/(k_\mathrm{B}T)}} \tag{3.40}$$

  Thereby avoiding the need to calculate intervening partition functions.

  The parallelization should not stop at the charge state level however, since the number of operations required for each charge state varies. For a naive parallelization this would lead to some workers finishing before others and all but one workers waiting on worker calculating the density of the charge state with the highest number of operations. The worst charge state i has to complete at best $\lceil \frac{Z}{2} \rceil$ steps to calculate $\frac{N_i}{N_\mathrm{r}}$, reference state

lies in the middle of all other charge states, and at worst Z steps, reference state lies at one end.

The number of steps being equal to how one has to either remove or add one electron to reach a given charge state form the reference state.

Instead operations should be more evenly distributed, thereby reducing the compute time for a given numbers of workers.

To do so, we keep the basic parallelization in charge states, but instead of letting one worker perform all operations required for one charge state on its own, we utilize the associativity of multiplications of floating point numbers to sub-divide the operations into chunks with equal number of operations. All workers are assigned one subtask initially and are assigned the next subtask upon completion, until all subtask have been completed, a concept known in PIConGPU as a for-each-functor[17]. Under ideal conditions this approximately halfs the time required again, see below for a more detailed explanation.

Assume the best case of a central reference state and without loss of generality that Z is even. According to equation 3.40, the number of multiplications and summations requiredfor a given charge state scales with the difference between charge state and reference state. For the charge states furthest away from the reference state $\frac{Z}{2}$ multiplications are required, every step closer to the reference states decreases the number of multiplications required by one until we reach the reference state. After this the number of multiplications increases by one once again until we reach once again the maximum value $\frac{Z}{2}$.

$$\begin{pmatrix} \frac{Z}{2} \\ \frac{Z}{2}-1 \\ \vdots \\ 1 \\ 0 \\ 1 \\ \vdots \\ \frac{Z}{2}-1 \\ \frac{Z}{2} \end{pmatrix} \dots \text{number of multiplications required} \tag{3.41}$$

The total number of multiplications required is therefore,

$$2 \cdot \sum_{i=1}^{\frac{Z}{2}} i \quad \overset{\text{gaus}\sum}{=} \quad \cancel{2} \cdot \frac{(\frac{Z}{2}+1) \cdot \frac{Z}{2}}{\cancel{2}} \tag{3.42}$$

which can be distributed over the original Z workers, resulting in a maximum of $\lceil \frac{Z}{4} + \frac{1}{2} \rceil$ operations per worker, or about half of the time required for the naive best case parallelization with one worker for every charge state but the reference state. For uneven Z the exact number changes but the general argument still holds true.

- Step 3:

  This step is easily parallelized naively, since equation 3.13 allows the independent direct calculation with constant number of operations.

- Step 1 and 2:

  In both of these steps we essentially first sum over a vector, calculate the scaling factor depending on this sum and then multiply all vector components with this number. The calculation of the scaling factor can not be parallelized, but the summations and factor applications can be parallelized by sub dividing the addition and multiplication in chunks of equal size. Several workers can then work in parallel on these chunks in the same way already described.

In addition to being able to parallelize the steps itself we can also change the order in which they are performed and execute some of these steps in parallel. The steps 1 and 3 are independent of all other, and as such can be executed in parallel. Only the step 2 and 4 depend on previous results, with step 2 depending on the result of step 1, and step 4 depends on the results of steps 1, 2 and 3.
As such the the execution of the steps 1 and 3 can be done in parallel, step 2 must wait on the completion of step 1 but afterwards can be done in parallel with step 3, while step 4 must be completed last and can not completed in parallel to any other step.
The parallelization does require additional memory to store intermediary results, but this memory scales with the number of workers used, something we can choose freely. The number of workers used may therefore be optimized with regards to memory and runtime available for specific hardware and physical setup.

3.3.1.4 Local thermal equilibrium conditions, LTE

If a global thermal equilibrium can not be assumed, the next step of generalization is to assume a local thermal equilibrium. The term local thermal equilibrium refers to a plasma in which the atomic physics transitions are much faster than the changes in plasma conditions, resulting in an atomic population vector which is, practically always, in a stable state for its current plasma conditions and changes as these plasma conditions change with time[1]. The plasma itself is not assumed to be in a global equilibrium, i.e. it has a time dependent evolution, but the fact that local plasma conditions change sufficiently slowly can allow the definition of a local electron temperature.

Under these assumption, it is possible to use the solver we developed for TE-conditions for every point of a grid, with the local plasma conditions used as input. This requires considerable more computational power, since instead of solving for the atomic population vector once, we now need to solve it for every time step, for every grid point. Due to the efforts invested into parallelization previously this can still be computed with reasonable performance if a small enough and comprehensive enough set of atomic states is available.
Unfortunately LTE conditions are of limited applicability in PIC-simulations, since PIC-Simulations are usually used when LTE conditions are not present. Nevertheless they are of use for validating more general solvers in more limited plasma conditions. The parallelized LTE-solver may also be used as a cheap model for atomic physics if compute resources are not available for more elaborate models.

3.3.1.5 Non-Local Thermal Equilibrium, NLTE

The next step up in complexity from local thermal equilibrium is the inclusion of non-local effects. The term NLTE, describes a plasma in which still the atomic physics process are much faster than the plasma evolution, in which long ranged interactions are present and the plasma is not in equilibrium over the interaction range.
Previously we assumed interactions with atomic states to be limited to interacting with one thermal bath for each point in space. This approximation breaks down if long ranged atomic physics interactions are present and neighboring grid points are not in equilibrium.

> Typical long ranged interactions being, propagating radiation or fast electron spectrum components.

Since plasma conditions may vary widely over space time region and long ranged interactions couple the atomic population vector to a region, the definition a single temperature becomes impossible[1]. This also prevents the use of the equations 3.11 and 3.13, since the existence of a single temperature was one of the primary assumptions used in their derivation.

Instead of using a temperature, long ranged interactions can be modeled as a local interaction field $\vec{I}(\vec{r})$, with the interaction field being generated by the entire interaction region and provided as input data[3], for details of this interaction field see section 3.2. This allows us to model interactions consistently, even if large variation of plasma conditions are present in the interaction region. It also allows us to shift retardation effects to the interaction field dynamics, where they are easier to model, see chapter 4 for details.
The local interaction field is assumed to be in equilibrium with the local atomic population vector, still based on the assumption that the plasma evolution is much slower than the atomic state dynamics, and we therefore solve for every grid point and time step, for a stable state of the atomic rate equation.
In equilibrium conditions the macroscopic atomic population vector is by definition constant

with time. This reduces the rate equation from a differential equation to a matrix equation[1], the stable state atomic rate equation, simplifying the solution considerably.

$$\frac{d}{dt}\vec{n} = 0 = R \cdot \vec{n} \qquad ;\text{with } R = R\left(\vec{I}\right) \tag{3.43}$$

The resulting system of linear equations is undetermined, since, as detailed in section 3.1, the main diagonal is not linear independent of the off-diagonal elements[1]. A unique solution therefore requires an additional boundary condition, typically the charge neutrality condition 3.15[1].
The stable state atomic rate equation can be solved using standard matrix solvers for a given interaction field and electron density at a given point[1], but special care must be taken if large, $> 10^5$, sets of atomic states are used. For large sets of atomic states both memory and compute time required can be problematic, with the memory required for a naive storage of the rate matrix growing as $\mathcal{O}(n^2)$ with the number of atomic states n, and the computation time required, typically scaling between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$[1]. Highly parallelized and optimized solvers utilizing the typical sparseness of the rate matrix to decrease both the memory and time required, are available, but tend to require specific matrix structures to assure stability[1]. Detailed discussions of the different solvers are available in the existing literature[1] and will therefore not be repeated here.

