rydiqule

Release 1.0.0rc1

Quantum Technology Center
DEVCOM Army Research Laboratory
Naval Air Warfare Center - Weapons Division

Unclassified - Approved for Public Release

Dr. Kevin C Cox - kevin.c.cox29.civ@army.mil

Dr. Christopher O'brien - christopher.m.obrien22.civ@us.navy.mil

Dr. David H Meyer - david.h.meyer3.civ@army.mil

Mr. Benjamin Miller - benjamin.n.miller@navy.mil

May 07, 2023

GETTING STARTED

A python library for calculating Rydberg electrometer response to arbitrary RF fields in steady-state or time domains. It is a general density matrix-based master equation solver, optimized for speed to solve problems with large parameter spaces while maintaining flexibility to define novel problems. It leverages a graph-based system definition, computationally-efficient equation "stacking" in the form of tensors, and external computational libraries such as `numpy`, `scipy`, and `ARC`.

For more details, see the *Rydiqule Overview*.

For detailed usage examples, see the *Introduction to Rydiqule* Jupyter notebook.

CHAPTER

ONE

INSTALLATION

Presently, installation must be done manually using a copy of the repository.

1.1 Pure pip installation

To install in an editable way (which allows edits of the source code), run:

```
pip install -e .
```

from within the top level `Rydiqule` directory (i.e. where the `setup.cfg` file resides). This command will use pip to install all necessary dependencies.

To install normally, run:

```
pip install .
```

from the same directory.

1.2 Conda/pip installation

If using a conda environment, it can often be advantageous to install as many packages via conda before running the pip installation command.

Assuming you have not already created a separate environment for RydIQule (recommended), run the following to create a new environment:

```
(base) ~/Rydiqule> conda create -n rydiqule python=3.9
(base) ~/Rydiqule> conda activate rydiqule
```

RydIQule currently requires python 3.8 or 3.9.

Now install dependencies that are available via conda.

```
(rydiqule) ~/Rydiqule> conda install numpy scipy matplotlib tqdm networkx psutil
# ARC specific dependencies available via conda
(rydiqule) ~/Rydiqule> conda install sympy
```

Now use pip to install rydiqule and remaining dependencies.

```
# for editable installation, so source can be modified locally
(rydiqule) ~/Rydiqule> pip install -e .
# for normal installation
(rydiqule) ~/Rydiqule> pip install .
```

1.3 Confirm installation

Proper installation can be confirmed by executing the following commands in a python terminal.

```
>>> import rydiqule as rq
>>> rq.about()

        Rydiqule
    ================

Rydiqule Version:     0.3.0
Installation Path:    c:\users\naqsl\src\rydiqule\src\rydiqule

      Dependencies
    ================

NumPy Version:        1.20.3
SciPy Version:        1.7.1
Matplotlib Version:   3.5.0
ARC Version:          3.2.0
Python Version:       3.8.12
Python Install Path:  C:\Users\naqsL\Miniconda3\envs\arc
Platform Info:        Windows (AMD64)
CPU Count:            12
Total System Memory:  128 GB
```

1.4 Updating an existing installation

Upgrading an existing installation is simple. Simply run the pip installation commands described above with the update flag.

```
pip install -U .
```

This command will also install any new dependencies that are required.

If using an editable install, simply replacing the files in the same directory is sufficient. Though it is recommended to also run the appropriate pip update command as well.

```
pip install -U -e .
```

1.5 Dependencies

This package requires installation of the excellent ARC package, which is used to get Rydberg atomic properties. It also requires other standard computation dependencies, such as `numpy`, `scipy`, `matplotlib`, etc. These will be automatically installed by pip if not already present.

If installing dependencies from `conda`, ensure they are installed before installing `rydiqule` to avoid `pip`/ `conda` conflicts.

CHAPTER

TWO

RYDIQULE OVERVIEW

Rydiqule (RYDberg sensing Interactive Quantum ModULE) is a python module that can calculate response of a Rydberg sensor to RF fields. It uses a semi-classical approximation of the Schroedinger equation known as the Lindblad equation to create equations of motion that describe the interaction of the sensor with optical and RF fields.

In order to use rydiqule at its basic level, you need to understand a few core elements. These elements are shown in Fig. **??**.

