Quantum Trajectories simulation of Λ-cooling

Thomas Langin

July 29, 2022

In this writeup, I will include a quick summary of how to use the Quantum Trajectories Λ-cooling simulation code.

Step 0: This is written in C++ and requires Armadillo to be installed on your system. For cluster simulations (at least at UChicago), since the cluster already has armadillo, you just have to add 'module load armadillo/7.8' (insert whatever version is on the cluster in place of 7.8, if you like) to your batch submission file. For home computations, follow the instructions here for installing armadillo (personally, I like using the linux subsystem for windows, but feel free to use whatever you like. These instructions though are for an ubuntu system)

Step 1: Once armadillo is installed (or, on cluster, the armadillo module is loaded), compiling the program is simple, just type (or include in your shell script): g++ -std=c++11 lambdaCoolingFinalAddCaOH.cpp -o testVaryCaFSrFNoPathLengthCorrectPol -larmadillo -O3 -fopenmp

- -std=c++11: I use a few things from the c++ standard library (mostly random number generators, and some statistics packages), so this needs to be included
- -o: Let's you pick the name of the executable file (this will be the string after "-o", in our case "testVaryCaFSrFNoPathLengthCorrectPol", but this can be whatever you want it to be)
- -larmadillo: Let's the compiler know that the armadillo library is used in the code
- -O3: I haven't done much testing on how important this is. Basically though, this optimizes some math processes for speed.
- -fopenmp: The code uses some parallelization, and -fopenmp is needed to enable that.

Step2: After compiling the code into an executable file, you can run it. The code requires the following inputs, in order:

- Δ (in units of Γ), the overall detuning
- δ_{Raman} (also in units of Γ), the raman detuning (defined such that positive δ_{Raman} increases the energy difference between the laser components)
- satParam: The saturation parameter I/I_{max} of one laser pass (NOTE: code does not consider finite beam width, but rather assumes molecules all see central intensity), including both hyperfine components. In my code, saturation intensity is the 'bare' saturation intensity (as in, without any angular components/Clebsch-Gordan like terms), $I_{sat} = \frac{\hbar c \Gamma \omega^3}{4\omega \times 3c^2}$. For SrF X$\rightarrow$A, this is $2.96\,\mathrm{mW/cm^2}$. For CaF X$\rightarrow$A this is $4.85\,\mathrm{mW/cm^2}$. For CaOH X$\rightarrow$A this is $3.89\,\mathrm{mW/cm^2}$.
- R_{21}: Ratio of power addressing 'second' level to the power addressing 'first' level.
- firstState: The 'first' level addressed by hyperfine lasers. This code assumes the normal $|X\Sigma, N = 1\rangle$ hyperfine structure with 4 hyperfine components.
  - firstState= 0: first laser addresses $|F = 1, J = 1/2\rangle$ (this is the case for all CaF and SrF experimental data. But, in case a user wants to experiment with addressing a pair of hyperfine levels that doesn't include this state, I decided not to hard-code it.)
  - firstState= 1: first laser addresses $|F = 0, J = 1/2\rangle$
  - firstState= 2: first laser addresses $|F = 1, J = 3/2\rangle$
  - firstState= 3: first laser addresses $|F = 2, J = 3/2\rangle$

  Really what this defines is the frequency of the 'first' laser. So, in other words, if $\Delta = 2.8$ and firstState=0, there will be a laser component that is 2.8Γ blue of $|F = 1, J = 1/2\rangle$. This **does not** turn off coupling of this laser frequency to other states (e.g., population in $|F = 0, J = 1/2\rangle$ will see this laser at $2.8+(E_{F=0,J=1/2} - E_{F=1,J=1/2})$ (E in units of Γ as well)). For SrF, for example, this would be $2.8\Gamma+7.5\Gamma$. For CaOH, this would be $2.8\Gamma+.001\Gamma$, since the hyperfine interaction is very small).
- secondState: The 'second' level addressed by hyperfine lasers (number order is the same as firstState). For example, if secondState=2, then the second laser will be set at a frequency $\Delta - \delta_{Raman}$ to the blue of $|F = 1, J = 3/2\rangle$.
- CaOHOrCaFOrSrF: Sets what molecule is being addressed (the energy levels of the hyperfine states, and the j mixing term a (relevant for calculating coupling to excited states) for each molecule are stored in the code, this just tells the program which ones to use).
  - CaOHOrCaFOrSrF=0: Code uses SrF variables
  - CaOHOrCaFOrSrF=1: Code uses CaF variables

