pylcp

Release 1.0.0

Steve Eckel, Daniel Barker, Eric Norrgard, and others

Jun 24, 2022

CONTENTS

1 Installation instructions **2**
- 1.1 Prerequisites . . . 2
- 1.2 Recommended installation: via Python pip . . . 2
- 1.3 Manual installation . . . 2

2 Getting Started **3**
- 2.1 Creating the Hamiltonian . . . 3
- 2.2 Laser beams . . . 4
- 2.3 Magnetic field . . . 4
- 2.4 Governing equation . . . 4
- 2.5 Next steps . . . 5

3 Examples **6**
- 3.1 Basic Examples . . . 6
  - 3.1.1 Power broadening . . . 6
  - 3.1.2 Rabi Flopping . . . 10
  - 3.1.3 Damped Rabi Flopping . . . 12
  - 3.1.4 Adiabatic Rapid Passage . . . 15
  - 3.1.5 Optical Pumping . . . 21
  - 3.1.6 Three-level susceptibility & EIT . . . 28
  - 3.1.7 STIRAP . . . 32
- 3.2 Optical Molasses . . . 35
  - 3.2.1 Two-level molasses in 1D . . . 35
  - 3.2.2 $F = 0 \to F' = 1$ 1D molasses . . . 46
  - 3.2.3 $F = 2 \to F' = 3$ 1D molasses . . . 53
  - 3.2.4 $F \to F'$ 1D molasses . . . 67
- 3.3 Magneto-optical traps . . . 71
  - 3.3.1 $F = 0 \to F' = 1$ MOT forces . . . 72
  - 3.3.2 $F = 0 \to F' = 1$ MOT capture . . . 77
  - 3.3.3 $F = 0 \to F' = 1$ MOT forces with the OBEs . . . 85
  - 3.3.4 $F = 0 \to F = 1$ MOT temperature with the OBE . . . 93
  - 3.3.5 An $F \to F + 1$ MOT with small ground state g-factor . . . 105
  - 3.3.6 General $F \to F'$ 3D MOTs . . . 112
  - 3.3.7 Real atoms in a MOT . . . 115
  - 3.3.8 Two color MOT . . . 124
  - 3.3.9 Recoil-limited MOT . . . 126
  - 3.3.10 CaF MOT . . . 135
- 3.4 Other examples . . . 140
  - 3.4.1 The bichromatic force . . . 140

4 Detailed Reference **144**
4.1 Hamiltonian Functions . . . 144
4.1.1 Other Hamiltonians . . . 144
4.1.2 Overview . . . 147
4.1.3 Detailed functions . . . 147
4.2 Hamiltonian Class . . . 150
4.2.1 Overview . . . 150
4.2.2 Detailed functions . . . 150
4.3 Magnetic Fields . . . 153
4.3.1 Overview . . . 153
4.3.2 Details . . . 153
4.4 Laser Fields . . . 157
4.4.1 Overview . . . 157
4.4.2 Details . . . 157
4.5 Governing Equations . . . 167
4.5.1 Overview . . . 167
4.5.2 Details . . . 167
4.6 Atom Class . . . 184
4.6.1 Overview . . . 184
4.6.2 Detailed functions . . . 184

5 Support **188**

6 Credits **189**

7 Indices and tables **190**

Python Module Index **191**

Index **192**

pylcp is a python package meant to help with the calculation of a variety of interesting quantities in laser cooling physics. At its heart, it allows for automatic generation of the optical Bloch equations or some approximation thereof given a atom or molecule internal Hamiltonian, a set of laser beams, and a possible magnetic field. If you find *pylcp* useful in your research, please cite our paper describing the package: https://doi.org/10.1016/j.cpc.2021.108166

CHAPTER

ONE

INSTALLATION INSTRUCTIONS

1.1 Prerequisites

Install python packages and packages for scientific computing in python. Specifically, *pylcp* uses *numpy*, *scipy*, *numba*. We recommend installing python and the supporting packages via the Anaconda distribution; *pylcp* has been tested and found to work with Anaconda versions 2020.02+ (python 3.7).

1.2 Recommended installation: via Python pip

Install via pip:

```
pip install pylcp
```

This automatically install *pylcp* into your python installation. Please report issues to the GitHub page if you have any problems.

1.3 Manual installation

One can also manually check out the package from GitHub, navigate to the directory, and use:

```
python setup.py install
```

If one wishes to participate in development, one should use:

```
python setup.py develop
```

which does the standard thing and puts an entry for pylcp in your easy_path.pth in your python installation.

CHAPTER

TWO

GETTING STARTED

pylcp has been designed to be as easy to use as possible and get the user solving complicated problems in as few lines of code as possible.

The basic workflow for *pylcp* is to define the elements of the problem (the laser beams, magnetic field, and Hamiltonian), combine these together in a governing equation, and then calculate something of interest.

2.1 Creating the Hamiltonian

The first step is define the Hamiltonian. The full Hamiltonian is represented as a series of blocks. Each diagonal block represents a single state or a group of states called a âĂIJmanifoldâĂİ. The diagonal blocks contain a field independent part, H_0, and a magnetic field dependent part μ_q. Off diagonal blocks connect different manifold together with electric fields, and thus these correspond to dipole matrix elements between the states d_q.

Note that the Hamiltonian is constructed in the rotating frame, such that each manifoldâĂŹs optical frequency is removed from the H_0 component. As a result, the manifolds are generally separated by optical frequencies. However, that need not be a requirement when constructing a Hamiltonian.

As a first example, let us consider a single ground state (labeled g) and an excited state (labeled e) with some detuning δ:

```
Hg = np.array([[0.]])
He = np.array([[-delta]])
mu_q = np.zeros((3, 1, 1))
d_q = np.zeros((3, 1, 1))
d_q[1, 0, 0] = 1.
```

Here we have defined the magnetic field independent part of the Hamiltonian H_g and H_e, the magnetic field dependent part μ_q (which is identically equal to zero) and the electric field d_q dependent part that drivers transitions between g and e. The μ_q and d_q elements are represented in the spherical basis, so they are actually vectors of matrcies, with the first element being the $q = -1$, the second $q = 0$, and the third $q = +1$. In our example, the two-level system is magnetic field insensitive, and can only be driven with σ^+ light.

There are shape requirements on the matrices and the arrays used to construct the Hamiltonian. Assume you create two manifolds, g and e, with n and m states, respectively. In this case, H_g must have shape $n \times n$ and the ground state μ_q must have shape $3 \times n \times n$. Likewise for the excited states. The d_q matrix must have shape $3 \times n \times m$.

We then combine the whole thing together into the pylcp.hamiltonian class:

```
hamiltonian = pylcp.hamiltonian(Hg, He, mu_q, mu_q, d_q, mass=mass)
```

There are a host of functions for returning individual components of this block Hamiltonian, documented in *Hamiltonian Functions*.

2.2 Laser beams

The next components is to define a collection of laser beams. For example, two create two counterpropagating laser beams

```
laserBeams = pylcp.laserBeams([
        {'kvec':np.array([1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
         'pol_coord':'spherical', 'delta':delta, 's':norm_intensity},
        {'kvec':np.array([-1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
         'pol_coord':'spherical', 'delta':delta, 's':norm_intensity}
        ], beam_type=pylcp.infinitePlaneWaveBeam)
```

Here, we make the laser beam collection by passing a list of dictionaries, each dictionary containing the keyword arguments to make individual *pylcp.infinitePlaneWaveBeam* beams. *kvec* specifies the k-vector of the laser, *pol* specifies its polarization in the coordinate system specified by *pol_coord*, *delta* specifies its frequency in the rotating frame (typically the detuning), and *beta* specifies is saturation parameter. The optioanl *beam_type* argument specifies the subclass of pylcp.laserBeam to use in constructing the individual laser beams. More information can be found in *Laser Fields*.

2.3 Magnetic field

The last component that one specifies the magnetic field. For this example, we will create a quadrupole magnetic field

```
magField = pylcp.quadrupoleMagneticField(alpha)
```

Here, α is the strength of the magnetic field gradient. There are many types of magnetic fields to choose from, documented in *Magnetic Fields*.

2.4 Governing equation

Once all the components are created, we can combine them together into a govening equation. In this case, it is an optical Bloch equation

```
obe = pylcp.obe(laserBeams, magField, hamiltonian)
```

And once you have your governing equation, you simply calculate the thing of interest. For example, if you wanted to calculate the force at locations R and velocities V, you could use the *generate_force_profile* method

```
obe.generate_force_profile(R, V)
```

All methods of the governing equations are documented in *Governing Equations*.

2.5 Next steps

Start looking at the *Examples* for next steps; they contain a host of useful code that can be easily borrowed to start a calculation.

CHAPTER

THREE

EXAMPLES

All of these examples are contained in the *doc/examples/* subdirectory of the *pylcp* package as *Juypter* notebooks. The original source makes an excellent starting point for building your own script for *pylcp*.

3.1 Basic Examples

These examples focus on internal dynamics in an atom, without concerning itself with the motion.

3.1.1 Power broadening

Power broadening and saturation are simple effects that can replicated using the rate equations. This example shows how this works.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
```

Two levels with a single laser

This is the simplest case to consider. We will plot up the scattering rate $R_{sc} = \sum_l R_{ge,l}(N_e - N_g)$, where $R_{ge,l}$ is the pumping rate due to laser l. We can compare to the standard analytical expression for a two-level system

$$R_{sc} = \frac{1}{2}\frac{s}{1 + s + 4\Delta^2/\Gamma^2},$$

which is shown in the plots as the black dashed lines.

We start by generating the Hamiltonian. Here, we use an $F = 0 \to F' = 1$ transition as our model system, but we will consider coupling only two of these four states together using a circularly-polarized laser beam propogating along $\hat{z}$.

[2]:
```
Hg, Bgq = pylcp.hamiltonians.singleF(F=0, muB=0)
He, Beq = pylcp.hamiltonians.singleF(F=1, muB=1)

dijq = pylcp.hamiltonians.dqij_two_bare_hyperfine(0, 1)

ham = pylcp.hamiltonian(Hg, He, Bgq, Beq, dijq)
```

Add in a constant, small magnetic field to establish a quantization axis along the $\hat{x}$ axis:

[3]:
```
magField = lambda R: np.array([1e-5, 0.0, 0.0])
```

We next run through a loop of both detuning and intensity, remaking the single laser every time. Note that it is σ^- polarized relative to *both* its k vector and the quantization axis.

[4]:
```
# Make two independent axes: dets/betas:
dets = np.arange(-5.0, 5.1, 0.1)
intensities = np.logspace(-2, 2, 5)

fig, ax = plt.subplots(nrows=1, ncols=1)
for intensity in intensities:
    Rijl = np.zeros(dets.shape)
    Neq = np.zeros(dets.shape + (4,))
    for ii, det in enumerate(dets):
        laserBeams = pylcp.laserBeams(
            [{'kvec':np.array([1., 0, 0.]),
              's':intensity, 'pol':-1, 'delta':det}]
        )

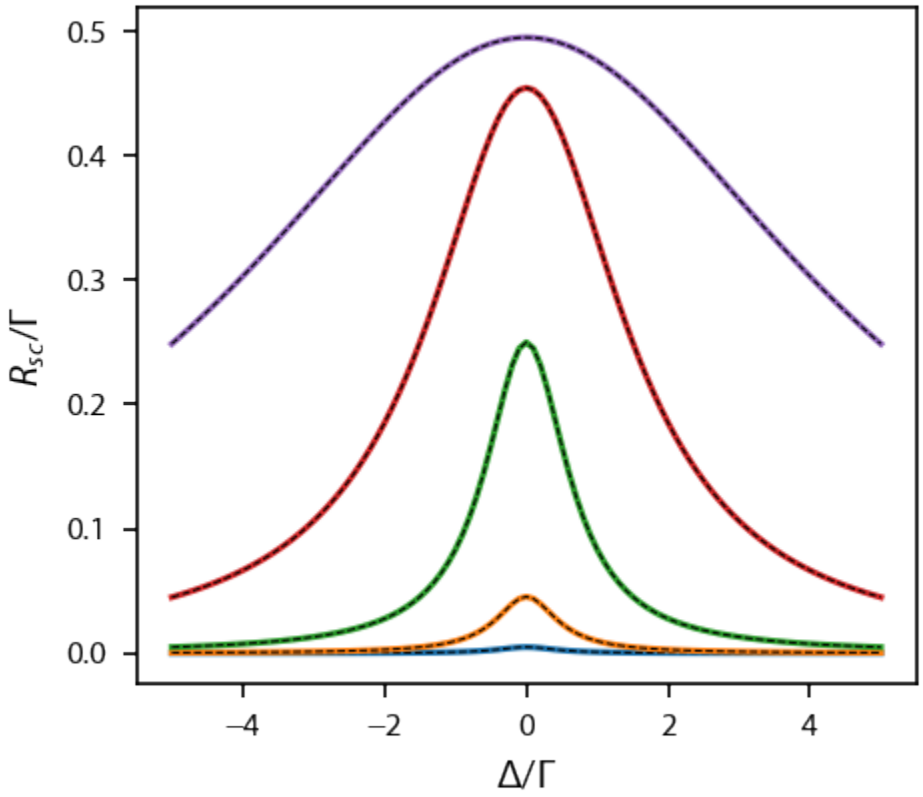
        rateeq = pylcp.rateeq(laserBeams, magField, ham)

        Neq[ii] = rateeq.equilibrium_populations(
            np.array([0., 0., 0.]),
            np.array([0., 0., 0.]),
            0.
        )

        Rijl[ii] = np.sum(rateeq.Rijl['g->e'], axis=2)[0][0]

    ax.plot(dets, Rijl*(Neq[:, 0]-Neq[:, 1]))
    ax.plot(dets, intensity/2/(1+intensity+4*dets**2), 'k--', linewidth=0.5)

ax.set_xlabel('$\Delta/\Gamma$')
ax.set_ylabel('$R_{sc}/\Gamma$');
```



Coupling three states together

Instead of just considering a single laser coupling the $m_F = 0$ and $m'_F = -1$, we can add in a second laser and consider coupling $m_F = 0$ to $m'_F = +1$ as well. We add this coupling laser by adding a counter propogating laser with the same circular polarization *relative to its :math:`k` vector*. Relative to the quanitization axis, this laser has σ^+ light.

We then compare to the modified analytic formula

$$R_{sc} = \frac{1}{2}\frac{s}{1 + 3s/2 + 4\Delta^2/\Gamma^2},$$

which is shown in the plot as black dashed lines.

[5]:
```
# Make two independent axes: dets/betas:
fig, ax = plt.subplots(nrows=1, ncols=1)
for intensity in intensities:
    Rijl = np.zeros(dets.shape + (2,))
    Neq = np.zeros(dets.shape + (4,))
    for ii, det in enumerate(dets):
        laserBeams = pylcp.laserBeams(
            [{'kvec':np.array([1., 0, 0.]),
              's':intensity, 'pol':-1, 'delta':det},
             {'kvec':np.array([-1., 0, 0.]),
              's':intensity, 'pol':-1, 'delta':det}]
```

(continues on next page)

(continued from previous page)

```
        )

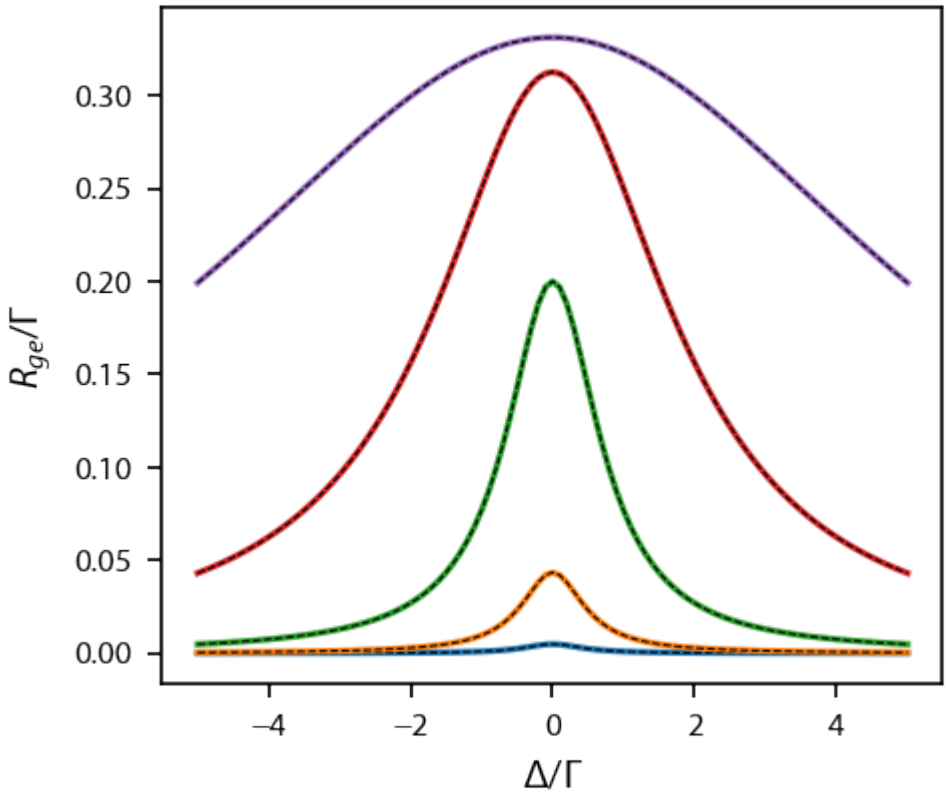
        rateeq = pylcp.rateeq(laserBeams, magField, ham)

        Neq[ii] = rateeq.equilibrium_populations(
            np.array([0., 0., 0.]),
            np.array([0., 0., 0.]),
            0.
        )

        Rijl[ii] = np.sum(rateeq.Rijl['g->e'], axis=2)[:, 0]

    ax.plot(dets, Rijl[:, 0]*(Neq[:, 0]-Neq[:, 1]))
    ax.plot(dets, intensity/2/(1+3*intensity/2+4*dets**2), 'k--', linewidth=0.5)

ax.set_xlabel('$\Delta/\Gamma$')
ax.set_ylabel('$R_{ge}/\Gamma$');
```



3.1.2 Rabi Flopping

This example covers Rabi flopping of a single spin in a magnetic field. WeâĂŹll make a spin with quantum number $F = 1/2$, place it in a magnetic field to split the states, then evolve it to watch the spin rotate.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
from pylcp.common import spherical2cart, cart2spherical
```

Define the problem

In this example, we include only one manifold (the ground manifold) simply because we do not need to worry about lasers and damping. We also want to create the spin operator $\mathbf{F} = -\boldsymbol{\mu}/(\mu_B g_F)$, which tells us which direction the spin is pointing on average. Combined with `obe.observable()`, we calculate $\langle \mathbf{F} \rangle$. Note that in `pylcp`, $g_F > 0$ implies a magnetic moment $\boldsymbol{\mu}$ that is antialigned with $\mathbf{F}$. This convention is standard when the magnetic moment is dominated by the electron. In this configuration, the spin will rotate counter-clockwise when viewed from the tip of the magnetic field vector.

[2]:
```
gF = 1
H_0, mu_q = pylcp.hamiltonians.singleF(1/2, gF=gF, muB=1)

# Construct operators for calculation of expectation values of spin (F) and mu:
mu = spherical2cart(mu_q)
F = -mu/gF # Note that muB=1

hamiltonian = pylcp.hamiltonian()
hamiltonian.add_H_0_block('g', H_0)
hamiltonian.add_mu_q_block('g', mu_q)

magField = pylcp.constantMagneticField(np.array([1., 0., 0.]))
laserBeams = {}
```

Evolve with $B = (1, 0, 0)$

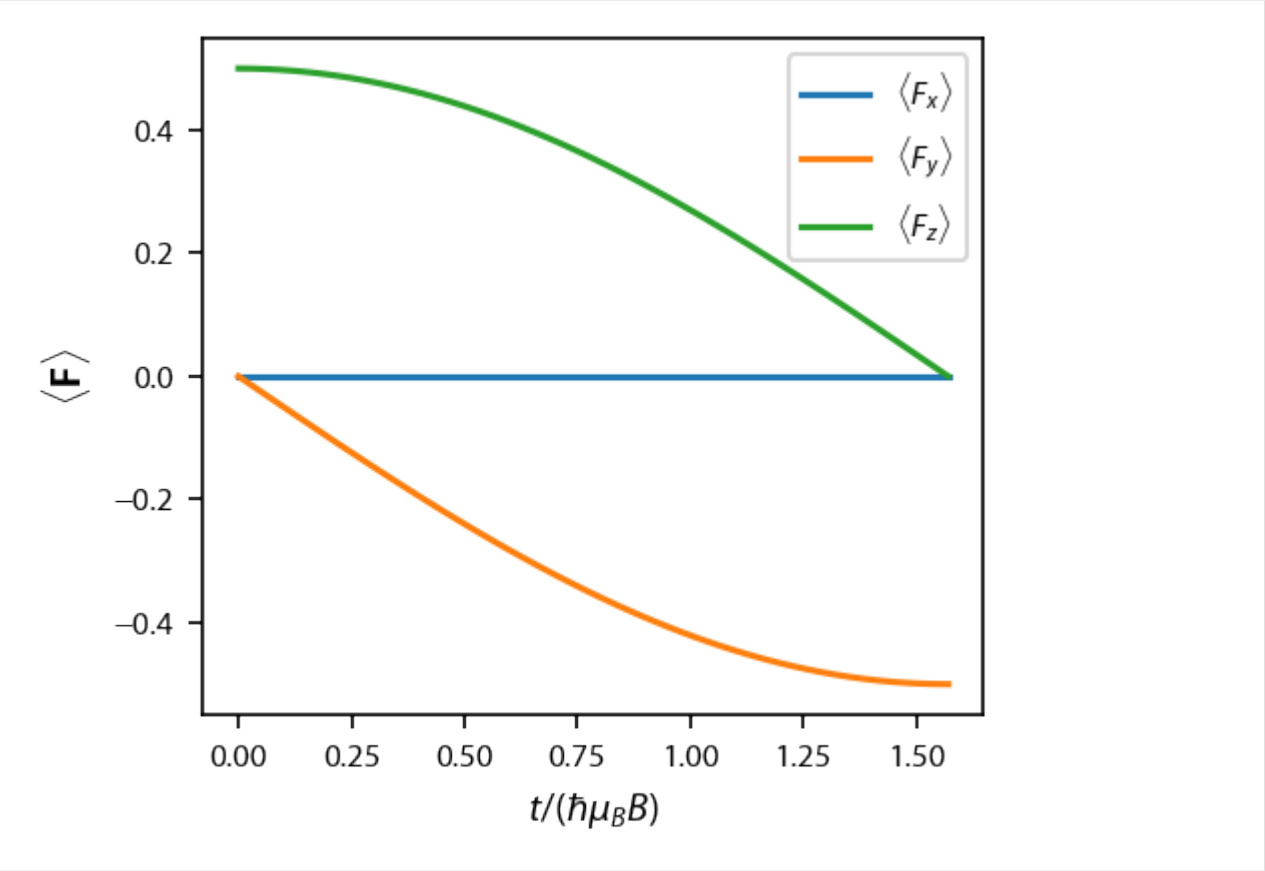
This should make it rotate in the $\hat{y}$-$\hat{z}$ plane. After $t = \pi\hbar\mu_B B$, it should end up along $-\hat{y}$, if $g_F > 0$.

[3]:
```
obe = pylcp.obe(laserBeams, magField, hamiltonian, transform_into_re_im=False)
pop = np.zeros((H_0.shape[0],))
pop[-1] = 1

obe.set_initial_rho_from_populations(pop)
obe.evolve_density([0, np.pi/2], t_eval=np.linspace(0, np.pi/2, 51))
avF = obe.observable(F)

fig, ax = plt.subplots(1, 1)

lbls = ['$\\langle F_x\\rangle$', '$\\langle F_y\\rangle$', '$\\langle F_z\\rangle$']
[ax.plot(obe.sol.t, avF[ii], label=lbls[ii]) for ii in range(3)]
ax.set_xlabel('$t/(\hbar \mu_B B)$')
ax.set_ylabel('$\langle \\mathbf{F}\\rangle$')
ax.legend(fontsize=8);
```



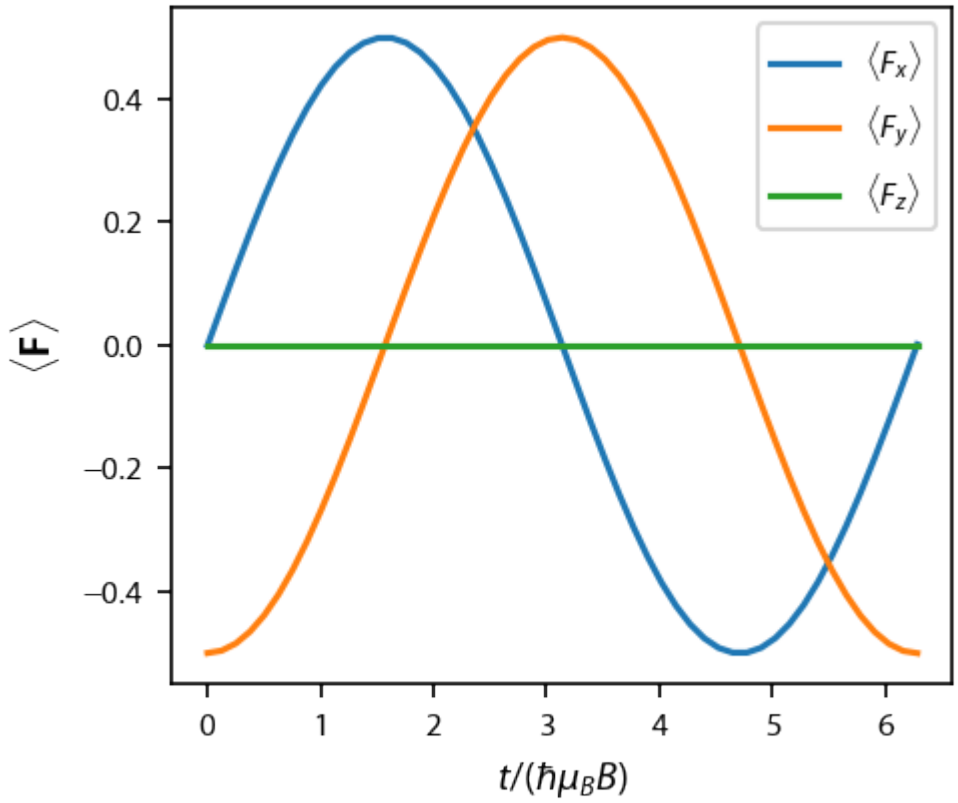
Evolve again with $B = (0, 0, 1)$

This should make it rotate in the $\hat{x}$-$\hat{y}$ plane, couter clockwise when viewed from $+\hat{z}$.

Note that rather than redefining the OBEs, I just replaced its internal `magField` variable. This is possible, because the OBEs construction in `pylcp` are only dependent on the Hamiltonian, not the external fields.

[4]:
```
obe.set_initial_rho(obe.sol.rho[:, :, -1])
obe.magField = pylcp.magField(lambda R: np.array([0., 0., 1.]))
obe.evolve_density([0, 2*np.pi], t_eval=np.linspace(0, 2*np.pi, 51))
avF = obe.observable(F)

fig, ax = plt.subplots(1, 1)
[ax.plot(obe.sol.t, avF[ii], label=lbls[ii]) for ii in range(3)]
ax.set_xlabel('$t/(\hbar \mu_B B)$')
ax.set_ylabel('$\langle \\mathbf{F}\\rangle$')
ax.legend(fontsize=8);
```



3.1.3 Damped Rabi Flopping

This example covers damped Rabi flopping as calculated with the optical Bloch equations.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
```

Define the problem

As always, we must define the laser beams, magnetic field, and Hamiltonian. Here, we will make a two-state system that is magnetic field insensitive and connected only by π light. In this particular case, we show how we can define the rotating frame such that the excited state of the Hamiltonian can rotate and the laser beams can rotate, or some combination of the two. The total detuning is the sum of `ham_det` and `laser_det`.

[3]:
```
ham_det = -2.
laser_det = -2.

laserBeams = pylcp.laserBeams(
    [{'kvec':np.array([1., 0., 0.]), 'pol':np.array([0., 0., 1.]),
      'pol_coord':'cartesian', 'delta':laser_det, 's':20.}]
)
```

(continues on next page)

(continued from previous page)

```
magField = lambda R: np.zeros(R.shape)

# Now define the extremely simple Hamiltonian:
Hg = np.array([[0.]])
mugq = np.array([[[0.]], [[0.]], [[0.]]])
He = np.array([[-ham_det]])
mueq = np.array([[[0.]], [[0.]], [[0.]]])
dijq = np.array([[[0.]], [[1.]], [[0.]]])

gamma = 1

hamiltonian = pylcp.hamiltonian(Hg, He, mugq, mugq, dijq, gamma=gamma)
hamiltonian.print_structure()
```

```
[[((<g|H_0|g> 1x1), (<g|mu_q|g> 1x1)) (<g|d_q|e> 1x1)]
 [(<e|d_q|g> 1x1) ((<e|H_0|e> 1x1), (<e|mu_q|e> 1x1))]]
```

Create the governing equation

In this example, we create both the rate equations and optical Bloch equations to compare. We also print out the decay rates given the chosen Γ to ensure the decay matrix evolution is being constructed properly for a two-level system.

[4]:
```
# First the OBE:
obe = pylcp.obe(laserBeams, magField, hamiltonian,
                transform_into_re_im=False)
rateeq = pylcp.rateeq(laserBeams, magField, hamiltonian)
print(obe.ev_mat['decay'])
```

```
[[ 0. +0.j  0. +0.j  0. +0.j  1. +0.j]
 [ 0. +0.j -0.5+0.j  0. +0.j  0. +0.j]
 [ 0. +0.j  0. +0.j -0.5+0.j  0. +0.j]
 [ 0. +0.j  0. +0.j  0. +0.j -1. +0.j]]
```

Evolve the state

We are going to evolve for $T = 4\pi/\Gamma$, to see multiple oscillations.

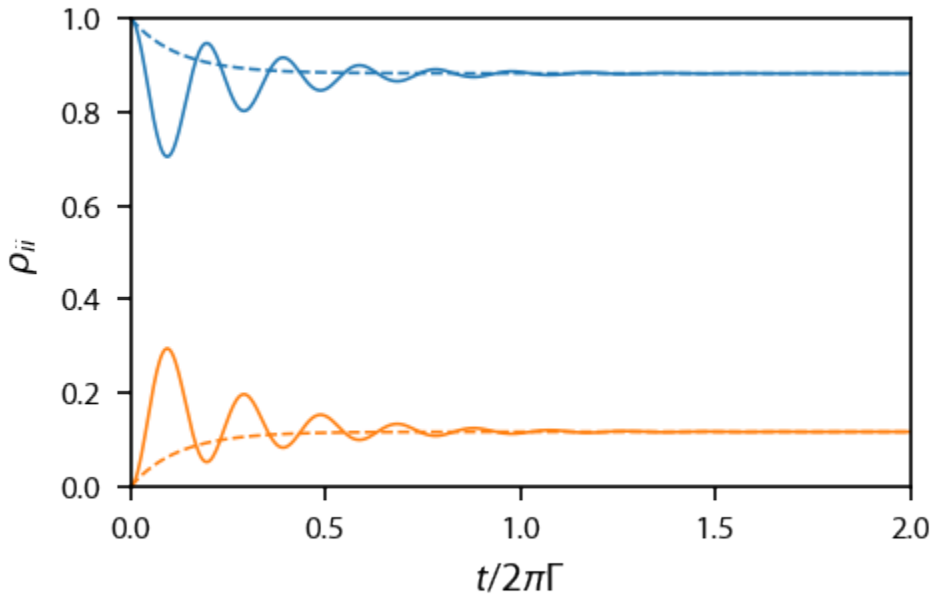
[5]:
```
t_eval = np.linspace(0, 4*np.pi/gamma, 501)
rho0 = np.zeros((hamiltonian.n**2,), dtype='complex128')
rho0[0] = 1.
obe.set_initial_rho(rho0)
obe.evolve_density([t_eval[0], t_eval[-1]], t_eval=t_eval)

N0 = np.zeros((rateeq.hamiltonian.n,))
N0[0] = 1
rateeq.set_initial_pop(N0)
rateeq.evolve_populations([t_eval[0], t_eval[-1]], t_eval=t_eval)
```

Plot it all up:

[6]:
```
def final_value(s, det):
    return s/2/(1+s+4*det**2)

fig, ax = plt.subplots(1, 1, num='evolution', figsize=(3.25, 2.))
ax.plot(obe.sol.t*gamma/2/np.pi, np.abs(obe.sol.rho[0, 0, :]), linewidth=0.75,
        label='$\\rho_{00}$')
ax.plot(obe.sol.t*gamma/2/np.pi, np.abs(obe.sol.rho[1, 1, :]), linewidth=0.75,
        label='$\\rho_{11}$')
ax.plot(rateeq.sol.t*gamma/2/np.pi, np.abs(rateeq.sol.y[0, :]), linewidth=0.75,
        label='$\\rho_{00}$ (rate eq.)', color='C0', linestyle='--')
ax.plot(rateeq.sol.t*gamma/2/np.pi, np.abs(rateeq.sol.y[-1, :]), linewidth=0.75,
        label='$\\rho_{11}$ (rate eq.)', color='C1', linestyle='--')
# ax.plot(obe.sol.t[-1]*gamma/2/np.pi,
#          final_value(len(laserBeams)*laserBeams[0].beta(np.array([0., 0., 0.])),
#                      ham_det+laser_det) ,'o')
# ax.legend(fontsize=6)
ax.set_xlabel('$t/2\pi\Gamma$')
ax.set_ylabel('$\\rho_{ii}$')
ax.set_xlim((0., 2))
ax.set_ylim((0., 1))
fig.subplots_adjust(bottom=0.2)
```



3.1.4 Adiabatic Rapid Passage

This example covers an example with both laser frequency and amplitude modulation: rapid adiabatic passage. It reproduces Fig. 2 from T. Lu, X. Miao, and H. Metcalf, âĂIJBloch theorem on the Bloch sphereâĂİ *Physical Review A* **71**, 061405(R) (2005), http://dx.doi.org/10.1103/PhysRevA.71.061405

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
import pylcp
from pylcp.common import progressBar
```

Code the basic Hamiltonian

The Hamiltonian is a simple two-state Hamiltonian. Here, we code up a method to return the time-dependent Hamiltonian matrix to evolve it with the Schrodinger equation. We take the modulation to be the same as in Lu, *et. al.*, above: $\Omega(t) = \Omega_0 \sin(\omega_m t)$ and $\Delta = \Delta_0 \cos(\omega_m t)$.

[2]:
```
def H(t, Delta0, Omega0, omegam):
    Delta = Delta0*np.cos(omegam*t)
    Omega = Omega0*np.sin(omegam*t)

    return 1/2*np.array([[Delta, Omega], [Omega, -Delta]])
```

Define the problem in pylcp

Unlike the above Hamiltonian, which is clearly time dependent, the only time dependence available to us in pylcp is through the fields. First, we need to remember for the amplitude modulation that $I/I_{\rm sat} = 2\Omega^2/\Gamma^{=}2[\Omega_0 \sin(\omega_m t)]^2/\Gamma^2$. Second, we need to frequency modulate the laser beams. Remember that if the lasers have a temporal phase ϕ, the frequency is $\omega = \frac{d\phi}{dt}$. Thus, if the detuning $\Delta(t)$ is specified, then $\phi = \int \Delta(t)\, dt$. pylcp contains a built-in integrator in order to convert the detuning to a phase, but it might not always be reliable. So you can also reproduce the detuning by modulating the phase ϕ. To reproduce the $\Delta(t) = \Delta_0 \cos(\omega_m t)$, we need $\phi = \Delta_0/\omega_m \sin(\omega_m t)$.

[3]:
```
def return_lasers(Delta0, Omega0, omegam):
    laserBeams = pylcp.laserBeams([
        {'kvec':np.array([1., 0., 0.]),
         'pol':np.array([0., 0., 1.]),
         'pol_coord':'cartesian',
         'delta': lambda t: Delta0*np.cos(omegam*t),
         'phase': 0,#lambda t: Delta0/omegam*np.sin(omegam*t),
         's': lambda R, t: 2*(Omega0*np.sin(omegam*t))**2
        }])

    return laserBeams

magField = lambda R: np.zeros(R.shape)

# Now define the extremely simple Hamiltonian:
Hg = np.array([[0.]])
mugq = np.array([[[0.]], [[0.]], [[0.]]])
```

(continues on next page)

(continued from previous page)

```
He = np.array([[0.]])
mueq = np.array([[[0.]], [[0.]], [[0.]]])
dijq = np.array([[[0.]], [[1.]], [[0.]]])

gamma = 1.

hamiltonian = pylcp.hamiltonian(Hg, He, mugq, mueq, dijq, gamma=gamma)
hamiltonian.print_structure()
```

```
[[((<g|H_0|g> 1x1), (<g|mu_q|g> 1x1)) (<g|d_q|e> 1x1)]
 [(<e|d_q|g> 1x1) ((<e|H_0|e> 1x1), (<e|mu_q|e> 1x1))]]
```

Evolve a single state

We solve with both the Schrodinger Equation and the OBEs. We can play with the modulation parameters (Δ_0, Ω_0, ω_m), and see how the result chanages.

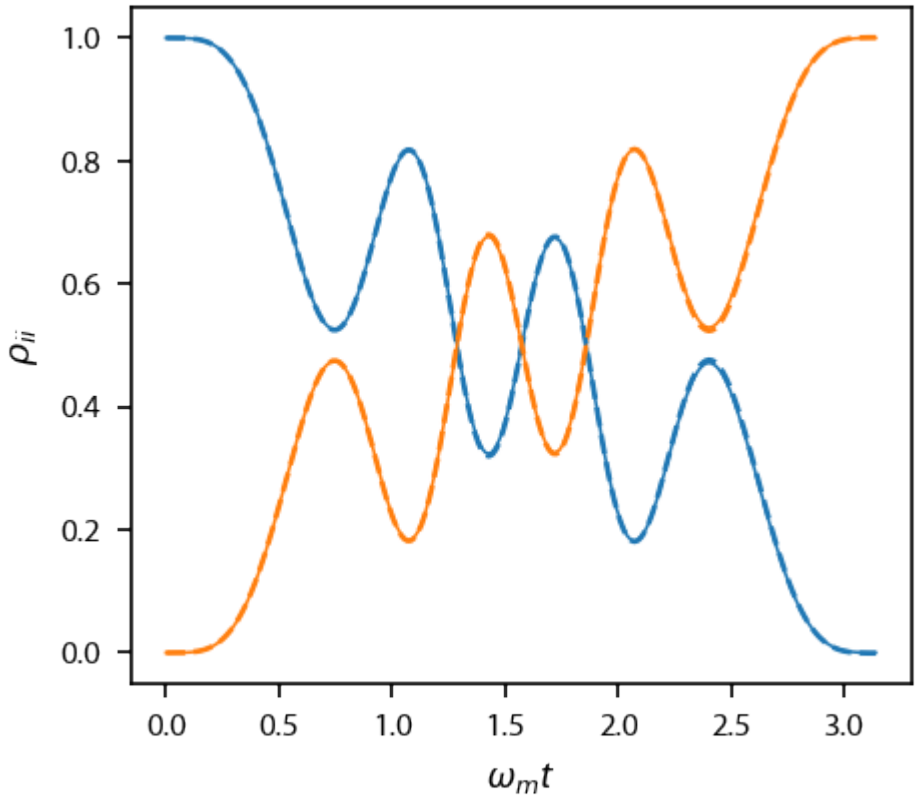
[4]:
```
Delta0 = 5.
Omega0 = 10.
omegam = 1.

t = np.linspace(0., np.pi/omegam, 201)
sol_SE = solve_ivp(lambda t, y: -1j*H(t, Delta0, Omega0, omegam)@y, [0, np.pi/omegam],
                   np.array([1., 0.], dtype='complex128'), t_eval=t)

laserBeams = return_lasers(Delta0, Omega0, omegam)
obe = pylcp.obe(laserBeams, magField, hamiltonian)
obe.ev_mat['decay'] = np.zeros(obe.ev_mat['decay'].shape) # Turn off damping to compare␣
↪to Schrodinger Equation.
obe.set_initial_rho_from_populations(np.array([1., 0.]))
sol_OBE = obe.evolve_density([0, np.pi/omegam], t_eval=t)
```

Plot it up. Dashed is OBEs from `pylcp` and solid is the Schrodiner equation. Orange is the $|1\rangle$ state; blue is the $|0\rangle$ state.

[5]:
```
fig, ax = plt.subplots(1, 1)
ax.plot(sol_SE.t, np.abs(sol_SE.y[0])**2, linewidth=0.75)
ax.plot(sol_SE.t, np.abs(sol_SE.y[1])**2, linewidth=0.75)
ax.plot(sol_OBE.t, np.real(sol_OBE.rho[0, 0]), '--', color='C0', linewidth=1.25)
ax.plot(sol_OBE.t, np.real(sol_OBE.rho[1, 1]), '--', color='C1', linewidth=1.25)
ax.set_xlabel('$\omega_m t$')
ax.set_ylabel('$\\rho_{ii}$');
```



Reproduce Fig. 2

This involves scanning over Δ_0 and Ω_0. This takes a long time, only because there is a fair amount of overhead in regenerating the optical Bloch equations on every iteration.

[6]:
```
Delta0s = np.arange(0.25, 25.1, 0.25)
Omega0s = np.arange(0.25, 25.1, 0.25)
t = np.linspace(0., np.pi, 201)

DELTA0S, OMEGA0S = np.meshgrid(Delta0s, Omega0s)

it = np.nditer([DELTA0S, OMEGA0S, None])

progress = progressBar()

for (Delta0, Omega0, rhogg) in it:
    del laserBeams

    # Set up new laser beams:
    laserBeams = return_lasers(Delta0, Omega0, omegam)

    # Set up OBE:
    obe = pylcp.obe(laserBeams, magField, hamiltonian)
    obe.ev_mat['decay'] = np.zeros(obe.ev_mat['decay'].shape) # Turn off damping to␣
↪compare to Schrodinger Equation.
```

(continues on next page)

(continued from previous page)

```
    obe.set_initial_rho_from_populations(np.array([1., 0.]))

    # Solve:
    sol_OBE = obe.evolve_density([0, np.pi], t_eval=t)

    # Save result:
    rhogg[...] = np.real(sol_OBE.rho[0, 0, -1])

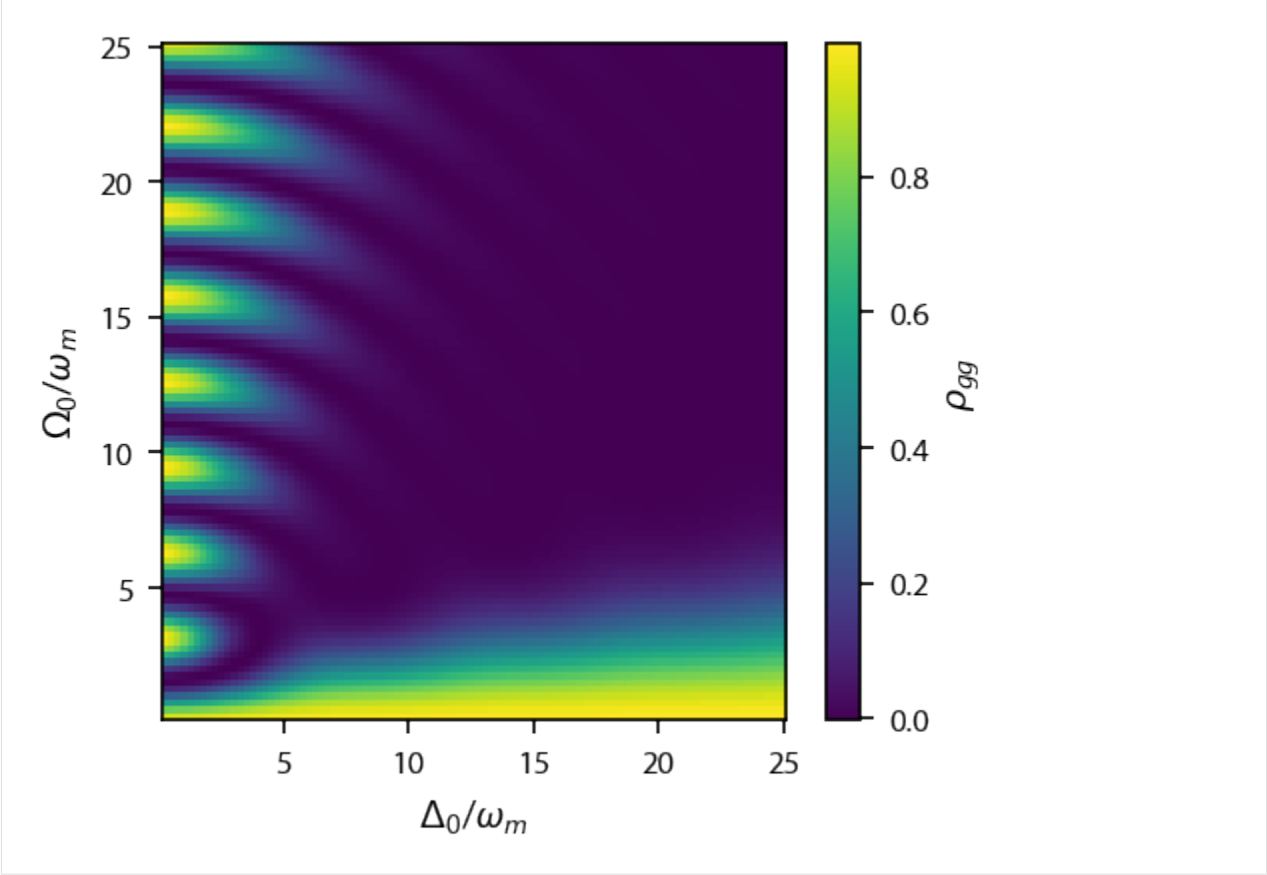
    # Update progress bar:
    progress.update((it.iterindex+1)/it.itersize)

RHOGG = it.operands[2]
```

```
Completed in 8:02.
```

Plot it up:

[7]:
```
dDelta0 = np.mean(np.diff(Delta0s))
dOmega0 = np.mean(np.diff(Omega0s))
fig, ax = plt.subplots(1, 1)
im = ax.imshow(RHOGG, origin='lower',
               extent=(np.amin(Delta0s)-dDelta0/2, np.amax(Delta0s)+dDelta0/2,
                       np.amin(Omega0s)-dOmega0/2, np.amax(Omega0s)+dOmega0/2),
               aspect='auto')
ax_cbar = plt.colorbar(im)
ax_cbar.set_label('$\\rho_{gg}$')
ax.set_xlabel('$\Delta_0/\omega_m$')
ax.set_ylabel('$\Omega_0/\omega_m$');
```



Add in damping

These two cells are exactly the same as above, except the user must now specify ω_m/Γ ($\Gamma = 1$) and we merely comment out the line that eliminate the damping part of the OBEs.

[8]:
```
omegam = 2
Delta0s = np.arange(0.25, 25.1, 0.25) # These will be normalized to the value of omegam
Omega0s = np.arange(0.25, 25.1, 0.25)
t = np.linspace(0., np.pi/omegam, 201)

DELTA0S, OMEGA0S = np.meshgrid(Delta0s, Omega0s)

it = np.nditer([DELTA0S, OMEGA0S, None])

progress = progressBar()

for (Delta0, Omega0, rhogg) in it:
    # Set up new laser beams:
    laserBeams = return_lasers(omegam*Delta0, omegam*Omega0, omegam)

    # Set up OBE:
    obe = pylcp.obe(laserBeams, magField, hamiltonian)
    obe.set_initial_rho_from_populations(np.array([1., 0.]))
```

(continues on next page)

(continued from previous page)

```
    # Solve:
    sol_OBE = obe.evolve_density([0, np.pi/omegam], t_eval=t)

    # Save result:
    rhogg[...] = np.real(sol_OBE.rho[0, 0, -1])

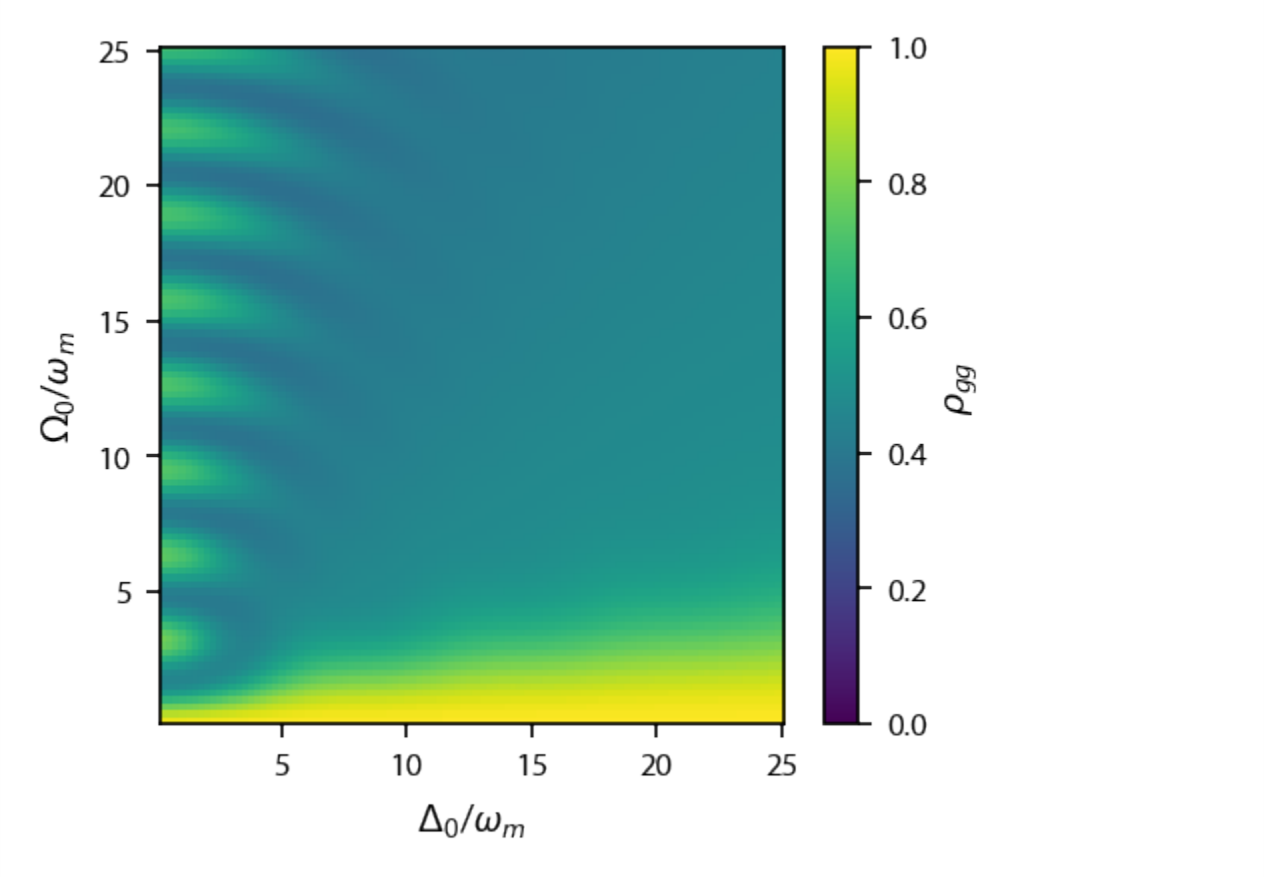
    # Update progress bar:
    progress.update((it.iterindex+1)/it.itersize)

RHOGG = it.operands[2]
```

```
Completed in 7:22.
```

Plot it up:

[9]:
```
dDelta0 = np.mean(np.diff(Delta0s))
dOmega0 = np.mean(np.diff(Omega0s))
fig, ax = plt.subplots(1, 1)
im = ax.imshow(RHOGG, origin='lower',
               extent=(np.amin(Delta0s)-dDelta0/2, np.amax(Delta0s)+dDelta0/2,
                       np.amin(Omega0s)-dOmega0/2, np.amax(Omega0s)+dOmega0/2),
               aspect='auto', clim=(0, 1))
ax_cbar = plt.colorbar(im)
ax_cbar.set_label('$\\rho_{gg}$')
ax.set_xlabel('$\Delta_0/\omega_m$')
ax.set_ylabel('$\Omega_0/\omega_m$');
```



3.1.5 Optical Pumping

This little script tests the optical pumping from the optical Bloch equations and rate equations. It reproduces Fig. 5 of Ungar, P. J., Weiss, D. S., Riis, E., & Chu, S., âĂIJOptical molasses and multilevel atoms: theoryâĂİ, *Journal of the Optical Society of America B*, **6** (11), 2058 (1989). http://doi.org/10.1364/JOSAB.6.002058

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
from pylcp.common import spherical2cart

transform = False # Change the variable to transform OBEs into re/im components.
```

Define the problem

As always, we first define the laserBeams, Hamiltonian, and magnetic field. Here, we are interested in a $F = 2 \rightarrow F' = 3$ transition under linearly polarized light. We make three combinations of laser beams, each with linear polarization along a different axis. Note that agreement between rate equations and the optical Bloch equations will only occur with the rate equations in the case of a single laser beam. This is because the rate equations assume that thelasers are incoherent (their electric fields do not add to give twice the amplitude) whereas the optical Bloch equations do. Specifically, two coherent beams doubles the electric field which quadrupoles the intensity, so to compare the rate equations, we have to mulitply by 4. We do that for π_y and π_z polarizations. For the π_x beams, we separate it into two beams.

Compared to Ungar *et. al.*, the saturation parameter is defined differently. ThereâĂŹs is defined using $\gamma = \Gamma/2 - i\Delta$, which is different from ours. Namely, the ratio between the two is $(\Gamma^2 + \Delta^2)/\Gamma^2 = 1 + \Delta^2/\Gamma^2$.

Finally, one can put the detuning on the laser or put the detuning on the Hamiltonian (or some combination of the two). The latter appears to be faster.

[2]:
```
gamma = 1 # Also can demonstrate how to change gamma from 1

# First the laser beams:
laserBeams = {}
laserBeams['$\\pi_z$']= pylcp.laserBeams([
    {'kvec': np.array([1., 0., 0.]), 'pol':np.array([0., 0., 1.]),
     'pol_coord':'cartesian', 'delta':-2.73*gamma, 's':4*0.16*(1+2.73**2)}
    ])
laserBeams['$\\pi_y$']= pylcp.laserBeams([
    {'kvec': np.array([0., 0., 1.]), 'pol':np.array([0., 1., 0.]),
     'pol_coord':'cartesian', 'delta':-2.73*gamma, 's':4*0.16*(1+2.73**2)}
    ])
laserBeams['$\\pi_x$']= pylcp.laserBeams([
    {'kvec': np.array([0., 0., 1.]), 'pol':np.array([1., 0., 0.]),
     'pol_coord':'cartesian', 'delta':-2.73*gamma, 's':0.16*(1+2.73**2)},
    {'kvec': np.array([0., 0., -1.]), 'pol':np.array([1., 0., 0.]),
     'pol_coord':'cartesian', 'delta':-2.73*gamma, 's':0.16*(1+2.73**2)}
    ])

# Then the magnetic field:
magField = lambda R: np.zeros(R.shape)

# Hamiltonian for F=2->F=3
H_g, muq_g = pylcp.hamiltonians.singleF(F=2, gF=1, muB=1)
H_e, mue_q = pylcp.hamiltonians.singleF(F=3, gF=1, muB=1)
d_q = pylcp.hamiltonians.dqij_two_bare_hyperfine(2, 3)
hamiltonian = pylcp.hamiltonian()
hamiltonian.add_H_0_block('g', H_g)
hamiltonian.add_H_0_block('e', H_e-0.*np.eye(H_e.shape[0]))
hamiltonian.add_d_q_block('g', 'e', d_q, gamma=gamma)

hamiltonian.print_structure()
```

```
[[(<g|H_0|g> 5x5) (<g|d_q|e> 5x7)]
 [(<e|d_q|g> 7x5) (<e|H_0|e> 7x7)]]
```

Evolve the density/populations

Using both the `rateeq` and `obe`, we will calculate the population transfer.

[3]:
```
obe = {}
rateeq = {}
rateeq['$\\pi_z$'] = pylcp.rateeq(laserBeams['$\\pi_z$'], magField,
                                  hamiltonian)
obe['$\\pi_z$'] = pylcp.obe(laserBeams['$\\pi_z$'], magField, hamiltonian,
                            transform_into_re_im=transform)
```

(continues on next page)

(continued from previous page)

```
# Run the rate equations:
N0 = np.zeros((rateeq['$\\pi_z$'].hamiltonian.n,))
N0[0] = 1
rateeq['$\\pi_z$'].set_initial_pop(N0)
rateeq['$\\pi_z$'].evolve_populations([0, 600/gamma],
                                      max_step=1/gamma)

# Run the OBEs:
rho0 = np.zeros((obe['$\\pi_z$'].hamiltonian.n**2,))
rho0[0] = 1.
obe['$\\pi_z$'].set_initial_rho(np.real(rho0))
obe['$\\pi_z$'].evolve_density(t_span=[0, 600/gamma],
                               progress_bar=True)

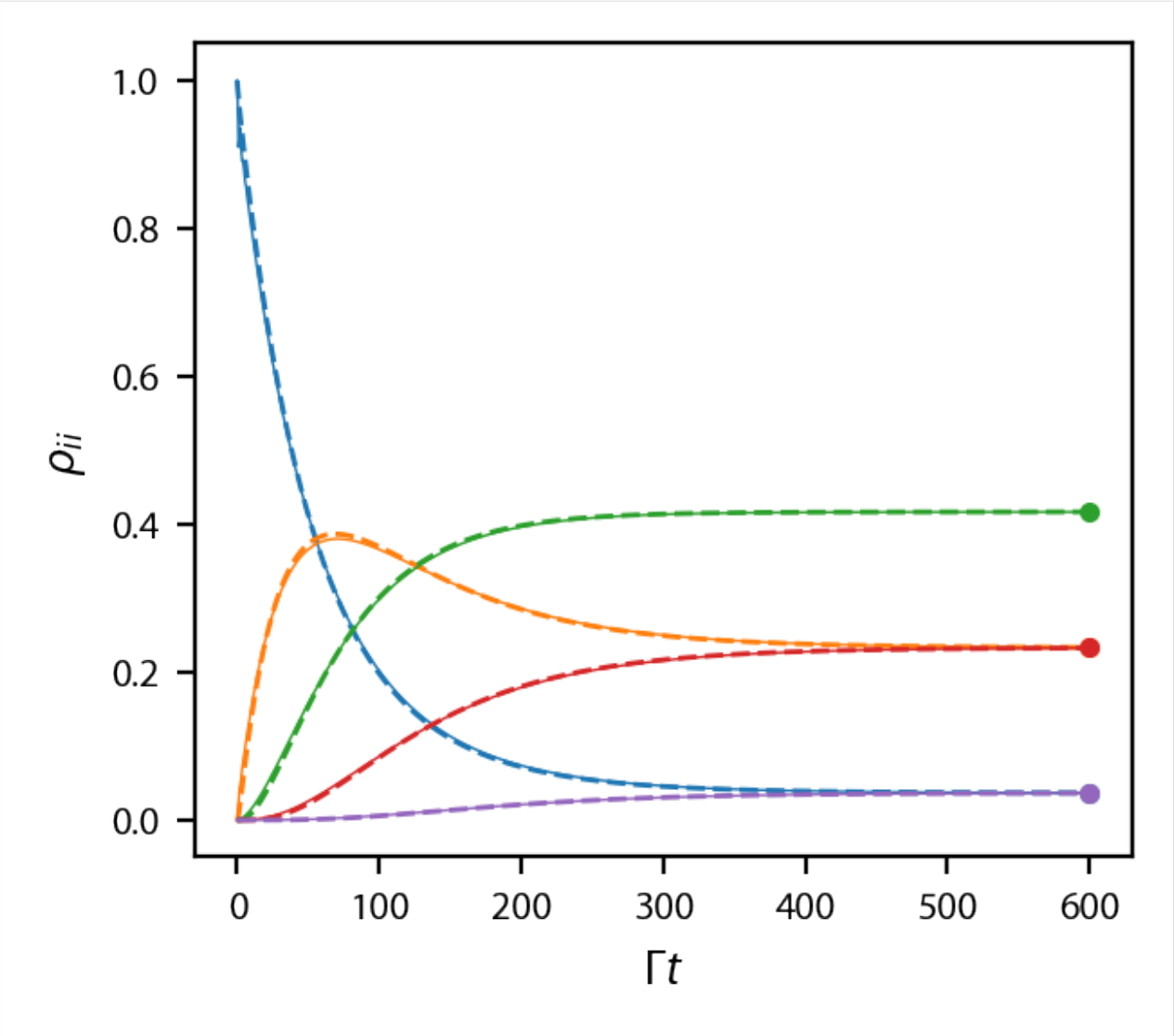
# Calculate the equilibrium populations:
Neq = rateeq['$\\pi_z$'].equilibrium_populations(np.array([0., 0., 0.]),
                                                 np.array([0., 0., 0.]), 0.)
```

```
Completed in 1.76 s.
```

Plot up the results:

[4]:
```
fig, ax = plt.subplots(1, 1)
for jj in range(5):
    ax.plot(gamma*rateeq['$\\pi_z$'].sol.t,
            rateeq['$\\pi_z$'].sol.y[jj, :], '--',
            color='C{0:d}'.format(jj),
            linewidth=1.0)
    ax.plot(gamma*obe['$\\pi_z$'].sol.t, np.abs(obe['$\\pi_z$'].sol.rho[jj, jj]), '-',
            color='C{0:d}'.format(jj),
            linewidth=0.5)
    ax.plot(gamma*obe['$\\pi_z$'].sol.t[-1], Neq[jj], '.', color='C{0:d}'.format(jj),
            linewidth=0.5)

ax.set_xlabel('$\\Gamma t$')
ax.set_ylabel('$\\rho_{ii}$');
```



Check rotations

Next, we want to check that our rotations are working properly, so we will run the same calculation for the $\hat{z}$ going beam with π_y polarization. But before we even bother working with the OBE, we need to create the initial state first, which involves rotating our state.

[5]:
```
mug = spherical2cart(muq_g)
S = -mug

# What are the eigenstates of 'y'?
E, U = np.linalg.eig(S[1])

# Let's now define a rotation matrix that rotates us into the m_F basis alogn y:
inds = np.argsort(E)
E = E[inds]
U = U[:, inds]
```

(continues on next page)

(continued from previous page)

```
Uinv = np.linalg.inv(U)

# In a positive magnetic field with g_F>0, I want the lowest eigenvalue. That
# corresponds to the -m_F state.
psi = U[:, 0]

# Now take that state and make the initial density matrix:
rho0 = np.zeros((hamiltonian.n, hamiltonian.n), dtype='complex128')
for ii in range(hamiltonian.ns[0]):
    for jj in range(hamiltonian.ns[0]):
        rho0[ii, jj] = psi[ii]*np.conjugate(psi[jj])