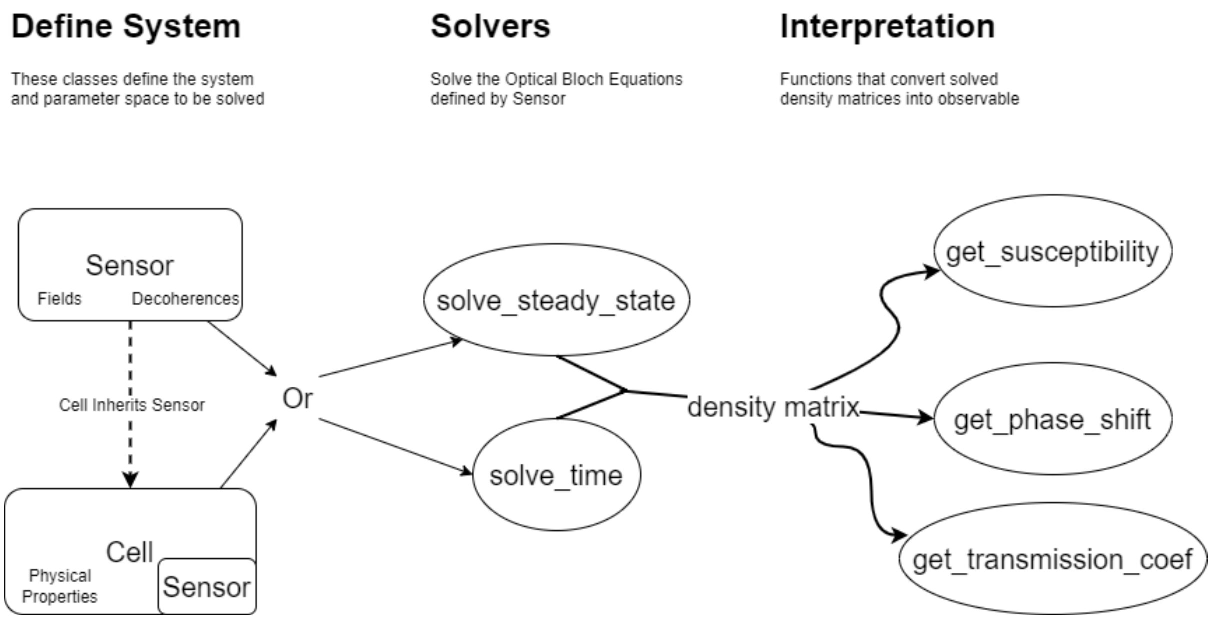


Fig. 2.1: The basic organizational structure for using Rydiqule.

A calculation needs three general components.

1. Define the system to be solved.
2. Solve the system.
3. Interpret the results to observable quantities.

2.1 Define the System

A system can be defined using one of two classes: *`Sensor`* or *`Cell`*. In either case, multiple values for a parameter can be set to produce parameter scans in the solve stage. The Sensor class defines the bare minimum of information necessary to produce a system of equations to solve. This class allows for arbitrary definitions of the system. The Cell class defines a physical gas of atoms for the system that in turn produces a Sensor for doing calculations. This class relies on ARC to provide physical parameters.

2.2 Solve the System

We currently have two solvers implemented.

1. A steady state solver (*`solve_steady_state()`*) that finds the steady state density matrix of the system. This can solve in a few different conditions:
    1. Optically-thin, cold ensemble
    2. Optically-thin, doppler-averaged ensemble
2. A time solver (*`solve_time()`*) that allows for fields to be defined arbitrarily in time.

Each solver takes a Sensor or Cell object and solves the system. The output is the corresponding density matrix in the steady-state, spanning the defined parameter space, or a series of density matrices versus time in the case of the time solver.

2.3 Interpret the Results

Once the solultions are made, we need to interpret the density matrices into observable values, typically some change to the optical probing field. We implement a few functions to get the *`susceptibility`*, *`probe absorption`*, and *`probe phase shift`*. Note that getting these values generally requires more information about the system than the bare minimum required to solve them. *`Raw density matrix elements`* can also be obtained.

rydiqule

CHAPTER

THREE

INTRODUCTION TO RYDIQULE

RydIQule, the Rydberg Interactive Quantum Module, is a python library built to simulate the interaction of Rydberg atoms and light using a semi-classical approach. This notebook will illustrate some of its core functionality, and demonstrate how to use the tool to model simple systems. The intent here is not to demonstrate discoveries in physics. Rather it will use cartoonish but nontrivial examples to demonstrate how to use the module.

3.1 Design philosophy

Rydiqule was built with a few core principles in mind:

1. **Rydiqule is simple** - Setting and solving an atomic system can be done with just a handful of lines of code while behaving in an intuitive way.
2. **Rydiqule is fast** - Under the hood, the library makes broad use of fast numpy matrix braodcasting and compiled code in places that would be slowed down by native python. The result is a toolbox that can produce meaningful results in a few minutes or less.
3. **Rydiqule is flexible** - Rydiqule can model a huge variety of semiclassical Rydberg atomic systems with no code modification. For users with more particular modelling needs who wish to extend Rydiqule, the `Sensor` class provides a minimal physical system that can easily be inherited and overloaded for more involved experimental setups.

3.2 Limitations

While we have worked hard to make Rydiqule as good as possible, there are some areas that can cause issues:

1. **Memory** - For systems with many laser parameter values, many levels, doppler averaging in several dimensions, or especially a combination of these, the equations of motion generated by rydiqule simply have to be very large, often requiring more memory than is in a typical laptop or simple desktop. For very large systems, the memory footprint may even outpace a powerful workstation. Rydiqule has built-in functionality to handle some of these cases, but it is far from perfect and will need to be iteratated upon to be as flexible as possible.
2. **Speed** - While huge improvements have been made in the speed of `rydiqule`, there are certain situations that can still cause some slowdowns. For longer simulations, in particular for the poorly-conditioned equations produced with large doppler width, solving can still be slow.
3. **Quantum Back-action** - We treat the optical fields as static, and do not include them explicitly in the semi-classical equations of motion. Rydiqule does not account for atom-field back-action effects. This approximation is valid for low optical depth samples, and is known to give valid results for SNR in moderate optical depth samples. However, for quantitative analysis of quantum noise in high optical depth samples, Rydiqule may not be accurate.
4. **Device Modelling** - Rydiqule is a physics solver, and does not currently have user-friendly support for device-level modelling.

