Getting started with Julia Code for simulating molecular OBE

Thomas Langin

May 14, 2024

1 'Home Computer' Simulation

Note: For the SrF redMOT (the example that this code is currently set up to run), this code will simulate $a(z, v_z)$ with relatively low resolution (e.g., large-ish spacing between discrete (z, v_z) points) and somewhat large uncertainty (only 2 trials (see variable 'numTrialsPerValueSet' defined in line 15 of 'SrFTestCode.jl') in the random cube defined by $(x = 0, y = 0, z = z_{trial})$ and $(x = \lambda, y = \lambda, z = z_{trial} + \lambda)$). This is mostly fine, especially at the high velocities relevant to the redMOT, but for lower velocities (like for a blueMOT or sub-Doppler cooling), we'd probably want a little more (**someone should quantify convergence at some point, if they are interested**)

The simulation takes me **6 minutes and 30 seconds** on my work desktop (Intel i3 core, so not the most amazing CPU). It does run trials in parallel, so you will need to configure julia to have access to multiple cores (see here).

1.1 Files

1.1.1 TestCode.jl

This will be the code that you run. Right now I have it set to run in two 'chunks' in VSCode. First chunk is up to line 361 (the ## splits the two chunks) and, when run, defines a bunch of vaiables relevant to the simulation but does not actually run the simulation. Running the subsequent chunk will run the simulation. Feel free to get rid of the ## if you want and it will run all at once. Code is commented reasonably well in my opinion but if you have any questions feel free to ask.

The basic gist is that first we load the auxilliary programs (which I'll describe in a bit), then a bunch of user choices regarding saving files and simulation parameters, including the MOT displacements (z_{trial}) and velocities (v) over which you want to simulate. I've set this code to run such that z is the 'slowing axis', from which the molecules enter the chamber (see Figure 4a of here) and molecules also move on this axis (basically, we are trying to get the '1D' trapping acceleration field, ideally of a form $a(z, v_z) = -\beta v_z - \alpha z$ such that there are both velocity and spatial damping forces, but, crucially, the fields and lasers are still **three dimensional**. In other words, this is a 1D 'force model' but not a 1D simulation, the 3D nature of the field and the lasers is important especially when simulating molecules, which require cycling out of dark states & also accurate modeling of dark state forces).

Then we set the laser parameters. These are currently set to the parameters ('5D' in the code) we use for the SrF red-MOT described here. I have a bunch of other possible laser settings commented out.

Then finally the simulation is ran. For a given (z_{trial}, v_z), the evolution of the density matrix for the random choices in the 'random cube' described earlier is calculated. As described in here, the density matrix does not reach a steady state (since we have a time-dependent Hamiltonian). The periodicity of the Hamiltonian is set by the value to which we round off all frequencies (for $v > 0.5\Gamma/k$, this is the nearest 0.05 (e.g., periodicity is $2\pi/.05 \sim 126$ time units), for lower speeds we round to lower values, see 'vRound' variable). The force is then calculated and averaged over this period, and also averaged over all the random choices, for a given (z_{trial}, v_z). The calculated values of $\vec{a} \cdot \vec{v_z}$ and $\vec{a} \cdot \vec{z}$, and the population in each quantum state $|F, J\rangle$ for a set of v_z are saved in a file with a name 'forceVsSpeedDisplacment**xx**MMSameDir.dat' where **xx** is z in millimeters.

So, at the end, you will have a folder with a number of files equal to the number of z_{trial} values (here, I set $z_{trial} = \{0.5, 1.5, 3.0, 4.5, 6.0, 7.5\}$ mm). The number of entries in each file is set to the number of velocities you chose to iterate over. We also save the laserVariables for this run in that same folder. This folder will be saved in a master folder called **saveData**.

1.1.2 auxFunctions.jl

Holds all of the functions relevant to actually performing the simulation, ranging from setting randomized values in the 'λ'-cube, to defining the 'laser structure', the clebsch-gordan like coupling matrices between the XΣ and the AΠ and BΣ states ('createCouplingTermszndLaserMasks' + makeCouplingMatrices!), the magnetic field terms ('makeBCouplingMatrices!' + 'makeBFieldTerms!), the laser field terms (makeFieldTerms!), and finally evolving the density matrix (densityMatrixChangeTerms!) and subsequently calculating the force experienced for a molecule at time t given the previously derived density matrix $\rho(t)$ (makeForceVsTime!). For more on what is actually going on here, consult **section 1 of writeUpV3.pdf**, as well as the comments in the code.