– CaOHOrCaFOrSrF=2: Code uses CaOH variables

Concrete example: To execute code that will simulate CaF Λ cooling for the conditions in the bottom left corner of Fig. 2a ($\delta_R = -500\,\text{kHz}$ (-0.06Γ), $I_0 = 6.8\,\text{mW/cm}^2$ ($s_0 = 1.4$), $R_{21} = 0.92$, $\Delta = 2.9\Gamma$) of the CaF Λ cooling paper, you would enter:

./testVaryCaFSrFNoPathLengthCorrectPol 2.9 -.06 1.4 0.92 0 3 1

1 Brief description of included files

1.1 lambdaCoolingFinalAddCaOH.cpp

This is the actual simulation file. I'll admit that this isn't the most clear code in the world (though I have added comments to help fix this). A brief summary of some important features:

1.1.1 General Goal

This code propagates the 'quantum trajectories' of a user determined (via variable 'const int N0', which I've set to 20 here, as this happens to be the number of cluster cpus I use for simulation. The results will obviously get less noisy if you simulate more particles. Here, I assume that each particle somewhat 'ergodically' samples the distribution of available states, such that simulating for a long enough time-frame gives you decent statistics even with a small number of particles) number of particles.

Each 'particle' has three things associated with it: its wavefunction (cx_mat wvFns[N0]), its velocity (V[3][N0]), and its position (X[3][N0]). The code steps the time by the value TIMESTEP (here it's 0.01 (units Γ^{-1}), see Fig 1.2 of my 'QTOBESimulationWriteup'). During this time, the wavefunction evolves according to the 'atom-light+dissipation' hamiltonian unless it is randomly determined (with likelihood TIMESTEP*P_{exc}, where P_{exc} is $\sum_i \langle e_i|\psi\rangle\langle\psi|e_i\rangle$ and the i means you are summing over the excited states) that a quantum 'jump' has occured. Then, various random numbers are 'rolled' to determine where the particle 'jumped' to. If there's no jump, the velocity changes as well according to the force $\langle F\rangle = -\langle[\nabla, H]\rangle$. If there's a jump, two photons kicks (absorption+emission) in random directions are added to the velocity. The position is then evolved by V*TIMESTEP. I'm skipping a lot of details here, for more see my writeup 'QTOBESimulationWriteup', and also Mikhail Lukin's textbook, Chapter 6

I initialize the particles in a random super position of the $|X\Sigma, N = 1\rangle$ hyperfine states (I find initialization doesn't really matter much). This is done in 'init()'. The particles are also given random velocities taken from a normal distribution with standard deviation (vSpread=0.06, which gives an initial temp of 100μK in SrF and 70μK in CaF, close enough to what you would start from after '4-laser molasses'. I start from here to demonstrate that the code simulates cooling, as seen in the decay of temperature T with

time, see Fig. **??**, not just a totally 'random' walk or something.

Every 10Γ (TIMESTEP*sampleFreq), I record the average population in all 16 states and $\langle v^2$ in a file called 'statePopulations.dat' (from this, $T(t)$ can be determined). I also record the velocities of all the particles in files called 'velDistributionVx.dat' (Vy, Vz, etc.)

I find that, for reasonable values of s_0, the temperature stabilizes within $t_{stab} = 300000\Gamma^{-1}$ (6 ms for CaF, 7 ms for SrF). So, I simulate to $t_{max} = 500000\Gamma^{-1}$ and only use the data from $t > t_{stab}$ to determine the equilibrium temperature.