#Print out the density matrix (in z-basis), and the rotated density matrix (in y-basis):
#print(rho0[:5,:5])
#print(Uinv@rho0[:5,:5]@U)

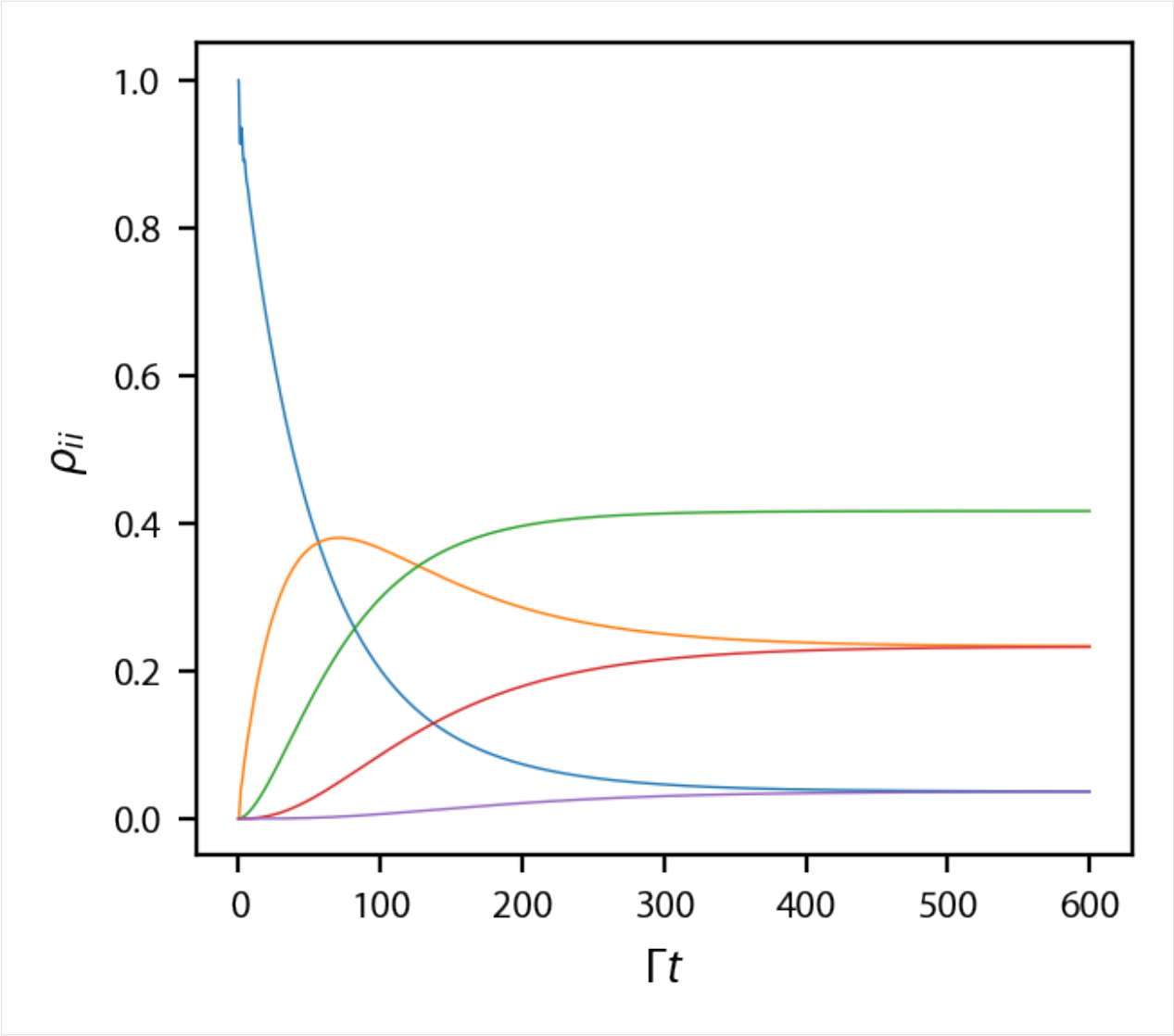
# Evolve:
obe['$\\pi_y$'] = pylcp.obe(laserBeams['$\\pi_y$'], magField, hamiltonian,
                            transform_into_re_im=transform)
obe['$\\pi_y$'].set_initial_rho(rho0.reshape(hamiltonian.n**2,))
obe['$\\pi_y$'].evolve_density(t_span=[0, 600],
                               progress_bar=True)

# Now rotate the denisty matrix back to be along $y$:
for jj in range(obe['$\\pi_y$'].sol.t.size):
    obe['$\\pi_y$'].sol.rho[:5, :5, jj] = Uinv@obe['$\\pi_y$'].sol.rho[:5, :5, jj]@U
```

```
Completed in 1.71 s.
```

Now plot it up:

[6]:
```
fig, ax = plt.subplots(1, 1)
for jj in range(5):
    ax.plot(obe['$\\pi_y$'].sol.t,
            np.abs(obe['$\\pi_y$'].sol.rho[jj, jj]), '-',
            color='C{0:d}'.format(jj),
            linewidth=0.5)
ax.set_xlabel('$\\Gamma t$')
ax.set_ylabel('$\\rho_{ii}$');
```



Now, letâĂŹs do the same thing for π_x, except this time we have two laser beams, with 1/4 of the intensity:

[7]:
```
# What are the eigenstates of 'y'?
E, U = np.linalg.eig(S[0])

inds = np.argsort(E)
E = E[inds]
U = U[:, inds]
Uinv = np.linalg.inv(U)

# In a positive magnetic field with g_F>0, I want the lowest eigenvalue. That
# corresponds to the -m_F state.
psi = U[:, 0]

rho0 = np.zeros((hamiltonian.n, hamiltonian.n), dtype='complex128')
for ii in range(hamiltonian.ns[0]):
    for jj in range(hamiltonian.ns[0]):
```

(continues on next page)

(continued from previous page)

```
        rho0[ii, jj] = psi[ii]*np.conjugate(psi[jj])

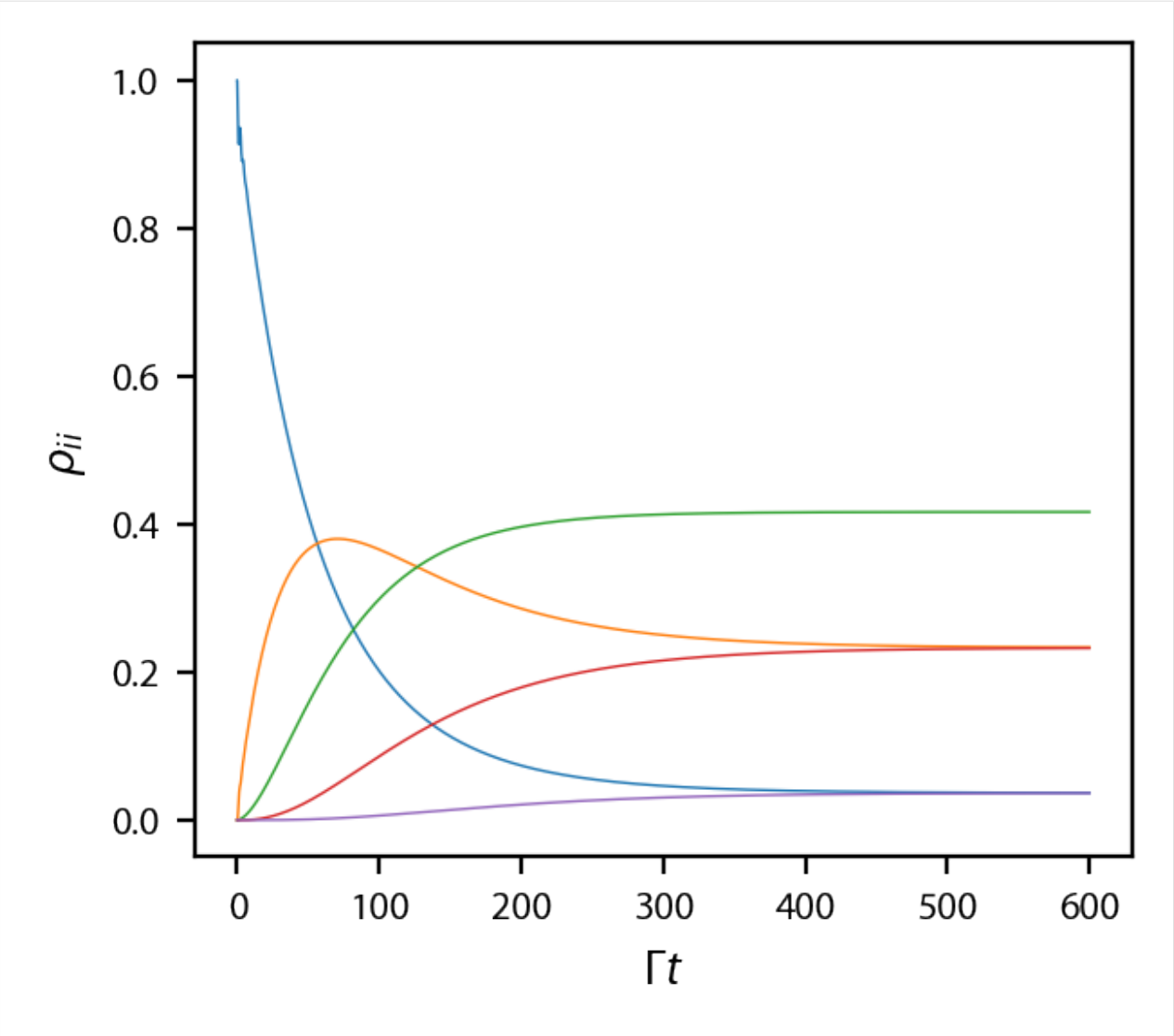
obe['$\\pi_x$'] = pylcp.obe(laserBeams['$\\pi_x$'], magField, hamiltonian,
                            transform_into_re_im=transform)
obe['$\\pi_x$'].set_initial_rho(rho0.reshape(hamiltonian.n**2,))
obe['$\\pi_x$'].evolve_density(t_span=[0, 600],
                               progress_bar=True)

for jj in range(obe['$\\pi_x$'].sol.t.size):
    obe['$\\pi_x$'].sol.rho[:5, :5, jj] = Uinv@obe['$\\pi_x$'].sol.rho[:5, :5, jj]@U
```

```
Completed in 1.97 s.
```

Plot this up too:

[8]:
```
fig, ax = plt.subplots(1, 1)
for jj in range(5):
    ax.plot(obe['$\\pi_x$'].sol.t,
            np.abs(obe['$\\pi_x$'].sol.rho[jj, jj]), '-',
            color='C{0:d}'.format(jj),
            linewidth=0.5)
ax.set_xlabel('$\\Gamma t$')
ax.set_ylabel('$\\rho_{ii}$');
```



3.1.6 Three-level susceptibility & EIT

This example demonstrates damped Rabi flopping as calculated with the optical Bloch equations for a three level system and calculates the three-level susceptibility to demonstrate EIT. It the first example with a three-manifold sytem, so we will focus on the construction of the Hamiltonian.

State notation used within:

```
    ---- |e>


          ---- |r>
---- |g>
```

The three level detuning δ and two level detuning Δ are the standard detunings for a Λ system.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
```

Define the problem

Hamiltonian

To construct the three-manifold Hamiltonian, we add the blocks to Hamiltonian one at a time. The order in which they are added is important, as the manifolds are assumed to be in increasing energy order, even though their field-independent elements should already be placed in the appropriate rotating frames, implying that those elements should not contain optical-frequency components.

Our three manifold system is really quite simple, with each manifold containing a single state. Detunings are placed on the $|r\rangle$ and $|e\rangle$ states. Finally, the states are connected through π-polarized light. We write a method to return the Hamiltonian given the specified detunings.

[2]:
```
Delta = -2; delta = 0.

H0 = np.array([[1.]])
mu_q = np.zeros((3, 1, 1))
d_q = np.zeros((3, 1, 1))
d_q[1, 0, 0,] = 1/np.sqrt(2)

def return_three_level_hamiltonian(Delta, delta):
    hamiltonian = pylcp.hamiltonian()
    hamiltonian.add_H_0_block('g', 0.*H0)
    hamiltonian.add_H_0_block('r', delta*H0)
    hamiltonian.add_H_0_block('e', -Delta*H0)
    hamiltonian.add_d_q_block('g','e', d_q)
    hamiltonian.add_d_q_block('r','e', d_q)

    return hamiltonian

hamiltonian = return_three_level_hamiltonian(Delta, delta)
hamiltonian.print_structure()
```

```
[[(<g|H_0|g> 1x1) None (<g|d_q|e> 1x1)]
 [None (<r|H_0|r> 1x1) (<r|d_q|e> 1x1)]
 [(<e|d_q|g> 1x1) (<e|d_q|r> 1x1) (<e|H_0|e> 1x1)]]
```

Lasers and magnetic fields

Here, a method returns the lasers for a given intensity. Note that function returns a dictionary of two lasers, one addressing the $|g\rangle \to |e\rangle$ transition and the other the $|r\rangle \to |e\rangle$ transition. Finally, we assume a constant zero magnitude magnetic field.

[3]:
```
# First, define the lasers (functionalized for later):
Ige = 4; Ire = 4;
def return_three_level_lasers(Ige, Ire):
    laserBeams = {}
```

(continues on next page)

(continued from previous page)

```
    laserBeams['g->e'] = pylcp.laserBeams(
        [{'kvec':np.array([1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
          'pol_coord':'spherical', 'delta':0., 's':Ige}],
        beam_type=pylcp.infinitePlaneWaveBeam
    )
    laserBeams['r->e'] = pylcp.laserBeams(
        [{'kvec':np.array([1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
          'pol_coord':'spherical', 'delta':0., 's':Ire}],
        beam_type=pylcp.infinitePlaneWaveBeam
    )
    return laserBeams

laserBeams = return_three_level_lasers(Ige, Ire)

# Second, magnetic field:
magField = lambda R: np.zeros(R.shape)
```

Damped Rabi oscillations

We first evolve the optical Bloch equations to see damped Rabi oscillations in the three-level system.

[4]:
```
obe = pylcp.obe(laserBeams, magField, hamiltonian,
                transform_into_re_im=True)
obe.set_initial_rho_from_populations(np.array([0., 1., 0.]))
obe.evolve_density([0, 100])

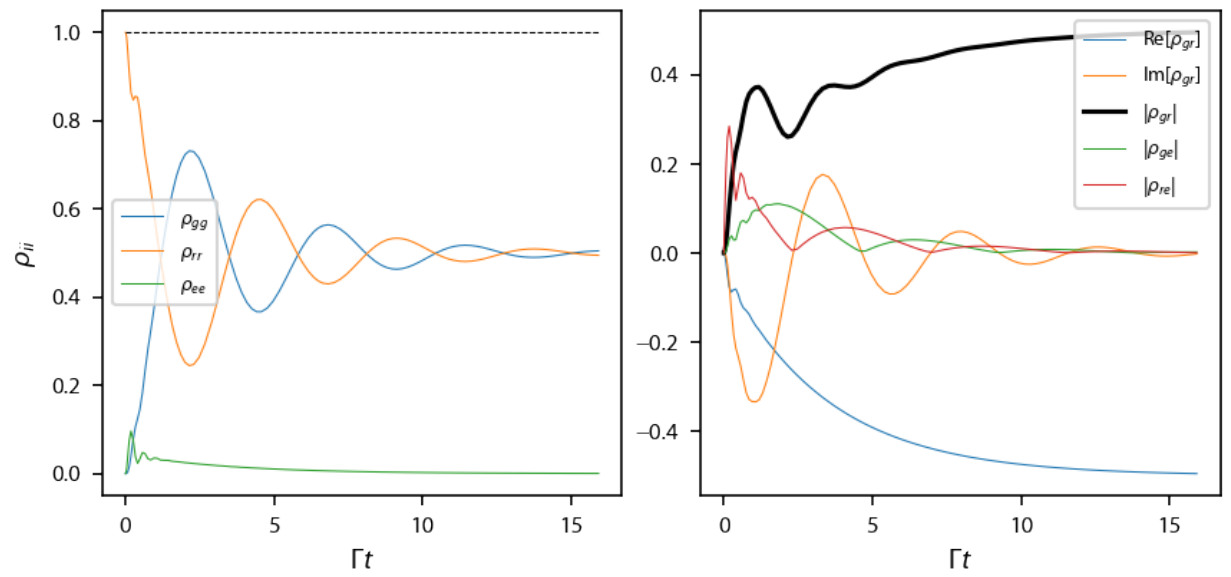
fig, ax = plt.subplots(1, 2, figsize=(6.5, 2.75))
ax[0].plot(obe.sol.t/2/np.pi, np.real(obe.sol.rho[0, 0]), linewidth=0.5, label='$\\rho_
↪{gg}$')
ax[0].plot(obe.sol.t/2/np.pi, np.real(obe.sol.rho[1, 1]), linewidth=0.5, label='$\\rho_
↪{rr}$')
ax[0].plot(obe.sol.t/2/np.pi, np.real(obe.sol.rho[2, 2]), linewidth=0.5, label='$\\rho_
↪{ee}$')
ax[0].plot(obe.sol.t/2/np.pi, np.real(obe.sol.rho[0, 0]+obe.sol.rho[1, 1]+obe.sol.rho[2,␣
↪2]), 'k--', linewidth=0.5)
ax[0].legend(fontsize=7)
ax[0].set_xlabel('$\Gamma t$')
ax[0].set_ylabel('$\\rho_{ii}$')

ax[1].plot(obe.sol.t/2/np.pi, np.real(obe.sol.rho[0, 1]), linewidth=0.5,
           label='Re$[\\rho_{gr}]$')
ax[1].plot(obe.sol.t/2/np.pi, np.imag(obe.sol.rho[0, 1]), linewidth=0.5,
           label='Im$[\\rho_{gr}]$')
ax[1].plot(obe.sol.t/2/np.pi, np.abs(obe.sol.rho[0, 1]), 'k-',
           label='$|\\rho_{gr}|$')
ax[1].plot(obe.sol.t/2/np.pi, np.abs(obe.sol.rho[0, 2]), linewidth=0.5, label='$|\\rho_
↪{ge}|$')
ax[1].plot(obe.sol.t/2/np.pi, np.abs(obe.sol.rho[1, 2]), linewidth=0.5, label='$|\\rho_
↪{re}|$')
ax[1].legend(fontsize=7)
ax[1].set_xlabel('$\Gamma t$')
```

(continues on next page)

(continued from previous page)

```
fig.subplots_adjust(wspace=0.15)
```



Susceptibility and EIT

To see EIT, we want to compute the susceptibility of the laser addressed to the $|r\rangle \rightarrow |e\rangle$ transition, which is proportional to ρ_{re}. We will use our methods returning the appropriate lasers and Hamiltonian to loop through several three level detunings δ and two level detunings Δ.

[5]:
```
Deltas = np.array([-5., -1., -0.1])
deltas = np.arange(-7., 7., 0.1)
delta_random = np.random.choice(deltas)

it = np.nditer(np.meshgrid(Deltas, deltas)+ [None,])

laserBeams = return_three_level_lasers(1, 0.1)
for Delta, delta, rhore in it:
    hamiltonian = return_three_level_hamiltonian(Delta, delta)
    obe = pylcp.obe(laserBeams, magField, hamiltonian,
                    transform_into_re_im=True)
    obe.set_initial_rho_from_populations([0, 1, 0])
    obe.evolve_density([0, 2*np.pi*5])

    rhore[...] = np.abs(obe.sol.rho[1, 2, -1])
```

Plot it up:

[6]:
```
fig, ax = plt.subplots(3, 1, figsize=(3.25, 2.75))

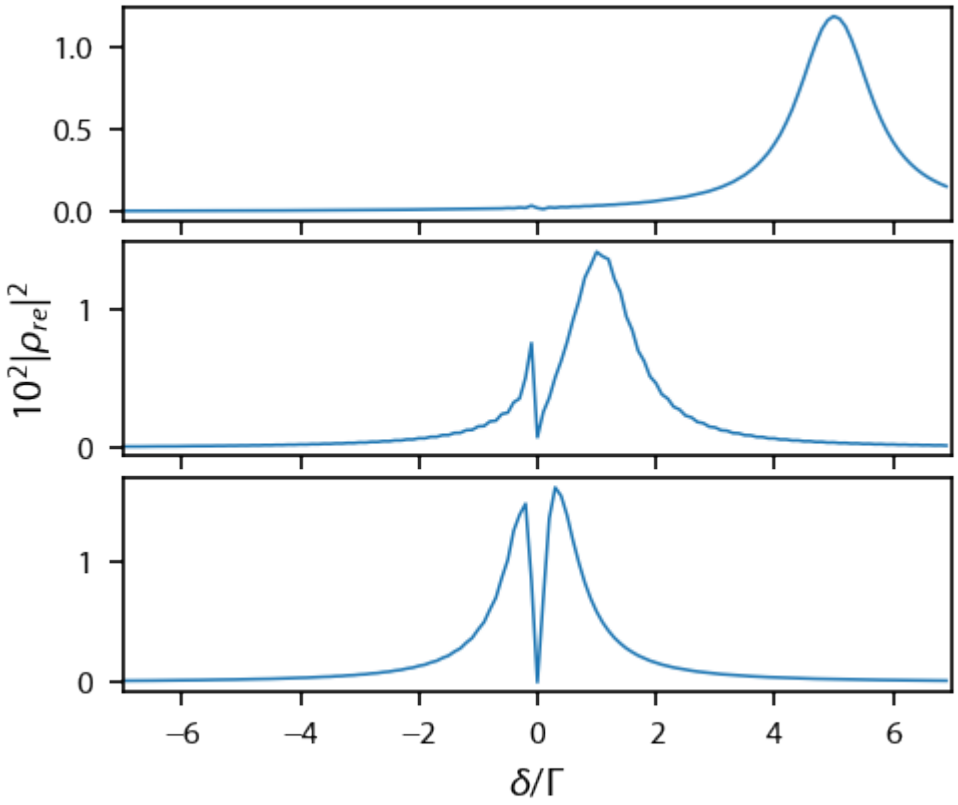
for ii, row in enumerate(it.operands[2].T):
    ax[ii].plot(deltas, 1e2*row**2, linewidth=0.75)
    ax[ii].set_xlim(-7, 7)
```

(continues on next page)

(continued from previous page)

```
    if ii<2:
        ax[ii].xaxis.set_ticklabels('')
ax[1].set_ylabel('$10^2|\\rho_{re}|^2$')
ax[-1].set_xlabel('$\delta/\Gamma$')

fig.subplots_adjust(bottom=0.15, left=0.13)
```



3.1.7 STIRAP

This example calculates the basic STIRAP effect. It demonstrates how to modulate the intensity of a laser with time in pylcp to get an interesting physical effect.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
```

Define the problem

This is the same as in the *last example* in setting up the three state system, except here we add a temporal Gaussian modulation to the intensity of the laser beam. Most STIRAP literature uses Ω rather than I, so remember that $I/I_{\rm sat} = 2\Omega^2/\Gamma^2$.

[2]:
```
# First, define the lasers (functionalized for later):
def return_three_level_lasers(Omegage, Omegare, t0, tsep, twid):
    laserBeams = {}
    laserBeams['g->e'] = pylcp.laserBeams(
        [{'kvec':np.array([1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
          'pol_coord':'spherical', 'delta':0.,
          's':lambda R, t: 2*(Omegage*np.exp(-(t-t0-tsep/2)**2/2/twid**2))**2}],
    )
    laserBeams['r->e'] = pylcp.laserBeams(
        [{'kvec':np.array([1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
          'pol_coord':'spherical', 'delta':0.,
          's':lambda R, t: 2*(Omegare*np.exp(-(t-t0+tsep/2)**2/2/twid**2))**2}],
    )
    return laserBeams

# Second, magnetic field:
magField = lambda R: np.zeros(R.shape)

# Now define the Hamiltonian (functionaized for later):
H0 = np.array([[1.]])
mu_q = np.zeros((3, 1, 1))
d_q = np.zeros((3, 1, 1))
d_q[1, 0, 0,] = 1/np.sqrt(2)

def return_three_level_hamiltonian(Delta, delta):
    hamiltonian = pylcp.hamiltonian()
    hamiltonian.add_H_0_block('g', 0.*H0)
    hamiltonian.add_H_0_block('r', delta*H0)
    hamiltonian.add_H_0_block('e', Delta*H0)
    hamiltonian.add_d_q_block('g','e', d_q)
    hamiltonian.add_d_q_block('r','e', d_q)

    return hamiltonian
```

Evolve the density

Let us evolve with the correct pulse order (address $r \to e$ before $g \to e$) and also the incorrect, opposite order. For simplicity, we choose a saturation parameter of 2 to get a $\Omega = \Gamma$ at the peak.

[3]:
```
hamiltonian = return_three_level_hamiltonian(-3, 0)
laserBeams = return_three_level_lasers(1, 1, 500, -125., 100)

obe = pylcp.obe(laserBeams, magField, hamiltonian)
obe.set_initial_rho_from_populations(np.array([1., 0., 0.]))
sol1 = obe.evolve_density([0, 1000], progress_bar=True)
```

(continues on next page)

(continued from previous page)

```
laserBeams = return_three_level_lasers(1, 1, 500., 125., 100.)

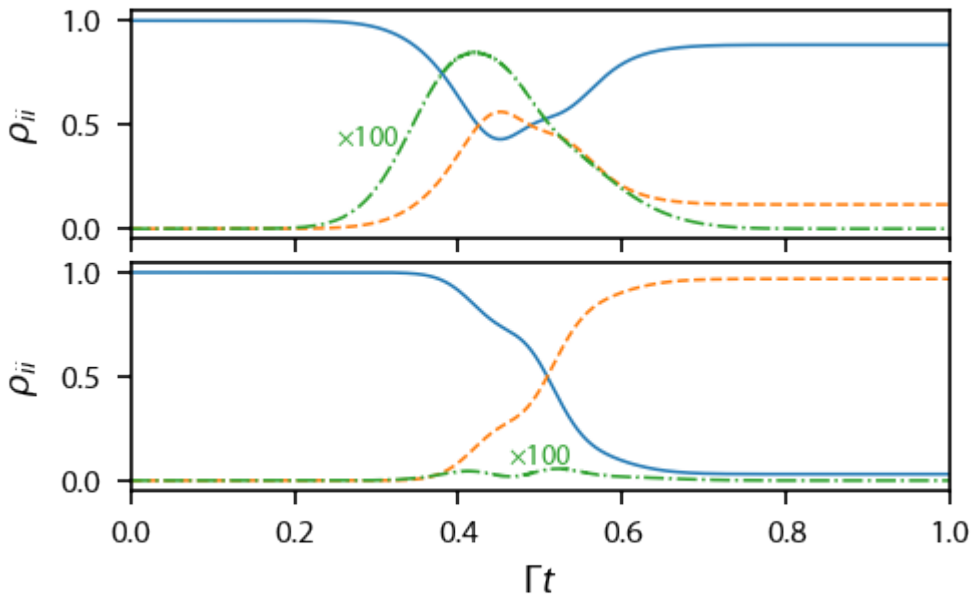
obe = pylcp.obe(laserBeams, magField, hamiltonian)
obe.set_initial_rho_from_populations(np.array([1., 0., 0.]))
sol2 = obe.evolve_density([0, 1000], progress_bar=True)
```

```
Completed in 1.41 s.
Completed in 1.03 s.
```

Plot up the state populations ρ_{gg} (blue), ρ_{rr} (orange), and ρ_{ee} (green) vs. time. We see that STIRAP is only effective in the correct order (solid), and it maintains a minium of population in $|e\rangle >$. The incorrect order (dashed) is nearly completely ineffective at state transfer.

[4]:
```
fig, ax = plt.subplots(2, 1, figsize=(3.25, 2.))
factors = [1, 1, 1e2]
linespecs = ['-', '--', '-.']
for ii, factor in enumerate(factors):
    ax[0].plot(sol1.t/1e3, factor*np.real(sol1.rho[ii, ii]),
               linespecs[ii], color='C%d'%ii, linewidth=0.75)
    ax[1].plot(sol2.t/1e3, factor*np.real(sol2.rho[ii, ii]),
               linespecs[ii], color='C%d'%ii, linewidth=0.75)

[ax[ii].set_ylabel('$\\rho_{ii}$') for ii in range (2)];
[ax[ii].set_xlim((0, 1)) for ii in range(2)];
ax[1].set_xlabel('$\Gamma t$')
ax[0].xaxis.set_ticklabels('')
ax[0].text(0.25, 0.4,'$\\times100$',fontsize=7, color='C2')
ax[1].text(0.46, 0.075,'$\\times100$',fontsize=7, color='C2')
fig.subplots_adjust(bottom=0.18, left=0.14)
```



3.2 Optical Molasses

These examples concern the simplest laser cooling setup: a one-dimensional optical molasses for slowing atoms.

3.2.1 Two-level molasses in 1D

This example covers a two level, 1D optical molasses and compares results to P. D. Lett, et. al., *J. Opt. Soc. Am. B* **6**, 2084 (1989), https://dx.doi.org/10.1364/JOSAB.6.002084. This example is an excellent oppportunity to review the subtlties of extracting accurate temperatures, like integration time and binning, even under the most basic approximations.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
import lmfit
import pathos # used for parallelization
from scipy.stats import iqr
from pylcp.common import progressBar
```

Define the problem

As with every example in `pylcp`, we must first define the Hamiltonian, lasers, and magnetic field. We will make a two-state system that is addressed only by π polarized light. Note that because we are also using the heuristic equation, we want to make sure that the detuning is not on the Hamiltonian, but on the lasers.

[2]:
```
mass = 200

# Make a method to return the lasers:
def return_lasers(delta, s):
    return pylcp.laserBeams([
        {'kvec':np.array([1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
         'pol_coord':'spherical', 'delta':delta, 's':s},
        {'kvec':np.array([-1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
         'pol_coord':'spherical', 'delta':delta, 's':s},
        ], beam_type=pylcp.infinitePlaneWaveBeam)

# Now define a two level Hamiltonian, connected using pi-light.
def return_hamiltonian(delta):
    Hg = np.array([[0.]])
    He = np.array([[-delta]])
    mu_q = np.zeros((3, 1, 1))
    d_q = np.zeros((3, 1, 1))
    d_q[1, 0, 0] = 1.

    return pylcp.hamiltonian(Hg, He, mu_q, mu_q, d_q, mass=mass)

hamiltonian = return_hamiltonian(0.)

magField = lambda R: np.zeros(R.shape)
```

Calculate equilibrium forces

Generate the equilibrium force profile

Do it for all three governing equations at the same step.

[3]:
```
delta = -2.
s = 1.5

laserBeams = return_lasers(delta, s)
hamiltonian = return_hamiltonian(0.)
eqns = {}
eqns['obe'] = pylcp.obe(laserBeams, magField, hamiltonian)
eqns['rateeq'] = pylcp.rateeq(laserBeams, magField, hamiltonian)
eqns['heuristiceq'] = pylcp.heuristiceq(laserBeams, magField)

extra_args = {}
extra_args['obe'] = {'progress_bar':True, 'deltat_tmax':2*np.pi*100, 'deltat_v':4,
                     'itermax':1000, 'rel':1e-4, 'abs':1e-6}
extra_args['rateeq'] = {}
extra_args['heuristiceq'] = {}

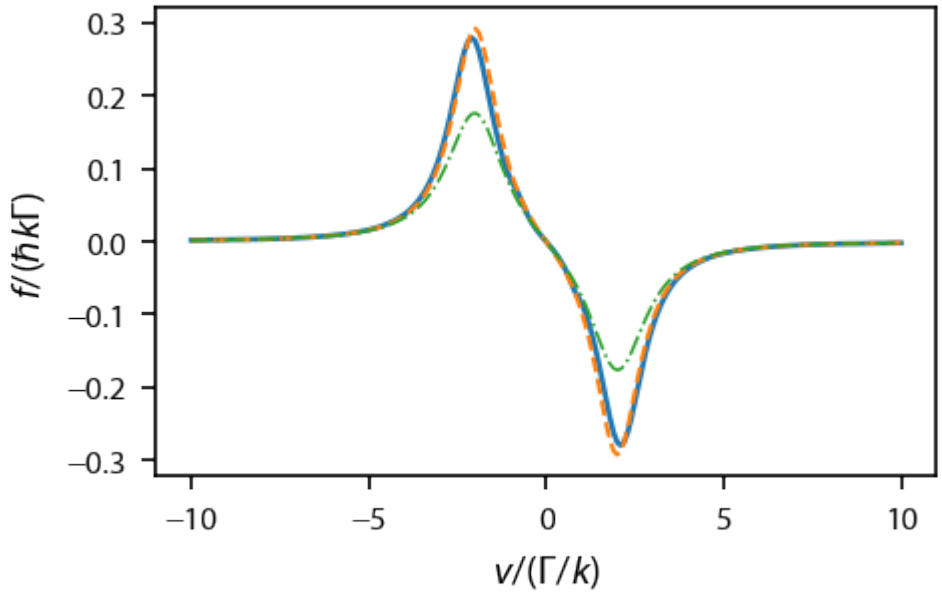
v = np.arange(-10., 10.1, 0.1)

for key in eqns:
    eqns[key].generate_force_profile(np.zeros((3,) + v.shape),
                                     [v, np.zeros(v.shape), np.zeros(v.shape)],
                                     name='molasses', **extra_args[key])
```

```
Completed in 23.30 s.
```

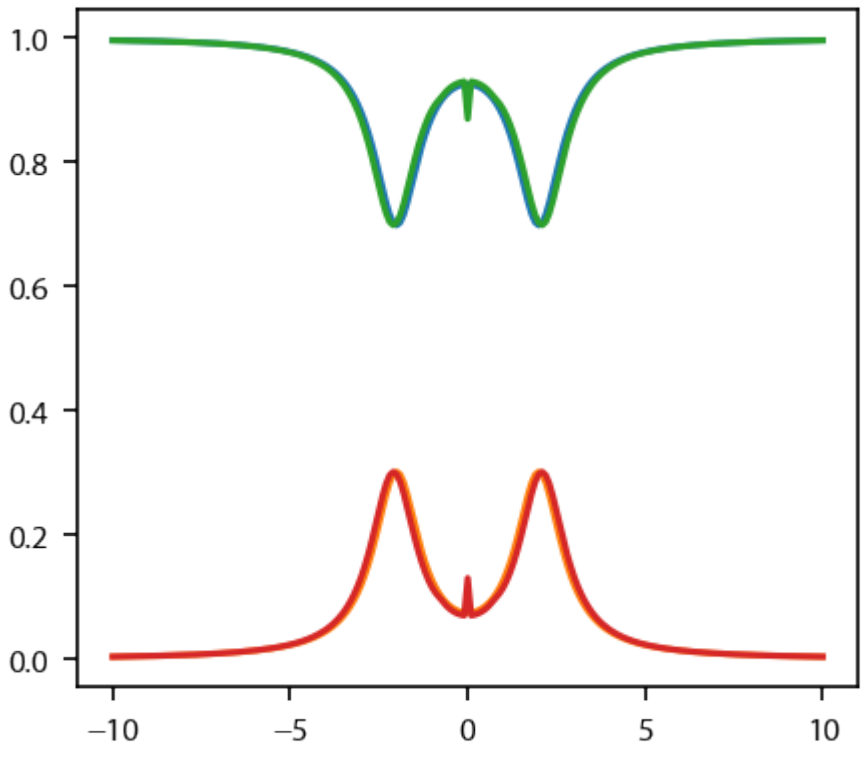
Plot up the equilibrium force profile, solid for the OBEs, dashed for the rate equations, and dashed-dot for the heuristic equation.

[4]:
```
fig, ax = plt.subplots(1, 1, figsize=(3.25, 2.))
lbls = {'obe':'OBE', 'rateeq':'Rate Eq.', 'heuristiceq':'Heuristic Eq.'}
styles = ['-','--','-.']
for ii, key in enumerate(eqns):
    ax.plot(v, eqns[key].profile['molasses'].F[0], styles[ii],
            label=lbls[key], linewidth=1.25-0.25*ii)
    #ax[1].plot(v, eqn.profile['molasses'].Neq)
#ax.legend(fontsize=7)
ax.set_xlabel('$v/(\Gamma/k)$')
ax.set_ylabel('$f/(\hbar k \Gamma)$')
fig.subplots_adjust(bottom=0.2)
```



For the rate equations and the OBEs, also plot up the equilibrium populations of the two states:

[5]:
```
fig, ax = plt.subplots(1, 1)
for key in ['rateeq', 'obe']:
    #ax[0].plot(v, eqn[key].profile['molasses'].F[0])
    ax.plot(v, eqns[key].profile['molasses'].Neq)
```



Calculate the damping parameter

We calculate the damping coefficient β as a function of s_0 and δ, and compare to the Lett expression for the damping.

[6]:
```
deltas = np.linspace(-3, 0., 101)
intensities = np.array([0.01, 0.1, 1, 10])

betas = {}
Deltas, Intensities = np.meshgrid(deltas, intensities)

eqns = {'heuristiceq':pylcp.heuristiceq, 'rateeq':pylcp.rateeq, 'obe':pylcp.obe}

extra_args['obe'] = {'deltat':2*np.pi*100, 'itermax':1000, 'rel':1e-4, 'abs':1e-6}
extra_args['rateeq'] = {}
extra_args['heuristiceq'] = {}

for key in eqns:
    it = np.nditer([Deltas, Intensities, None])
    progress = progressBar()
    for (delta, intensity, beta) in it:
        laserBeams = return_lasers(delta, intensity)
        hamiltonian = return_hamiltonian(0.)

        # Next, generate the OBE or rate equations:
        if key is 'heuristiceq':
            eqn = eqns[key](laserBeams, magField)
        else:
            eqn = eqns[key](laserBeams, magField, hamiltonian)

        # Use built in damping_coefficient() method:
        beta[...] = eqn.damping_coeff(axes=[0], **extra_args[key])

        progress.update((it.iterindex+1)/it.itersize)

    # Just update it to be sure.
    progress.update(1.)

    betas[key] = it.operands[2]
```

```
Completed in 0.42 s.
Completed in 0.94 s.
Completed in 9:26.
```

Plot it up:

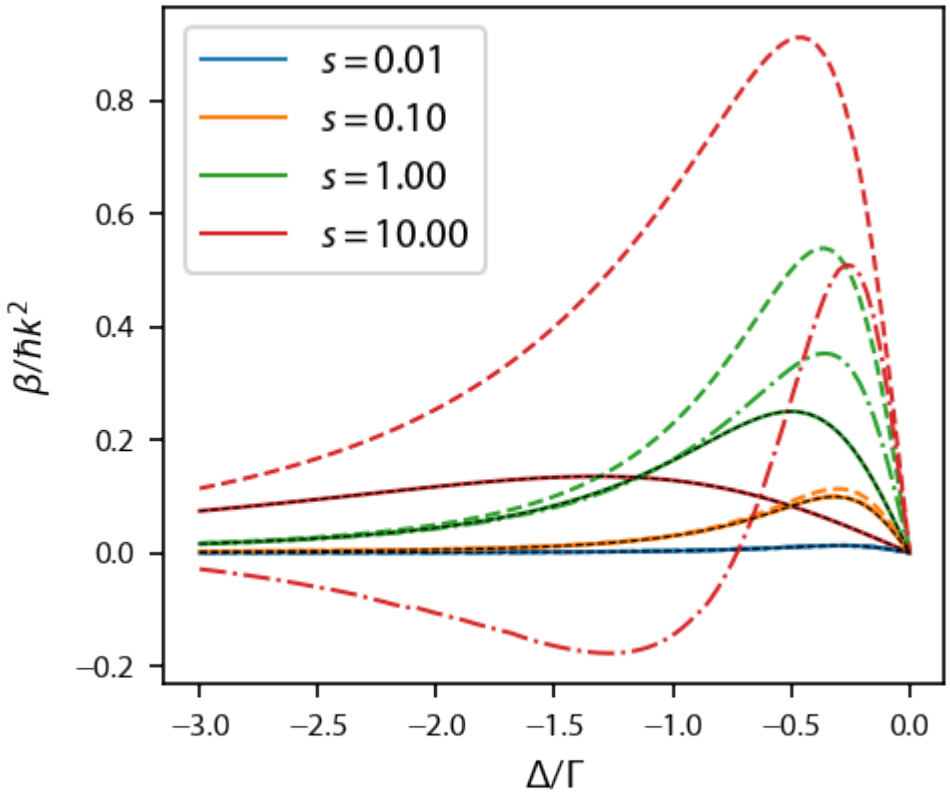
[7]:
```
fig, ax = plt.subplots(1, 1)
for ii, key in enumerate(eqns):
    for jj, betas_i in enumerate(betas[key]):
        if ii==0:
            kwargs = {'label':'$s=%.2f$'%intensities[jj]}
        else:
            kwargs = {}
        ax.plot(deltas, betas_i, styles[ii], color='C%d'%jj, linewidth=1., **kwargs)
```

(continues on next page)

(continued from previous page)

```
for ii, intensity in enumerate(intensities):
    ax.plot(deltas, -4*intensity*2*deltas/(1+2*intensity+4*deltas**2)**2, 'k--',
            linewidth=0.5)

ax.legend(loc='upper left')
ax.set_xlabel('$\Delta/\Gamma$')
ax.set_ylabel('$\\beta/\hbar k^2$');
```



Simulate many atoms to extract temperature

WeâĂŹll run about $N_A \approx 250$ for a time τ to generate some histograms and understand what velocities we obtain, etc.

Because we start at $v = 0$, the amount of time that we will need to integrate for will depend both on the mass and the scattering rate. We must integrate for sufficiently long to allow the atomsâĂŹs velocity to randomly walk to a standard deviation that corresponds to a temperature T. Using the basic Doppler limit $T_D = \hbar\Gamma/(2k_B)$, the expected standard deviation in the velocity at T is given by

$$\sigma_v^2 = \frac{k_B T}{m} = \frac{k_B}{m}\left(\frac{\hbar\Gamma}{2k_B}\right)\frac{T}{T_D} = \frac{1}{2}\frac{\hbar k}{m}\frac{\Gamma}{k}\frac{T}{T_D} = \frac{v_R v_D}{2}\frac{T}{T_D},$$

where the âĂIJDoppler velocityâĂİ is $v_D = \Gamma/k$ and the recoil velocity is $v_R = \hbar k/M$. In our default unit system, $t_0 = 1/\Gamma$, $x_0 = 1/k$, and the dimensionless mass, defined above simply as the variable `mass`, is $\bar{M} = x_0^2 M/\hbar t_0 = \Gamma M/\hbar k^2$. In these units, velocity is measured in v_D, so the corresponding dimensionless standard deviation $\bar{\sigma}_v$ is given by,

$$\bar{\sigma}_v^2 = \frac{\sigma_v^2}{v_D^2} = \frac{1}{2\bar{M}}\frac{T}{T_D}$$

where we have used the fact that the dimensionless mass can be written as $\bar{M} = v_D/v_R$.

Now, assuming that the atomsâĂŹ velocity engages in a random walk, after $N_{\rm sc}$ scattering events, the atomsâĂŹ velocity will have a variance of $\sigma_v^2/v_R^2 = N_{\rm sc}$ (assuming 2 recoils per scattering event and equal probability scattering left or right). Thus, we need at least $N_{\rm sc} = \sigma_v^2/v_R^2 = \bar{\sigma_v}^2(v_D/v_R)^2 = \bar{M}/2(T/T_D)$ scattering events.

To turn this into an integration time, we need a rough estimate of the scattering rate $R_{\rm sc}$. From the heuristic equation,

$$R_{\rm sc} = \frac{\Gamma}{2}\frac{s}{1+2s+4\delta^2}$$

for a given detuning $\delta = \Delta/\Gamma$ and saturation parameter s. Thus, the dimensionless total evolution time $\bar{\tau}$ should be *at least*

$$\bar{\tau} \geq \bar{M}\frac{T}{T_D}\frac{1+2s+4\delta^2}{s}$$

In practice, we might expect the maximum $T/T_D \approx 10$.

Plots up the first ten runs.

```
s = 3
delta = -1

laserBeams = return_lasers(delta, s)
hamiltonian = return_hamiltonian(0.)

eqn = pylcp.heuristiceq(laserBeams, magField, mass=mass)
#eqn = pylcp.rateeq(laserBeams, magField, hamiltonian, include_mag_forces=False)
#eqn = pylcp.obe(laserBeams, magField, hamiltonian)

N_atom = 256
v_final = np.zeros((N_atom,))
#num_of_scatters = np.zeros((N_atom,), dtype='int')
#num_of_steps = np.zeros((N_atom,), dtype='int')

fig, ax = plt.subplots(1, 1)
sols = []
progress = progressBar()
for ii in range(N_atom):
    eqn.set_initial_position_and_velocity(np.array([0., 0., 0.]), np.array([0., 0., 0.]))
    if isinstance(eqn, pylcp.rateeq):
        eqn.set_initial_pop_from_equilibrium()
    elif isinstance(eqn, pylcp.obe):
        eqn.set_initial_rho_from_rateeq()

    eqn.evolve_motion([0., 10*mass*(1+2*s+4*np.abs(delta)**2)/s],
                      random_recoil=True,
                      max_scatter_probability=0.25,
                      freeze_axis=[False, True, True])
    progress.update((ii+1.)/N_atom)

    if ii<10:
        ax.plot(eqn.sol.t, eqn.sol.v[0])

    v_final[ii] = eqn.sol.v[0, -1]
```

(continues on next page)

(continued from previous page)

```
    sols.append(eqn.sol)
    #num_of_scatters[ii] = sum(eqn.sol.n_random)
    #num_of_steps[ii] = len(eqn.sol.t)

ax.set_xlabel('$\Gamma t$')
ax.set_ylabel('$v/(\Gamma/k)$');

eqn.generate_force_profile(np.zeros((3,) + x_fit.shape),
                           [x_fit, np.zeros(x_fit.shape), np.zeros(x_fit.shape)],
                            name='molasses')
```

Adjust chunksize to equal the number of cores.

[8]:
```
s = 3
delta = -3

laserBeams = return_lasers(delta, s)
hamiltonian = return_hamiltonian(0.)

eqn = pylcp.heuristiceq(laserBeams, magField, mass=mass)
#eqn = pylcp.rateeq(laserBeams, magField, hamiltonian, include_mag_forces=False)
#eqn = pylcp.obe(laserBeams, magField, hamiltonian)

eqn.set_initial_position_and_velocity(np.array([0., 0., 0.]), np.array([0., 0., 0.]))
if isinstance(eqn, pylcp.rateeq):
    eqn.set_initial_pop_from_equilibrium()
elif isinstance(eqn, pylcp.obe):
    eqn.set_initial_rho_from_rateeq()

N_atom = 96 # Needs to be divisible by chunksize

if hasattr(eqn, 'sol'):
    del eqn.sol

def generate_random_solution(eqn, tmax, rng_seed):
    # We need to generate random numbers to prevent solutions from being seeded
    # with the same random number.
    eqn.evolve_motion(
        [0., tmax],
        random_recoil=True,
        max_scatter_probability=0.25,
        freeze_axis=[False, True, True],
        rng=np.random.default_rng(rng_seed)
    )

    return eqn.sol.v[0, -1]

chunksize = 4
v_final = []
ss = np.random.SeedSequence(12345) # "It's the same combination as my luggage!"
child_seeds = ss.spawn(N_atom)
```

(continues on next page)

(continued from previous page)

```
progress = progressBar()
for jj in range(int(N_atom/chunksize)):
    with pathos.pools.ProcessPool(nodes=4) as pool:
        v_final += pool.map(
            generate_random_solution,
            chunksize*[eqn],
            chunksize*[10*mass*(1+2*s+4*np.abs(delta)**2)/s],
            child_seeds[jj*chunksize:(jj+1)*chunksize]
        )
    progress.update((jj+1)/int(N_atom/chunksize))

x_fit = np.linspace(-1.1*np.amax(np.abs(v_final)), 1.1*np.amax(np.abs(v_final)), 101)

eqn.generate_force_profile(np.zeros((3,) + x_fit.shape),
                           [x_fit, np.zeros(x_fit.shape), np.zeros(x_fit.shape)],
                            name='molasses');
```

```
Completed in 4:47.
```

This should be zero, but it might not be because of bad seeding of random number generators during parallel execution:

[9]:
```
np.sum(np.diff(np.sort(v_final))==0)
```

[9]:
```
0
```

Bin the final data and extract temperature

To extract the final temperature with uncertainity, we must calculate σ_v and its uncertainty. The standard error in σ for a Gaussian distribution is $\sigma/\sqrt{2N-2}$ for N points. However, we will obtain more accurate estimates of the uncertainty by restricting the Gaussian distribution such that the mean is zero. To do this systematically, we will bin the data and fit the resulting histogram.

We use the Freedman–Diaconis rule to determine the bin size, and use bins that are symmetric about zero and span the whole range of $v_{\rm final}$. We normalize the counts in each bin to N_A, which gives us the âĂIJexperimentalâĂİ (i.e., through the numerics) probability of landing in the bin between $x - dx/2$ and $x + dx/2$. We then fit the numerics to the associated expectation from a normal distribution given by $p(x)dx$, where $p(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/2\sigma^2}$, σ is the standard deviation, and μ is the mean. Because the bin size here is fixed and known, we eliminate one additional variable compared to `lmfit`âĂŹs built in Gaussian model, namely the amplitude.

We also plot the distribution expected from Lett, *et. al.*, eq. (18):

$$\frac{T}{T_D} = \frac{1+2s+4\delta^2}{4\delta^2}$$

When simulating with the heuristic equation, this should basically be exact (as it is derived using the heuristic equation). The only exception is when the mass is low, $\bar{M} \lesssim 100$. In this case, the force vs. velocity curve across the distribution shows curvature, usually with increased damping out near the wings. This increased damping causes lower temperatures that that predicted by Lett, *et. al.*, eq. (26). We can easily see if this non-linearity exists by plotting the force vs. velocity curve across the distribution, shown here in black.

[10]:
```
#print(2*np.std(v_final)**2*mass)
def normaldist(x, mu, sigma, dx):
    # Gaussian probability distribution function
```

(continues on next page)

(continued from previous page)

```
    # probability of landing in a bin of width dx is p(x)dx
    return dx/sigma/np.sqrt(2*np.pi)*np.exp(-(x-mu)**2/2/sigma**2)

def lett_temperature(s, delta):
    """
    Returns the ratio of the expected temperature relative to the "bare" Doppler␣
↪temperature.
    """
    return 0.5*(1+2*s+4*delta**2)/2/np.abs(delta)

def fit_vfinal(v_final, N_atom):
    dx = 2*iqr(v_final)/N_atom**(1/3)
    xb = np.arange(dx/2, 1.1*np.amax(np.abs(v_final)), dx)
    xb = np.concatenate((-xb[::-1], xb))

    x = xb[:-1] + np.diff(xb)/2
    y = np.histogram(v_final, bins=xb)[0]/N_atom #Probability of an atom landing in this␣
↪bin.'

    ok = (y>0)
    weights = np.zeros(ok.shape)
    weights[ok] = 1./np.sqrt(y[ok]/N_atom)
    model = lmfit.Model(normaldist)
    params = model.make_params()
    params['dx'].value = dx # bin width, probability of landing in the bin is p(x) dx
    params['dx'].vary = False
    params['mu'].value = 0.
    params['mu'].vary = False
    params['sigma'].value = np.std(v_final)

    result = model.fit(y[ok], params, x=x[ok], weights=weights[ok])

    return result, x, y, dx

result, x, y, dx = fit_vfinal(v_final, N_atom)

fig, ax = plt.subplots(1, 1)

ax.bar(x, y, width=0.8*dx, yerr=np.sqrt(y/N_atom)) #Poissonian error

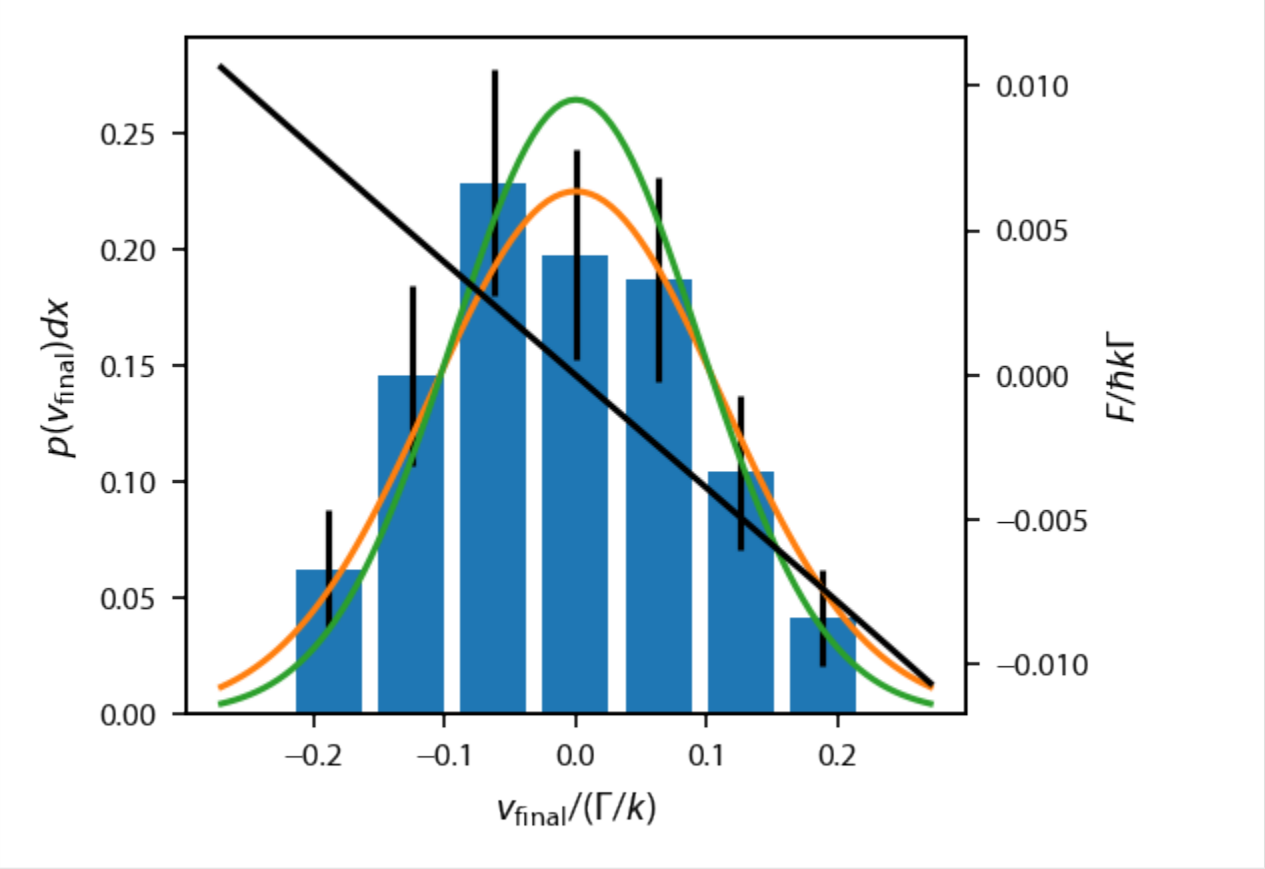
x_fit = np.linspace(-1.1*np.amax(np.abs(v_final)), 1.1*np.amax(np.abs(v_final)), 101)

ax.plot(x_fit, result.eval(x=x_fit))
ax.plot(x_fit, normaldist(x_fit, 0, np.sqrt(lett_temperature(s, delta)/2/mass), dx))

ax_twin = ax.twinx()
ax_twin.plot(x_fit, eqn.profile['molasses'].F[0], 'k-')

ax.set_ylabel('$p(v_{\\rm final}) dx$')
ax.set_xlabel('$v_{\\rm final}/(\Gamma/k)$');

ax_twin.set_ylabel('$F/\hbar k \Gamma$');
```



[11]:
```
result
```

[11]:
```
<lmfit.model.ModelResult at 0x7fd884595050>
```

Measure the temperature vs. detuning and intensity

We can compare to the formula in Lett, *et. al.*

[ ]:
```
deltas = np.array([-3, -2., -1., -0.5, -0.375, -0.25, -0.125])
intensities = np.array([0.3, 1, 3])

Deltas, Intensities = np.meshgrid(deltas, intensities)

N_atom = 256

v_final = {}
result = {}

# Make a progress bar:
progress = progressBar()

it = np.nditer([Deltas, Intensities, None, None])
for (delta, s, sigma, delta_sigma) in it:
    # First, generate the new laser beams and hamiltonian:
```

(continues on next page)

(continued from previous page)

```
    laserBeams = return_lasers(delta, s)
    hamiltonian = return_hamiltonian(0.)

    # Next, generate the OBE, rate equations or heuristic eqn:
    eqn = pylcp.heuristiceq(laserBeams, magField, mass=mass)
    #eqn = pylcp.rateeq(laserBeams, magField, hamiltonian, include_mag_forces=False)

    eqn.set_initial_position_and_velocity(np.array([0., 0., 0.]),
                                          np.array([0., 0., 0.]))
    if isinstance(eqn, pylcp.rateeq):
        eqn.set_initial_pop_from_equilibrium()
    elif isinstance(eqn, pylcp.obe):
        eqn.set_initial_rho_from_rateeq()

    key = (float(delta), float(s))
    v_final[key] = np.zeros((N_atom,))

    # Now, evolve however many times:
    # Non-parallel version
#     for ii in range(N_atom):
#         eqn.evolve_motion([0., 10*mass*(1+2*s+4*np.abs(delta)**2)/s],
#                           random_recoil=True,
#                           max_scatter_probability=0.25,
#                           freeze_axis=[False, True, True])

#         v_final[key][ii] = eqn.sol.v[0, -1]

    # Parallel version
    v_final[key] = []
    ss = np.random.SeedSequence()
    child_seeds = ss.spawn(N_atom)
    for jj in range(int(N_atom/chunksize)):
        with pathos.pools.ProcessPool(nodes=4) as pool:
            v_final[key] += pool.map(
                generate_random_solution,
                chunksize*[eqn],
                chunksize*[10*mass*(1+2*s+4*np.abs(delta)**2)/s],
                child_seeds[jj*chunksize:(jj+1)*chunksize]
            )

    # Now bin and fit, just as above:
    result[key], x, y, dx = fit_vfinal(v_final[key], N_atom)

    sigma[...] = result[key].best_values['sigma']
    delta_sigma[...] = result[key].params['sigma'].stderr

    sigma[...] = result[key].best_values['sigma']
    delta_sigma[...] = result[key].params['sigma'].stderr

    progress.update((it.iterindex+1)/it.itersize)

# Finish updating the progress bar just in case:
```

(continues on next page)

(continued from previous page)

```
progress.update(1.)
```

```
Progress: |âŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹ--------------------| 33.3%; time left: 2:20:08
```

[ ]:
```
deltas_thr = np.linspace(-3, -0.125, 51)
fig, ax = plt.subplots(1, 1)
for ii, (s, sigmas, err) in enumerate(zip(intensities, it.operands[2], it.operands[3])):
    plt.errorbar(deltas, 2*sigmas**2*mass, 4*sigmas*err*mass, fmt='.', color='C%d'%ii)
    plt.plot(deltas_thr, lett_temperature(s, deltas_thr), linewidth=0.75, color='C%d'%ii)
ax.set_xlabel('$\Delta/\Gamma$')
ax.set_ylabel('$T/T_D$')
ax.set_ylim((0.1, 5));
```

[ ]:
```
fig.savefig('20210610T1200_heuristic_eqn_M_200.pdf')
```

[ ]:
```

```

3.2.2 $F = 0 \to F' = 1$ 1D molasses

This example covers calculating the forces in a one-dimensional optical molasses using the optical bloch equations. This example does the boring thing and checks that everything is working on the $F = 0 \to F = 1$ transition, which of course has no sub-Doppler effect. It is a bit more complicated than the two level molasses example, as now different kinds of polarizations and $\hat{k}$ vectors can be used. By exploring multiple combinations, we can verify that the OBEs are working properly.