3.3.1.5.1 NLTE Solvers and PIC-simulations NLTE solvers are widely used due to their flexibility, with several existing implementations, for example flyCHK, scFly and the Los Alamos suite of atomic physics codes[1]. In contrast to LTE solvers, they can *in principle* be used to model atomic physics in PIC-simulations, due to their ability to model atomic physics with non thermal spectra and high gradients in plasma conditions. In addition, the matrix operations used by NLTE solvers are highly parallelizeable[1] and widely hardware accelerated, making their use computationally tractable. NLTE solvers are also by design highly modular, allowing developers to tune implementations for the desired accuracy or speed.
Unfortunately the fundamental assumption of equilibrium prevents their use in plasmas with the very fast dynamics commonly encountered in laser generated plasmas. This also limits their usefulness in PIC-simulations, which are often used to simulate laser generated plasmas.
A secondary limitation NLTE solver suffer from, is the necessity to provide accurate interaction fields as input data. These must be supplied by the user for every point and time of the plasma for which to solve the atomic rate equation. The input data NLTE solver require is usually only available as the output of another simulation for everything but boundary regions.
It has to be noted that, while NLTE solver will calculate the correct atomic population vector for the given electron and radiation spectrum, in reality the radiation and electron spectrum is also dependent on the atomic population vector itself. The propagation of radiation and the interaction of electrons with ions are both dependent on the atomic state of ions. The

plasma simulation therefore needs the current atomic state to correctly predict the electron and radiation spectra for every point in space, meaning that the atomic physics solver must be closely integrated with the plasma simulation.
Integrating atomic physics solvers is also necessary due to hardware limitations, if non thermal spectra are to be used. If thermal spectra can be assumed, input data are comperatively small, specifying only the electron and radiation temperature field in addition to the local electron density.
In the case of non-thermal spectra, the locally resolved electron and radiation spectra must be provided directly as input data. This requires much more memory, since instead of 2 values, electron and ion temperature, in addition to the local electron density, per point and time, we now have to store an entire distribution of unknown shape. This may requires 100s of different values per point and time, massively increasing the memory required to store the input data set.

Storing a single input data set for one time step, assuming:

- a 3 dimensional cube
- subdivided into 100^3 cells
- with a histograms with 128 bins for every cell for both the radiation- and electron spectrum
- using 32 bit floats for data storage

already requires about 1GB of memory.

$$\underbrace{100^3}_{\text{Num. cells}} \cdot \underbrace{128}_{\substack{\text{Num. Bins} \\ \text{per histogram}}} \cdot \underbrace{4}_{\substack{\text{4 Bytes} \\ \text{per 32bit float}}} \cdot \underbrace{2}_{\substack{\text{radiation and} \\ \text{electron spectrum}}}$$
$$= 1,024 \cdot 10^9 \text{ Byte}$$

Higher resolutions increase this quickly, with typical PIC-simulations using far greater number of cells.

This creates several problems for the use of NLTE solver in the post processing of plasma simulations.
If a NLTE solver is used in post processing only, the entire input data set must be written to memory by the plasma simulation used. While the total memory required to do so is commonly available in HPC clusters, transfer speeds become problematic.

Modern DDR4-3200 RAM has a maximum speed of about 25,6 $\frac{\text{GB}}{\text{s}}$ [18] per memory channel. With modern compute nodes having at best 8 channels, AMD EPYC[19], it would require about 5 ms to write 1 GB of data to RAM compared to about 1 ms of calculation time per PIConGPU time step. Therefore writing the small example input data set to memory requires about 5 times more time than the calculation itself.

The maximum amount of memory available is also limited, with modern compute nodes by design having at maximum 1-2 TB of RAM, GPU nodes are typically equipped with significant less memory than this maximum. Additionally not the complete RAM may be used since the plasma simulation itself also requires both bandwidth and memory. We can therefore at best hope to store a quite limited number of input data time steps. More memory is available in the file system, but access is typically several magnitudes slower, exacerbating the write problems described above.

Thus post processing use of NLTE solver is not practical, instead an NLTE solver must be run in parallel to plasma simulations alllowing us to store only the current input data set of every time step in memory.

3.3.1.6 Time dependent, TD

To be able to correctly model atomic physics in laser generated plasma, NLTE solver are not sufficient. We can reuse the basic framework used by NLTE solvers, but instead of assuming an equilibrium we solve the atomic rate equation time dependently in every grid point.

$$\frac{d}{dt}\vec{n}(\vec{r},t) = R(\vec{r},t)\cdot\vec{n}(\vec{r},t) \tag{3.5}$$

This requires a different boundary condition, instead of being able to use the electron density, we must specify the initial atomic population vector for every grid point.

The central idea of most time dependent solvers is stepwise approximate integration, with the rate matrix R assumed to be constant over the length of one step Δt. The integration itself can be done analytically, since the atomic rate equation with a constant rate matrix can be solved analytically, the fundamental solution taking the form

$$\vec{n}(t) = \sum_i \left(a_i \cdot \vec{\lambda}_i \cdot e^{\lambda_i \cdot t}\right) \tag{3.44}$$

λ_i ... i-th eigenvalue of the matrix R
$\vec{\lambda}_i$ ... i-th eigenvector of the matrix R
a_i ... i-th fitting constant, determined by starting value $\vec{n}_0$

Proof by insertion into the rate equation.

This is still an approximate solution, since the rate matrix is not independent of the atomic population vector.

Unfortunately determining the eigenvalues and eigenvectors of a matrix is both computationally and memory wise expensive for large matrices. Instead explicit or implicit solvers of varying order can be used[1] to reduce the computational demand considerably. These are described extensively in literature, for example in [16] and will therefore not be repeated here.

TD solver are in principle well suited for modeling atomic physics in PIC-simulations. They share NLTE solver's ability to model atomic physics under varied plasma conditions and are additionally able to model the transient effects of laser generated plasmas.

TD unfortunately also share the difficulties of NLTE solvers with regard to input data detailed in the previous section, since both share the same interface with plasma simulations.

3.3.2 TD solver for direct inclusion in PIC-Simulations

All solvers in the previous chapter operated on the atomic population vector $\vec{n}$ as their the basic quantity. This is natural if the solver developed separated from a plasma simulation, but as detailed in the previous section this is neither desirable, due to the interction between atomic states and PIC-simulation nor practical, due to memory write speed limits and total memory limits.

Since atomic rate solvers have to be tightly integrated into the PIC-simulation anyway, the use of the atomic population vector is no longer natural. As described in chapter 2.3 atomic states should not be represented as a atomic population vector, but a rather as atomic states sampled by macro-ions. The atomic population vector is therefore in PIC-simulations not an elementary quantity but rather a derived quantity, created by binning all macro particles weights according to their atomic state. We therefore need a new type of TD solver that may be applied directly to a single macro-ion.

These new solvers are still based on the same fundamental description of standard TD solvers, since macro-ions in a super cell only sample the corresponding atomic population vector.

But instead of being applied to the entire atomic population vector of a super cell, we use the distributivity of the matrix multiplication, assuming the rate matrix to be practically constant over a single super cell, to sub-divide the atomic population vector into several summands, with each summand representing one macro-ion sample of the super cell $\vec{n}_i$.

$$\frac{d}{dt}\vec{n} = R \cdot \vec{n} = R \cdot \sum_k \vec{n}_k = \sum_k (R \cdot \vec{n}_k) \tag{3.45}$$

$\vec{n}_k$ ... k-th sample of the atomic population vector $\vec{n}$

This allows us to also sub-divide the change of the atomic population vector.

$$\frac{d}{dt}\vec{n}_k := R \cdot \vec{n}_k \tag{3.46}$$

Since each macro-ion only stores a single atomic state, this is equivalent to the a single column

of the rate matrix.

$$\frac{d}{dt}\vec{n}_k = R \cdot \vec{n}_k = \begin{pmatrix} R_{11} & R_{12} & \dots & R_{1i} & \dots \\ R_{21} & R_{22} & \dots & R_{2i} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ R_{i1} & R_{i2} & \dots & R_{ii} & \dots \\ R_{(i+1)1} & R_{(i+1)2} & \dots & R_{(i+1)i} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ \vdots \\ n_i \\ 0 \\ \vdots \end{pmatrix} = \begin{pmatrix} R_{1i} \cdot n_i \\ R_{2i} \cdot n_i \\ \vdots \\ R_{ii} \cdot n_i \\ R_{(i+1)i} \cdot n_i \\ \vdots \end{pmatrix} \tag{3.47}$$

Which can be rewritten using $R_{jj} = -\sum_{i \in n/\{j\}} R_{ij}$.

$$\boxed{\frac{d}{dt}\vec{n}_k = \begin{pmatrix} R_{1i} \\ R_{2i} \\ \vdots \\ R_{(i-1)i} \\ -\sum_{j \in \{n\}/\{i\}} R_{ji} \\ R_{(i+1)i} \\ \vdots \end{pmatrix} \cdot n_i} \tag{3.48}$$

This single vector essentially describes how a macro-ion sample of the atomic population vector would change with time.

While directly equivalent to the original rate equation(3.5), this can not be applied to the a single macro-ion, since one atomic state is converted to many states and the macro-ion itself can only ever have a single atomic state at all times.

This problem can be solved by randomly choosing one new state for each macro particle such that on average the rate equation is reproduced, a so called Monte Carlo algorithm. We will discuss two different algorithms based on this idea in the following two sections, first the Monte-Carlo TD solver for which we will derive the basic description we are going to use, and secondly the further optimized Markov-chain TD solver.

3.3.2.1 Monte-Carlo TD solver

To construct a Monte-Carlo solver three things must be defined, what will be randomly changed, how do we change them and lastly with which probability will we change to a specific value.