3.2.1 Imports

Rydiqule is conventionally imported as `rq`. In addition, `numpy` and `matplotlib` are useful to have in notebooks, so we will import them as well. They are dependencies of `rydiqule`, so they should already be installed if you installed rydiqule.

```
import rydiqule as rq
import numpy as np
import matplotlib.pyplot as plt
```

```
#Auto-reload code for fast testing
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

3.3 1. Creating a `Sensor` object

The `Sensor` is the core object of rydiqule. It defines an atomic system, and while you can create an solve a Hamiltonian manually, the `Sensor` takes care of the bookeeping to generate Hamiltonian and solve its associated equations of motion. The base `Sensor` class is constructed with a single required argument, the basis size. Here we create a 3-level system. On its own, a `Sensor` contains no information about atomic structure; it is an abstraction that allows for a high degree of manual configuration.

```
basis_size = 3
sensor = rq.Sensor(basis_size)
```

3.3.1 Defining a simple Ladder system

This demonstrates how a minimal ladder system would be defined in `rydiqule`. States are coupled by defining dictionaries which have one key-value pair describing the basis states which are coupled as a tuple with two integer elements. For applied laser fields under the rotating wave approximation, you must also specify a rabi frequency and detuning. They are then added using the `add_couplings()` function with as many couplings as you'd like. All frequency values are in megarad/s.

Notes on detunings and rotating wave transformation

For defining detunings, the states in a coupling are always defined to go from lower to higher energy. For example, `(1,2)` means state 2 is higher than 1. Any coupling with a defined `detuning` will be treated in the rotating frame of the coupling.

```
laser_01 = {"states": (0,1), "detuning": 1, "rabi_frequency": 3}
laser_12 = {"states": (1,2), "detuning": 2, "rabi_frequency": 5}
sensor.add_couplings(laser_01, laser_12)
```

Once the system is defined, we can see the Hamiltonian matrix. You usually will not need to call this explicitly (it will be called internally by a solver), but it can be useful to make sure the system is defined as expected.

```
print(sensor.get_hamiltonian())
```

```
[[0. +0.j 1.5+0.j 0. +0.j]
 [1.5-0.j 1. +0.j 2.5+0.j]
 [0. +0.j 2.5-0.j 3. +0.j]]
```

3.3.2 Defining a simple V scheme

We can do something similar, but with a different arrangement of couplings. We can also pass fields in the constructor if we like

```
laser_01 = {"states": (0,1), "detuning": 1, "rabi_frequency": 3}
laser_02 = {"states": (0,2), "detuning": 2, "rabi_frequency": 5}
sensor_v = rq.Sensor(3, laser_01, laser_02)

print(sensor_v.get_hamiltonian())
```

```
[[0. +0.j 1.5+0.j 2.5+0.j]
 [1.5-0.j 1. +0.j 0. +0.j]
 [2.5-0.j 0. +0.j 2. +0.j]]
```

3.3.3 Defining a simple Lambda scheme

Here is another system, this time a lambda scheme.

```
laser_02 = {"states": (0,2), "detuning": 1, "rabi_frequency": 3}
laser_21 = {"states": (2,1), "detuning": 2, "rabi_frequency": 5}
sensor_lambda = rq.Sensor(3, laser_02, laser_21)

print(sensor_lambda.get_hamiltonian())
```

```
[[0. +0.j 0. +0.j 1.5+0.j]
 [0. +0.j 3. +0.j 2.5-0.j]
 [1.5-0.j 2.5+0.j 1. +0.j]]
```

Before we proceed further, we will make a quick note of what is happening under the hood. `rydiqule` is storing all of the information we specified when we added couplings on an object called a graph from the `networkx` library (https://networkx.org/). We treat the nodes of this graph as states and the edges as couplings. Internally, the graph is called couplings. It is not important to understand `networkx` to use rydiqule, but let's have a look at this `Sensor.couplings` attribute to hopefully help make it a little more transparent.

```
print(sensor_lambda.couplings.nodes)
print(sensor_lambda.couplings.edges(data=True))
```