It first checks the force along the $\hat{z}$-direction. One should look to see that things agree with what one expects whether or not one puts the detuning on the lasers or on the Hamilonian. One should also look at whether the force depends on transforming the OBEs into the real/imaginary components.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
```

Define the problem

We start with defining multiple laser beam polarizations. We store these in a dictionary, keyed by the polarization. In this example, we can also specify the rotating frame, and how the lasers might have a residual oscillation in that frame. The total detuning is the sum of the detuning of the lasers and hamiltonian (see the associated paper). Answers should of course be independent, but computational speed may not be.

[2]:
```
laser_det = 0.
ham_det = -2.
s = 1.25

laserBeams = {}
laserBeams['$\\sigma^+\\sigma^+$'] = pylcp.laserBeams([
    {'kvec':np.array([0., 0., 1.]), 'pol':np.array([0., 0., 1.]),
     'pol_coord':'spherical', 'delta':laser_det, 's':s},
    {'kvec':np.array([0., 0., -1.]), 'pol':np.array([0., 0., 1.]),
     'pol_coord':'spherical', 'delta':laser_det, 's':s},
```

(continues on next page)

(continued from previous page)

```
    ], beam_type=pylcp.infinitePlaneWaveBeam)

laserBeams['$\\sigma^+\\sigma^-$'] = pylcp.laserBeams([
    {'kvec':np.array([0., 0., 1.]), 'pol':np.array([0., 0., 1.]),
     'pol_coord':'spherical', 'delta':laser_det, 's':s},
    {'kvec':np.array([0., 0., -1.]), 'pol':np.array([1., 0., 0.]),
     'pol_coord':'spherical', 'delta':laser_det, 's':s},
    ], beam_type=pylcp.infinitePlaneWaveBeam)

laserBeams['$\\pi_x\\pi_x$'] = pylcp.laserBeams([
    {'kvec':np.array([0., 0., 1.]), 'pol':np.array([1., 0., 0.]),
     'pol_coord':'cartesian', 'delta':laser_det, 's':s},
    {'kvec':np.array([0., 0., -1.]), 'pol':np.array([1., 0., 0.]),
     'pol_coord':'cartesian', 'delta':laser_det, 's':s},
    ], beam_type=pylcp.infinitePlaneWaveBeam)

laserBeams['$\\pi_x\\pi_y$'] = pylcp.laserBeams([
    {'kvec':np.array([0., 0., 1.]), 'pol':np.array([1., 0., 0.]),
     'pol_coord':'cartesian', 'delta':laser_det, 's':s},
    {'kvec':np.array([0., 0., -1.]), 'pol':np.array([0., 1., 0.]),
     'pol_coord':'cartesian', 'delta':laser_det, 's':s},
    ], beam_type=pylcp.infinitePlaneWaveBeam)

laserBeams['$\\sigma^+\\sigma^-$'].total_electric_field_gradient(np.array([0., 0., 0.]),␣
↪0.)
magField = lambda R: np.zeros(R.shape)

# Hamiltonian for F=0->F=1
Hg, Bgq = pylcp.hamiltonians.singleF(F=0, gF=0, muB=1)
He, Beq = pylcp.hamiltonians.singleF(F=1, gF=1, muB=1)
dijq = pylcp.hamiltonians.dqij_two_bare_hyperfine(0, 1)
ham_F0_to_F1 = pylcp.hamiltonian(Hg, He - ham_det*np.eye(3), Bgq, Beq, dijq)
```

Calculate the equilibrium force profile

This checks to see that the rate equations and OBE agree for $F = 0 \to F' = 1$, two-state solution:

[3]:
```
obe={}
rateeq={}

# Define a v axis:
v = np.arange(-5.0, 5.1, 0.125)

for jj, key in enumerate(laserBeams.keys()):
    print('Working on %s:' % key)
    rateeq[key] = pylcp.rateeq(laserBeams[key], magField, ham_F0_to_F1)
    obe[key] = pylcp.obe(laserBeams[key], magField, ham_F0_to_F1,
                         transform_into_re_im=False, include_mag_forces=False)

    # Generate a rateeq model of what's going on:
    rateeq[key].generate_force_profile(
```

(continues on next page)

(continued from previous page)

```
        [np.zeros(v.shape), np.zeros(v.shape), np.zeros(v.shape)],
        [np.zeros(v.shape), np.zeros(v.shape), v],
        name='molasses'
    )

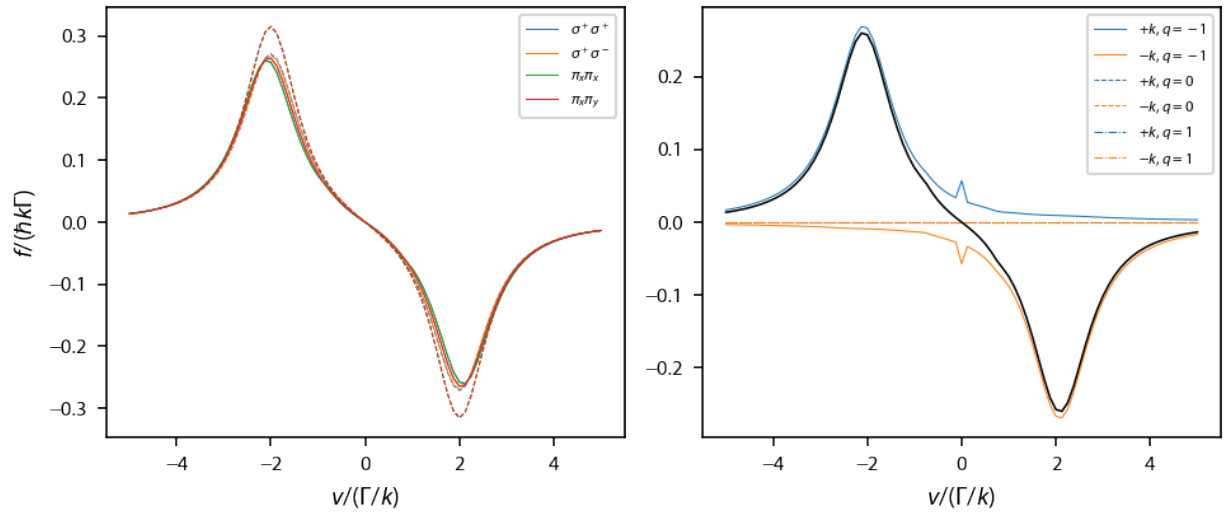
    obe[key].generate_force_profile(
        [np.zeros(v.shape), np.zeros(v.shape), np.zeros(v.shape)],
        [np.zeros(v.shape), np.zeros(v.shape), v],
        name='molasses', deltat_tmax=2*np.pi*100, deltat_v=4, itermax=1000,
        progress_bar=True,
    )
```

```
Working on $\sigma^+\sigma^+$:
Completed in 42.48 s.
Working on $\sigma^+\sigma^-$:
Completed in 50.15 s.
Working on $\pi_x\pi_x$:
Completed in 49.28 s.
Working on $\pi_x\pi_y$:
Completed in 45.20 s.
```

Plot up the results:

[4]:
```
fig, ax = plt.subplots(1, 2, num='Optical Molasses F=0->F1', figsize=(6.5, 2.75))
for jj, key in enumerate(laserBeams.keys()):
    ax[0].plot(obe[key].profile['molasses'].V[2],
               obe[key].profile['molasses'].F[2],
               label=key, linewidth=0.5, color='C%d'%jj)
    ax[0].plot(rateeq[key].profile['molasses'].V[2],
               rateeq[key].profile['molasses'].F[2], '--',
               linewidth=0.5, color='C%d'%jj)
ax[0].legend(fontsize=6)
ax[0].set_xlabel('$v/(\Gamma/k)$')
ax[0].set_ylabel('$f/(\hbar k \Gamma)$')

key = '$\\sigma^+\\sigma^+$'
types = ['-', '--', '-.']
for q in range(3):
    ax[1].plot(v, obe[key].profile['molasses'].fq['g->e'][2, :, q, 0], types[q],
            linewidth=0.5, color='C0', label='$+k$, $q=%d$'%(q-1))
    ax[1].plot(v, obe[key].profile['molasses'].fq['g->e'][2, :, q, 1], types[q],
            linewidth=0.5, color='C1', label='$-k$, $q=%d$'%(q-1))
ax[1].plot(v, obe[key].profile['molasses'].F[2], 'k-',
           linewidth=0.75)
ax[1].legend(fontsize=6)
ax[1].set_xlabel('$v/(\Gamma/k)$')
fig.subplots_adjust(left=0.08, wspace=0.15)
```



Run a simulation at resonance

This allows us to see what the coherences and such are doing.

[5]:
```
v_i=-(ham_det+laser_det)
key = '$\\sigma^+\\sigma^-$'

obe[key] = pylcp.obe(laserBeams[key], magField, ham_F0_to_F1,
                     transform_into_re_im=True)

obe[key].set_initial_position_and_velocity(
    np.array([0., 0., 0.]), np.array([0., 0., v_i])
    )
rho0 = np.zeros((obe[key].hamiltonian.n**2,), dtype='complex128')
rho0[0] = 1.

if v_i==0 or np.abs(2*np.pi*20/v_i)>500:
    t_max = 500
else:
    t_max = 2*np.pi*20/np.abs(v_i)

obe[key].set_initial_rho_from_rateeq()
obe[key].evolve_density(t_span=[0, t_max], t_eval=np.linspace(0, t_max, 1001),)

f, flaser, flaser_q, f_mag = obe[key].force(obe[key].sol.r, obe[key].sol.t,
                                            obe[key].sol.rho, return_details=True)

fig, ax = plt.subplots(2, 2, num='OBE F=0->F1', figsize=(6.25, 5.5))
ax[0, 0].plot(obe[key].sol.t, np.real(obe[key].sol.rho[0, 0]), label='$\\rho_{00}$')
ax[0, 0].plot(obe[key].sol.t, np.real(obe[key].sol.rho[1, 1]), label='$\\rho_{11}$')
ax[0, 0].plot(obe[key].sol.t, np.real(obe[key].sol.rho[2, 2]), label='$\\rho_{22}$')
ax[0, 0].plot(obe[key].sol.t, np.real(obe[key].sol.rho[3, 3]), label='$\\rho_{33}$')
ax[0, 0].legend(fontsize=6)
```

(continues on next page)

(continued from previous page)

```
ax[0, 1].plot(obe[key].sol.t, np.abs(obe[key].sol.rho[0, 1]), label='$|\\rho_{01}|$')
ax[0, 1].plot(obe[key].sol.t, np.abs(obe[key].sol.rho[0, 2]), label='$|\\rho_{02}|$')
ax[0, 1].plot(obe[key].sol.t, np.abs(obe[key].sol.rho[0, 3]), label='$|\\rho_{03}|$')
ax[0, 1].plot(obe[key].sol.t, np.abs(obe[key].sol.rho[1, 3]), label='$|\\rho_{13}|$')
ax[0, 1].legend(fontsize=6)

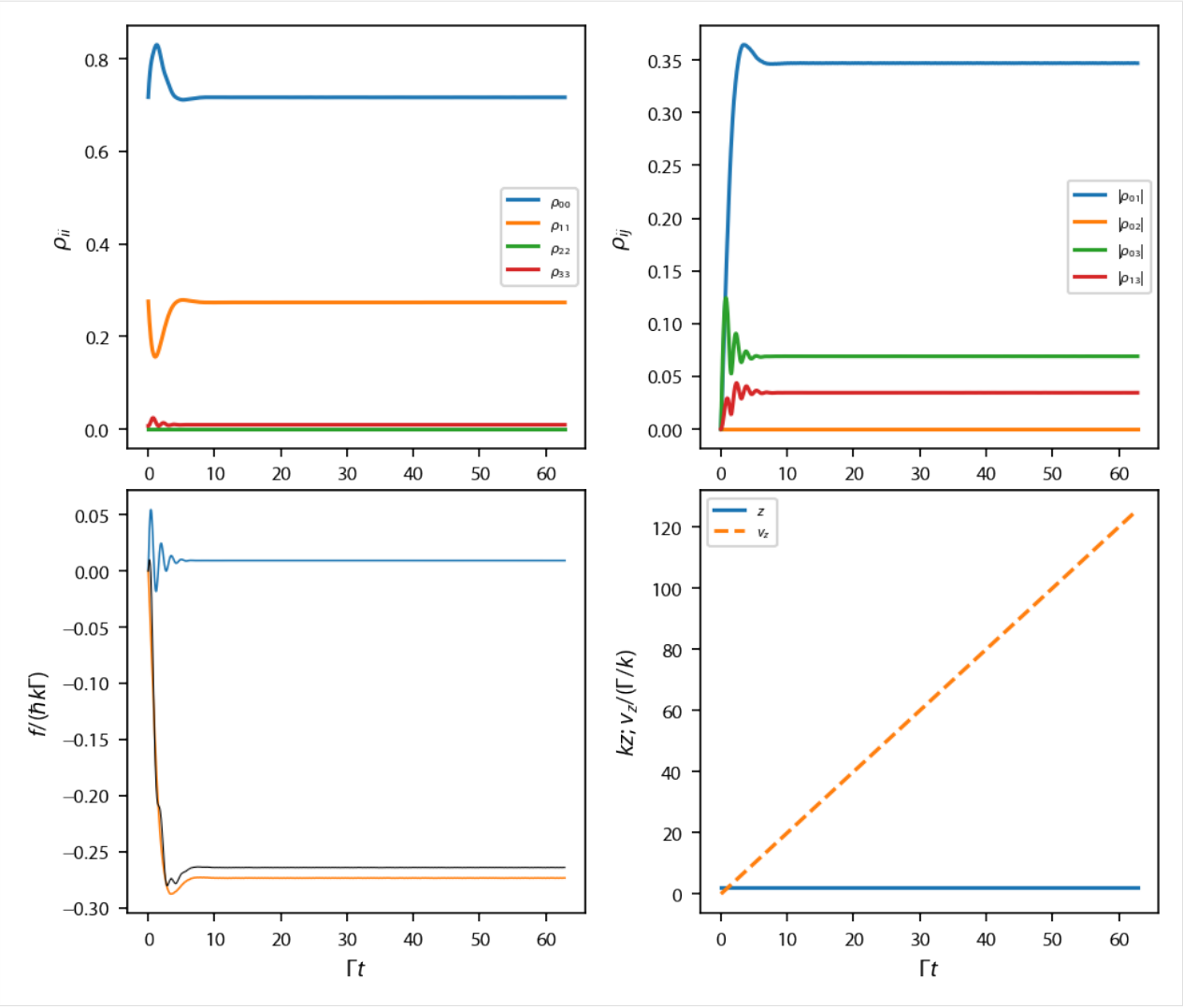
ax[1, 0].plot(obe[key].sol.t, flaser['g->e'][2, 0], '-', linewidth=0.75)
ax[1, 0].plot(obe[key].sol.t, flaser['g->e'][2, 1], '-', linewidth=0.75)
ax[1, 0].plot(obe[key].sol.t, f[2], 'k-', linewidth=0.5)

ax[1, 1].plot(obe[key].sol.t, obe[key].sol.v[-1], '-', label='$z$')
ax[1, 1].plot(obe[key].sol.t, obe[key].sol.r[-1], '--', label='$v_z$')
ax[1, 1].legend(fontsize=6);

ax[0, 0].set_ylabel('$\\rho_{ii}$')
ax[0, 1].set_ylabel('$\\rho_{ij}$')
ax[1, 0].set_ylabel('$f/(\hbar k \Gamma)$')
ax[1, 1].set_ylabel('$kz$; $v_z/(\Gamma/k)$')

ax[1, 0].set_xlabel('$\Gamma t$')
ax[1, 1].set_xlabel('$\Gamma t$')

fig.subplots_adjust(left=0.08, bottom=0.1, wspace=0.25)
```



Finally, calculate f when k is along x and y

This helps to make sure that everything is coded properly.

[6]:
```
laserBeams = {}
laserBeams['x'] = {}
laserBeams['x']['$\\sigma^+\\sigma^+$'] = pylcp.laserBeams([
    {'kvec':np.array([ 1., 0., 0.]), 'pol':+1, 'delta':laser_det, 's':s},
    {'kvec':np.array([-1., 0., 0.]), 'pol':-1, 'delta':laser_det, 's':s},
    ])
laserBeams['x']['$\\sigma^+\\sigma^-$'] = pylcp.laserBeams([
    {'kvec':np.array([ 1., 0., 0.]), 'pol':+1, 'delta':laser_det, 's':s},
    {'kvec':np.array([-1., 0., 0.]), 'pol':+1, 'delta':laser_det, 's':s},
    ])
laserBeams['y'] = {}
laserBeams['y']['$\\sigma^+\\sigma^+$'] = pylcp.laserBeams([
    {'kvec':np.array([0.,  1., 0.]), 'pol':+1, 'delta':laser_det, 's':s},
```

(continues on next page)

(continued from previous page)

```
    {'kvec':np.array([0., -1., 0.]), 'pol':-1, 'delta':laser_det, 's':s},
    ])
laserBeams['y']['$\\sigma^+\\sigma^-$'] = pylcp.laserBeams([
    {'kvec':np.array([0.,  1., 0.]), 'pol':+1, 'delta':laser_det, 's':s},
    {'kvec':np.array([0., -1., 0.]), 'pol':+1, 'delta':laser_det, 's':s},
    ])

obe = {}
rateeq = {}
for coord_key in laserBeams:
    obe[coord_key] = {}
    rateeq[coord_key] = {}
    for pol_key in laserBeams[coord_key]:
        print('Working on %s along %s.' % (pol_key, coord_key))
        rateeq[coord_key][pol_key] = pylcp.rateeq(laserBeams[coord_key][pol_key],
                                                  magField, ham_F0_to_F1)
        obe[coord_key][pol_key] = pylcp.obe(laserBeams[coord_key][pol_key],
                                            magField, ham_F0_to_F1,
                                            transform_into_re_im=False,
                                            include_mag_forces=True)

        if coord_key is 'x':
            V = [v, np.zeros(v.shape), np.zeros(v.shape)]
        elif coord_key is 'y':
            V = [np.zeros(v.shape), v, np.zeros(v.shape)]
        R = np.zeros((3,)+v.shape)
        # Generate a rateeq model of what's going on:
        rateeq[coord_key][pol_key].generate_force_profile(
            R, V, name='molasses'
        )

        obe[coord_key][pol_key].generate_force_profile(
            R, V, name='molasses', deltat_tmax=2*np.pi*100, deltat_v=4,
            itermax=1000, progress_bar=True
        )
```

```
Working on $\sigma^+\sigma^+$ along x.
Completed in 49.64 s.
Working on $\sigma^+\sigma^-$ along x.
Completed in 52.31 s.
Working on $\sigma^+\sigma^+$ along y.
Completed in 57.35 s.
Working on $\sigma^+\sigma^-$ along y.
Completed in 52.62 s.
```

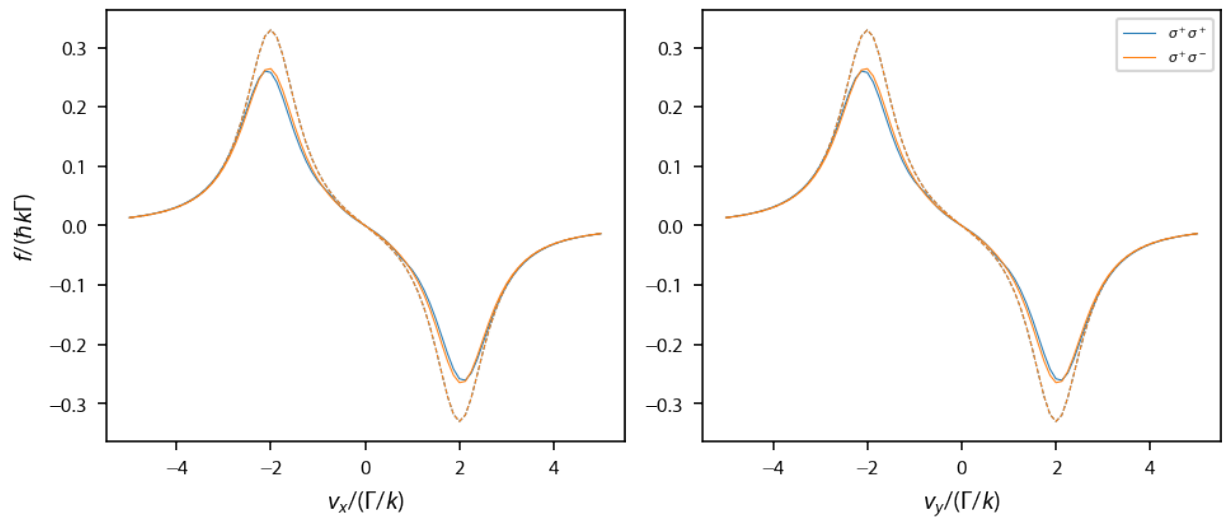
Plot up these results:

[7]:
```
fig, ax = plt.subplots(1, 2, num='Optical Molasses F=0->F1', figsize=(6.5, 2.75))
for ii, coord_key in enumerate(laserBeams.keys()):
    for jj, pol_key in enumerate(laserBeams[coord_key].keys()):
        ax[ii].plot(obe[coord_key][pol_key].profile['molasses'].V[ii],
                    obe[coord_key][pol_key].profile['molasses'].F[ii],
                    label=pol_key, linewidth=0.5, color='C%d'%jj)
```

(continues on next page)

(continued from previous page)

```
        ax[ii].plot(rateeq[coord_key][pol_key].profile['molasses'].V[ii],
                    rateeq[coord_key][pol_key].profile['molasses'].F[ii], '--',
                    linewidth=0.5, color='C%d'%jj)
ax[1].legend(fontsize=6)
ax[0].set_xlabel('$v_x/(\Gamma/k)$')
ax[1].set_xlabel('$v_y/(\Gamma/k)$')
ax[0].set_ylabel('$f/(\hbar k \Gamma)$')
fig.subplots_adjust(left=0.08, wspace=0.15)
```



3.2.3 $F = 2 \rightarrow F' = 3$ 1D molasses

This example covers calculating the forces in a one-dimensional optical molasses using the optical bloch equations. It attempts to reproduce several figures from Ungar, P. J., Weiss, D. S., Riis, E., & Chu, S. âĂIJOptical molasses and multilevel atoms: theory.âĂİ *Journal of the Optical Society of America B*, **6** 2058 (1989). http://doi.org/10.1364/JOSAB.6.002058

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
import pylcp
import scipy.constants as cts
from pylcp.common import progressBar
import lmfit
```

Define the problem

Unager, *et. al.* focus on sodium, so we first find the mass parameter for ^{23}Na:

[2]:
```
atom = pylcp.atom('23Na')
mass = (atom.state[2].gamma*atom.mass)/(cts.hbar*(100*2*np.pi*atom.transition[1].k)**2)
print(mass)
```

```
195.87538720801354
```

As with all other examples, we need to define the Hamiltonian, lasers, and magnetic field. Here, we will write methods to return the Hamiltonian and the lasers in order to sweep their parameters. Note that the magnetic field is always zero, so we define it as such. Lastly, we make a dictionary of different polarizations that we will explore in this example.

[3]:
```
def return_hamiltonian(Fl, Delta):
    Hg, Bgq = pylcp.hamiltonians.singleF(F=Fl, gF=0, muB=1)
    He, Beq = pylcp.hamiltonians.singleF(F=Fl+1, gF=1/(Fl+1), muB=1)
    dijq = pylcp.hamiltonians.dqij_two_bare_hyperfine(Fl, (Fl+1))
    hamiltonian = pylcp.hamiltonian(Hg, -Delta*np.eye(He.shape[0])+He, Bgq, Beq, dijq,␣
↪mass=mass)

    return hamiltonian

# Now, make 1D laser beams:
def return_lasers(delta, s, pol):
    if pol[0][2]>0 or pol[0][1]>0:
        pol_coord = 'spherical'
    else:
        pol_coord = 'cartesian'

    return pylcp.laserBeams([
        {'kvec':np.array([0., 0., 1.]), 'pol':pol[0],
         'pol_coord':pol_coord, 'delta':delta, 's':s},
        {'kvec':np.array([0., 0., -1.]), 'pol':pol[1],
         'pol_coord':pol_coord, 'delta':delta, 's':s},
        ], beam_type=pylcp.infinitePlaneWaveBeam)

magField = pylcp.constantMagneticField(np.array([0., 0., 0.]))

# Now make a bunch of polarization keys:
pols = {'$\\sigma^+\\sigma^-$':[np.array([0., 0., 1.]), np.array([1., 0., 0.])],
        '$\\sigma^+\\sigma^+$':[np.array([0., 0., 1.]), np.array([0., 0., 1.])]}

phi = [0, np.pi/4, np.pi/2]
phi_keys = ['$\phi=0$', '$\phi=\pi/4$', '$\phi=\pi/2$']
for phi_i, key_beam in zip(phi, phi_keys):
        pols[key_beam] = [np.array([1., 0., 0.]), np.array([np.cos(phi_i), np.sin(phi_i),
↪ 0.])]
```

Make a basic force profile

This is not contained in Ungar, *et. al.*, but it makes a nice figure that contains most of the essential elements thereof. We start by creating, using our functions defined above, a set of lasers, the Hamiltonian, and then the set of optical Bloch equations for each polarization specified in the dictionary defined above.

[4]:
```
det = -2.5
s = 1.0

hamiltonian = return_hamiltonian(2, det)

v = np.concatenate((np.arange(0.001, 0.01, 0.001),
                    np.arange(0.01, 0.02, 0.002),
                    np.arange(0.02, 0.03, 0.005),
                    np.arange(0.03, 0.1, 0.01),
                    np.arange(0.1, 5.1, 0.1)))

obe = {}
for key_beam in pols:
    laserBeams = return_lasers(0., s, pol=pols[key_beam])

    obe[key_beam] = pylcp.obe(
        laserBeams, magField, hamiltonian,
        include_mag_forces=False, transform_into_re_im=True
    )

    obe[key_beam].generate_force_profile(
        [np.zeros(v.shape), np.zeros(v.shape), np.zeros(v.shape)],
        [np.zeros(v.shape), np.zeros(v.shape), v],
        name='molasses', deltat_v=4, deltat_tmax=2*np.pi*5000, itermax=1000,
        rel=1e-8, abs=1e-10, progress_bar=True
    )
```

```
Completed in 7:11.
Completed in 3:25.
Completed in 7:19.
Completed in 7:49.
Completed in 7:43.
```

Plot it up:

[5]:
```
fig, ax = plt.subplots(1, 1)
axins = inset_axes(ax, width=1.0, height=0.8)
for key in obe:
    if 'phi' in key:
        linestyle='--'
    else:
        linestyle='-'

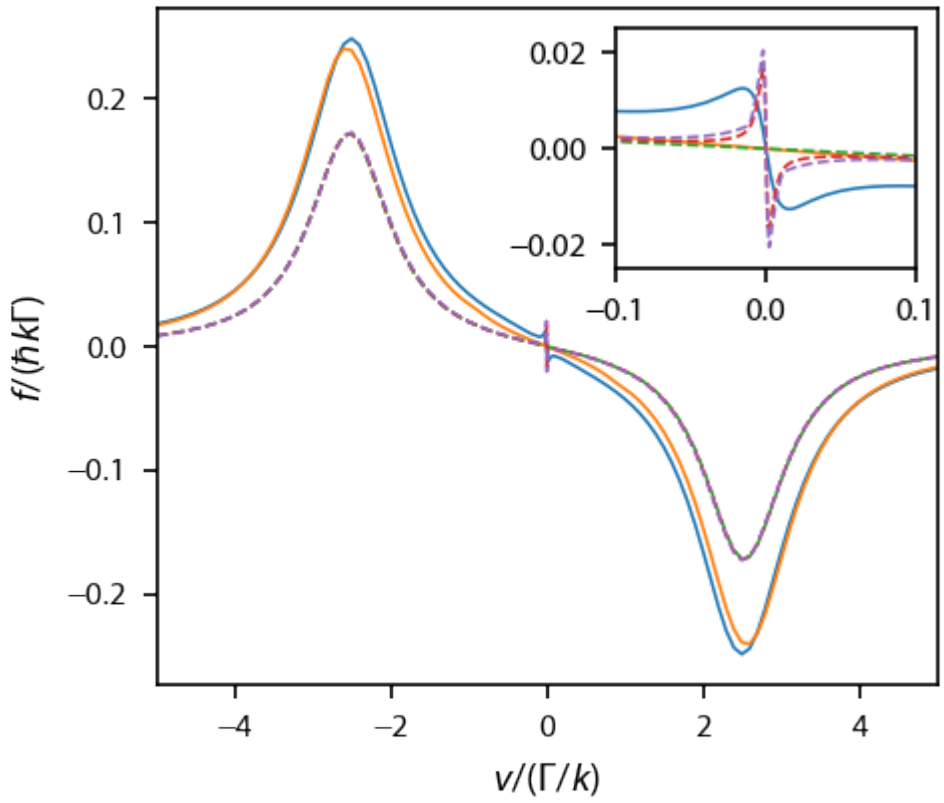
    ax.plot(np.concatenate((-v[::-1], v)),
            np.concatenate((-obe[key].profile['molasses'].F[2][::-1],
                            obe[key].profile['molasses'].F[2])),
            label=pols[key], linestyle=linestyle,
            linewidth=0.75
```

(continues on next page)

(continued from previous page)

```
            )
    axins.plot(np.concatenate((-v[::-1], v)),
               np.concatenate((-obe[key].profile['molasses'].F[2][::-1],
                               obe[key].profile['molasses'].F[2])),
            label=pols[key], linestyle=linestyle,
            linewidth=0.75
            )

ax.set_xlim(-5, 5)
ax.set_xlabel('$v/(\Gamma/k)$')
ax.set_ylabel('$f/(\hbar k \Gamma)$')
axins.set_xlim(-0.1, 0.1)
axins.set_ylim(-0.025, 0.025);
```



Reproduce various figures

Figure 6

This figure compares the force for various polarizations. To make it, we first make each OBE individually, and save the resulting forces. Note that in Ungar, *et. al.*, they define their velocity scale through $mv_0^2 = \hbar\Gamma$, resulting in forces in terms of $Mv_0\Gamma$. Compared to us, they measure their velocities in terms of

$$\frac{v_0}{k/\Gamma} = \sqrt{\frac{\hbar\Gamma}{m}}\frac{k}{\Gamma} = \sqrt{\frac{\hbar k^2}{\Gamma m}} = \sqrt{\bar{m}}$$

Likewise, the force ratio,

$$\frac{Mv_0\Gamma}{\hbar k\Gamma} = \frac{M\sqrt{\hbar\Gamma}}{\hbar k} = \sqrt{\frac{M\Gamma}{\hbar k^2}} = \sqrt{\bar{m}}$$

[7]:
```
det = -2.73
s = 1.25

v = np.concatenate((np.array([0.0]), np.logspace(-2, np.log10(4), 20)))/np.sqrt(mass)

keys_of_interest = ['$\\sigma^+\\sigma^+$', '$\\sigma^+\\sigma^-$',
                    '$\phi=0$', '$\phi=\pi/2$']

F = {}
for key in keys_of_interest:
    laserBeams = return_lasers(0., s, pol=pols[key])
    hamiltonian = return_hamiltonian(2, det)

    obe = pylcp.obe(laserBeams, magField, hamiltonian,
                    transform_into_re_im=True)

    obe.generate_force_profile(
        [np.zeros(v.shape), np.zeros(v.shape), np.zeros(v.shape)],
        [np.zeros(v.shape), np.zeros(v.shape), v],
        name='molasses', deltat_tmax=2*np.pi*1000, deltat_v=4, itermax=10,
        progress_bar=True
    )

    F[key] = obe.profile['molasses'].F[2]
```

```
Completed in 6:46.
Completed in 3:59.
Completed in 7:42.
Completed in 8:05.
```

Plot it up.

[8]:
```
fig, ax = plt.subplots(1, 1, num="Forces; F=2 to F=3")
for key in keys_of_interest:
    ax.plot(v*np.sqrt(mass), -1e3*F[key]/np.sqrt(mass), label=key)
ax.set_xlabel('$v/\sqrt{\hbar\Gamma/m}$')
ax.set_ylabel('$f/\sqrt{\hbar m \Gamma^3}$')
ax.legend(fontsize=8)
ax.set_xlim((0, 4.))
ax.set_ylim((0, 2.));
```

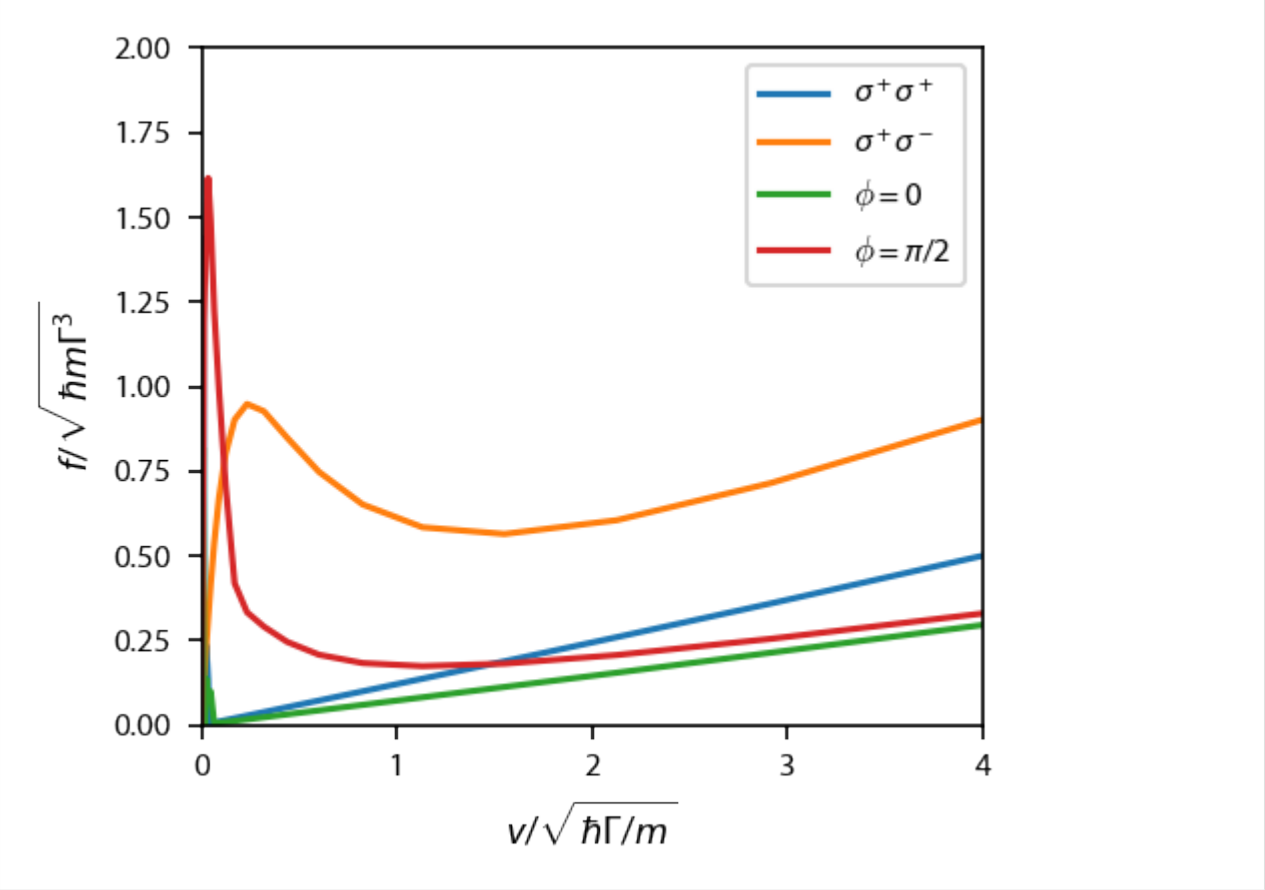


Figure 7

Compare the two-level vs. the corkscrew at large velocities.

[9]:
```
v = np.concatenate((np.arange(0, 1.5, 0.2),
                    np.arange(0.2, 50., 1.5)))/np.sqrt(mass)

s = 1.25
det = -2.73

keys_of_interest = ['$\\sigma^+\\sigma^+$', '$\\sigma^+\\sigma^-$']

f = {}
for key in keys_of_interest:
    laserBeams = return_lasers(0., s, pol=pols[key])
    hamiltonian = return_hamiltonian(2, det)

    obe = pylcp.obe(laserBeams, magField, hamiltonian,
                    transform_into_re_im=True)

    obe.generate_force_profile(
        [np.zeros(v.shape), np.zeros(v.shape), np.zeros(v.shape)],
        [np.zeros(v.shape), np.zeros(v.shape), v],
```

(continues on next page)

(continued from previous page)

```
        name='molasses', deltat_tmax=2*np.pi*100, deltat_v=4, itermax=1000,
        progress_bar=True
    )

    f[key] = obe.profile['molasses'].F[2]
```

```
Completed in 5:15.
Completed in 1:24.
```

Plot it up

[10]:
```
fig, ax = plt.subplots(1, 1)
for key in keys_of_interest:
    ax.plot(v*np.sqrt(mass), -1e3*f[key]/np.sqrt(mass), label=key)
ax.set_xlabel('$v/\sqrt{\hbar\Gamma/m}$')
ax.set_ylabel('$F/\sqrt{\hbar m \Gamma^3}$')
ax.set_ylim((0, 25))
ax.set_xlim((0, 50))
ax.legend(fontsize=8);
```

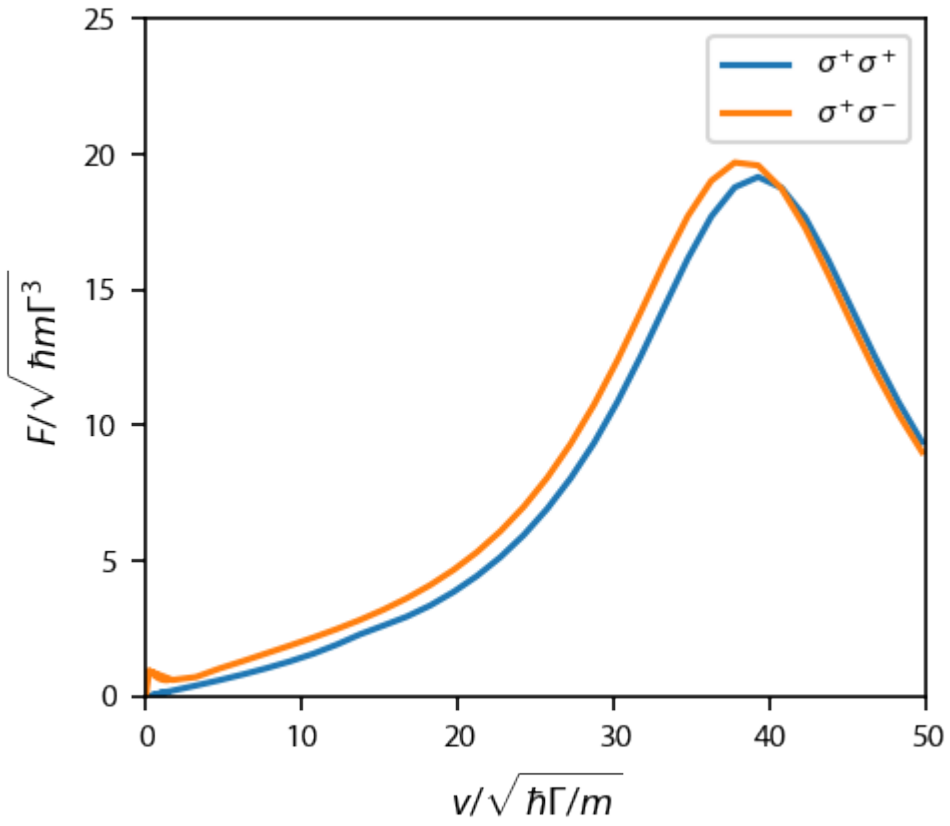


Figure 8

We run the detuning and saturation parameter for the corkscrew model.

[11]:
```
dets = [-1.0, -1.37, -2.73]
intensities = [0.5, 1.25, 2.5]
key = '$\\sigma^+\\sigma^-$'

v = np.arange(0.0, 1.025, 0.025)/np.sqrt(mass)
F_dets = [None]*3
for ii, det in enumerate(dets):
    laserBeams = return_lasers(0., intensities[1], pol=pols[key])
    hamiltonian = return_hamiltonian(2, det)

    obe = pylcp.obe(laserBeams, magField, hamiltonian)

    obe.generate_force_profile(
        [np.zeros(v.shape), np.zeros(v.shape), np.zeros(v.shape)],
        [np.zeros(v.shape), np.zeros(v.shape), v],
        name='molasses', deltat_tmax=2*np.pi*100, deltat_v=4, itermax=1000,
        progress_bar=True
    )

    F_dets[ii] = obe.profile['molasses'].F[2]

F_intensities = [None]*3
for ii, s in enumerate(intensities):
    laserBeams = return_lasers(0., s, pol=pols[key])
    hamiltonian = return_hamiltonian(2, dets[-1])

    obe = pylcp.obe(laserBeams, magField, hamiltonian)

    obe.generate_force_profile(
        [np.zeros(v.shape), np.zeros(v.shape), np.zeros(v.shape)],
        [np.zeros(v.shape), np.zeros(v.shape), v],
        name='molasses', deltat_tmax=2*np.pi*100, deltat_v=4, itermax=1000,
        progress_bar=True
    )

    F_intensities[ii] = obe.profile['molasses'].F[2]
```

```
Completed in 1:03.
Completed in 1:23.
Completed in 4:36.
Completed in 6:47.
Completed in 4:05.
Completed in 2:15.
```

Plot it up:

[12]:
```
fig, ax = plt.subplots(2, 1, figsize=(3.25, 1.5*2.75))
for (F, det) in zip(F_dets, dets):
    ax[0].plot(v*np.sqrt(mass), -1e3*F/np.sqrt(mass), label='$\delta = %f' % det)
for (F, s) in zip(F_intensities, intensities):
```

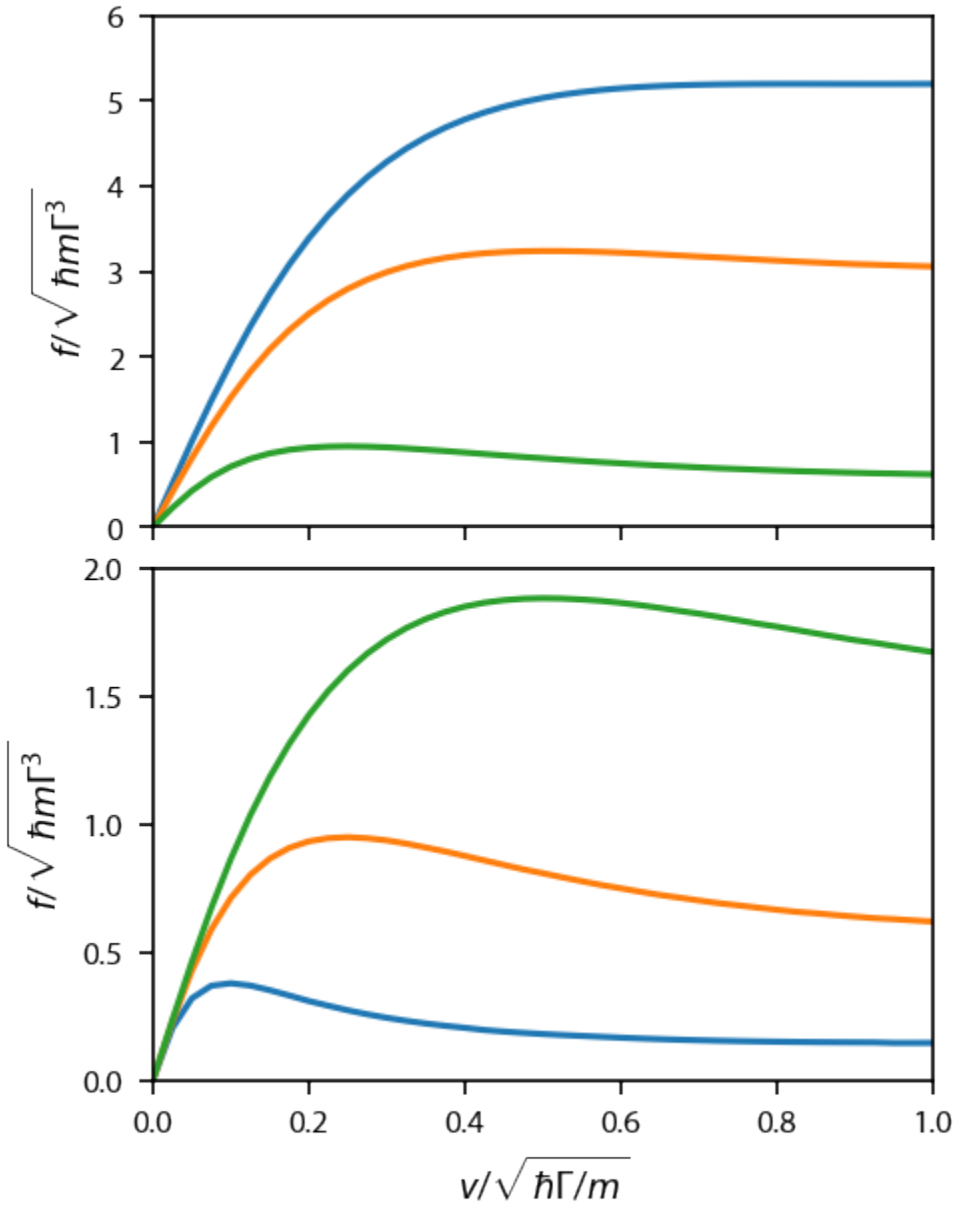
(continues on next page)

(continued from previous page)

```
    ax[1].plot(v*np.sqrt(mass), -1e3*F/np.sqrt(mass), label='$s = %f' % s)
ax[1].set_xlabel('$v/\sqrt{\hbar\Gamma/m}$')
ax[0].set_ylabel('$f/\sqrt{\hbar m \Gamma^3}$')
ax[0].set_ylim((0, 6))
ax[0].set_xlim((0, 1))
ax[1].set_ylabel('$f/\sqrt{\hbar m \Gamma^3}$')
ax[1].set_ylim((0, 2))
ax[1].set_xlim((0, 1))

ax[0].xaxis.set_ticklabels('')

fig.subplots_adjust(bottom=0.12, hspace=0.08)
```



Simulate many atoms and find the temperature:

We start with just a single atom just to make sure everything is OK.

[13]:
```
tmax = 1e4
det = -2.73
s = 1.25
key =  '$\\sigma^+\\sigma^-$'

laserBeams = return_lasers(0., s, pol=pols[key])
hamiltonian = return_hamiltonian(2, det)

obe = pylcp.obe(laserBeams, magField, hamiltonian,
                include_mag_forces=False)

obe.set_initial_position(np.array([0., 0., 0.]))
obe.set_initial_velocity(np.array([0., 0., 0.]))
obe.set_initial_rho_equally()

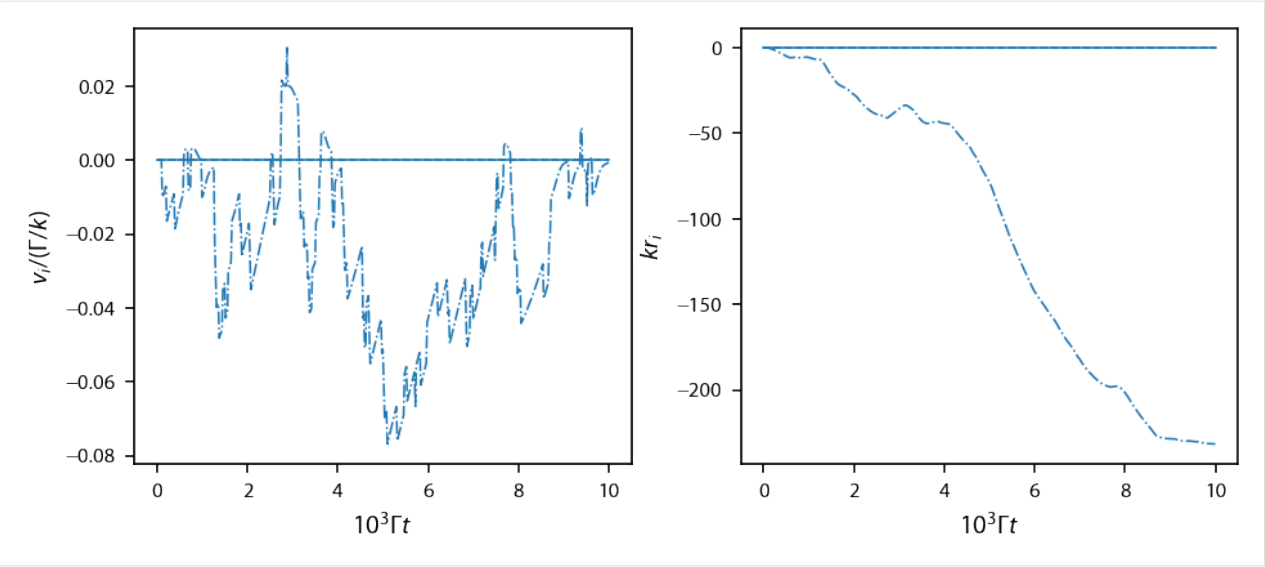
obe.evolve_motion(
    [0, tmax],
    t_eval=np.linspace(0, tmax, 501),
    random_recoil=True,
    progress_bar=True,
    max_scatter_probability=0.5,
    record_force=True,
    freeze_axis=[True, True, False]
);
```

```
Completed in 30.95 s.
```

Plot up the test simulation:

[14]:
```
fig, ax = plt.subplots(1, 2, figsize=(6.5, 2.75))
ll = 0
styles = ['-', '--', '-.']
for jj in range(3):
    ax[0].plot(obe.sol.t/1e3,
               obe.sol.v[jj], styles[jj],
               color='C%d'%ll, linewidth=0.75)
    ax[1].plot(obe.sol.t/1e3,
               obe.sol.r[jj],
               styles[jj], color='C%d'%ll, linewidth=0.75)

#ax[1].set_ylim(-5., 5.)
ax[0].set_ylabel('$v_i/(\Gamma/k)$')
ax[1].set_ylabel('$k r_i$')
ax[0].set_xlabel('$10^{3} \Gamma t$')
ax[1].set_xlabel('$10^{3} \Gamma t$')
fig.subplots_adjust(left=0.1, wspace=0.22)
```



Now run a large sim with 96 atoms:

Non-parallel version:

```
sols = []
for jj in range(Natoms):
    trap.set_initial_position(np.array([0., 0., 100.]))
    trap.set_initial_velocity(0.0*np.random.randn(3))

    trap.evolve_motion([0, 3e2],
                       t_eval=np.linspace(0, 1e2, 1001),
                       random_recoil=True,
                       recoil_velocity=v_R,
                       progress_bar=True,
                       max_scatter_probability=0.5,
                       record_force=True)

    sols.append(copy.copy(trap.sol))
```

Parallel version using `pathos`:

[15]:
```
import pathos
if hasattr(obe, 'sol'):
    del obe.sol

t_eval = np.linspace(0, tmax, 5001)

def generate_random_solution(x, tmax=1e4):
    # We need to generate random numbers to prevent solutions from being seeded
    # with the same random number.
    np.random.rand(256*x)
    obe.set_initial_position(np.array([0., 0., 0.]))
    obe.set_initial_velocity(np.array([0., 0., 0.]))
    obe.set_initial_rho_equally()
    obe.evolve_motion(
```

(continues on next page)

(continued from previous page)

```
        [0, tmax],
        t_eval=t_eval,
        random_recoil=True,
        max_scatter_probability=0.5,
        record_force=True,
        freeze_axis=[True, True, False]
    )

    return obe.sol

Natoms = 96
chunksize = 4
sols = []
progress = progressBar()
for jj in range(int(Natoms/chunksize)):
    with pathos.pools.ProcessPool(nodes=4) as pool:
        sols += pool.map(generate_random_solution, range(chunksize))
    progress.update((jj+1)/int(Natoms/chunksize))
```

```
Completed in 15:36.
```

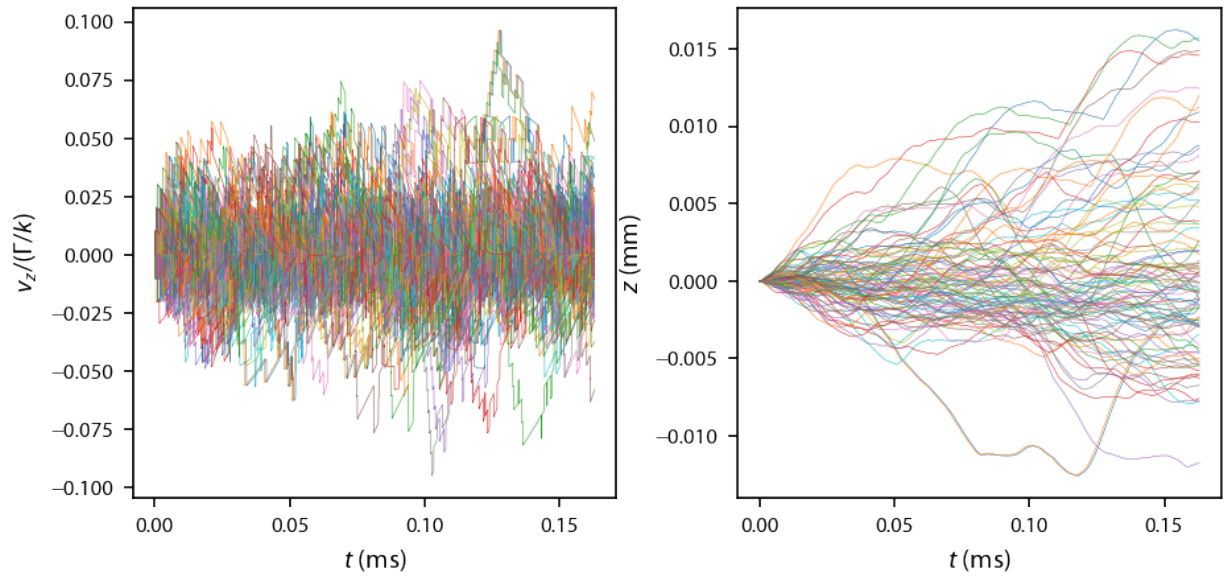
Now, plot all 96 trajectories:

[16]:
```
fig, ax = plt.subplots(1, 2, figsize=(6.25, 2.75))

for sol in sols:
    ax[0].plot(sol.t*atom.state[1].tau*1e3,
               sol.v[2], linewidth=0.25)
    ax[1].plot(sol.t*atom.state[1].tau*1e3,
               sol.r[2]/(2*np.pi*0.1*atom.transition[1].k), linewidth=0.25)

for ax_i in ax:
    ax_i.set_xlabel('$t$ (ms)')
ax[0].set_ylabel('$v_z/(\Gamma/k)$')
ax[1].set_ylabel('$z$ (mm)')

fig.subplots_adjust(left=0.1, bottom=0.08, wspace=0.25)
```



One interesting result from the Unager paper is to look at the average force as a function of position and velocity for the random particles.

[17]:
```
allv = np.concatenate([sol.v.T for sol in sols]).T
allF = np.concatenate([sol.F.T for sol in sols]).T

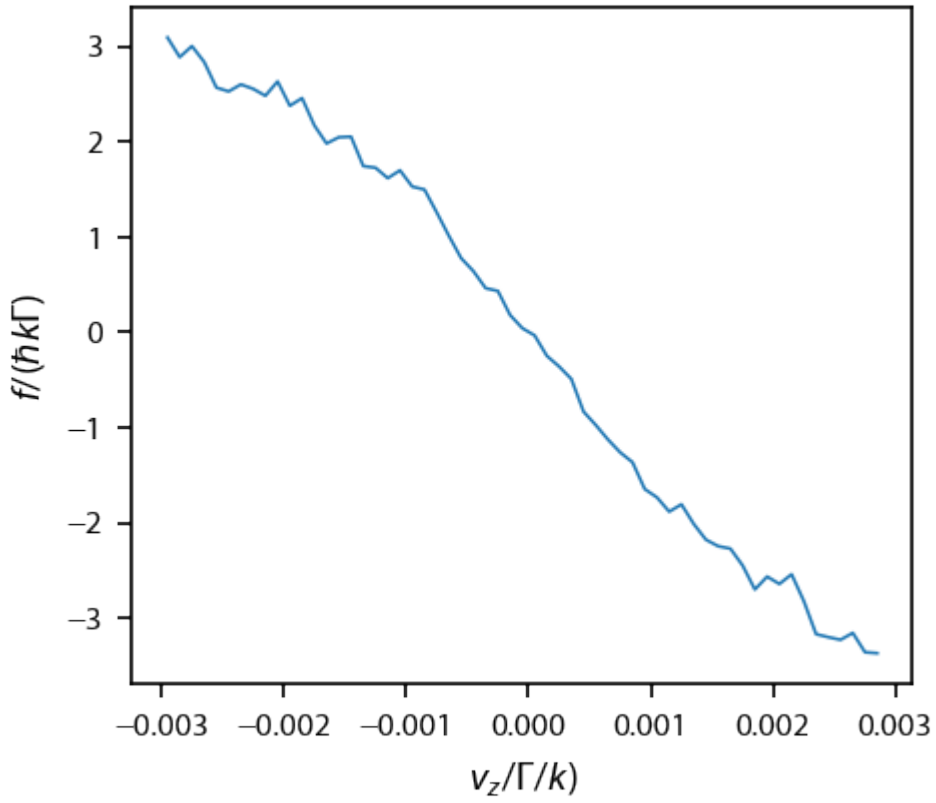
v = np.arange(-.003, 0.003, 0.0001)
vc = v[:-1]+np.mean(np.diff(v))/2

avgFv = np.zeros((3, 3, vc.size))
stdFv = np.zeros((3, 3, vc.size))

for jj in range(3):
    for ii, (v_l, v_r) in enumerate(zip(v[:-1],v[1:])):
        inds = np.bitwise_and(allv[jj] <= v_r, allv[jj] > v_l)
        if np.sum(inds)>0:
            for kk in range(3):
                avgFv[kk, jj, ii] = np.mean(allF[kk, inds])
                stdFv[kk, jj, ii] = np.std(allF[kk, inds])
        else:
            avgFv[:, jj, ii] = np.nan
            avgFv[:, jj, ii] = np.nan

fig, ax = plt.subplots(1, 1)
ax.plot(vc, 1e3*avgFv[2, 2], linewidth=0.75)
ax.set_xlabel('$v_z/\Gamma/k)$')
ax.set_ylabel('$f/(\hbar k \Gamma)$')
```

[17]:
```
Text(0, 0.5, '$f/(\\hbar k \\Gamma)$')
```



Now, letâĂŹs bin the final data and fit it the histogram to extract the temperature.