The first question has already been answered in the previous section, we want to select a new random atomic state form the set of all atomic states, only the last two remain to be answered. We will start by deriving the probability for changing to a given new state, while the second part of this section we will describe a pseudo code implementation of the basic solver to answer

how we will change the atomic state. Lastly we will discuss the presented algorithm in the context of including it in PIC-Simulations.

3.3.2.1.1 Monte-Carlo probability We want to model the change of one sample $\vec{n}_j$of the atomic population vector over a given time step Δt. Using equation 3.48 this can approximated in first order explicit euler method(1768) as,

$$\vec{n}_j(t_0 + \Delta t) = \frac{d}{dt}\vec{n}_j(t_0) \cdot \Delta t + \vec{n}_j(t_0) \tag{3.49}$$

Using equations 3.48 this can be rewritten.

$$\vec{n}(t_0+\Delta t)_j = \begin{pmatrix} R_{1i} \\ \vdots \\ R_{(i-1)i} \\ -\sum_{j\in\{n\}/\{i\}} R_{ji} \\ R_{(i+1)i} \\ \vdots \end{pmatrix} \cdot n_i \cdot \Delta t + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ n_i \\ 0 \\ \vdots \end{pmatrix} = \begin{pmatrix} R_{1i} \cdot \Delta t \\ \vdots \\ R_{(i-1)i} \cdot \Delta t \\ 1 - \left(\sum_{j\in\{n\}/\{i\}} R_{ji} \cdot \Delta t\right) \\ R_{(i+1)i} \cdot \Delta t \\ \vdots \end{pmatrix} \cdot n_i \tag{3.50}$$

Based on this expected value we may now try to find probabilities $P_{i\to k}$ that reproduce this vector on average.
If we randomly choose for each macro-ion one specific state to change to with probability $P_{i\to k}$ the average atomic vector becomes,

$$\vec{n}_j(t_0 + \Delta t) = \sum_i P_{i\to k} \cdot \vec{e}_k \cdot n_i = \begin{pmatrix} \vdots \\ P_{i\to k} \\ \vdots \end{pmatrix} \cdot n_i \tag{3.51}$$

A quick comparison of equations 3.50 and 3.51 gives us the required probabilities,

$$P_{i\to k} = \begin{cases} R_{ki} \cdot \Delta t & i \neq k \\ 1 - \left(\sum_{j\in\{n\}/\{i\}} R_{ji} \cdot \Delta t\right) & i = k \end{cases} \tag{3.52}$$

To ensure that all $P_{i\to k}$ do in fact fulfill all the axiomatic properties of probabilities we must limit the time step lengths Δt, as will be shown below.

- The sum over all $P_{i\to k}$ is always equal to one.

  This is always fulfilled.

$$\sum_k P_{i\to k} = \cancel{\left(\sum_{k\in\{n\}/i} R_{ki} \cdot \Delta t\right)} + 1 - \cancel{\left(\sum_{j\in\{n\}/\{i\}} R_{ji} \cdot \Delta t\right)} = 1 \tag{3.53}$$

- Every $P_{i\to k}$ is smaller than one:

  This can be fulfilled if the time step is chosen such that,

  $$\forall k: \; R_{ki} \cdot \Delta t \leq 1 \tag{3.54}$$

  this is also required for numeric stability in explicit Euler schemes.

  For the $i \neq k$ case our the limit to Δt directly follows from equation 3.52. For $i = k$ this follows from the definition of R and Δt, since $R_{ij} \geq 0$ if $i \neq j$ and $\Delta t \geq 0$.

  $$P_{i\to k} = R_{ki} \cdot \Delta t \leq 1 \tag{3.55}$$
  $$P_{i\to i} = 1 - \sum(\geq 0) \leq 1 \tag{3.56}$$

- All probabilities are greater or equal to zero:

  While all $P_{i\to k}$ for all $k \neq i$ are always positive, since both R_{ki} and $\Delta t \geq 0$, $P_{i\to i}$ can be smaller than 0, if the sum of all other $P_{i\to k}$ is greater than one.

  This specific case occurs if ions with the atomic state i are expected to on average change from the state i to another state more than once over the time step Δt, resulting in negative densities due to our linear approximation. This can be avoided if the step length Δt is reduced further.

  $$P_{i\to i} = 1 - \sum_{k\in\{n\}/i} P_{i\to k} = 1 - \sum_{k\in\{n\}/i} R_{ki} \cdot \Delta t \geq 0 \tag{3.57}$$

  $$\Rightarrow \boxed{\sum_{k\in\{n\}/i} R_{ki} \cdot \Delta t \leq 1} \tag{3.58}$$

  This condition is also mandated by numerical stability in addition to the need to avoid negative probabilities, see 3.3.2.1.3 for more information. The previous limit of Δt is also implied by this condition, since if any single summand is large than 1, a sum of positive summands will always be larger than 1.

3.3.2.1.2 Implementation The basic idea of the Monte Carlo TD solver is to randomly choose a new atomic state for each macro-ion in every time step, such that on average the rate equation is recreated.
Since we want to integrate this solver into a PIC-Simulation the length of this time step Δt is set by the PIC-Simulation and can not be changed on the fly. As described above, the Monte-Carlo solver maximum step length is limited by the rate Matrix, which is variable since it depends on the plasma conditions. It can therefore not be assured that the PIC time step Δt_{PIC} is always equal or less to the maximum time step of the Monte-Carlo TD solver $\Delta t_{\mathrm{maxMC}}$.

This can be avoided by sub stepping the PIC time step if rates rise too much. If the PIC time Δt_{PIC} step is shorter than the maximum time step of the Monte-Carlo TD solver $\Delta t_{\mathrm{maxMC}}$ no problem arises, since a shorter time step simply increases the probability of remaining in the current atomic state, Δt_{PIC} may simply be used as Δt.
If the PIC time step is longer, Δt_{PIC} is broken up in k sub steps Δt_k, with k a non zero natural number, such that $\Delta t_k \leq \Delta t_{\mathrm{maxMC}}$

$$\Delta t_k = \frac{\Delta t_{\mathrm{PIC}}}{k} \tag{3.59}$$

This allows us to both use a fixed time step Δt_{PIC} and fulfill the condition 3.58 by reducing the step length used by our solver as far as needed.
The Monte-Carlo TD solver therefore contains the following steps, executed for each macro-ion in parallel.

1. (determine if sub stepping is necessary and if how often)

   Check whether the condition 3.58 is fulfilled for Δt_{PIC}. If yes $k = 1$, if not find k such that Δt_k, fulfills the condition

2. (do sub steps)

   Repeat k times the following

   a) (choose new state randomly)

      Randomly choose with equal probability a new atomic state j from the set containing all possible atomic states.

   b) (check whether we change to new state)

      Calculate probability $P_{i\to j}$ according to equation 3.52. Generate a random number x, $0 \leq x \leq 1$ with uniform probability. If $x \leq P_{i\to j}$ change the atomic state of the macro-ion from i to j and exit, otherwise go to step 2a.

3.3.2.1.3 Inclusion in PIC-Simulation This solver has three major advantages compared to standard TD solver,

- It is parallel in macro-ions.

  Since there are many macro-ions in PIC-Simulation, this allows this solver to scale very well with GPUs. The large number of independent task allow a near ideal utilization of the massively parallel architecture of GPUs, consisting of thousands of execution units. In addition, the standard parallelization structures developed for PIC-algorithms can be reused, greatly simplifying the implementation and optimization.

- It does not require the entire rate matrix at once.

This reduces the memory consumption significantly, allowing us to fit the required data into the smaller lower level memory, with lower access latencies and higher write and read speeds. This significantly increases the speed of the computation since less time is spent waiting for memory reads and writes.

- It allows different step lengths for different atomic states.

  Standard TD solver must use a single step length Δt for all atomic states, requiring them to use the smallest necessary for all atomic states, this means more steps and consequently more operations than strictly necessary. The Monte-Carlo TD solver must only use the smallest Δt necessary in a single column of the rate matrix, and will therefore need less steps, with consequently less operations and less compute time required.

It is therefore better suited for integration into PIC-Simulations, but may still be improved, as explained below.
As derived in the previously, every transitions $i \to j$ has its own maximum time step length $\Delta t_{i \to j}$

$$\Delta t_{i \to j} = \frac{1}{R_{ji}} \tag{3.60}$$

Since the Monte-Carlo TD solver use the same time step Δt for all transitions from the initial state i transitions, the time step length Δt is therefore limited by the lowest maximum time step $\Delta t i \to j$ of all transitions, i.e. limited by the highest rate of change of a single component of the atomic population vector.

The highest rate of change is always experienced by the initial atomic state i, atomic population vector component n_i, since the rate of change this component experiences is equal to the sum of the rate of change of all others. The maximum step length of the Monte-Carlo solver is therefore limited by this rate, condition 3.58, and to be able to compute how many sub steps must be made this value must be available in memory or calculated each time.