```
[0, 1, 2]
[(0, 2, {'rabi_frequency': 3, 'detuning': 1, 'phase': 0, 'kvec': (0, 0, 0), 'no_
↪rwa_warning': False, 'label': '(0,2)'}), (2, 1, {'rabi_frequency': 5, 'detuning
↪': 2, 'phase': 0, 'kvec': (0, 0, 0), 'no_rwa_warning': False, 'label': '(2,1)'})]
```

The edges contain all of the data that we have added! Again, this is not crucial, but if you are interested in how data is stored, this is a useful demo.

3.3.4 Systems that are not fully coupled

We can also define a system in which not all states are connected explicitly by couplings. This allows us to solve systems which, for example, have states coupled only by decoherence. This exact use case will be demonstrated later, but we can set up the system and show the hamiltonian here.

The hamiltonaian works by treating an uncoupled states as a "second ground state", and calculates digonal hamiltonian elements from there. We show a 4-level system in which state 4 is coupled to state 5 via a steady state rf transition, but to no other states. It should be noted that calling an rf transition does not change the way the system is solved here.

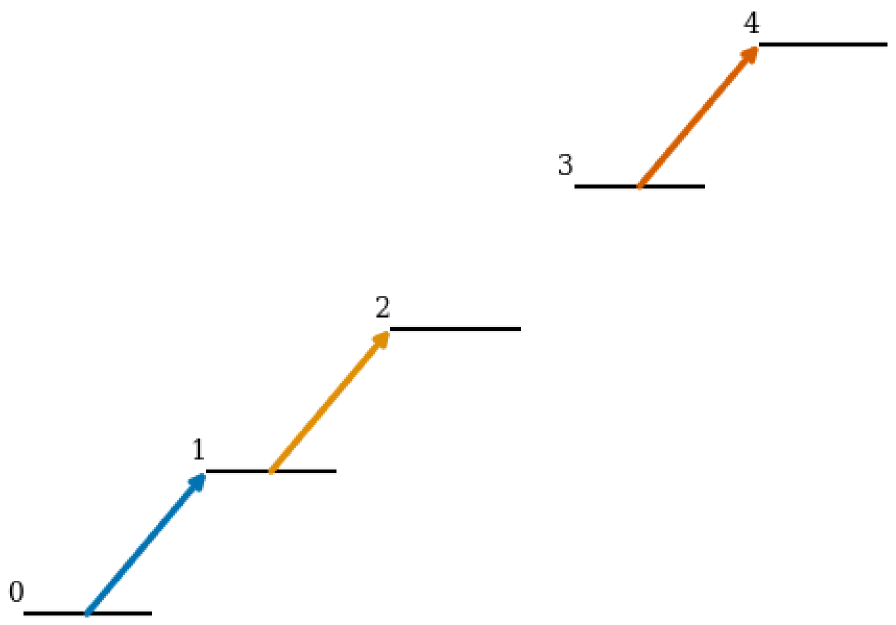
Looking at the hamiltonian, we can see that the 4th term along the diagonal is 0, and the 5th term counts from there.

```
sensor_uncoupled = rq.Sensor(5)
laser_01 = {"states": (0,1), "detuning": 1, "rabi_frequency": 3}
laser_12 = {"states": (1,2), "detuning": 2, "rabi_frequency": 5}
rf = {"states": (3,4), "detuning": 8, "rabi_frequency": 1}
sensor_uncoupled.add_couplings(laser_01, laser_12, rf)

print(sensor_uncoupled.get_hamiltonian())
```

```
[[0.  +0.j 1.5+0.j 0.  +0.j 0.  +0.j 0.  +0.j]
 [1.5-0.j 1.  +0.j 2.5+0.j 0.  +0.j 0.  +0.j]
 [0.  +0.j 2.5-0.j 3.  +0.j 0.  +0.j 0.  +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 0.5-0.j 8.  +0.j]]
```

In some more complicated cases, we may want a more concrete visual reassurance that everything is defined correctly. `rydiqule` makes use of the `atomic_energy_diagram` library to help draw visual representations of a `Sensor`. In the drawing below, we can see that the diagram indeed shows that states 2 and 3 are not coupled.

```
rq.draw_diagram(sensor_uncoupled);
```



3.4 2. Solving a `Sensor` in the steady-state case

So far we have just created sensor objects and shown their corresponding Hamiltonians. The most important functionality of `rydiqule`, however, is solving their associated equations of motion automatically. Here we will demonstrate some of the ways to solve a simple system using `rydiqule`.

3.4.1 Basic Solving in the steady state

If all we need steady state behavior of a particular system, it is straightforward and quick in rydiqule. By wrapping a simple matrix differential equation solver, we can quickly get the steady state frequencies of each element of the density matrix ρ of the system. We start by re-defining the same ladder system we had at the beginning of the notebook.

```
basis_size = 3
sensor = rq.Sensor(basis_size)
laser_01 = {"states": (0,1), "detuning": 1, "rabi_frequency": 3}
laser_12 = {"states": (1,2), "detuning": 2, "rabi_frequency": 5}
sensor.add_couplings(laser_01, laser_12)
```

Gamma matrix

However, before we solve the system, we must define the decoherence and dephasing rates of the system, which is defined by the `gamma_matrix`. This is square matrix whose dimensionality matches that of the system Hamiltonian. Diagonal terms of this matrix represent represent dephasing rates for each state, while off-diagonal terms represent the rate of spontaneous population decay between states. Just like with the rest of rydiqule, the units are in Mrad/s.

This step is crucial. Without a gamma matrix (or with a matrix of all zeros), the equations of motion will be singular, and the system will not have a well-defined steady-state solution.

The gamma matrix is a matrix where each element Γ_{ij} defines the incoherent decay rate of atomic population from state i to state j. Diagonal elements Γ_{ii} represent incoherend dephasing from state i. More details may be found in Appendix A of https://arxiv.org/abs/2105.10494

```
gamma = np.zeros((basis_size, basis_size))
gamma[1,0] = 0.1
sensor.set_gamma_matrix(gamma)
```

Instead of creating the entire gamma matrix and adding all at once, you can also add individual decoherences using the `add_decoherence` method of `Sensor` as well as the related, specialized helper methods `add_self_broadening` and `add_transit_broadening`.

Here is the equivalent method of defining the gamma matrix for the system using these methods.