1.1.2 How to read the code

The first $\sim$ 200 lines are global variables (yeah I know you aren't supposed to use global variables in C++...I think of this more as a script though as opposed to an 'object-oriented' C-style program). Everything here can be referred to in subsequent functions. I define a bunch of variables related to the molecules in question ('ground energies' (of each ground-state hyperfine manifold), 'aParam' (for J-Mixing), 'vKick' = $\hbar k^2/(m\Gamma^2$ (the velocity of a photon kick, also shows up in the normalized force term for the 'atom-light' force)). I also have a couple of user adjustable parameters (tmax and 'retroPathLength'. See Sec. 1.1.3 for what 'retroPathLength' is. Normally set this to 0, unless you want to mess around with this feature). I also pre-initialize a bunch of values that are filled in later on depending on, for example, the random initialization (wvFns) or else the user's inputs (detuning, satParam, etc.).

Main(): Upon execution, after the global variables are read, the first thing that happens when the code is executed is 'main()' (this is how C++ programs work). This at the very end of the code. Here, the users arguments (the stuff after ./executableFileName) are read in. Based on this, groundEnergies, detunings, etc. are all chosen, and values for rabi-Frequency of each of the two frequency components, laser energies, etc. are all calculated. A folder to store the data
'Det__NumIons__SatParam__DetRaman__R21__SecondState__CaFOrSrFOrCaOH__',

where the underscores are filled in with the actual user-chosen values (**NOTE: I multiply DetRaman by 1000 in the folder name, as this is usually a very small value**), is created.

This is also where the atom-light hamiltonians are determined. The σ^+, σ^- and π coupling matrices are established (the cs's...note that these only have the 'top-right' term...I add the hermitian conjugate during the 'qstep()' function) as are the coupling coefficients ('Clebsch-Gordan' like terms, see section 1.1 and appendix A of 'mainOBE-Writeup.pdf'), the 'gs's'.

Finally, in 'main()', the atom-light hamiltonian is comprised of an 'energy' term (energy

of each state on diagonals, 'HamEnergyTerm') and a atom-light coupling term (Rabi-frequency × gs × cs, see 'hamCouplignTermOnlySigPlus' (SigMinus, Pi, etc.), and a dissipation term 'hamDecayTerm'. For the origin of the latter, see Lukin's textbook). Finally, the **init()** function is called to initialize velocities and randomize initial wavefunctions.

The 'pre-initialization' is now over, the final thing in **main()** is a loop that runs until $t = t_{max}$. In this loop, during each step, the quantum trajectory (qStep), velocities and positions (step()) are updated. Also, every 1000 timesteps, files containing the populations in each state and the velocity distributions are updated (outputData). Next, I will summarize each of these functions called within the loop.

qStep(): Handles the evolution of the quantum trajectories in parallel (see the stuff with #pragma omp parallel, etc.) Each time this is called, a weighted dice is rolled in order to determine whether the particle jumps or not. If it doesn't, the wavefunction is evolved according to the quantum trajectories approach (again, see Lukin's book for more details) and the force is calculated ('forceMatX, etc.') and the velocities are changed ('kickX', etc.). I implemented my own 4th order Runge-Kutta stepper for this evolution, it seems to work fine.

I'll note here that this is where the lasers are incorporated (see 'hamCouplingTerm'). The system is moved into a frame rotating with the frequency difference between $|X\Sigma, F = 1, J = 1/2\rangle$ and $|A\Pi\rangle$ (assume that there's no hyperfine splitting in the $|A\Pi\rangle$ state). This creates the energy terms on the diagonal of 'hamEnergyTerm' (the overall 'detuning' energy is 'given' to the excited state, see definition of hamEnergyTerm in **main()**). The laser addressing the other state (typically either $|F = 1, J = 3/2\rangle$ or $|F = 2, J = 3/2\rangle$) has an additional factor $\exp[i \times \text{detLaserDiff} \times t]$ (see definition of 'hamCouplingTerm' in the **qstep()** function). DetLaserDiff is just the energy difference between the two coupled hyperfine states **plus** the raman detuning (this is defined in **main()**). I'll also note here that it is easy to add other lasers in a similar way, for example, if you wanted to simulate '4-laser grey molasses'.