This is improved if every transition is considered independently, allowing us to increase the effective step length and avoid sub stepping. The Markov-chain solver implements just that.

3.3.2.2 Markov-chain TD solver

The Markov-chain TD solver is based on modeling the time development as a stochastic process[1] with intermediary states.
Basis of this stochastic process is a set of atomic states with known probabilities $P_{k \to m}$ to change from state k to state m. The time development of a macro-ions atomic state is modeled as a chain of transitions over time, starting with the initial state i and ending with the final state

[1] a variation of a Markov chain to be more precise

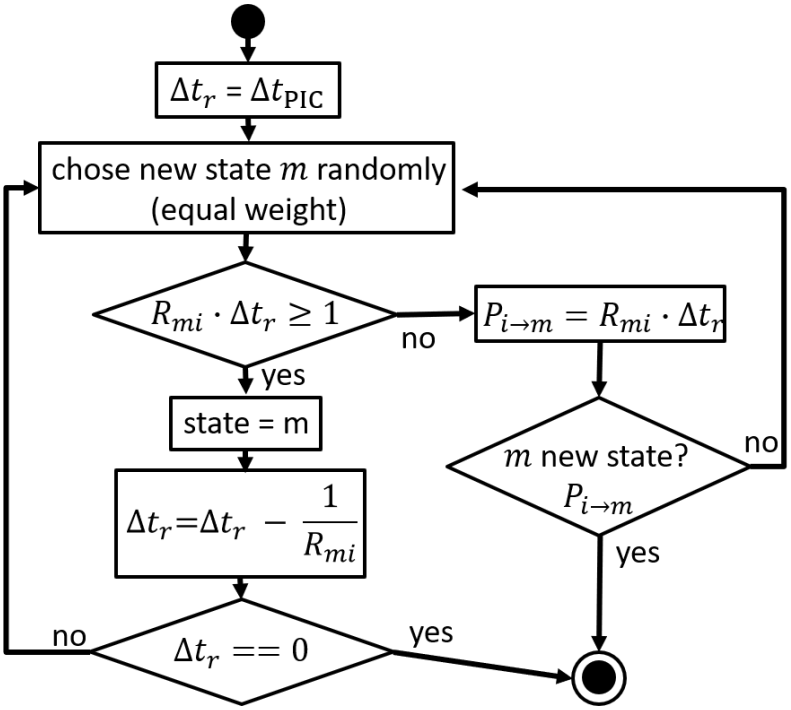

Figure 3.1: Flowchart of the Markov Chain solver algorithm

f after a fixed time Δt_{PIC} has passed[2]. Each transition $(k \to m)$ reduces the remaining time by $\frac{1}{R_{mk}}$, the average time associated with the transition. If more than one transition is possible, the transition is chosen randomly with equal probability for all possible transitions. If the remaining time Δt_r is not sufficient to complete a transition, the transition is only completed with the probability $P_{k\to m}(t_r)$, calculated according to equation 3.52, and the remaining time t_r is set to zero, otherwise the transition is rejected and a new transition chosen. If the remaining time reaches zero the the current state is the final state of this macro ion.
This algorithm is applied in parallel to all macro-ions, on average reproduces the rate equation and only requires a single rate in memory at one time.

[2]as the naming suggest, Δt_{PIC} is assumed to be the time step length of the PIC-simulation, but this is not necessary

4 Integrating atomic rate equation solvers into PIC-simulations

In this chapter we will develop the components needed as an interface between the previously derived TD-solvers and the existing PIC-simulation. Special care will taken to minimizes the memory used, to allow storage of in shared memory and avoid the longer access latencies of device memory wherever possible. Since shared memory of SMs of current GPUs is quite small, 128 KB per SM on V100 GPUs[9], this is not trivial.

4.1 Adaptive Electron energy histogram

To be able to calculate the rate of a given transition due to interactions with electrons, the local electron spectral density function is required. This function is approximated by binning the all macro-electrons in a the current super cell, with the value of the bin being the sum of all macro electrons weights with an energy in this bin.
The resulting histogram may span a wide energy range, with some electrons reaching more than 100 MeV in laser plasma experiments. This is problematic, since the complete range of electron energies starting with 0 must be covered be the electron energy histogram and bin widths are limited by the need to accurately calculate the rate. If the cross section changes fast and uniform large bin widths are used, the resulting approximate rate may be distorted heavily. This leads to fixed bin width histograms containing many more bins than strictly necessary, and consequently requiring much more memory, since the fixed bin width may not be larger than the smallest acceptable bin width of the entire energy range.
To reduce the number of bins we can use variable bin widths, choosing the optimal bin width for the current energy range. While in principle easy, the big question that remains to be answered is, how to find such an acceptable bin size.
This question is not new, in fact there is a publication by Sanders and Fabian from 2018[2], describing such an adaptive binning algorithm. Based on their core ideas I will develop an adaptive histogram suitable for our purpose.
The main problem we face in designing an adaptive histogram is that most adaptive binning algorithms depend on the content of the histogram itself. This means that all particles must be binned twice, once to get the distribution and calculate the bin width for each bin and a

second time to actually bin the particles in the new bins. This is not practical due to the effort required to bin all macro-electrons and the fact that we want to bin them every time step. The bin boundaries must be defined before the binning even begins.
This problem can be avoided if an relative error, as done in the publication by Aanders and Fabian [2], is used. This allows us to make the bin width calculation independent of the actual content of the histogram.
Bin widths are calculated using the following algorithm. Using a given relative error function, an initial bin width Δx_0 and an fixed boundary, we first calculate the initial relative error.
If the initial relative error is lower than the given target relative error, we double the initial bin width until we first exceed the target relative error. The last bin width whose relative error did not exceed the target is then used as bin width.
If the initial relative error is higher than the given target relative error, we half the initial bin width until we first fall below the target relative error. This last bin width is then used as bin width.

4.1.1 Relative error function

In our case the quantity we are interested is the atomic rate for a given energy bin R_B,

$$R_\mathrm{B}(E_I, \delta E) = \int_{E_i - \frac{\Delta E}{2}}^{E_i + \frac{\Delta E}{2}} \sigma(E) \cdot v_e(E) \cdot f(E) dE \qquad \text{(3.9 simplefied)}$$

with the corresponding relative error $\frac{\Delta R_\mathrm{B}}{f_\mathrm{B}}$.

R_B ... rate of energy bin B

f_B ... electron density of the energy bin B

The error ΔR_B can be approximated using a taylor expansion of the function integrated over,

$$R_\mathrm{B}(E_I, \delta E) = \int_{E_i - \frac{\Delta E}{2}}^{E_i + \frac{\Delta E}{2}} \sum_{n=0}^{\infty} \left(\frac{1}{n!} \frac{\delta^n}{\delta E^n} \Big[f(E) \cdot \sigma(E) \cdot v_e(E) \Big] \Big|_{E=E_i} \cdot (E - E_i)^n \right) dE \qquad (4.1)$$

changing the integration variable gives us,

$$R_\mathrm{B}(E_I, \delta E) = \int_{-\frac{\Delta E}{2}}^{\frac{\Delta E}{2}} \sum_{n=0}^{\infty} \left(\frac{1}{n!} \frac{\delta^n}{\delta E^n} \Big[\cdot f(E) \cdot \sigma(E) \cdot v_e(E) \Big] \Big|_{E=E_i} \cdot (\delta E)^n \right) d(\delta E) \qquad (4.2)$$

Using the generalized product rule for higher orders of derivatives twice results in,

$$R_\mathrm{B}(E_I, \delta E) = \int_{-\frac{\Delta E}{2}}^{+\frac{\Delta E}{2}} \sum_{n=0}^{\infty} (\delta E)^n \cdot \sum_{k=0}^{n} \frac{1}{k!} \cdot f^{(k)}(E_i) \cdot \sum_{l=0}^{n-k} \frac{1}{l! \cdot (n-k-l)!} \cdot \sigma^{(l)}(E_i) \cdot v_e^{(n-k-l)}(E_i) d(\delta E) \qquad (4.3)$$

everything but the δE term may be taken our from the integral since they are constants, and

the remaining integral solved

$$R_{\mathrm{B}}(E_I,\delta E)=\sum_{n=0}^{\infty}\cdot\sum_{k=0}^{n}\frac{1}{k!}\cdot f^{(k)}(E_i)\cdot\sum_{l=0}^{n-k}\frac{1}{l!\cdot(n-k-l)!}\cdot\sigma^{(l)}(E_i)\cdot v_e^{(n-k-l)}(E_i)\cdot\left[\frac{1}{n+1}\cdot(\delta E)^{n-1}\right]_{-\frac{\Delta E}{2}}^{+\frac{\Delta E}{2}} \tag{4.4}$$

with

$$\left[\frac{1}{n+1}\cdot(\delta E)^{n-1}\right]_{-\frac{\Delta E}{2}}^{+\frac{\Delta E}{2}}=\begin{cases}\frac{1}{n+1}\cdot\left(\frac{\Delta E}{2}\right)^{n}\cdot 2 & ;n+1\text{ uneven}\\ 0 & ;n+1\text{ even}\end{cases} \tag{4.5}$$

resulting in the final form,

$$R_{\mathrm{B}}(E_I,\delta E)=\sum_{a=0}^{\infty}\sum_{k=0}^{2a}\sum_{l=0}^{2a-k}\frac{1}{k!l!(2a-l-k)!}\cdot f^{(k)}(E_i)\cdot\sigma^{(l)}(E_i)\cdot v_e^{(n-k-l)}(E_i)\cdot\frac{1}{2a+1}\left(\frac{\Delta E}{2}\right)^{2a+1}\cdot 2 \tag{4.6}$$