```
sensor.add_decoherence((1,0), 0.1)
```

Now, instead of showing the Hamiltonian, we can just call `rydiqule.solve_steady_state()` function on `sensor` and get the result.

```
solution = rq.solve_steady_state(sensor)
print(type(solution))
```

```
<class 'rydiqule.sensor_solution.Solution'>
```

As you can see, the solution itself is not an array, but rather a `rydiqule.Solution` object. This is a bunch-type object that functions just like a python dictionary from which you access values as class variables instead of the ususal `d[key]` syntax. This makes accessing elements a little cleaner. Here we can see that accessing with the usual dictionary key syntax or as a class variable. As you can see below, they are the same.

```
print(solution.rho)
print(solution["rho"])
```

```
[ 0.17354416 -0.24474176  0.00800973  0.24029191 -0.20024326 -0.00667478
  0.          0.22694236]
[ 0.17354416 -0.24474176  0.00800973  0.24029191 -0.20024326 -0.00667478
  0.          0.22694236]
```

Adjustments to the equations

So why does the solution have 8 elements? Why is it real? It represents a density matrix, which should have n^2 complex-valued elements ($3^2 = 9$ in this case).

For numerical stability, `rydiqule` removes the ground state population equations, so the ρ_{00} term is omitted from the solution (typically $\rho_{00} >> \rho_{ij}$ for $ij \neq 00$, which is the source of the numerical problems). If needed, it can be inferred as $\rho_{00} = 1 - \sum_{i=1}^{n} \rho_{ii}$. Furthermore, rydiqule transforms the basis so that all values are real. Rather than an $n \times n$ Hermitian matrix, `rydiqule` parameterizes the density matrix ρ as $n \times n$ real values before removing the ground state. For more details, refer to the documentation of the `get_basis_transformation()` function in the `sensor_utils` module.

So how do we know (ideally quickly) which element of the solution corresponds to which density matrix element? The `Sensor` class contains a function called `basis()` which does exactly this, returning a list of strings showing the info we want. Since the density matrix is hermitian and trace 1, this array does indeed contain all of the information of the density matrix. Note that `basis()` does not include ρ_{00} since the ground state is removed by the solver (see docs)

```
print(sensor.basis())
```

```
['01_real' '02_real' '01_imag' '11_real' '12_real' '02_imag' '12_imag'
 '22_real']
```

Note: there are technically arguments to disable this and solve the full complex equations, but they are intended for internal use only, and almost certainly will not work for a general system. We recommend keeping these values at their defaults.

3.4.2 Solving for multiple values of a parameter

Getting a single result is nice, but really just a single data point. One of the simplest types of experiments you might want to simulate is to scan, for example, over a range of detuning values and see the behavior of the system at each value. Fortunately, `rydiqule` makes this type of experiment trivial. Suppose we wanted to sweep our probe laser detuning between -10 MHz and 10 MHz. Without `rydiqule` we might write an explicit loop over a set of values, modifying and solving the system at each iteration, and storing the results in a new list. In python especially, this sort of approach has a high computational overhead. With `rydiqule`, We can set up the system similarly to above, this time as a 4-level system. However, this time, instead of using a single `float` value for a coupling parameter, we will define the $0 \leftrightarrow 1$ coupling detunuing value as a list-like object, in our case a 1-dimensional numpy array constructed with the `linspace` function.

```
basis_size = 4
sensor_sweep = rq.Sensor(basis_size)

detunings = 2*np.pi*np.linspace(-10,10,201) #201 values between -10 and 10 MHz
probe = {"states":(0,1), "detuning": detunings, "rabi_frequency": 3}
coupling = {"states":(1,2), "detuning": 0, "rabi_frequency": 5}
rf = {"states":(2,3), "detuning": 0, "rabi_frequency":7}

```

(continues on next page)

(continued from previous page)

```
sensor_sweep.add_couplings(probe, coupling, rf)

