MOT simulation
- Code documentation -

Ramon Gabriel Teixeira Rosa

June 8, 2018

1 Introduction

A Magneto-Optical Trap (MOT) can be used on the trapping and cooling of atoms. For high efficiency, MOTs required a fine adjustment of the experimental parameters like laser detuning, intensity and polarization. We propose a computational tool to allow the simulation of such systems and optimization of their parameters.

We simulate the magneto-optical trapping of different species using the Monte Carlo method based on the quantitative treatment of light scattering rates for multiple transitions. The trapping times distribution functions were analyzed and the results were compared to experimental data from the literature.

2 Methods

2.1 Magnetic field and laser beams

The MOT is generated by a pair of coils on the anti-Helmholtz configuration, producing a quadrupole magnetic field $\vec{B}(\vec{r})$ described as

$$\begin{aligned} \vec{B} &= \frac{A}{2}\left(x\hat{e_x} \ + \ y\hat{e_y} \ - \ 2z\hat{e_z}\right) \\ &= B\hat{e_B}, \end{aligned} \tag{1}$$

with $r_{1,2,3} \equiv x,\ y,\ z$ and $\hat{e}_{1,2,3} \equiv \hat{e_x},\ \hat{e_y},\ \hat{e_z}$, and A is the magnetic field gradient.

The laser beams are assumed to be gaussian, so their intensity profile is given by

$$I(r_\perp) = I_{peak} \cdot \exp\left(-2\frac{r_\perp^2}{\xi^2}\right), \tag{2}$$

where $r_\perp$ is the distance to the beam center axis, I_{peak} is the intensity of the peak ($I_{peak} = I(r_\perp = 0)$), where ξ is the $1/e^2$ beam waist.

We simulate a MOT composed of N_{beams} beams, each beam (of index k) with a propagation direction $\hat{e_k}$ and a polarization vector $\hat{\psi}_k$, with $k = 1,\ 2,\ ...,\ N_{beams}$. For example, a right-handed circular polarized beam propagating upwards would have:

$$\hat{e_k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \text{ and } \hat{\phi}_k = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -i \\ 0 \end{pmatrix}.$$

The polarization vectors of each beam are written on the lab frame, so even beams with the same polarization handedness propagating at different directions would have polarization vectors written differently.

2.2 Polarization-allowed atomic transitions

An atom experiencing the Zeeman effect of an external magnetic field has a well-defined quantization axis, which is usually defined as $\hat{e_z}$. Using this definition, it is possible to show that transitions with $\Delta m_j = -1,\ 0,\ +1$, namely σ^-, π, and σ^+ transitions, are only dipole-allowed by the perturbation of electric fields of the form [1]:

$$\begin{aligned}
\vec{E}_{\sigma^-} &= E_0\ e^{i\omega t}\ \frac{\hat{e_x} - i\hat{e_y}}{\sqrt{2}}, \\
\vec{E}_{\pi} &= E_0\ e^{i\omega t}\ \hat{e_z}, \\
\vec{E}_{\sigma^+} &= E_0\ e^{i\omega t}\ \frac{\hat{e_x} + i\hat{e_y}}{\sqrt{2}}.
\end{aligned}$$

We define transition vectors as the electric field polarization vectors necessary

in order to access those transition, so in this case:

$$\hat{\sigma^-} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -i \\ 0 \end{pmatrix}, \ \hat{\pi} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \ \hat{\sigma^+} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ i \\ 0 \end{pmatrix}.$$

An atom at an arbitrary position will experience a magnetic field $\vec{B} = B\hat{e_B}$ given by the equation 1, so its quantization axis will be defined by $\hat{e_B}$. The transition unit vectors have to be calculated. We start doing so by defining the π, which points in the direction of the magnetic field:

$$\hat{\pi} = \hat{e_B}. \tag{3}$$

Now we have to define a new orthogonal basis $\{\hat{h_x}, \hat{h_y}, \hat{h_z}\}$, where $\hat{h_z} = \hat{e_B}$ and the vector space orientation is positive ($\hat{h_x} \times \hat{h_y} = \hat{h_z}$).

First, we create a random[1] vector $\hat{q_1}$ (not parallel to $\hat{e_B}$, and orthogonalize it with respect to $\hat{e_B}$:

$$\vec{q_1}^{\perp} = \hat{q_1} - \hat{e_B}(\hat{q_1} \cdot \hat{e_B}).$$

So we have the first basis vector

$$\hat{h_1} = \frac{\vec{q_1}^{\perp}}{\left|\vec{q_1}^{\perp}\right|}, \tag{4}$$

and can calculate the second basis vector already imposing the vector space orientation as follows:

$$\hat{h_2} = \hat{e_B} \times \hat{h_1}. \tag{5}$$

Now, the transition vectors can be written as:

[1]For this whole document, the term "random" is used loosely and represent pseudo-random numbers generated by deterministic random bit generators.

$$\hat{\sigma^-} = \frac{1}{\sqrt{2}}\left(\hat{h_1} - i\hat{h_2}\right), \tag{6}$$

$$\hat{\pi} = \hat{e_B}, \tag{7}$$

$$\hat{\sigma^+} = \frac{1}{\sqrt{2}}\left(\hat{h_1} + i\hat{h_2}\right). \tag{8}$$

2.3 Scattering rate, beam selection and recoil

The scattering rate for an atom considering a single transition with bandwidth Γ and a beam with detuning δ_0, intensity I and polarization vector equal to the transition vector is given by[1]:

$$R = \frac{\Gamma}{2}\frac{s}{1 + s + 4\left(\delta_0/\Gamma\right)^2}, \tag{9}$$

where $s = I/I_{sat}$ is the saturation parameter.

In our case, the polarization vectors may not be equal to the transition vectors, so that multiple levels may be coupled by the same excitation beam. We need to calculate the scattering rate $R_{k,u}$ of each beam k related to each transition u (with polarization vector $\hat{u}$) for an atom at an arbitrary position $\vec{r}$. First, we calculate the magnetic field at this position using the equation 1. We then calculate the transition vectors using equations 6, 7, and 8.

The beam electric field can be decomposed in the orthonormal basis of transition vectors. We can calculate the Poynting vector in this basis. The effective saturation parameter for each transition u for each beam k can be calculated by projection of the beam polarization vectors $\hat{\phi_k}$ to $\hat{u}$ squared.

$$s_{k,u}^{\text{eff}} = s\left|\langle\hat{\phi_k}, \hat{u}\rangle\right|^2, \tag{10}$$

where

$$\langle\vec{a}, \vec{b}\rangle = \sum_{j=1}^{3} a_j^* b_j,$$

being a_j (or b_j) the $j-th$ coordinate of the vector $\vec{a}$ (or $\vec{b}$), and a_j^* being the complex conjugate of a_j.

Additionally, the detuning *delta* is defined in terms of the unperturbed atom. The atom in our model, however, experience a Zeeman shift on both ground and excited states, so there is a net Zeeman frequency shift δ_Z equal to:

$$\delta_Z = \frac{\mu_B \left|\vec{B}\right| (g_{J_g} m_J^{\text{gnd}} - g_{J_e} m_J^{\text{exc}})}{h}, \tag{11}$$

where μ_B is the Bohr magneton, h is the planck constant, g_{J_g} and g_{J_e} are the Landé g-factor of ground and excited states respectively, and m_J^{gnd} and m_J^{exc} are the m_J quantum numbers of ground and excited states respectively.

The Doppler effect also plays a role on the effective local detuning. For an atom with velocity $\vec{v}$, a beam propagating at a direction $\hat{e_B}$ and frequency ν_0 at the lab frame, will have a frequency ν given by

$$\nu = \nu_0 \left(1 - \frac{\vec{v} \cdot \hat{e_B}}{c}\right), \tag{12}$$

where c is the speed of light and we are assuming a non-relativistic limit. If the atom has a velocity that is opposed to the propagation direction of the beam, the light will be blueshifted whereas it will be redshifted otherwise. So, the Doppler effect produces a shift on the laser frequency equal to

$$\delta_D = -\nu_0 \frac{\vec{v} \cdot \hat{e_B}}{c}. \tag{13}$$

The effective local detuning δ^{eff} is the contribution of the laser detuning, and the Zeeman and Doppler shifts.

$$\begin{aligned} \delta_{k,u}^{\text{eff}} &= \delta_0 + \delta_Z + \delta_D \\ &= \delta_0 + \frac{\mu_B \left|\vec{B}\right| (g_{J_g} m_J^{\text{gnd}} - g_{J_e} m_J^{\text{exc}})}{h} - \nu_0 \frac{\vec{v} \cdot \hat{e_B}}{c}. \end{aligned} \tag{14}$$

Now, we can calculate $R_{k,u}$:

$$R_{k,u} = \frac{\Gamma}{2} \frac{s_{k,u}^{\text{eff}}}{1 + s_{k,u}^{\text{eff}} + 4\left(\delta_{k,u}^{\text{eff}}/\Gamma\right)^2}. \tag{15}$$

2.4 Aditional forces

The two additional forces being used here are the force of the magnetic field on the atom magnetic dipole, and the gravitational force.