From this we then must disregarding all terms used to calculate the actual rate from the histogram, shift all terms with $k>0$ into higher order terms, cut $f(E_i)$ giving us an approximate of the relative error of p-th order.

$$\boxed{\frac{\Delta R_{\mathrm{B}}(E_I,\delta E)}{f(E_i)}_{p}=\sum_{a=m}^{p}\sum_{l=0}^{2a}\frac{1}{k!l!(2a-l)!}\cdot\sigma^{(l)}(E_i)\cdot v_e^{(n-l)}(E_i)\cdot\frac{1}{2a+1}\left(\frac{\Delta E}{2}\right)^{2a+1}\cdot 2+\mathcal{O}(f^{(1)})} \tag{4.7}$$

To be able to calculate this term we need the arbitary order derivatives of both $\sigma(E)$ and the electron velocity $v_e(E)$. In the case of classical velocities the latter can be determined analytically.

$$v_e(E)=\sqrt{\frac{2\cdot E}{m}} \tag{4.8}$$

$$\Rightarrow v_e^{(Z)}(E)=\left(\frac{2\cdot E}{m}\right)^{Z}\cdot\left(\frac{2}{m}\right)^{Z}\cdot\prod_{l=0}^{Z-1}\left(\frac{1}{2}-l\right) \tag{4.9}$$

For the relativistic case an analytic term also does exist, but it can not be written in a closed form for arbitary order.
No such expression exists for $\sigma(E)$ in general, thus forcing us to rely on numerical derivatives.

4.1.2 Numerical derivative of arbitrary order using finite differences

In the following we will essential rederive a publication “Generation of finite Difference Formulas on Arbitrarily Space Grids” by Bengt Fornberg from 1988[5]. This is necessary since the original publication is too short to contain the derivation of the algorithm presented and the presented algorithm can not directly be implemented. I will indicate in the derivation where the publication actually ended.

This publication essentially describes a recursion algorithm for weights of finite difference formulas. The basic idea used by Fornberg is to approximate the function f using polynomials of at least one degree higher than the order of derivative, and find the correct finite difference coefficients from the derivative of the approximate polynomials.

For the approximation of the function we assume that the function value $f(x)$ is given at some arbitrary set sample points $\{\alpha_\nu\}$, with $0 \leq \nu \leq n$.

We want our approximation polynomial $p(x)$ to have the form,

$$p(x) = \sum_{\nu=0}^{n} F_{n,\nu}(x) \cdot f(\alpha_\nu) \tag{4.10}$$

To make sure that our approximation always intersects with the actual function at the sample points,

$$p(\alpha_\nu) \stackrel{!}{=} f(\alpha_\nu) \tag{4.11}$$

we need assure that,

$$F_{n,\nu}(\alpha_k) = \begin{cases} 1 & ;\nu = k \\ 0 & ;\nu \neq k \end{cases} \tag{4.12}$$

This may be achieved using the following construction. We begin by defining the function $w_n(x)$,

$$w_n(x) = \prod_{k=0}^{n}(x - \alpha_k) \tag{4.13}$$

the derivative of this function is,

$$w_n'(x) = \sum_{l=0}^{n} \left(\prod_{k=0}^{l-1}(x - \alpha_k) \cdot 1 \cdot \prod_{k=l+1}^{n}(x - \alpha_k) \right) \tag{4.14}$$

With these we define $F_{n,\nu}$,

$$F_{n,\nu}(x) = \frac{w_n(x)}{w_n'(\alpha_\nu) \cdot (x - \alpha_\nu)} \tag{4.15}$$

which fullfills condition 4.12.

$$F_{n,\nu}(\alpha_k) = \frac{w_n(\alpha_k)}{w_n'(\alpha_\nu) \cdot (\alpha_k - \alpha_\nu)} \qquad ;\nu \neq k \tag{4.16}$$

$$= \frac{0}{(\neq 0)} = 0 \tag{4.17}$$

$$F_{n,\nu}(\alpha_\nu) = \frac{w_n(\alpha_\nu)}{w_n'(\alpha_\nu) \cdot (\alpha_\nu - \alpha_\nu)} \qquad ;\nu \neq k \tag{4.18}$$

$$= \frac{0}{(\neq 0) \cdot 0} \rightarrow \frac{w_n'(\alpha_\nu)}{w_n'(\alpha_\nu)} = 1 \tag{4.19}$$

With the thus defined approximation polynomial we want to find coefficients $\delta^m_{n,\nu}$ such that,

$$\frac{d^m}{dx^m} f(x) \approx \frac{d^m}{dx^m} p_(x) = \sum_{\nu=0}^{n} \delta^m_{n,\nu} \cdot f(\alpha_\nu) \tag{4.20}$$

inserting the definition of $p(x)$,

$$\sum_{\nu=0}^{n} \frac{d^m}{dx^m} F_{n,\nu}(x) \cdot f(\alpha_\nu) = \sum_{\nu=0}^{n} \delta^m_{n,\nu} \cdot f(\alpha_\nu) \tag{4.21}$$

and comparing gives us,

$$\frac{d^m}{dx^m} F_{n,\nu}(x) = \delta^m_{n,\nu} \tag{4.22}$$

which we can integrate for $x = 0$, without loss of generality, since replacing x with $(x - x_0)$ always allows us to reach this value.

$$F_{n,\nu}(x) = \sum_{m=0}^{n} \delta^m_{n,\nu} \frac{x^m}{m!} \tag{4.23}$$

In addition the definition of $F_{n,\nu}$ gives us,

$$F_{n,\nu} = \begin{cases} F_{n-1,\nu} \cdot \frac{x-\alpha_n}{\alpha_\nu - \alpha_n} & ;\nu \leq n-1 \\ F_{n-1,n-1} \cdot \frac{(x-\alpha_{n-1}) \cdot w_{n-2}(\alpha_{n-1})}{w_{n-1}(\alpha_\nu)} & \nu = n \end{cases} \tag{4.24}$$

substituting equation 4.23 and comparing by power gives us the following 6 equations, two of which are explicitly mentioned in the original publication.

$$\delta^m_{n,\nu} = \delta^{m-1}_{n-1,\nu} \cdot \frac{m}{\alpha_\nu - \alpha_n} + \delta^m_{n-1,\nu} \cdot \frac{\alpha_n}{\alpha_n - \alpha_\nu} \qquad ;1 \leq m \leq n-1 \tag{4.25}$$

$$\delta^0_{n,\nu} = \delta^0_{n-1,\nu} \cdot \frac{\alpha_n}{\alpha_n - \alpha_\nu} \tag{4.26}$$

$$\delta^n_{n,\nu} = \delta^{m-1}_{n-1,n-1} \cdot \frac{n}{\alpha_\nu - \alpha_n} \tag{4.27}$$

$$\delta^0_{n,n} = \delta^0_{n-1,n-1} \cdot \alpha_{n-1} \cdot \frac{w_{n-2}(\alpha_{n-1})}{w_{n-1}(\alpha_n)} \tag{4.28}$$

$$\delta^m_{n,n} = \frac{w_{n-2}(\alpha_{n-1})}{w_{n-1}(\alpha_n)} \cdot \left[m \cdot \delta^{m-1}_{n-1,n-1} - \alpha_{n-1} \cdot \delta^m_{n-1,n-1} \right] \qquad 1 \leq m \leq n-1 \tag{4.29}$$

$$\delta^n_{n,n} = \delta^{n-1}_{n-1,n-1} \cdot n \tag{4.30}$$

These equation allow us to recursively calculate the finite difference coefficients, if correctly chained. While the original publication gives a pseudo code implementation using these equations to calculate the coefficents, I was not able verify this implementation, since all further steps were omitted. In addition the given implementation seems to assume having always

access to all previous values, therefore requiring relatively large amounts of memory. We will derive a variant of this algorithm that does not rely on this assumption and minimizes the memory required.
To do this we first abstract form the actual formulas. In essence all $\delta^m_{n,\nu}$ can be identified by a the tripple (m, n, ν) and the six equation above can be summarized as follows,