If a jump occurs, a photon is considered to be 'absorbed' and another to be 'emitted', providing two random kicks. The molecule also jumps to the ground state. I've seen this procedure be implemented in a number of ways. Here, I just roll for whether emission is σ^+ σ^- or π (weighted by population in each 'Zeeman' excited state). Then, based on that, I roll for which hyperfine manifold it decays to (reasoning: In principle, the energy of the photon could be 'measured' and thus whether the decay is to $|F = 1, J = 1/2\rangle$ or, say, $|F = 1, J = 3/2\rangle$ could be determined. I **dont** choose which 'Zeeman' state within the manifold the particle decays to, since these are degenerate in absence of a magnetic field, instead, I weight the population going to each Zeeman state within the manifold based on the decay probabilities and excited state populations. I haven't really checked if the 'jump' method matters much).

Finally, to avoid small errors in the wavefunction propagation, I renormalize the wavefunction on every step, such $\sum_i \langle i|\psi\rangle\langle\psi i\rangle = 1$ where the sum is over all eigenstates $|i\rangle$

step(): This is very basic, this just steps the position according to the velocity. Stepping the position matters because the force comes from the motion of the particle though the polarization and intensity gradient created by the light field, as is the case for all gray-molasses cooling. These changes are very small, and so I find a basic 'euler' approach to the position evolution is fine.

outputData(): Here, we write to the files I described previously (statePopulations.dat and the velocityDistribution data files).

1.1.3 Investigating whether path length matters

I originally ignored this when writing the code. But, upon reading a paper by the Ospelkaus group, in which they try to engineer sub-doppler cooling forces in a MOT by adding a high intensity blue detuned laser, I saw that they found that the path length between passes of the retro-reflected beam mattered quite a bit.

This is because the laser frequency components addressing the two hyperfine components will accrue a phase difference $\partial\phi = \frac{\partial\omega}{c}L$ where L is the path length between the incident and retro-reflected beam and $\partial\omega$ is the frequency difference. Consider coupling the $|F = 1, J = 1/2\rangle$ and $|F = 1, J = 3/2\rangle$ states in SrF. For $L \sim 0.3\,\mathrm{m}$ (a reasonable value), this gives a phase accrual of $\partial\phi = 0.8$, a non-trivial fraction of 2π.

What does this mean in the code? It means that the polarization interference pattern of one laser will be shifted by $\partial\phi$ with respect to the other. This, in the code, is handled by the terms 'randPhaseX' (Y, Z, etc.) = retroPathLength×gamOverC×detLaserDiff (here I assume that retro path length is the same for all three sets of beams, but this doesn't have to be the case; you would just need to define three different retroPathLengthX (Y,Z, etc.)).

Does it matter? I see some preliminary evidence that it does, in the simulations. It would be nice to see it experimentally before I go into too much more detail on simulating it. It could be that other effects in the experiment (intensity imbalance, finite beam waist) dwarf this one. **For all results in this writeup, I set the value of $\delta\phi$ to zero**

1.2 testBatch.sbatch

This code is what I use to run the program on the cluster. It requests a day to run on 20 cpus. It loads armadillo, and compiles the file (g++ line). I currently have the script set up to then iterate through a few values of 'satParam' and δ_{Raman} for Λ-coupling between the $|F = 1, J = 1/2\rangle$ and $|F = 1, J = 3/2\rangle$ states of SrF, with R_{21} of 0.9. I

have a few commented out sections that show iterations over parameters for CaF as well.

At least at UChicago, submitting the job is as simple as typing 'sbatch testBatch.sbatch'

1.3 testBash.sh

Similar to testBatch, except this is what you would use to run the code on your own computer. To run, just type ./testBash.sh into your 'linux for windows' (or whatever you are using) terminal. I will warn you; this will take a while on your own computer, and, since I've implemented parallelization, it will eat up your cpu power while it is running.