sensor_sweep.add_decoherence((1,0), 0.1)
sensor_sweep.add_decoherence((2,1), 0.1)
sensor_sweep.add_decoherence((3,0), 0.1)
```

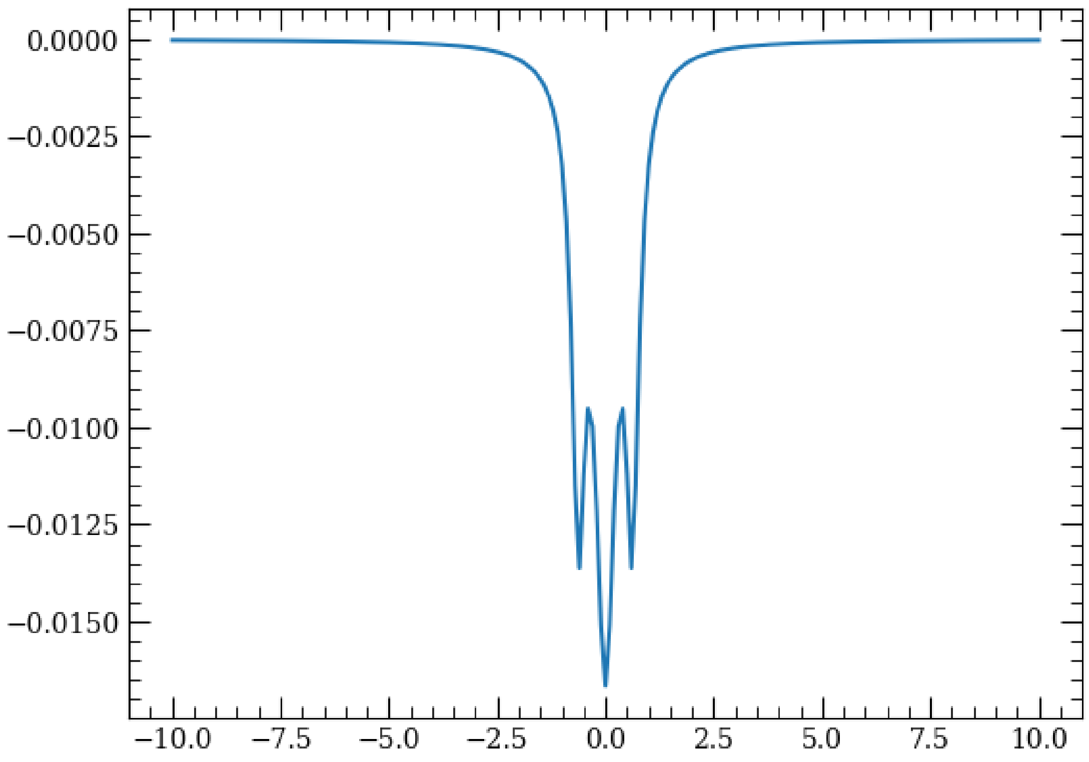
From here, we solve exactly as before. Rydiqule will automagically solve the system for every detuning value (very quickly). We also show the utility function `get_rho_ij()` which extracts ρ_{ij} from the density matrix ρ of any solution (regardless of how many dimensions it is). We use it to get ρ_{01}.

```
solution = rq.solve_steady_state(sensor_sweep)
print(f"Solution shape: {solution.rho.shape}")
absorption = rq.get_rho_ij(solution.rho,0,1).imag
print(f"Absorption_shape: {absorption.shape}")
```

```
Solution shape: (201, 15)
Absorption_shape: (201,)
```

We can see that there is a 15-element solution for each one of the detuning values. After we get the absorption using `get_rho_ij()`, we can see that we are left with a single array with 201 elements, corresponding to to the absorption at each value of detuning. The function doesn't do anything special, it just is a quick way to get common info you might want out of a solution. We can now do a quick-and-dirty plot to see what it looks like:

```
plt.plot(detunings/2/np.pi, absorption);
```



3.4.3 Solving for multiple values of multiple parameters

If you like the simplicity of scanning an single laser detuning in `rydiqule` you will be even more excited to learn that rydiqule can handle scans over multple different parameters simulataneously. We will set up the sensor as before, with a couple of changes. We will now also scan the dephasing rate and show how the absorption changes.

```
basis_size = 4
sensor_sweep_2 = rq.Sensor(basis_size)

detunings = 2*np.pi*np.linspace(-10, 10, 201) #201 values between -10 and 10 MHz
probe = {"states":(0,1), "detuning": detunings, "rabi_frequency": 3, "label":"probe
↪"}
coupling = {"states":(1,2), "detuning": 0, "rabi_frequency": 5, "label": "coupling
↪"}
rf = {"states":(2,3), "detuning": 0, "rabi_frequency":7, "label": "rf"}

sensor_sweep_2.add_couplings(probe, coupling, rf)

gamma10 = np.linspace(0.1, 1.0, 10)
sensor_sweep_2.add_decoherence((1,0), gamma10)
sensor_sweep_2.add_decoherence((2,1), 0.1)
sensor_sweep_2.add_decoherence((3,0), 0.1)
```

Once again, no extra steps are required, just call `solve_steady_state()` as normal, and look at the shape of the solution and the absorption and rydiqule will quickly find the solution for every combination of those 2 parameters.

```
solution_2 = rq.solve_steady_state(sensor_sweep_2)
print(f"Solution shape: {solution_2.rho.shape}")
absorption_2 = rq.get_rho_ij(solution_2.rho,0,1).imag
print(f"Absorption_shape: {absorption_2.shape}")
```