1. $(m, n, \nu) \leftarrow (m, n-1, \nu) \wedge (m-1, n-1, \nu)$
2. $(0, n, \nu) \leftarrow (0, n-1, \nu)$
3. $(m, m, \nu) \leftarrow (m-1, m-1, \nu)$
4. $(0, n, n) \leftarrow (0, n-1, n-1)$
5. $(m, n, n) \leftarrow (m, n-1, n-1) \wedge (m-1, n-1, n-1)$
6. $(m, m, m) \leftarrow (m-1, m-1, m-1)$

As the necessary start of the recursion we use $\delta^0_{0,0} = (0, 0, 0) = 1$
Based on combination of these equations we can directly derive a few simple cases,

- (m, m, ν):
  $(0,0,0) \xrightarrow{\nu \times 6.} (\nu, \nu, \nu) \xrightarrow{(m-\nu) \times 3.} (m, m, \nu)$
- $(m, m, 0)$:
  $(0,0,0) \xrightarrow{m \times 3.} (m, m, 0)$
- $(0, n, \nu)$:
  $(0,0,0) \xrightarrow{\nu \times 4.} (0, \nu, \nu) \xrightarrow{(n-\nu) \times 2.} (0, n, \nu)$

All of these case have in common that only a single variable must be kept in memory, since no branches are necessary. All other cases require the use equation 1. or 5., which both branch, making their use much more complicated. It is worthwhile to thoroughly analyze the equations 1. and 5. before we continue further, starting with equation 5..
Equation 5. may only be applied to triple of the form (m, n, n), if $1 \leq m \leq n-1$. Each such triple branches in two other triples I: $(m, n-1, n-1)$ and II: $(m-1, n-1, n-1)$ both once again of the original form.
Repeatedly applying equation to 5., leads in the branch I to an ever decreasing n while keeping m the same until m is equal to $n-1$ and equation 5. may no longer be applied. At his point we have reached the triple (m, m, m) which can be directly derived from $(0, 0, 0)$ using equation 6..
The branch II in contrast reduces, both m and n equally, keeping the number of steps in the branch I until a triple of the from (n,n,n) constant, until $m - 1 = 0$ is reached, also forbidding

the continued appllication of equation 5.. We will then have reached a triple of the form (0,n,n) which also has been solved already and can be directly derived from (0,0,0) using equation 4.. It also has to be noted that if we start with a given triple (m, n, n) the following chain may be built,

$$\begin{array}{ccc} (m,n,n) & \overset{5.\mathrm{I}}{\rightarrow} & (m,n-1,n-1) \\ 5.\mathrm{II}\downarrow & & 5.,\ \mathrm{II}\downarrow \\ (m-1,n-1,n-1) & \overset{5.\mathrm{I}}{\rightarrow} & (m-1,n-2,n-2) \end{array}$$

interweaving the different branches created in each application of the equation 5.
Based on this knowledge we can invert this chain and construct a recursion starting from $(0, 0, 0)$ to all triples of the form (m, n, n).

$$\begin{array}{ccccccc} (0,0,0) & \overset{4.}{\rightarrow} & (0,1,1) & \overset{4.}{\rightarrow} & \dots & \overset{4.}{\rightarrow} & (0,n-m,n-m) \\ 6.\downarrow & & 5.,\ \mathrm{II}\downarrow & & \dots & & 5.,\ \mathrm{II}\downarrow \\ (1,1,1) & \overset{5.\mathrm{I}}{\rightarrow} & (1,2,2) & \overset{5.\mathrm{I}}{\rightarrow} & \dots & \overset{5.\mathrm{I}}{\rightarrow} & (1,n-m+1,n-m+1) \\ \vdots & & & & & & \vdots \\ (1,1,1) & \overset{5.\mathrm{I}}{\rightarrow} & (1,2,2) & \overset{5.\mathrm{I}}{\rightarrow} & \dots & \overset{5.\mathrm{I}}{\rightarrow} & (m,n,n) \end{array}$$

This can be implemented as using a set of $k = m - n$ different variables (x_0, x_1, …, x_k) with the following algorithm.

```
x[0] = 1 // Init with (0,0,0)

for( 1 <= i <= k ):{
 x[i] = 4(x[i]) // Init with (0,i,i)
}
for( 1 <= j <= m ):{
 x[0] = 6(x[0]) // (j-1, j-1, j-1) -> (j,j,j)

 for( 1 <= i <= k ):{
  x[i] = 5(I=x[i-1], II=x[i], i, j )
  // I = (j,(i-1)+j,(i-1)+j), II = (j,i+(j-1),i+(j-1)) -> (j,i+j,i+j)
 }

}
// => x[k] = (m,n,n)
```

Alternatively this network may also be traversed, with swapped axis, requiring m variables $\{x_i\}$ instead of $n-m$, allowing us to choose whichever version requires less memory for a given input triple (m, n, n)

Similar consideration lead to the similar algorithms for all other case, but these tend to be much more complicated, so instead of deriving them in detail, only the results are given here.

- $(m, n, nu), m - n > \nu$

  Using $m + 1$ different x_i,

```
x[0] = 1 // Init with (0,0,0)

for( 1 <= i <= nu ):{
  x[i] = 6(x[i-1]) // Init with (i,i,i), x[i-1] = (i-1,i-1,i-1)
}
for( 1 <= i <= m-nu ):{
  x[nu+i] = 3(x[nu+i-1])
  // Init with (nu+i, nu+i, nu), x[nu+i-1] = (nu+i-1, nu+i-1, nu)
}
for( 1 <= j <= nu ):{
  x[0] = 4(x[0]) // (0,j-1, j-1) -> (0,j,j)

  for( 1 <= i <= nu-1 ):{
    if( i+j <= nu ):{
      x[i] = 5( I=x[i], II=x[i-1], i, j )
      // x[i] = (i,(j-1)+i,(j-1)+i)
      // x[i-1] = (i-1,j+(i-1),j+(i-1))
      // -> (i, i+j, i+j)
    }
    else:{
      x[i] = 1(I = x[i], II = x[i-1], i, j)
      // x[i] = (i,(j-1)+i,nu)
      // x[i-1] = (i-1,j+(i-1),nu)
      // -> (i, i+j, nu)
    }
  }
  for( nu <= i <= m ):{
          x[i] = 1(I=x[i], II=x[i-1], i, j )
    // I = (j,(i-1)+j,nu), II = (i-1,j+(i-1),nu) -> (i,j+i,nu)
  }
}
for( nu+1 <= j <= n-m ):{
  x[0] = 2(x[0]) // (0,j-1, nu) -> (0,j,nu)
```

```
  for( 1 <= i <= m ):{
          x[i] = 1(I=x[i], II=x[i-1], i, j )
    // I = (i,i+(j-1),nu), II = (i-1,(i-1)+j,nu) -> (i,i+j,nu)
  }
// => x[k] = (m,n,nu)
```

- $(m, n, nu), m - n \leq \nu, m \geq \nu$

  Using $m + 1$ different x_i,

```
x[0] = 1 // Init with (0,0,0)

for( 1 <= i <= nu ):{
  x[i] = 6(x[i-1]) // Init with (i,i,i), x[i-1] = (i-1,i-1,i-1)
}
for( nu+1 <= i <= m ):{
  x[nu+i] = 3(x[i-1])
  // Init with (i, i, nu), x[i-1] = (i-1, i-1, nu)
}
for( 1 <= j <= m-n ):{
  x[0] = 4(x[0]) // (0,j-1, j-1) -> (0,j,j)

  for( 1 <= i <= nu-m+n ):{
          x[i] = 5(I=x[i], II=x[i-1], i, j )
    // I = (i,i+(j-1),i+(j-1)), II = (i-1,(i-1)+j,(i-1)+j) -> (i,i+j,i+j)
  }
  for( nu-m+n+1 <= i <= nu ):{
    if( i+j <= nu ):{
      x[i] = 5( I=x[i], II=x[i-1], i, j )
      // x[i] = (i,(j-1)+i,(j-1)+i)
      // x[i-1] = (i-1,j+(i-1),j+(i-1))
      // -> (i, i+j, i+j)
    }
    else:{
      x[i] = 1(I = x[i], II = x[i-1], i, j)
      // x[i] = (i,(j-1)+i,nu)
      // x[i-1] = (i-1,j+(i-1),nu)
      // -> (i, i+j, nu)
    }
  }
```

```
  for( nu+1 <= i <= m ):{
          x[i] = 1(I=x[i], II=x[i-1], i, j )
    // I = (i,i+(j-1),nu), II = (i-1,(i-1)+j,nu) -> (i,i+j,nu)
  }
}
// => x[k] = (m,n,nu)
```