1.4 CaFDoyleReplicationAttempt

This folder contains data folders from an attempt to replicate the second row of the CaF Λ cooling paper, along with an analysis script (in matlab, 'analysis.m'). The script assumes that the temperature converges after 2.4 ms (black lines in $T(t)$ curves), and plots the temperature averaged over that time frame as a function of δ_R.

I decided to run this simulation at a fairly high initial temperature of 400μK, in order to demonstrate the robustness of Λ cooling. Unfortunately, this meant that, for a few of the simulations, there were somewhere between 1-3 (out of 20) particles that started with a high enough initial velocity that they were not cooled effectively, which throw off the data. I added a section that gets rid of those particles. In general, this is not necessary if you start your system with a more reasonable temperature (say, 100μK, as I do in the next section).

Results of this simulation can be seen in Fig. 1. As you can see, it's not a 1-to-1 match, but some of the features line up.

If you investigate closely, you'll find that the T_z is consistently slightly higher than T_x and T_y. This was due to a bug in my code that systematically resulted in the spontaneous 'kicks' from absorption+emission being delivered preferentially along the z axis. I've since fixed this bug, and have seen that, in subsequent simulations all temperatures align. It didn't really change the overall result though and so I didn't bother re-running this whole simulation.

Another interesting feature is that the velocity distributions are not gaussian at all. In Fig. 2 I consider the $f(v_x)$ (other axes look the same) after the temperature stabilizes for $\delta_R = -498$ kHz, $\delta_R = 0$, $\delta_R = +166$ kHz. At first glance, it looks like the $\delta_R = 0$ case should be hotter than the -498 kHz case, since it's less sharply peaked. But, it turns out that the tails on the -498 kHz are very broad, as can easily be seen in the log plots (right hand side of Fig. 2), which ultimately leads to the higher temperature. Theoretically, we should be able to see this in the time-of-flight data if we had some better signal-to-noise,