In this cell, I have included the ability to fit with a two-component Gaussian, but a single component should be sufficient

[29]:
```
vs = np.array([sol.v[2] for sol in sols])

xb = np.arange(-0.1, 0.1, 0.005)
fig, ax = plt.subplots(1, 1)
ax.hist(vs[:, 1000::250].flatten(), bins=xb)
x = xb[:-1] + np.diff(xb)/2
y = np.histogram(vs[:, 1000::250].flatten(), bins=xb)[0]

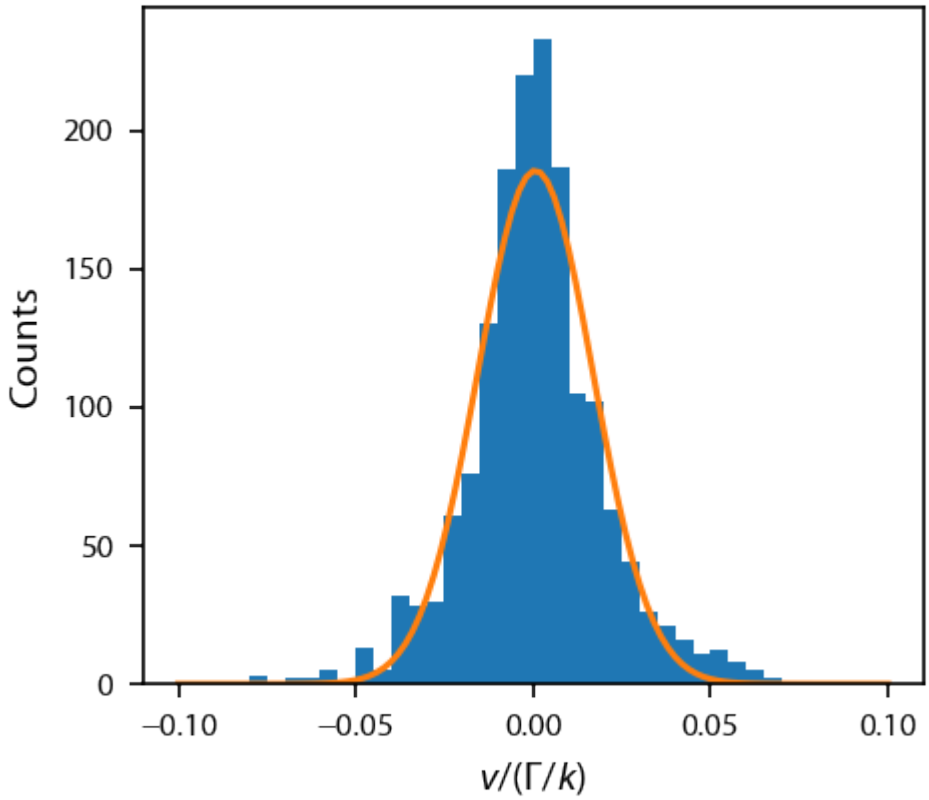
model = lmfit.models.GaussianModel(prefix='A_')# + lmfit.models.GaussianModel(prefix='B_')

params = model.make_params()
params['A_sigma'].value = 0.05
#params['B_sigma'].value = 0.01

ok = y>0
result = model.fit(y[ok], params, x=x[ok], weights=1/np.sqrt(y[ok]))

x_fit = np.linspace(-0.1, 0.1, 101)
ax.plot(x_fit, result.eval(x=x_fit))
ax.set_xlabel('$v/(\Gamma/k)$')
ax.set_ylabel('Counts')
```

[29]:
```
Text(0, 0.5, 'Counts')
```



Print the best fit model results:

[30]:
```
print('Temperature: %.1f uK'%(2*result.best_values['A_sigma']**2*mass*cts.hbar*atom.
↪state[2].gamma/2/cts.k*1e6))
result
```

```
Temperature: 24.2 uK
```

[30]:
```
<lmfit.model.ModelResult at 0x7fda809ef890>
```

3.2.4 $F \to F'$ 1D molasses

This example covers calculating the forces in a one-dimensional optical molasses using the optical bloch equations. It reproduces Fig. 1 of Devlin, J. A. and Tarbutt, M. R. âĂŸThree-dimensional Doppler, polarization-gradient, and magneto-optical forces for atoms and molecules with dark statesâĂŹ, *New Journal of Physics*, **18** 123017 (2016). http://dx.doi.org/10.1088/1367-2630/18/12/123017.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
```

Define the problem

For this particular example, we want to run multiple polarizations and multiple Hamiltonians, but all with the same detuning and intensity. So weâĂŹll make a dictionary of `laserBeams` objects corresponding to different polarizations and a dictionary of `hamiltonian` objects, keyed by the relevant ground and excited states.

[2]:
```
det = -2.5
s = 1.0

laserBeams = {}
laserBeams['$\\sigma^+\\sigma^-$'] = pylcp.laserBeams([
    {'kvec':np.array([0., 0., 1.]), 'pol':np.array([0., 0., 1.]),
     'pol_coord':'spherical', 'delta':0, 's':s},
    {'kvec':np.array([0., 0., -1.]), 'pol':np.array([1., 0., 0.]),
     'pol_coord':'spherical', 'delta':0, 's':s},
    ], beam_type=pylcp.infinitePlaneWaveBeam)

phi = [0, np.pi/8, np.pi/4, 3*np.pi/8, np.pi/2]
phi_keys = ['$\phi=0$', '$\phi=\pi/8$', '$\phi=\pi/4$', '$\phi=3\pi/8$', '$\phi=\pi/2$']
for phi_i, key_beam in zip(phi, phi_keys):
    laserBeams[key_beam] = pylcp.laserBeams([
        {'kvec':np.array([0., 0., 1.]), 'pol':np.array([1., 0., 0.]),
         'pol_coord':'cartesian', 'delta':0, 's':s},
        {'kvec':np.array([0., 0., -1.]),
         'pol':np.array([np.cos(phi_i), np.sin(phi_i), 0.]),
         'pol_coord':'cartesian', 'delta':0, 's':s}
        ], beam_type=pylcp.infinitePlaneWaveBeam)

hamiltonian = {}
for Fg, Fe in zip([1, 1, 2], [2, 1, 1]):
    Hg, Bgq = pylcp.hamiltonians.singleF(F=Fg, gF=0, muB=1)
    He, Beq = pylcp.hamiltonians.singleF(F=Fe, gF=1/Fe, muB=1)
    dijq = pylcp.hamiltonians.dqij_two_bare_hyperfine(Fg, Fe)
    hamiltonian['Fg%d;Fe%d'%(Fg,Fe)] = pylcp.hamiltonian(
        Hg, He - det*np.eye(2*Fe+1), Bgq, Beq, dijq
        )

magField = pylcp.constantMagneticField(np.zeros((3,)))
```

Calculate equilibrium forces

Next, we calculate the equilibrium forces, making a compound dictionary of `obe` objects that is keyed by both the relevant ground and excited states *and* polarizations

[3]:
```
obe = {}
v = np.concatenate((np.arange(0.0, 0.1, 0.001),
                    np.arange(0.1, 5.1, 0.1)))
#v = np.arange(-0.1, 0.1, 0.01)

for key_ham in hamiltonian.keys():
    if key_ham not in obe.keys():
        obe[key_ham] = {}
```

(continues on next page)

(continued from previous page)

```
    for key_beam in laserBeams.keys():
        print('Running %s w/ %s' % (key_ham, key_beam) + '...')
        obe[key_ham][key_beam] = pylcp.obe(laserBeams[key_beam],
                                           magField, hamiltonian[key_ham],
                                           transform_into_re_im=True,
                                           use_sparse_matrices=False)

        obe[key_ham][key_beam].generate_force_profile(
            [np.zeros(v.shape), np.zeros(v.shape), np.zeros(v.shape)],
            [np.zeros(v.shape), np.zeros(v.shape), v],
            name='molasses', deltat_v=4, deltat_tmax=2*np.pi*5000, itermax=1000,
            rel=1e-8, abs=1e-10, progress_bar=True
        )
```

```
Running Fg1;Fe2 w/ $\sigma^+\sigma^-$...
Completed in 8:27.
Running Fg1;Fe2 w/ $\phi=0$...
Completed in 6:34.
Running Fg1;Fe2 w/ $\phi=\pi/8$...
Completed in 7:08.
Running Fg1;Fe2 w/ $\phi=\pi/4$...
Completed in 6:54.
Running Fg1;Fe2 w/ $\phi=3\pi/8$...
Completed in 7:09.
Running Fg1;Fe2 w/ $\phi=\pi/2$...
Completed in 7:42.
Running Fg1;Fe1 w/ $\sigma^+\sigma^-$...
Completed in 4:07.
Running Fg1;Fe1 w/ $\phi=0$...
Completed in 5:23.
Running Fg1;Fe1 w/ $\phi=\pi/8$...
Completed in 7:28.
Running Fg1;Fe1 w/ $\phi=\pi/4$...
Completed in 7:18.
Running Fg1;Fe1 w/ $\phi=3\pi/8$...
Completed in 5:57.
Running Fg1;Fe1 w/ $\phi=\pi/2$...
Completed in 4:48.
Running Fg2;Fe1 w/ $\sigma^+\sigma^-$...
Completed in 6:05.
Running Fg2;Fe1 w/ $\phi=0$...
Completed in 7:22.
Running Fg2;Fe1 w/ $\phi=\pi/8$...
Completed in 9:34.
Running Fg2;Fe1 w/ $\phi=\pi/4$...
Completed in 9:13.
Running Fg2;Fe1 w/ $\phi=3\pi/8$...
Completed in 9:36.
Running Fg2;Fe1 w/ $\phi=\pi/2$...
Completed in 9:59.
```

Plot up the results:

[4]:
```
fig, ax = plt.subplots(3, 2, num='F=1->F=2', figsize=(6.25, 2*2.75))

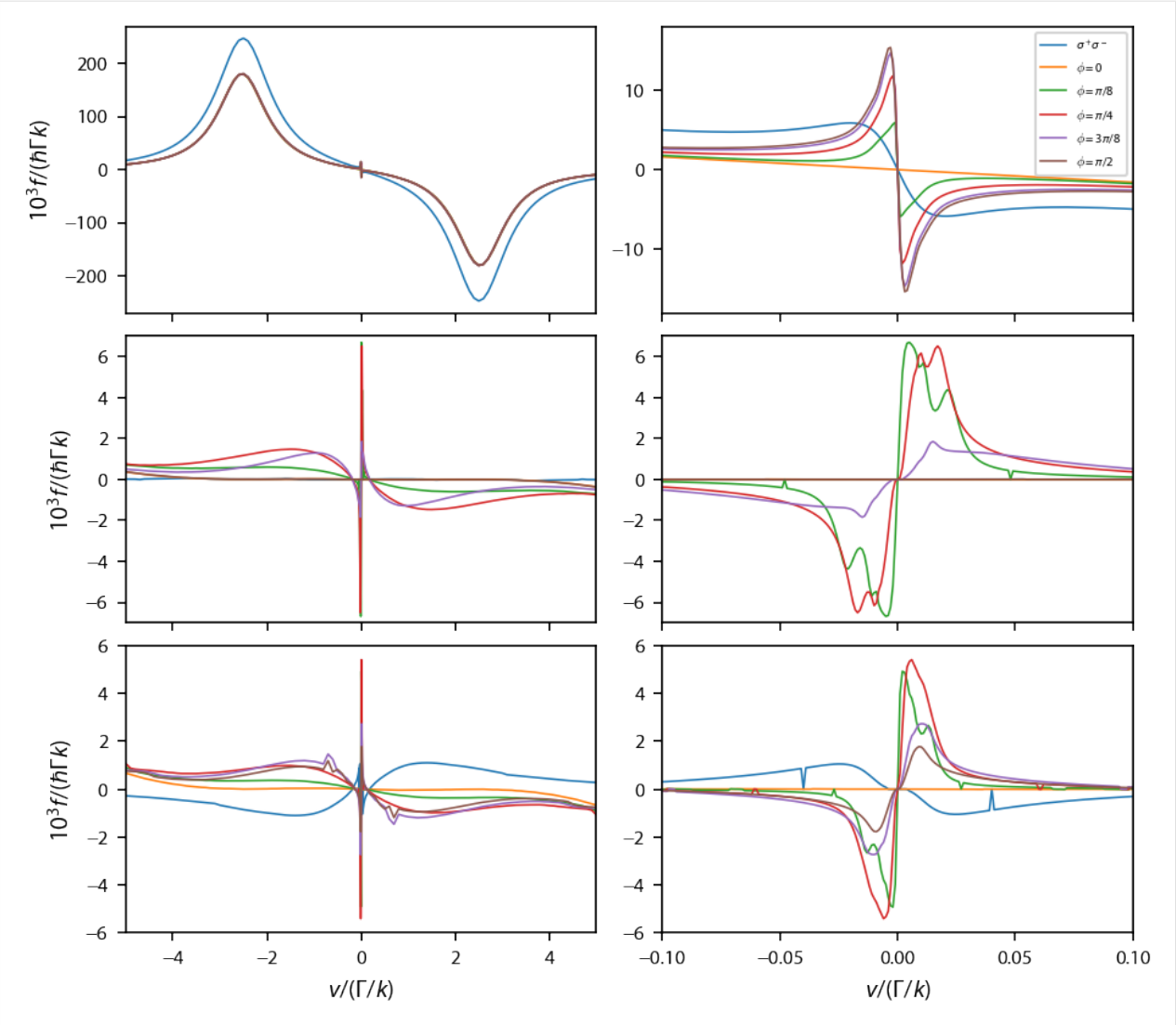
ylims = [[270, 7, 6], [18, 7, 6]]
for ii, key_ham in enumerate(hamiltonian.keys()):
    for key_beam in laserBeams.keys():
        ax[ii, 0].plot(np.concatenate((-v[::-1], v)),
                       1e3*np.concatenate(
                           (-obe[key_ham][key_beam].profile['molasses'].F[2][::-1],
                            obe[key_ham][key_beam].profile['molasses'].F[2])
                           ),
                       label=key_beam, linewidth=0.75)
        ax[ii, 1].plot(np.concatenate((-v[::-1], v)),
                       1e3*np.concatenate(
                           (-obe[key_ham][key_beam].profile['molasses'].F[2][::-1],
                            obe[key_ham][key_beam].profile['molasses'].F[2])
                           ),
                       label=key_beam, linewidth=0.75)

    ax[ii, 0].set_xlim((-5, 5))
    ax[ii, 1].set_xlim((-0.1, 0.1))
    ax[ii, 0].set_ylim((-ylims[0][ii], ylims[0][ii]))
    ax[ii, 1].set_ylim((-ylims[1][ii], ylims[1][ii]))

ax[-1, 0].set_xlabel('$v/(\Gamma/k)$')
ax[-1, 1].set_xlabel('$v/(\Gamma/k)$')
for ii in range(len(hamiltonian)):
    ax[ii, 0].set_ylabel('$10^3 f/(\hbar\Gamma k)$')

for ii in range(len(hamiltonian)-1):
    for jj in range(2):
        ax[ii, jj].set_xticklabels([])

ax[0, 1].legend(fontsize=5)
fig.subplots_adjust(wspace=0.14, hspace=0.08, left=0.1, bottom=0.08)
```



The results agree very well with Devlin, *et. al.*, with the exception of a few glitches in the type-II molasses where the convergence criterion for the equilibrium force clearly failed.

3.3 Magneto-optical traps

These examples focus on understanding the magneto-optical trap.

3.3.1 $F = 0 \to F' = 1$ MOT forces

This example covers calculating the textbook example of forces in a one-dimensional MOT with an $F = 0 \to F' = 1$ atom. We will focus on two governing equations: the `heuristic` equation and the `rateeq`. In this example, weâĂŹll calculate the forces using both and compare. We will also calculate the trapping frequencies and damping coefficients as well.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
from pylcp.atom import atom
import scipy.constants as cts
```

Choose the units

Whatever units we use, letâĂŹs run numbers that are realistic for a common atom, Rb.:

[2]:
```
rb87 = atom('87Rb')

klab = 2*np.pi*rb87.transition[1].k # Lab wavevector (without 2pi) in cm^{-1}
taulab = rb87.state[2].tau  # Lifetime of 6P_{3/2} state (in seconds)
gammalab = 1/taulab
Blab = 15 # About 15 G/cm is a typical gradient for Rb

print(klab, taulab, gammalab/2/np.pi)
```

```
80528.75481555492 2.62348e-08 6066558.277246076
```

As with any problem in `pylcp`, the units that one chooses are arbitrary. We will denote all explicit units with a subscript and all quantities where we have removed the units with an overbar, e.g. $\bar{x} = x/x_0$. Our choice in this script will be different from the default choices of $x_0 = 1/k$ and $t_0 = 1/\Gamma$. LetâĂŹs try a system where lengths are measured in terms of $x_0 = \hbar\Gamma/\mu_B B'$ and $t_0 = kx_0/\Gamma$. This unit system has the advantage that it measures lengths in terms of Zeeman detuning in the trap. Moreover, velocities are measured in Γ/k. Namely,

$$v = \bar{v}\frac{x_0}{t_0} = \bar{v}\frac{\Gamma}{k}$$

From the documentation, the consistent mass scale is

$$\bar{m} = m\frac{x_0^2}{\hbar t_0}$$

```
# Now, here are our `natural' length and time scales:
x0 = cts.hbar*gammalab/(cts.value('Bohr magneton')*1e-4*15) # cm
t0 = klab*x0*taulab # s
mass = 87*cts.value('atomic mass constant')*(x0*1e-2)**2/cts.hbar/t0

# And now our wavevector, decay rate, and magnetic field gradient in these units:
k = klab*x0
gamma = gammalab*t0
alpha = 1.0*gamma     # The magnetic field gradient parameter

print(x0, t0, mass, k, gamma, alpha)
```

Alternatively, we can just use the default unit system:

[3]:
```
# Now, here are our `natural' length and time scales:
x0 = 1/klab  # cm
t0 = taulab  # s

mass = 87*cts.value('atomic mass constant')*(x0*1e-2)**2/cts.hbar/t0

# And now our wavevector, decay rate, and magnetic field gradient in these units:
k = klab*x0
gamma = gammalab*t0
alpha = cts.value('Bohr magneton')*1e-4*15*x0*t0/cts.hbar

print(x0, t0, mass, k, gamma, alpha)
```

```
1.2417924532552691e-05 2.62348e-08 805.2161255642717 1.0 1.0 4.297436175809039e-05
```

It turns out there is another choice of units, useful in the next example, that works extremely well if the radiative force is the *only* force.

Define the problem

One has to define the Hamiltonian, laser beams, and magnetic field.

[4]:
```
# Define the atomic Hamiltonian:
Hg, mugq = pylcp.hamiltonians.singleF(F=0, muB=1)
He, mueq = pylcp.hamiltonians.singleF(F=1, muB=1)

dijq = pylcp.hamiltonians.dqij_two_bare_hyperfine(0, 1)

ham = pylcp.hamiltonian(Hg, He, mugq, mueq, dijq, mass=mass, gamma=gamma, k=k)

det = -4.
s = 1.5

# Define the laser beams:
laserBeams = pylcp.laserBeams(
    [{'kvec':k*np.array([1., 0., 0.]), 's':s, 'pol':-1, 'delta':det*gamma},
     {'kvec':k*np.array([-1., 0., 0.]), 's':s, 'pol':-1, 'delta':det*gamma}],
    beam_type=pylcp.infinitePlaneWaveBeam
)

# Define the magnetic field:
linGrad = pylcp.magField(lambda R: -alpha*R)
```

Define both governing equations

WeâĂŹll add in the OBEs in a later example.

[5]:
```
rateeq = pylcp.rateeq(laserBeams, linGrad, ham, include_mag_forces=True)
heuristiceq = pylcp.heuristiceq(laserBeams, linGrad, gamma=gamma, k=k, mass=mass)
```

Calculate equilibrium forces

LetâĂŹs define both positions and velocities and calculate the forces in the resulting 2D phase space

[6]:
```
x = np.arange(-30, 30, 0.4)/(alpha/gamma)
v = np.arange(-30, 30, 0.4)

X, V = np.meshgrid(x, v)

Rvec = np.array([X, np.zeros(X.shape), np.zeros(X.shape)])
Vvec = np.array([V, np.zeros(V.shape), np.zeros(V.shape)])

rateeq.generate_force_profile(Rvec, Vvec, name='Fx', progress_bar=True)
heuristiceq.generate_force_profile(Rvec, Vvec, name='Fx', progress_bar=True)
```

```
Completed in 16.61 s.
Completed in 4.26 s.
```

[6]:
```
<pylcp.common.base_force_profile at 0x7ff20bb69c90>
```

Now letâĂŹs plot it up, and compare to the heuristic force equation for this geometry, which is given by

$$f = \frac{\hbar k\Gamma}{2}\left(\frac{s}{1+2s+4(\Delta - kv - \mu_B B' x/\hbar)^2/\Gamma^2} - \frac{s}{1+2s+4(\Delta + kv + \mu_B B' x/\hbar)^2/\Gamma^2}\right).$$

where $s = I/I_{\rm sat}$. Note that the quadrupole field parameter defined above $\alpha = \mu_B B'/\hbar$

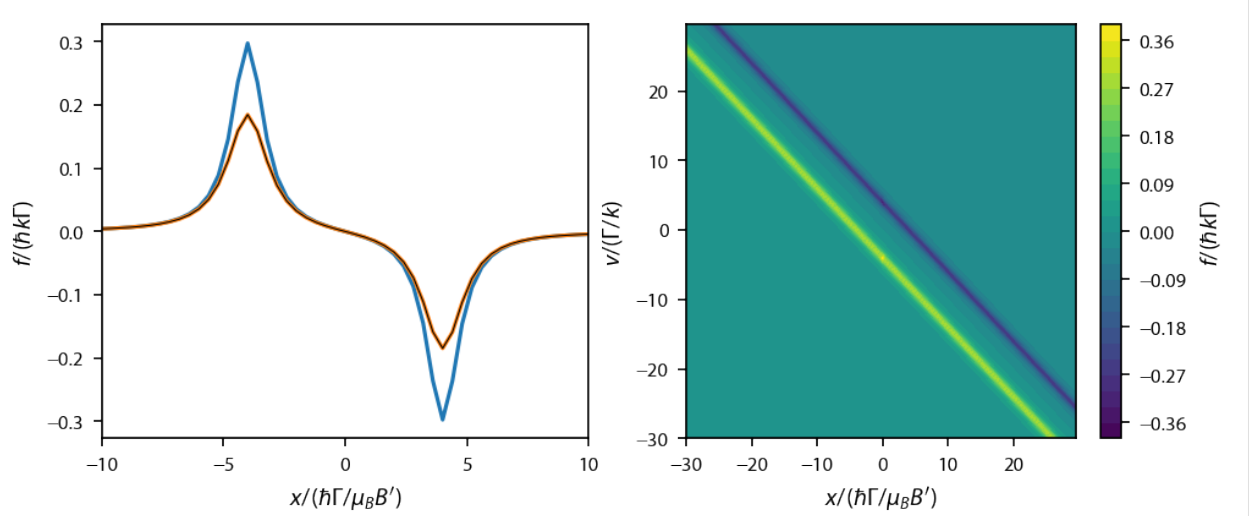
No matter what unit system we use above, we want to measure lengths in terms of the magnetic field. Dividing by the field gradient is enough to accomplish that:

[7]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, num="Expression", figsize=(6.5, 2.75))

ax[0].plot(x*(alpha/gamma), rateeq.profile['Fx'].F[0, int(np.ceil(x.shape[0]/2)), :]/
↪gamma/k)
ax[0].plot(x*(alpha/gamma), heuristiceq.profile['Fx'].F[0, int(np.ceil(x.shape[0]/2)), :
↪]/gamma/k)
ax[0].plot(x*(alpha/gamma), (s/(1+2*s + 4*(det-alpha*x/gamma)**2) - s/(1+2*s +␣
↪4*(det+alpha*x/gamma)**2))/2, 'k-', linewidth=0.5)
ax[0].set_ylabel('$f/(\hbar k \Gamma)$')
ax[0].set_xlabel('$x/(\hbar \Gamma/\mu_B B\')$')
ax[0].set_xlim((-10, 10))

im1 = ax[1].contourf(X*(alpha/gamma), V, rateeq.profile['Fx'].F[0]/gamma/k, 25)
fig.subplots_adjust(left=0.08, wspace=0.2)
cb1 = plt.colorbar(im1)
cb1.set_label('$f/(\hbar k \Gamma)$')
ax[1].set_xlabel('$x/(\hbar \Gamma/\mu_B B\')$')
ax[1].set_ylabel('$v/(\Gamma/k)$')
```

[7]:
```
Text(0, 0.5, '$v/(\\Gamma/k)$')
```



Note that heuristic equation produces less force at resonance in x, because it is over-estimating the total saturation.

Compute the trap frequencies and damping rates

And we can compare to simple 1D theory. LetâĂŹs take the force equation above and expand it about $x = 0$ and $v = 0$. We get

$$f \approx \frac{\hbar k}{2\Gamma}\left(\frac{s\Delta}{1+2s+4\Delta^2/\Gamma^2}\right)\left(\frac{\mu_B B'}{\hbar}x + kv\right)$$

The damping coefficient β, most easily expressed in units of $\hbar k^2$, and is given by

$$\frac{\beta}{\hbar k^2} = \frac{8s\delta}{1+2s+4\delta^2}$$

where $\delta = \Delta/\Gamma$. Note that the trapping frequency is defined through

$$\ddot{x} - \frac{\beta}{m}\dot{x} + \omega^2 x = 0$$

and therefore, its square is most easily measured in units of $k\mu_B B'/m$,

$$\frac{\omega^2}{k\mu_B B'/m} = \frac{8s\delta}{1+2s+4\delta^2}$$

Note as well that we defined α, the quadrupole field parameter above to be $\alpha = \hbar\mu_B B'$.

[8]:
```
dets = np.arange(-5, 0.05, 0.05)
intensities = np.array([0.1, 1, 10])

omega = np.zeros(intensities.shape + dets.shape)
dampingcoeff = np.zeros(intensities.shape + dets.shape)

omega_heuristic = np.zeros(intensities.shape + dets.shape)
dampingcoeff_heuristic = np.zeros(intensities.shape + dets.shape)
```

(continues on next page)

(continued from previous page)

```
for ii, det in enumerate(dets):
    for jj, intensity in enumerate(intensities):
        # Define the laser beams:
        laserBeams = [None]*2
        laserBeams[0] = pylcp.laserBeam(kvec=k*np.array([1., 0., 0.]), s=intensity,
                                        pol=-1, delta=gamma*det)
        laserBeams[1] = pylcp.laserBeam(kvec=k*np.array([-1., 0., 0.]), s=intensity,
                                        pol=-1, delta=gamma*det)

        rateeq = pylcp.rateeq(laserBeams, linGrad, ham, include_mag_forces=False)
        heuristiceq = pylcp.heuristiceq(laserBeams, linGrad, gamma=gamma, mass=mass)

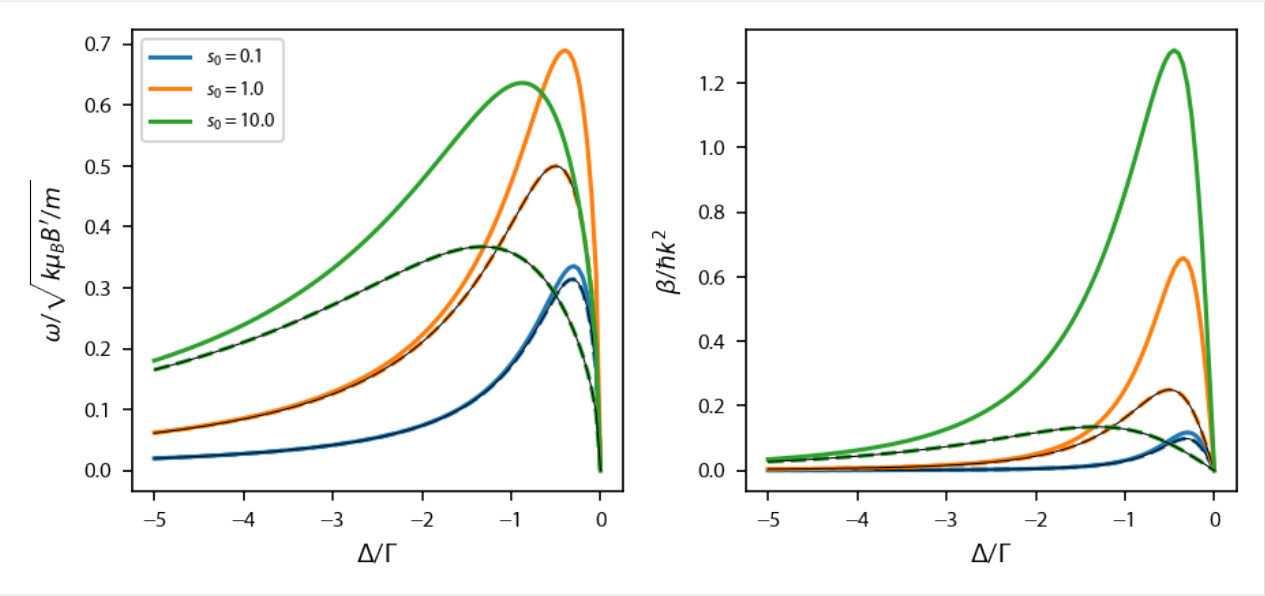
        omega[jj, ii] = rateeq.trapping_frequencies(axes=[0], eps=0.0002)
        dampingcoeff[jj, ii] = rateeq.damping_coeff(axes=[0], eps=0.0002)

        omega_heuristic[jj, ii] = heuristiceq.trapping_frequencies(axes=[0], eps=0.0002)
        dampingcoeff_heuristic[jj, ii] = heuristiceq.damping_coeff(axes=[0], eps=0.0002)

fig, ax = plt.subplots(1, 2, figsize=(6, 2.75))
for ii, intensity in enumerate(intensities):
    ax[0].plot(dets, omega[ii]/np.sqrt(k*alpha/mass), '-', color='C{0:d}'.format(ii),
               label='$s_0 = %.1f$'%intensity)
    ax[0].plot(dets, omega_heuristic[ii]/np.sqrt(k*alpha/mass), '--', color='C{0:d}'.
↪format(ii))
    ax[0].plot(dets, np.sqrt(-8*dets*intensity/(1 + 2*intensity + 4*dets**2)**2), 'k-',␣
↪linewidth=0.5)
    ax[1].plot(dets, dampingcoeff[ii]/k**2, '-', color='C{0:d}'.format(ii))
    ax[1].plot(dets, dampingcoeff_heuristic[ii]/k**2, '--', color='C{0:d}'.format(ii))
    ax[1].plot(dets, -8*dets*intensity/(1 + 2*intensity + 4*dets**2)**2, 'k-',␣
↪linewidth=0.5)

ax[0].legend(fontsize=7, loc='upper left')
ax[1].set_ylabel('$\\beta/\hbar k^2$')
ax[0].set_ylabel('$\\omega/\sqrt{k \mu_B B\'/m}$')
ax[0].set_xlabel('$\Delta/\Gamma$')
ax[1].set_xlabel('$\Delta/\Gamma$')

fig.subplots_adjust(left=0.08, wspace=0.25)
```



3.3.2 $F = 0 \to F' = 1$ MOT capture

This script shows examples about how to solve for the dynamics of a 1D MOT and calculate things like the capture velocity. We will deal specifically with a 1D MOT. We can compare results to those of 1 D. Haubrich, A. HÃűpe, and D. Meschede, âĂIJA simple model for optical capture of atoms in strong magnetic quadrupole fieldsâĂİ *Optics Communications* **102**, 225 (1993). http://dx.doi.org/10.1016/0030-4018(93)90387-K

In this example, we will mostly focus on the heuristic force equation for an $F = 0 \to F' = 1$ atom in a magnetic field:

$$\mathbf{f} = \frac{\hbar \mathbf{k} \Gamma}{2} \sum_{q,i} \frac{s_i (\epsilon'_{q,i})^2}{1 + \sum_j s_j + 4[\Delta - \mathbf{k}_i \cdot \mathbf{v} - q\mu_B B(r)/\hbar]^2/\Gamma^2}$$

where $\mathbf{f}$ is the force, Γ is the decay, $q = -1, 0, 1$, s_i, $\boldsymbol{\epsilon}'_{q,i}$, and $\mathbf{k}_i$ are the intensity, polarization (rotated along the local magnetic field) and wavevector of the ith laser beam, respectively. All laser parameters can can depend on time t and position $\mathbf{r}$. This equation is encoded in `pylcp.heuristiceq`. Of course, one can also switch to the rate equations by loading `pylcp.rateeq`.

WeâĂŹll use the standard 3D MOT quadrupole field,

$$\mathbf{B} = B' \left(-\frac{1}{2}(x\hat{x} + y\hat{y}) + z\hat{z} \right)$$

where B' is the magnetic field gradient.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants as cts
from scipy.optimize import bisect #For root finding
import pylcp
import pylcp.atom as atom
from pylcp.common import progressBar
```

Choose the units:

As with any problem in `pylcp`, the units that one chooses are arbitrary. For this example, we are going to get fancy and use a special unit system that is only possible with the heuristic equation or, when magnetic forces are not included, the rate equations. As in the documentation, we will denote all explicit units with a subscript and all quantities where we have removed the units with an overbar, e.g. $\bar{x} = x/x_0$. LetâĂŹs choose units where the heuristic force is given by (along the $\hat{z}$ axis):

$$\bar{\mathbf{f}} = \frac{\hat{\mathbf{k}}}{2}\sum_{q,i}\frac{s_i(\epsilon'_{q,i})^2}{1+\sum_j s_j + 4(\delta - \hat{\mathbf{k}}_i\cdot\bar{\mathbf{v}} - q\bar{z})^2}$$

where $\delta = \Delta/\Gamma$. This is equivalent the above equation by setting $k/\Gamma = 1$ and $\mu_B B'/(\hbar\Gamma) = 1$. Or, in other words, we want a unit system that measures velocities in terms of Γ/k, positions in terms of $\hbar\Gamma/\mu_B B'$, and forces in terms of $\hbar k\Gamma$.

Programmatically, it allows us to just specify the *unit* vector for $\mathbf{k}$ when we program `laserBeams`, set the magnetic field gradient parameter $\alpha = 1$, and set $\Gamma = 1$ ($\hbar = 1$ by default).

So what are the length and time units of this system? Well, the length unit is given by $x_0 = \hbar\Gamma/\mu_B B'$ and t_0 is defined such that

$$\bar{v} = \frac{kv}{\Gamma} = k\frac{x_0}{\Gamma t_0}\bar{v}$$

implying that

$$t_0 = \frac{kx_0}{\Gamma}$$

Finally, we need the mass, which is defined through the prefactor to the force equation. WeâĂŹll factor out the magnitude of the $\mathbf{k}$ vector because :

$$\ddot{\mathbf{r}} = \frac{\mathbf{f}}{m} = \frac{\hbar k\Gamma}{m}\hat{\mathbf{k}}$$

Note that I neglected the sum, since that is dimensionless already. I can now put in the units explicitly:

$$\frac{x_0}{t_0^2}\ddot{\bar{\mathbf{r}}} = \frac{\hbar k\Gamma}{m}\hat{\mathbf{k}}$$

Rearranging,

$$\ddot{\bar{\mathbf{r}}} = \frac{\hbar k\Gamma t_0^2}{m x_0}\hat{\mathbf{k}} = \frac{\hbar k^2 t_0}{m}\hat{\mathbf{k}} = \frac{\bar{\mathbf{f}}}{\bar{m}}$$

where $\bar{m} = m/(\hbar k^2 t_0)$.

Note again that this unit system is effectively measuring lengths in two different ways - one in terms of k and the other in terms of the magnetic field. This works because we have the mass term which we can adjust. However, if you wanted to include magnetic forces, or use the optical Bloch equations, this unit system will not work as the forces calculated in those schemes have quite a different constant.

Plugging in the numbers, we find:

[2]:
```
x0 = (6/1.4/15) # cm
k = 2*np.pi/780E-7 # cm^{-1}
kbar = k*x0

gamma = 2*np.pi*6e6
```

(continues on next page)

(continued from previous page)

```
t0 = k*x0/gamma
print(x0, k, kbar, 1/gamma, t0)

mass = 86.909180527*cts.value('atomic mass constant')/(cts.hbar*(k*1e2)**2*t0)
print(mass)
```

```
0.2857142857142857 80553.65778435367 23015.33079552962 2.6525823848649224e-08 0.
↪0006105006105006105
0.03454474231473474
```

Define the problem

As always, we must define the laser beams, magnetic field and Hamiltonian.

[3]:
```
det = -1.5
alpha = 1.0
s = 1.0

laserBeams = pylcp.laserBeams([
    {'kvec':np.array([0., 0., 1.]), 'pol':np.array([0., 0., 1.]), 's':s, 'delta':det},
    {'kvec':np.array([0., 0., -1.]), 'pol':np.array([1., 0., 0.]), 's':s, 'delta':det}],
    beam_type=pylcp.infinitePlaneWaveBeam
)

magField = pylcp.quadrupoleMagneticField(alpha)

# Use the heuristic equation (or comment it out):
eqn = pylcp.heuristiceq(laserBeams, magField, gamma=1, mass=mass)

# Define the atomic Hamiltonian:
# Hg, muqg = pylcp.hamiltonians.singleF(F=0, muB=1)
# He, muqe = pylcp.hamiltonians.singleF(F=1, muB=1)

# dq = pylcp.hamiltonians.dqij_two_bare_hyperfine(0, 1)

# hamiltonian = pylcp.hamiltonian(Hg, He, muqg, muqe, dq, mass=mass)

# eqn = pylcp.rateeq(laserBeams, magField, hamiltonian, include_mag_forces=False)
```

Compute equilibrium force

As with the previous example, we will calculate over the 2D phase space defined by z and v_z.

[4]:
```
dz = 0.1
dv = 0.1
z = np.arange(-20, 20+dz, dz)
v = np.arange(-20, 20+dv, dv)

Z, V = np.meshgrid(z, v)
```

(continues on next page)

(continued from previous page)

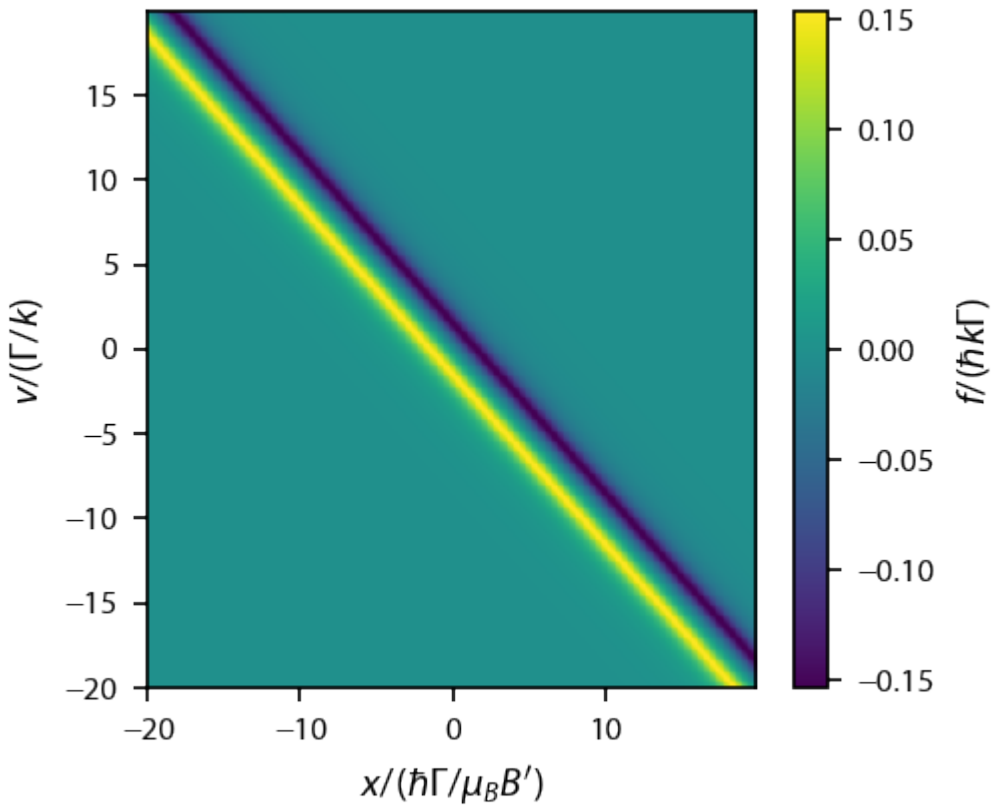
```
Rfull = np.array([np.zeros(Z.shape), np.zeros(Z.shape), Z])
Vfull = np.array([np.zeros(Z.shape), np.zeros(Z.shape), V])

eqn.generate_force_profile([np.zeros(Z.shape), np.zeros(Z.shape), Z],
                           [np.zeros(V.shape), np.zeros(V.shape), V],
                           name='Fz', progress_bar=True);
```

```
Completed in 29.65 s.
```

Plot up the result:

[5]:
```
fig, ax = plt.subplots(1, 1)
plt.imshow(eqn.profile['Fz'].F[2], origin='bottom',
           extent=(np.amin(z)-dz/2, np.amax(z)-dz/2,
                   np.amin(v)-dv/2, np.amax(v)-dv/2),
           aspect='auto')
cb1 = plt.colorbar()
cb1.set_label('$f/(\hbar k \Gamma)$')
ax.set_xlabel('$x/(\hbar \Gamma/\mu_B B\')$')
ax.set_ylabel('$v/(\Gamma/k)$')
fig.subplots_adjust(left=0.12,right=0.9)
```



Add trajectories in phase space

WeâĂŹll use the evolve motion method to evolve the particle and simulate capture. But we also need to define some stop conditions, either when the atom is captured at the origin or lost to âĂIJ+∞âĂİ.

[6]:
```
v0s = np.arange(1, 15.5, 1)

# See solve_ivp documentation for event function discussion:
def captured_condition(t, y, threshold=1e-5):
    if(y[-4]<threshold and y[-1]<1e-3):
        val = -1.
    else:
        val = 1.

    return val

def lost_condition(t, y, threshold=1e-5):
    if y[-1]>20.:
        val = -1.
    else:
        val = 1.

    return val

captured_condition.terminal=True
lost_condition.terminal=True

sols = []
for v0 in v0s:
    eqn.set_initial_position_and_velocity(np.array([0., 0., z[0]]),
                                          np.array([0., 0., v0]))
    if isinstance(eqn, pylcp.rateeq):
        eqn.set_initial_pop(np.array([1., 0., 0., 0.]))

    eqn.evolve_motion([0., 100.], events=[captured_condition, lost_condition], max_
↪step=0.1)

    sols.append(eqn.sol)
```

Now, plot up the additional trajectories in white:

[7]:
```
fig, ax = plt.subplots(1, 1)
plt.imshow(eqn.profile['Fz'].F[2], origin='bottom',
           extent=(np.amin(z)-dz/2, np.amax(z)-dz/2,
                   np.amin(v)-dv/2, np.amax(v)-dv/2),
           aspect='auto')
cb1 = plt.colorbar()
cb1.set_label('$f/(\hbar k \Gamma)$')
ax.set_xlabel('$x/(\hbar \Gamma/\mu_B B\')$')
ax.set_ylabel('$v/(\Gamma/k)$')

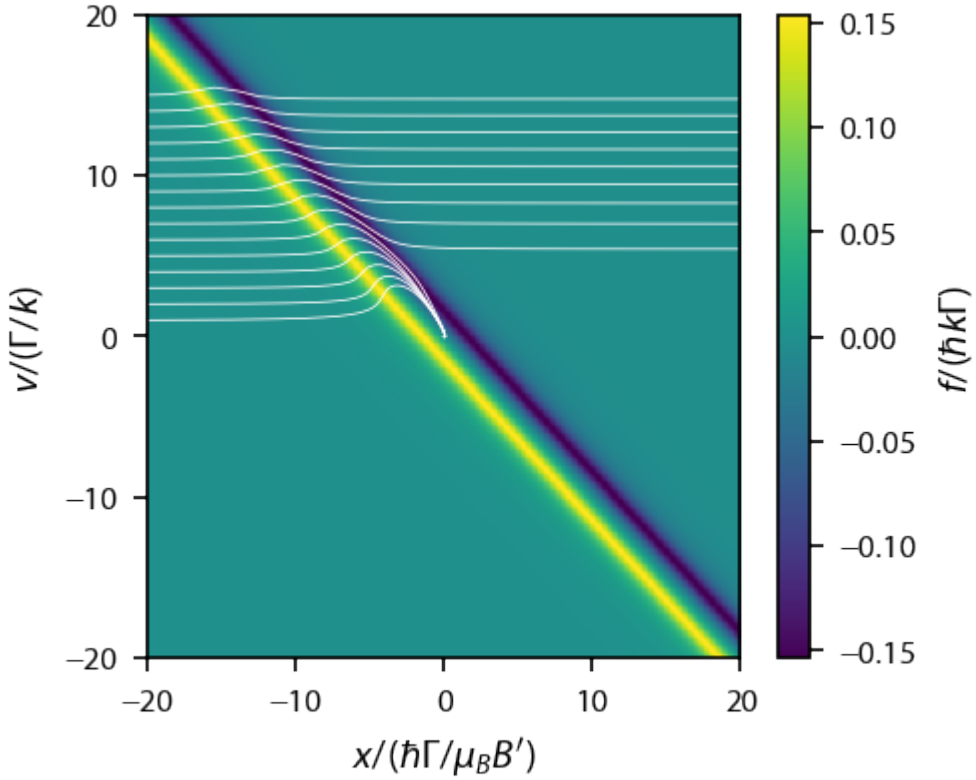
fig.subplots_adjust(left=0.15, right=0.91, bottom=0.2)
```

(continues on next page)

(continued from previous page)

```
for sol in sols:
    ax.plot(sol.r[2], sol.v[2], 'w-', linewidth=0.375)

ax.yaxis.set_ticks([-20, -10, 0, 10, 20])
# Display the figure at the end of the thing.
ax.set_xlim((-20, 20))
ax.set_xlim((-20, 20));
```



By having two conditions, we can tell if the atom was lost or captured:

[8]:
```
for sol in sols:
    if len(sol.t_events[0]) == 1:
        print('captured')
    elif len(sol.t_events[1]) == 1:
        print('lost')
```

```
captured
captured
captured
captured
captured
captured
lost
lost
lost
lost
```

(continues on next page)

(continued from previous page)

```
lost
lost
lost
lost
lost
```

Solve for the capture velocity v_c

LetâĂŹs define a function that figures out if we were captured or not, then use that to find the capture velocity:

[9]:
```
def iscaptured(v0, z0, eqn, captured_condition, lost_condition, tmax=1000, max_step=0.1,␣
↪**kwargs):
    eqn.set_initial_position_and_velocity(np.array([0., 0., z0]),
                                          np.array([0., 0., v0]))
    eqn.evolve_motion([0., tmax], events=[captured_condition, lost_condition],
                      max_step=max_step)

    if len(eqn.sol.t_events[0]) == 1:
        return +1.
    else:
        return -1.

iscaptured(1.3, z[0], eqn, captured_condition, lost_condition)
```

[9]:
```
1.0
```

Use `scipy.optimize.bisect` to see where the `iscaptured` function changes from -1 (false) to 1 (true):

[10]:
```
bisect(iscaptured, 1.0, 15.,
       args=(z[0], eqn, captured_condition, lost_condition),
       xtol=1e-4, rtol=1e-4, full_output=True
      )
```

[10]:
```
(6.20770263671875,
       converged: True
            flag: 'converged'
  function_calls: 17
      iterations: 15
            root: 6.20770263671875)
```

Dependence of v_c on detuning and intensity

We will figure out how the capture velocity depends on and compare to this equation from the paper in the introduction:

$$v_c = \left(\frac{a_0^2 s^2 \kappa}{(1+s)^{3/2}}\right)^{1/3} \left(\frac{8\pi\delta^2}{1+s+4\delta^2}\right)^{1/3} \zeta^{-2/3}$$

where $a_0 = \hbar k\Gamma/(2m)$, $\zeta = \mu_B B'/(\hbar\Gamma)$, and $\kappa = 2\pi/(\lambda\Gamma) = k/\Gamma$. To compare, we need to express it in a way which connects with our formulae above. The first thing to note is that $\zeta = 1/x_0$. We also need to multiple both sides by k/Γ, so that we have $v_c/(\Gamma/k)$ on the left side, which is our observable. Then, we realize that

$$\frac{\hbar k\Gamma}{2m} = \frac{1}{2\bar{m}}\frac{x_0}{t_0^2} \quad \text{and} \quad \frac{k}{\Gamma} = \frac{t_0}{x_0}$$

Putting it all together:

$$\frac{v_c}{\Gamma/k} = \frac{t_0}{x_0}\left(\frac{1}{2\bar{m}}\right)^{2/3}\frac{x_0^{2/3}}{t_0^{4/3}}\frac{t_0^{1/3}}{x_0^{1/3}}x_0^{2/3}\left(\frac{s^2}{(1+s)^{3/2}}\right)^{1/3}\left(\frac{8\pi\delta^2}{1+s+4\delta^2}\right)^{1/3} = \left(\frac{1}{2\bar{m}}\right)^{2/3}\left(\frac{s^2}{(1+s)^{3/2}}\right)^{1/3}\left(\frac{8\pi\delta^2}{1+s+4\delta^2}\right)^{1/3}$$

[11]:
```
dets = -np.logspace(-1, np.log10(5), 30)[::-1]
intensities = np.array([0.3, 1., 3.])

DETS, INTENSITIES = np.meshgrid(dets, intensities)

it = np.nditer([DETS, INTENSITIES, None, None],
               op_dtypes=['float64', 'float64', 'float64', np.object])

progress = progressBar()
for (det, s, vc, full_results) in it:
    laserBeams = pylcp.laserBeams([
        {'kvec':np.array([0., 0., 1.]), 'pol':np.array([0., 0., 1.]), 's':s, 'delta':det}
↪,
        {'kvec':np.array([0., 0., -1.]), 'pol':np.array([1., 0., 0.]), 's':s, 'delta':
↪det}],
        beam_type=pylcp.infinitePlaneWaveBeam
    )

    # Heuristic equation or rate equation?
    eqn = pylcp.heuristiceq(laserBeams, magField, gamma=1, mass=mass)
    #eqn = pylcp.rateeq(laserBeams, magField, hamiltonian, include_mag_forces=False)

    if isinstance(eqn, pylcp.rateeq):
        eqn.set_initial_pop(np.array([1., 0., 0., 0.]))

    vc[...], full_results[...] = bisect(
        iscaptured, 0.5, 15.0,
        args=(z[0], eqn, captured_condition, lost_condition),
        rtol=1e-4, xtol=1e-4, full_output=True
    )

    progress.update((it.iterindex+1)/it.itersize)
```

```
Completed in 15:07.
```

[12]:
```
def vc_from_paper(delta, s, mbar):
    return 1/(2*mbar)**(2./3.)*(s**2/(1+s)**(3./2.))**(1./3.)*(8*np.pi*delta**2/
↪(1+s+4*delta**2))**(1./3.)
```

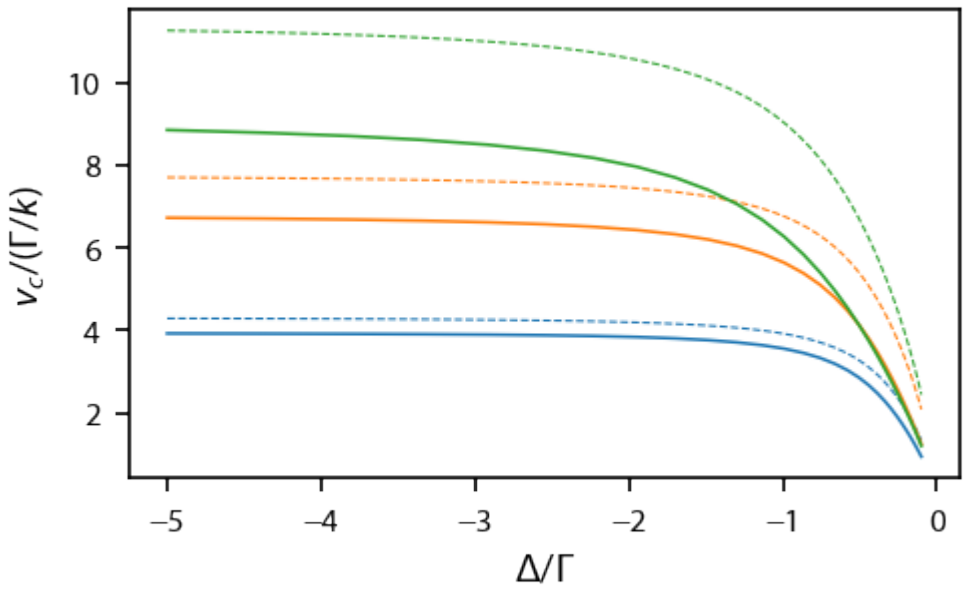
[13]:
```
dets_thr = -np.logspace(-1, np.log10(5), 51)[::-1]
fig, ax = plt.subplots(1, 1, figsize=(3.25, 2))
for ii, (s, vc_vs_det) in enumerate(zip(intensities, it.operands[2])):
    ax.plot(dets, vc_vs_det, label='$s=%.1f$' % s,
            color='C%d'%ii, linewidth=0.75)
    ax.plot(dets_thr, vc_from_paper(dets_thr, s, mass), '--',
            color='C%d'%ii, linewidth=0.5)
```

(continues on next page)

(continued from previous page)

```
#ax.legend(fontsize=8)
ax.set_xlabel('$\Delta/\Gamma$')
ax.set_ylabel('$v_c/(\Gamma/k)$')

fig.subplots_adjust(left=0.13, bottom=0.2)
```



3.3.3 $F = 0 \to F' = 1$ MOT forces with the OBEs

This example covers calculating the forces in a one-dimensional MOT using the optical bloch equations. This example does the boring thing and checks that everything is working on the $F = 0 \to F' = 1$ transition.

It first checks the force along the $\hat{z}$-direction. One should look to see that things agree with what one expects whether or not one puts the detuning on the lasers or on the Hamilonian. One should also look at whether the force depends on transforming the OBEs into the real/imaginary components.

It then checks the force along the $\hat{x}$ and $\hat{y}$ directions. This is important because the OBEs solve this in a different way compared to the rate equations. Whereas the rate equatons rediagonalize the Hamiltonian for a given direction, the OBE solves everything in the $\hat{z}$-basis.

Finally, we plop an atom into the 1D MOT and see how it evolves with time.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
import time
```

Define the problem

We will define multiple laser beam configurations to start, namely with beams pointed along each axis. Of course, we must also define the Hamiltonian and magnetic field.

[2]:
```
laser_det = 0
ham_det = -2.5
s = 1.
transform = True

laserBeams = {}
laserBeams['x'] = pylcp.laserBeams([
    {'kvec':np.array([ 1., 0., 0.]), 'pol':-1, 'delta':laser_det, 's':s},
    {'kvec':np.array([-1., 0., 0.]), 'pol':-1, 'delta':laser_det, 's':s},
    ], beam_type=pylcp.infinitePlaneWaveBeam)
laserBeams['y'] = pylcp.laserBeams([
    {'kvec':np.array([0.,  1., 0.]), 'pol':-1, 'delta':laser_det, 's':s},
    {'kvec':np.array([0., -1., 0.]), 'pol':-1, 'delta':laser_det, 's':s}
    ], beam_type=pylcp.infinitePlaneWaveBeam)
laserBeams['z'] = pylcp.laserBeams([
    {'kvec':np.array([0., 0.,  1.]), 'pol':+1, 'delta':laser_det, 's':s},
    {'kvec':np.array([0., 0., -1.]), 'pol':+1, 'delta':laser_det, 's':s}
    ], beam_type=pylcp.infinitePlaneWaveBeam)

alpha = 1e-3
magField = pylcp.quadrupoleMagneticField(alpha)

# Hamiltonian for F=0->F=1
H_g, muq_g = pylcp.hamiltonians.singleF(F=0, gF=0, muB=1)
H_e, muq_e = pylcp.hamiltonians.singleF(F=1, gF=1, muB=1)
d_q = pylcp.hamiltonians.dqij_two_bare_hyperfine(0, 1)
hamiltonian = pylcp.hamiltonian(H_g, H_e - ham_det*np.eye(3), muq_g, muq_e, d_q,␣
↪mass=250)
```

Compute equilibrium force along $\hat{z}$

This checks to make sure that the rate equations and OBE agree:

[3]:
```
obe={}
rateeq={}

obe['z'] = {}
rateeq['z'] = {}

z = np.arange(-5.0, 5.01, 0.25)

rateeq['z'] = pylcp.rateeq(laserBeams['z'], magField, hamiltonian)
rateeq['z'].generate_force_profile(
    [np.zeros(z.shape), np.zeros(z.shape), z/alpha],
    np.zeros((3,) + z.shape),
    name='MOT_1'
)
```

(continues on next page)

(continued from previous page)

```
obe['z'] = pylcp.obe(laserBeams['z'], magField, hamiltonian,
                     transform_into_re_im=transform,
                     include_mag_forces=True)
obe['z'].generate_force_profile(
    [np.zeros(z.shape), np.zeros(z.shape), z/alpha],
    np.zeros((3,) + z.shape),
    name='MOT_1', deltat_tmax=2*np.pi*100, deltat_r=4/alpha,
    itermax=1000, progress_bar=True
);
```

```
Completed in 7.75 s.
```

Plot up the results:

[4]:
```
fig, ax = plt.subplots(3, 2, num='Optical Molasses F=0->F1', figsize=(6.5, 3*2.25))

Es = np.zeros((z.size, 4))
for ii, z_i in enumerate(z):
    Bq = np.array([0., magField.Field(np.array([0., 0., z_i/alpha]))[2], 0])
    Es[ii, :] = np.real(np.diag(hamiltonian.return_full_H({'g->e':np.array([0., 0., 0.])}
↪, Bq)))

[ax[0, 0].plot(z, Es[:, 1+jj], label='$m_F=%d$'%(jj-1)) for jj in range(3)]
ax[0, 0].legend(fontsize=6)
ax[0, 0].set_ylabel('$E/(\hbar \Gamma)$')

types = ['-', '--', '-.']
lbls = ['+k','-k']
ax[0, 1].plot(obe['z'].profile['MOT_1'].R[2]*alpha,
           obe['z'].profile['MOT_1'].F[2],
           label='OBE', linewidth=0.75)
ax[0, 1].plot(rateeq['z'].profile['MOT_1'].R[2]*alpha,
           rateeq['z'].profile['MOT_1'].F[2],
           label='Rate Eq.', linewidth=0.5)
for jj in range(2):
    ax[0, 1].plot(obe['z'].profile['MOT_1'].R[2]*alpha,
                  obe['z'].profile['MOT_1'].f['g->e'][2, :, jj],
                  types[jj+1], color='C0', linewidth=0.75, label=lbls[jj])
    ax[0, 1].plot(rateeq['z'].profile['MOT_1'].R[2]*alpha,
                  rateeq['z'].profile['MOT_1'].f['g->e'][2, :, jj],
                  types[jj+1], color='C1', linewidth=0.5)
ax[0, 1].legend(fontsize=6)
ax[0, 1].set_ylabel('$f/(\hbar k \Gamma)$')

for q in range(3):
    ax[1, 1].plot(z, obe['z'].profile['MOT_1'].fq['g->e'][2, :, q, 0], types[q],
            linewidth=0.5, color='C0', label='$+k$, $q=%d$'%(q-1))
    ax[1, 1].plot(z, obe['z'].profile['MOT_1'].fq['g->e'][2, :, q, 1], types[q],
            linewidth=0.5, color='C1', label='$-k$, $q=%d$'%(q-1))
ax[1, 1].plot(z, obe['z'].profile['MOT_1'].F[2], 'k-',
           linewidth=0.75)
ax[1, 1].legend(fontsize=6)
```

(continues on next page)

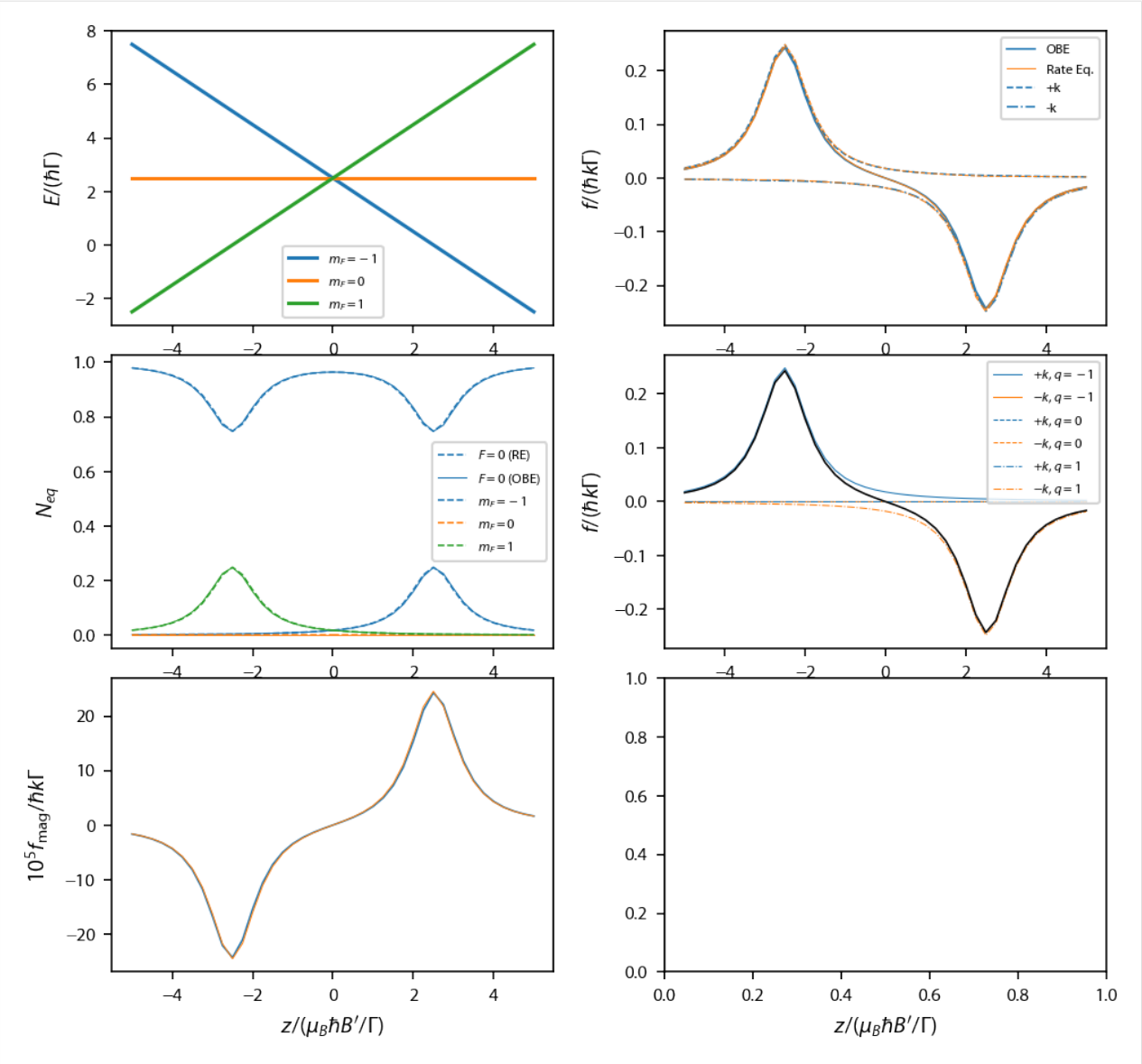
(continued from previous page)

```
ax[1, 1].set_xlabel('$z/(\mu_B \hbar B\'/\Gamma)$')
ax[1, 1].set_ylabel('$f/(\hbar k \Gamma)$')
fig.subplots_adjust(wspace=0.15)

ax[1, 0].plot(z, rateeq['z'].profile['MOT_1'].Neq[:, 0], '--',
              linewidth=0.75, label='$F=0$ (RE)')
ax[1, 0].plot(z, obe['z'].profile['MOT_1'].Neq[:, 0], '-',
              linewidth=0.5, color='C0', label='$F=0$ (OBE)')
for jj in range(3):
    ind = z<=0
    ax[1, 0].plot(z[ind], rateeq['z'].profile['MOT_1'].Neq[ind, 3-jj], '--',
                  linewidth=0.75, color='C%d'%jj, label='$m_F=%d$'%(jj-1))
    ind = z>=0
    ax[1, 0].plot(z[ind], rateeq['z'].profile['MOT_1'].Neq[ind, jj+1], '--',
                  linewidth=0.75, color='C%d'%jj)
    ax[1, 0].plot(z, obe['z'].profile['MOT_1'].Neq[:, jj+1], '-',
                  linewidth=0.5, color='C%d'%jj)

ax[1, 0].legend(fontsize=6)
ax[1, 0].set_ylabel('$N_{eq}$')

ax[2, 0].plot(z, 1e5*obe['z'].profile['MOT_1'].f_mag[2], linewidth=0.75)
ax[2, 0].plot(z, 1e5*rateeq['z'].profile['MOT_1'].f_mag[2], linewidth=0.5)
ax[2, 0].set_ylabel('$10^5 f_{\\rm mag}/\hbar k \Gamma$')
ax[2, 0].set_xlabel('$z/(\mu_B \hbar B\'/\Gamma)$')
ax[2, 1].set_xlabel('$z/(\mu_B \hbar B\'/\Gamma)$')
fig.subplots_adjust(left=0.08, wspace=0.25)
```



Compute equilibrium forces along $\hat{x}$ and $\hat{y}$

Same as before, this steps makes sure that the OBEs are correctly evolving in other directions.

[5]:
```
R = {}
R['x'] = [2*z/alpha, np.zeros(z.shape), np.zeros(z.shape)]
R['y'] = [np.zeros(z.shape), 2*z/alpha, np.zeros(z.shape)]

for key in ['x','y']:
    rateeq[key] = pylcp.rateeq(laserBeams[key], magField,
                               hamiltonian)
    rateeq[key].generate_force_profile(
        R[key], np.zeros((3,) + z.shape), name='MOT_1'
    )
```

(continues on next page)

(continued from previous page)

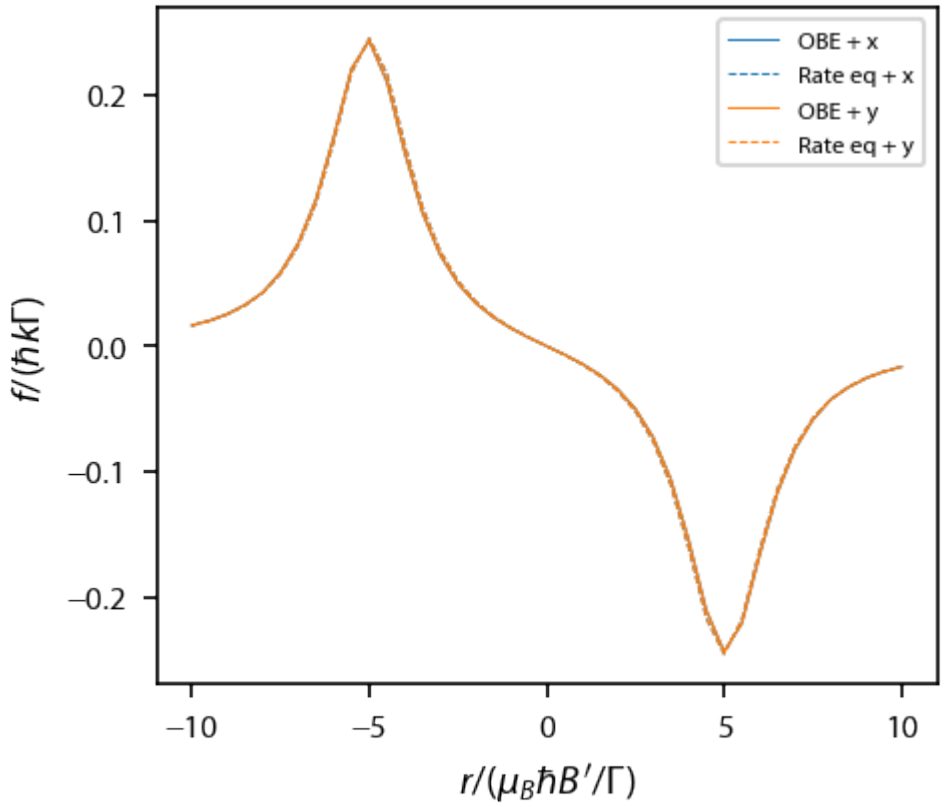
```
    obe[key] = pylcp.obe(laserBeams[key], magField,  hamiltonian,
                         transform_into_re_im=transform,
                         include_mag_forces=False)

    obe[key].generate_force_profile(
        R[key], np.zeros((3,) + z.shape), name='MOT_1',
        deltat_tmax=2*np.pi*100, deltat_r=4/alpha, itermax=1000,
        progress_bar=True,
    )
```

```
Completed in 6.66 s.
Completed in 6.48 s.
```

Plot this one up:

[6]:
```
fig, ax = plt.subplots(1, 1)
for ii, key in enumerate(['x','y']):
    ax.plot(obe[key].profile['MOT_1'].R[ii]*alpha,
            obe[key].profile['MOT_1'].F[ii],
            label='OBE + %s' % key, color='C%d'%ii, linewidth=0.5)
    ax.plot(rateeq[key].profile['MOT_1'].R[ii]*alpha,
            rateeq[key].profile['MOT_1'].F[ii], '--',
            label='Rate eq + %s' % key, color='C%d'%ii, linewidth=0.5)
ax.legend(fontsize=6)
ax.set_xlabel('$r/(\mu_B \hbar B\'/\Gamma)$')
ax.set_ylabel('$f/(\hbar k \Gamma)$');
```



Place an atom in this MOT

WeâĂŹll place it at rest, but displaced from the origin. It should first be accelerated toward the origin then damped. Depending on the mass parameter, it may execute an oscillation or two.

[7]:
```
for key in obe:
    if key is 'x':
        ii = 0
    elif key is 'y':
        ii = 1
    elif key is 'z':
        ii = 2

    r0 = np.zeros((3,))
    r0[ii] = 10.

    obe[key].set_initial_position(r0)
    obe[key].set_initial_velocity(np.zeros((3,)))
    obe[key].set_initial_rho_from_rateeq()

    freeze_axis = [True]*3
    freeze_axis[ii] = False

    obe[key].evolve_motion([0, 5e4],
```

(continues on next page)

(continued from previous page)

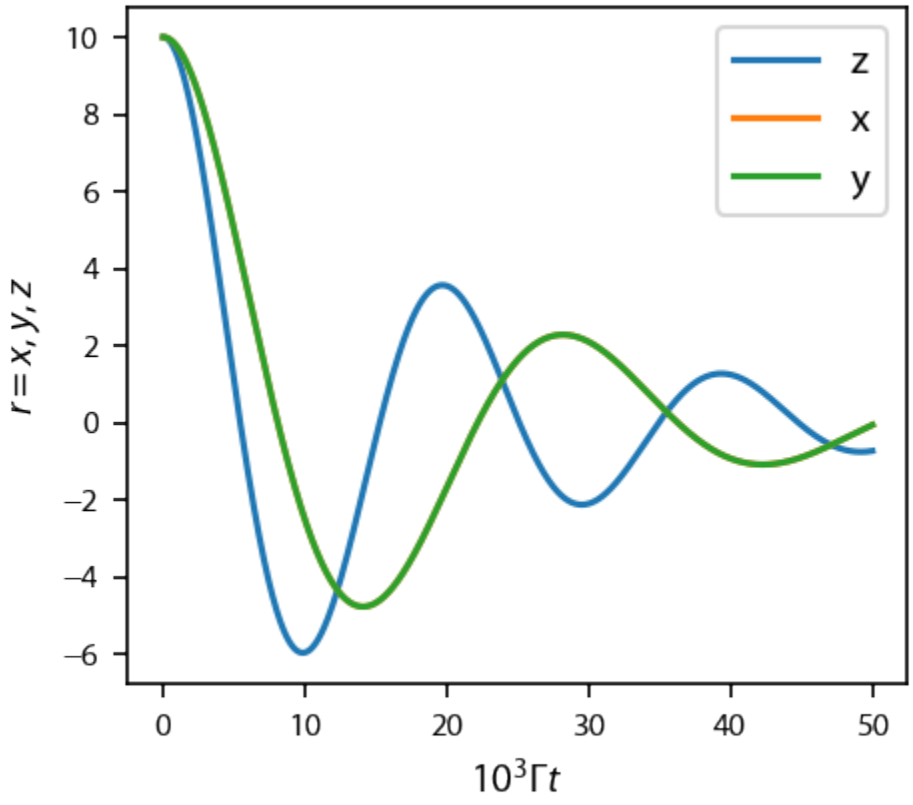
```
                            freeze_axis=freeze_axis,
                            progress_bar=True,
                            random_recoil=False
                           )
```

```
Completed in 2:06.
Completed in 1:42.
Completed in 1:41.
```

[9]:
```
fig, ax = plt.subplots(1, 1)
for key in obe:
    if key is 'x':
        ii = 0
    elif key is 'y':
        ii = 1
    elif key is 'z':
        ii = 2

    ax.plot(obe[key].sol.t/1e3, obe[key].sol.r[ii], label=key)

ax.legend()
ax.set_xlabel('$10^3 \Gamma t$')
ax.set_ylabel('$r=x,y,z$');
```



3.3.4 $F = 0 \to F = 1$ MOT temperature with the OBE

This example covers single atom evolution in a 3D MOT with no gravity using the optical Bloch equations. It highlights an interesting effect of the 3D lattice that is inherent in all MOTs.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
import lmfit
from pylcp.common import progressBar
```

Define the problem

Laser beams, magnetic field, and Hamiltonian.