The quantization axis of an atom experiencing an external magnetic field $\vec{B} = B\hat{e_B}$ will be $\hat{e_B}$. The component of its magnetic dipole moment on this direction is given by $\mu_{z'} = g_J m_J \mu_B$:

The magnetic force is given by [2]

$$\vec{F}_{mag} = -\nabla\left[(g_J m_J \mu_B\ \hat{e_B}) \cdot \vec{B}\right] = -g_J m_J \mu_B \nabla B. \tag{16}$$

The gradient of the magnetic field strength is given by

$$\begin{aligned}
\nabla B &= \nabla\left(\frac{A}{2}\sqrt{x^2 + y^2 - 4z^2}\right) \\
&= \left(\hat{e_x}\frac{\partial}{\partial x} + \hat{e_y}\frac{\partial}{\partial y} + \hat{e_y}\frac{\partial}{\partial y}\right)\left(\frac{A}{2}\sqrt{x^2 + y^2 + 4z^2}\right) \\
&= \frac{A}{2}\frac{1}{2\left(x^2 + y^2 + 4z^2\right)^{1/2}}\left(2x\hat{e_x} + 2y\hat{e_y} + 8z\hat{e_z}\right) \\
&= \frac{A}{\sqrt{x^2 + y^2 + 4z^2}}\left(\frac{x}{2}\hat{e_x} + \frac{y}{2}\hat{e_y} + 2z\hat{e_z}\right).
\end{aligned} \tag{17}$$

Similarly, for arbitrary magnetic field gradients on each direction,

$$\vec{B} = G_x x\hat{e_x} + G_y y\hat{e_y} + G_z z\hat{e_z}, \tag{18}$$

the gradient of the magnetic field strength is given by

$$\nabla B = \frac{1}{\sqrt{{G_x}^2 x^2 + {G_y}^2 y^2 + {G_z}^2 z^2}} \left({G_x}^2 x \hat{e_x} + {G_y}^2 y \hat{e_y} + {G_z}^2 z \hat{e_z} \right). \tag{19}$$

In this case, the condition

$$\nabla \cdot \vec{B} = G_x + G_y + G_z = 0 \tag{20}$$

must be imposed when defining G_x, G_y, and G_z.

The gravitational force is also relevant to this simulation. It is given by

$$\vec{F_g} = M\vec{g}, \tag{21}$$

being M the atom mass and $\vec{g}$ the gravitational acceleration.

The total external force is given by

$$\vec{F}_{\text{ext}} = \vec{F}_{mag} + \vec{F}_g. \tag{22}$$

3 Simulation

This Monte Carlo simulation calculates the classical trajectory of an atom in the MOT. The initial conditions for the simulation are described on the next subsection.

3.1 Initial conditions

We start by defining an initial position and velocity for the atom. The initial position is a random position following a gaussian probability distribution with width defined by the user.

The inital velocity is also random and follows the Maxwell-Boltzmann distribution for a temperature defined by the user. We do so by setting each cartesian component of the initial velocity to be random following a gaussian distribution of width $\sqrt{k_B T/M}$

centered at zero .

The random numbers following a gaussian distributions are generated using the Box-Muller transformation.[3]

The algorithm for each iteration is described on the next subsection.

3.2 Stochastic evolution

The magnetic field at the atom position is calculated using the equation 1. The effective local detuning $\delta_{k,u}^{\text{eff}}$ is calculated to each beam k and transition u (the effective may be different for different beams due to the Doppler effect) using equation 14, and then the scattering rate is calculated also for each beam and each transition using equation 15.

Having calculated the scattering rate $R_{k,u}$ for each beam and each transition, we then proceed by selecting each beam the atom absorbs. Instead of splitting the simulation time steps into small fractions of the typical scattering times as implemented by Hanley *et al.*,[4] we speed the simulation by calculating the individual scattering times for each iteration and allowing the iteration time evolution to continuously adapt to the calculated individual scattering time. This way, each iteration represents one photon absorption and one photon emission, without the need of intermediate calculation steps.

The rate equations for absorption and decay have exponential solutions with average lifetimes $\tau_{k,u} = 1/R_{k,u}$. We generate random lifetimes following a probability distribution $P(t) = e^{-t/\tau_{k,u}}$ for each beam and each transition. The pair beam and transition with shorter lifetime, which is the first transition to happen in that given configuration, is chosen.

The random numbers following an exponential distribution are generated by the cumulative distribution inverse transformation method.[5] We want to generate random numbers following the normalized distribution:

$$P(t) = \frac{1}{\tau} e^{-t/\tau}.$$

The cumulative distribution function of P is given by:

$$\bar{P}(t') = \int_0^{t'} dt' \; P(t) = 1 - e^{-t'/\tau}.$$

Solving for t, we get

$$t = -\tau \; ln\left(1 - \bar{P}(t)\right).$$

Replacing the cumulative distribution function $\bar{P}$ on this equation by a random number κ following an uniform distribution in the interval $[0, 1[$, we get

$$t_{\text{exp}} = -\tau \; ln\left(1 - \kappa\right), \tag{23}$$

where t_{exp} is a random number following an exponential distribution with average τ.

Once the absorbed beam is chosen and the lifetime δt is determined, the simulation calculates the next velocity and position of the atom after the time δt is passed. The position, velocity, and time at the $n - th$ iteration are:

$$\vec{r_n} = \vec{r}_{n-1} + \vec{v}_{n-1}t + \frac{\vec{F}_{\text{ext}}}{2M}\delta t^2, \tag{24}$$

$$\vec{v_n} = \vec{v}_{n-1} + \frac{h}{\lambda}\hat{e_k} - \frac{h}{\lambda}\hat{e_r} + \frac{\vec{F}_{\text{ext}}}{M}\delta t, \tag{25}$$

$$t_n = t_{n-1} + \delta t, \tag{26}$$

where $\hat{e_k}$ is the propagation direction of the absorbed photon and $\hat{e_r}$ is a random unit vector related to the isotropic photon emission, and δt is calculated for each iteration.

This process is repeated until determined otherwise by one of the stopping criteria. We used two stopping criteria, which are:

- Maximum number of iterations reached;
- Atom distance to the MOT center becomes larger than an user defined threshold distance.

We define the Trapping Time as the last time value in which the velocity vector is opposed to the position vector, which is t_n for the last n in which $\vec{v_n} \cdot \vec{r_n} < 0$. If the simulation stops due to the first stopping criteria being reached the Trapping Time is

meaningless, but in the second case it can be correlated to the trapping efficiency and can be used to optimize the MOT configuration.

3.3 Simulation output

An x,y,z histogram is built during the simulation. A 3-D meshgrid of zeros is created at the start of the execution. At every iteration, the value of the voxel corresponding to the position of the atom is incremented by δt. An output file consisting of a 2-D dataset generated from the sum of the 3-D histogram in the y direction is created.

The positions history of the atom is not stored as the attempt to do so typically results in stack or heap overflow for simulations with large number of iterations.

The simulation is carried out a number of times defined by the user and the histograms are averaged. The simulation can also be performed while varying parameters like the laser detuning, having multiple averages for each set of parameters that are being varied. The trapping time is also stored for each time the simulation was executed.

We hypothesize that curves of trapping time *versus* detuning can be correlated with curves of number of atoms on the MOT *versus* detuning, as shown by Ilzhofer *et al.*[6]

3.4 Simulation parameters

The parameters chosen by the user for the simulation are:

- Gravity $[m/s^2]$;
- Magnetic field gradient $[T/m]$;
- Laser beams propagation directions ($\hat{e_k}$);
- Laser detuning [units of Γ];
- Beams polarizations, which can be:

  "r": Right-handed circular polarization;

  "l": Left-handed circular polarization;

  "x": Polarization vector on plane defined by propagation vector and x-axis (polarization vector orthogonal to propagation direction vector);

"y": Polarization vector on plane defined by propagation vector and y-axis (polarization vector orthogonal to propagation direction vector);

"z": Polarization vector on plane defined by propagation vector and z-axis (polarization vector orthogonal to propagation direction vector);

- Beam peak intensity relative to saturation intensity (I_{peak}/I_{sat}, being I_{peak} defined on equation 2);
- Gaussian beam $1/e^2$ diameter (ξ on equation 2) $[m]$.

- Atom mass $[kg]$;
- Transition linewidth $[Hz]$;
- Transition wavelength $[m]$;
- J quantum number for ground state;
- J quantum number for excited state;
- Landé g-factor for ground state;
- Landé g-factor for excited state;

- Initial position center (gaussian distribution of initial positions will be located on this position);
- Standard deviation of gaussian initial position distribution;
- Initial velocity offset (Maxwell-Boltzmann distribution will be added of this value);
- Temperature (initial velocities will be random and follow Maxwell-Boltzmann velocity distribution);

- Maximum number of iterations;
- Threshold distance to interrupt simulation;
- MOT center coordinates;
- Number of configurations to simulate;
- Number of averages for each simulation;
- Number of bins on each dimension of the position histogram;

4 Implementation

The simulation was implemented on C. The main code is written bellow.