- $(m, n, nu), n > \nu > m, k \leq m$

  Using $n - m + 1$ different x_i,

```
x[0] = 1 // Init with (0,0,0)

for( 1 <= i <= n - m  ):{
  x[i] = 4(x[i-1]) // Init with (0,i,i), x[i-1] = (0,i-1,i-1)
}
for( 1 <= j <= m-n+nu ):{
  x[0] = 6(x[0]) // (j-1,j-1, j-1) -> (j,j,j)

  for( 1 <= i <= n-m ):{
          x[i] = 5(I=x[i-1], II=x[i], i, j )
    // I = (j,(i-1)+j,(i-1)+j), II = (j-1,i+(j-1),i+(j-1)) -> (j,i+j,i+j)
  }
}
for( m-n+nu+1 <= i <= m ):{
  x[0] = 6(x[0]) // (j-1,j-1, j-1) -> (j,j,j)
  for( 1 <= i <= j+nu ):{
      x[i] = 5( I=x[i-1], II=x[i], i, j )
      // x[i] = (j-1,(j-1)+i,(j-1)+i)
      // x[i-1] = (j,j+(i-1),j+(i-1))
      // -> (i, i+j, i+j)
  }
  for( j+nu+1 <= i <= n-m ):{
      x[i] = 1( I=x[i-1], II=x[i], i, j )
      // x[i] = (j-1,(j-1)+i,nu)
      // x[i-1] = (j,j+(i-1),nu)
      // -> (i, i+j,nu)
  }
}
// => x[k] = (m,n,nu)
```

- $(m, n, nu), n > \nu > m, k > m$

  Using $n - m + 1$ different x_i,

```
x[0] = 1 // Init with (0,0,0)

for( 1 <= i <= nu  ):{
  x[i] = 4(x[i-1]) // Init with (0,i,i), x[i-1] = (0,i-1,i-1)
}
for( nu + 1 <= i <= n - m):{
  x[i] = 2(x[i-1]) // Init with (0,i,nu), x[i-1] = (0,i-1,nu)
}
for( 1 <= j <= m ):{
  x[0] = 6(x[0]) // (j-1,j-1, j-1) -> (j,j,j)

  for( 1 <= i <= nu -j ):{
          x[i] = 5(I=x[i-1], II=x[i], i, j )
    // I = (j,(i-1)+j,(i-1)+j), II = (j-1,i+(j-1),i+(j-1)) -> (j,i+j,i+j)
  }
  for( nu - j <= i <= n-m ):{
      x[i] = 1( I=x[i-1], II=x[i], i, j )
      // x[i] = (j-1,(j-1)+i,nu)
      // x[i-1] = (j,j+(i-1),nu)
      // -> (i, i+j, i+j)
  }
}
// => x[k] = (m,n,nu)
```

4.1.3 Implementation Adaptive Histogram

The adaptive Histogram is implemented as a list of histogram bins, for each bin containing the accumulated weight of all macro-particles in this bin, in addition to the central energy and bin width of the bin, the storage of bin widths being necessary to allow gaps in the histogram The list is implemented as a fixed length array to avoid the use of pointers and their accompanying memory usage.

4.2 Feedback to electrons

As electrons interaction with ions and case a atomic state transition the energy of the electron may change.

This feedback of atomic processes to the actual electron distribution must be modeled to ensure energy conservation in the simulation. This is done by storing the sum of energy released to or taken from the electron spectrum due due to each single atomic process in a second histogram over the energy of the interaction partner. This is possible since we resolve interaction process by energy bin, distinguishing between different energy bins of the same processes when a random transition is chosen. After the actual atomic rate solver step is completed, this change in energy is then applied equally to all electrons in the respective energy bin, thus modeling feedback of atomic process to the electron spectrum.

5 Outlook

In this thesis a memory frugal, highly parallelized atomic physics model that is directly coupled to a PIC-simulation has been derived. The model is computationally feasible and rests on a solid theoretical derivation. Based on the theoretical derived model a functional implementation was created for the simulation code PCIonGPU. While a first step in the right direction, further work must be invested in optimizing the computational performance, and validating the derived model both against experimental data and other simulations. In addition several possible future improvements have already been envisioned.

6 Acknowledgment

I would like to extend my deep thanks to thank Prof. Ulrich Schramm for granting me my thesis topic and to my supervision Dr. Thomas Kluge for his support and intellectual input. Furthermore I want to thank Dr. Sergei Bastrakov, Dr. Klaus Steiniger and Dr. Richard Pausch for stimulating discussions.

7 Bibliography

[1] Yuri Ralchenko, “Modern Methods in Collisional-radiative Modeling of Plasmas”, 2016, Springer nature

[2] J.S. Sanders, A.C. Fabian, “Adaptive Binning of X-ray galaxy cluster images”, 2018

[3] H. K. Chung, R. W. Lee, M. H. Chen, Y. Ralchenko, “The How to for FLYCHK @NIST”, 2008, `https://nlte.nist.gov/FLY/Doc/Manual_FLYCHK_Nov08.pdf`, accessed in 2019

[4] Richard W. Lee,”The How to For FLY”, 1995, `https://nlte.nist.gov/FLY/Doc/FLY95.pdf`, accesed in 2019

[5] Bengt Fornberg, “Generation of Finite Difference Formulas on Arbirarily Space Grids”, 1988, Mathematics of Computation, Volume 31, Number 184, accessed in 2019

[6] R. Mewe, “Interpolation Formula for the electron Impact Excitation of Ions in the H-, He-, Li- and Ne-Sequences”, 1972, Astron. & Astrophys. 20, 215-221, accessed in 2019

[7] Agner Fog, ”4. instruction tables, List of instruction latencies, throughputs and micro operation breakdowns for Intel, AMD and VIA-CPUs”, 2020, `www.agner.org/optimize`, accessed 2020

[8] Jouko Niiranen, ”Fast and accurate symmetric Euler algorithm for electromechanical simulations NOTE: The method became later known as "Symplectic Euler"”, 1999, 6th. Int. Conf. Electrimacs’ 99, Proceedings Vol. 1, pp. 71-78, accessed in 2019

[9] Author collective, “NVIDIA TESLA V100 GPU ARCHITECTURE”, 2017, `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, accessed 2020

[10] A. Burgess and M. R. H. Rudge,“The Ionization of Hydrogenic Positive Ions by Electron Impact”, 1963, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, Vol. 273, No. 1354 (May 21, 1963), pp. 372-386, Royal Society, `https://www.jstor.org/stable/2414457`, accessed 2020

[11] Alan Burgess and Marita c. Chidichimo, Electron Impact ionization of complex ions, 1983, Mon. Not. R. astr. Soc., 203, 1269-1280, accessed 2019

[12] M. R. H. Rudge and S. B. Schwartz, “The ionization of He+ by electron impact”, 1965, Proc. Phys. Soc. 86 773, accesed 2019

[13] Wolfgang Lotz, “Electron-Impact Ionization Cross-Sections for Atoms up to Z– 108", 1970, Z. Physik, 232, 101-107, accessed 2019

[14] Wolfgang Lotz, “An Empirical Formula for the Electron-Impact Ionization Cross-Section*”, 1967, Z. Physik, 206, 205-211, accessed 2019

[15] Y. Sentoku and A.J. Kemp, ”Numerical methods for particle simulations at extreme densities and temperatures: Weighted particles, relativistic collisions and reduced currents, 2008, Journal of Computational Physics 227, 6846-6861, accessed 2019

[16] C.W.Gear, “Numerical initial value problemsin ordinary differential equations”, 1971, Prentice Hall, New Jersey

[17] Heiko Burau, Rene Widera, U. Schramm, T. Cowan, et. al, “PIConGPU: A fully relativistic particle-in-cell code for a GPU cluster”, 2010 IEEE Transactions on Plasma Science, 38, 2831-2839, DOI:10.1109/TPS.2010.20643010

[18] author collective, “DDR4 SDRAM”, `https://en.wikipedia.org/wiki/DDR4_SDRAM`, accessed November, 2020

[19] author collective,”AMD EPYC 7002 Series Processors Data Sheet” `https://www.amd.com/system/files/documents/AMD-EPYC-7002-Series-Datasheet.pdf`, accessed November 2020

[20] T. Kluge, M. Bussmann, U. Schramm, t. Cowan, et. all., ”Nanoscale femtosecond imaging of transient hot solid density plasmas with elemental and charge state sensitivity using resonant coherent diffraction”, 2016, Physics of Plasmas 23, 033103, `https://doi.org/10.1063/1.4942786`

[21] Mark Harris, “Using Shared Memory in CUDA C/C++”, 2013, `https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/`,

[22] R. Mishra, P.Leblance, et all., “Collisional particle-in-cell modeling for energy transport accompanied by atomic processes in dense plasmas”, 2013, Physics of Plasmas, 20, 7, DOI:10.1063/1.4812701

[23] auhto collcetive, Documentation of the Smilie PIC code, accesed November 2020

[24] T. Kluge, M. Bussman, U. Schramm, T. Cowan, 2016, “Nanoscale femtosecond imaging of transient hot solid density plasmas with elemental and charge state sensitivity using resonant coherent diffraction”, Physics of Plasmas 23, 033103, `https://doi.org/10.1063/1.4942786`

[25] Source Coed Repository of the PIC-code PIConGPU, https://github.com/ComputationalRadiationPhysics/picongpu

A Proof of consistency of enumeration and indexation

The number of super configurations $\#_{\vec{N}}$ can be calculated using both the equation 2.17 as well as equation 2.23.