```
Solution shape: (201, 10, 15)
Absorption_shape: (201, 10)
```

How do we know which axis corresponds to the probe detuning and which is the coupling detuning? `Sensor` has a method called `axis_labels()` which does just that, and this is where labeling our couplings comes in handy. Note that if couplings are not labeled, the axes will default to being labeled by the states they couple. For example `["(0,1)_detuning", (1,2)_detuning"]`.

This function will not label the axes for the density matrix, or the time or doppler axes (discussed later), since it is just a method of `Sensor`. The density matrix will always be last, time second to last (if solved in the time domain), and the axes for n dimensions of doppler solveing will be the first n axes.

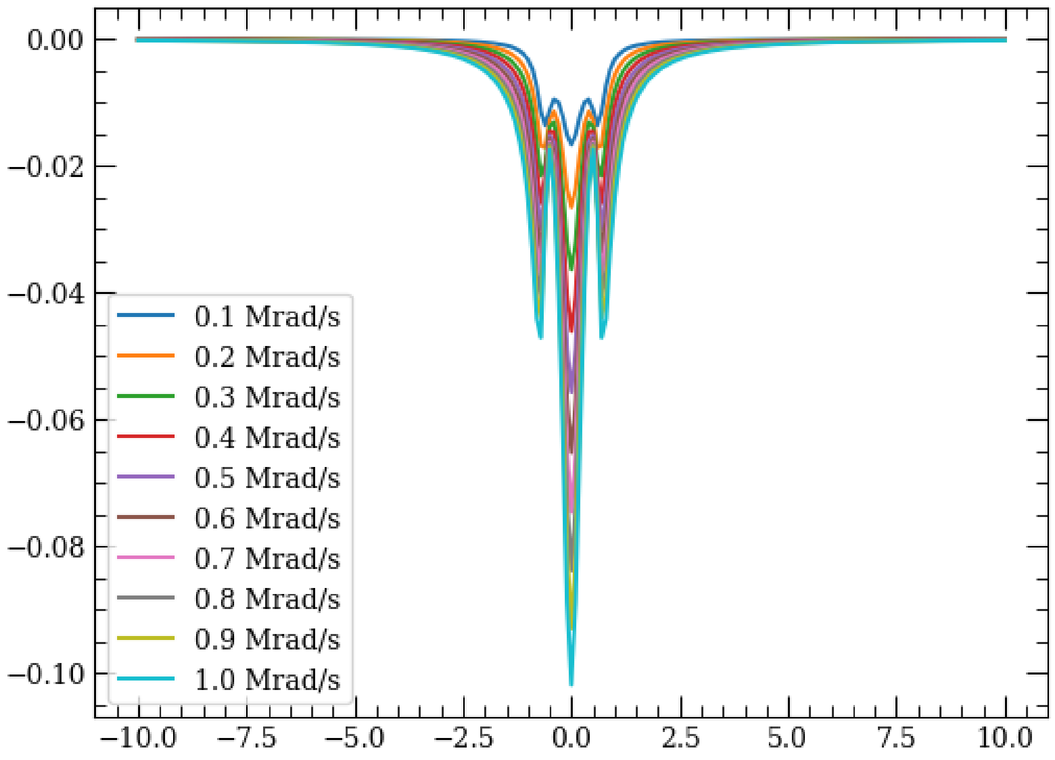
```
print(sensor_sweep_2.axis_labels())
```

```
['probe_detuning', '(1,0)_gamma']
```

So the first axis corresponds to the probe laser detuning, and the second axis corresponds to $\Gamma_{1,0}$. Now calling a function like `np.argmin()` or `np.argmax()` could quickly be used to optimize some value over a large parameter space. `rydiqule` lets us add as many parameters as you like as a list, detunings, rabi frequencies, or dephasings. It's handy, but beware that the memory footprint can quickly balloon out of control when generating equations of motion if you get too ambitious, increasing by a factor of n when you add an axis with n elements, especially given that many values used in internal calculations are 128-bit complex arrays. Currently, `rydiqule` has internal functions to spot this before it happens and break it into more manageable chunks if it can, but it is certainly possible to make a system which even it cannot handle. If the solution does not fit in memory, no amount of splitting up the equations will allow the system to be solved, and you should reconsider the size of your parameter space. Even in the above example, `rydiqule` is solving $201 \times 10 \times 15 \approx 30000$ equations simultaneously.

We can plot these probe sweeps at the same time fairly simply.

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

for i in range(absorption_2.shape[1]):
    ax.plot(detunings/2/np.pi, absorption_2[:,i], label=f'{gamma10[i]:.1f} Mrad/s')

ax.legend();
```



3.5 3. Solving in the time domain

Suppose you have defined a sensor, and we want to see the response of the sensor of some well-defined rf input function. In this case, the steady-state solution above is not enough. We will set up a sensor similar to the one above, but define the rf field differently. We will use the `"time dependence"` keyword in the dictionary, and supply with a function which takes a single argument of time in microseconds. First we will define a function below, which turns on a static field at $t = 1\mu s$.

```
rf_freq = 10

def turn_on_field(t):
    if t < 1:
        return 0.0
    else:
        return 1.0
```

`rydiqule` treats time dependent functions as a modulation of the rabi frequency at time *t*. So if we define the field we apply this function to as having a rabi frequency of 7 Mrad/s as above, the field will turn on with a rabi frequency of 7 Mrad/s at $t = 1$. Note that because we are still specifying a detuning, the field is still treated as being in the rotating frame.

Note: Specifying time dependence means you can no longer call `solve_steady_state()`

```
basis_size = 4
sensor_time = rq.Sensor(basis_size)

probe = {"states":(0,1), "detuning": 2, "rabi_frequency": 3, "label":"probe"}
coupling = {"states":(1,2), "detuning": 2, "rabi_frequency": 5, "label": "coupling
↪"}
rf = {"states":(2,3), "detuning": 0, "rabi_frequency":7, "label": "rf", "time_
↪dependence":turn_on_field}

sensor_time.add_couplings(probe, coupling, rf)

gamma = np.zeros((basis_size, basis_size))
gamma[1:,0] = 0.1
sensor_time.set_gamma_matrix(gamma)
```

With everything set up, we now call the `rq.solve_time()` function, which behaves as you might expect. The only difference is that you need a couple of extra arguments: - `end_time`, specifying the length of the simulation time in microsenconds. - `num_pts`, which specifys the number of points on the time interval $(0, t_{end})$ at which to sample the solution evenly.