4.1 Main code

```
/*
Magneto-Optical Trapping Simulation - Header file
Code release version: v1.1


Monte Carlo trajectory simulation for narrow-line MOTs.
Generates 2D x-z histogram files and trapping time files.



Ramon Gabriel Teixeira Rosa, PhD
Optics and Photonics Research Center (CePOF)
University of Sao Paulo - Sao Carlos Institute of Physics
ramongabriel.tr@usp.br
+55(16)3373.9810 (Ext: 225)

June 07, 2018
*/

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include "MOTsimHeader_v1.h"


///-- TITLE FOR OUTPUT FILE HEADER --///
char TITLE[] = "NoCollisionTest_Dy5B";
//without_spaces
//will be used on output folder name



///---- CONSTANTS ----///
const double h = 6.62607004e-34;
const double mub = 9.274009994e-24;
const double e = 1.60217662e-19;
```

```
40 const double kb = 1.38064852e-23;
41 const double c = 2.99792458e8;
42 // SI units
43
44
45 /// ---- ENVIRONMENT ---- ///
46 double gx=+0.0;
47 double gy=+0.0;
48 double gz=-9.8;
49 // gravity [m/s2]
50 double GBx = 0.046/2;
51 double GBy = 0.046/2;
52 double GBz = -0.046;
53 // Magnetic field gradient [T/m]    (1 G/cm = 0.01 T/m)
54 double B0[] = {0,0,0}; // DO NOT CHANGE BEFORE MODIFYING MAGNETIC FORCE ON
       DIPOLE TO INCLUDE BIAS!!!
55 // Magnetic field bias [T]    (1 G = 1e-4 T)
56 double detuning;
57 // units of Gamma
58 #define NUMBEAMS 6
59 double BEAMS[][3] = {
60         {-1, 0, 0},
61         {+1, 0, 0},
62         { 0,-1, 0},
63         { 0,+1, 0},
64         { 0, 0,-1},
65         { 0, 0,+1},
66 };
67 // All beams cross the point (0,0,0);
68 const char BEAMS_POL[NUMBEAMS] = "llllrr";  // 'r': RHCP
69                                             // 'l': LHCP
70                                             // 'x': polarization vector on
       plane defined by X and ek (beam propagation vector)
71                                             // 'y': [...]
72                                             // 'z': [...]
73
74 //const double so[] = {50,50,50,50,50,50};
75 const double so[] = {160,160,160,160,0,160};
76 //const double so[] = {0.6,0.6,0.6,0.6,0.6,0.6};
77 // peak intensity/saturation_intensity
78 const double BEAMS_WAIST[] = {0.036,0.036,0.036,0.036,0.036,0.036};
79 // 1/e2 gaussian beam waist [m]
80 double complex BEAMS_POL_VECTORS[NUMBEAMS][3];
81 ///---- ATOM ----///
82 // Dy
83 const double m = (1.660539040e-27)*163.9291748;
84 const double Gamma = 136e3;
```

```
85 const double lambda = 626e-9;
86 const double Jgnd = 8;
87 const double Jexc = 9;
88 const double gjg = 1.24;
89 const double gje = 1.29;
90
91 //Er
92 //const double m = (1.660539040e-27)*166.0;
93 //const double Gamma = 190e3;
94 //const double lambda = 583e-9;
95 //const double Jgnd = 6;
96 //const double Jexc = 7;
97 //const double gjg = 1.167;
98 //const double gje = 1.195;
99
100 ///--- INITIAL CONDITIONS ----///
101 double xx0c = 0;
102 double yy0c = 0;
103 double zz0c = 0;
104 // initial position
105 double vx0c = 0;
106 double vy0c = 0;
107 double vz0c = 0;
108 // inital velocity
109 double STDxx0 = 2e-3;
110 double STDyy0 = 2e-3;
111 double STDzz0 = 2e-3;
112 // standard deviation of initial position gaussian distribution (around [
    xx0c,yy0c,zz0c])
113 double T = 10e-6;
114 // Temperature [K]
115 // Velocity distribution following Maxwell-Boltzmann distribution with
    offset [vx0c,vy0c,vz0c];
116 double xx0,yy0,zz0,vx0,vy0,vz0;
117
118 ///---- SIMULATION ----///
119 const int NUMITER=1e5;
120 //maximum number of iterations
121 const double BOUNDARY = 20e-3;
122 const double xc = 0;
123 const double yc = 0;
124 const double zc = 0;
125 // if( sqrt((x-xc)^2 + (y-yc)^2 + (z-zc)^2) > BOUNDARY ), assume atom
    escaped trap
126 // [m]
127 const int NumVar = 31;
128 // number of different configurations
```

```
129 const int NumAvg = 250;
130 // averages per configuration
131 const double DetuningRange[] = {0,-150};
132 // detuning range in units of gamma;
133 const int PRINT_STEPBYSTEP = 0;
134 // print simulation results step by step (test and debug) [0,1]
135 ///---- RESULTS ----///
136 #define NumVoxels 200
137 double PositionHistogram[NumVoxels][NumVoxels][NumVoxels];
138 double xbins[NumVoxels];
139 double ybins[NumVoxels];
140 double zbins[NumVoxels];
141
142
143 /// ---- END OF USER DEFINED PARAMETERS ----///
144 /// ---- END OF USER DEFINED PARAMETERS ----///
145 /// ---- END OF USER DEFINED PARAMETERS ----///
146
147
148 // Definition of structs for return of functions
149 struct BeamSelection{
150     int BEAMindex;
151     double CycleTime;
152     int dmj;
153 };
154 struct SimulationResults{
155     int FLAG;
156     double TrapTime;
157     int Iterations;
158 };
159 struct CollisionMomentum{
160     double dvxc;
161     double dvyc;
162     double dvzc;
163 };
164
165
166
167
168
169 //Calculate magnetic fields given x,y,z
170 //Returns field strength B and modifies Bn[] (unit vector of field
    direction)
171 double MagneticField (double x,double y, double z,double Bn[]){
172
173     double Bx = GBx*x + B0[0];
174     double By = GBy*y + B0[1];
```

```
175     double Bz = GBz*z + B0[2];
176     double B;
177     double r0,r1,r2;
178 
179     B = sqrt(Bx*Bx + By*By + Bz*Bz);
180 
181     if(B==0){
182         r0 = (randr()-0.5)*2;
183         r1 = (randr()-0.5)*2;
184         r2 = (randr()-0.5)*2;
185         Bn[0] = r0 / sqrt(r0*r0 + r1*r1 + r2*r2);
186         Bn[1] = r1 / sqrt(r0*r0 + r1*r1 + r2*r2);
187         Bn[2] = r2 / sqrt(r0*r0 + r1*r1 + r2*r2);
188         return B;
189     }
190 
191     Bn[0]=Bx/B;
192     Bn[1]=By/B;
193     Bn[2]=Bz/B;
194 
195     return B;
196 }
197 
198 //Calculate polarization vectors on the lab frame
199 void CalculatePolarizationVectors(void){
200     int i;
201     int ERROR = 0;
202     double k[3],ex[3],ey[3];
203     double r[3];
204     double exek;
205 
206     char p;
207         // 'r': RHCP
208         // 'l': LHCP
209         // 'x': polarization vector on plane defined by X and ek (beam
    propagation vector)
210         // 'y': [...]
211         // 'z': [...]
212 
213 
214     for (i=0;i<NUMBEAMS;i++){
215         k[0] = BEAMS[i][0];
216         k[1] = BEAMS[i][1];
217         k[2] = BEAMS[i][2];
218         p = BEAMS_POL[i];
219 
220         if(p == 'r'){
```

```
221             // random ex not parallel to k
222             do{
223                 r[0] = (randr()-0.5)*2;
224                 r[1] = (randr()-0.5)*2;
225                 r[2] = (randr()-0.5)*2;
226                 ex[0] = r[0] / sqrt(dotproduct(r,r));
227                 ex[1] = r[1] / sqrt(dotproduct(r,r));
228                 ex[2] = r[2] / sqrt(dotproduct(r,r));
229                 exek = dotproduct(ex,k);
230             }while(abs(exek) == 1);
231             // orthogonalization
232             r[0] = ex[0] - exek*k[0];
233             r[1] = ex[1] - exek*k[1];
234             r[2] = ex[2] - exek*k[2];
235             // renormalization
236             ex[0] = r[0] / sqrt(dotproduct(r,r));
237             ex[1] = r[1] / sqrt(dotproduct(r,r));
238             ex[2] = r[2] / sqrt(dotproduct(r,r));
239             // ey = k(*)ex
240             ey[0] = k[1]*ex[2] - k[2]*ex[1];
241             ey[1] = k[2]*ex[0] - k[0]*ex[2];
242             ey[2] = k[0]*ex[1] - k[1]*ex[0];
243             // RHCP
244             BEAMS_POL_VECTORS[i][0] = sqrt(0.5)*(ex[0] - I*ey[0]);
245             BEAMS_POL_VECTORS[i][1] = sqrt(0.5)*(ex[1] - I*ey[1]);
246             BEAMS_POL_VECTORS[i][2] = sqrt(0.5)*(ex[2] - I*ey[2]);
247         }
248         else if(p == 'l'){
249             // random ex not parallel to k
250             do{
251                 r[0] = (randr()-0.5)*2;
252                 r[1] = (randr()-0.5)*2;
253                 r[2] = (randr()-0.5)*2;