Using the maximum index possible,

$$\#_{\vec{N}} = \max\{\#\} + 1 \tag{A.1}$$

which can in turn be calculated using equation 2.23, or directly using equation 2.17. Both are consistent as will be shown now.

Proof. The maximum index corresponds to the occupation number vector with the maximum occupation number for each level, since equation 2.23 is monotone increasing in all occupation numbers.

$$\# = \sum_{i=1}^{n_{\max}} N_i \cdot \prod_{j=1}^{i-1} (\min(g(i), Z) + 1) \qquad ; g(i) = 2 \cdot i^2 > 0, Z > 0$$

(2.23, revisited)

$$\Rightarrow \max\{\#\} = \#(\vec{N} = (\max(N_1), \max(N_2), \dots)) \qquad ; N_i \leq min(g(i), Z) \tag{A.2}$$

$$\Rightarrow \vec{N}_{\max} = (\min(g(1), Z),\ \min(g(2), Z), \dots) \tag{A.3}$$

$$\Rightarrow \max(\#) = \sum_{i=1}^{n_{\max}} \left(\min(g(i), Z) \cdot \prod_{j=1}^{i-1} (\min(g(j), Z) + 1) \right) \tag{A.4}$$

$$\Rightarrow \#_{\vec{N}} = \left(\sum_{i=1}^{n_{\max}} \left(\min(g(i), Z) \cdot \prod_{j=1}^{i-1} (\min(g(j), Z) + 1) \right) \right) + 1 \tag{A.5}$$

$$= \left(\sum_{i=0}^{n_{\max}} \left(g^*(i) \cdot \prod_{j=0}^{i-1} (g^*(i) + 1) \right) \right) + 1 \tag{A.6}$$

While using equation 2.17 gives us,

$$\#_{\vec{N}} = \prod_{n=1}^{n_{\max}} (\min(g(n), Z) + 1) \qquad \text{(2.17, revisited)}$$

$$= \prod_{i=1}^{n_{\max}} (g^*(i) + 1) \qquad ; g^*(i) \stackrel{!}{=} \min(g(i), Z) \qquad \text{(A.7)}$$

To proof consistency we start with equation A.7 and separate the last multiplicative term of our product,

$$\#_{\vec{N}} = \left(\prod_{i=1}^{n_{\max}-1} (g^*(i) + 1) \right) \cdot (g^*(n_{\max}) + 1) \qquad \text{(A.8)}$$

expanding the multiplication yields a term, similar in structure to equation 2.23 and the original product once again, now with a reduced upper limit of the index range.

$$\#_{\vec{N}} = g^*(n_{\max}) \cdot \left(\prod_{i=1}^{n_{\max}-1} (g^*(i) + 1) \right) + 1 \cdot \left(\prod_{i=1}^{n_{\max}-1} (g^*(i) + 1) \right) \qquad \text{(A.9)}$$

Rewriting to better conform to the structure of equation 2.23 by inserting a sum over one component, yields the following.

$$\#_{\vec{N}} = \sum_{i=n_{\max}}^{n_{\max}} \left(g^*(i) \cdot \prod_{j=1}^{i-1} (g^*(j) + 1) \right) + \left(\prod_{i=1}^{n_{\max}-1} (g^*(i) + 1) \right) \qquad \text{(A.10)}$$

Of course we can continue expanding the remaining product one term at a time

$$\#_{\vec{N}} = \sum_{i=n_{\max}}^{n_{\max}} \left(g^*(i) \cdot \prod_{j=1}^{i-1} (g^*(j) + 1) \right) + \left(\prod_{i=1}^{n_{\max}-2} (g^*(i) + 1) \right) \cdot (g^*(n_{\max} - 1) + 1) \qquad \text{(A.11)}$$

which once again gives a term of the same structure as one summand of equation 2.23 in addition to our product with the upper limit of its index range reduced further.

$$\#_{\vec{N}} = \sum_{i=n_{\max}}^{n_{\max}} \left(g^*(i) \cdot \prod_{j=1}^{i-1} (g^*(j) + 1) \right) + g^*(n_{\max}-1) \cdot \left(\prod_{i=1}^{n_{\max}-2} (g^*(i) + 1) \right) + \left(\prod_{i=1}^{n_{\max}-2} (g^*(i) + 1) \right) \qquad \text{(A.12)}$$

The second term we integrate into our sum, the third term remains.

$$\#_{\vec{N}} = \sum_{i=n_{\max}-1}^{n_{\max}} \left(g^*(i) \cdot \prod_{j=1}^{i-1} (g^*(j) + 1) \right) + \left(\prod_{i=1}^{n_{\max}-2} (g^*(i) + 1) \right) \qquad \text{(A.13)}$$

Repeating this procedure, or complete induction if you prefer, gives the following.

$$\Rightarrow \#_{\vec{N}} = \sum_{i=2}^{n_{\max}} \left(g^*(i) \cdot \left(\prod_{j=1}^{i-1} (g^*(j)+1) \right) \right) + \left(\prod_{i=1}^{1} (g^*(i)+1) \right) \tag{A.14}$$

Which we can simplefy,

$$\#_{\vec{N}} = \sum_{i=2}^{n_{\max}} \left(g^*(i) \cdot \left(\prod_{j=1}^{i-1} (g^*(j)+1) \right) \right) + (g^*(i)+1) \tag{A.15}$$

$$= \sum_{i=2}^{n_{\max}} \left(g^*(i) \cdot \left(\prod_{j=1}^{i-1} (g^*(j)+1) \right) \right) + g^*(1) \cdot 1 + 1 \tag{A.16}$$

$$= \sum_{i=2}^{n_{\max}} \left(g^*(i) \cdot \left(\prod_{j=1}^{i-1} (g^*(j)+1) \right) \right) + g^*(1) \cdot \prod_{j=1}^{0} (g^*(j)+1) + 1 \tag{A.17}$$

$$\prod_{i=1}^{n_{\max}} (g^*(i)+1) = \sum_{i=1}^{n_{\max}} \left(g^*(i) \cdot \left(\prod_{j=1}^{i-1} (g^*(j)+1) \right) \right) + 1 \quad \blacksquare \tag{A.18}$$

which is indeed equation A.6, q.e.d.

B Estimation of necessary cell size of a PIC-Simulation in solid density plasmas

The cell size Δx of a PIC-simulation is required be small enough to resolve the plasma oscillation Δx, $\Delta x \approx$ nm

$$\Delta x \leq \frac{\lambda_{\text{plasma}}}{N} \qquad ; N > 2 \tag{B.1}$$

$$\lambda_{\text{plasma}} = \frac{\text{c} \cdot 2\pi}{\omega_{\text{plasma}}} \tag{B.2}$$

$$\omega_{\text{plasma}} = \sqrt{\left(\frac{n_e \cdot e^2}{\epsilon_0 \cdot m_e}\right)} \tag{B.3}$$

$$\Delta x \leq \frac{\text{c} \cdot 2\pi}{\sqrt{\left(\frac{n_e \cdot e^2}{\epsilon_0 \cdot m_e}\right)} \cdot N} \tag{B.4}$$

N	...	number of cells per wavelength
c	...	speed of light
n_e	...	electron number density
m_e	...	electron mass
ϵ_0	...	electric field constant
e	...	elementary charge

C Source code files of prototype implementation

The concepts described in my thesis have not only been derived theoretically but also practically implemented in PIConGPU [25]. Due to the large size of the implementation , in total about 8900 lines, I can not add the implementation to the apendix but, the entire source code is freely available on github under `https://github.com/ComputationalRadiationPhysics/picongpu/pull/3145`.

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit im Rahmen der Betreuung am Institut für Strahlenphysik ohne unzulässige Hilfe Dritter verfasst und alle Quellen als solche gekennzeichnet habe.

Brian Edward Marré
Dresden, 23. November 2020