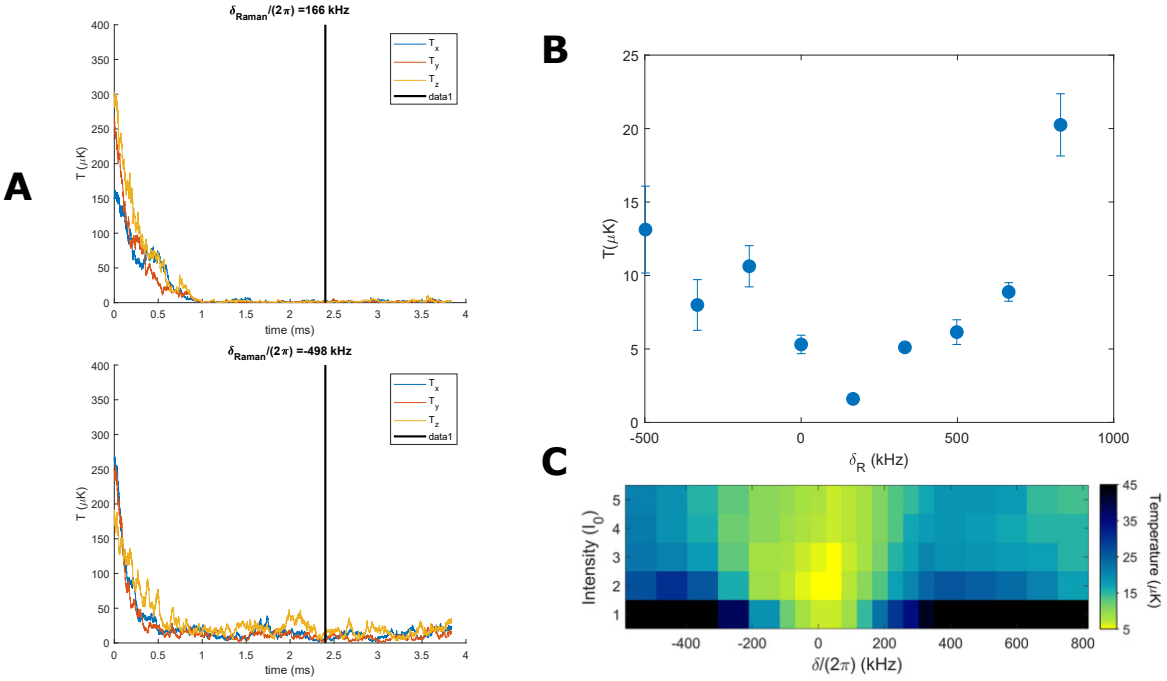

Figure 1: **A** $T(t)$ curve. Black line indicates time t_{equil} at which I consider the temperature equilibrated. Only data from $t > t_{equil}$ are used for calculating T in **B**. **B** T vs δ_R. **C**: Doyle group experimental results. Simulations in **B** are done under the conditions matching the second row ($I = 2I_0$). The results aren't an identical match (no experimental data is quite as low as the $\sim 2\mu$K predicted by the simulations, but the features match.)

if indeed this is real. It would be interesting. I think this is just a natural consequence when you have cooling and heating terms that are non-linear in v (e.g., the heating terms turn off for VSCPT at $v = 0$, but are otherwise present).