[2]:
```
laser_det = 0
det = -2.5
s = 1.25
transform = True

laserBeams = pylcp.conventional3DMOTBeams(
    s=s, delta=0., beam_type=pylcp.infinitePlaneWaveBeam
)
#laserBeams.beam_vector[2:7] = [] # Delete the y,z beams
#laserBeams.num_of_beams = 2

alpha = 1e-4
magField = pylcp.quadrupoleMagneticField(alpha)

# Hamiltonian for F=0->F=1
H_g, muq_g = pylcp.hamiltonians.singleF(F=0, gF=0, muB=1)
H_e, muq_e = pylcp.hamiltonians.singleF(F=1, gF=1, muB=1)
d_q = pylcp.hamiltonians.dqij_two_bare_hyperfine(0, 1)
hamiltonian = pylcp.hamiltonian(H_g, -det*np.eye(3)+H_e, muq_g, muq_e, d_q, mass=100)

obe = pylcp.obe(laserBeams, magField, hamiltonian,
                transform_into_re_im=transform)
```

Calculate the equilibrium force

LetâĂŹs try looking at a single force profile along each axis, $\hat{x}$, $\hat{y}$, and $\hat{z}$:

[3]:
```
z = np.arange(-5.01, 5.01, 0.25)

R = {}
R['x'] = [2*z/alpha, np.zeros(z.shape), np.zeros(z.shape)]
R['y'] = [np.zeros(z.shape), 2*z/alpha, np.zeros(z.shape)]
R['z'] = [np.zeros(z.shape), np.zeros(z.shape), z/alpha]

V = {
    'x':[0.0*np.ones(z.shape), np.zeros(z.shape), np.zeros(z.shape)],
```

(continues on next page)

(continued from previous page)

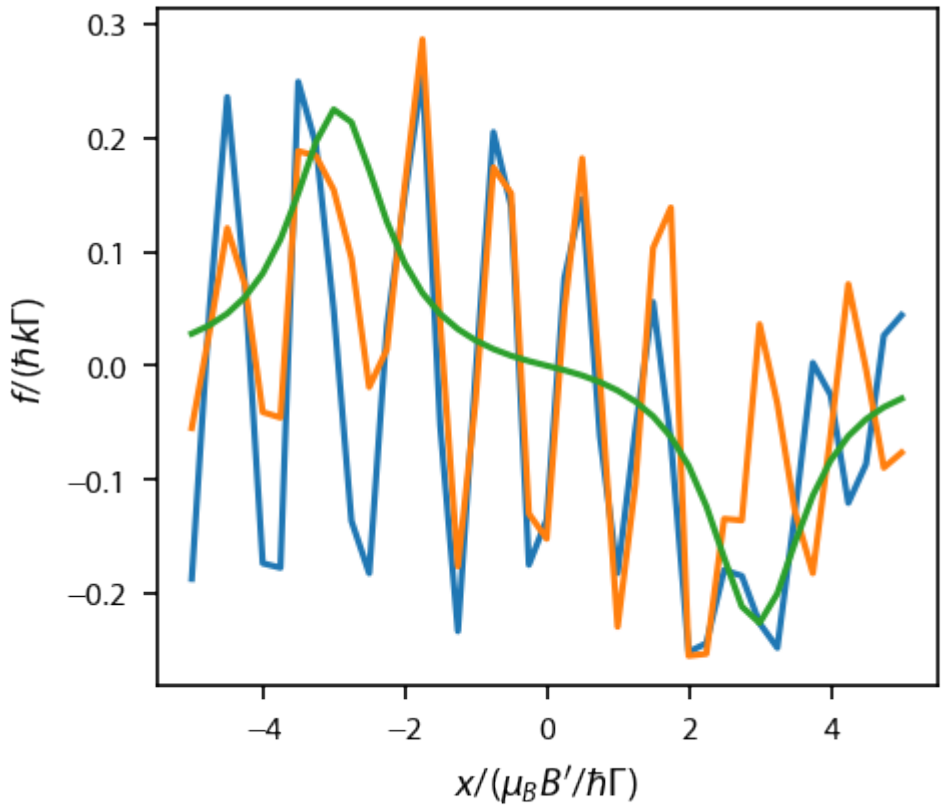
```
    'y':[np.zeros(z.shape), 0.0*np.ones(z.shape), np.zeros(z.shape)],
    'z':[np.zeros(z.shape), np.zeros(z.shape), 0.0*np.ones(z.shape)]
}
for key in R:
    obe.generate_force_profile(
        R[key], V[key],
        name=key, deltat_tmax=2*np.pi*100, deltat_r=4/alpha,
        itermax=1000, progress_bar=True
    )
```

```
Completed in 10.03 s.
Completed in 9.11 s.
Completed in 10.50 s.
```

Plot it up:

[4]:
```
fig, ax = plt.subplots(1, 1)
for ii, key in enumerate(obe.profile):
    ax.plot(z, obe.profile[key].F[ii])

ax.set_xlabel('$x/(\mu_B B\'/\hbar \Gamma)$')
ax.set_ylabel('$f/(\hbar k \Gamma)$');
```



Obviously there is something going on with the $\hat{x}$ and $\hat{y}$ directions. The thing that is going on is interference in the optical lattice created by the 6 beams, and it is pronounced because the atom is not moving. LetâĂŹs repeat this exercise, but average over a period of the laser lattice.

Average over the lattice

Again, note the the x and y calculation is a factor of 2 larger than z.

[5]:
```
z = np.arange(-5.01, 5.01, 0.25)

R = {}
R['x'] = [2*z/alpha, np.zeros(z.shape), np.zeros(z.shape)]
R['y'] = [np.zeros(z.shape), 2*z/alpha, np.zeros(z.shape)]
R['z'] = [np.zeros(z.shape), np.zeros(z.shape), z/alpha]

V = {
    'x':[0.0*np.ones(z.shape), np.zeros(z.shape), np.zeros(z.shape)],
    'y':[np.zeros(z.shape), 0.0*np.ones(z.shape), np.zeros(z.shape)],
    'z':[np.zeros(z.shape), np.zeros(z.shape), 0.0*np.ones(z.shape)]
}

Npts = 128
for key in R:
    progress = progressBar()
    for ii in range(Npts):
        obe.generate_force_profile(
            R[key] + 2*np.pi*(np.random.rand(3)-0.5).reshape(3,1), V[key],
            name=key + '_%d'%ii, deltat_tmax=2*np.pi*100, deltat_r=4/alpha,
            itermax=1000, progress_bar=False
        )
        progress.update((ii+1)/Npts)
```

```
Completed in 23:25.
Completed in 41:04.
Completed in 35:21.
```

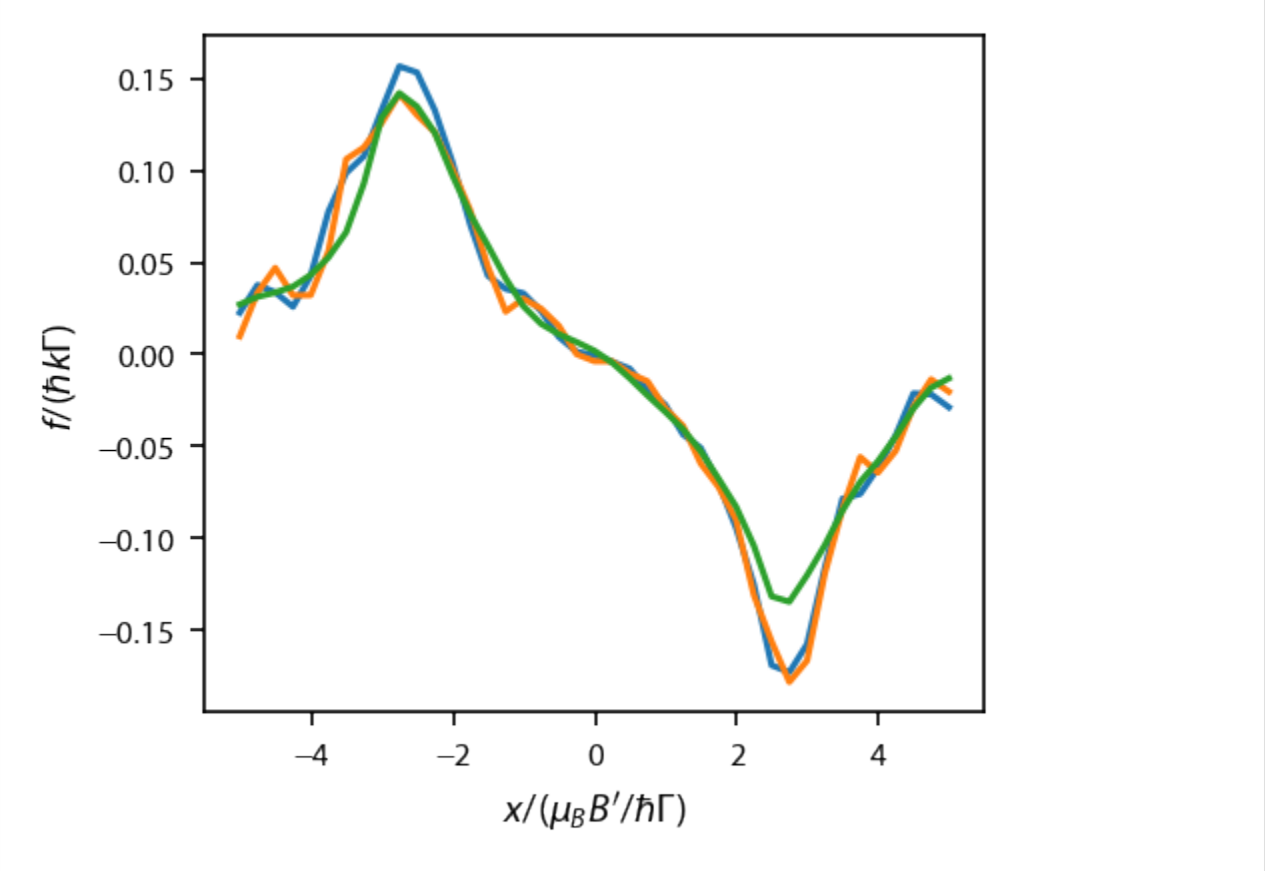
Now take the average:

[6]:
```
avgF = {}
for coord_key in R:
    avgF[coord_key] = np.sum([obe.profile[key].F for key in obe.profile if coord_key in␣
↪key], axis=0)/Npts
```

Now plot it up:

[7]:
```
fig, ax = plt.subplots(1, 1)
for ii, key in enumerate(R):
    ax.plot(z, avgF[key][ii])

ax.set_xlabel('$x/(\mu_B B\'/\hbar \Gamma)$')
ax.set_ylabel('$f/(\hbar k \Gamma)$');
```



That looks much better.

Evolution without random scattering

One can choose various initial states. Sometimes we appear to get trapped in some lattice if we choose our initial state poorly.

[22]:
```
# %% Now try to evolve some initial state!
obe.v0 = np.array([0., 0., 0.])
obe.r0 = np.random.randn(3)/alpha
#obe.r0 = np.array([0., 100., 0.])
#obe.r0 = np.array([0., 0., 10.])
obe.set_initial_rho_from_rateeq()
# obe.set_initial_rho_equally()

t_span = [0, 1e4]

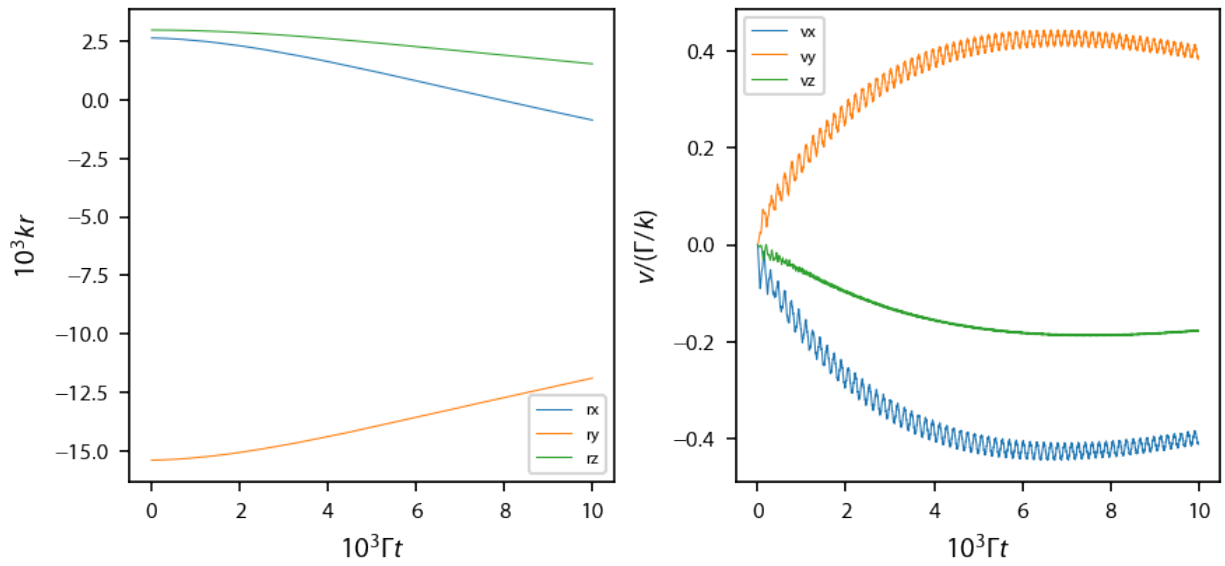
obe.evolve_motion(t_span,
                  progress_bar=True,
                  random_recoil=False
                 );
```

```
Completed in 1:00.
```

Plot it up:

[23]:
```
fig, ax = plt.subplots(1, 2, num='Optical Molasses F=0->F1', figsize=(6.5, 2.75))
ax[0].plot(obe.sol.t/1e3, obe.sol.r[0]/1e3,
           label='rx', linewidth=0.5)
ax[0].plot(obe.sol.t/1e3, obe.sol.r[1]/1e3,
           label='ry', linewidth=0.5)
ax[0].plot(obe.sol.t/1e3, obe.sol.r[2]/1e3,
           label='rz', linewidth=0.5)
ax[0].legend(fontsize=6)
ax[0].set_xlabel('$10^3 \Gamma t$')
ax[0].set_ylabel('$10^3 kr$')

ax[1].plot(obe.sol.t/1e3, obe.sol.v[0],
           label='vx', linewidth=0.5)
ax[1].plot(obe.sol.t/1e3, obe.sol.v[1],
           label='vy', linewidth=0.5)
ax[1].plot(obe.sol.t/1e3, obe.sol.v[2],
           label='vz', linewidth=0.5)
ax[1].legend(fontsize=6)
ax[1].set_xlabel('$10^3 \Gamma t$')
ax[1].set_ylabel('$v/(\Gamma/k)$')
fig.subplots_adjust(wspace=0.25)
```



Evolution with random scattering

First run another test simulation:

[24]:
```
# %% Now try to evolve some initial state!
obe.v0 = 0.1*np.random.randn(3)
obe.r0 = 0.1*np.random.randn(3)/alpha
obe.set_initial_rho_from_rateeq()
# obe.set_initial_rho_equally()
```

(continues on next page)

(continued from previous page)

```
t_span = [0, 5e4]

obe.evolve_motion(t_span,
                  progress_bar=True,
                  random_recoil=True
                 );
```

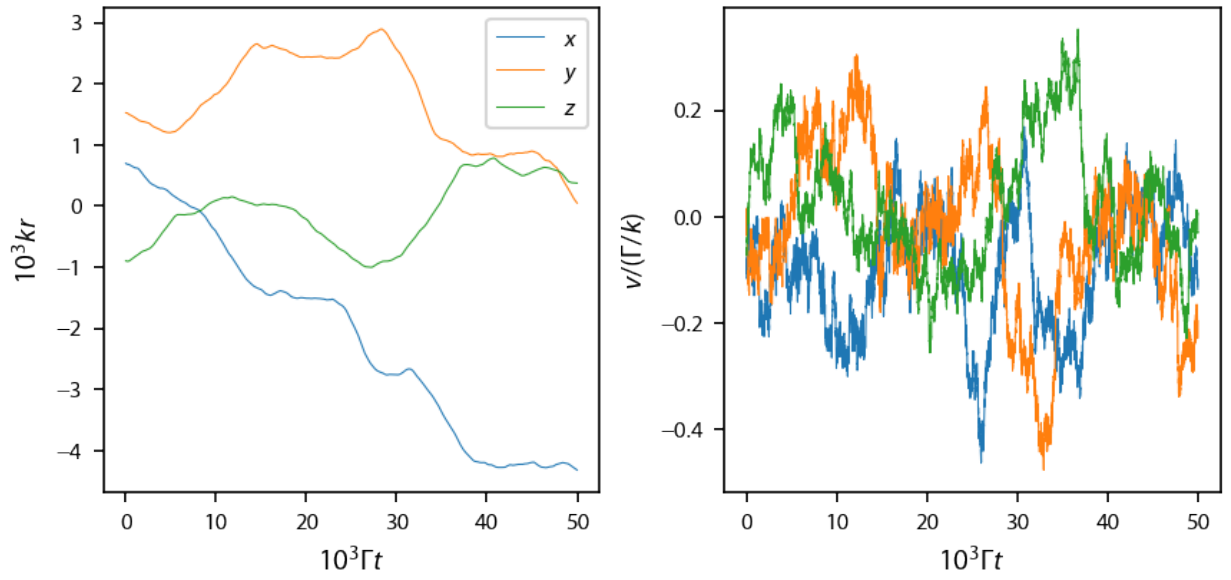
```
Completed in 5:40.
```

Plot it up:

[25]:
```
fig, ax = plt.subplots(1, 2, num='Optical Molasses F=0->F1', figsize=(6.5, 2.75))
ax[0].plot(obe.sol.t/1e3, obe.sol.r[0]/1e3,
           label='$x$', linewidth=0.5)
ax[0].plot(obe.sol.t/1e3, obe.sol.r[1]/1e3,
           label='$y$', linewidth=0.5)
ax[0].plot(obe.sol.t/1e3, obe.sol.r[2]/1e3,
           label='$z$', linewidth=0.5)
ax[0].legend(fontsize=8)
ax[0].set_xlabel('$10^3 \Gamma t$')
ax[0].set_ylabel('$10^3 kr$')

ax[1].plot(obe.sol.t/1e3, obe.sol.v[0],
           label='vx', linewidth=0.5)
ax[1].plot(obe.sol.t/1e3, obe.sol.v[1],
           label='vy', linewidth=0.5)
ax[1].plot(obe.sol.t/1e3, obe.sol.v[2],
           label='vz', linewidth=0.5)
ax[1].set_xlabel('$10^3 \Gamma t$')
ax[1].set_ylabel('$v/(\Gamma/k)$')
fig.subplots_adjust(wspace=0.25)
```

```
/Users/steve/opt/anaconda3/lib/python3.7/site-packages/IPython/core/pylabtools.py:132:␣
↪UserWarning: Creating legend with loc="best" can be slow with large amounts of data.
  fig.canvas.print_figure(bytes_io, **kw)
```



Now run 96 atoms. Again, we parallelize using `pathos`:

[12]:
```
import pathos
if hasattr(obe, 'sol'):
    del obe.sol

tmax = 1e5
args = ([0, tmax], )
kwargs = {'t_eval':np.linspace(0, tmax, 5001),
          'random_recoil':True,
          'progress_bar':False,
          'max_scatter_probability':0.5,
          'record_force':False}

rscale = np.array([2, 2, 2])/alpha
roffset = np.array([0.0, 0.0, 0.0])
vscale = np.array([0.1, 0.1, 0.1])
voffset = np.array([0.0, 0.0, 0.0])

def generate_random_solution(x, tmax=1e5):
    # We need to generate random numbers to prevent solutions from being seeded
    # with the same random number.
    np.random.rand(256*x)
    obe.set_initial_position(rscale*np.random.randn(3) + roffset)
    obe.set_initial_velocity(vscale*np.random.randn(3) + voffset)
    obe.set_initial_rho_from_rateeq()
    obe.evolve_motion(*args, **kwargs)

    return obe.sol

Natoms = 96
chunksize = 4
sols = []
```

(continues on next page)

(continued from previous page)

```
progress = progressBar()
for jj in range(int(Natoms/chunksize)):
    with pathos.pools.ProcessPool(nodes=4) as pool:
        sols += pool.map(generate_random_solution, range(chunksize))
    progress.update((jj+1)/int(Natoms/chunksize))
```

```
Completed in 4:23:31.
```

HereâĂŹs another potential way to parallelize. We export a file that contains all the relevant data, and then execute the script `run_single_sim.py`. That grabs the data from the pickled file, and executes the sim 12 times and dumps the results into pickled files.

```
import dill, os

if hasattr(obe, 'sol'):
    del obe.sol

tmax = 1e5
args = ([0, tmax], )
kwargs = {'t_eval':np.linspace(0, tmax, 5001),
          'random_recoil':True,
          'recoil_velocity':0.01,
          'progress_bar':True,
          'max_scatter_probability':0.5,
          'record_force':False}

rscale = np.array([2, 2, 2])/alpha
roffset = np.array([0.0, 0.0, 0.0])
vscale = np.array([0.1, 0.1, 0.1])
voffset = np.array([0.0, 0.0, 0.0])

with open('parameters.pkl', 'wb') as output:
    dill.dump(obe, output)
    dill.dump(args, output)
    dill.dump(kwargs, output)
    dill.dump((rscale, roffset, vscale, voffset), output)
```

This code reads the pickled files:

```
files = os.listdir(path='./sims/')

sols = []
for file in files:
    if file.endswith('.pkl'):
        with open('sims/'+ file, 'rb') as input:
            sols.append(dill.load(input))

print(len(sols))
```

No matter which way it was parallelized, letâĂŹs plot up the result:

[13]:
```
fig, ax = plt.subplots(3, 2, figsize=(6.25, 2*2.75))
for sol in sols:
```

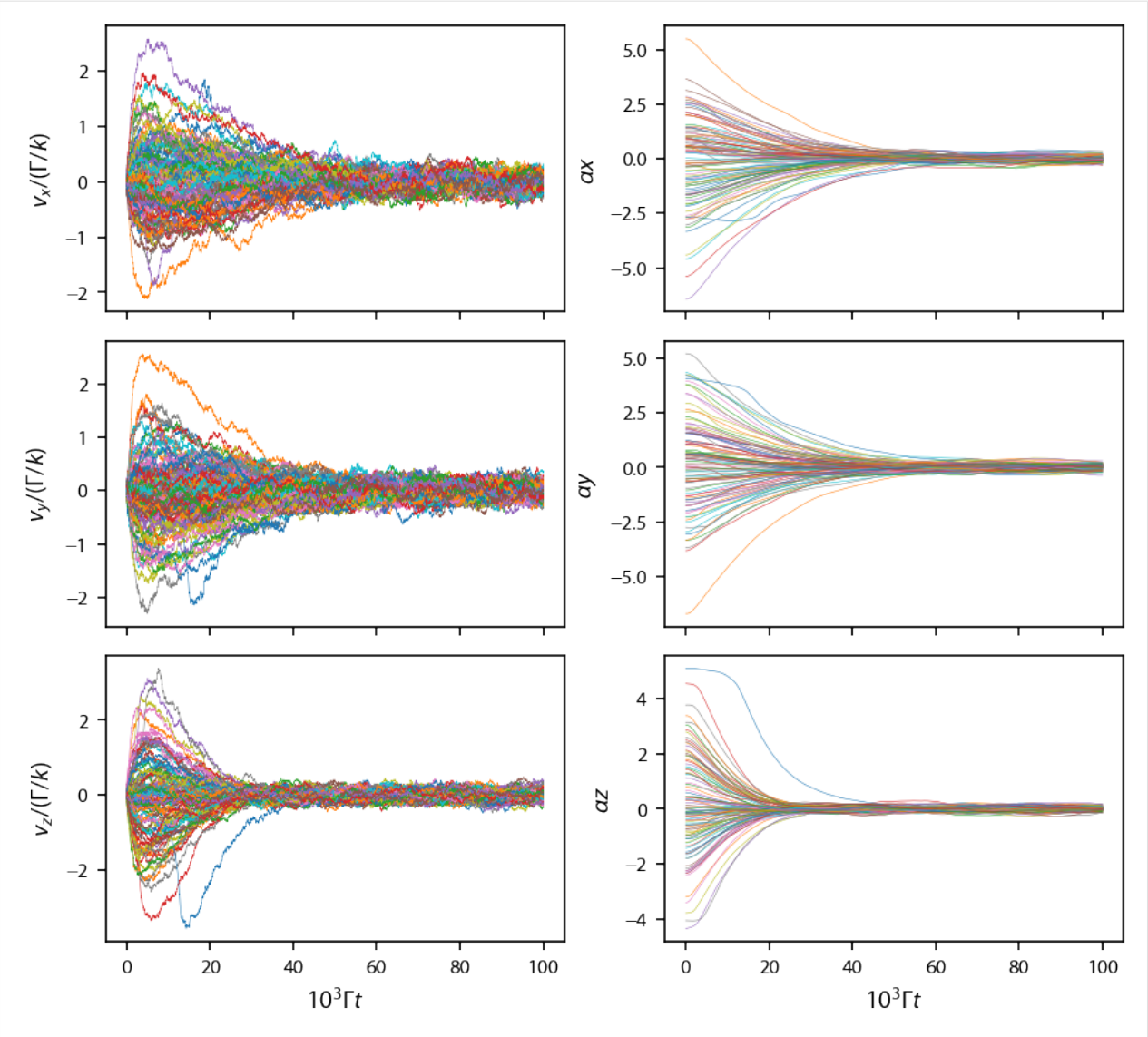
(continues on next page)

(continued from previous page)

```
    for ii in range(3):
        ax[ii, 0].plot(sol.t/1e3, sol.v[ii], linewidth=0.25)
        ax[ii, 1].plot(sol.t/1e3, sol.r[ii]*alpha, linewidth=0.25)

"""for ax_i in ax[:, 0]:
    ax_i.set_ylim((-0.75, 0.75))
for ax_i in ax[:, 1]:
    ax_i.set_ylim((-4., 4.))"""
for ax_i in ax[-1, :]:
    ax_i.set_xlabel('$10^3 \Gamma t$')
for jj in range(2):
    for ax_i in ax[jj, :]:
        ax_i.set_xticklabels('')
for ax_i, lbl in zip(ax[:, 0], ['x','y','z']):
    ax_i.set_ylabel('$v_' + lbl + '/(\Gamma/k)$')
for ax_i, lbl in zip(ax[:, 1], ['x','y','z']):
    ax_i.set_ylabel('$\\alpha ' + lbl + '$')

fig.subplots_adjust(left=0.1, bottom=0.08, wspace=0.22)
```



Reconstruct the force:

[14]:
```
for sol in sols:
    sol.F, sol.f_laser, sol.f_laser_q, sol.f_mag = obe.force(sol.r, sol.t, sol.rho,␣
↪return_details=True)
```

Concatenate all the positions and velocities and forces:

[15]:
```
allr = np.concatenate([sol.r[:, 500:].T for sol in sols]).T
allv = np.concatenate([sol.v[:, 500:].T for sol in sols]).T
allF = np.concatenate([sol.F[:, 500:].T for sol in sols]).T
```

Try to simulate what an image might look like (but we have to make it far more grainy because we have far fewer atoms):

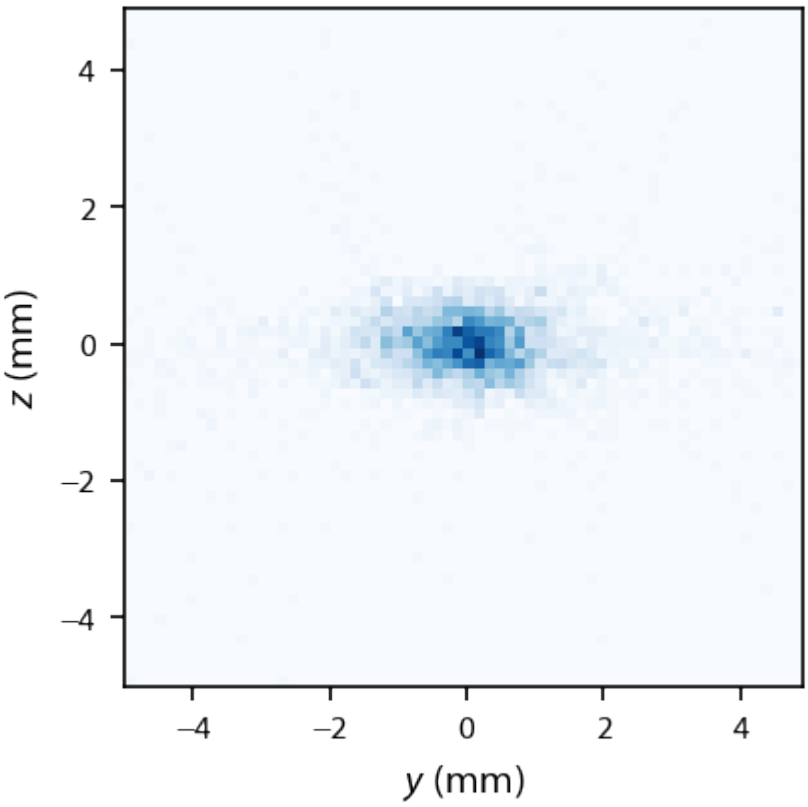
[19]:
```
k = np.pi/2/780E-6
img, y_edges, z_edges = np.histogram2d(allr[1, ::100]/k, allr[2, ::100]/k, bins=[np.
↪arange(-5., 5.01, 0.15), np.arange(-5., 5.01, 0.15)])
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(1, 1)
im = ax.imshow(img.T, origin='bottom',
               extent=(np.amin(y_edges), np.amax(y_edges),
                       np.amin(z_edges), np.amax(z_edges)),
               cmap='Blues',
               aspect='equal')

ax.set_xlabel('$y$ (mm)')
ax.set_ylabel('$z$ (mm)');
```



LetâĂŹs evaluate the temperature as a function of time:

[20]:
```
t_eval = kwargs['t_eval']
vs = np.nan*np.zeros((len(sols), 3, len(t_eval)))
for v, sol in zip(vs, sols):
    v[:, :sol.v.shape[1]] = sol.v

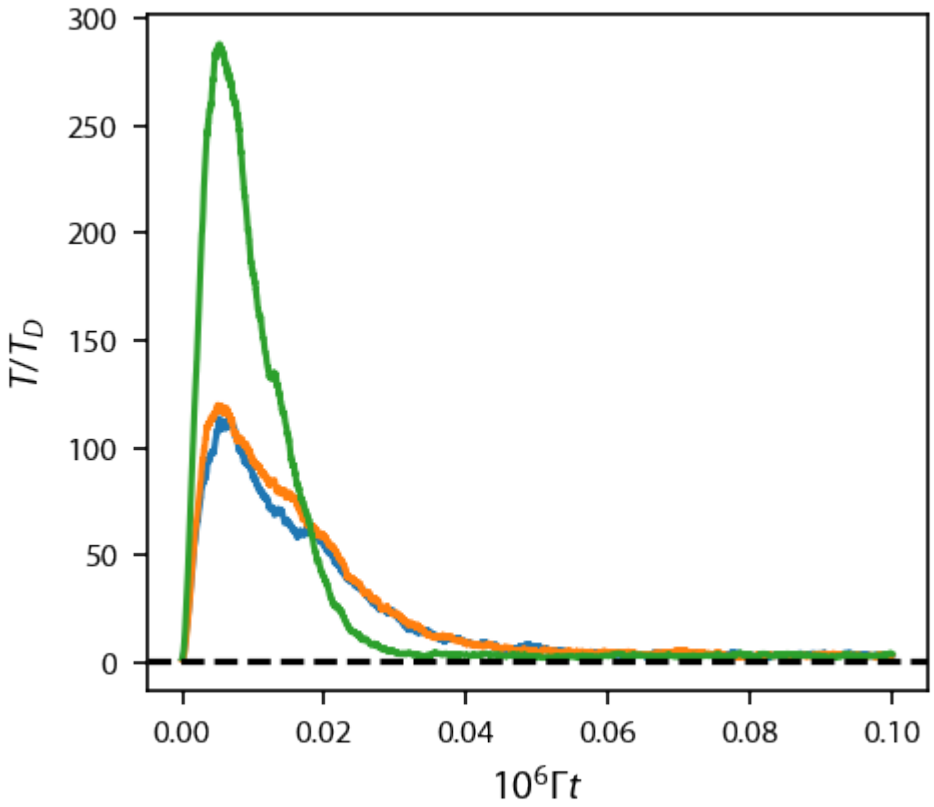
sigma_v = np.nanstd(vs, axis=0)
sigma_v.shape

fig, ax = plt.subplots(1, 1)
ax.plot(t_eval*1e-6, 2*sigma_v.T**2*hamiltonian.mass)
ax.axhline(1, color='k', linestyle='--')
ax.set_ylabel('$T/T_D$')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('$10^6 \Gamma t$');
```



[21]:
```
# Make a bunch of bins:
xb = np.arange(-0.5, 0.5001, 0.05)
lbls = ['x', 'y', 'z']
fig, ax = plt.subplots(1, 3, figsize=(6.5, 2.75))

for ii, lbl in enumerate(lbls):
    # Make the histogram:
    ax[ii].hist(vs[:, ii, 2500::500].flatten(), bins=xb)

    # Extract the data:
    x = xb[:-1] + np.diff(xb)/2
    y = np.histogram(vs[:, ii, 2500::500].flatten(), bins=xb)[0]

    # Fit it:
    model = lmfit.models.GaussianModel()
    result = model.fit(y, x=x)

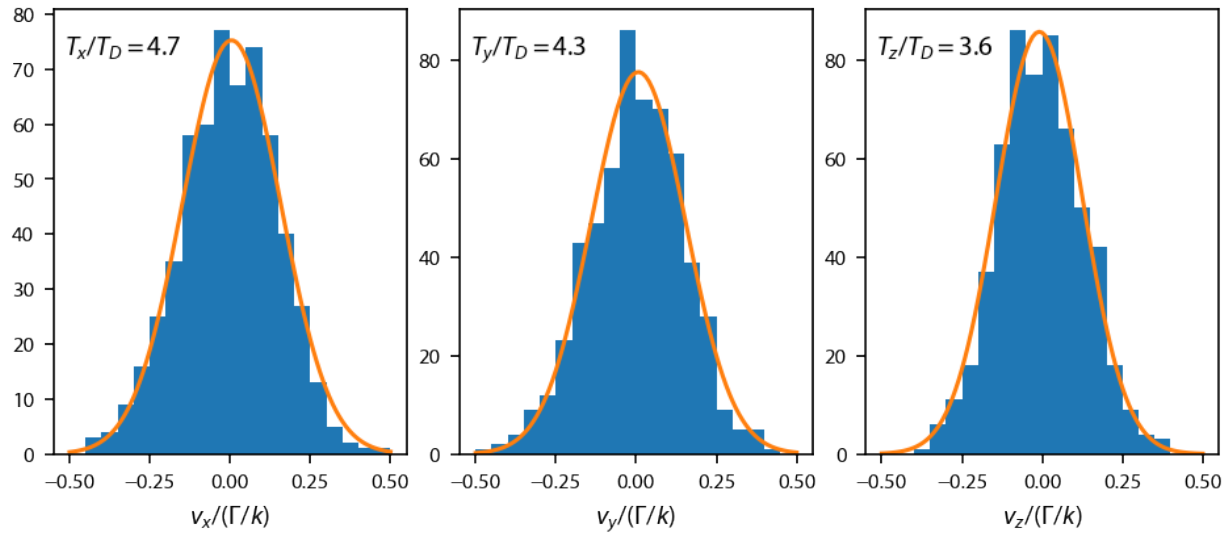
    # Plot up the fit:
    x_fit = np.linspace(-0.5, 0.5, 101)
    ax[ii].plot(x_fit, result.eval(x=x_fit))
    ax[ii].set_xlabel('$v_%s/(\Gamma/k)$'%lbl)
```

(continues on next page)

(continued from previous page)

```
    # Add the temperature
    plt.text(0.03, 0.9,
             '$T_%s/T_D = %.1f$'%(lbl, 2*result.best_values['sigma']**2*hamiltonian.
↪mass),
             transform=ax[ii].transAxes)

fig.subplots_adjust(left=0.07, wspace=0.15)
```



3.3.5 An $F \to F+1$ MOT with small ground state g-factor

This example covers calculating the forces in a type-I, three-dimensional MOT in a variety of different ways and comparing the various results.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
```

Define the problem

WeâĂŹll define the laser beams, magnetic field, and hamiltonian that weâĂŹll use:

[2]:
```
# Laser beams parameters:
det = -1.5
alpha = 1.0
s = 1.0

# Define the laser beams:
laserBeams = pylcp.laserBeams(
    [{'kvec':np.array([1, 0, 0]), 's':s, 'pol':-1, 'delta':det},
     {'kvec':np.array([-1, 0, 0]), 's':s, 'pol':-1, 'delta':det}],
    beam_type=pylcp.infinitePlaneWaveBeam
```

(continues on next page)

(continued from previous page)

```
)

# Actual linear gradient:
linGrad = pylcp.magField(lambda R: np.array([-alpha*R[0], np.zeros(R[1].shape),
                                             np.zeros(R[2].shape)]))

# Define the atomic Hamiltonian:
Fl = 1
Hg, muq_g = pylcp.hamiltonians.singleF(F=Fl, gF=0, muB=1)
He, muq_e = pylcp.hamiltonians.singleF(F=Fl+1, gF=1/(Fl+1), muB=1)

dijq =  pylcp.hamiltonians.dqij_two_bare_hyperfine(1, 2)

ham = pylcp.hamiltonian(Hg, He, muq_g, muq_e, dijq)
```

Calculate equilibrium forces

WeâĂŹll do this with two governing equations again, one `rateeq` and one `heuristiceq` for comparison.

[3]:
```
rateeq = pylcp.rateeq(laserBeams, linGrad, ham, svd_eps=1e-10, include_mag_forces=False)
heuristiceq = pylcp.heuristiceq(laserBeams, linGrad)
```

Let now define a coordinate system over which to calculate and run both:

[93]:
```
x = np.arange(-20+1e-9, 20.1, 0.1)
v = np.arange(-20, 20.1, 0.1)

X, V = np.meshgrid(x, v)

# Define the three-vectors used in the calculation:
Rvec = np.array([X, np.zeros(X.shape), np.zeros(X.shape)])
Vvec = np.array([V, np.zeros(V.shape), np.zeros(V.shape)])

heuristiceq.generate_force_profile(Rvec, Vvec, name='Fx')
rateeq.generate_force_profile(Rvec, Vvec, name='Fx', progress_bar=True,
                              default_axis=np.array([1., 0., 0.]))
```

```
Completed in 2:02.
```

Plot up the resulting forces:

[36]:
```
F0 = heuristiceq.profile['Fx'].F[0]
F2 = rateeq.profile['Fx'].F[0]

# Now plot it up:
fig, ax = plt.subplots(nrows=1, ncols=2, num="Force F=2->F=3", figsize=(6.5,2.75))
im1 = ax[1].imshow(F2,extent=(np.amin(X[0,:]), np.amax(X[0,:]),
                              np.amin(V[:,0]), np.amax(V[:,0])),
                   origin='bottom',
                   aspect='auto')
cb1 = plt.colorbar(im1)
cb1.set_label('$f/(\hbar k\Gamma)$')
```

(continues on next page)

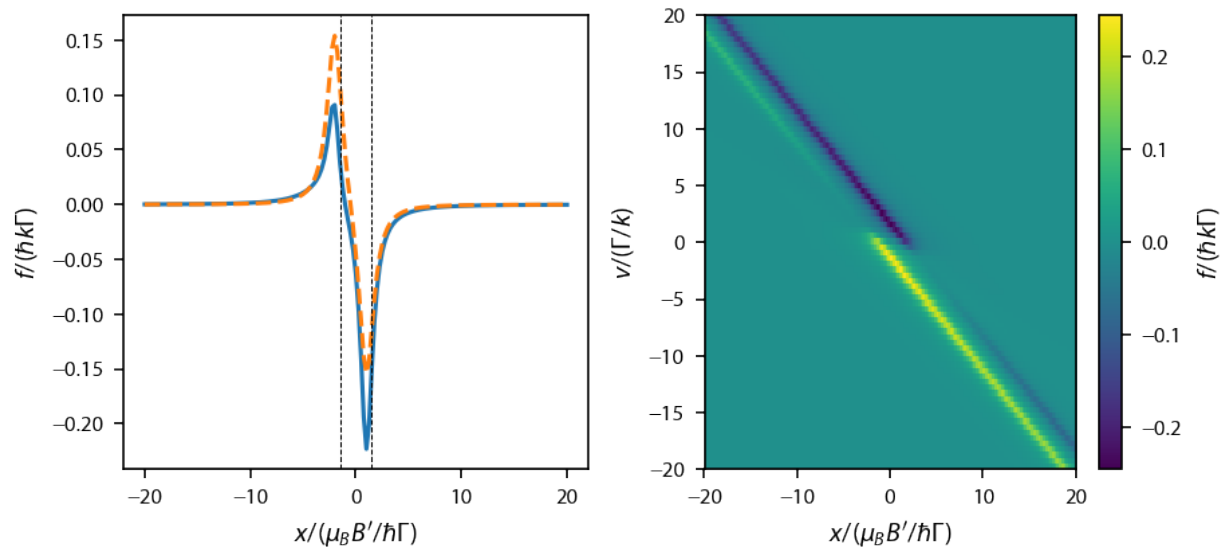
(continued from previous page)

```
ax[1].set_xlabel('$x/(\mu_B B\'/\hbar \Gamma)$')
ax[1].set_ylabel('$v/(\Gamma/k)$')

ax[0].plot(x, F2[int(np.ceil(F2.shape[0]/2)),:], '-', color='C0')
ax[0].plot(x, F0[int(np.ceil(F0.shape[0]/2)),:], '--', color='C1')
ax[0].set_ylabel('$f/(\hbar k \Gamma)$')
ax[0].set_xlabel('$x/(\mu_B B\'/\hbar \Gamma)$')

ax[0].axvline(-det, color='k', linewidth=0.5, linestyle='--')
ax[0].axvline(+det, color='k', linewidth=0.5, linestyle='--')

fig.subplots_adjust(wspace=0.25)
```



Why is the oppositely directly force dissappearing at large magnetic field?

It turns out that we are getting an interesting effect when the non-cycling transition from other beam starts to affect the population in the cycling transition ground state. LetâĂŹs start by going figuring out the line to go along for to stay in resonance with the rightward going beam.

LetâĂŹs first look at this along lines that are going along $\hat{x}$ for various v:

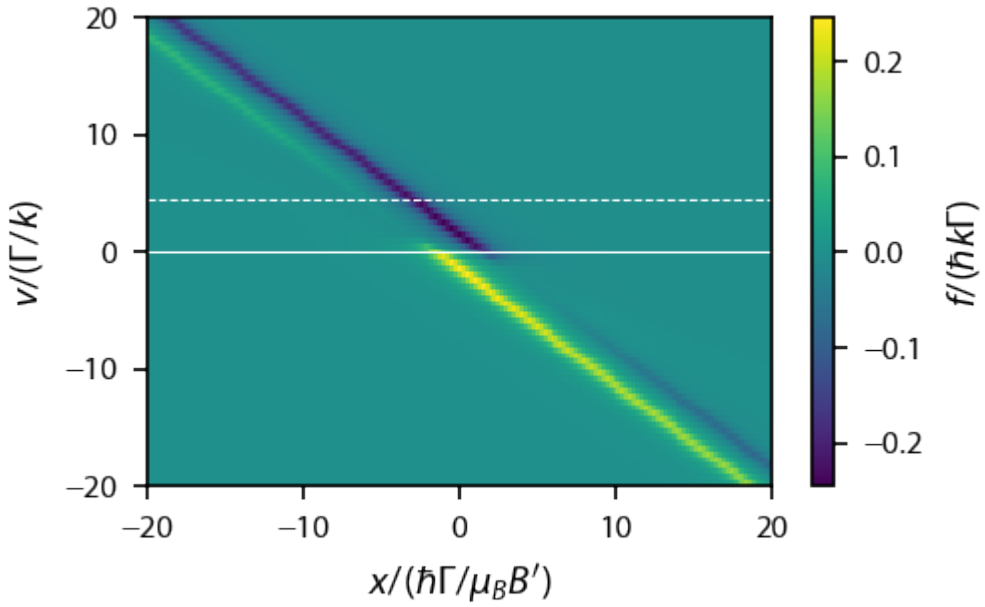
[168]:
```
v_inds = [200, 245]

fig, ax = plt.subplots(1, 1, figsize=(3.25, 2.))
im1 = ax.imshow(F2,extent=(np.amin(X[0,:]), np.amax(X[0,:]),
                           np.amin(V[:,0]), np.amax(V[:,0])),
                origin='bottom',
                aspect='auto')
cb1 = plt.colorbar(im1, ax=ax)
ax.axhline(rateeq.profile['Fx'].V[0, v_inds[0], 0], color='w', linestyle='-',␣
↪linewidth=0.5)
ax.axhline(rateeq.profile['Fx'].V[0, v_inds[1], 0], color='w', linestyle='--',␣
↪linewidth=0.5)
```

(continues on next page)

(continued from previous page)

```
cb1.set_label('$f/(\hbar k \Gamma)$')
ax.set_xlabel('$x/(\hbar \Gamma/\mu_B B\')$')
ax.set_ylabel('$v/(\Gamma/k)$')
ax.set_ylim((np.amin(v), np.amax(v)))
fig.subplots_adjust(left=0.15, bottom=0.2, right=0.95)
```



[171]:
```
fig, ax = plt.subplots(2, 1, figsize=(3.25, 3))
ax_twin = np.array([ax_i.twinx() for ax_i in ax])

for jj, v_ind in enumerate(v_inds):
    [ax[jj].plot(x, np.concatenate((rateeq.profile['Fx'].Neq[v_ind, x<0, ii],
                                    rateeq.profile['Fx'].Neq[v_ind, x>=0, 2-ii])),
                 '-', color='C%d'%ii, linewidth=0.75) for ii in range(3)];
    [ax_twin[jj].plot(x, np.concatenate((np.sum(rateeq.profile['Fx'].Rijl['g->e'][v_ind,␣
↪x<0, 0, ii, :], axis=1),
                                         np.sum(rateeq.profile['Fx'].Rijl['g->e'][v_ind,␣
↪x>=0, 0, 2-ii, :], axis=1))),
                 '--', color='C%d'%ii, linewidth=0.75) for ii in range(3)];
    [ax_twin[jj].plot(x, np.concatenate((np.sum(rateeq.profile['Fx'].Rijl['g->e'][v_ind,␣
↪x<0, 1, ii, :], axis=1),
                                         np.sum(rateeq.profile['Fx'].Rijl['g->e'][v_ind,␣
↪x>=0, 1, 2-ii, :], axis=1))),
                 '-.', color='C%d'%ii, linewidth=0.75) for ii in range(3)];

    #ax[jj].axvline(-rateeq.profile['Fx'].V[0, v_ind, 0]-det, color='k', linewidth=0.5,␣
↪linestyle='--')
    #ax[jj].axvline(-rateeq.profile['Fx'].V[0, v_ind, 0]+det, color='k', linewidth=0.5,␣
↪linestyle='--')

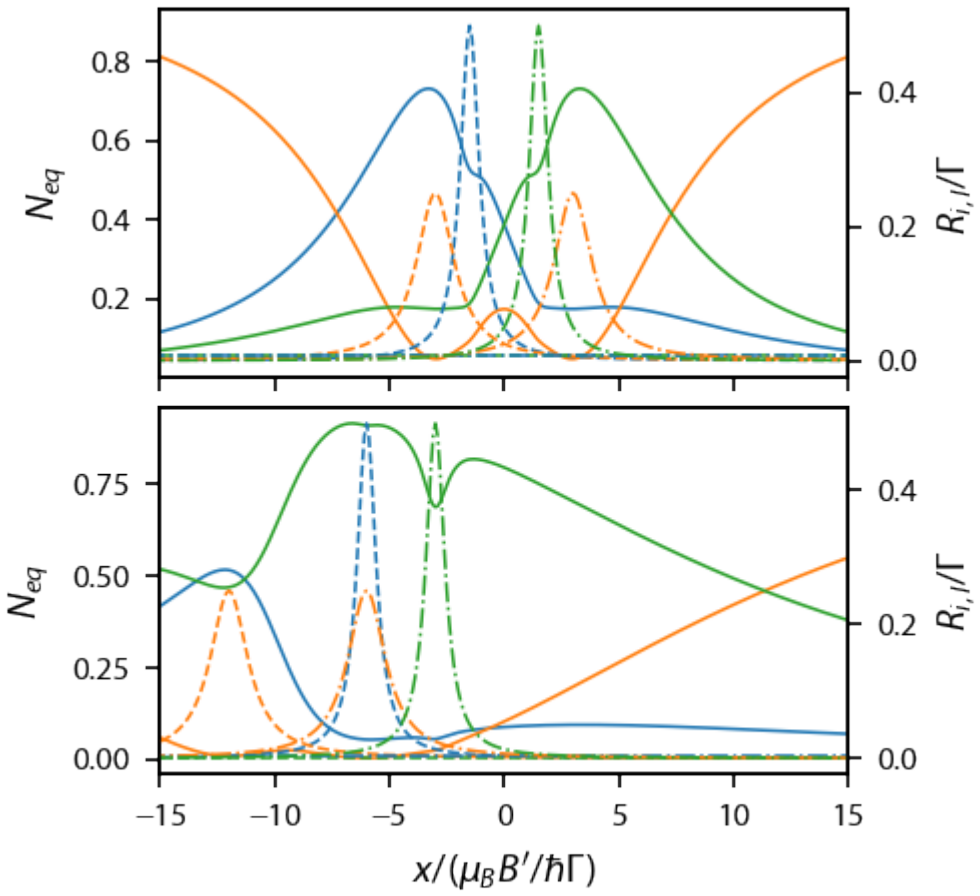
    ax[jj].set_xlim(-15, 15)
```

(continues on next page)

(continued from previous page)

```
[ax[ii].set_ylabel('$N_{eq}$') for ii in range(2)]
[ax_twin[ii].set_ylabel('$R_{i,l}/\Gamma$') for ii in range(2)]
ax[1].set_xlabel('$x/(\mu_B B\'/\hbar \Gamma)$')

ax[0].xaxis.set_ticklabels('');
fig.subplots_adjust(left=0.155, bottom=0.13, hspace=0.08, right=0.86)
```



Look along the line of resonance

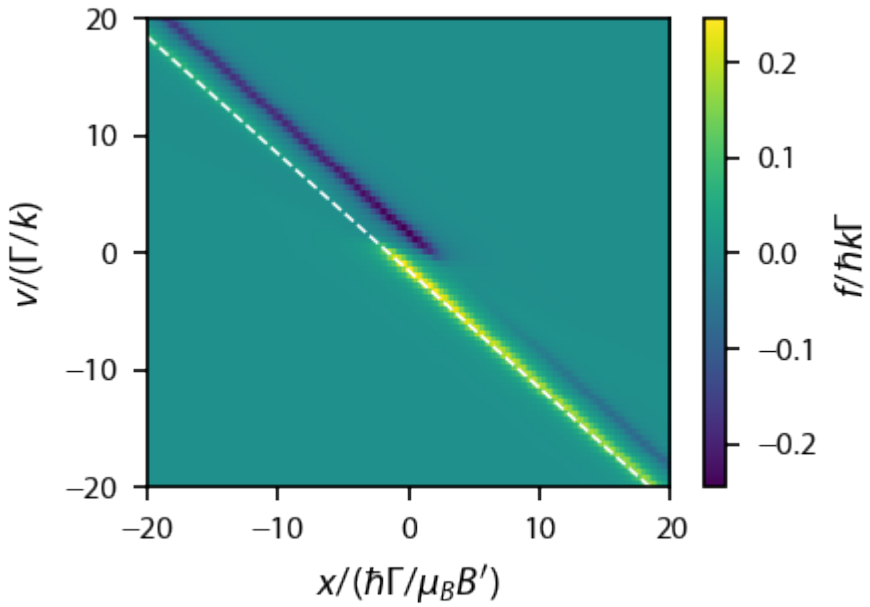
Firt, compute the line where the rightward going beam is always supposed to be resonant:

[88]:
```
fig, ax = plt.subplots(1, 1, figsize=(3.25, 2.))
im1 = ax.imshow(F2,extent=(np.amin(X[0,:]), np.amax(X[0,:]),
                           np.amin(V[:,0]), np.amax(V[:,0])),
                origin='bottom',
                aspect='auto')
cb1 = plt.colorbar(im1, ax=ax)
ax.plot(x, det - x, 'w--', linewidth=0.75)
cb1.set_label('$f/\hbar k \Gamma$')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('$x/(\hbar \Gamma/\mu_B B\')$')
ax.set_ylabel('$v/(\Gamma/k)$')
ax.set_ylim((np.amin(v), np.amax(v)))
fig.subplots_adjust(bottom=0.2, right=0.85)
```



Now, compute the parameters along that line:

[172]:
```
# Along the line v = delta - alpha x, we
v_path = det - alpha*x

rateeq.generate_force_profile([x, np.zeros(x.shape), np.zeros(x.shape)],
                              [v_path, np.zeros(x.shape), np.zeros(x.shape)],
                              name='res_path', default_axis=np.array([1., 0., 0.]))
```

Now, plot up the scattering rate and populations along that line:

[173]:
```
fig, ax = plt.subplots(2, 1, figsize=(3.25, 2.5))

for ii in range(2*Fl+1):
    ax[0].plot(x, np.concatenate((rateeq.profile['res_path'].Rijl['g->e'][x<0, 0, ii],␣
↪ii],
                                  rateeq.profile['res_path'].Rijl['g->e'][x>=0, 0, -1-ii],
↪ -1-ii])),
               color='C{0:d}'.format(ii+1), linewidth=0.5,
               label='$m_F= {0:d}\\rightarrow {1:d}$'.format(ii-1,ii-2))

for ii in range(2*Fl+1):
    ax[0].plot(x, np.concatenate((rateeq.profile['res_path'].Rijl['g->e'][x<0, 1, ii],␣
↪ii+2],
                                  rateeq.profile['res_path'].Rijl['g->e'][x>=0, 1, -1-ii,
↪ -1-ii-2])),
```

(continues on next page)

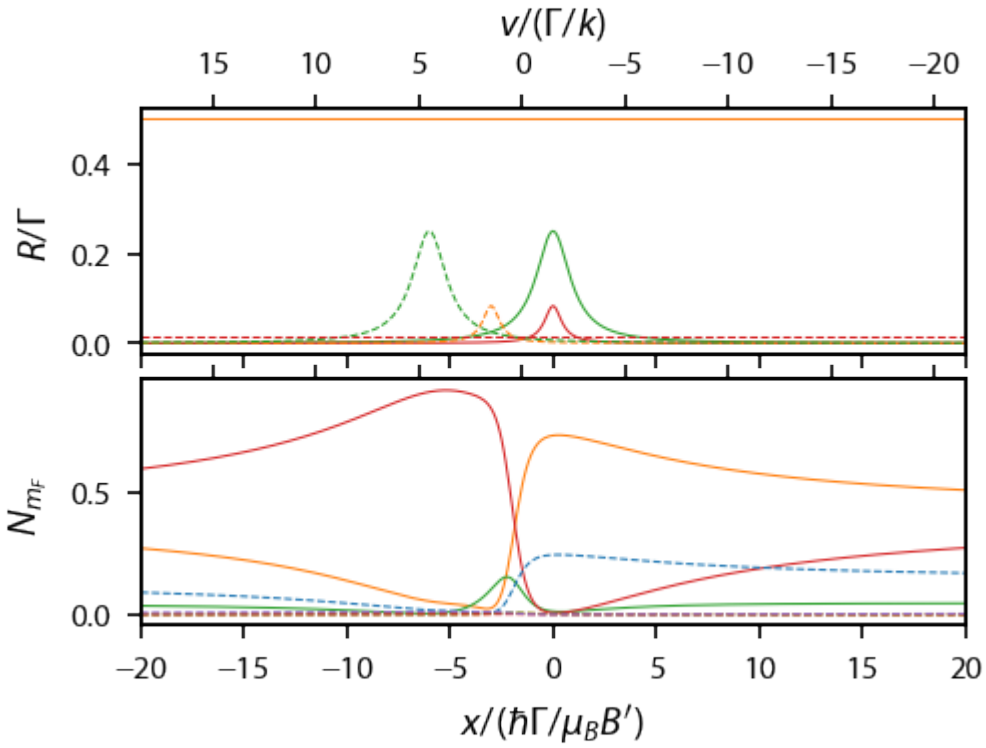
(continued from previous page)

```
                '--', color='C{0:d}'.format(ii+1), linewidth=0.5,
                label='$m_F= {0:d}\\rightarrow {1:d}$'.format(ii-1,ii))

for ii in range(2*Fl+1):
    ax[1].plot(x, np.concatenate((rateeq.profile['res_path'].Neq[x<0, ii],
                                  rateeq.profile['res_path'].Neq[x>=0, ham.ns[0]-ii-1])),
                color='C{0:d}'.format(ii+1), linewidth=0.5)

for ii in range(2*(Fl+1)+1):
    ax[1].plot(x, np.concatenate((rateeq.profile['res_path'].Neq[x<0, ham.ns[0]+ii],
                                   rateeq.profile['res_path'].Neq[x>=0, -1-ii])),
                '--', color='C{0:d}'.format(ii), linewidth=0.5)

[ax_i.set_xlim(-20, 20) for ax_i in ax];
ax_twin = [ax[ii].twiny() for ii in range(2)]
[ax_twin[ii].set_xlim(det - alpha*np.array(ax[ii].get_xlim())) for ii in range(2)];
ax_twin[1].xaxis.set_ticklabels('')
ax[0].xaxis.set_ticklabels('')
ax[1].set_xlabel('$x/(\hbar\Gamma/\mu_B B\')$')
ax_twin[0].set_xlabel('$v/(\Gamma/k)$')
ax[0].set_ylabel('$R/\Gamma$')
ax[1].set_ylabel('$N_{m_F}$')
fig.subplots_adjust(left=0.135, top=0.85)
```



3.3.6 General $F \to F'$ 3D MOTs

This example covers calculating the forces in both type-I and type-II three-dimensional MOTs. In particular, we are trying to reproduce the results of M.R. Tarbutt, âĂIJMagneto-optical trapping forces for atoms and molecules with complex level structuresâĂİ *New Journal of Physics* **17**, 015007 (2015) http://dx.doi.org/10.1088/1367-2630/17/1/015007

We also take a look at what a subset of these MOTs look like as a function of both **v** and **r** as well.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
```

First, can we reproduce all the figures?

In the reference above, the forces for a variety of $F \to F'$ MOTs are calculated for a variety of different upper state g_u-factors and three lower state g_l-factors.

Note that we turn of the magnetic forces for `rateeq` because we are using the same unit system as that in `01_F0_to_F1_1D_MOT_capture.ipynb`. Here, we are not calculating any dynamics.