254                 ex[0] = r[0] / sqrt(dotproduct(r,r));
255                 ex[1] = r[1] / sqrt(dotproduct(r,r));
256                 ex[2] = r[2] / sqrt(dotproduct(r,r));
257                 exek = ex[0]*k[0] + ex[1]*k[1] + ex[2]*k[2];
258             }while(exek == 1);
259             // orthogonalization
260             r[0] = ex[0] - exek*k[0];
261             r[1] = ex[1] - exek*k[1];
262             r[2] = ex[2] - exek*k[2];
263             // renormalization
264             ex[0] = r[0] / sqrt(dotproduct(r,r));
265             ex[1] = r[1] / sqrt(dotproduct(r,r));
266             ex[2] = r[2] / sqrt(dotproduct(r,r));
267             // ey = k(*)ex
```

```
268             ey[0] = k[1]*ex[2] - k[2]*ex[1];
269             ey[1] = k[2]*ex[0] - k[0]*ex[2];
270             ey[2] = k[0]*ex[1] - k[1]*ex[0];
271             // LHCP
272             BEAMS_POL_VECTORS[i][0] = sqrt(0.5)*(ex[0] + I*ey[0]);
273             BEAMS_POL_VECTORS[i][1] = sqrt(0.5)*(ex[1] + I*ey[1]);
274             BEAMS_POL_VECTORS[i][2] = sqrt(0.5)*(ex[2] + I*ey[2]);
275         }
276         else if(p == 'x' || p == 'y' || p == 'z'){
277             r[0]=0;
278             r[1]=0;
279             r[2]=0;
280             if(p=='x') r[0] = 1;
281             if(p=='y') r[1] = 1;
282             if(p=='z') r[2] = 1;
283 
284             if(abs(dotproduct(r,k))==1){
285                 ERROR = 1;
286                 printf("\n\n\nERROR: UNCONSISTENT POLARIZATION [BEAM #%d]\n
    \n",i);
287                 printf("Polarization: %c\n",p);
288                 printf("Beam propagation: [%+.2e %+.2e %+.2e]\n\n",k[0],k
    [1],k[2]);
289                 printf("Polarization vector cannot be parallel to beam
    propagation direction!\n\n\n\n");
290                 printf("Press any key to exit");
291                 getchar();
292                 printf("\n\n");
293             }
294             r[0] = r[0] - dotproduct(r,k)*k[0];
295             r[1] = r[1] - dotproduct(r,k)*k[1];
296             r[2] = r[2] - dotproduct(r,k)*k[2];
297             BEAMS_POL_VECTORS[i][0] = r[0]/sqrt(dotproduct(r,r));
298             BEAMS_POL_VECTORS[i][1] = r[1]/sqrt(dotproduct(r,r));
299             BEAMS_POL_VECTORS[i][2] = r[2]/sqrt(dotproduct(r,r));
300 
301         }
302         else{
303             ERROR = 1;
304             printf("\n\n\nERROR: POLARIZATION DEFINITION NOT RECOGNIZED [
    BEAM #%d]\n\n",i);
305             printf("Polarization: %c\n\n",p);
306             printf("Polarization must be defined as one of the following
    options:\n['r','l','x','y','z']\n\n\n\n");
307             printf("Press any key to exit");
308             getchar();
309             printf("\n\n");
```

```
310         }
311     }
312     if(ERROR) exit(-1);
313 }
314 
315 // Calculate transition vectors given magnetic field direction
316 void CalculateTransitionVectors (double Bn[],double complex polsm[],double
    complex polsp[],double complex polpi[]){
317     int i;
318     double r[3],ex[3],ey[3];
319     double exek;
320 
321     // polpi = Bn
322     for (i=0;i<3;i++)
323         polpi[i]=Bn[i];
324 
325     // random ex not parallel to k
326     do{
327         for(i=0;i<3;i++)
328             r[i] = (randr()-0.5)*2;
329         for(i=0;i<3;i++)
330             ex[i] = r[i] / sqrt(dotproduct(r,r));
331         exek = dotproduct(ex,Bn);
332     } while(abs(exek)==1);
333     // orthogonalization
334     for(i=0;i<3;i++)
335         r[i] = ex[i] - exek*Bn[i];
336     // renormalization
337     for(i=0;i<3;i++)
338         ex[i] = r[i] / sqrt(dotproduct(r,r));
339     // ey = k(*)ex
340     ey[0] = Bn[1]*ex[2] - Bn[2]*ex[1];
341     ey[1] = Bn[2]*ex[0] - Bn[0]*ex[2];
342     ey[2] = Bn[0]*ex[1] - Bn[1]*ex[0];
343     // SIGMA-
344     polsm[0] = sqrt(0.5)*(ex[0] - I*ey[0]);
345     polsm[1] = sqrt(0.5)*(ex[1] - I*ey[1]);
346     polsm[2] = sqrt(0.5)*(ex[2] - I*ey[2]);
347     // SIGMA+
348     polsp[0] = sqrt(0.5)*(ex[0] + I*ey[0]);
349     polsp[1] = sqrt(0.5)*(ex[1] + I*ey[1]);
350     polsp[2] = sqrt(0.5)*(ex[2] + I*ey[2]);
351 }
352 
353 // Selects beam to be absorbed given x,y,z,vx,vy,vz;
354 // Returns beam index, time step and transition type (delta mj)
```

```
355 struct BeamSelection ChooseBeam (double x,double y,double z,double vx,
    double vy, double vz){
356     int BEAMindex=0,i=0,j=0,iT=0;
357     double B=0.,Bn[]={0.,0.,0.},BEAM[3];
358     double dZeeman=0.,mje=0.,mjg=0.,sop=0.,delta=0.,Rscatt=0.,doppler;
359     double complex polsm[3];
360     double complex polsp[3];
361     double complex polpi[3];
362     double complex TransitionVectors[3][3]; //TransitionVectors[transition
    index][x,y,z]
363     double complex trpolvec[3];             //trpolvec[x,y,z]
364     double complex bPOL[3];
365     double cycletime[NUMBEAMS][3];          //TransitionVectors[beam index
    ][transition index]
366     double soeff,trpolvec_beampol;
367
368     double DEBUG1[NUMBEAMS][3];
369     double DEBUG2[NUMBEAMS][3];
370     double DEBUG3[NUMBEAMS][3];
371
372
373     B = MagneticField(x,y,z,Bn);
374     CalculateTransitionVectors(Bn,polsm,polsp,polpi);
375
376
377     for (i=0;i<3;i++){
378         TransitionVectors[0][i] = polsm[i];
379         TransitionVectors[1][i] = polpi[i];
380         TransitionVectors[2][i] = polsp[i];
381     }
382
383     mjg = -Jgnd; // <------- [#REVIEW][Ground state mj is being always
    defined as -Jgnd, but it is not in practice.]
384
385     for (i=0;i<NUMBEAMS;i++){
386         //BEAM
387         for (j=0;j<3;j++){
388             BEAM[j]=BEAMS[i][j];
389             bPOL[j]=BEAMS_POL_VECTORS[i][j];
390         }
391
392         //Gaussian beam;
393         sop = so[i]*exp(-2*(pow(BEAM[1]*z - BEAM[2]*y,2) + pow(BEAM[2]*x -
    BEAM[0]*z,2) + pow(BEAM[0]*y - BEAM[1]*x,2))/pow(BEAMS_WAIST[i],2));
394
395         // Loop over possible transitions
396         mje=mjg-1;
```

```
397         for (iT=0;iT<3;iT++){
398             if(fabs(mjg)>Jexc)
399                 cycletime[i][iT] = 1./0.;    // State does not exist
400             else{
401                 // State does exist
402                 for(j=0;j<3;j++)
403                     trpolvec[j] = TransitionVectors[iT][j];
404                 trpolvec_beampol = pow(AbsDotProductComplex(bPOL,trpolvec)
    ,2);
405                 dZeeman = mub*B*(gjg*mjg - gje*mje )/h;
406                 doppler = -(c/lambda)*(vx*BEAM[0] + vy*BEAM[1] + vz*BEAM
    [2])/c;
407                 delta = detuning*Gamma + dZeeman + doppler;
408                 soeff = sop*trpolvec_beampol;
409                 Rscatt = (Gamma/2)*soeff/( 1+soeff+(4*(delta*delta)/(Gamma*
    Gamma)) );
410                 cycletime[i][iT] = (1/Rscatt)*RandomExpDist();
411 
412                 DEBUG1[i][iT] = Rscatt;
413                 DEBUG2[i][iT] = delta/doppler;
414                 DEBUG3[i][iT] = dZeeman/doppler;
415 
416             }
417             mje++;
418         }
419 
420     }
421 
422 
423     int dmj[]={-1,0,1};
424     int xdmj = 0;
425     double dt = 1/0.;
426 
427     for (i=0;i<NUMBEAMS;i++){
428         for (iT=0;iT<3;iT++){
429             if(cycletime[i][iT]<dt){
430                 dt = cycletime[i][iT];
431                 xdmj = dmj[iT];
432                 BEAMindex = i;
433             }
434         }
435     }
436 
437 
438 
439     /// ---- #DEBUG ----///
440     if(PRINT_STEPBYSTEP){
```

```
441                 double AUX;
442                 static int numA=0,numB=0;
443                 printf("\n\n\n");
444                 printf("\n\n\nr = %+.2e  %+.2e  %+.2e\n",x,y,z);
445                 printf("v = %+.2e  %+.2e  %+.2e\n\n",vx,vy,vz);
446                 for (i=0;i<NUMBEAMS;i++){