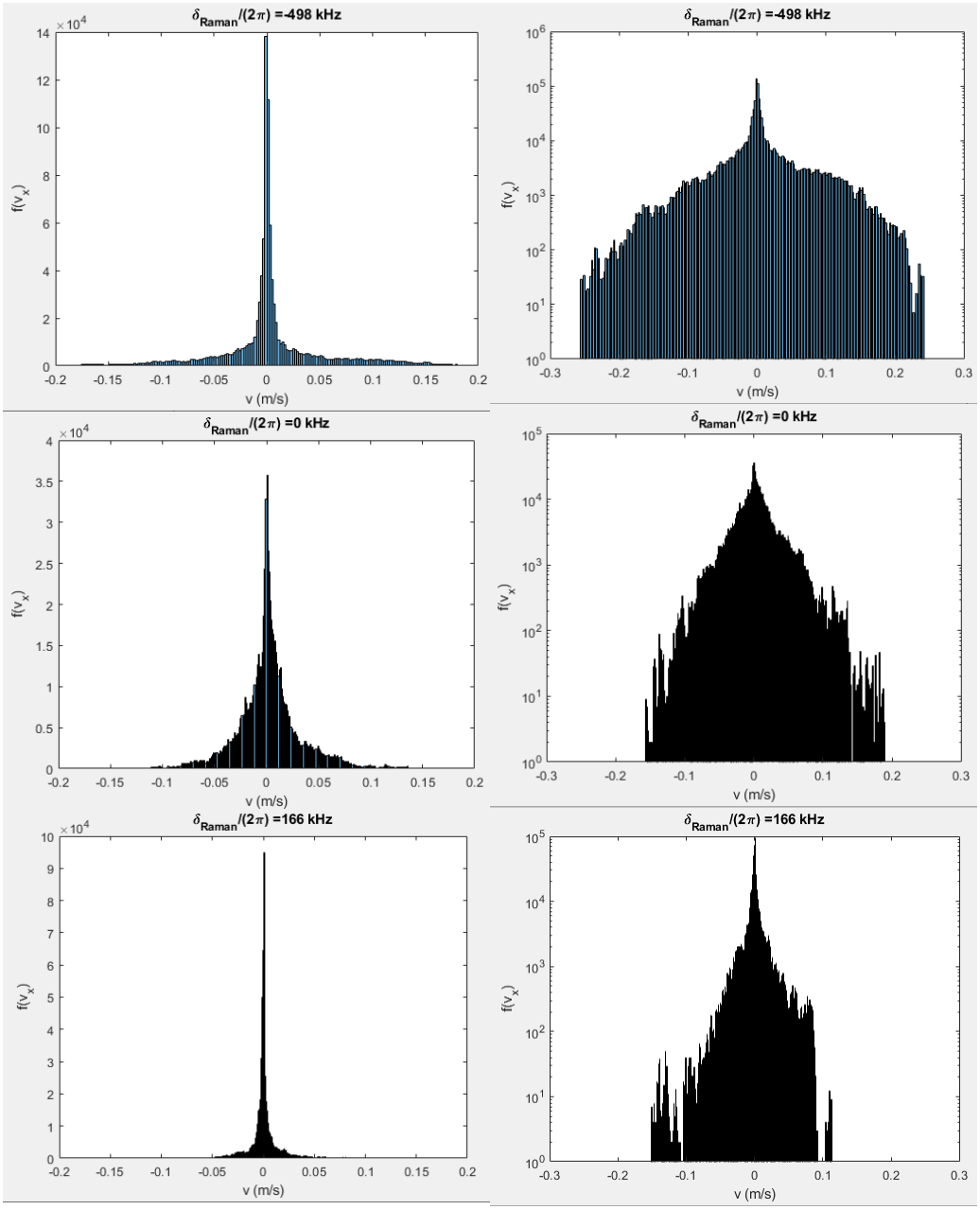

Figure 2: Left: $f(v_x)$ on linear scale. At a quick glance, it looks like $\delta_R = 0$ kHz should be 'hotter' than $\delta_R = -498$ kHz, as it is less sharply peaked. This is not what we see though; see Fig. 1B. Right: $f(v_x)$ on log scale. Here, we see that the tails for -498 kHz are much broader (even though the system has stabilized, see bottom of Fig. 1A). This is what ultimately leads to the higher temperature $T = m\langle v^2 \rangle / k_B$ measurement for -498 kHz. Presumably, with enough signal-to-noise, the effect of these non-gaussian velocity distributions (if they are real and not some weird simulation bug) should be observable in experiment in the time-of-flight data.

1.5 CaFSrFCompare

Folder contains a bunch of data folders from a comparison between CaF and SrF Λ cooling, for the specific sase $R_{21} = 0.9$. The transitions $|F = 1, J = 1/2\rangle \rightarrow |F = 2, J = 3/2\rangle$ and $|F = 1, J = 1/2\rangle \rightarrow |F = 1, J = 3/2\rangle$ are considered. I considered satParam= $\{1.5, 3, 8\}$ and $\delta_R/\Gamma = \{-0.1, -0.02, 0, 0.02, 0.1\}$.

The results are summarized in Fig. 3. I find that, while both transitions work well in CaF, in SrF, only $|F = 1, J = 1/2\rangle \rightarrow |F = 1, J = 3/2\rangle$ actually results in cooling; for all configurations of δ_R/Γ, it turns out that $|F = 1, J = 1/2\rangle \rightarrow |F = 2, J = 3/2\rangle$ results in **heating** in SrF. **This is a good illustration for why these sorts of codes are helpful**. What actually happened in real life is that, in SrF, we tried $|F = 1, J = 1/2\rangle \rightarrow |F = 2, J = 3/2\rangle$ for quite some time (since it worked in CaF), and were **completely baffled for upwards of 3 months** about why it wasn't working, which motivated me to write this code. Though I can't say the simulations have informed me about exactly what makes it heat for SrF, at least they reflect that reality, and so if we had them at the time, it could've saved a lot of heartache.