[2]:
```
det = -1.0
s = 1.0

# Make a z-axis
z = np.linspace(1e-10, 5, 100)

# Make the figure
fig, ax = plt.subplots(3, 3,figsize=(9, 6.5))

# Define the lasers:
laserBeams = pylcp.conventional3DMOTBeams(delta=det, s=s,
                                          beam_type=pylcp.infinitePlaneWaveBeam)
# Define the magnetic field
magField = pylcp.quadrupoleMagneticField(1.)

Fs = [[1, 2], [1, 1], [2, 1]] # The Fs we want to run through
gl = np.array([0., 1., -1.])  # The lower state g factors to run
gu = np.arange(0.1, 1.1, 0.1) # The upper state g factors to run
for jj, (Fl, Fu) in enumerate(Fs):
    for ii, gl_i in enumerate(gl):
        for gu_i in gu:
            # Case 1: F=1 -> F=2
            Hg, Bgq = pylcp.hamiltonians.singleF(F=Fl, gF=gl_i, muB=1)
            if Fu<Fl: # Reverse the upper state g-factor to get a confining force
                He, Beq = pylcp.hamiltonians.singleF(F=Fu, gF=-gu_i, muB=1)
            else:
                He, Beq = pylcp.hamiltonians.singleF(F=Fu, gF=gu_i, muB=1)

            dijq = pylcp.hamiltonians.dqij_two_bare_hyperfine(Fl, Fu)

            hamiltonian = pylcp.hamiltonian(Hg, He, Bgq, Beq, dijq)
```

(continues on next page)

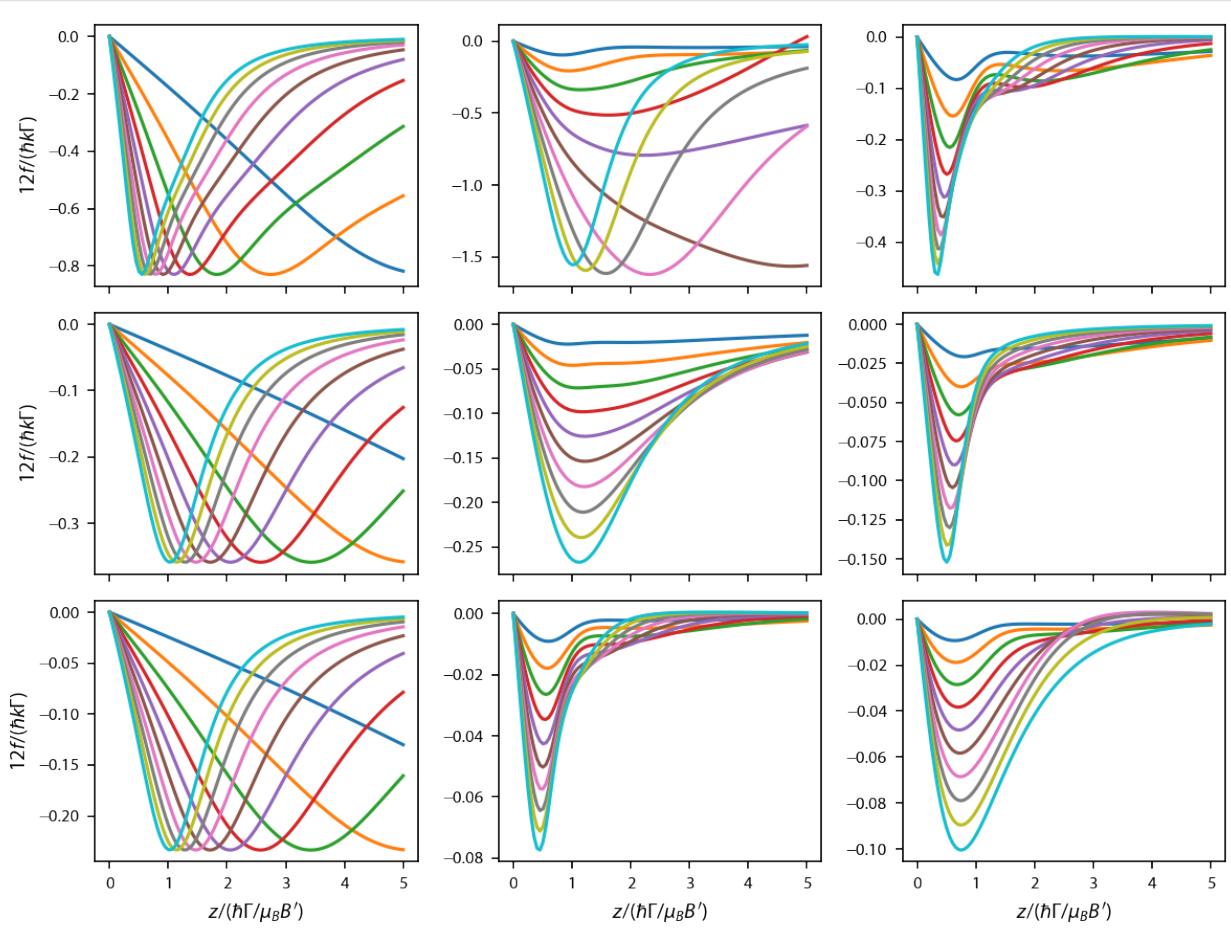
(continued from previous page)

```
            rateeq = pylcp.rateeq(laserBeams, magField, hamiltonian,
                                  svd_eps=1e-10, include_mag_forces=False)

            rateeq.generate_force_profile(
                [np.zeros(z.shape), np.zeros(z.shape), z],
                [np.zeros(z.shape), np.zeros(z.shape), np.zeros(z.shape)],
                name='Fz')
            Fz = rateeq.profile['Fz'].F[2]

            ax[jj, ii].plot(z, 12*Fz)
            if jj<2:
                ax[jj, ii].xaxis.set_ticklabels([])

for ii in range(3):
    ax[ii, 0].set_ylabel('$12f/(\hbar k \Gamma)$')
    ax[2, ii].set_xlabel('$z/(\hbar \Gamma/\mu_B B\')$')
fig.subplots_adjust(wspace=0.25)
```



The rows correspond to (top to bottom) $F = 1 \rightarrow F' = 2$, $F = 1 \rightarrow F' = 1$, and $F = 2 \rightarrow F' = 1$, respectively. The columns correspond to $g_l = 0., 1., -1.$, from left to right. Note that we use normalized units here, unlike in the reference, which uses lab units. They also use a Gaussian beam with a width defined in lab units, while we use infinite plane-wave beams. That accounts for the slight differences at large z.

Now letâĂŹs look at the force in phase space:

WeâĂŹll constrict ourselves to $g_l = 0$ and $g_u = 1/F'$.

[3]:
```
x = np.arange(-5, 5.1, 0.2)
v = np.arange(-5, 5.1, 0.2)

X, V = np.meshgrid(x, v)

det = -3.0
alpha = 1.0
s = 2.0

# Definte laser beams and magnetic field again:
laserBeams = pylcp.conventional3DMOTBeams(delta=det, s=s,
                                          beam_type=pylcp.infinitePlaneWaveBeam)
magField = pylcp.quadrupoleMagneticField(alpha)

Fs = [[0, 1], [1, 2], [1, 1], [2, 1]] # The Fs we want to run through
plot_inds = [(0, 0), (0, 1), (1, 0), (1, 1)] # a map from Fs index to plot index
fig, ax = plt.subplots(2, 2, num="Comparison of F_z") # make the plot

for jj, (Fl, Fu) in enumerate(Fs):
    # Generate the pieces of the hamiltonian:
    Hg, Bgq = pylcp.hamiltonians.singleF(F=Fl, gF=0, muB=1)
    if Fu<Fl: # Reverse the upper state g-factor to get a confining force
        He, Beq = pylcp.hamiltonians.singleF(F=Fu, gF=-1/Fu, muB=1)
    else:
        He, Beq = pylcp.hamiltonians.singleF(F=Fu, gF=1/Fu, muB=1)
    dijq = pylcp.hamiltonians.dqij_two_bare_hyperfine(Fl, Fu)

    # Put all the pieces together into the full Hamiltonian:
    hamiltonian = pylcp.hamiltonian(Hg, He, Bgq, Beq, dijq)

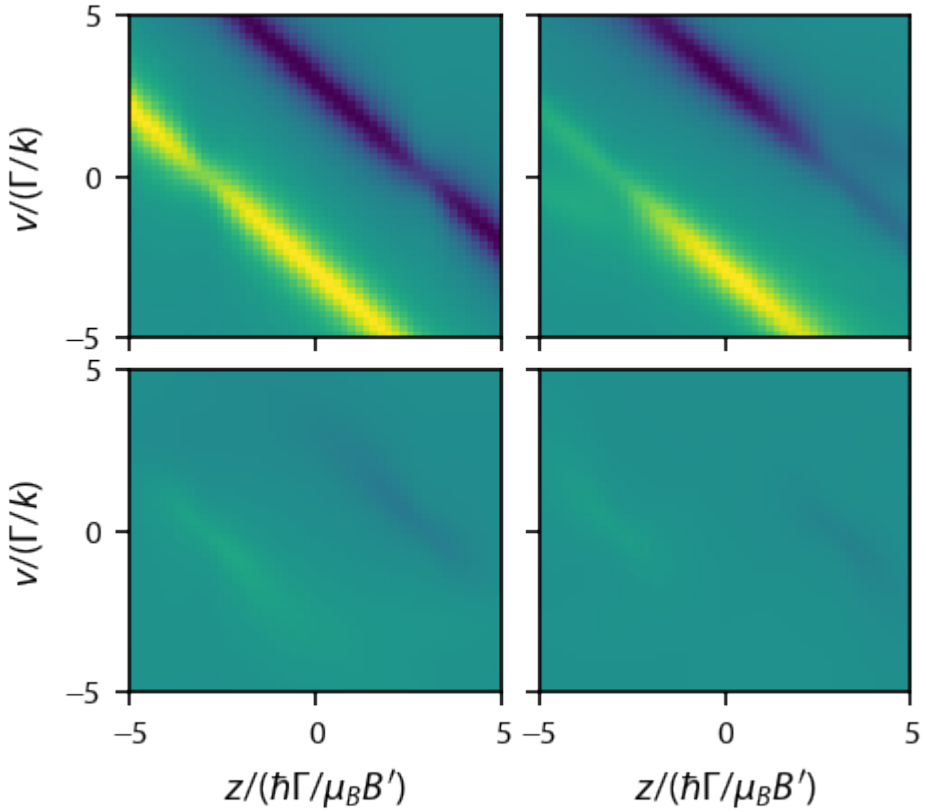
    # Make the rateequations and generate the force profile:
    rateeq = pylcp.rateeq(laserBeams, magField, hamiltonian,
                          svd_eps=1e-10, include_mag_forces=False)
    rateeq.generate_force_profile(
        [np.zeros(X.shape), np.zeros(X.shape), X],
        [np.zeros(V.shape), np.zeros(V.shape), V],
        name='Fz')
    Fz0to1 = rateeq.profile['Fz'].F[2]

    #  Plot it up
    ax[plot_inds[jj]].imshow(
        Fz0to1,
        extent=(np.amin(X[0, :]), np.amax(X[0, :]),
                np.amin(V[:, 0]), np.amax(V[:, 0])),
        origin='bottom',
        aspect='auto',
        vmin=-0.3,
        vmax=0.3
    )
```

(continues on next page)

(continued from previous page)

```
# Put in axis labels, and turn off superfluous tick labels:
for ii in range(2):
    ax[ii, 1].yaxis.set_ticklabels([])
    ax[ii, 0].set_ylabel('$v/(\Gamma/k)$')
    ax[0, ii].xaxis.set_ticklabels([])
    ax[1, ii].set_xlabel('$z/(\hbar \Gamma/\mu_B B\')$')
```



3.3.7 Real atoms in a MOT

This example covers calculating the forces in various type-I and type-II three-dimensional MOT with real atoms and comparing the results.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants as cts
import pylcp
```

^{7}Li: compare the D_2 line with a basic $F = 2 \rightarrow F = 3$ transition

WeâĂŹll do this specifically for ^{7}Li. As usual, we first define the Hamiltonian, laser Beams, and magnetic field. We start with the full D_2 line. Note that we need to specify a repump laser, which, for ^{7}Li, generally has the same red detuning as the main cooling beam.

[2]:
```
det = -2.0
alpha = 1.0
s = 1.0

# Define the atomic Hamiltonian for 7Li:
atom = pylcp.atom("7Li")
H_g_D2, mu_q_g_D2 = pylcp.hamiltonians.hyperfine_coupled(
    atom.state[0].J, atom.I, atom.state[0].gJ, atom.gI,
    atom.state[0].Ahfs/atom.state[2].gammaHz, Bhfs=0, Chfs=0,
    muB=1)
H_e_D2, mu_q_e_D2 = pylcp.hamiltonians.hyperfine_coupled(
    atom.state[2].J, atom.I, atom.state[2].gJ, atom.gI,
    Ahfs=atom.state[2].Ahfs/atom.state[2].gammaHz,
    Bhfs=atom.state[2].Bhfs/atom.state[2].gammaHz, Chfs=0,
    muB=1)

dijq_D2 = pylcp.hamiltonians.dqij_two_hyperfine_manifolds(
    atom.state[0].J, atom.state[2].J, atom.I)

E_e_D2 = np.unique(np.diagonal(H_e_D2))
E_g_D2 = np.unique(np.diagonal(H_g_D2))

hamiltonian_D2 = pylcp.hamiltonian(H_g_D2, H_e_D2, mu_q_g_D2, mu_q_e_D2, dijq_D2)

# Now, we need to sets of laser beams -> one for F=1->2 and one for F=2->3:
laserBeams_cooling_D2 = pylcp.conventional3DMOTBeams(
    s=s, delta=(E_e_D2[0] - E_g_D2[1]) + det)
laserBeams_repump_D2 = pylcp.conventional3DMOTBeams(
    s=s, delta=(E_e_D2[1] - E_g_D2[0]) + det)
laserBeams_D2 = laserBeams_cooling_D2 + laserBeams_repump_D2

magField = pylcp.quadrupoleMagneticField(alpha)
```

Construct the rate equations for the full D_2 line and calculate a force profile:

[3]:
```
x = np.arange(-5, 5.1, 0.2)
v = np.arange(-5, 5.1, 0.2)

dx = np.mean(np.diff(x))
dv = np.mean(np.diff(v))

X, V = np.meshgrid(x, v)

# Define the trap:
trap_D2 = pylcp.rateeq(
    laserBeams_D2, magField, hamiltonian_D2,
    include_mag_forces=False
```

(continues on next page)

(continued from previous page)

```
)
trap_D2.generate_force_profile(
    [np.zeros(X.shape), np.zeros(X.shape), X],
    [np.zeros(V.shape), np.zeros(V.shape), V],
    name='Fz')
FzLi_D2 = trap_D2.profile['Fz'].F[2]
```

Now, repeat the same procedure for the simpler $F = 2 \rightarrow F' = 3$ transition, making sure we keep the g-factors the same:

[4]:
```
# Define the atomic Hamiltonian for F-> 2 to 3:
H_g_23, mu_q_g_23 = pylcp.hamiltonians.singleF(F=2, gF=1/2, muB=1)
H_e_23, mu_q_e_23 = pylcp.hamiltonians.singleF(F=3, gF=2/3, muB=1)

dijq_23 = pylcp.hamiltonians.dqij_two_bare_hyperfine(2, 3)

hamiltonian_23 = pylcp.hamiltonian(H_g_23, H_e_23, mu_q_g_23, mu_q_e_23, dijq_23)

# Define the laser beams for 2->3
laserBeams_23 = pylcp.conventional3DMOTBeams(s=s, delta=det)

# Make the trap for 2->3
trap_23 = pylcp.rateeq(
    laserBeams_23, magField, hamiltonian_23, include_mag_forces=False)
trap_23.generate_force_profile(
    [np.zeros(X.shape), np.zeros(X.shape), X],
    [np.zeros(V.shape), np.zeros(V.shape), V],
    name='Fz')
Fz2to3 = trap_23.profile['Fz'].F[2]
```

Plot up the results:

[5]:
```
fig, ax = plt.subplots(2, 2, figsize=(1.5*3.25, 1.5*2.75))
ax[0, 0].imshow(FzLi_D2, origin='bottom',
                extent=(np.amin(x)-dx/2, np.amax(x)+dx/2,
                        np.amin(v)-dv/2, np.amax(v)+dv/2),
                aspect='auto')
ax[0, 1].imshow(Fz2to3, origin='bottom',
                extent=(np.amin(x)-dx/2, np.amax(x)+dx/2,
                        np.amin(v)-dv/2, np.amax(v)+dv/2),
                aspect='auto')
ax[1, 0].plot(X[int(X.shape[0]/2), :],
              FzLi_D2[int(X.shape[0]/2), :])
ax[1, 0].plot(X[int(X.shape[0]/2), :],
              Fz2to3[int(X.shape[0]/2), :], '--',
              linewidth=0.75)
ax[1, 1].plot(V[:, int(X.shape[1]/2)+1],
              FzLi_D2[:, int(X.shape[1]/2)+1], label='$^7$Li')
ax[1, 1].plot(V[:, int(X.shape[1]/2)+1],
              Fz2to3[:, int(X.shape[1]/2)+1], '--',
              label='$F=2 \\rightarrow F\'=3$',
              linewidth=0.75)
```

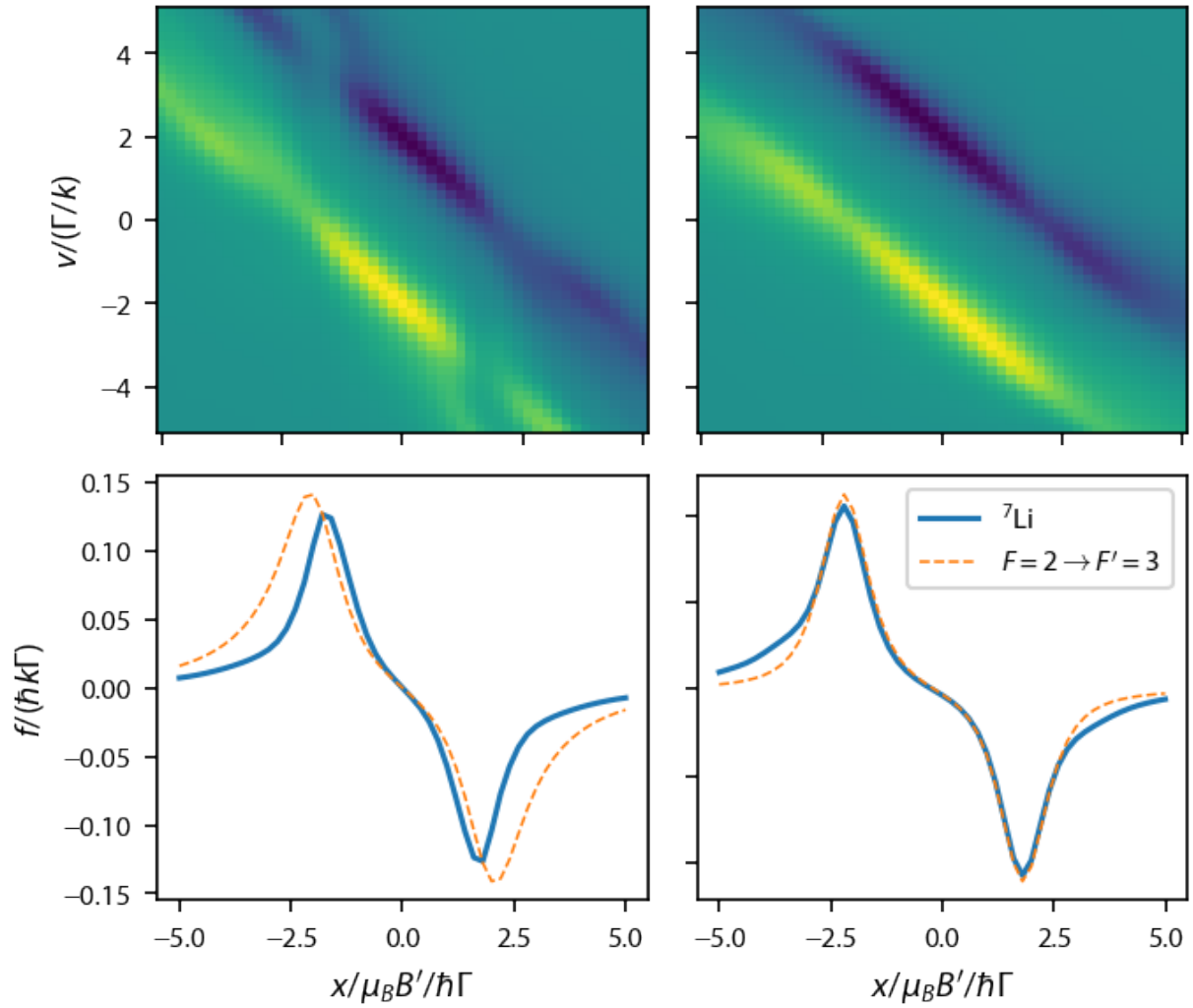
(continues on next page)

(continued from previous page)

```
ax[1, 1].legend(fontsize=8)

[ax[ii, 1].yaxis.set_ticklabels('') for ii in range(2)]
[ax[0, ii].xaxis.set_ticklabels('') for ii in range(2)]

ax[0, 0].set_ylabel('$v/(\Gamma/k)$')
ax[1, 0].set_ylabel('$f/(\hbar k \Gamma)$')
ax[1, 0].set_xlabel('$x/\mu_B B\'/\hbar\Gamma$')
ax[1, 1].set_xlabel('$x/\mu_B B\'/\hbar\Gamma$');
```



Now, it seems to me that because of the un-resolved hyperfine structure in the excited state that is inherent in 7Li, the repump, which drives $F = 1 \rightarrow F' = 2$ transitions will also contribute the trapping and cause most of the difference between the $F = 2 \rightarrow F = 3$ and the full Hamiltonian calculation.

Switch to ^{87}Rb

By switching to 87Rb we can bring the repump to resonance, and turn down its intensity to 1/100 of the main cooling light.

[6]:
```
atom = pylcp.atom("87Rb")
H_g_D2, mu_q_g_D2 = pylcp.hamiltonians.hyperfine_coupled(
    atom.state[0].J, atom.I, atom.state[0].gJ, atom.gI,
    atom.state[0].Ahfs/atom.state[2].gammaHz, Bhfs=0, Chfs=0,
    muB=1)
H_e_D2, mu_q_e_D2 = pylcp.hamiltonians.hyperfine_coupled(
    atom.state[2].J, atom.I, atom.state[2].gJ, atom.gI,
    Ahfs=atom.state[2].Ahfs/atom.state[2].gammaHz,
    Bhfs=atom.state[2].Bhfs/atom.state[2].gammaHz, Chfs=0,
    muB=1)
mu_q_g_D2[1]
dijq_D2 = pylcp.hamiltonians.dqij_two_hyperfine_manifolds(
    atom.state[0].J, atom.state[2].J, atom.I)

E_e_D2 = np.unique(np.diagonal(H_e_D2))
E_g_D2 = np.unique(np.diagonal(H_g_D2))

hamiltonian_D2 = pylcp.hamiltonian(H_g_D2, H_e_D2, mu_q_g_D2, mu_q_e_D2, dijq_D2)

# Now, we need to sets of laser beams -> one for F=1->2 and one for F=2->3:
laserBeams_cooling_D2 = pylcp.conventional3DMOTBeams(
    s=s, delta=(E_e_D2[-1] - E_g_D2[-1]) + det)
laserBeams_repump_D2 = pylcp.conventional3DMOTBeams(
    s=0.01*s, delta=(E_e_D2[-2] - E_g_D2[-2]))
laserBeams_D2 = laserBeams_cooling_D2 + laserBeams_repump_D2
```

Construct the full rate equations for the D_2 line:

[7]:
```
trap_D2 = pylcp.rateeq(
    laserBeams_D2, magField, hamiltonian_D2, include_mag_forces=False)
trap_D2.generate_force_profile(
    [np.zeros(X.shape), np.zeros(X.shape), X],
    [np.zeros(V.shape), np.zeros(V.shape), V],
    name='Fz')
FzRb_D2 = trap_D2.profile['Fz'].F[2]
```

Plot up the results:

[8]:
```
fig, ax = plt.subplots(2, 2, figsize=(1.5*3.25, 1.5*2.75))
ax[0, 0].imshow(FzLi_D2, origin='bottom',
                extent=(np.amin(x)-dx/2, np.amax(x)+dx/2,
                        np.amin(v)-dv/2, np.amax(v)+dv/2),
                aspect='auto')
ax[0, 1].imshow(Fz2to3, origin='bottom',
                extent=(np.amin(x)-dx/2, np.amax(x)+dx/2,
                        np.amin(v)-dv/2, np.amax(v)+dv/2),
                aspect='auto')
ax[1, 0].plot(X[int(X.shape[0]/2), :],
              FzLi_D2[int(X.shape[0]/2), :])
```

(continues on next page)

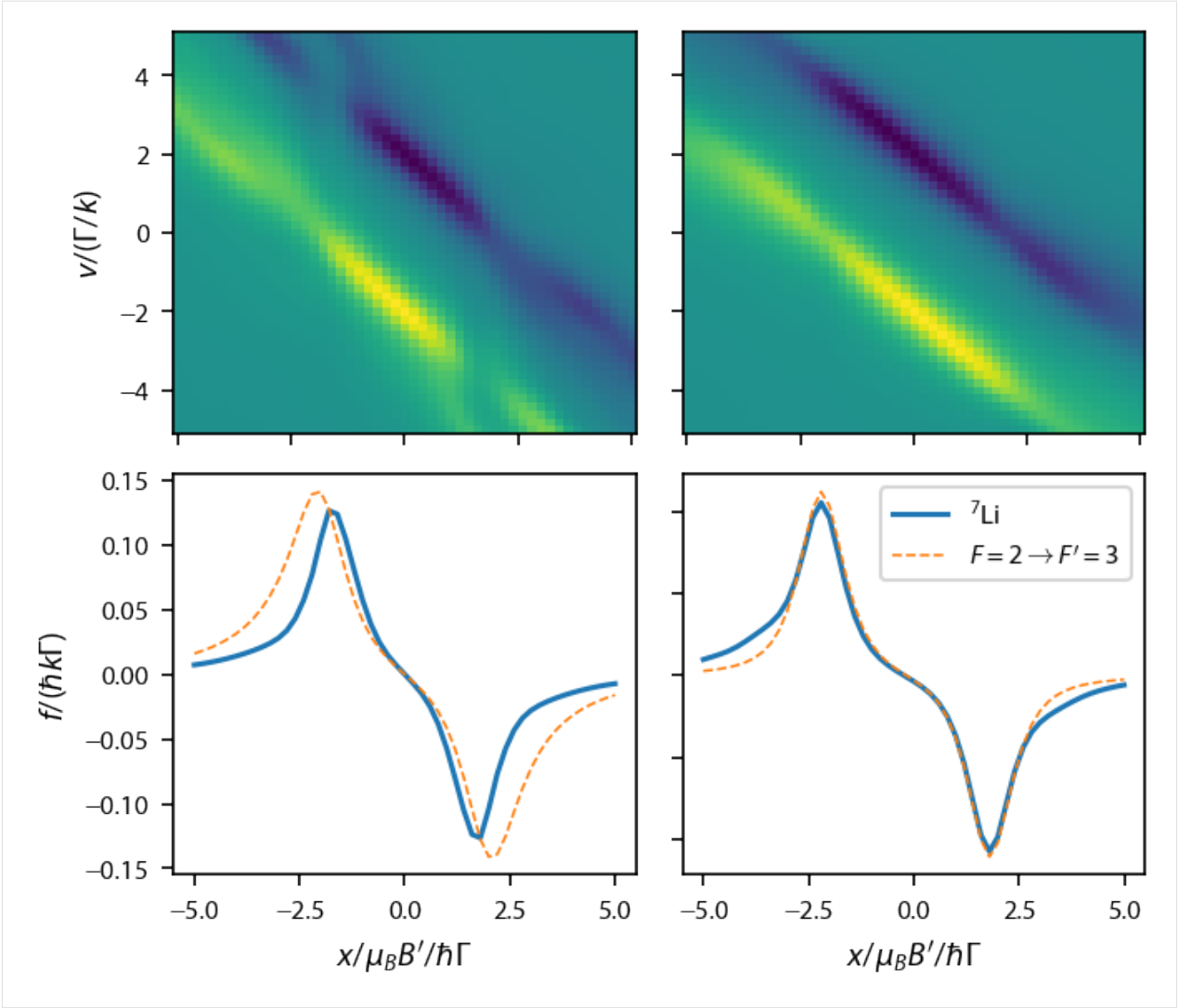
(continued from previous page)

```
ax[1, 0].plot(X[int(X.shape[0]/2), :],
              Fz2to3[int(X.shape[0]/2), :], '--',
              linewidth=0.75)
ax[1, 1].plot(V[:, int(X.shape[1]/2)+1],
              FzLi_D2[:, int(X.shape[1]/2)+1], label='$^7$Li')
ax[1, 1].plot(V[:, int(X.shape[1]/2)+1],
              Fz2to3[:, int(X.shape[1]/2)+1], '--',
              label='$F=2 \\rightarrow F\'=3$',
              linewidth=0.75)

ax[1, 1].legend(fontsize=8)

[ax[ii, 1].yaxis.set_ticklabels('') for ii in range(2)]
[ax[0, ii].xaxis.set_ticklabels('') for ii in range(2)]

ax[0, 0].set_ylabel('$v/(\Gamma/k)$')
ax[1, 0].set_ylabel('$f/(\hbar k \Gamma)$')
ax[1, 0].set_xlabel('$x/\mu_B B\'/\hbar\Gamma$')
ax[1, 1].set_xlabel('$x/\mu_B B\'/\hbar\Gamma$');
```



^{23}Na: type-I MOTs

Now letâĂŹs cover the four type I (type-I/type-II) MOTs of J. Flemming, et. al., Opt. Commun. 135, 269 (1997). We must loop through the four types.

[9]:
```
atom = pylcp.atom("23Na")
H_g_D1, Bq_g_D1 = pylcp.hamiltonians.hyperfine_coupled(
    atom.state[0].J, atom.I, atom.state[0].gJ, atom.gI,
    atom.state[0].Ahfs/atom.state[1].gammaHz, Bhfs=0, Chfs=0,
    muB=3)
H_e_D1, Bq_e_D1 = pylcp.hamiltonians.hyperfine_coupled(
    atom.state[1].J, atom.I, atom.state[1].gJ, atom.gI,
    Ahfs=atom.state[1].Ahfs/atom.state[1].gammaHz,
    Bhfs=atom.state[1].Bhfs/atom.state[1].gammaHz, Chfs=0,
    muB=3)

E_g_D1 = np.unique(np.diagonal(H_g_D1))
```

(continues on next page)

(continued from previous page)

```
E_e_D1 = np.unique(np.diagonal(H_e_D1))

dijq_D1 = pylcp.hamiltonians.dqij_two_hyperfine_manifolds(
    atom.state[0].J, atom.state[1].J, atom.I)

hamiltonian_D1 = pylcp.hamiltonian(H_g_D1, H_e_D1, Bq_g_D1, Bq_e_D1, dijq_D1)

# Conditions taken from the paper, Table 1:
conds = np.array([[1, 1, -25/10, -1, 2, 1, -60/10, +1],
                  [1, 1, -20/10, -1, 2, 2, -30/10, +1],
                  [1, 2, -20/10, +1, 2, 1, -50/10, +1],
                  [1, 2, -40/10, +1, 2, 2, -60/10, +1]])

FzNa_D1 = np.zeros((4,) + FzRb_D2.shape)
for ii, cond in enumerate(conds):
    laserBeams_laser1 = pylcp.conventional3DMOTBeams(
        s=s, delta=(E_e_D1[int(cond[1]-1)] - E_g_D1[0]) + cond[2], pol=cond[3])
    laserBeams_laser2 = pylcp.conventional3DMOTBeams(
        s=s, delta=(E_e_D1[int(cond[5]-1)] - E_g_D1[1]) + cond[6], pol=cond[7])
    laserBeams_D1 = laserBeams_laser1 + laserBeams_laser2

    # Calculate the forces:
    trap_D1 = pylcp.rateeq(
        laserBeams_D1, magField, hamiltonian_D1, include_mag_forces=False)
    trap_D1.generate_force_profile(
        [np.zeros(X.shape), np.zeros(X.shape), X],
        [np.zeros(V.shape), np.zeros(V.shape), V],
        name='Fz')
    FzNa_D1[ii] = trap_D1.profile['Fz'].F[2]
```

Plot up the force vs. classical phase space:

[10]:
```
lim = np.amax(np.abs(FzNa_D1))
fig, ax = plt.subplots(2, 2, figsize=(1.5*3.25, 1.5*2.75))
ax[0, 0].imshow(FzNa_D1[0], origin='bottom',
                extent=(np.amin(x)-dx/2, np.amax(x)+dx/2,
                        np.amin(v)-dv/2, np.amax(v)+dv/2),
                aspect='auto', clim=(-0.01, 0.01))
ax[0, 1].imshow(FzNa_D1[1], origin='bottom',
                extent=(np.amin(x)-dx/2, np.amax(x)+dx/2,
                        np.amin(v)-dv/2, np.amax(v)+dv/2),
                aspect='auto', clim=(-0.01, 0.01))
ax[1, 0].imshow(FzNa_D1[2], origin='bottom',
                extent=(np.amin(x)-dx/2, np.amax(x)+dx/2,
                        np.amin(v)-dv/2, np.amax(v)+dv/2),
                aspect='auto', clim=(-0.01, 0.01))
ax[1, 1].imshow(FzNa_D1[3], origin='bottom',
                extent=(np.amin(x)-dx/2, np.amax(x)+dx/2,
                        np.amin(v)-dv/2, np.amax(v)+dv/2),
                aspect='auto', clim=(-0.01, 0.01))

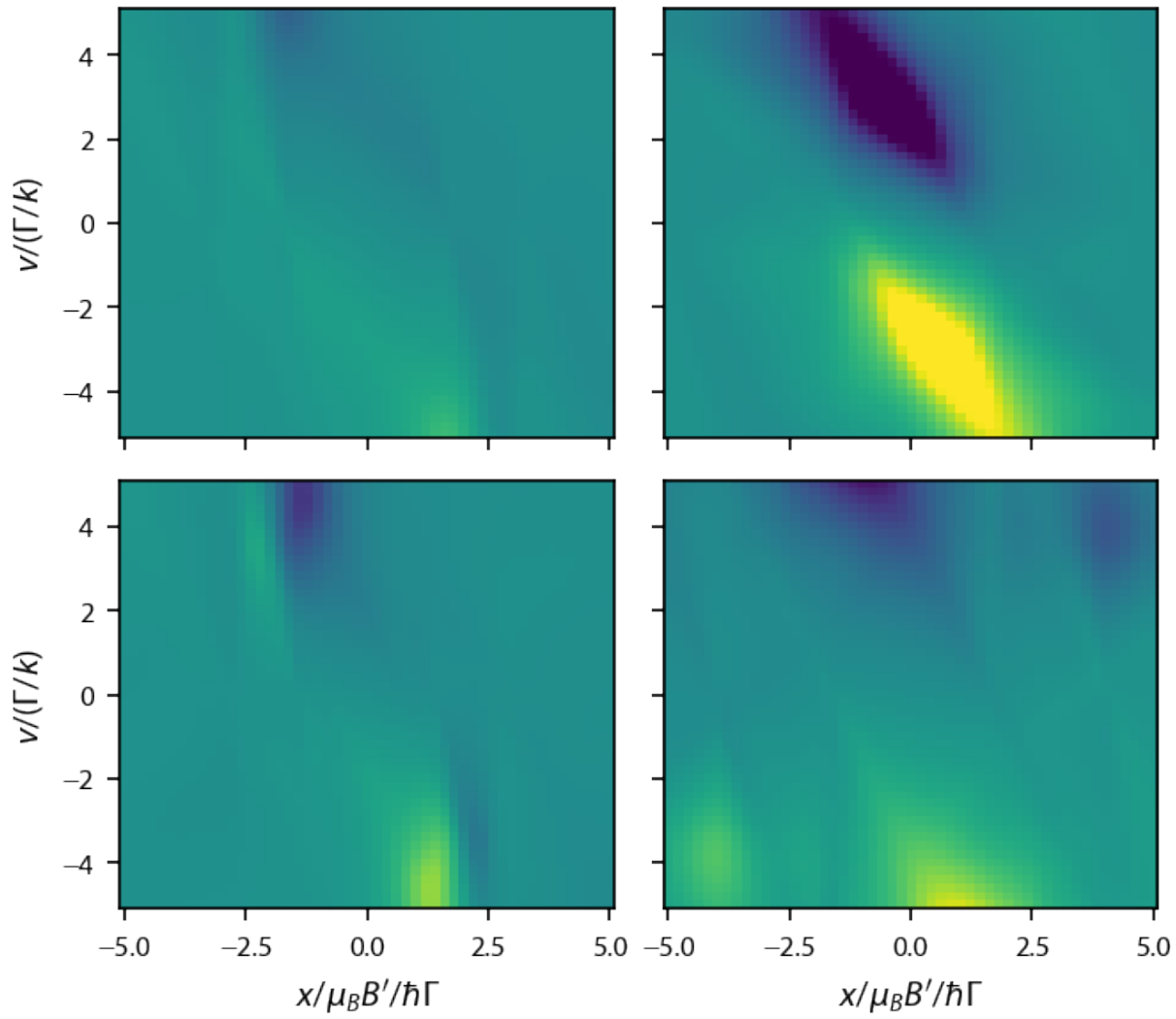
[ax[ii, 1].yaxis.set_ticklabels('') for ii in range(2)]
```

(continues on next page)

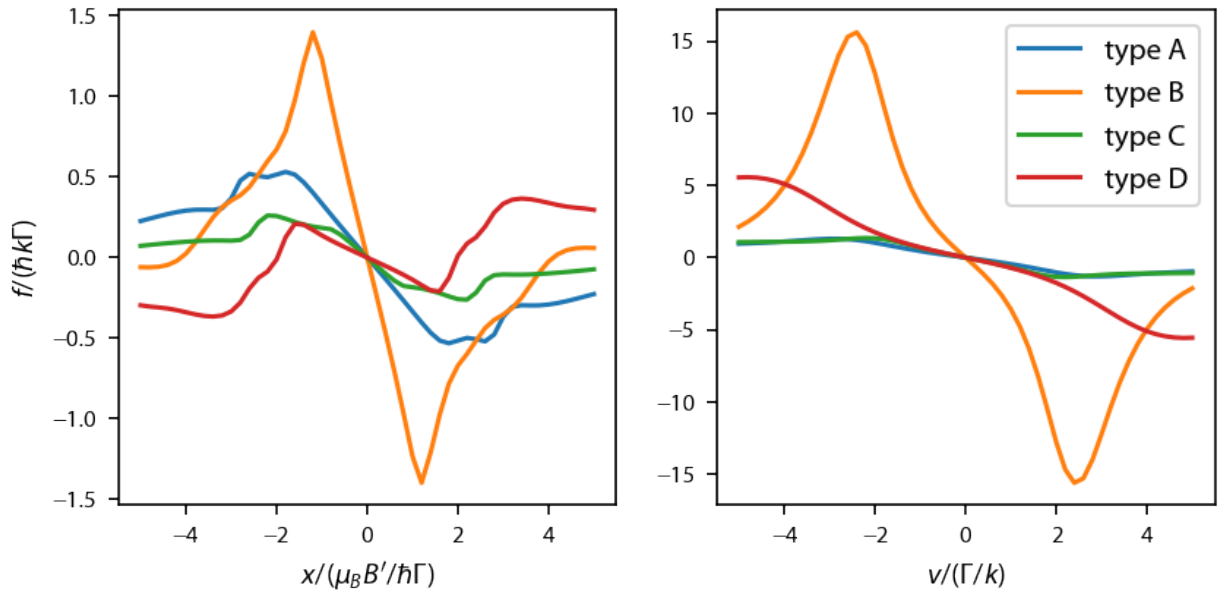
(continued from previous page)

```
[ax[0, ii].xaxis.set_ticklabels('') for ii in range(2)]

ax[0, 0].set_ylabel('$v/(\Gamma/k)$')
ax[1, 0].set_ylabel('$v/(\Gamma/k)$')
ax[1, 0].set_xlabel('$x/\mu_B B\'/\hbar\Gamma$')
ax[1, 1].set_xlabel('$x/\mu_B B\'/\hbar\Gamma$');
```



[16]:
```
fig, ax = plt.subplots(1, 2, figsize=(6.25, 2.75))
types = ['A','B','C','D']
for ii in range(4):
    ax[0].plot(x, 1e3*FzNa_D1[ii][int(len(x)/2),:], label='type ' + types[ii])
    ax[1].plot(v, 1e3*FzNa_D1[ii][:,int(len(v)/2)], label='type ' + types[ii])
ax[1].legend()
ax[0].set_xlabel('$x/(\mu_B B\'/\hbar \Gamma)$')
ax[1].set_xlabel('$v/(\Gamma/k)$')
ax[0].set_ylabel('$f/(\hbar k \Gamma)$')
fig.subplots_adjust(wspace=0.2)
```



3.3.8 Two color MOT

This example covers calculating the forces in a two-color type-II three-dimensional MOT. This example is based on Fig. 1 of M.R. Tarbutt and T.C. Steimle, âĂIJModeling magneto-optical trapping of CaF moleculesâĂİ *Physical Review A* **92**, 053401 (2015). http://dx.doi.org/10.1103/PhysRevA.92.053401

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants as cts
import pylcp
import pylcp.tools
```

Specify the problem

Here, we use $F = 2 \rightarrow F' = 1$ with $g_l = 1$ and $g_u = 0$. For details about the units, chosen here see `01_F0_to_F1_1D_MOT_capture.ipynb`. This notebook uses the *hybrid* unit system that requires us to neglect the magnetic forces in the rate equation.

[2]:
```
Hg, Bgq = pylcp.hamiltonians.singleF(F=2, gF=0.5, muB=1)
He, Beq = pylcp.hamiltonians.singleF(F=1, gF=0, muB=1)

dijq = pylcp.hamiltonians.dqij_two_bare_hyperfine(2, 1)

# Define the full Hamiltonian:
hamiltonian = pylcp.hamiltonian(Hg, He, Bgq, Beq, dijq)

# Define the magnetic field:
magField = pylcp.quadrupoleMagneticField(1)
```

Run the detuning of the blue detuned beam

At every detuning point, calculate the trapping frequency and damping coefficient for the MOT.

[3]:
```
# Define the detunings:
dets = np.linspace(-5, 5, 101)
s = 3.6

# The red detuned beams are constant in this calculation, so let's make that
# collections once:
r_beams = pylcp.conventional3DMOTBeams(
    delta=-1, s=s, pol=+1,
    beam_type=pylcp.infinitePlaneWaveBeam
)

it = np.nditer([dets, None, None])
for (det_i, omega_i, beta_i) in it:
    # Make the blue-detued beams:
    b_beams = pylcp.conventional3DMOTBeams(
        delta=det_i, s=s, pol=-1,
        beam_type=pylcp.infinitePlaneWaveBeam
    )

    all_beams = pylcp.laserBeams(b_beams.beam_vector + r_beams.beam_vector)

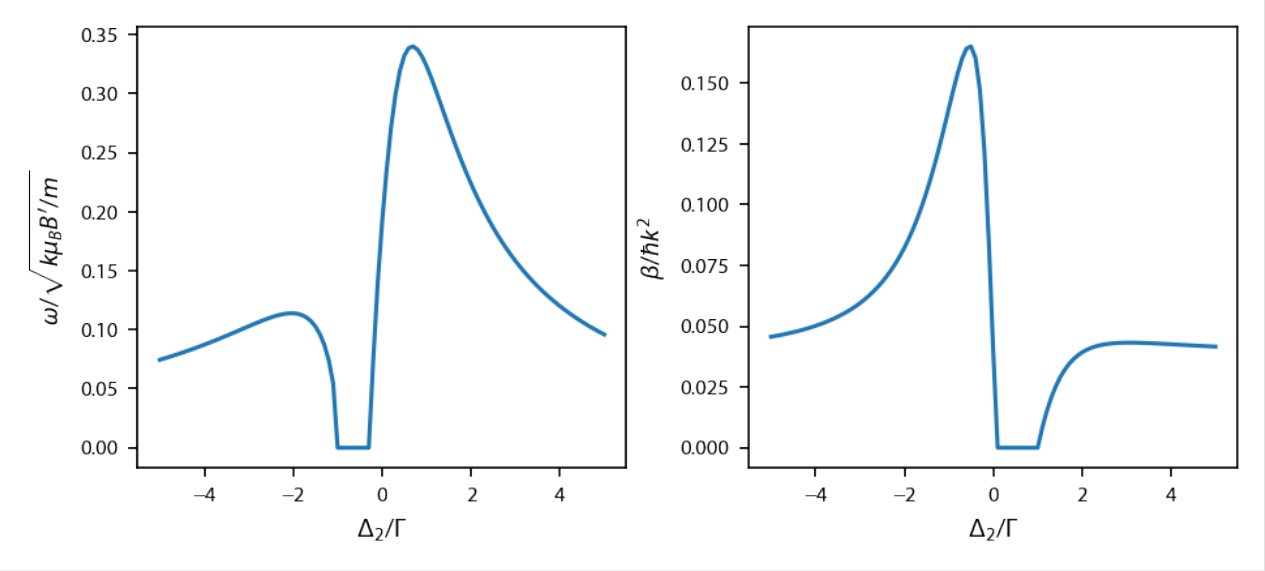
    trap = pylcp.rateeq(all_beams, magField, hamiltonian, include_mag_forces=False)
    omega_i[...] = trap.trapping_frequencies(axes=[2], eps=0.0001)
    beta_i[...] = trap.damping_coeff(axes=[2], eps=0.0001)
```

Plot up the result:

[4]:
```
fig, ax = plt.subplots(1, 2, figsize=(6.25, 2.75))
ax[0].plot(dets, it.operands[1])
ax[1].plot(dets, it.operands[2])

[ax_i.set_xlabel('$\Delta_2/\Gamma$') for ax_i in ax];
ax[0].set_ylabel('$\\omega/\sqrt{k \mu_B B\'/m}$')
ax[1].set_ylabel('$\\beta/\hbar k^2$')

fig.subplots_adjust(left=0.08, wspace=0.25)
```



3.3.9 Recoil-limited MOT

In this example, we simulate a recoil-limited MOT like the Sr red MOT. LetâĂŹs use the bosonic isotope, which is a $F = 0 \to F' = 1$ transition. In this case, we can use `heuristiceq` along with either `obe` or `rateeq`. Our results can be compared to R.K. Hanley, P. Huillery, N.C. Keegan, A.D. Bounds, D. Boddy, R. Faoro, and M.P.A. Jones, âĂIJQuantitative simulation of a magneto-optical trap operating near the photon recoil limitâĂİ *Journal of Modern Optics* **65**, 667 (2018). https://dx.doi.org/10.1080/09500340.2017.1401679

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylcp
import scipy.constants as cts
from pylcp.common import progressBar
```

LetâĂŹs develop a unit system appropriate for ^{88}Sr:

[2]:
```
k = 2*np.pi/689E-7    # cm^{-1}
x0 = 1/k              # our length scale in cm
gamma = 2*np.pi*7.5e3 # 7.5 kHz linewidth
t0 = 1/gamma          # our time scale in s

# Magnetic field gradient parameter (the factor of 3/2 comes from the
# excited state g-factor.)
alpha = (3/2)*cts.value('Bohr magneton in Hz/T')*1e-4*8*x0/7.5E3

# The unitless mass parameter:
mass = 87.8*cts.value('atomic mass constant')*(x0*1e-2)**2/cts.hbar/t0

# Gravity
g = -np.array([0., 0., 9.8*t0**2/(x0*1e-2)])

print(x0, t0, mass, alpha, g)
```

```
1.0965775579031588e-05 2.1220659078919377e-05 0.7834067174281623 0.02455674894694879 [-0.
↪          -0.          -0.04024431]
```

Now define the problem:

We define a preliminary detuning, intensity, along with the basic Hamiltonian, laser beams, and magnetic field:

[3]:
```
s = 25
det = -200/7.5

magField = pylcp.quadrupoleMagneticField(alpha)

laserBeams = pylcp.conventional3DMOTBeams(delta=det, s=s,
                                          beam_type=pylcp.infinitePlaneWaveBeam)

Hg, mugq = pylcp.hamiltonians.singleF(F=0, muB=1)
He, mueq = pylcp.hamiltonians.singleF(F=1, muB=1)

dq = pylcp.hamiltonians.dqij_two_bare_hyperfine(0, 1)

hamiltonian = pylcp.hamiltonian(Hg, He, mugq, mueq, dq, mass=mass)

#eqn = pylcp.heuristiceq(laserBeams, magField, g, mass=mass)
eqn = pylcp.rateeq(laserBeams, magField, hamiltonian, g)
```

Generate a force profile:

We can compare this to Fig. 1 of the reference above.

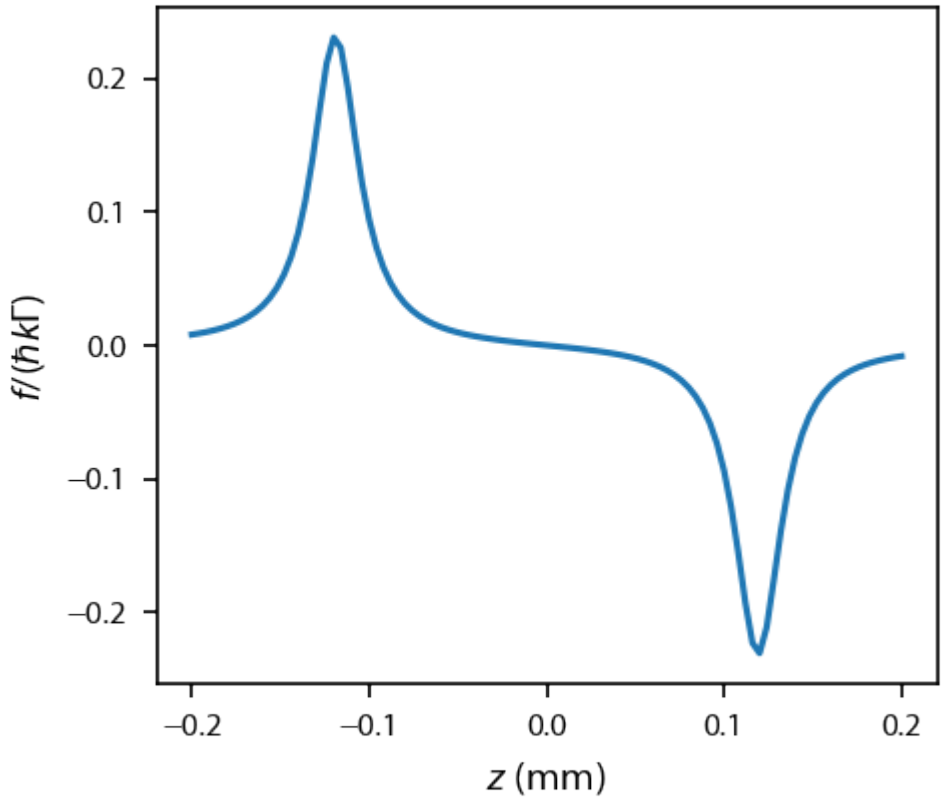
[4]:
```
z = np.linspace(-0.2, 0.2, 101)/(10*x0)

R = np.array([np.zeros(z.shape), np.zeros(z.shape), z])
V = np.zeros((3,) + z.shape)

eqn.generate_force_profile(R, V, name='Fz')

fig, ax = plt.subplots(1, 1)

ax.plot(z*(10*x0), eqn.profile['Fz'].F[2])
ax.set_xlabel('$z$ (mm)')
ax.set_ylabel('$f/(\hbar k \Gamma)$');
```



Note in the paper they have the F in units of N. When adding the $\hbar k\Gamma$ to mine, I find my force is a factor of 2 lower than theirs in the plot.

Dynamics

Before we run hundreds of simultions, letâĂŹs first run a single simulation of an atom just to make sure that everything is working:

[5]:
```
tmax = 0.05/t0
if isinstance(eqn, pylcp.rateeq):
    eqn.set_initial_pop(np.array([1., 0., 0., 0.]))
eqn.set_initial_position(np.array([0., 0., 0.]))
eqn.evolve_motion([0, 0.05/t0], random_recoil=True, progress_bar=True, max_step=1.)
```

```
Completed in 18.50 s.
```

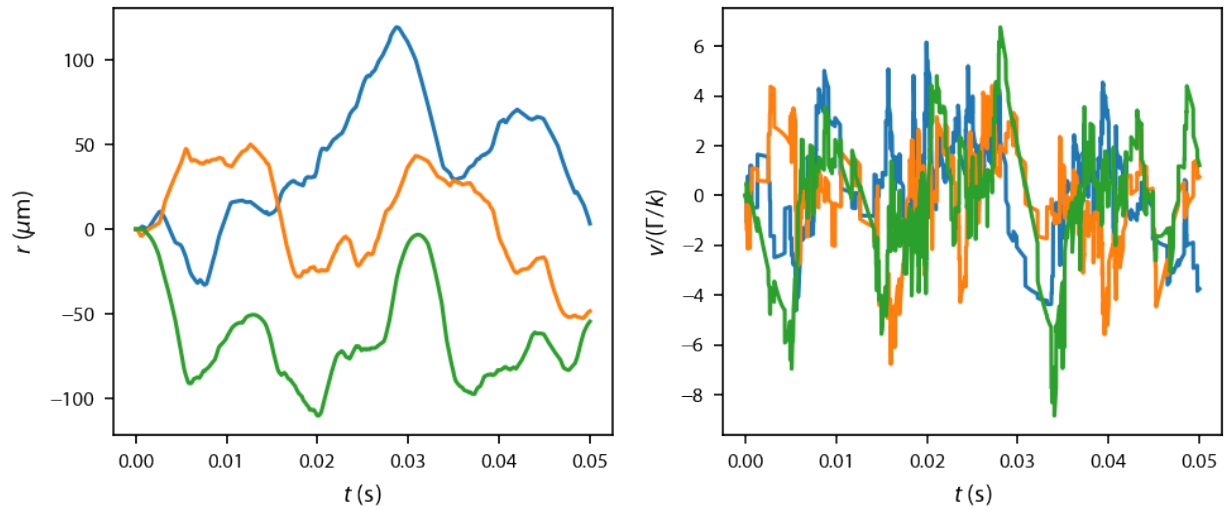
Plot up this test solution:

[6]:
```
fig, ax = plt.subplots(1, 2, figsize=(6.5, 2.75))
ax[0].plot(eqn.sol.t*t0, eqn.sol.r.T*(1e4*x0))
ax[1].plot(eqn.sol.t*t0, eqn.sol.v.T)
ax[0].set_ylabel('$r$ ($\mu$m)')
ax[0].set_xlabel('$t$ (s)')
ax[1].set_ylabel('$v/(\Gamma/k)$')
```

(continues on next page)

(continued from previous page)

```
ax[1].set_xlabel('$t$ (s)')
fig.subplots_adjust(left=0.08, wspace=0.22)
```



Now simulate *many* atoms

Here, we use the `pathos` package to do parallel processing

[7]:
```
import pathos

if hasattr(eqn, 'sol'):
    del eqn.sol

def generate_random_solution(x, eqn=eqn, tmax=tmax):
    # We need to generate random numbers to prevent solutions from being seeded
    # with the same random number.
    import numpy as np

    np.random.rand(256*x)
    eqn.evolve_motion(
        [0, tmax],
        t_eval=np.linspace(0, tmax, 1001),
        random_recoil=True,
        progress_bar=False,
        max_step=1.
    )

    return eqn.sol

Natoms = 1024
chunksize = 4
sols = []
progress = progressBar()
for jj in range(int(Natoms/chunksize)):
    with pathos.pools.ProcessPool(nodes=4) as pool:
```

(continues on next page)

(continued from previous page)

```
        sols += pool.map(generate_random_solution, range(chunksize))
    progress.update((jj+1)/(Natoms/chunksize))
```

```
Completed in 1:33:54.
```

Plot up all the trajectories. We make a basic ejected criterion, which says that if the atom flies more than 500 μm away from the origin in either $\hat{x}$ or $\hat{y}$, we say that atom is ejected:

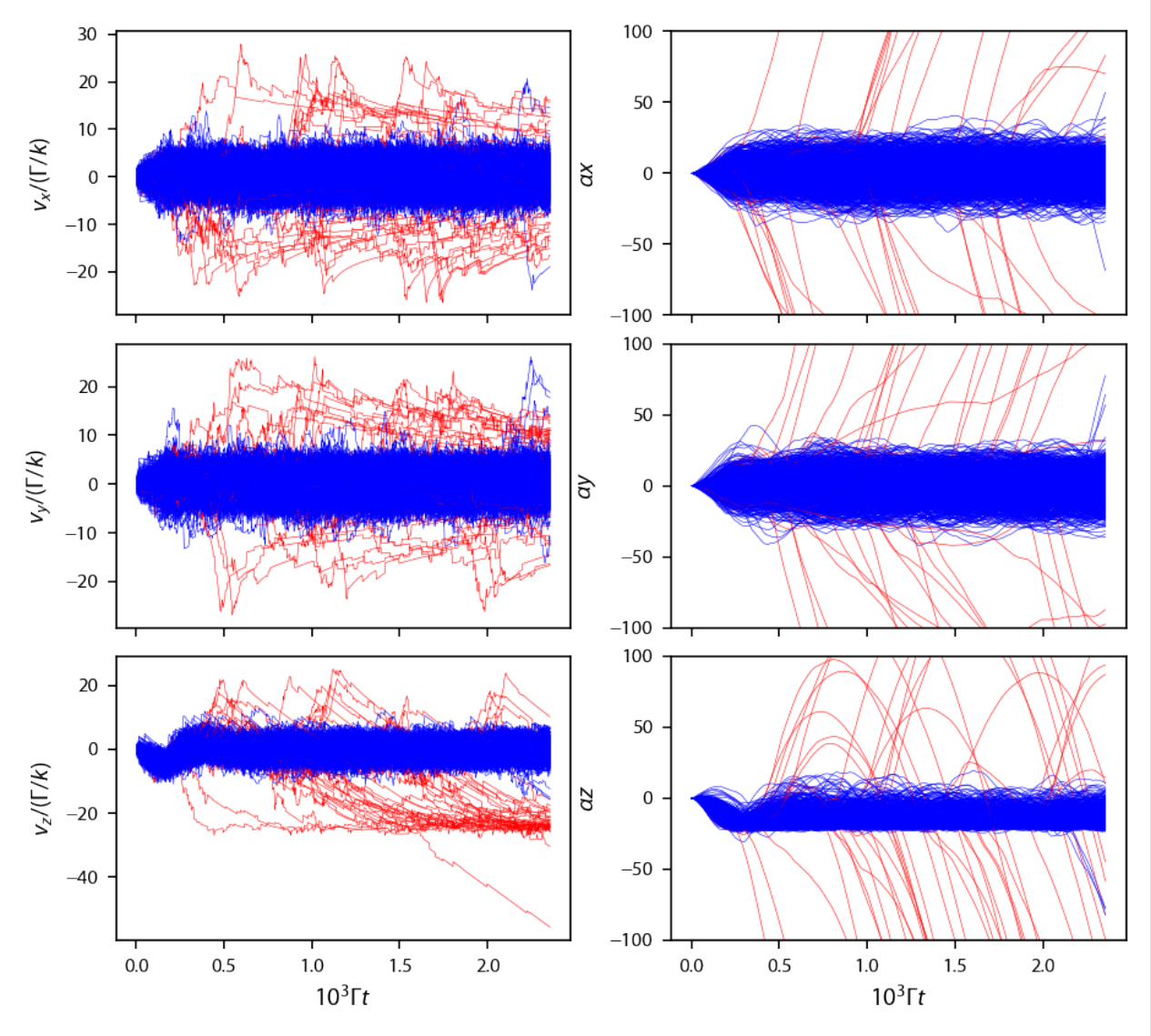
[8]:
```
ejected = [np.bitwise_or(
    np.abs(sol.r[0, -1]*(1e4*x0))>500,
    np.abs(sol.r[1, -1]*(1e4*x0))>500
) for sol in sols]

print('Number of ejected atoms: %d' % np.sum(ejected))
fig, ax = plt.subplots(3, 2, figsize=(6.25, 2*2.75))
for sol, ejected_i in zip(sols, ejected):
    for ii in range(3):
        if ejected_i:
            ax[ii, 0].plot(sol.t/1e3, sol.v[ii], color='r', linewidth=0.25)
            ax[ii, 1].plot(sol.t/1e3, sol.r[ii]*alpha, color='r', linewidth=0.25)
        else:
            ax[ii, 0].plot(sol.t/1e3, sol.v[ii], color='b', linewidth=0.25)
            ax[ii, 1].plot(sol.t/1e3, sol.r[ii]*alpha, color='b', linewidth=0.25)

"""for ax_i in ax[:, 0]:
    ax_i.set_ylim((-0.75, 0.75))
for ax_i in ax[:, 1]:
    ax_i.set_ylim((-4., 4.))"""
for ax_i in ax[-1, :]:
    ax_i.set_xlabel('$10^3 \Gamma t$')
for jj in range(2):
    for ax_i in ax[jj, :]:
        ax_i.set_xticklabels('')
for ax_i, lbl in zip(ax[:, 0], ['x','y','z']):
    ax_i.set_ylabel('$v_' + lbl + '/(\Gamma/k)$')
for ax_i, lbl in zip(ax[:, 1], ['x','y','z']):
    ax_i.set_ylim((-100, 100))
    ax_i.set_ylabel('$\\alpha ' + lbl + '$')

fig.subplots_adjust(left=0.1, bottom=0.08, wspace=0.22)
```

```
Number of ejected atoms: 39
```



Now, every 0.1 ms, bin the x and z coordinates, make a histogram, and simulate an image:

[9]:
```
allx = np.array([], dtype='float64')
allz = np.array([], dtype='float64')

for sol in sols:
    allx = np.append(allx, sol.r[0][200::100]*(1e4*x0))
    allz = np.append(allz, sol.r[2][200::100]*(1e4*x0))

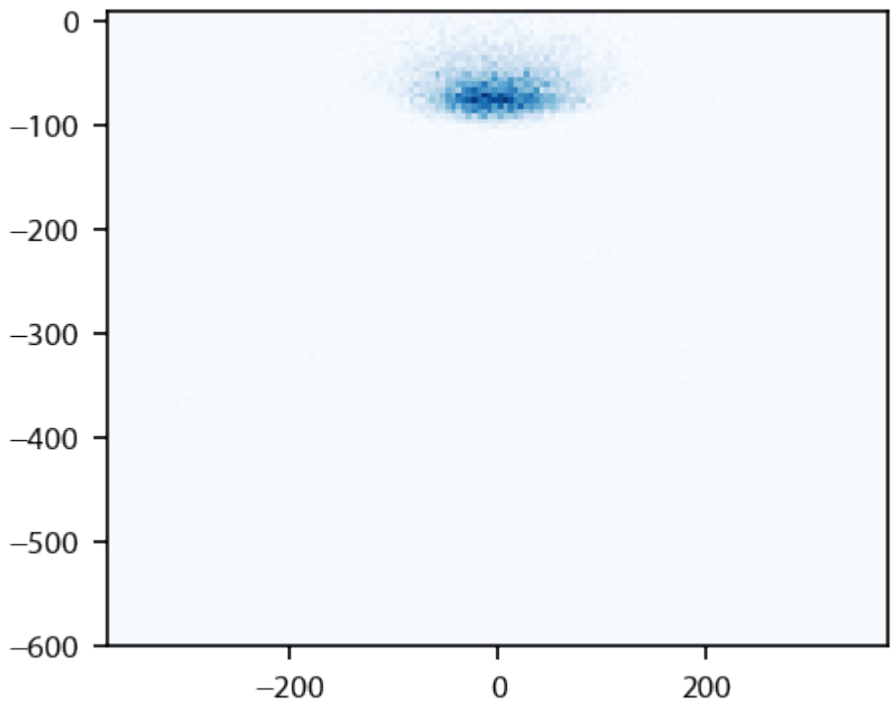
img, x_edges, z_edges = np.histogram2d(allx, allz, bins=[np.arange(-375, 376, 5.), np.
↪arange(-600., 11., 5.)])

fig, ax = plt.subplots(1, 1)
im = ax.imshow(img.T, origin='bottom',
               extent=(np.amin(x_edges), np.amax(x_edges),
                       np.amin(z_edges), np.amax(z_edges)),
```

(continues on next page)

(continued from previous page)

```
            cmap='Blues',
            aspect='equal')
```



Now, letâĂŹs run the detuning

And produce the resulting simulated MOT images

[10]:
```
dets = np.array([det, -400/7.5, -600/7.5, -800/7.5])
#s = 9

imgs = np.zeros(dets.shape + img.shape)
num_of_ejections = np.zeros(dets.shape)
num_of_ejections[0] = np.sum(ejected)
imgs[0] = img

for ii, det in enumerate(dets[1:]):
    # Rewrite the laser beams with the new detuning
    laserBeams = pylcp.conventional3DMOTBeams(delta=det, s=s,
                                              beam_type=pylcp.infinitePlaneWaveBeam)

    # Make the equation:
    eqn = pylcp.rateeq(laserBeams, magField, hamiltonian, g)
    if isinstance(eqn, pylcp.rateeq):
        eqn.set_initial_pop(np.array([1., 0., 0., 0.]))