447                     for (iT=0;iT<3;iT++){
448                         //printf("%+.2e  ",cycletime[i][iT]);
449                         printf("%+.8e  ",DEBUG1[i][iT]);
450                     }
451                     if(i==BEAMindex){
452                         printf("(dmj = %+d)  ",xdmj);
453                         AUX = BEAMS[i][0]*x + BEAMS[i][1]*y + BEAMS[i][2]*z
    ;
454                         if(AUX>0){
455                             printf("[+]");
456                             numA++;
457                         }
458                         if(AUX<0){
459                             printf("[-]");
460                             numB++;
461                         }
462                         if(AUX==0)
463                             printf("[0]");
464                     }
465                     printf("\n");
466                     for (iT=0;iT<3;iT++){
467                         //printf("%+.2e  ",cycletime[i][iT]);
468                         printf("%+.8e  ",DEBUG2[i][iT]);
469                     }
470                     printf("\n");
471                     for (iT=0;iT<3;iT++){
472                         //printf("%+.2e  ",cycletime[i][iT]);
473                         printf("%+.8e  ",DEBUG3[i][iT]);
474                     }
475                     printf("\n\n");
476                 }
477                 printf("\n\n[%d\t\t%d]\n",numA,numB);
478                 getchar();
479     }
480   /// ------------------ ///
481
482
483     struct BeamSelection result;
484     result.BEAMindex = BEAMindex;
485     result.CycleTime = dt;
486     result.dmj = xdmj;
```

```
487     return result;
488 }
489 
490 // Calculate collision momentum transfer assuming collision of the atom
    with another atom with a temperature T
491 /// ---- <<STILL BEING TESTED>> ---- ///
492 struct CollisionMomentum CalculateCollisions(double x, double y, double z,
    double vx, double vy, double vz, double dt){
493     // Canonical ensemble
494     struct CollisionMomentum pc;
495     double dv[3];
496     double v, tau;
497 
498     v = sqrt(vx*vx + vy*vy + vz*vz);
499     //tau = (0.40/v)*RandomExpDist();
500     tau = (5e-6/v)*RandomExpDist();
501 
502     if(tau<=dt){
503         dv[0] = gaussian()*sqrt(kb*T/m);
504         dv[1] = gaussian()*sqrt(kb*T/m);
505         dv[2] = gaussian()*sqrt(kb*T/m);
506     }
507     else{
508         dv[0] = 0;
509         dv[1] = 0;
510         dv[2] = 0;
511     }
512 
513     pc.dvxc=0*dv[0];
514     pc.dvyc=0*dv[1];
515     pc.dvzc=0*dv[2];
516 
517     return pc;
518 };
519 
520 //Simulates path of one atom with defined configuration
521 struct SimulationResults RunSimulation (){
522     struct SimulationResults SimRes;
523     if(abs(Jexc - Jgnd)!=1){
524         printf("\n\n\t\tError!\n\n\t\tJexc != Jgnd +- 1 \n\n");
525         SimRes.FLAG = -1;
526         SimRes.Iterations=0;
527         SimRes.TrapTime=0;
528         exit(-1);
529     }
530 
531     int i,BEAMindex;
```

```
532     double Bn[3],B,x,y,z,vx,vy,vz,t;
533     struct BeamSelection BEAMresult;
534     struct CollisionMomentum pCollision_Struct;
535     double r0,r1,r2,pxe,pye,pze,pxa,pya,pza;
536     int ix,iy,iz; //Histogram indexes
537     double TRAPTIME=0;
538     double dvxc,dvyc,dvzc;
539     int dmj=0;
540
541
542     double dvx,dvy,dvz,dt;
543     double Amagx,Amagy,Amagz;
544
545     // Initial conditions
546     t=0;
547     vx = vx0;
548     vy = vy0;
549     vz = vz0;
550     x = xx0;
551     y = yy0;
552     z = zz0;
553
554     int FLAG = 0;
555
556     for (i=0;i<NUMITER && FLAG==0;i++){
557         BEAMresult = ChooseBeam (x,y,z,vx,vy,vz);
558         BEAMindex = BEAMresult.BEAMindex;
559         dt = BEAMresult.CycleTime;
560         dmj = BEAMresult.dmj;
561         B = MagneticField(x,y,z,Bn);
562
563         // Emission in random direction
564         r0 = (randr()-0.5)*2;
565         r1 = (randr()-0.5)*2;
566         r2 = (randr()-0.5)*2;
567         pxe = (h/lambda)*r0/sqrt(r0*r0 + r1*r1 + r2*r2);
568         pye = (h/lambda)*r1/sqrt(r0*r0 + r1*r1 + r2*r2);
569         pze = (h/lambda)*r2/sqrt(r0*r0 + r1*r1 + r2*r2);
570         // Absorption
571         pxa = (h/lambda)*BEAMS[BEAMindex][0];
572         pya = (h/lambda)*BEAMS[BEAMindex][1];
573         pza = (h/lambda)*BEAMS[BEAMindex][2];
574         // Collisions
575         pCollision_Struct = CalculateCollisions(x,y,z,vx,vy,vz,dt);
576         dvxc=pCollision_Struct.dvxc;
577         dvyc=pCollision_Struct.dvyc;
578         dvzc=pCollision_Struct.dvzc;
```

```
579
580         dvx = (pxe + pxa)/m + dvxc;
581         dvy = (pye + pya)/m + dvyc;
582         dvz = (pze + pza)/m + dvzc;
583
584         Amagx = -gjg*Jgnd*mub*(GBx*GBx*x)/sqrt(pow(GBx*x,2) + pow(GBy*y,2)
    + pow(GBz*z,2));
585         Amagy = -gjg*Jgnd*mub*(GBy*GBy*y)/sqrt(pow(GBx*x,2) + pow(GBy*y,2)
    + pow(GBz*z,2));
586         Amagz = -gjg*Jgnd*mub*(GBz*GBz*z)/sqrt(pow(GBx*x,2) + pow(GBy*y,2)
    + pow(GBz*z,2));
587
588          x += vx*dt + (gx/2)*dt*dt + (Amagx/2)*dt*dt;
589          y += vy*dt + (gy/2)*dt*dt + (Amagy/2)*dt*dt;
590          z += vz*dt + (gz/2)*dt*dt + (Amagz/2)*dt*dt;
591         vx += gx*dt + dvx + Amagx*dt;
592         vy += gy*dt + dvy + Amagy*dt;
593         vz += gz*dt + dvz + Amagz*dt;
594          t += dt;
595
596
597
598          if((vx*x + vy*y + vz*z)<0)
599             TRAPTIME = t;
600
601
602
603         if(sqrt((x-xc)*(x-xc) + (y-yc)*(y-yc) + (z-zc)*(z-zc))>BOUNDARY)
604             FLAG = 1;
605         else{
606             ix = round((x+BOUNDARY)*(NumVoxels-1)/(2*BOUNDARY));
607             iy = round((y+BOUNDARY)*(NumVoxels-1)/(2*BOUNDARY));
608             iz = round((z+BOUNDARY)*(NumVoxels-1)/(2*BOUNDARY));
609             PositionHistogram[ix][iy][iz] += dt;
610         }
611     }
612
613     SimRes.FLAG = FLAG;
614     SimRes.TrapTime = TRAPTIME;
615     SimRes.Iterations = i;
616     return SimRes;
617 }
618
619 // Write output file header
620 void WriteResultsHeader(FILE *fid){
621     int i;
622     time_t t = time(NULL);
```

```
623     struct tm tm = *localtime(&t);
624
625     fprintf(fid,"%s;\n",TITLE);
626     fprintf(fid,"Data starts after exclamation mark;\n");
627     fprintf(fid,"%d-%02d-%02d %02d:%02d:%02d;\n",tm.tm_year+1900,tm.tm_mon
        +1,tm.tm_mday,tm.tm_hour,tm.tm_min,tm.tm_sec);
628     fprintf(fid,"Gravity = (%f,%f,%f) m/s2;\n",gx,gy,gz);
629     fprintf(fid,"Magnetic field gradient = (%e,%e,%e) T/m;\n",GBx,GBy,GBz);
630     fprintf(fid,"Magnetic field bias = (%e %e %e) T;\n",B0[0],B0[1],B0[2]);
631     fprintf(fid,"Detuning = %f (units of gamma);\n",detuning);
632     fprintf(fid,"Number of beams = %d;\n",NUMBEAMS);
633
634     fprintf(fid,"BEAMS: propagation direction = \n");
635     fprintf(fid,"((%+f,%+f,%+f)",BEAMS[0][0],BEAMS[0][1],BEAMS[0][2]);
636     for (i=1;i<NUMBEAMS;i++)
637         fprintf(fid,",\n(%+f,%+f,%+f)",BEAMS[i][0],BEAMS[i][1],BEAMS[i][2])
        ;
638     fprintf(fid,");\n");
639
640     fprintf(fid,"BEAMS: polarization definition = (%c",BEAMS_POL[0]);
641     for (i=1;i<NUMBEAMS;i++)
642         fprintf(fid,",%c",BEAMS_POL[i]);
643     fprintf(fid,");\n");
644
645     fprintf(fid,"BEAMS: polarization vectors = \n");
646     fprintf(fid,"((%+f%+fi,%+f%+fi,%+f%+fi)",creal(BEAMS_POL_VECTORS[0][0])
        ,cimag(BEAMS_POL_VECTORS[0][0]),creal(BEAMS_POL_VECTORS[0][1]),cimag(
        BEAMS_POL_VECTORS[0][1]),creal(BEAMS_POL_VECTORS[0][2]),cimag(
        BEAMS_POL_VECTORS[0][2]));;
647     for (i=1;i<NUMBEAMS;i++)
648         fprintf(fid,",\n(%+f%+fi,%+f%+fi,%+f%+fi)",creal(BEAMS_POL_VECTORS[
        i][0]),cimag(BEAMS_POL_VECTORS[i][0]),creal(BEAMS_POL_VECTORS[i][1]),