1.1.3 SrFVariables.jl

Contains variables relevant to SrF. I have written the code such that everything in **TestCode.jl** is molecule agnostic; all molecule information is stored in files like SrFVariables.jl, or CaFVariables.jl. These include things like the wavelength, vibrational branching ratios, natural linewidth, mass, J-mixing coefficients (a and b), magnetic moments of each hyperfine state, hyperfine energies, etc. If you want to simulate something else instead of SrF (e.g. CaF), all you need to do is change line 3 of TestCode.jl to read, e.g., include(CaFVariables.jl). You will almost certaintly also want to change your 'laserEnergy' variable whenever you do this, such that the lasers have the correct energies for whatever hyperfine structure your molecule has.

1.1.4 analysis.m

Contains code for analysis of the output folder from Julia. You could rewrite all of this in Julia but I don't like the way plotting works in VSCode using Julia, so I just do my analysis of the simulation ex-post-facto in matlab. Finer plots of $a(z, v)$ are generated from the discrete data for (z_{trials},v_z) via 2D spline interpolation ('interp2').

These splines are also used to simulate particle trajectories using $\dot{z} = v_z$, $\dot{v}_z = a(z, v)$ for $z(t = 0) = -\text{Max}(z_{trial})$. For a given initial velocity $v_z(t = 0)$, we determine whether the particle is trapped (if, in a reasonable time of 20 ms, the particle has not reached a value $z > \text{Max}(z_{trial})$). The initial $v_z(t = 0)$ is incremented until it becomes large enough such that the particle is not trapped. This defines the capture velocity (e.g., any atom/molecule with velocity below this untrappable velocity is assumed to be trappable).

We also plot $a(z) = \int_{-2vT}^{+2vT} a(z, v)dv$ and $a(v_z) = \int_{-2\sigma}^{2\sigma} a(z, v)dz$, where vT is what we assume for a thermal velocity (here, 1.5 m/s, for 25 mK) and σ is what we assume for a gaussian cloud radius (here, 3.5 mm). This gives more intuition regarding the spatial confinement and velocity damping forces (e.g., the sub-Doppler heating forces become obvious).

1.1.5 def_molecule.m

Has set of values for various molecules. Used in analysis.m.

2 Cluster Simulation

Oftentimes, especially for a blueMOT, you will want to run at very low velocities (e.g., for SrF, 20μK corresponds to $v_T = 0.04$ m/s which corresponds to $v_T/(\Gamma/k) \sim 0.01$ in simulation units). This sets the resolution of the simulation: for a given velocity for which you are simulating, you want to round all your velocity components to something substantially less (ideally $v/5$ or even lower, e.g., in simulation units $v_{round} \leq 0.002$ if

$v = 0.01$). The rounding value sets the periodicity of the hamiltonian (see here), and thus the timescale over which you need to run the simulation (here, τ =1/.002=500Γt, plus whatever you decide is required for quasi-steady state. I usually assume steady state is reached in roughly 2τ but I've never done a rigorous convergence check). For a higher velocity you can get away with higher values of v_{round} up to 0.1 (no point going any higher than this since all other frequencies, etc. rounded to nearest 0.1).

Additionally, you will get more accurate results if you run more trials for a given (z, v), or to run with higher resolution. For this reason, it is sometimes helpful to use the cluster. To take advantage of UChicago cluster core parallelization (all cpus have 20 cores or more), I usually set 'numTrialsPerValueSet' in 'testCodeCluster.jl' to be a multiple of 10 (since there are 2 actual runs per given trial').

2.1 Files

2.1.1 TestCodeCluster.jl

This code will run on the cluster. It takes in some inputs (ARGS[]) from the .sbatch file (see later) regarding what the molecule is, whether repump lasers should be included, some other variables (depending on what you are scanning over, you will want to rewrite this code. Right now its set to run some trials related to the SrF blue MOT.). New folders are generally created for every new simulation corresponding to a set of ARGS (see 'saveDataFolderTag' and 'folderString' variables) in which the '.dat' files are saved (similar to the 'at home' code).

2.1.2 auxFunctionsForCluster.jl

Similar to 'auxFunctions.jl' (it's actually probably exactly the same but I don't feel like verifying it.)

2.1.3 testBatch.sbatch

Code for submitting jobs to the UChicago cluster. The SBATCH variables will obviously need to be rewritten for submitting to whatever cluster you are on. This is set to first load julia (NOTE: you will also need to install julia packages like 'DifferentialEquations' and 'LinearAlgebra', see auxFunctions.jl. The way this is done is probably cluster dependent. For UChicago, I believe it was straightforward to open a julia terminal in commandline and just 'add Packages' in the conventional way. Somehow they all wound up in the right place but YMMV). Then it sets up to scan over the variables in ARGS (see TestCodeCluster.jl).

To submit, you should have this file, and the relevant .jl files (testCodeCluster, auxFunctionsForCluster, the relevant SrFVariable.jl type files, etc.) all in the same folder on the cluster. Then just type in

```
sbatch testBatch.sbatch
```

and it should work.