    # Use the last equilibrium position to set this position:
    eqn.set_initial_position(np.array([0., 0., np.mean(allz)]))
```

(continues on next page)

(continued from previous page)

```
    # Re-define the random soluton:
    def generate_random_solution(x, eqn=eqn, tmax=tmax):
        # We need to generate random numbers to prevent solutions from being seeded
        # with the same random number.
        import numpy as np

        np.random.rand(256*x)
        eqn.evolve_motion(
            [0, tmax],
            t_eval=np.linspace(0, tmax, 1001),
            random_recoil=True,
            progress_bar=False,
            max_step=1.
        )

        return eqn.sol

    # Generate the solution:
    sols = []
    progress = progressBar()
    for jj in range(int(Natoms/chunksize)):
        with pathos.pools.ProcessPool(nodes=4) as pool:
            sols += pool.map(generate_random_solution, range(chunksize))
        progress.update((jj+1)/(Natoms/chunksize))

    # Generate the image:
    allx = np.array([], dtype='float64')
    allz = np.array([], dtype='float64')

    for sol in sols:
        allx = np.append(allx, sol.r[0][200::100]*(1e4*x0))
        allz = np.append(allz, sol.r[2][200::100]*(1e4*x0))

    img, x_edges, z_edges = np.histogram2d(allx, allz, bins=[x_edges, z_edges])

    # Save the image:
    imgs[ii+1] = img

    # Count the number of ejections:
    num_of_ejections[ii+1] = np.sum([np.bitwise_or(
        np.abs(sol.r[0, -1]*(1e4*x0))>500,
        np.abs(sol.r[1, -1]*(1e4*x0))>500
    ) for sol in sols])
```

```
Completed in 1:26:44.
Completed in 1:11:48.
Completed in 1:23:13.
```

Print out the statistics of the ejections:

[11]:
```
print('Number of ejections: ', num_of_ejections)
print('Estimated lifetime: ', (-np.log((Natoms-num_of_ejections)/Natoms)/(tmax*t0)))
```

```
Number of ejections:  [39. 27. 48. 39.]
Estimated lifetime:  [0.77660329 0.53442071 0.96018438 0.77660329]
```

Now plot it up, with the ellipse indicating when the Zeeman shift from the magnetic field gradient equals the detuning

[12]:
```
from matplotlib.patches import Ellipse

fig, ax = plt.subplots(1, 4, figsize=(6.5, 1.625))

clims = [43, 35, 30, 25]
for ii in range(4):
    # Want to adjust scale for the increasing size of the MOT.  I thought this was␣
↪clever:
    counts, bins = np.histogram(imgs[ii].flatten(), bins=np.arange(10, 50, 1))

    im = ax[ii].imshow(imgs[ii].T/(2.5*bins[np.argmax(counts)]), origin='bottom',
                       extent=(np.amin(x_edges), np.amax(x_edges),
                               np.amin(z_edges), np.amax(z_edges)),
                       cmap='Blues', clim=(0, 1))
    ax[ii].set_title('$\Delta/\Gamma = %.1f$'%dets[ii])
    ax[ii].set_xlabel('$x$ ($\mu$m)')

    ellip = Ellipse(xy=(0,0),
                    width=4*dets[ii]/alpha*(1e4*x0),
                    height=2*dets[ii]/alpha*(1e4*x0),
                    linestyle='--',
                    linewidth=0.5,
                    facecolor='none',
                    edgecolor='red')

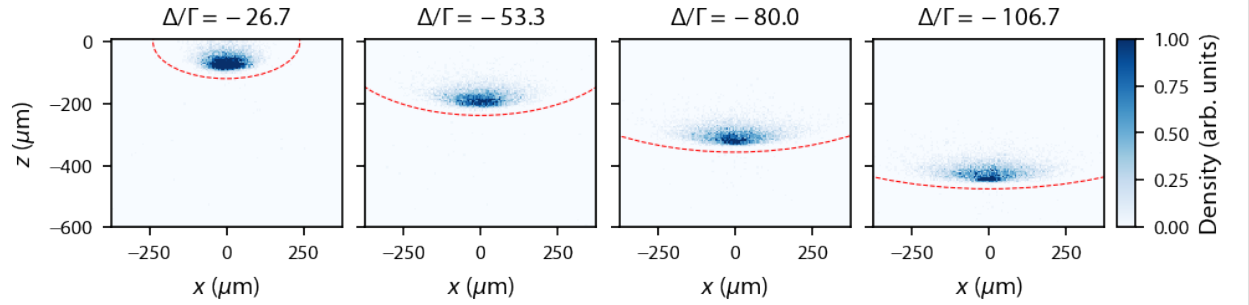
    ax[ii].add_patch(ellip)
    if ii>0:
        ax[ii].yaxis.set_ticklabels('')

fig.subplots_adjust(left=0.08, bottom=0.12, top=0.97, right=0.9)

pos = ax[-1].get_position()
cbar_ax = fig.add_axes([0.91, pos.y0, 0.015, pos.y1-pos.y0])
fig.colorbar(im, cax=cbar_ax)
cbar_ax.set_ylabel('Density (arb. units)')

ax[0].set_ylabel('$z$ ($\mu$m)')
```

[12]:
```
Text(0, 0.5, '$z$ ($\\mu$m)')
```



3.3.10 CaF MOT

This example covers calculating the forces in a standard, six-beam CaF MOT. The goal is to reproduce the figures in M.R. Tarbutt and T.C. Steimle, âĂIJModeling magneto-optical trapping of CaF moleculesâĂİ *Physical Review A* **92** 053401 (2015); http://dx.doi.org/10.1103/PhysRevA.92.053401

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants as cts
import pylcp
```

Define the problem

Define the Hamiltonian

The spectroscopic numbers for the ground state come from W.J. Childs, G.L. Goodman, and L.S. Goodman, âĂIJPrecise determination of the v and N dependence of the spin-rotation and hyperfine interactions in the CaF $X^2\Sigma_{1/2}$ ground stateâĂİ, *Journal of Molecular Spectroscopy* **86** 365 (1981); https://doi.org/10.1016/0022-2852(81)90288-5.

[4]:
```
Gamma = 8.3 # MHz

H0_X, Bq_X, U_X, Xbasis = pylcp.hamiltonians.XFmolecules.Xstate(
    N=1, I=1/2, B=0, gamma=39.65891/Gamma,
    b=109.1893/Gamma, c=40.1190/Gamma, CI=2.876e-2/Gamma, q0=0, q2=0,
    gS=2.0023193043622, gI=0.,
    muB=cts.value('Bohr magneton in Hz/T')/1e6*1e-4/Gamma, return_basis=True
    )
E_X = np.unique(np.diag(H0_X))

H0_A, Bq_A, Abasis = pylcp.hamiltonians.XFmolecules.Astate(
    J=1/2, I=1/2, P=+1, a=(3/2*4.8)/Gamma, glprime=-3*.0211,
    muB=cts.value('Bohr magneton in Hz/T')/1e6*1e-4/Gamma, return_basis=True
    )
E_A = np.unique(np.diag(H0_A))

dijq = pylcp.hamiltonians.XFmolecules.dipoleXandAstates(
    Xbasis, Abasis, I=1/2, S=1/2, UX=U_X
    )
```

(continues on next page)

(continued from previous page)

```
hamiltonian = pylcp.hamiltonian(H0_X, H0_A, Bq_X, Bq_A, dijq)
```

Set up the other parameters

There are some small things to consider. First, they use total power in each beam rather than saturation intensity. In the paper, they assume unclipped Gaussian beams of $w = 12$ mm $1/e^2$ radius and use powers of 5, 15, 40 and 100 mW. The relationship between intensity I and power P is given by $I = 2P/\pi w^2$. Moreover, $I_{\rm sat} = 2\pi^2\hbar c\Gamma/\lambda^2$. They also assume 20 G/cm. Given that we define μ_B above in terms of linewidth/G, it should be just as simple as plugging in 2 G/mm in the magField.

[5]:
```
omega = 2*np.pi*(cts.c/606e-9)
Isat = cts.hbar*omega**3*(2*np.pi*Gamma*1e6)/(12*np.pi*cts.c**2)
#print("I_sat = ", Isat*1e-4*1e3)

# Make the magnetic field (2 G/mm):
magField = pylcp.quadrupoleMagneticField(2)

# A litle helper function to make the MOT:
def six_beam_CaF_MOT(s, det):
    laserBeams = pylcp.laserBeams()
    for ii, Eg_i in enumerate(E_X):
        if ii<3:
            laserBeams += pylcp.conventional3DMOTBeams(
                s=s, delta=(E_A[-1] - Eg_i)+det, pol=+1,
                beam_type=pylcp.infinitePlaneWaveBeam
                )
        else:
            laserBeams += pylcp.conventional3DMOTBeams(
                s=s, delta=(E_A[-1] - Eg_i)+det, pol=-1,
                beam_type=pylcp.infinitePlaneWaveBeam
                )

    return laserBeams
```

Reproduce Fig. 3

Note that we do not use Gaussian beams, so there will probably be some disagreement out at large x:

[6]:
```
# The detunings used in the PRAs:
dets = np.array([-0.25, -0.5, -1, -2])
P = np.array([0.005, 0.015, 0.040, 0.1])
intensities = 2.*P/(np.pi*0.012**2)/Isat

# Make the axis:
z = np.linspace(1e-10, 20., 101)
v = np.linspace(0., 4., 101)

# Start the figure:
fig, ax = plt.subplots(2, 2, figsize=(6.25, 4), num="Forces in CaF MOT")
```

(continues on next page)

(continued from previous page)

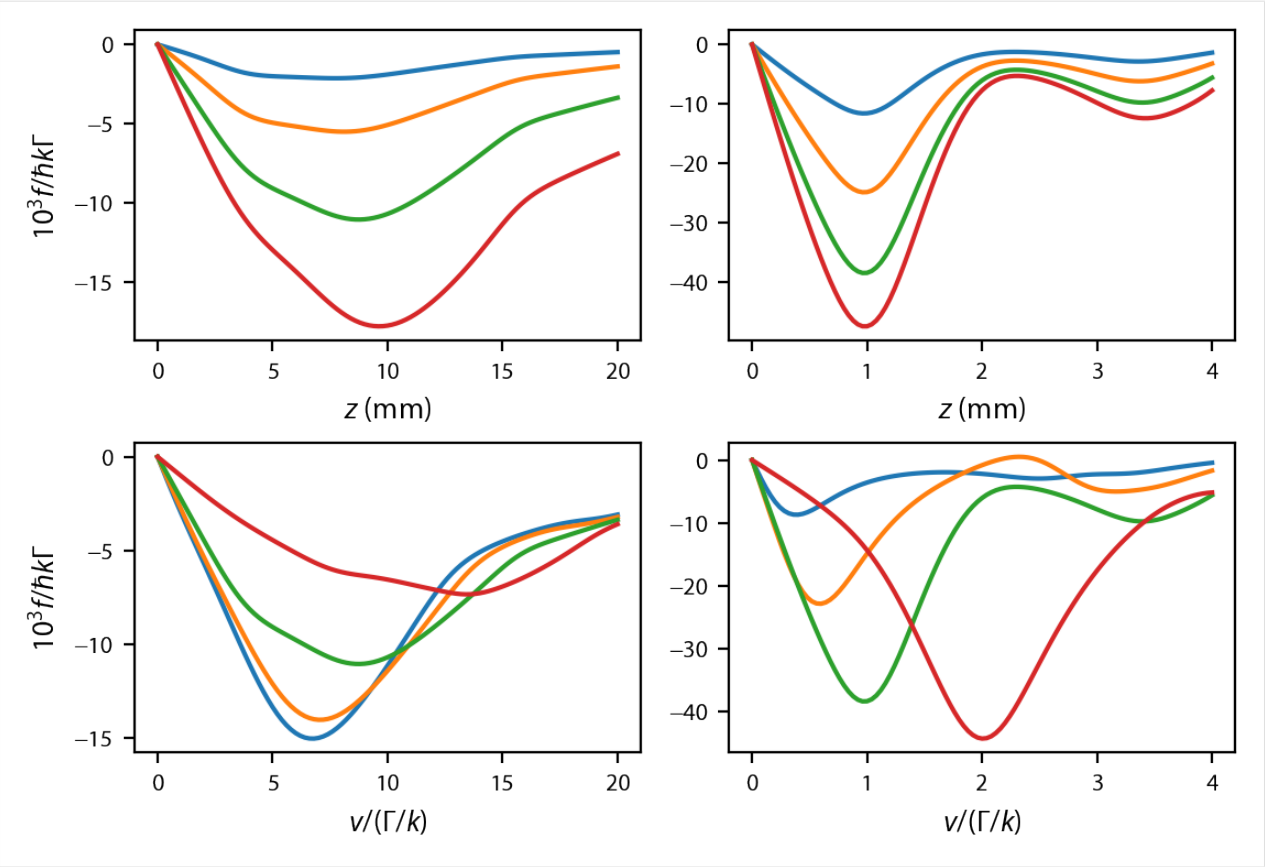
```
for ii, intensity in enumerate(intensities):
    laserBeams = six_beam_CaF_MOT(intensity, dets[2])
    trap = pylcp.rateeq(laserBeams, magField, hamiltonian, include_mag_forces=False)
    trap.generate_force_profile(
        [np.zeros(z.shape), np.zeros(z.shape), z],
        [np.zeros(z.shape), np.zeros(z.shape), np.zeros(z.shape)],
        name='Fz')
    trap.generate_force_profile(
        [np.zeros(v.shape), np.zeros(v.shape), 1e-9*np.ones(v.shape)],
        [np.zeros(v.shape), np.zeros(v.shape), v],
        name='Fv')
    ax[0, 0].plot(z, 1e3*trap.profile['Fz'].F[2], color='C{0:d}'.format(ii))
    ax[0, 1].plot(v, 1e3*trap.profile['Fv'].F[2], color='C{0:d}'.format(ii))

for ii, det_i in enumerate(dets):
    laserBeams = six_beam_CaF_MOT(intensities[2], det_i)
    trap = pylcp.rateeq(laserBeams, magField, hamiltonian, include_mag_forces=False)
    trap.generate_force_profile(
        [np.zeros(z.shape), np.zeros(z.shape), z],
        [np.zeros(z.shape), np.zeros(z.shape), np.zeros(z.shape)],
        name='Fz')
    trap.generate_force_profile(
        [np.zeros(v.shape), np.zeros(v.shape), 1e-9*np.ones(v.shape)],
        [np.zeros(v.shape), np.zeros(v.shape), v],
        name='Fv')
    ax[1, 0].plot(z, 1e3*trap.profile['Fz'].F[2], color='C{0:d}'.format(ii))
    ax[1, 1].plot(v, 1e3*trap.profile['Fv'].F[2], color='C{0:d}'.format(ii))

ax[0, 0].set_ylabel('$10^3 f/\hbar k \Gamma$')
ax[1, 0].set_ylabel('$10^3 f/\hbar k \Gamma$')

for ax_i in ax[0, :]:
    ax_i.set_xlabel('$z$ (mm)')
for ax_i in ax[1, :]:
    ax_i.set_xlabel('$v/(\Gamma/k)$')

fig.subplots_adjust(hspace=0.33, wspace=0.175)
```



Simulate the two color MOT

This reproduces Fig. 4.

[7]:
```
laserBeams = six_beam_CaF_MOT(intensities[2], dets[2])
laserBeams += pylcp.conventional3DMOTBeams(
    s=intensities[2], delta=(E_A[-1] - E_X[0]) + 2, pol=-1
    )

fig, ax = plt.subplots(2, 1, figsize=(3.25, 4),
                       num="Forces in two-color CaF MOT")

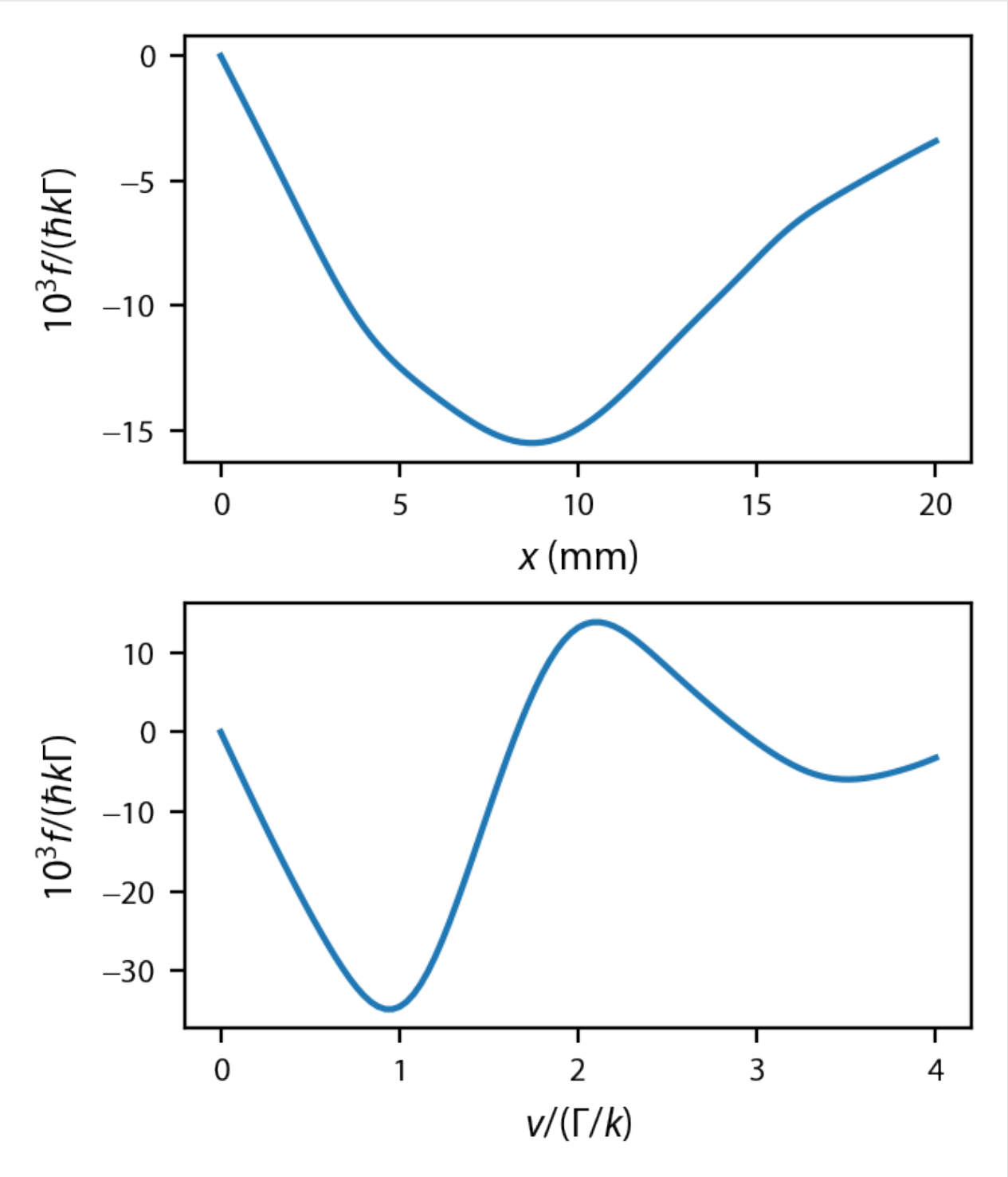
trap = pylcp.rateeq(laserBeams, magField, hamiltonian, include_mag_forces=False)
trap.generate_force_profile(
    [np.zeros(z.shape), np.zeros(z.shape), z],
    [np.zeros(z.shape), np.zeros(z.shape), np.zeros(z.shape)],
    name='Fz')
trap.generate_force_profile(
    [np.zeros(v.shape), np.zeros(v.shape), 1e-3*np.ones(v.shape)],
    [np.zeros(v.shape), np.zeros(v.shape), v],
    name='Fv')
ax[0].plot(z, 1e3*trap.profile['Fz'].F[2])
ax[1].plot(v, 1e3*trap.profile['Fv'].F[2])
```

(continues on next page)

(continued from previous page)

```
ax[0].set_xlabel('$x$ (mm)')
ax[0].set_ylabel('$10^3 f/(\hbar k \Gamma)$')
ax[1].set_xlabel('$v/(\Gamma/k)$')
ax[1].set_ylabel('$10^3 f/(\hbar k \Gamma)$')

fig.subplots_adjust(hspace=0.33)
```



[ ]:

3.4 Other examples

3.4.1 The bichromatic force

This example covers calculating the forces involved in the bichormatic force, or in the stimulated emission of light into two travelling wavepackets. It attempts to replicate Fig. 1 of J. SÃűding, R. Grimm, Y. Ovchinnikov, P. Bouyer, and C. Salomon, Short-Distance Atomic Beam Deceleration with a Stimulated Light ForceâĂİ, *Phys. Rev. Lett.* **78**, 1420 (1997) http://dx.doi.org/10.1103/PhysRevLett.78.1420

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
import pylcp
from IPython.display import HTML
```

Define the problem

As always, the first step is to define the laser beams, magnetic field, and Hamiltonian. The two level Hamiltonian here is the same as many others, like that in the *rapid adiabatic passage* and *two-level molasses* examples.

Because we are dealing with a two state system addressable only with π light, we keep the geometry pretty straight forward by having all lasers move along $\hat{x}$. Note that because we have positive and negative frequencies about resonance, we will put the detuning on the lasers themselves, since the average detuning is zero.

The last thing to think about is the beat phase of the lasers. I follow the phase convention of L. Aldridge, *The Bichromatic Force in Multi-Level Systems*, Ph.D. thesis, 2016.

[2]:
```
# Make a method to return the lasers:
def return_lasers(delta, s):
    return pylcp.laserBeams([
        {'kvec':np.array([1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
         'pol_coord':'spherical', 'delta':delta, 's':s, 'phase':-np.pi/8},
        {'kvec':np.array([1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
         'pol_coord':'spherical', 'delta':-delta, 's':s, 'phase':np.pi/8},
        {'kvec':np.array([-1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
         'pol_coord':'spherical', 'delta':delta, 's':s, 'phase':np.pi/8},
        {'kvec':np.array([-1., 0., 0.]), 'pol':np.array([0., 1., 0.]),
         'pol_coord':'spherical', 'delta':-delta, 's':s, 'phase':-np.pi/8},
        ], beam_type=pylcp.infinitePlaneWaveBeam)


# Standard two-level Hamiltonian:
Hg = np.array([[0.]])
He = np.array([[0.]])
mu_q = np.zeros((3, 1, 1))
d_q = np.zeros((3, 1, 1))
d_q[1, 0, 0] = 1.
```

(continues on next page)

(continued from previous page)

```
hamiltonian = pylcp.hamiltonian(Hg, He, mu_q, mu_q, d_q)

magField = lambda R: np.zeros(R.shape)
```

Examine the phase

LetâĂŹs specifically compare our electric field with Eq. 2.6 in L. Aldridge, *The Bichromatic Force in Multi-Level Systems*, Ph.D. thesis, 2016. To do this, we first make the `laserBeams`, then divide them into the $+\hat{k}$ (rightward) and $-\hat{z}$ (leftward) going components.

[3]:
```
delta = 39.
intensity = 2*39**2

laserBeams = return_lasers(delta, intensity)
laserBeams_rightward = pylcp.laserBeams(laserBeams.beam_vector[:2])
laserBeams_leftward = pylcp.laserBeams(laserBeams.beam_vector[2:])
```

Set up the figure for the animation. We want to capture the output, hence the `%%capture` statement in this cell. The output is just the figure we will draw the animation into.

[4]:
```
%%capture
fig, ax = plt.subplots(1, 1)
line_thr, = ax.plot([], [], lw=1.0)
line_exp, = ax.plot([], [], lw=0.75, color='k', linestyle='--')

ax.set_ylim((-350, 350));
ax.set_xlim((-4*np.pi, 4*np.pi))
ax.set_xlabel('$kx$')
ax.set_ylabel('$E/E_0$')
x = np.linspace(-4*np.pi, 4*np.pi, 1001)

def init():
    line_thr.set_data([], [])
    line_exp.set_data([], [])
    return (line_thr, line_exp)

def animate(i):
    t = i/50*(np.pi/delta)
    #ax.plot(x, np.real(laserBeams_rightward.total_electric_field(np.array([x,]+[np.
↪zeros(x.shape)]*2), t))[1])
    #ax.plot(x, np.real(laserBeams_leftward.total_electric_field(np.array([x,]+[np.
↪zeros(x.shape)]*2), t))[1])
    line_thr.set_data(x, np.real(laserBeams.total_electric_field(np.array([x,]+[np.
↪zeros(x.shape)]*2), t))[1])
    line_exp.set_data(x, 4*np.sqrt(2*intensity)*np.real(np.cos(x)*np.cos(delta*t)*np.
↪cos(np.pi/8)+1j*np.sin(x)*np.sin(delta*t)*np.sin(np.pi/8)))

    return (line_thr, line_exp)
```

Now make the animation. The dashed lines are the expectation from Aldridge, and the solid is the result from `pylcp`.

[5]:
```
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=100, interval=20,
                               blit=True)

HTML(anim.to_html5_video())
```

[5]:
```
<IPython.core.display.HTML object>
```

Generate a force profile

Using the same parameters as Fig. 1 of the PRL. We also use the same time-ending criteria as Aldridge.

[7]:
```
delta = 39
intensities = [2*39**2, 2*43**2, 2*47**2]

v = np.arange(-50., 50.1, 0.5)

obe ={}
for intensity in intensities:
    laserBeams = return_lasers(delta, intensity)

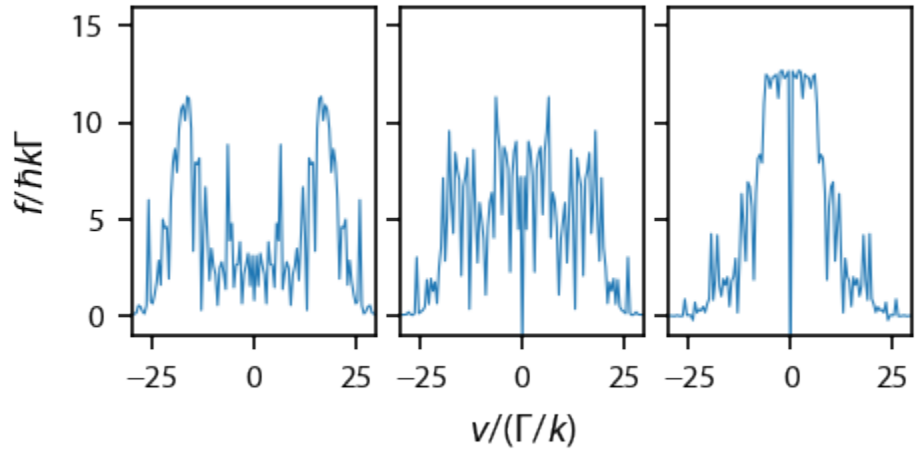
    obe[intensity] = pylcp.obe(laserBeams, magField, hamiltonian, transform_into_re_
↪im=True)
    obe[intensity].generate_force_profile(
        np.zeros((3,) + v.shape),
        [v, np.zeros(v.shape), np.zeros(v.shape)],
        name='molasses', progress_bar=True,
        deltat_func=lambda r, v: 2*np.pi*(np.amin([10., 1./(np.linalg.norm(v)+1e-9)]) +␣
↪200./delta),
        itermax=3, rel=1e-4, abs=1e-6
    )
```

```
Completed in 35:44.
Completed in 33:30.
Completed in 45:44.
```

[9]:
```
fig, ax = plt.subplots(1, 3, figsize=(3.25, 1.5))
for ii, intensity in enumerate(intensities):
    ax[ii].plot(v, obe[intensity].profile['molasses'].F[0], linewidth=0.5)
    ax[ii].set_ylim(-1, 16)
    ax[ii].set_xlim(-30, 30)

for ii in range(1, 3):
    ax[ii].yaxis.set_ticklabels('')