        cimag(BEAMS_POL_VECTORS[i][1]),creal(BEAMS_POL_VECTORS[i][2]),cimag(
        BEAMS_POL_VECTORS[i][2]));;
649     fprintf(fid,");\n");
650
651     fprintf(fid,"BEAMS: peak intensity/saturation intensity = (%f",so[0]);
652     for (i=1;i<NUMBEAMS;i++)
653         fprintf(fid,",%f",so[i]);
654     fprintf(fid,");\n");
655     fprintf(fid,"BEAMS: 1/e2 waist = (%e",BEAMS_WAIST[0]);
656     for (i=1;i<NUMBEAMS;i++)
657         fprintf(fid,",%e",BEAMS_WAIST[i]);
658     fprintf(fid,") m;\n");
659     fprintf(fid,"ATOM: mass = %e kg;\n",m);
660     fprintf(fid,"ATOM: transition gamma = %e Hz;\n",Gamma);
661     fprintf(fid,"ATOM: transition wavelength = %e m;\n",lambda);
```

```
662     fprintf(fid,"ATOM: J (ground) = %d;\n",(int)Jgnd);
663     fprintf(fid,"ATOM: J (excited) = %d;\n",(int)Jexc);
664     fprintf(fid,"ATOM: g_lande (ground) = %f;\n",gjg);
665     fprintf(fid,"ATOM: g_lande (excited) = %f;\n",gje);
666     fprintf(fid,"SIMULATION: maximum number of iterations = %d;\n",NUMITER)
    ;
667     fprintf(fid,"SIMULATION: boundary radius = %e m;\n",BOUNDARY);
668     fprintf(fid,"SIMULATION: boundary center = (%e,%e,%e) m;\n",xc,yc,zc);
669     fprintf(fid,"SIMULATION: initial position = (%e,%e,%e) m;\n",xx0,yy0,
    zz0);
670     fprintf(fid,"SIMULATION: initial velocity = (%e,%e,%e) m/s;\n",vx0,vy0,
    vz0);
671     fprintf(fid,"<<<DATA>>>!\n");
672 }
673
674 // Main
675 int main (void){
676     srand(time(NULL));
677     struct SimulationResults SimRes;
678     int i,j,k;
679     int ix,iy,iz;
680     double PosHist2Dxy[NumVoxels][NumVoxels],PosHist2Dxz[NumVoxels][
    NumVoxels],PosHist2Dyz[NumVoxels][NumVoxels];
681
682     // Magnetic field check
683     if(GBx+GBy+GBz!=0){
684         printf("Magnetic field divergence != 0\n\n GBx + GBy + GBz !=0\b\b"
    );
685         getchar();
686         exit(-1);
687     }
688
689     ///---- Files ----///
690     time_t datetime;
691     struct tm tm;
692     double clock1,clock2;
693     datetime = time(NULL);
694     tm = *localtime(&datetime);
695     clock1 = clock();
696
697     char OUTPUTFOLDER[200];
698     char TRAPTIMEFILE[400];
699
700     sprintf(OUTPUTFOLDER,"./Results/%s_%04d%02d%02d_%02d%02d%02d",TITLE,tm.
    tm_year+1900,tm.tm_mon+1,tm.tm_mday,tm.tm_hour,tm.tm_min,tm.tm_sec);
701     sprintf(TRAPTIMEFILE,"%s/Trapping_Time.dat",OUTPUTFOLDER);
702
```

```
703     mkdir(OUTPUTFOLDER);
704
705     FILE *fLOG;
706     fLOG = fopen("outputlog.txt","a");
707     // log file
708     FILE *fTT;
709     fTT = fopen(TRAPTIMEFILE,"w");
710     // traptime file
711
712     if(fTT == NULL){
713         printf("\n\n\n\n FAILED TO CREATE:\n%s \n\n\n\n\n",TRAPTIMEFILE);
714         printf("Press <enter> to continue");
715         getchar();
716         printf("\n\n\n\n\n\n\n\n\n");
717     }
718
719
720
721
722
723     fprintf(fLOG,"\n\n\n\n\n\n\n\n\n");
724     for (k=0;k<100;k++)
725         fprintf(fLOG,"#");
726     fprintf(fLOG,"\n<<<Simulation started (%d_%02d_%02d %02d:%02d:%02d)>>>\
    n",tm.tm_year+1900,tm.tm_mon+1,tm.tm_mday,tm.tm_hour,tm.tm_min,tm.tm_sec
    );
727     /// ---------------- ///
728
729
730     // Generate xbins,ybins,zbins
731     for (k=0;k<NumVoxels;k++){
732         xbins[k] = -BOUNDARY + (2*BOUNDARY)*k/NumVoxels;
733         ybins[k] = -BOUNDARY + (2*BOUNDARY)*k/NumVoxels;
734         zbins[k] = -BOUNDARY + (2*BOUNDARY)*k/NumVoxels;
735     }
736     // Normalize BEAMS
737     double S;
738     for (i=0;i<NUMBEAMS;i++){
739         S=0;
740         for (j=0;j<3;j++)
741             S += BEAMS[i][j]*BEAMS[i][j];
742         S = sqrt(S);
743         for (j=0;j<3;j++)
744             BEAMS[i][j] = BEAMS[i][j]/S;
745     }
746
747
```

```
748     CalculatePolarizationVectors();
749
750
751     double detuningVar[NumVar];
752     linspace(DetuningRange[0],DetuningRange[1],NumVar,detuningVar);
753
754
755     // Execute simulation multiple times for averaging and varying detuning
756     for (j=0;j<NumVar;j++){
757         detuning = detuningVar[j];
758
759         fprintf(fTT,"%e\t",detuning);
760
761         // Zero position histograms (every
762         for (ix=0;ix<NumVoxels;ix++){
763             for (iy=0;iy<NumVoxels;iy++){
764                 PosHist2Dxy[ix][iy]=0;
765                 PosHist2Dxz[ix][iy]=0;
766                 PosHist2Dyz[ix][iy]=0;
767                 for (iz=0;iz<NumVoxels;iz++){
768                     PositionHistogram[ix][iy][iz]=0;
769                 }
770             }
771         }
772
773         printf("\n Detuning = %f\n\n",detuning);
774         fprintf(fLOG,"\n Detuning = %f (units of Gamma)\n\n",detuning);
775
776         for (i=0;i<NumAvg;i++){
777             //Random starting position and velocity
778             xx0 = xx0c + gaussian()*STDxx0;
779             yy0 = yy0c + gaussian()*STDyy0;
780             zz0 = zz0c + gaussian()*STDzz0;
781             vx0 = vx0c + gaussian()*sqrt(kb*T/m);
782             vy0 = vy0c + gaussian()*sqrt(kb*T/m);
783             vz0 = vz0c + gaussian()*sqrt(kb*T/m);
784
785
786             SimRes = RunSimulation();
787                 printf("[%-5d/%d var; %5d/%d avg]  (%d)  %-8d  %.3e s\n",
    j+1,NumVar,i+1,NumAvg,SimRes.FLAG,SimRes.Iterations,SimRes.TrapTime);
788             fprintf(fLOG,"[%-5d/%d var; %5d/%d avg]  (%d)  %-8d  %.3e s\n",
    j+1,NumVar,i+1,NumAvg,SimRes.FLAG,SimRes.Iterations,SimRes.TrapTime);
789             fprintf(fTT,"%e\t",SimRes.TrapTime);
790         }
791         fprintf(fTT,"\n");
792
```

```
793         for (ix=0;ix<NumVoxels;ix++){
794             for (iy=0;iy<NumVoxels;iy++){
795                 for (iz=0;iz<NumVoxels;iz++){
796                     PosHist2Dxy[ix][iy] += PositionHistogram[ix][iy][iz];
797                     PosHist2Dxz[ix][iz] += PositionHistogram[ix][iy][iz];
798                     PosHist2Dyz[iy][iz] += PositionHistogram[ix][iy][iz];
799                 }
800             }
801         }
802 
803 
804 
805         /// ---- 2D HISTOGRAMS FILES ----///
806             FILE *fid;
807             char filename[200];
808             sprintf(filename,"%s/det%f.dat",OUTPUTFOLDER,detuning);
809             fid = fopen(filename,"w");
810             // 2D histograms files
811 
812             if(fid == NULL){
813                 printf("\n\n\n\n FAILED TO CREATE:\n%s \n\n\n\n\n\n",filename
    );
814                 printf("Press <enter> to continue");
815                 getchar();
816                 printf("\n\n\n\n\n\n\n\n\n");
817             }
818 
819             WriteResultsHeader(fid);
820             fprintf(fid,"%e\t",.0);
821 
822             for (ix=0;ix<NumVoxels;ix++){
823                 fprintf(fid,"%e\t",xbins[ix]);
824             }
825             fprintf(fid,"\n");
826             for (iy=0;iy<NumVoxels;iy++){
827                 fprintf(fid,"%e\t",ybins[iy]);
828                 for (ix=0;ix<NumVoxels;ix++){
829                         fprintf(fid,"%e\t",PosHist2Dxz[ix][iy]);
830                 }
831                 fprintf(fid,"\n");
832             }
833             fflush(fid);
834             fclose(fid);
835         /// ---------------- ///
836 
837 
838 
```

```
839     } //var
840
841     datetime = time(NULL);
842     tm = *localtime(&datetime);
843     clock2 = clock();
844     fprintf(fLOG,"\n\nElapsed time: %f s\n",(clock2 - clock1)/
        CLOCKS_PER_SEC);
845     fprintf(fLOG,"<<<Simulation ended (%d-%02d-%02d %02d:%02d:%02d)>>>\n",
        tm.tm_year+1900,tm.tm_mon+1,tm.tm_mday,tm.tm_hour,tm.tm_min,tm.tm_sec);
846     for (k=0;k<100;k++)
847         fprintf(fLOG,"#");
848     fprintf(fLOG,"\n\n\n\n\n\n\n\n\n");
849     fflush(fLOG);
850     fclose(fLOG);
851
852     return 0;
853 }
```