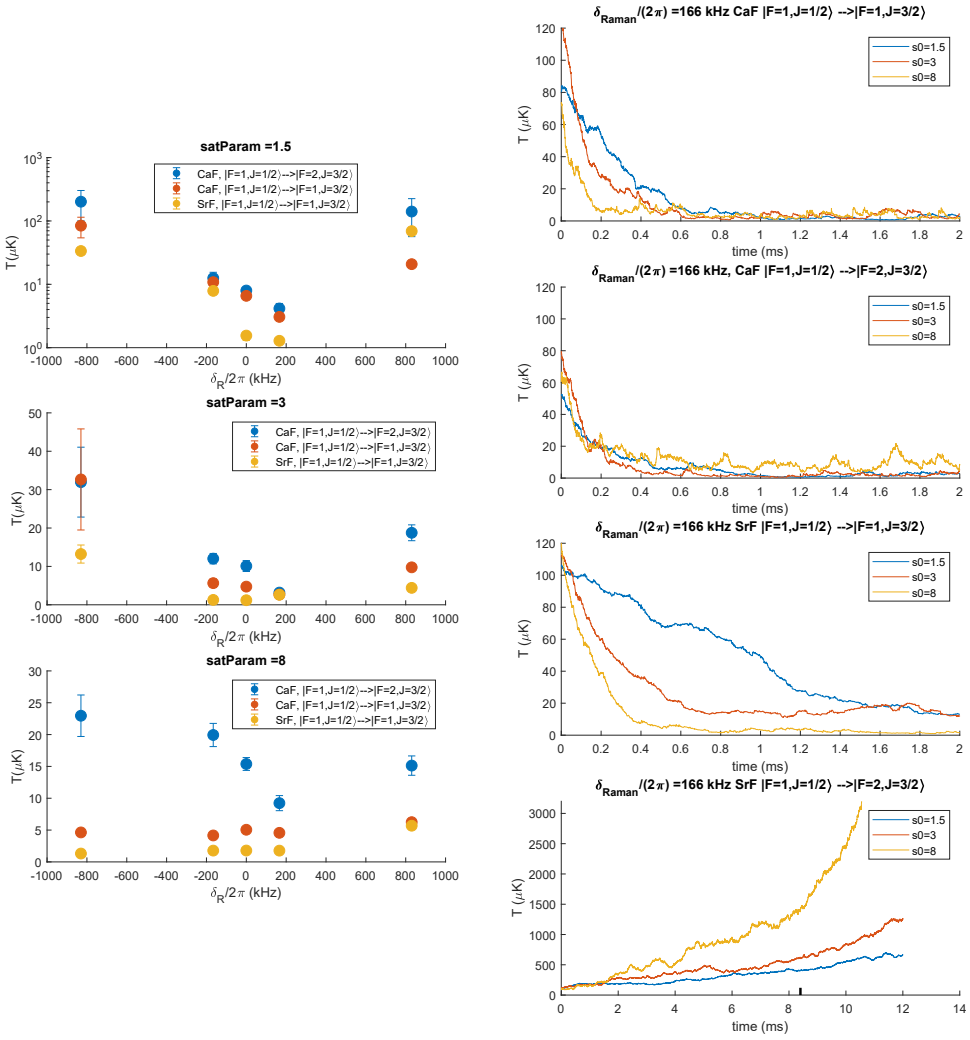

Figure 3: Left side: $T(\delta_R)$ for CaF and SrF for all successful Λ cooling approaches (e.g., not including $|F=1,J=1/2\rangle \rightarrow |F=2,J=3/2\rangle$ for SrF). Note that, for the lowest satParam, I plotted T on a log scale so that all of the data would be visible (high $|\delta_R|$ resulted in high T here). In general, there's not a huge difference in the two couplings that work for CaF, except for at high power, where $|F=1,J=1/2\rangle \rightarrow |F=1,J=3/2\rangle$ is clearly better (**maybe using this transition would also make CaF ODT cooling better? Since it's more robust wrt δ_R, and thus maybe more robust with respect to vector/tensor shifts?**). Right: $T(t)$ curves for each molecule+coupling-scheme for $\delta_R = 166\,\text{kHz}$. Clearly the $|F=2\rangle$ case heats in SrF. Also, it seems that the SrF cooling timescale strongly depends on power (we see this in experiment). For CaF, the timescale dependence is weak in $|F=1,J=3/2\rangle$ and non-existent in $|F=2,J=3/2\rangle$

1.6 QTOBESimulationWriteup.pdf

A longer writeup of mine from a while ago as I developed this code, explaining it in some greater detail. This has some more evidence that the code replicates results from basic laser cooling, along with some more exotic things like lin⊥lin gray molasses, as seen in Mike Tarbutt's paper here.

1.7 mainOBEWriteup.pdf

Very long writeup of another simulation code for simulation of optical bloch equations. Only the appendix really matters here, which is where I write out the coupling matrices (with J-Mixing terms included).