ax[0].set_ylabel('$f/\hbar k \Gamma$')
ax[1].set_xlabel('$v/(\Gamma/k)$')
fig.subplots_adjust(bottom=0.25)
```


15
10
5
0
f/ħkΓ
−25
0
25
−25
0
25
−25
0
25
v/(Γ/k)

CHAPTER

FOUR

DETAILED REFERENCE

This reference contains the full API for *pylcp*

4.1 Hamiltonian Functions

Some useful functions for creating various âĂŸgenericâĂŹ Hamiltonians (i.e. useful for alkali atoms), including their field dependent & independent parts, and connections between various manifolds.

Note: The Hamiltonians have the value of the Bohr magneton specified in Hz/G by default. To use *pylcp* default units, the Bohr magneton values must be overridden.

4.1.1 Other Hamiltonians

In addition to the Hamiltonians specified here, there are other, more specialized Hamiltonians for other quantum systems like molecules.

Molecular Hamiltonians

This subset of Hamiltonian functions is useful for simulating alkaline-earth fluoride molecules like CaF, SrF, etc.

Note: The molecular Hamiltonians have the values of the Bohr and nuclear magneton specified in MHz/G by default. To use *pylcp* default units, the Bohr and nuclear magneton values must be overridden.

Overview

`Xstate`(N, I[, B, gamma, b, c, CI, q0, q2, ...])	Defines the field-free and magnetic field-dependent components of the $X^2\Sigma^+$ ground state Hamiltonian.
`Astate`(J, I, P[, B, D, H, a, b, c, eQq0, p, ...])	Defines the field-free and magnetic field-dependent components of the excited $A^2\Pi_{1/2}$ state Hamiltonian.
`dipoleXandAstates`(xbasis, abasis[, I, S, ...])	Calculate the oscillator strengths between the X and A states.

Detailed functions

pylcp.hamiltonians.XFmolecules.**Astate**(*J*, *I*, *P*, *B=0.0*, *D=0.0*, *H=0.0*, *a=0.0*, *b=0.0*, *c=0.0*, *eQq0=0.0*, *p=0.0*, *q=0.0*, *gS=2.00231930436256*, *gL=1*, *gl=0*, *glprime=0*, *gr=0*, *greprime=0*, *gN=0*, *muB=1.39962449361*, *muN=0.0007622593229068622*, *return_basis=False*)

Defines the field-free and magnetic field-dependent components of the excited $A^2\Pi_{1/2}$ state Hamiltonian.

Parameters

- **J** (*int*) – Rotational quantum number(s)
- **I** (*int or float*) – Nuclear spin quantum number
- **P** (*int or float*) – Parity quantum number (± 1)
- **B** (*float*) – Rotational constant. Default: 0.
- **D** (*float*) – First non-rigid rotor rotational constant. Default: 0.
- **H** (*float*) – Second non-rigid rotor rotational constant. Default: 0.
- **a** (*float*) – Frosch and Foley a parameter. Default: 0.
- **b** (*int or float*) – Frosch and Foley b parameter. Default: 0.
- **c** (*int or float*) – Frosch and Foley c parameter. Default: 0.
- **eQq0** (*int or float*) – electric quadrupole hyperfine constant (only valid for I>=1). Default: 0.
- **p** (*float*) – Lambda-doubling constant. Default: 0.
- **q** (*float*) – Lambda-doubling constant. Default: 0.
- **muB** (*float*) – Bohr Magneton. Default value is the CODATA value in MHz/G.
- **muN** (*float*) – Nuclear Magneton. Default value is the CODATA value in MHz/G.
- **gS** (*float*) – Electron spin g-factor. Default: CODATA value.
- **gL** (*float*) – Orbital g-factor. Note that it may deviate slightly from 1 due to relativistic, diamagnetic, and non-adiabatic contributions. Default: $g_L = 1$.
- **gr** (*float*) – Rotational g-factor. Default: 0.
- **gl** (*float*) – Anisotropic electron spin g-factor. Default: 0.
- **glprime** (*float*) – Parity-dependent anisotropic electron spin g-factor. A reasonable approximation is that $g_l' \sim p/2B$. Default: 0.
- **greprime** (*float*) – Parity-dependent electron contribution to rotational g-factor. A resonable approximation is that $g_{re}' \sim -q/B$. Default: 0.
- **gN** (*float*) – Nuclear spin g-factor. It changes negligibly in most molecules. Default: 0.
- **return_basis** (*boolean, optional*) – Boolean to specify whether to return the basis as well as the Hamiltonian matrices. Default: True.

Notes

Assumes the A state is in HundâĂŹs case (a), namely $|\Lambda, J, \Omega, I, F, m_F, P\rangle$. By definition, $\Sigma = \Omega - \Lambda$. For the A state, $\Sigma = 1/2$, $\Lambda = 1$, and then $\Omega = 1/2$. The full Hamiltonian is a combination of a Brown and Carrington, *Rotational Spectroscopy of Diatomic Molecules*, Eqs. 6.196 (rotation), 8.401 (Λ-doubling), 8.372 (nuclear spin-orbit coupling), 8.374 (Fermi contact interaction), 8.506 (quadrupole), 9.57, 9.58, 9.59, 9.60, 9.70, and 9.71 (Zeeman interaction). See the comments in the code for more details on equations used and approximations made. Most Hamiltonian parameters are both keyword arguments and by default zero so that the user can easily turn on only the relavent terms easily.

pylcp.hamiltonians.XFmolecules.**Xstate**(*N*, *I*, *B=0.0*, *gamma=0.0*, *b=0.0*, *c=0.0*, *CI=0.0*, *q0=0*, *q2=0*, *gS=2.00231930436256*, *gI=5.5856946893*, *muB=1.39962449361*, *muN=0.0007622593229068622*, *return_basis=False*)

Defines the field-free and magnetic field-dependent components of the $X^2\Sigma^+$ ground state Hamiltonian.

Parameters

- **N** (*int*) – Rotational quantum number.
- **I** (*int or float*) – Nuclear spin quantum number of the fluorine, usually 1/2
- **B** (*float*) – Rotational constant. Default: 0.
- **gamma** (*float*) – Electron-spin rotational coupling constant. Default: 0.
- **b** (*float*) – Isotropic spin-spin interaction. Default: 0.
- **c** (*float*) – Anisotropic spin-spin interaction. Default: 0.
- **CI** (*float*) – Nuclear-spin rotational coupling constant. Default: 0.
- **q0** (*float*) – Electron quadrupole constant. Default: 0.
- **q2** (*float*) – Electron quadrupole constant. Default: 0.
- **gS** (*float*) – Electron spin g-factor. Default: CODATA value.
- **gI** (*float*) – Nuclear (proton) g-factor. Default: CODATA value.
- **muB** (*float, optional*) – Bohr Magneton. Default value is the CODATA value in MHz/G.
- **muN** (*float*) – Nuclear Magneton. Default value is the CODATA value in MHz/G
- **return_basis** (*boolean, optional*) – Boolean to specify whether to return the basis as well as the Hamiltonian matrices. Default: True.

Notes

Assuming HundâĂŹs case (b) with basis $|\Lambda, N, \Sigma, J, F, m_F, P\rangle$, $\Lambda = 0$ and $\Sigma = 1/2$. The full Hamiltonian is a combination of a Brown and Carrington, *Rotational Spectroscopy of Diatomic Molecules*, Eqs. 9.88 (rotation), 9.89 (spin-rotation), 9.90 (hyperfine), 9.91 (dipole-dipole interaction), 9.53 (electric quadrupole), 8.183 (electronic spin Zeeman) and 8.185 (nuclear spin Zeeman). See the comments in the code for more details on equations used and approximations made. Most Hamiltonian parameters are both keyword arguments and by default zero so that the user can easily turn on only the relavent terms easily.

pylcp.hamiltonians.XFmolecules.**dipoleXandAstates**(*xbasis*, *abasis*, *I=0.5*, *S=0.5*, *UX=[]*, *return_intermediate=False*)

Calculate the oscillator strengths between the X and A states.

Parameters

- **xbasis** (*list or array_like*) – List of basis vectors for the X state

- **abasis** (*list or array_like*) – List of basis vectors for the A state
- **I** (*int or float*) – Nuclear spin angular momentum. Default: 1/2.
- **S** (*int or float*) – Σ quantum number. Default: 1/2.
- **UX** (*two-dimensional array, optional*) – a rotation matrix for case (b) into the intermediate eigenbasis. Default: empty
- **return_intermediate** (*boolean, optional*) – Argument to return the intermediate bases and transformation matrices.

Notes

The X state is assumed to be HundâĂŹs case (b) while the A state is assumed to be HundâĂŹs case (a). Thus, this function makes an intermediate basis to transform between the two.

4.1.2 Overview

singleF(F[, gF, muB, return_basis])	Construct the Hamiltonian for a lonely angular momentum state
hyperfine_uncoupled(J, I, gJ, gI, Ahfs[, ...])	Construct the hyperfine Hamiltonian in the coupled basis.
hyperfine_coupled(J, I, gJ, gI, Ahfs[, ...])	Construct the hyperfine Hamiltonian in the coupled basis.
fine_structure_uncoupled(L, S, I, xi, a_c, ...)	Returns the full fine structure manifold in the uncoupled basis.
dqij_two_bare_hyperfine(F, Fp[, normalize])	Calculates the dqij matrix for two bare hyperfine states.
dqij_two_hyperfine_manifolds(J, Jp, I[, ...])	Dipole matrix element matrix elements for transitions between two hyperfine manifolds.
dqij_two_fine_stucture_manifolds_uncoupled(...)	Return the coupling between two fine structure manifolds

4.1.3 Detailed functions

pylcp.hamiltonians.**dqij_two_bare_hyperfine**(*F*, *Fp*, *normalize=True*)

Calculates the dqij matrix for two bare hyperfine states. Specifically, it returns the matrix of the operator d_q, where a photon is created by a transition from the excited state to the ground state.

Parameters

- **F** (*integer or float (half integer)*) – Total angular momentum quantum number of the F state.
- **Fp** (*integer or float (half integer)*) – Total angular momentum quantum number of the FâĂŹ state.
- **normalize** (*boolean*) – By default, âĂŸnormalizeâĂŹ is True

pylcp.hamiltonians.**dqij_two_fine_stucture_manifolds_uncoupled**(*basis_g*, *basis_e*)

Return the coupling between two fine structure manifolds

Parameters

- **basis_g** (*list or array_like*) – A list of the basis vectors for the ground state. In the uncoupled basis, they are of the form $|m_L, m_S, m_I\rangle$
- **basis_e** (*list or array_like*) – A list of the basis vectors for the ground state. In the uncoupled basis, they are of the form $|m'_L, m'_S, m'_I\rangle$

Returns

d_q – The dipole coupling array. N is the number of ground states and M is the number of excited states.

Return type

array with shape (3, N, M)

pylcp.hamiltonians.**dqij_two_hyperfine_manifolds**(*J*, *Jp*, *I*, *normalize=True*, *return_basis=False*)

Dipole matrix element matrix elements for transitions between two hyperfine manifolds.

Parameters

- **J** (*int or float*) – Lower hyperfine manifold J quantum number
- **Jp** (*int or float*) – Upper hyperfine manifold J' quantum number
- **I** (*int or float*) – Nuclear spin associated with both manifolds
- **normalize** (*boolean, optional*) – Normalize the d_q to one. Default: True
- **return_basis** (*boolean, optional*) – If true, returns the basis states as well as the d_q

Returns

- **d_q** (*array_like*) – Dipole matrix elements between hyperfine manifolds
- **basis_g** (*list*) – If return_basis is true, list of (F, m_F)
- **basis_e** (*list*) – If return_basis is true, list of (F', m'_F)

pylcp.hamiltonians.**fine_structure_uncoupled**(*L*, *S*, *I*, *xi*, *a_c*, *a_orb*, *a_dip*, *gL*, *gS*, *gI*, *muB=1399624.49361*, *return_basis=False*)

Returns the full fine structure manifold in the uncoupled basis.

Parameters

- **L** (*int*) – Orbital angular momentum of interest
- **S** (*int or float*) – Spin angular momentum of interest
- **I** (*int or float*) – Nuclear angular momentum of interest
- **xi** (*float*) – Fine structure splitting
- **a_c** (*float*) – Contact interaction constant
- **a_orb** (*float*) – Orbital interaction constant
- **a_dip** (*float*) – Dipole interaction constant
- **gL** (*float*) – Orbital g-factor
- **gS** (*float*) – Spin g-factor
- **gI** (*float*) – Nuclear g-factor
- **muB** (*float, optional*) – Bohr magneton. Default: the CODATA value in Hz/G
- **return_basis** (*bool, optional*) – Return the basis vectors as well as gthe

Returns

- **H_0** (*array (NxN)*) – Field free Hamiltonian, where N is the number of states
- **mu_q** (*array (3xNxN)*) – Zeeman splitting array

Notes

See J.D.Lyons and T.P.Das, Phys.Rev.A,2,2250 (1970) and H.Orth et al,Z.Physik A,273,221 (1975) for details of Hamiltonian and splitting constants.

This function is adapted from the one found in Tollet, âĂIJPermanent magnetic trap for Li atomsâĂİ, thesis, Rice University, 1994.

pylcp.hamiltonians.**hyperfine_coupled**(*J*, *I*, *gJ*, *gI*, *Ahfs*, *Bhfs=0*, *Chfs=0*, *muB=1399624.49361*, *return_basis=False*)

Construct the hyperfine Hamiltonian in the coupled basis.

For parameterization of this Hamiltonian, see Steck, Alkali D line data, which contains a useful description of the hyperfine Hamiltonian.

Parameters

- **J** (*int or float*) – Lower hyperfine manifold J quantum number
- **I** (*int or float*) – Nuclear spin associated with both manifolds
- **gJ** (*float*) – Electronic Lande g-factor
- **gI** (*float*) – Nuclear g-factor
- **Ahfs** (*float*) – Hyperfine A parameter
- **Bhfs** (*float, optional*) – Hyperfine B parameter. Default: 0.
- **Chfs** (*float, optional*) – Hyperfine C parameter. Default: 0.
- **muB** (*float, optional*) – Bohr magneton. Default: the CODATA value in Hz/G
- **return_basis** (*boolean, optional*) – If true, return the basis. Default: False

Returns

- **H_0** (*array_like*) – Field independent component of the Hamiltonian
- **mu_q** (*array_like*) – Magnetic field dependent component of the Hamiltonian
- **basis** (*list*) – List of (F, m_F) basis states

pylcp.hamiltonians.**hyperfine_uncoupled**(*J*, *I*, *gJ*, *gI*, *Ahfs*, *Bhfs=0*, *Chfs=0*, *muB=1399624.49361*, *return_basis=False*)

Construct the hyperfine Hamiltonian in the coupled basis.

For parameterization of this Hamiltonian, see Steck, Alkali D line data, which contains a useful description of the hyperfine Hamiltonian.

Parameters

- **J** (*int or float*) – Lower hyperfine manifold J quantum number
- **I** (*int or float*) – Nuclear spin associated with both manifolds
- **gJ** (*float*) – Electronic Lande g-factor
- **gI** (*float*) – Nuclear g-factor
- **Ahfs** (*float*) – Hyperfine A parameter

- **Bhfs** (*float, optional*) – Hyperfine B parameter. Default: 0.
- **Chfs** (*float, optional*) – Hyperfine C parameter. Default: 0.
- **muB** (*float, optional*) – Bohr magneton. Default: the CODATA value in Hz/G
- **return_basis** (*boolean, optional*) – If true, return the basis. Default: False

Returns

- **H_0** (*array_like*) – Field independent component of the Hamiltonian
- **mu_q** (*array_like*) – Magnetic field dependent component of the Hamiltonian
- **basis** (*list*) – List of (J, I, m_J, m_I) basis states

pylcp.hamiltonians.**singleF**(*F*, *gF=1*, *muB=1399624.49361*, *return_basis=False*)

Construct the Hamiltonian for a lonely angular momentum state

Parameters

- **F** (*int or float*) – Angular momentum quantum number
- **gF** (*float*) – Associated Lande g-factor
- **muB** (*float, optional*) – Bohr magneton. Default: the CODATA value in Hz/G
- **return_basis** (*boolean, optional*) – If true, return the basis. Default: False

Returns

- **H_0** (*array_like*) – Field independent component of the Hamiltonian
- **mu_q** (*array_like*) – Magnetic field dependent component of the Hamiltonian
- **basis** (*list*) – List of (F, m_F) basis states

4.2 Hamiltonian Class

This is the main Hamiltonian class that combines everything together to make a complete Hamiltonian.

4.2.1 Overview

hamiltonian(*args[, mass, muB, gamma, k])	A representation of the Hamiltonian in blocks

4.2.2 Detailed functions

class pylcp.**hamiltonian**(**args*, *mass=1.0*, *muB=1*, *gamma=1.0*, *k=1*)

A representation of the Hamiltonian in blocks

Diagonal blocks describe the internal structure of a manifold, and off-diagonal blocks describe how those manifolds are connected via laser Beams and the associated dipole matrix elements. For most cases, the Hamiltonian is usually just a two level system. In this case, the Hamiltonian can be initiated using the optional parameters below and the two manifolds are given the labels g and e. You must supply the five positional arguments below in order to initiate the Hamiltonian in this way.

For other constructions with more than two manifolds, one should construct the Hamiltonian using the *pylcp.hamiltonian.add_H_0_block()*, *pylcp.hamiltonian.add_mu_q_block()* and

pylcp.hamiltonian.add_d_q_block(). Note that the order in which the diagonal blocks are added is the energy ordering of the manifolds, which is often obscured after the rotating wave approximation is taken (and implicitly assumed to be taken before construction of this Hamiltonian object).

For more information, see the accompanying paper that describes the block nature of the Hamiltonian.

Parameters

- **H0_g** (*array_like, shape (N, N), optional*) – Ground manifold field-independent matrix
- **H0_e** (*array_like, shape (M, M), optional*) – Excited manifold field-independent matrix
- **muq_g** (*array_like, shape (3, N, N), optional*) – Ground manifold magnetic field-dependent component, in spherical basis.
- **muq_e** (*array_like, shape (3, M, M), optional*) – Excited manifold magnetic field-dependent component, in spherical basis.
- **d_q** (*array_like, shape (3, N, M), optional*) – Dipole operator that connects the ground and excited manifolds, in spherical basis.
- **mass** (*float*) – Mass of the atom or molecule
- **muB** (*Bohr magneton*) – Value of the Bohr magneton in the units of choice
- **gamma** (*float*) – Value of the decay rate associated with d_q
- **k** (*float*) – Value of the wavevector associated with d_q

ns

Total number of states in the Hamiltonian

Type
int

state_labels

Updated list of the state labels used in the Hamiltonian.

Type
list of char

laser_keys

The laser keys dictionary translates laser pumping keys like *g->e* into block indices for properly extracting the associated d_q matrix.

Type
dict

add_H_0_block(*state_label*, *H_0*)

Adds a new H_0 block to the hamiltonian

Parameters

- **state_label** (*str*) – Label for the manifold for which this new block applies
- **H_0** (*array_like, with shape (N, N)*) – Square matrix that describes the field-independent part of this manifoldâĂŹs Hamiltonian. This manifold must have N states.

add_d_q_block(*label1*, *label2*, *d_q*, *k=1*, *gamma=1*)

Adds a new d_q block to the hamiltonian to connect two manifolds together.

Parameters

- **label1** (*str*) – Label for the first manifold to which this block applies
- **label2** (*str*) – Label for the second manifold to which this block applies
- **d_q** (*array_like, with shape (3, N, M)*) – Matrix that describes the electric field dependent part of this dipole matrix element. The first manifold must
- **k** (*float, optional*) – The mangitude of the k-vector for this d_q block. Default: 1
- **gamma** (*float, optional*) – The mangitude of the decay rate associated with this d_q block. Default: 1

add_mu_q_block(*state_label*, *mu_q*, *muB=1*)

Adds a new mu_q block to the hamiltonian

Parameters

- **state_label** (*str*) – Label for the manifold for which this new block applies
- **mu_q** (*array_like, with shape (3, N, N)*) – Square matrix that describes the magnetic field dependent part of this manifoldâĂŹs Hamiltonian.

diag_static_field(*B*)

Block diagonalize at a specified magnetic field

This function diagonalizes the HamiltonianâĂŹs diagonal blocks separately based on the value of the static magnetic field B, and then rotates the d_q sets the quantization axis, and they rotate the coordinate system appropriately, so we only ever need to consider the z-component of the field.

Parameters

- **B** (*float*) – The magnetic field value at which to diagonalize. It is always assumed to be along the $\hat{z}$ direction.
- **Returns** –
- **H** (pylcp.hamiltonian) – A block-structured Hamiltonian with diagonal elemented diagonalized and d_q objects rotated

make_full_matrices()

Returns the full matrices that define the Hamiltonian.

Assembles the full Hamiltonian matrices from the stored block representation, and returns the Hamiltonian in the appropriate parts. For this function, n is the number of states

Returns

- **H_0** (*array_like, shape (n, n)*) – The diagonal portion of the Hamiltonian
- **mu_q** (*array_like, shape (3, N, N)*) – The magnetic field dependent portion, in spherical basis.
- **d_q** (*dictionary of array_like, shape (3, N, N)*) – The electric field dependent portion, in spherical basis, arranged by keys that describe the manifolds connected by the specific d_q. This usually gets paired with E^*
- **d_q_star** (*dictionary of array_like, shape (3, N, N)*) – The electric field dependent portion, in spherical basis, arranged by keys that describe the manifolds connected by the specific d_q. This usually gets paired with E

print_structure()

Print structure of the Hamiltonian

return_full_H(*Eq*, *Bq*)

Assemble the block diagonal Hamiltonian into a single matrix

Parameters

- **Eq** (*array_like or dictionary of array_like*) – The electric field(s) driving transitions between manifolds, each expressed in the spherical basis. Each electric field driving a transition between manifolds needs to specified with the correct key in the dictionary. For example, for a two-manifold Hamiltonian with manifold labels *g* and *e*, the dictionary should contain a single entry with *g->e*. If the electric field is given as a single array_like, it is assumed to drive the *g->e* transition.
- **Bq** (*array_like, shape (3,)*) – The magnetic field in spherical basis.

Returns
H – The full Hamiltonian matrix

Return type
array_like

set_mass(*mass*)

Sets the HamiltonianâĂŹs mass parameter

Parameters
mass (*float*) – The mass of the atom or molecule of the Hamiltonian

4.3 Magnetic Fields

Objects for creating magnetic field objects, along with some common components.

4.3.1 Overview

magField(field[, eps])	Base magnetic field class
constantMagneticField(B0)	Spatially constant magnetic field
quadrupoleMagneticField(alpha[, eps])	Spherical quadrupole magnetic field
iPMagneticField(B0, B1, B2[, eps])	Ioffe-Pritchard trap magnetic field

4.3.2 Details

class pylcp.**magField**(*field*, *eps=1e-05*)

Base magnetic field class

Stores a magnetic defined magnetic field and calculates useful derivatives for *pylcp*.

Parameters

- **field** (*array_like with shape (3,) or callable*) – If constant, the magnetic field vector, specified as either as an array_like with shape (3,). If a callable, it must have a signature like (R, t), (R), or (t) where R is an array_like with shape (3,) and t is a float and it must return an array_like with three elements.
- **eps** (*float, optional*) – Small distance to use in calculation of numerical derivatives. By default *eps=1e-5*.

eps

small epsilon used for computing derivatives

Type
float

FieldMag(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Magnetic field magnitude at R and t:

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
B – the magnetic field mangitude at position R and time t.

Return type
float

gradField(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Full spaitial derivative of the magnetic field at R and t:

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

dB – the full gradient of the magnetic field, with elements

$$\begin{pmatrix} \frac{dB_x}{dx} & \frac{dB_y}{dx} & \frac{dB_z}{dx} \\ \frac{dB_x}{dy} & \frac{dB_y}{dy} & \frac{dB_z}{dy} \\ \frac{dB_x}{dz} & \frac{dB_y}{dz} & \frac{dB_z}{dz} \end{pmatrix}$$

Return type
array_like, shape (3, 3)

Notes

This method calculates the derivative stupidly, just using first order numerical differentiation using the *eps* parameter.

gradFieldMag(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Gradient of the magnetic field magnitude at R and t:

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
dB – $\nabla|B|$, the gradient of the magnetic field magnitude at position R and time t.

Return type
array_like, shape (3,)

class pylcp.**constantMagneticField**(*B0*)

Spatially constant magnetic field

Represents a magnetic field of the form

$$\mathbf{B} = \mathbf{B}_0$$

Parameters
val (*array_like with shape (3,)*) – The three-vector defintion of the constant magnetic field.

gradField(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Gradient of the magnetic field magnitude at R and t:

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
dB – $\nabla|B| = 0$, the gradient of the magnitude of a constant magnetic field is always zero.

Return type
np.zeros((3,))

gradFieldMag(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Gradient of the magnetic field magnitude at R and t:

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
dB – The gradient of a constant magnetic field magnitude is always zero.

Return type
np.zeros((3,))

class pylcp.**quadrupoleMagneticField**(*alpha*, *eps=1e-05*)

Spherical quadrupole magnetic field

Represents a magnetic field of the form

$$\mathbf{B} = \alpha\left(-\frac{x\hat{x}}{2} - \frac{y\hat{y}}{2} + z\hat{z}\right)$$

Parameters
alpha (*float*) – strength of the magnetic field gradient.

gradField(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Full spaitial derivative of the magnetic field at R and t:

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

dB – the full gradient of the magnetic field, with elements

$$\begin{pmatrix} -\alpha/2 & 0 & 0 \\ 0 & -\alpha/2 & 0 \\ 0 & 0 & \alpha \end{pmatrix}$$

Return type
array_like, shape (3, 3)

class pylcp.**iPMagneticField**(*B0*, *B1*, *B2*, *eps=1e-05*)

Ioffe-Pritchard trap magnetic field

Generates a magnetic field of the form

$$\mathbf{B} = B_1 x\hat{x} - B_1 y\hat{y} + \left(B_0 + \frac{B_2}{2}z^2\right)\hat{z}$$

Parameters

- **B0** (*float*) – Constant offset field
- **B1** (*float*) – Magnetic field gradient in x-y plane
- **B2** (*float*) – Magnetic quadratic component along z direction.

Notes

It is currently missing extra terms that are required for it to fulfill MaxwellâĂŹs equations at second order.

gradField(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Full spaitial derivative of the magnetic field at R and t:

Parameters

- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

dB – the full gradient of the magnetic field, with elements

$$\begin{pmatrix} \frac{dB_x}{dx} & \frac{dB_y}{dx} & \frac{dB_z}{dx} \\ \frac{dB_x}{dy} & \frac{dB_y}{dy} & \frac{dB_z}{dy} \\ \frac{dB_x}{dz} & \frac{dB_y}{dz} & \frac{dB_z}{dz} \end{pmatrix}$$

Return type

array_like, shape (3, 3)

Notes

This method calculates the derivative stupidly, just using first order numerical differentiation using the *eps* parameter.

gradFieldMag(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Gradient of the magnetic field magnitude at R and t:

Parameters

- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

dB – $\nabla|B|$, the gradient of the magnetic field magnitude at position R and time t.

Return type

array_like, shape (3,)

4.4 Laser Fields

Objects for creating laser beam objects, along with some common components.

4.4.1 Overview

laserBeam([kvec, s, pol, delta, phase, ...])	The base class for a single laser beam
infinitePlaneWaveBeam(kvec, pol, s, delta, ...)	Infinte plane wave beam
gaussianBeam(kvec, pol, s, delta, wb, **kwargs)	Collimated Gaussian beam
clippedGaussianBeam(kvec, pol, s, delta, wb, ...)	Clipped, collimated Gaussian beam
laserBeams([laserbeamparams, beam_type])	The base class for a collection of laser beams
conventional3DMOTBeams([k, pol, ...])	A collection of laser beams for 6-beam MOT

4.4.2 Details

class pylcp.**laserBeam**(*kvec=None*, *s=None*, *pol=None*, *delta=None*, *phase=0.0*, *pol_coord='spherical'*, *eps=1e-05*)

The base class for a single laser beam

Attempts to represent a laser beam as

$$\frac{1}{2}\hat{\boldsymbol{\epsilon}}(r,t)E_0(r,t)e^{i\mathbf{k}(r,t)\cdot\mathbf{r}-i\int dt\Delta(t)+i\phi(r,t)}$$

where $\hat{\epsilon}$ is the polarization, E_0 is the electric field magnitude, $\mathbf{k}(r,t)$ is the k-vector, $\mathbf{r}$ is the position, $\Delta(t)$ is the deutning, t is the time, and ϕ is the phase.

Parameters

- **kvec** (*array_like with shape (3,) or callable*) – The k-vector of the laser beam, specified as either a three-element list or numpy array or as callable function. If a callable, it must have a signature like (R, t), (R), or (t) where R is an array_like with shape (3,) and t is a float and it must return an array_like with three elements.
- **pol** (*int, float, array_like with shape (3,), or callable*) – The polarization of the laser beam, specified as either an integer, float array_like with shape(3,), or as callable function. If an integer or float, if *pol<0* the polarization will be left circular polarized relative to the k-vector of the light. If *pol>0*, the polarization will be right circular polarized. If array_like, polarization will be specified by the vector, whose basis is specified by *pol_coord*. If a callable, it must have a signature like (R, t), (R), or (t) where R is an array_like with shape (3,) and t is a float and it must return an array_like with three elements.
- **s** (*float or callable*) – The intensity of the laser beam, normalized to the saturation intensity, specified as either a float or as callable function. If a callable, it must have a signature like (R, t), (R), or (t) where R is an array_like with shape (3,) and t is a float and it must return a float.
- **delta** (*float or callable*) – Detuning of the laser beam. If a callable, it must have a signature like (t) where t is a float and it must return a float.
- **phase** (*float, optional*) – Phase of laser beam. By default, zero.
- **pol_coord** (*string, optional*) – Polarization basis of the input polarization vector: âĂŸcartesianâĂŹ or âĂŸsphericalâĂŹ (default).

- **eps** (*float, optional*) – Small distance to use in calculation of numerical derivatives. By default *eps=1e-5*.

eps

Small epsilon used for computing derivatives

Type
float

phase

Overall phase of the laser beam.

Type
float

cartesian_pol(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Returns the polarization in Cartesian coordinates.

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
pol – polarization of the laser beam at R and t in Cartesian basis.

Return type
array_like, size (3,)

delta(*t=0.0*)

Returns the detuning of the laser beam at time t

Parameters
t (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
delta – detuning of the laser beam at time t

Return type
float or array like

electric_field(*R*, *t*)

The electric field at position R and t

Parameters
- **R** (*array_like, size (3,)*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float*) – time at which to return the k-vector. By default, t=0.

Returns
Eq – electric field in the spherical basis.

Return type
array_like, shape (3,)

electric_field_gradient(*R*, *t*)

The full derivative of electric field at position R and t

Parameters
- **R** (*array_like, size (3,)*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float*) – time at which to return the k-vector. By default, t=0.

Returns

dEq – The full gradient of the electric field, in spherical coordinates.

$$\begin{pmatrix} \frac{dE_{-1}}{dx} & \frac{dE_0}{dx} & \frac{dE_{+1}}{dx} \\ \frac{dE_{-1}}{dy} & \frac{dE_0}{dy} & \frac{dE_{+1}}{dy} \\ \frac{dE_{-1}}{dz} & \frac{dE_0}{dz} & \frac{dE_{+1}}{dz} \end{pmatrix}$$

Return type
array_like, shape (3, 3)

`intensity`(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the intensity of the laser beam at position R and t

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
s – Saturation parameter of the laser beam at R and t.

Return type
float or array_like

`jones_vector`(*xp*, *yp*, *R=array([0.0, 0.0, 0.0])*, *t=0*)

Returns the Jones vector at position

Parameters
- **xp** (*array_like, shape (3,)*) – The x vector of the basis in which to calculate the Jones vector. Must be orthogonal to k.
- **yp** (*array_like, shape (3,)*) – The y vector of the basis in which to calculate the Jones vector. Must be orthogonal to k and *xp*.
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
pol – Jones vector of the laser beam at R and t in Cartesian basis.

Return type
array_like, size (2,)

`kvec`(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the k-vector of the laser beam

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
kvec – the k vector at position R and time t.

Return type
array_like, size(3,)

`pol`(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the polarization of the laser beam at position R and t

The polarization is returned in the spherical basis.

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
pol – polarization of the laser beam at R and t in spherical basis.

Return type
array_like, size (3,)

`polarization_ellipse`(*xp*, *yp*, *R=array([0.0, 0.0, 0.0])*, *t=0*)

The polarization ellipse parameters of the laser beam at R and t

Parameters

- **xp** (*array_like, shape (3,)*) – The x vector of the basis in which to calculate the polarization ellipse. Must be orthogonal to k.
- **yp** (*array_like, shape (3,)*) – The y vector of the basis in which to calculate the polarization ellipse. Must be orthogonal to k and *xp*.
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

- **psi** (*float*) – ψ parameter of the polarization ellipse
- **chi** (*float*) – χ parameter of the polarization ellipse

project_pol(*quant_axis*, *R=array([0.0, 0.0, 0.0])*, *t=0*, *treat_nans=False*, *calculate_norm=False*, *invert=False*)

Project the polarization onto a quantization axis.

Parameters

- **quant_axis** (*array_like, shape (3,)*) – A normalized 3-vector of the quantization axis direction.
- **R** (*array_like, shape (3,), optional*) – If polarization is a function of R is the 3-vectors at which the polarization shall be calculated.
- **calculate_norm** (*bool, optional*) – If true, renormalizes the quant_axis. By default, False.
- **treat_nans** (*bool, optional*) – If true, every place that nan is encoutnered, replace with the $hat{z}$ axis as the quantization axis. By default, False.
- **invert** (*bool, optional*) – If true, invert the process to project the quantization axis onto the specified polarization.

Returns

projected_pol – The polarization projected onto the quantization axis.

Return type

array_like, shape (3,)

stokes_parameters(*xp*, *yp*, *R=array([0.0, 0.0, 0.0])*, *t=0*)

The Stokes Parameters of the laser beam at R and t

Parameters

- **xp** (*array_like, shape (3,)*) – The x vector of the basis in which to calculate the Stokes parameters. Must be orthogonal to k.
- **yp** (*array_like, shape (3,)*) – The y vector of the basis in which to calculate the Stokes parameters. Must be orthogonal to k and *xp*.
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

pol – Stokes parameters for the laser beam, [Q, U, V]

Return type

array_like, shape (3,)

class pylcp.**infinitePlaneWaveBeam**(*kvec*, *pol*, *s*, *delta*, ***kwargs*)

Infinte plane wave beam

A beam which has spatially constant intensity, k-vector, and polarization.

$$\frac{1}{2}\hat{\boldsymbol{\epsilon}}E_0 e^{i\mathbf{k}\cdot\mathbf{r}-i\int dt\Delta(t)+i\phi(r,t)}$$

where $\hat{\boldsymbol{\epsilon}}$ is the polarization, E_0 is the electric field magnitude, $\mathbf{k}(r,t)$ is the k-vector, $\mathbf{r}$ is the position, $\Delta(t)$ is the deutning, t is the time, and ϕ is the phase.

Parameters

- **kvec** (*array_like with shape (3,) or callable*) – The k-vector of the laser beam, specified as either a three-element list or numpy array.
- **pol** (*int, float, array_like with shape (3,), or callable*) – The polarization of the laser beam, specified as either an integer, float array_like with shape(3,). If an integer or float, if *pol<0* the polarization will be left circular polarized relative to the k-vector of the light. If *pol>0*, the polarization will be right circular polarized. If array_like, polarization will be specified by the vector, whose basis is specified by *pol_coord*.
- **s** (*float or callable*) – The intensity of the laser beam, specified as either a float or as callable function.
- **delta** (*float or callable*) – Detuning of the laser beam. If a callable, it must have a signature like (t) where t is a float and it must return a float.
- ****kwargs** – Additional keyword arguments to pass to laserBeam superclass.

Notes

This implementation is much faster, when it can be used, compared to the base laserBeam class.

electric_field_gradient(*R*, *t*)

The full derivative of electric field at position R and t

Parameters

- **R** (*array_like, size (3,)*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float*) – time at which to return the k-vector. By default, t=0.

Returns

dEq – The full gradient of the electric field, in spherical coordinates.

$$\begin{pmatrix} \frac{dE_{-1}}{dx} & \frac{dE_0}{dx} & \frac{dE_{+1}}{dx} \\ \frac{dE_{-1}}{dy} & \frac{dE_0}{dy} & \frac{dE_{+1}}{dy} \\ \frac{dE_{-1}}{dz} & \frac{dE_0}{dz} & \frac{dE_{+1}}{dz} \end{pmatrix}$$

Return type

array_like, shape (3, 3)

class pylcp.**gaussianBeam**(*kvec*, *pol*, *s*, *delta*, *wb*, ***kwargs*)

Collimated Gaussian beam

A beam which has spatially constant k-vector and polarization, with a Gaussian intensity modulation. Specifically,

$$\frac{1}{2}\hat{\boldsymbol{\epsilon}} E_0 e^{-\mathbf{r}^2/w_b^2} e^{i\mathbf{k}\cdot\mathbf{r} - i\int dt \Delta(t) + i\phi(r,t)}$$

where $\hat{\boldsymbol{\epsilon}}$ is the polarization, E_0 is the electric field magnitude, $\mathbf{k}(r,t)$ is the k-vector, $\mathbf{r}$ is the position, $\Delta(t)$ is the deutning, t is the time, and ϕ is the phase. Note that because $I \propto E^2$, w_b is the $1/e^2$ radius.

Parameters

- **kvec** (*array_like with shape (3,) or callable*) – The k-vector of the laser beam, specified as either a three-element list or numpy array.
- **pol** (*int, float, array_like with shape (3,), or callable*) – The polarization of the laser beam, specified as either an integer, float array_like with

shape(3,). If an integer or float, if *pol<0* the polarization will be left circular polarized relative to the k-vector of the light. If *pol>0*, the polarization will be right circular polarized. If array_like, polarization will be specified by the vector, whose basis is specified by *pol_coord*.

- **s** (*float or callable*) – The maximum intensity of the laser beam at the center, specified as either a float or as callable function.
- **delta** (*float or callable*) – Detuning of the laser beam. If a callable, it must have a signature like (t) where t is a float and it must return a float.
- **wb** (*float*) – The $1/e^2$ radius of the beam.
- ****kwargs** – Additional keyword arguments to pass to the laserBeam superclass.

intensity(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the intensity of the laser beam at position R and t

Parameters

- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

s – Saturation parameter of the laser beam at R and t.

Return type

float or array_like

class pylcp.**clippedGaussianBeam**(*kvec*, *pol*, *s*, *delta*, *wb*, *rs*, ***kwargs*)

Clipped, collimated Gaussian beam

A beam which has spatially constant k-vector and polarization, with a Gaussian intensity modulation. Specifically,

$$\frac{1}{2}\hat{\boldsymbol{\epsilon}}E_0 e^{-\mathbf{r}^2/w_b^2}(|\mathbf{r}| < r_s)e^{i\mathbf{k}\cdot\mathbf{r}-i\int dt\Delta(t)+i\phi(r,t)}$$

where $\hat{\boldsymbol{\epsilon}}$ is the polarization, E_0 is the electric field magnitude, r_s is the radius of the stop, $\mathbf{k}(r,t)$ is the k-vector, $\mathbf{r}$ is the position, $\Delta(t)$ is the deutning, t is the time, and ϕ is the phase. Note that because $I \propto E^2$, w_b is the $1/e^2$ radius.

Parameters

- **kvec** (*array_like with shape (3,) or callable*) – The k-vector of the laser beam, specified as either a three-element list or numpy array.
- **pol** (*int, float, array_like with shape (3,), or callable*) – The polarization of the laser beam, specified as either an integer, float array_like with shape(3,). If an integer or float, if *pol<0* the polarization will be left circular polarized relative to the k-vector of the light. If *pol>0*, the polarization will be right circular polarized. If array_like, polarization will be specified by the vector, whose basis is specified by *pol_coord*.
- **s** (*float or callable*) – The maximum intensity of the laser beam at the center, specified as either a float or as callable function.
- **delta** (*float or callable*) – Detuning of the laser beam. If a callable, it must have a signature like (t) where t is a float and it must return a float.
- **wb** (*float*) – The $1/e^2$ radius of the beam.
- **rs** (*float*) – The radius of the stop.
- ****kwargs** – Additional keyword arguments to pass to the laserBeam superclass.

intensity(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the intensity of the laser beam at position R and t

Parameters

- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

s – Saturation parameter of the laser beam at R and t.

Return type

float or array_like

class pylcp.**laserBeams**(*laserbeamparams=None*, *beam_type=<class 'pylcp.fields.laserBeam'>*)

The base class for a collection of laser beams

Parameters

- **laserbeamparams** (*array_like of laserBeam or array_like of dictionaries*) – If array_like contains laserBeams, the laserBeams in the array will be joined together to form a collection. If array_like is a list of dictionaries, the dictionaries will be passed as keyword arguments to beam_type
- **beam_type** (laserBeam *or laserBeam subclass, optional*) – Type of beam to use in the collection of laserBeams. By default *beam_type=laserBeam*.

add_laser(*new_laser*)

Add a laser to the collection

Parameters

new_laser (laserBeam *or laserBeam subclass*) –

cartesian_pol(*R=array([0.0, 0.0, 0.0])*, *t=0*)

Returns the polarization of all laser beams in Cartesian coordinates.

Parameters

- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the polarization. By default, the origin.
- **t** (*float, optional*) – time at which to return the polarization. By default, t=0.

Returns

pol – polarization of the laser beam at R and t in Cartesian basis.

Return type

array_like, shape (num_of_beams, 3)

delta(*t=0*)

Returns the detuning of the laser beam at time t

Parameters

t (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

delta – detuning of the laser beam at time t for all laser beams

Return type

float or array like

electric_field(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the electric field of the laser beams

Parameters

- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns

E – the electric field vectors at position R and time t for each laser beam.

Return type
list of array_like, size(3,)

`electric_field_gradient`(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the gradient of the electric field of the laser beams

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
dE – the electric field gradient matrices at position R and time t for each laser beam.

Return type
list of array_like, size(3,)

`intensity`(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the intensity of the laser beam at position R and t

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
s – Saturation parameters of all laser beams at R and t.

Return type
list of float or array_like

`jones_vector`(*xp*, *yp*, *R=array([0.0, 0.0, 0.0])*, *t=0*)

Jones vector at position R and time t

Parameters
- **xp** (*array_like, shape (3,)*) – The x vector of the basis in which to calculate the Jones vector. Must be orthogonal to k.
- **yp** (*array_like, shape (3,)*) – The y vector of the basis in which to calculate the Jones vector. Must be orthogonal to k and *xp*.
- **R** (*array_like, size (3,), optional*) – vector of the position at which to evaluate the Jones vector. By default, the origin.
- **t** (*float, optional*) – time at which to evaluate the Jones vector. By default, t=0.

Returns
pol – Jones vector of the laser beams at R and t in Cartesian basis.

Return type
array_like, size (num_of_beams, 2)

`kvec`(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the k-vector of the laser beam

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
kvec – the k vector at position R and time t for each laser beam.

Return type
list of array_like, size(3,)

`pol`(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the polarization of the laser beam at position R and t

The polarization is returned in the spherical basis.

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
pol – polarization of each laser beam at R and t in spherical basis.

Return type
list of array_like, size (3,)

polarization_ellipse(*xp*, *yp*, *R=array([0.0, 0.0, 0.0])*, *t=0*)

The polarization ellipse parameters of the laser beam at R and t

Parameters
- **xp** (*array_like, shape (3,)*) – The x vector of the basis in which to calculate the polarization ellipse. Must be orthogonal to k.
- **yp** (*array_like, shape (3,)*) – The y vector of the basis in which to calculate the polarization ellipse. Must be orthogonal to k and *xp*.
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
list of (psi, chi) – list of (ψ, χ) parameters of the polarization ellipses for each laser beam

Return type
list of tuples

project_pol(*quant_axis*, *R=array([0.0, 0.0, 0.0])*, *t=0*, ***kwargs*)

Project the polarization onto a quantization axis.

Parameters
- **quant_axis** (*array_like, shape (3,)*) – A normalized 3-vector of the quantization axis direction.
- **R** (*array_like, shape (3,), optional*) – If polarization is a function of R is the 3-vectors at which the polarization shall be calculated.
- **calculate_norm** (*bool, optional*) – If true, renormalizes the quant_axis. By default, False.
- **treat_nans** (*bool, optional*) – If true, every place that nan is encoutnered, replace with the $hat{z}$ axis as the quantization axis. By default, False.
- **invert** (*bool, optional*) – If true, invert the process to project the quantization axis onto the specified polarization.

Returns
projected_pol – The polarization projected onto the quantization axis for all laser beams

Return type
list of array_like, shape (3,)

stokes_parameters(*xp*, *yp*, *R=array([0.0, 0.0, 0.0])*, *t=0*)

The Stokes Parameters of the laser beam at R and t

Parameters
- **xp** (*array_like, shape (3,)*) – The x vector of the basis in which to calculate the Stokes parameters. Must be orthogonal to k.
- **yp** (*array_like, shape (3,)*) – The y vector of the basis in which to calculate the Stokes parameters. Must be orthogonal to k and *xp*.
- **R** (*array_like, size (3,), optional*) – vector of the position at which to calculate the Stokes parameters. By default, the origin.
- **t** (*float, optional*) – time at which to calculate the Stokes parameters. By default, t=0.

Returns
pol – Stokes parameters for the laser beams, [Q, U, V]
Return type
array_like, shape (num_of_beams, 3)

`total_electric_field`(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the total electric field of the laser beams

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
E – the total electric field vector at position R and time t of all the laser beams
Return type
array_like, size(3,)

`total_electric_field_gradient`(*R=array([0.0, 0.0, 0.0])*, *t=0.0*)

Returns the total gradient of the electric field of the laser beams

Parameters
- **R** (*array_like, size (3,), optional*) – vector of the position at which to return the kvector. By default, the origin.
- **t** (*float, optional*) – time at which to return the k-vector. By default, t=0.

Returns
dE – the total electric field gradient matrices at position R and time t of all laser beams.
Return type
array_like, size(3,)

class pylcp.**conventional3DMOTBeams**(*k=1, pol=1, rotation_angles=[0.0, 0.0, 0.0], rotation_spec='ZYZ', beam_type=<class 'pylcp.fields.laserBeam'>, **kwargs*)

A collection of laser beams for 6-beam MOT

The standard geometry is to generate counter-progagating beams along all orthogonal axes $(\hat{x}, \hat{y}, \hat{z})$.

Parameters

- **k** (*float, optional*) – Magnitude of the k-vector for the six laser beams. Default: 1
- **pol** (*int or float, optional*) – Sign of the circular polarization for the beams moving along $\hat{z}$. Default: +1. Orthogonal beams have opposite polarization by default.
- **rotation_angles** (*array_like*) – List of angles to define a rotated MOT. Default: [0., 0., 0.]
- **rotation_spec** (*str*) – String to define the convention of the Euler rotations. Default: âĂŸZYZâĂŹ
- **beam_type** (pylcp.laserBeam *or subclass*) – Type of beam to generate.
- ****kwargs** – other keyword arguments to pass to beam_type

4.5 Governing Equations

The classes here correspond to the three governing equations.

4.5.1 Overview

governingeq.governingeq(laserBeams, magField)	Governing equation base class
heuristiceq(laserBeams, magField[, a, mass, ...])	Heuristic force equation
rateeq(laserBeams, magField, hamitlonian[, ...])	The rate equations
obe(laserBeams, magField, hamitlonian[, a, ...])	The optical Bloch equations

4.5.2 Details

class pylcp.governingeq.**governingeq**(*laserBeams*, *magField*, *hamiltonian=None*, *a=array([0.0, 0.0, 0.0])*, *r0=array([0.0, 0.0, 0.0])*, *v0=array([0.0, 0.0, 0.0])*)

Governing equation base class

This class is the basis for making all the governing equations in *pylcp*, including the rate equations, heuristic equation, and the optical Bloch equations. Its methods are available to other governing equations.

Parameters

- **laserBeams** (*dictionary of pylcp.laserBeams, pylcp.laserBeams, or list of pylcp.laserBeam*) – The laserBeams that will be used in constructing the optical Bloch equations. which transitions in the block diagonal hamiltonian. It can be any of the following:
  - A dictionary of pylcp.laserBeams: if this is the case, the keys of the dictionary should match available d^{nm} matrices in the pylcp.hamiltonian object. The key structure should be *n->m*.
  - pylcp.laserBeams: a single set of laser beams is assumed to address the transition *g->e*.
  - a list of pylcp.laserBeam: automatically promoted to a pylcp.laserBeams object assumed to address the transtion *g->e*.
- **magField** (*pylcp.magField or callable*) – The function or object that defines the magnetic field.
- **hamiltonian** (*pylcp.hamiltonian or None*) – The internal hamiltonian of the particle.
- **a** (*array_like, shape (3,), optional*) – A default acceleraiton to apply to the particleâĂŹs motion, usually gravity. Default: [0., 0., 0.]
- **r0** (*array_like, shape (3,)*) – Initial position. Default: [0.,0.,0.]
- **v0** (*array_like, shape (3,)*) – Initial velocity. Default: [0.,0.,0.]

damping_coeff(*axes*, *r=None*, *eps=0.01*, ***kwargs*)

Find the damping coefficent

Uses the find_equilibrium force() method to calculate the damping coefficient for the particular configuration.

Parameters

- **axes** (*array_like*) – A list of axis indices to compute the damping coefficient(s) along. Here, $\hat{x}$ is index 0, $\hat{y}$ is index 1, and $\hat{z}$ is index 2. For example, *axes=[2]* calculates the damping parameter along $\hat{z}$.
- **r** (*array_like, optional*) – The position at which to calculate the damping coefficient. By default r=None, which defaults to calculating at the equilibrium position as found by the find_equilibrium_position() method. If this method has not been run, it defaults to the origin.
- **eps** (*float*) – The small numerical ϵ parameter used for calculating the df/dv derivative. Default: 0.01
- **kwargs** – Any additional keyword arguments to pass to find_equilibrium_force()

Returns
beta – The damping coefficients along the selected axes.

Return type
list or float

find_equilibrium_force()

Find the equilibrium force at the initial conditions

Returns
force – Equilibrium force experienced by the atom

Return type
array_like

find_equilibrium_position(*axes*, ***kwargs*)

Find the equilibrium position

Uses the find_equilibrium force() method to calculate the where the $\mathbf{f}(\mathbf{r}, \mathbf{v} = 0) = 0$.

Parameters

- **axes** (*array_like*) – A list of axis indices to compute the trapping frequencies along. Here, $\hat{x}$ is index 0, $\hat{y}$ is index 1, and $\hat{z}$ is index 2. For example, *axes=[2]* calculates the trapping frquency along $\hat{z}$.
- **kwargs** – Any additional keyword arguments to pass to find_equilibrium_force()

Returns
r_eq – The equilibrium positions along the selected axes.

Return type
list or float

force()

Find the instantaneous force

Returns
force – Force experienced by the atom

Return type
array_like

generate_force_profile()

Map out the equilibrium force vs. position and velocity

Parameters

- **R** (*array_like, shape(3, ...)*) – Position vector. First dimension of the array must be length 3, and corresponds to x, y, and z components, repsectively.

- **V** (*array_like, shape(3, ...)*) – Velocity vector. First dimension of the array must be length 3, and corresponds to v_x, v_y, and v_z components, repsectively.
- **name** (*str, optional*) – Name for the profile. Stored in profile dictionary in this object. If None, uses the next integer, cast as a string, (i.e., âĂŸ0âĂŹ) as the name.
- **progress_bar** (*boolean, optional*) – Displays a progress bar as the proceeds. Default: False

Returns
profile – Resulting force profile.

Return type
pylcp.common.base_force_profile

set_initial_position(*r0*)

Sets the initial position

Parameters
r0 (*array_like, shape (3,)*) – Initial position. Default: [0.,0.,0.]

set_initial_position_and_velocity(*r0*, *v0*)

Sets the initial position and velocity

Parameters

- **r0** (*array_like, shape (3,)*) – Initial position. Default: [0.,0.,0.]
- **v0** (*array_like, shape (3,)*) – Initial velocity. Default: [0.,0.,0.]

set_initial_velocity(*v0*)

Sets the initial velocity

Parameters
v0 (*array_like, shape (3,)*) – Initial position. Default: [0.,0.,0.]

trapping_frequencies(*axes*, *r=None*, *eps=0.01*, ***kwargs*)

Find the trapping frequency

Uses the find_equilibrium force() method to calculate the trapping frequency for the particular configuration.

Parameters

- **axes** (*array_like*) – A list of axis indices to compute the trapping frequencies along. Here, $\hat{x}$ is index 0, $\hat{y}$ is index 1, and $\hat{z}$ is index 2. For example, *axes=[2]* calculates the trapping frquency along $\hat{z}$.
- **r** (*array_like, optional*) – The position at which to calculate the damping coefficient. By default r=None, which defaults to calculating at the equilibrium position as found by the find_equilibrium_position() method. If this method has not been run, it defaults to the origin.
- **eps** (*float, optional*) – The small numerical ϵ parameter used for calculating the df/dr derivative. Default: 0.01
- **kwargs** – Any additional keyword arguments to pass to find_equilibrium_force()

Returns
omega – The trapping frequencies along the selected axes.

Return type
list or float

class pylcp.**heuristiceq**(*laserBeams*, *magField*, *a=array([0.0, 0.0, 0.0])*, *mass=100*, *gamma=1*, *k=1*, *r0=array([0.0, 0.0, 0.0])*, *v0=array([0.0, 0.0, 0.0])*)

Heuristic force equation

The heuristic equation governs the atom or molecule as if it has a single transition between an $F = 0$ ground state to an $F' = 1$ excited state.

Parameters

- **laserBeams** (*dictionary of pylcp.laserBeams, pylcp.laserBeams, or list of pylcp.laserBeam*) – The laserBeams that will be used in constructing the optical Bloch equations. which transitions in the block diagonal hamiltonian. It can be any of the following:
  - A dictionary of pylcp.laserBeams: if this is the case, the keys of the dictionary should match available d^{nm} matrices in the pylcp.hamiltonian object. The key structure should be *n->m*. Here, it must be *g->e*.
  - pylcp.laserBeams: a single set of laser beams is assumed to address the transition *g->e*.
  - a list of pylcp.laserBeam: automatically promoted to a pylcp.laserBeams object assumed to address the transtion *g->e*.
- **magField** (*pylcp.magField or callable*) – The function or object that defines the magnetic field.
- **hamiltonian** (*pylcp.hamiltonian*) – The internal hamiltonian of the particle.
- **a** (*array_like, shape (3,), optional*) – A default acceleraiton to apply to the particleâĂŹs motion, usually gravity. Default: [0., 0., 0.]
- **r0** (*array_like, shape (3,), optional*) – Initial position of the atom or molecule. Default: [0., 0., 0.]
- **v0** (*array_like, shape (3,), optional*) – Initial velocity of the atom or molecule. Default: [0., 0., 0.]
- **mass** (*float, optional*) – Mass of the atom or molecule. Default: 100
- **gamma** (*float, optional*) – Decay rate of the single transition in the atom or molecule. Default: 1
- **k** (*float, optional*) – Magnitude of the k vector for the single transition in the atom or molecule. Default: 1

evolve_motion(*t_span, freeze_axis=[False, False, False], random_recoil=False, random_force=False, max_scatter_probability=0.1, progress_bar=False, rng=Generator(PCG64) at 0x7F69229EBA50, **kwargs*)

Evolve the motion of the atom in time.

Parameters

- **t_span** (*list or array_like*) – A two element list or array that specify the initial and final time of integration.
- **freeze_axis** (*list of boolean*) – Freeze atomic motion along the specified axis. Default: [False, False, False]
- **random_recoil** (*boolean*) – Allow the atom to randomly recoil from scattering events. Default: False
- **random_force** (*boolean*) – Rather than calculating the force using the heuristieq.force() method, use the calculated scattering rates from each of the laser beam to randomly add

photon absorption events that cause the atom to recoil randomly from the laser beam(s). Default: False

- **max_scatter_probability** (*float*) – When undergoing random recoils, this sets the maximum time step such that the maximum scattering probability is less than or equal to this number during the next time step. Default: 0.1
- **progress_bar** (*boolean*) – If true, show a progress bar as the calculation proceeds. Default: False
- **rng** (*numpy.random.Generator()*) – A properly-seeded random number generator. Default: calls `numpy.random.default.rng()`
- ****kwargs** – Additional keyword arguments get passed to solve_ivp_random, which is what actually does the integration.

Returns

sol –

Bunch object that contains the following fields:

- t: integration times found by solve_ivp
- v: atomic velocity
- r: atomic position

It contains other important elements, which can be discerned from scipyâĂŹs solve_ivp documentation.

Return type
OdeSolution

find_equilibrium_force(*return_details=False*)

Finds the equilibrium force at the initial position

Parameters
return_details (*boolean, optional*) – If True, returns the forces from each laser and the scattering rate matrix.

Returns

- **F** (*array_like*) – total equilibrium force experienced by the atom
- **F_laser** (*array_like*) – If return_details is True, the forces due to each laser.
- **R** (*array_like*) – The scattering rate matrix.

force(*r*, *v*, *t*)

Calculates the instantaneous force

Parameters

- **r** (*array_like*) – Position at which to calculate the force
- **v** (*array_like*) – Velocity at which to calculate the force
- **t** (*float*) – Time at which to calculate the force

Returns

- **F** (*array_like*) – total equilibrium force experienced by the atom

- **F_laser** (*dictionary of array_like*) – If return_details is True, the forces due to each laser, indexed by the manifold the laser addresses. The dictionary is keyed by the transition driven, and individual lasers are in the same order as in the pylcp.laserBeams object used to create the governing equation.

generate_force_profile(*R*, *V*, *name=None*, *progress_bar=False*)

Map out the equilibrium force vs. position and velocity

Parameters

- **R** (*array_like, shape(3, ...)*) – Position vector. First dimension of the array must be length 3, and corresponds to x, y, and z components, repsectively.
- **V** (*array_like, shape(3, ...)*) – Velocity vector. First dimension of the array must be length 3, and corresponds to v_x, v_y, and v_z components, repsectively.
- **name** (*str, optional*) – Name for the profile. Stored in profile dictionary in this object. If None, uses the next integer, cast as a string, (i.e., âĂŸ0âĂŹ) as the name.
- **progress_bar** (*boolean, optional*) – Displays a progress bar as the proceeds. Default: False

Returns

profile – Resulting force profile.

Return type

pylcp.common.base_force_profile

scattering_rate(*r*, *v*, *t*, *return_kvecs=False*)

Calculates the scattering rate

Parameters

- **r** (*array_like*) – Position at which to calculate the force
- **v** (*array_like*) – Velocity at which to calculate the force
- **t** (*float*) – Time at which to calculate the force
- **return_kvecs** (*bool*) – If true, returns both the scattering rate and the k-vecotrs from the lasers.

Returns

- **R** (*array_like*) – Array of scattering rates associated with the lasers driving the transition.
- **kvecs** (*array_like*) – If return_kvecs is True, the k-vectors of each of the lasers. This is used in heuristiceq.force, where it calls this function to calculate the scattering rate first. By returning the k-vectors with the scattering rates, it prevents the need of having to recompute the k-vectors again.

class pylcp.common.**base_force_profile**(*R*, *V*, *laserBeams*, *hamiltonian*)

Base force profile

The force profile object stores all of the calculated quantities created by the governingeq.generate_force_profile() method. It has the following attributes:

R

Positions at which the force profile was calculated.

Type

array_like, shape (3, âĂę)

V

Velocities at which the force profile was calculated.

Type
array_like, shape (3, âĂę)

F

Total equilibrium force at position R and velocity V.

Type
array_like, shape (3, âĂę)

f_mag

Magnetic force at position R and velocity V.

Type
array_like, shape (3, âĂę)

f

The forces due to each laser, indexed by the manifold the laser addresses. The dictionary is keyed by the transition driven, and individual lasers are in the same order as in the pylcp.laserBeams object used to create the governing equation.

Type
dictionary of array_like

Neq

Equilibrium population found.

Type
array_like

class pylcp.**rateeq**(*laserBeams*, *magField*, *hamitlonian*, *a=array([0.0, 0.0, 0.0])*, *include_mag_forces=True*, *svd_eps=1e-10*, *r0=array([0.0, 0.0, 0.0])*, *v0=array([0.0, 0.0, 0.0])*)

The rate equations

This class constructs the rate equations from the given laser beams, magnetic field, and hamiltonian.

Parameters

- **laserBeams** (*dictionary of pylcp.laserBeams*, pylcp.laserBeams, *or list of pylcp.laserBeam*) – The laserBeams that will be used in constructing the optical Bloch equations. which transitions in the block diagonal hamiltonian. It can be any of the following:
  - A dictionary of pylcp.laserBeams: if this is the case, the keys of the dictionary should match available d^{nm} matrices in the pylcp.hamiltonian object. The key structure should be *n->m*.
  - pylcp.laserBeams: a single set of laser beams is assumed to address the transition *g->e*.
  - a list of pylcp.laserBeam: automatically promoted to a pylcp.laserBeams object assumed to address the transtion *g->e*.
- **magField** (pylcp.magField *or callable*) – The function or object that defines the magnetic field.
- **hamiltonian** (pylcp.hamiltonian) – The internal hamiltonian of the particle.
- **a** (*array_like, shape (3,), optional*) – A default acceleraiton to apply to the particleâĂŹs motion, usually gravity. Default: [0., 0., 0.]

- **include_mag_forces** (*boolean*) – Optional flag to inculde magnetic forces in the force calculation. Default: True
- **r0** (*array_like, shape (3,), optional*) – Initial position. Default: [0., 0., 0.]
- **v0** (*array_like, shape (3,), optional*) – Initial velocity. Default: [0., 0., 0.]

construct_evolution_matrix(*r*, *v*, *t=0.0*, *default_axis=array([0.0, 0.0, 1.0])*)

Constructs the evolution matrix at a given position and time.

Parameters

- **r** (*array_like, shape (3,)*) – Position at which to calculate the equilibrium population
- **v** (*array_like, shape (3,)*) – Velocity at which to calculate the equilibrium population
- **t** (*float*) – Time at which to calculate the equilibrium population

equilibrium_populations(*r*, *v*, *t*, ***kwargs*)

Returns the equilibrium population as determined by the rate equations

This method uses singular matrix decomposition to find the equilibrium state of the rate equations at a given position, velocity, and time.

Parameters

- **r** (*array_like, shape (3,)*) – Position at which to calculate the equilibrium population
- **v** (*array_like, shape (3,)*) – Velocity at which to calculate the equilibrium population
- **t** (*float*) – Time at which to calculate the equilibrium population
- **return_details** (*boolean, optional*) – In addition to the equilibrium populations, return the full population evolution matrix and the scattering rates for each of the lasers

Returns

- **Neq** (*array_like*) – Equilibrium population vector
- **Rev** (*array_like*) – If return details is True, the evolution matrix for the state populations.
- **Rijl** (*dictionary of array_like*) – If return details is True, the scattering rates for each laser and each combination of states between the manifolds specified by the dictionaryâĂŹs index.

evolve_motion(*t_span, freeze_axis=[False, False, False], random_recoil=False, random_force=False, max_scatter_probability=0.1, progress_bar=False, record_force=False, rng=Generator(PCG64) at 0x7F69229EBE50, **kwargs*)

Evolve the populations N and the motion of the atom in time.

This function evolves the rate equations, moving the atom through space, given the instantaneous force, for some period of time.

Parameters

- **t_span** (*list or array_like*) – A two element list or array that specify the initial and final time of integration.
- **freeze_axis** (*list of boolean*) – Freeze atomic motion along the specified axis. Default: [False, False, False]

- **random_recoil** (*boolean*) – Allow the atom to randomly recoil from scattering events. Default: False
- **random_force** (*boolean*) – Rather than calculating the force using the rateeq.force() method, use the calculated scattering rates from each of the laser beam (combined with the instantaneous populations) to randomly add photon absorption events that cause the atom to recoil randomly from the laser beam(s). Default: False
- **max_scatter_probability** (*float*) – When undergoing random recoils and/or force, this sets the maximum time step such that the maximum scattering probability is less than or equal to this number during the next time step. Default: 0.1
- **progress_bar** (*boolean*) – If true, show a progress bar as the calculation proceeds. Default: False
- **record_force** (*boolean*) – If true, record the instantaneous force and store in the solution. Default: False
- **rng** (*numpy.random.Generator()*) – A properly-seeded random number generator. Default: calls `numpy.random.default.rng()`
- ****kwargs** – Additional keyword arguments get passed to solve_ivp_random, which is what actually does the integration.

Returns

sol –

Bunch object that contains the following fields:

- t: integration times found by solve_ivp
- N: population vs. time
- v: atomic velocity
- r: atomic position

It contains other important elements, which can be discerned from scipyâĂŹs solve_ivp documentation.

Return type
OdeSolution

evolve_populations(*t_span*, ***kwargs*)

Evolve the state population in time.

This function integrates the rate equations to determine how the populations evolve in time. Any initial velocity is kept constant. It is analogous to obe.evolve_density().

Parameters

- **t_span** (*list or array_like*) – A two element list or array that specify the initial and final time of integration.
- ****kwargs** – Additional keyword arguments get passed to solve_ivp, which is what actually does the integration.

Returns

sol –

Bunch object that contains the following fields:

- t: integration times found by solve_ivp

- rho: density matrix
- v: atomic velocity (constant)
- r: atomic position

It contains other important elements, which can be discerned from scipyâĂŹs solve_ivp documentation.

Return type
OdeSolution

`find_equilibrium_force`(*return_details=False*, ***kwargs*)

Finds the equilibrium force at the initial position

This method works by finding the equilibrium population through the rateeq.equilibrium_population() function, then calculating the resulting force.

Parameters

- **`return_details`** (*`boolean, optional`*) – If true, returns the forces from each laser and the scattering rate matrix. Default: False
- **`kwargs`** – Any additional keyword arguments to be passed to equilibrium_populations()

Returns

- **F** (*array_like*) – total equilibrium force experienced by the atom
- **F_laser** (*array_like*) – If return_details is True, the forces due to each laser.
- **Neq** (*array_like*) – If return_details is True, the equilibrium populations.
- **Rijl** (*dictionary of array_like*) – If return details is True, the scattering rates for each laser and each combination of states between the manifolds specified by the dictionaryâĂŹs index.
- **F_mag** (*array_like*) – If return_details is True, the forces due to the magnetic field.
- **ii** (*int*) – Number of iterations needed to converge.

`force`(*r*, *t*, *N*, *return_details=True*)

Calculates the instantaneous force

Parameters

- **`r`** (*`array_like`*) – Position at which to calculate the force
- **`t`** (*`float`*) – Time at which to calculate the force
- **`N`** (*`array_like`*) – Relative state populations
- **`return_details`** (*`boolean, optional`*) – If True, returns the forces from each laser and the magnetic forces.

Returns

- **F** (*array_like*) – total equilibrium force experienced by the atom
- **F_laser** (*dictionary of array_like*) – If return_details is True, the forces due to each laser, indexed by the manifold the laser addresses. The dictionary is keyed by the transition driven, and individual lasers are in the same order as in the pylcp.laserBeams object used to create the governing equation.
- **F_mag** (*array_like*) – If return_details is True, the forces due to the magnetic field.

generate_force_profile(*R*, *V*, *name=None*, *progress_bar=False*, ***kwargs*)

Map out the equilibrium force vs. position and velocity

Parameters

- **R** (*array_like, shape(3, ...)*) – Position vector. First dimension of the array must be length 3, and corresponds to x, y, and z components, repsectively.
- **V** (*array_like, shape(3, ...)*) – Velocity vector. First dimension of the array must be length 3, and corresponds to v_x, v_y, and v_z components, repsectively.
- **name** (*str, optional*) – Name for the profile. Stored in profile dictionary in this object. If None, uses the next integer, cast as a string, (i.e., âĂŸ0âĂŹ) as the name.
- **progress_bar** (*boolean, optional*) – Displays a progress bar as the proceeds. Default: False
- **kwargs** – Any additional keyword arguments to be passed to rateeq.find_equilibrium_force()

Returns
profile – Resulting force profile.

Return type
pylcp.rateeq.force_profile

set_initial_pop(*N0*)

Sets the initial populations

Parameters
N0 (*array_like*) – The initial state population vector N_0. It must have n elements, where n is the total number of states in the system.

set_initial_pop_from_equilibrium()

Sets the initial populations based on the equilibrium population at the initial position and velocity and time t=0.

class pylcp.rateeq.**force_profile**(*R*, *V*, *laserBeams*, *hamiltonian*)

Rate equation force profile

The force profile object stores all of the calculated quantities created by the rateeq.generate_force_profile() method. It has following attributes:

R

Positions at which the force profile was calculated.

Type
array_like, shape (3, âĂę)

V

Velocities at which the force profile was calculated.

Type
array_like, shape (3, âĂę)

F

Total equilibrium force at position R and velocity V.

Type
array_like, shape (3, âĂę)

f_mag

Magnetic force at position R and velocity V.

Type
array_like, shape (3, âĂę)

f

The forces due to each laser, indexed by the manifold the laser addresses. The dictionary is keyed by the transition driven, and individual lasers are in the same order as in the pylcp.laserBeams object used to create the governing equation.

Type
dictionary of array_like

Neq

Equilibrium population found.

Type
array_like

Rijl

The pumping rates of each laser, indexed by the manifold the laser addresses. The dictionary is keyed by the transition driven, and individual lasers are in the same order as in the pylcp.laserBeams object used to create the governing equation.

Type
dictionary of array_like

class pylcp.**obe**(*laserBeams*, *magField*, *hamitlonian*, *a=array([0.0, 0.0, 0.0])*, *transform_into_re_im=True*, *use_sparse_matrices=None*, *include_mag_forces=True*, *r0=array([0.0, 0.0, 0.0])*, *v0=array([0.0, 0.0, 0.0])*)

The optical Bloch equations

This class constructs the optical Bloch equations from the given laser beams, magnetic field, and hamiltonian.

Parameters

- **laserBeams** (*dictionary of pylcp.laserBeams*, pylcp.laserBeams, *or list of pylcp.laserBeam*) – The laserBeams that will be used in constructing the optical Bloch equations. which transitions in the block diagonal hamiltonian. It can be any of the following:
  - A dictionary of pylcp.laserBeams: if this is the case, the keys of the dictionary should match available d^{nm} matrices in the pylcp.hamiltonian object. The key structure should be *n->m*.
  - pylcp.laserBeams: a single set of laser beams is assumed to address the transition *g->e*.
  - a list of pylcp.laserBeam: automatically promoted to a pylcp.laserBeams object assumed to address the transtion *g->e*.
- **magField** (pylcp.magField *or callable*) – The function or object that defines the magnetic field.
- **hamiltonian** (pylcp.hamiltonian) – The internal hamiltonian of the particle.
- **a** (*array_like, shape (3,), optional*) – A default acceleraiton to apply to the particleâĂŹs motion, usually gravity. Default: [0., 0., 0.]
- **transform_into_re_im** (*boolean*) – Optional flag to transform the optical Bloch equations into real and imaginary components. This helps to decrease computaiton time as it uses the symmetry $\rho_{ji} = \rho_{ij}^*$ to cut the number of equations nearly in half. Default: True

- **use_sparse_matrices** (*boolean or None*) – Optional flag to use sparse matrices. If none, it will use sparse matrices only if the number of internal states > 10, which would result in the evolution matrix for the density operators being a 100x100 matrix. At that size, there may be some speed up with sparse matrices. Default: None
- **include_mag_forces** (*boolean*) – Optional flag to inculde magnetic forces in the force calculation. Default: True
- **r0** (*array_like, shape (3,), optional*) – Initial position. Default: [0., 0., 0.]
- **v0** (*array_like, shape (3,), optional*) – Initial velocity. Default: [0., 0., 0.]

evolve_density(*t_span*, *progress_bar=False*, ***kwargs*)

Evolve the density operators ρ_{ij} in time.

This function integrates the optical Bloch equations to determine how the populations evolve in time. Any initial velocity is kept constant while the atom potentially moves through the light field. This function is therefore useful in determining average forces. Any constant acceleration set when the OBEs were generated is ignored. It is analogous to rateeq.evolve_populations().

Parameters

- **t_span** (*list or array_like*) – A two element list or array that specify the initial and final time of integration.
- **progress_bar** (*boolean*) – Show a progress bar as the calculation proceeds. Default:False
- ****kwargs** – Additional keyword arguments get passed to solve_ivp, which is what actually does the integration.

Returns

sol –

Bunch object that contains the following fields:

- t: integration times found by solve_ivp
- rho: density matrix
- v: atomic velocity (constant)
- r: atomic position

It contains other important elements, which can be discerned from scipyâĂŹs solve_ivp documentation.

Return type
OdeSolution

evolve_motion(*t_span, freeze_axis=[False, False, False], random_recoil=False, max_scatter_probability=0.1, progress_bar=False, record_force=False, rng=Generator(PCG64) at 0x7F69216EAE50, **kwargs*)

Evolve ρ_{ij} and the motion of the atom in time.

This function evolves the optical Bloch equations, moving the atom along given the instantaneous force, for some period of time.

Parameters

- **t_span** (*list or array_like*) – A two element list or array that specify the initial and final time of integration.

- **freeze_axis** (*list of boolean*) – Freeze atomic motion along the specified axis. Default: [False, False, False]
- **random_recoil** (*boolean*) – Allow the atom to randomly recoil from scattering events. Default: False
- **max_scatter_probability** (*float*) – When undergoing random recoils, this sets the maximum time step such that the maximum scattering probability is less than or equal to this number during the next time step. Default: 0.1
- **progress_bar** (*boolean*) – If true, show a progress bar as the calculation proceeds. Default: False
- **record_force** (*boolean*) – If true, record the instantaneous force and store in the solution. Default: False
- **rng** (*numpy.random.Generator()*) – A properly-seeded random number generator. Default: calls `numpy.random.default.rng()`
- ****kwargs** – Additional keyword arguments get passed to solve_ivp_random, which is what actually does the integration.

Returns

sol –

Bunch object that contains the following fields:

- t: integration times found by solve_ivp
- rho: density matrix
- v: atomic velocity
- r: atomic position

It contains other important elements, which can be discerned from scipyâĂŹs solve_ivp documentation.

Return type
OdeSolution

find_equilibrium_force(*deltat=500*, *itermax=100*, *Npts=5001*, *rel=1e-05*, *abs=1e-09*, *debug=False*, *initial_rho='rateeq'*, *return_details=False*, ***kwargs*)

Finds the equilibrium force at the initial position

This method works by solving the OBEs in a chunk of time ΔT, calculating the force during that chunck, continuing the integration for another chunck, calculating the force during that subsequent chunck, and comparing the average of the forces of the two chunks to see if they have converged.

Parameters

- **deltat** (*float*) – Chunk time ΔT. Default: 500
- **itermax** (*int, optional*) – Maximum number of iterations. Default: 100
- **Npts** (*int, optional*) – Number of points to divide the chunk into. Default: 5001
- **rel** (*float, optional*) – Relative convergence parameter. Default: 1e-5
- **abs** (*float, optional*) – Absolute convergence parameter. Default: 1e-9
- **debug** (*boolean, optional*) – If true, pring out debug information as it goes.
- **initial_rho** (*'rateeq' or 'equally'*) – Determines how to set the initial rho at the start of the calculation.

- **return_details** (*boolean, optional*) – If true, returns the forces from each laser and the scattering rate matrix.

Returns

- **F** (*array_like*) – total equilibrium force experienced by the atom
- **F_laser** (*dictionary of array_like*) – If return_details is True, the forces due to each laser, indexed by the manifold the laser addresses. The dictionary is keyed by the transition driven, and individual lasers are in the same order as in the pylcp.laserBeams object used to create the governing equation.
- **F_laser** (*dictionary of array_like*) – If return_details is True, the forces due to each laser and its q component, indexed by the manifold the laser addresses. The dictionary is keyed by the transition driven, and individual lasers are in the same order as in the pylcp.laserBeams object used to create the governing equation.
- **F_mag** (*array_like*) – If return_details is True, the forces due to the magnetic field.
- **Neq** (*array_like*) – If return_details is True, the equilibrium populations.
- **ii** (*int*) – Number of iterations needed to converge.

force(*r*, *t*, *rho*, *return_details=False*)

Calculates the instantaneous force

Parameters

- **r** (*array_like*) – Position at which to calculate the force
- **t** (*float*) – Time at which to calculate the force
- **rho** (*array_like*) – Density matrix with which to calculate the force
- **return_details** (*boolean, optional*) – If true, returns the forces from each laser and the scattering rate matrix.

Returns

- **F** (*array_like*) – total equilibrium force experienced by the atom
- **F_laser** (*array_like*) – If return_details is True, the forces due to each laser.
- **F_laser_q** (*array_like*) – If return_details is True, the forces due to each laser and itâĂŹs q component of the polarization.
- **F_mag** (*array_like*) – If return_details is True, the forces due to the magnetic field.

full_OBE_ev(*r*, *t*)

Calculate the evolution for the density matrix

This function calculates the OBE evolution matrix by assembling pre-stored versions of the component matries. This should be significantly faster than full_OBE_ev_scratch, but it may suffer bugs in the evolution that full_OBE_ev_scratch will not. If Bq is None, it will compute Bq based on r, t

Parameters

- **r** (*array_like, shape (3,)*) – Position at which to calculate evolution matrix
- **t** (*float*) – Time at which to calculate evolution matrix

Returns
ev_mat – Evolution matrix for the densities

Return type
array_like

full_OBE_ev_scratch(*r*, *t*)

Calculate the evolution for the density matrix

This function calculates the OBE evolution matrix at position t and r from scratch, first computing the full Hamiltonian, then the OBE evolution matrix computed via commutators, then adding in the decay matrix evolution. If *Bq* is *None*, it will compute *Bq*.

Parameters

- **r** (*array_like, shape (3,)*) – Position at which to calculate evolution matrix
- **t** (*float*) – Time at which to calculate evolution matrix

Returns
ev_mat – Evolution matrix for the densities

Return type
array_like

generate_force_profile(*R*, *V*, *name=None*, *progress_bar=False*, ***kwargs*)

Map out the equilibrium force vs. position and velocity

Parameters

- **R** (*array_like, shape(3, ...)*) – Position vector. First dimension of the array must be length 3, and corresponds to x, y, and z components, repsectively.
- **V** (*array_like, shape(3, ...)*) – Velocity vector. First dimension of the array must be length 3, and corresponds to v_x, v_y, and v_z components, repsectively.
- **name** (*str, optional*) – Name for the profile. Stored in profile dictionary in this object. If None, uses the next integer, cast as a string, (i.e., âĂŸ0âĂŹ) as the name.
- **progress_bar** (*boolean, optional*) – Displays a progress bar as the proceeds. Default: False

Returns
profile – Resulting force profile.

Return type
pylcp.obe.force_profile

observable(*O*, *rho=None*)

Observable returns the obervable O given density matrix rho.

Parameters

- **O** (*array or array-like*) – The matrix form of the observable operator. Can have any shape, representing scalar, vector, or tensor operators, but the last two axes must correspond to the matrix of the operator and have the same dimensions of the generating Hamiltonian. For example, a vector operator might have the shape (3, n, n), where n is the number of states and the first axis corresponds to x, y, and z.
- **rho** (*[optional] array or array-like*) – The density matrix. The first two dimensions must have sizes (n, n), but there may be multiple instances of the density matrix tiled in the higher dimensions. For example, a rho with (n, n, m) could have m instances of the density matrix at different times.

  If not specified, will get rho from the current solution stored in memory.

Returns
observable – observable has shape (O[:-2])+(rho[2:])

Return type
float or array

`set_initial_rho`(*rho0*)

Sets the initial ρ matrix

Parameters
rho0 (*array_like*) – The initial ρ. It must have n^2 elements, where n is the total number of states in the system. If a flat array, it will be reshaped.

`set_initial_rho_equally`()

Sets the initial ρ matrix such that all states have the same population.

`set_initial_rho_from_populations`(*Npop*)

Sets the diagonal elements of the initial ρ matrix

Parameters
Npop (*array_like*) – Array of the initial populations of the states in the system. The length must be n, where n is the number of states.

`set_initial_rho_from_rateeq`()

Sets the diagonal elements of the initial ρ matrix using the equilibrium populations as determined by pylcp.rateeq

`class pylcp.obe.force_profile`(*R*, *V*, *laserBeams*, *hamiltonian*)

Optical Bloch equation force profile

The force profile object stores all of the calculated quantities created by the rateeq.generate_force_profile() method. It has the following attributes:

R

Positions at which the force profile was calculated.

Type
array_like, shape (3, âĂę)

V

Velocities at which the force profile was calculated.

Type
array_like, shape (3, âĂę)

F

Total equilibrium force at position R and velocity V.

Type
array_like, shape (3, âĂę)

f_mag

Magnetic force at position R and velocity V.

Type
array_like, shape (3, âĂę)

f

The forces due to each laser, indexed by the manifold the laser addresses. The dictionary is keyed by the transition driven, and individual lasers are in the same order as in the pylcp.laserBeams object used to create the governing equation.

Type
dictionary of array_like

f_q

The force due to each laser and its q component, indexed by the manifold the laser addresses. The dictionary is keyed by the transition driven, and individual lasers are in the same order as in the pylcp.laserBeams object used to create the governing equation.

Type
dictionary of array_like

Neq

Equilibrium population found.

Type
array_like

4.6 Atom Class

The atom class contains useful reference numbers for a given atomic species, like mass, nuclear g-factor, with states and transitions useful for laser cooling. Data comes from Daniel SteckâĂŹs âĂIJAlkali D-line dataâĂİ, Tobias TieckeâĂŹs âĂIJProperties of PotassiumâĂİ and Michael GehmâĂŹs âĂIJProperties of 6LiâĂİ.

4.6.1 Overview

atom(species)	A class containing reference data for select laser-coolable alkali atoms

4.6.2 Detailed functions

class pylcp.**atom**(*species*)

A class containing reference data for select laser-coolable alkali atoms

Parameters
species (*string*) – The isotope number and species of alkali atom. For lithium-7, species can be eiter âĂIJ7LiâĂİ or âĂIJLi7âĂİ, for example. Supported species are âĂIJ6LiâĂİ, âĂIJ7LiâĂİ, âĂIJ23NaâĂİ, âĂIJ39KâĂİ, âĂIJ40KâĂİ, âĂIJ41KâĂİ, âĂIJ85RbâĂİ, âĂIJ87RbâĂİ, and âĂIJ133CsâĂİ.

I

Nuclear spin of the isoptope

Type
float

gI

Nuclear g-factor of the isotope. Note that the nuclear g-factor is specified relative to the Bohr magneton, not the nuclear magneton.

Type
float

mass

Mass, in kg, of the atom.

Type
float

states

States of the atom useful for laser cooling, in order of increasing energy.

Type
list of pylcp.atom.state

transitions

Transitions in the atom useful for laser cooling. All transitions are from the ground state.

Type
list of pylcp.atom.transition

class pylcp.atom.**state**(*n=None*, *S=None*, *L=None*, *J=None*, *lam=None*, *E=None*, *tau=inf*, *gJ=1*, *Ahfs=0*, *Bhfs=0*, *Chfs=0*)

The quantum state and its parameters for an atom.

Parameters

- **n** (*integer*) – Principal quantum number of the state.
- **S** (*integer or float*) – Total spin angular momentum of the state.
- **L** (*integer or float*) – Total orbital angular momentum of the state.
- **J** (*integer or float*) – Total electronic angular momentum of the state.
- **lam** (*float, optional*) – Wavelength, in meters, of the photon necessary to excite the state from the ground state. electronic angular momentum of the state.
- **E** (*float, optional*) – Energy of the state above the ground state in cm^{-1}.
- **tau** (*float, optioanl*) – Lifetime of the state in s. If not specified, it is assumed to be infinite (the ground state).
- **gJ** (*float*) – Total angular momentum Lande g-factor.
- **Ahfs** (*float*) – A hyperfine coefficient.
- **Bhfs** (*float*) – B hyperfine coefficient.
- **Chfs** (*float*) – C hyperfine coefficient.

gamma

Lifetime in s^{-1}

Type
float

gammaHz

Corresponding linewidth in Hz, given by $\gamma/2\pi$.

Type
float

energy

The energy in cm^{-1}

Type
float

Notes

All the parameters passed to the class on creation are stored as attrributes, with the exception of *lam* and *E*, one of which defines the stored attribute *energy*. One of these two optional variable must be specified.

This construction of the state assumes L-S coupling.

class pylcp.atom.**transition**(*state1*, *state2*, *mass*)

Reference numbers for transitions.

Parameters

- **state1** (pylcp.atom.state) – The lower state of the transition.
- **state2** (pylcp.atom.state) – The upper state of the transition.
- **mass** (*float*) – Mass of the atom in kg

k

Wavevector in cm^{-1}.

Type
float

lam

Wavelength in m.

Type
float

nu

Frequency in Hz of the transition.

Type
float

omega

Angular frequency in rad/s of the transition.

Type
float

Isat

Saturation intensity of the transition in $\mathrm{mW/cm}^2$.

Type
float

a0

Maximum acceleration $a_0 = \hbar k/2\Gamma$ in $\mathrm{cm/s}^2$.

Type
float

v0

Doppler velocity $v_0 = k/\Gamma$ in cm/s.

Type
float

x0

Length scale $x_0 = v_0^2/a_0$ in cm.

Type
float

t0

Time scale $t_0 = v_0/a_0$ in s.

Type
float

CHAPTER

FIVE

SUPPORT

Join our google group: https://groups.google.com/g/pylcp/

CHAPTER

SIX

CREDITS

Authors

Stephen Eckel, Daniel Barker, Eric Norrgard

Contributors

Abrar Sheikh, Shangjie Guo, Chad Ropp, Leo Wang

Cite as

Stephen Eckel, Daniel S. Barker, Eric B. Norrgard, and Julia Scherschligt, âĂIJPyLCP: A python package for computing laser cooling physicsâĂİ, Computer Physics Communications 270, 108166 (2020). https://doi.org/10.1016/j.cpc.2021.108166

Version

1.0.0 of 2020/11/10

Disclaimer

The full description of the procedures used in this documentation requires the identification of certain commercial products. The inclusion of such information should in no way be construed as indicating that such products are endorsed by NIST or are recommended by NIST or that they are necessarily the best software for the purposes described.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pylcp.hamiltonians, 147
pylcp.hamiltonians.XFmolecules, 145

INDEX

A

a0 (*pylcp.atom.transition attribute*), 186
add_d_q_block() (*pylcp.hamiltonian method*), 151
add_H_0_block() (*pylcp.hamiltonian method*), 151
add_laser() (*pylcp.laserBeams method*), 163
add_mu_q_block() (*pylcp.hamiltonian method*), 152
Astate() (*in module pylcp.hamiltonians.XFmolecules*), 145
atom (*class in pylcp*), 184

B

base_force_profile (*class in pylcp.common*), 172

C

cartesian_pol() (*pylcp.laserBeam method*), 158
cartesian_pol() (*pylcp.laserBeams method*), 163
clippedGaussianBeam (*class in pylcp*), 162
constantMagneticField (*class in pylcp*), 154
construct_evolution_matrix() (*pylcp.rateeq method*), 174
conventional3DMOTBeams (*class in pylcp*), 166

D

damping_coeff() (*pylcp.governingeq.governingeq method*), 167
delta() (*pylcp.laserBeam method*), 158
delta() (*pylcp.laserBeams method*), 163
diag_static_field() (*pylcp.hamiltonian method*), 152
dipoleXandAstates() (*in module pylcp.hamiltonians.XFmolecules*), 146
dqij_two_bare_hyperfine() (*in module pylcp.hamiltonians*), 147
dqij_two_fine_stucture_manifolds_uncoupled() (*in module pylcp.hamiltonians*), 147
dqij_two_hyperfine_manifolds() (*in module pylcp.hamiltonians*), 148

E

electric_field() (*pylcp.laserBeam method*), 158
electric_field() (*pylcp.laserBeams method*), 163
electric_field_gradient() (*pylcp.infinitePlaneWaveBeam method*), 161
electric_field_gradient() (*pylcp.laserBeam method*), 158
electric_field_gradient() (*pylcp.laserBeams method*), 164
energy (*pylcp.atom.state attribute*), 185
eps (*pylcp.laserBeam attribute*), 158
eps (*pylcp.magField attribute*), 153
equilibrium_populations() (*pylcp.rateeq method*), 174
evolve_density() (*pylcp.obe method*), 179
evolve_motion() (*pylcp.heuristiceq method*), 170
evolve_motion() (*pylcp.obe method*), 179
evolve_motion() (*pylcp.rateeq method*), 174
evolve_populations() (*pylcp.rateeq method*), 175

F

F (*pylcp.common.base_force_profile attribute*), 173
f (*pylcp.common.base_force_profile attribute*), 173
F (*pylcp.obe.force_profile attribute*), 183
f (*pylcp.obe.force_profile attribute*), 183
F (*pylcp.rateeq.force_profile attribute*), 177
f (*pylcp.rateeq.force_profile attribute*), 178
f_mag (*pylcp.common.base_force_profile attribute*), 173
f_mag (*pylcp.obe.force_profile attribute*), 183
f_mag (*pylcp.rateeq.force_profile attribute*), 177
f_q (*pylcp.obe.force_profile attribute*), 184
FieldMag() (*pylcp.magField method*), 154
find_equilibrium_force() (*pylcp.governingeq.governingeq method*), 168
find_equilibrium_force() (*pylcp.heuristiceq method*), 171
find_equilibrium_force() (*pylcp.obe method*), 180
find_equilibrium_force() (*pylcp.rateeq method*), 176
find_equilibrium_position() (*pylcp.governingeq.governingeq method*), 168

fine_structure_uncoupled() (*in module pylcp.hamiltonians*), 148
force() (*pylcp.governingeq.governingeq method*), 168
force() (*pylcp.heuristiceq method*), 171
force() (*pylcp.obe method*), 181
force() (*pylcp.rateeq method*), 176
force_profile (*class in pylcp.obe*), 183
force_profile (*class in pylcp.rateeq*), 177
full_OBE_ev() (*pylcp.obe method*), 181
full_OBE_ev_scratch() (*pylcp.obe method*), 181

G

gamma (*pylcp.atom.state attribute*), 185
gammaHz (*pylcp.atom.state attribute*), 185
gaussianBeam (*class in pylcp*), 161
generate_force_profile() (*pylcp.governingeq.governingeq method*), 168
generate_force_profile() (*pylcp.heuristiceq method*), 172
generate_force_profile() (*pylcp.obe method*), 182
generate_force_profile() (*pylcp.rateeq method*), 176
gI (*pylcp.atom attribute*), 184
governingeq (*class in pylcp.governingeq*), 167
gradField() (*pylcp.constantMagneticField method*), 155
gradField() (*pylcp.iPMagneticField method*), 156
gradField() (*pylcp.magField method*), 154
gradField() (*pylcp.quadrupoleMagneticField method*), 155
gradFieldMag() (*pylcp.constantMagneticField method*), 155
gradFieldMag() (*pylcp.iPMagneticField method*), 156
gradFieldMag() (*pylcp.magField method*), 154

H

hamiltonian (*class in pylcp*), 150
heuristiceq (*class in pylcp*), 169
hyperfine_coupled() (*in module pylcp.hamiltonians*), 149
hyperfine_uncoupled() (*in module pylcp.hamiltonians*), 149

I

I (*pylcp.atom attribute*), 184
infinitePlaneWaveBeam (*class in pylcp*), 160
intensity() (*pylcp.clippedGaussianBeam method*), 162
intensity() (*pylcp.gaussianBeam method*), 162
intensity() (*pylcp.laserBeam method*), 159
intensity() (*pylcp.laserBeams method*), 164
iPMagneticField (*class in pylcp*), 155
Isat (*pylcp.atom.transition attribute*), 186

J

jones_vector() (*pylcp.laserBeam method*), 159
jones_vector() (*pylcp.laserBeams method*), 164

K

k (*pylcp.atom.transition attribute*), 186
kvec() (*pylcp.laserBeam method*), 159
kvec() (*pylcp.laserBeams method*), 164

L

lam (*pylcp.atom.transition attribute*), 186
laser_keys (*pylcp.hamiltonian attribute*), 151
laserBeam (*class in pylcp*), 157
laserBeams (*class in pylcp*), 163

M

magField (*class in pylcp*), 153
make_full_matrices() (*pylcp.hamiltonian method*), 152
mass (*pylcp.atom attribute*), 184
module
    pylcp.hamiltonians, 147
    pylcp.hamiltonians.XFmolecules, 145

N

Neq (*pylcp.common.base_force_profile attribute*), 173
Neq (*pylcp.obe.force_profile attribute*), 184
Neq (*pylcp.rateeq.force_profile attribute*), 178
ns (*pylcp.hamiltonian attribute*), 151
nu (*pylcp.atom.transition attribute*), 186

O

obe (*class in pylcp*), 178
observable() (*pylcp.obe method*), 182
omega (*pylcp.atom.transition attribute*), 186

P

phase (*pylcp.laserBeam attribute*), 158
pol() (*pylcp.laserBeam method*), 159
pol() (*pylcp.laserBeams method*), 164
polarization_ellipse() (*pylcp.laserBeam method*), 159
polarization_ellipse() (*pylcp.laserBeams method*), 165
print_structure() (*pylcp.hamiltonian method*), 152
project_pol() (*pylcp.laserBeam method*), 160
project_pol() (*pylcp.laserBeams method*), 165
pylcp.hamiltonians
    module, 147
pylcp.hamiltonians.XFmolecules
    module, 145

Q

quadrupoleMagneticField (*class in pylcp*), 155

R

R (*pylcp.common.base_force_profile attribute*), 172
R (*pylcp.obe.force_profile attribute*), 183
R (*pylcp.rateeq.force_profile attribute*), 177
rateeq (*class in pylcp*), 173
return_full_H() (*pylcp.hamiltonian method*), 152
Rijl (*pylcp.rateeq.force_profile attribute*), 178

S

scattering_rate() (*pylcp.heuristiceq method*), 172
set_initial_pop() (*pylcp.rateeq method*), 177
set_initial_pop_from_equilibrium() (*pylcp.rateeq method*), 177
set_initial_position() (*pylcp.governingeq.governingeq method*), 169
set_initial_position_and_velocity() (*pylcp.governingeq.governingeq method*), 169
set_initial_rho() (*pylcp.obe method*), 183
set_initial_rho_equally() (*pylcp.obe method*), 183
set_initial_rho_from_populations() (*pylcp.obe method*), 183
set_initial_rho_from_rateeq() (*pylcp.obe method*), 183
set_initial_velocity() (*pylcp.governingeq.governingeq method*), 169
set_mass() (*pylcp.hamiltonian method*), 153
singleF() (*in module pylcp.hamiltonians*), 150
state (*class in pylcp.atom*), 185
state_labels (*pylcp.hamiltonian attribute*), 151
states (*pylcp.atom attribute*), 185
stokes_parameters() (*pylcp.laserBeam method*), 160
stokes_parameters() (*pylcp.laserBeams method*), 165

T

t0 (*pylcp.atom.transition attribute*), 187
total_electric_field() (*pylcp.laserBeams method*), 166
total_electric_field_gradient() (*pylcp.laserBeams method*), 166
transition (*class in pylcp.atom*), 186
transitions (*pylcp.atom attribute*), 185
trapping_frequencies() (*pylcp.governingeq.governingeq method*), 169

V

V (*pylcp.common.base_force_profile attribute*), 172
V (*pylcp.obe.force_profile attribute*), 183
V (*pylcp.rateeq.force_profile attribute*), 177
v0 (*pylcp.atom.transition attribute*), 186

X

x0 (*pylcp.atom.transition attribute*), 186
Xstate() (*in module pylcp.hamiltonians.XFmolecules*), 146