4.2 Header file

The header file `MOTsimHeader_v1.h` contains some functions necessary for the execution of the main code. The header file content is written bellow.

```
1  /*
2  Magneto-Optical Trapping Simulation - Header file
3  Header release version: v1.0
4
5
6  Contains auxiliary functions for the Magneto-Optical Trapping Simulation
       code.
7
8
9
10
11 Ramon Gabriel Teixeira Rosa, PhD
12 Optics and Photonics Research Center (CePOF)
13 University of Sao Paulo - Sao Carlos Institute of Physics
14 ramongabriel.tr@usp.br
15 +55(16)3373.9810 (Ext: 225)
16
17 May 29, 2018
18 */
19
20 #include <stdio.h>
21 #include <math.h>
22 #include <time.h>
```

```
#include <stdlib.h>
#include <complex.h>
#define pi 3.14159265359

//Creates evenly spaced array
void linspace(double xi,double xf,int N,double x[]){
    int i;

    if(N==1)
        x[0] = xi;
    else{
        for (i=0;i<N;i++)
            x[i] = xi + (xf-xi)*i/(N-1);
    }
}

//Pipes data to gnuplot (not being used)
void plotGNU (double x[], double y[],int NUM_ELEMENTS,char options[]){
    int i;

    FILE *gp = popen("\"C:/Program Files/gnuplot/bin/gnuplot.exe\" -
    persistent", "w");
    if(gp==NULL)
        printf("\n\nERROR OPENING GNUPLOT\n\n");
    else{
        fprintf(gp,"%s\n",options);
        fprintf(gp,"plot '-' with lines\n");
        for (i=0;i<NUM_ELEMENTS;i++){
            fprintf(gp,"%f %f\n",(double)x[i],(double)y[i]);
            //printf("%f %f\n",(double)x[i],(double)y[i]);
        }
        fprintf(gp, "e\n");
        fflush(gp);
    }
}

// Returns random number on the interval [0,1]
double randr(){
    return ((double)rand() / (double)(RAND_MAX));
}

// Returns random number following a exponential decay probability function
    with average 1
double RandomExpDist(){
    double tau;
    tau = -log(1- ((double)rand() / (double)(RAND_MAX + 1)) );
    return tau;
```

```
68 }
69
70 // Returns random number following gaussian distribution with standard
       deviation 1 and mean 0
71 double gaussian(void){
72     //Box-Muller transform
73     static double v, fac;
74     static int phase = 0;
75     double S, Z, U1, U2, u;
76
77     if (phase)
78         Z = v * fac;
79     else
80     {
81         do
82         {
83             U1 = (double)rand() / RAND_MAX;
84             U2 = (double)rand() / RAND_MAX;
85
86             u = 2. * U1 - 1.;
87             v = 2. * U2 - 1.;
88             S = u * u + v * v;
89         } while(S >= 1);
90
91         fac = sqrt (-2. * log(S) / S);
92         Z = u * fac;
93     }
94
95     phase = 1 - phase;
96
97     return Z;
98 }
99
100 // Returns number following Maxwell-Boltzmann velocity distribution using
       sqrt(k*T/m)=1
101 double MaxwellBoltzmann(){
102     // Multiply output by k*T/m for correct velocity distribution
103     double x1 = gaussian();
104     double x2 = gaussian();
105     double x3 = gaussian();
106     return sqrt(x1*x1 + x2*x2 + x3*x3);
107 }
108
109 // Returns absolute value of inner product between complex vectors of size
       3
110 double AbsDotProductComplex(double complex A[],double complex B[]){
111     double complex S=0;
```

```
112     int i;
113     for (i=0;i<3;i++)
114         S = S + A[i]*conj(B[i]);
115     return (double)cabs(S);
116 }
117 
118 // Returns dot product between vectors of size 3
119 double dotproduct(double A[],double B[]){
120     double S=0;
121     int i;
122     for (i=0;i<3;i++)
123         S = S + A[i]*B[i];
124     return S;
125 }
126 
127 // Returns norm of vector of size 3
128 double norm (double v[]){
129     return sqrt(dotproduct(v,v));
130 }
```

4.3 Matlab code for visualizing results

A Matlab function was written to read and plot the results generated during the simulation. The function `MOTsim_ViewResults1.m` is written bellow.

```
1  function [DT,TTm] = MOTsim_ViewResults1 (plotresults)
2    if(nargin==0)
3         plotresults=1;
4    end
5 
6     close all;
7 
8     OVERLAY_XY_HIST = 0;
9 
10    %% Select folder with results
11    PATH = uigetdir('../Results/','Select folder');
12    %PATH = 'G:\Meu Drive\PosDoc\2018\Simulacoes\MOT_Simulation_v1\Results\
     TESTE';
13 
14    %% Get files names
15    FILES = ls(strcat(PATH,'/*.dat'));
16    N = size(FILES,1);
17    for k=1:N
18         if(strcmp(strtrim(FILES(k,:)),'Trapping_Time.dat'))
19             FILES = FILES([1:k-1,k+1:N],:);
```

```
            break;
        end
    end
    N = N-1;


    %% Read
    wbh = waitbar(0,'Loading files');
    filename = strcat(PATH,'/',strtrim(FILES(1,:)));
    M = importdata(filename);
    HEADER = M.textdata;
    DATA = M.data;
    splL7 = strsplit(HEADER{7},{'=','('});
    detuning = str2double(splL7{2});
    H = DATA(2:end,2:end);
    H = H/sum(H(:));
    x = DATA(1,2:end);
    z = DATA(2:end,1);
    x=x(:);
    z=z(:);
    NX = size(H,1);
    NY = size(H,2);

    MH = zeros(NX,NY,N);
    MD = zeros(N,1);

    MH(:,:,1) = H;
    MD(1) = detuning;



    waitbar(1,wbh);
    for k=2:N
        filename = strcat(PATH,'/',strtrim(FILES(k,:)));
        M = importdata(filename);
        HEADER = M.textdata;
        DATA = M.data;
        splL7 = strsplit(HEADER{7},{'=','('});
        detuning = str2double(splL7{2});
        H = DATA(2:end,2:end);
        H = H/sum(H(:));
        MH(:,:,k) = H;
        MD(k) = detuning;
        waitbar(k/N,wbh);
    end
    close(wbh);

```

```
67
68  %% Read Trapping_Time file
69    MT = importdata(strcat(PATH,'/','Trapping_Time.dat'));
70    DT = MT(:,1);
71    TT = MT(:,2:end);
72    [MD,siMD] = sort(MD);
73    MH = MH(:,:,siMD);
74    [DT,siDT] = sort(DT);
75    TT = TT(siDT,:);
76    TTm = mean(TT,2);
77    FILES=FILES(siMD,:);
78
79    Navg = size(TT,2);
80
81    TTbinsMAX = 6*max(TTm(:));
82    TTbins = linspace(0,TTbinsMAX,50);
83    TTcounts = hist(TT',TTbins);
84
85    %% Calculate z-mean and z-std
86    zmean=bsxfun(@rdivide,sum(sum(bsxfun(@times,z,MH),1),2),sum(sum(MH,1)
      ,2));
87    zmean=zmean(:);
88
89    zstd=sqrt(bsxfun(@rdivide,sum(sum(bsxfun(@times,z.^2,MH),1),2),sum(sum(
      MH,1),2)) - bsxfun(@rdivide,sum(sum(bsxfun(@times,z,MH),1),2),sum(sum(MH
      ,1),2)).^2);
90    zstd=zstd(:);
91
92    if(plotresults)
93        if(N>1)
94            %% Plot
95            fh1=figure('units','normalized','outerposition',[0.50 0.05 0.50
       0.45]); %#ok
96                plot(DT,TTm,'Marker','o','MarkerEdgeColor',[0 0 0],'
      MarkerFaceColor',[.49 1 .63],'LineStyle','-','Linewidth',2,'Color',[0 0
      1]);
97                xlim([min(DT),max(DT)]);
98                ylim([0,1.05*max(TTm)]);
99                set(gca, 'XDir','reverse');
100               xlabel('Detuning (units of \Gamma)');
101               ylabel('Average trapping time (s)');
102
103           Qx=1;
104           fh2=figure('units','normalized','outerposition',[0.20 0.50 0.60
       0.50]);
105           fh2sp1 = subplot(1,2,1);
106           fh2sp2 = subplot(1,2,2);
```

```
107             set(fh2sp1,'units','normalized','position',[0.05 0.10 0.40
    0.80]);
108             set(fh2sp2,'units','normalized','position',[0.50 0.10 0.30
    0.80]);
109             detslide = uicontrol('Style','slider','min',1,'max',N,'
    SliderStep',[1/N 5/N],'Value',1,'units','normalized','Position',[0.85
    0.80 0.14 0.03],'Callback',@UpdateHist2D);
110             uicontrol('Style','text','String','Detuning','units','
    normalized','Position',[0.85 0.83 0.15 0.03],'HorizontalAlignment','Left
    ');
111             ovrlcontrol = uicontrol('Style','checkbox','min',0,'max',1,'
    value',0,'units','normalized','Position',[0.85 0.70 0.04 0.04],'Callback
    ',@UpdateHist2D);
112             overlslide = uicontrol('Style','slider','min',0,'max',1,'Value'
    ,0.1,'units','normalized','Position',[0.87 0.70 0.10 0.03],'Callback',
    @UpdateHist2D);
113             uicontrol('Style','text','String','Overlay 1D x/z histograms','
    units','normalized','Position',[0.85 0.74 0.15 0.03],'
    HorizontalAlignment','Left');
114             uicontrol('Style','pushbutton','String','Save results','units',
    'normalized','position',[0.85 0.5 0.14 0.06],'Callback',@SaveAllResults)
    ;
115             UpdateHist2D();
116
117             fh3=figure('units','normalized','outerposition',[0.00 0.05 0.50
     0.45]);  %#ok
118                 errorbar(MD,zmean*1000,zstd*1000,'Marker','o','
    MarkerFaceColor',[1 0.3 0.3],'MarkerEdgeColor',[0 0 0],'color',[0 0 0],'
    linestyle','-');
119                 set(gca, 'XDir','reverse');
120                 xlabel('Detuning (units of \Gamma)');
121                 ylabel('{\langle}z{\rangle} (mm)');
122         else
123             imagesc(x*1e3,z*1e3,MH(:,:));
124             xlabel('x (mm)');
125             ylabel('z (mm)');
126             set(gca, 'YDir','normal');
127             colorbar;
128             axis image;
129             title(sprintf('Detuning = %.0f \\Gamma',MD(1)));
130         end
131     end
132
133
134
135     function UpdateHist2D (~,~)
136         Qx = round(get(detslide,'Value'));
```

```
137         set(detslide,'Value',Qx);
138         Q = N+1-Qx;
139 
140         OVERLAY_XY_HIST = get(ovrlcontrol,'Value');
141         SCALEOVERLAY  = get(overlslide,'Value')*abs(x(1)-x(end))*1000;
142 
143         figure(fh2);
144 
145         subplot(fh2sp1);
146             imagesc(x*1e3,z*1e3,MH(:,:,Q));
147             xlabel('x (mm)');
148             ylabel('z (mm)');
149             set(gca, 'YDir','normal');
150             colorbar;
151             axis image;
152             title(sprintf('Detuning = %.0f \\Gamma',MD(Q)));
153             if(OVERLAY_XY_HIST)
154                 hold on;
155                     Xc = sum(MH(:,:,Q),1);
156                     Zc = sum(MH(:,:,Q),2);
157                     Xc = Xc'/max(Xc(:));
158                     Zc = Zc/max(Zc(:));
159                     fillh1 = fill(1000*[x;x(end);x(1);],[Xc;Xc(end);Xc(1)]*
    SCALEOVERLAY + 1000*min(z),[1 0 0]);
160                     alpha(fillh1,0.3);
161                     plot(x*1000,Xc*SCALEOVERLAY + 1000*min(z),'color',[1 0
    0],'linewidth',2);
162                     fillh2 = fill([Zc;Zc(end);Zc(1)]*SCALEOVERLAY + 1000*
    min(x),[z;z(end);z(1)]*1000,[1 0 0]);
163                     alpha(fillh2,0.3);
164                     plot(Zc*SCALEOVERLAY + 1000*min(x),z*1000,'color',[1 0
    0],'linewidth',2);
165                 hold off;
166             end
167 
168         subplot(fh2sp2);
169             bar(TTbins,100*TTcounts(:,Q)/Navg,'BarWidth',1,'FaceColor',[0.5
     0.5 1],'EdgeColor',[0 0 1]);
170             xlim([0,TTbinsMAX]);
171             ylim([0,1.05*100*max(TTcounts(:,Q))]/Navg);
172             xlabel('Trapping time (s)');
173             ylabel('Relative frequency (%)');
174 
175     end
176 
177 
178 
```

```
179     function SaveAllResults(~,~)
180         if(exist(strcat(PATH,'\Results\'),'dir')==0)
181             mkdir(strcat(PATH,'\Results\'));
182         end
183 
184         OVERLAY_XY_HIST = get(ovrlcontrol,'Value');
185         SCALEOVERLAY    = get(overlslide,'Value')*abs(x(1)-x(end))*1000;
186 
187         close all;
188 
189 
190 
191         fprintf('Saving: %4.1f%%\n',((0)/(N+2))*100);
192         fhH1 = figure('visible','off');
193             plot(DT,TTm,'Marker','o','MarkerEdgeColor',[0 0 0],'
    MarkerFaceColor',[.49 1 .63],'LineStyle','-','Linewidth',2,'Color',[0 0
    1]);
194             xlim([min(DT),max(DT)]);
195             ylim([0,1.05*max(TTm)]);
196             set(gca, 'XDir','reverse');
197             xlabel('Detuning (units of \Gamma)');
198             ylabel('Average trapping time (s)');
199 
200             set(gcf, 'PaperUnits','inches');
201             set(gcf, 'PaperPositionMode','Manual');
202             set(gcf, 'PaperPosition', [0 0 6 4]);
203             print(fhH1,strcat(PATH,'\Results\TrappingTime.png'),'-r500','-
    dpng');
204         fprintf('Saving: %4.1f%%\n',((1)/(N+2))*100);
205 
206         fhH3 = figure('visible','off');
207             errorbar(MD,zmean*1000,zstd*1000,'Marker','o','MarkerFaceColor'
    ,[1 0.3 0.3],'MarkerEdgeColor',[0 0 0],'color',[0 0 0],'linestyle','-');
208             set(gca, 'XDir','reverse');
209             xlabel('Detuning (units of \Gamma)');
210             ylabel('{\langle}z{\rangle} (mm)');
211 
212             set(gcf, 'PaperUnits','inches');
213             set(gcf, 'PaperPositionMode','Manual');
214             set(gcf, 'PaperPosition', [0 0 6 4]);
215             print(fhH3,strcat(PATH,'\Results\z_avg_std.png'),'-r500','-dpng
    ');
216         fprintf('Saving: %4.1f%%\n',((2)/(N+2))*100);
217 
218         fhH2 = figure('visible','off');
219         fhH2sp1=subplot(1,2,1);
220         fhH2sp2=subplot(1,2,2);
```

```
221         set(fhH2sp1,'units','normalized','position',[0.05  0.15  0.55  0.8]);
222         set(fhH2sp2,'units','normalized','position',[0.70  0.15  0.25  0.8]);
223 
224         for n=1:N
225             subplot(fhH2sp1);
226                 imagesc(x*1e3,z*1e3,MH(:,:,n));
227                 xlabel('x (mm)');
228                 ylabel('z (mm)');
229                 set(gca, 'YDir','normal');
230                 colorbar;
231                 axis image;
232                 title(sprintf('Detuning = %.0f \\Gamma',MD(n)));
233                 if(OVERLAY_XY_HIST)
234                     hold on;
235                     Xc = sum(MH(:,:,n),1);
236                     Zc = sum(MH(:,:,n),2);
237                     Xc = Xc'/max(Xc(:));
238                     Zc = Zc/max(Zc(:));
239                     fillh1 = fill(1000*[x;x(end);x(1);],[Xc;Xc(end);Xc(1)]*
    SCALEOVERLAY + 1000*min(z),[1 0 0]);
240                     alpha(fillh1,0.3);
241                     plot(x*1000,Xc*SCALEOVERLAY + 1000*min(z),'color',[1 0
    0],'linewidth',2);
242                     fillh2 = fill([Zc;Zc(end);Zc(1)]*SCALEOVERLAY + 1000*
    min(x),[z;z(end);z(1)]*1000,[1 0 0]);
243                     alpha(fillh2,0.3);
244                     plot(Zc*SCALEOVERLAY + 1000*min(x),z*1000,'color',[1 0
    0],'linewidth',2);
245                     hold off;
246                 end
247 
248             subplot(fhH2sp2);
249                 bar(TTbins,100*TTcounts(:,n)/Navg,'BarWidth',1,'FaceColor'
    ,[0.5  0.5  1],'EdgeColor',[0  0  1]);
250                 xlim([0,TTbinsMAX]);
251                 ylim([0,1.05*100*max(TTcounts(:,n))]/Navg);
252                 xlabel('Trapping time (s)');
253                 ylabel('Relative frequency (%)');
254 
255                 set(gcf, 'PaperUnits','inches');
256                 set(gcf, 'PaperPositionMode','Manual');
257                 set(gcf, 'PaperPosition', [0 0 8 4]);
258                 [~,flnm,~] = fileparts(FILES(n,:));
259                 print(fhH2,strcat(PATH,'\Results\',flnm,'.png'),'-r500','-
    dpng');
260             fprintf('Saving: %4.1f%%\n',((2+n)/(N+2))*100);
261         end
```

```
262     end
263 end
```

References

[1] Christopher J Foot. *Atomic physics*, volume 7. Oxford University Press, 2005.

[2] Andrew J Berglund, James L Hanssen, and Jabez J McClelland. Narrow-line magneto-optical cooling and trapping of strongly magnetic atoms. *Physical review letters*, 100(11):113002, 2008.

[3] George EP Box, Mervin E Muller, et al. A note on the generation of random normal deviates. *The annals of mathematical statistics*, 29(2):610–611, 1958.

[4] Ryan K Hanley, Paul Huillery, Niamh C Keegan, Alistair D Bounds, Danielle Boddy, Riccardo Faoro, and Matthew PA Jones. Quantitative simulation of a magneto-optical trap operating near the photon recoil limit. *Journal of Modern Optics*, 65(5-6):667–676, 2018.

[5] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986.

[6] P Ilzhöfer, G Durastante, A Patscheider, A Trautmann, MJ Mark, and F Ferlaino. Two-species five-beam magneto-optical trap for erbium and dysprosium. *Physical Review A*, 97(2):023633, 2018.