In addition, the function can take the optional argument `use_nkode` which is a `bool` that indicates whether to use the `numbakit_ode` compiled differential equation solver as a backend. This can result in speedups in some situations, particularly very long time simulation ($\approx > 100\mu s$ or so). However, it introduces overhead with compiling input functions that means it is not guaranteed to provide a speedup so it defaults to `False`. Because its speed vs the default `scipy.optimize.solve_ivp()` backend is highly dependent on the particular problem, and even version/platform to a certain extent, our recommendation is to experiment with both `True` and `False` to see what works best for the types of problems you need to solve.

```
end_time = 10 #microseconds
num_pts = 100

solution_time = rq.solve_time(sensor_time, end_time, num_pts)
print(type(solution_time))
```

```
<class 'rydiqule.sensor_solution.Solution'>
```

Once again we have a `Solution` object. However, for a time solve, this object contains two fields, `rho` as before, and now also `t`, which are the time values at which the solution was evaluated. Let's have a look at the shape of each one.

```
print(f"Solution shape: {solution_time.rho.shape}")
print(f"t shape: {solution_time.t.shape}")
```

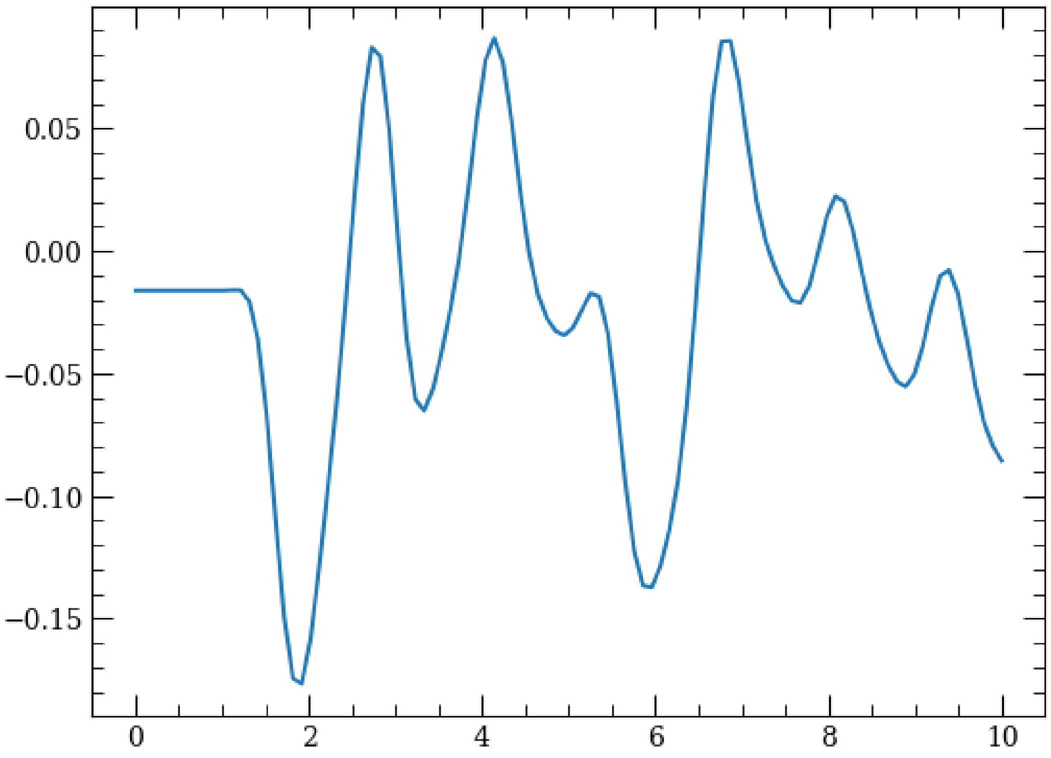
```
Solution shape: (100, 15)
t shape: (100,)
```

So the density matrix solution is an array of shape `(100, 15)`. The 15 is obviously the flattened real density matrix discussed above ($4^2 - 1 = 15$) and, as you might expect, the 100-element axis represents the time. So accessing `solution_time.rho[50,:]` or just `solution_time.rho[50]` would give the density matrix of the system at time given by the $50th$ element of `solution_time.t[50]`.

Now, we can look at what a particular density matrix element looks like as a function of time using matplotlib. This will be a minimal plot of ρ_{01} vs t just to show how you might do this for your own system. The `get_rho_ij` function makes it easy, since the axis for the density matrix is always the last one for any `rydiqule` solution.

```
absorption_time = rq.get_rho_ij(solution_time.rho,0,1).imag
print(f"Shape: {absorption_time.shape}")

plt.plot(solution_time.t, absorption_time);
```

```
Shape: (100,)
```



As we can see, the density matrix element is in a steady-state until the field is turned on at time $t = 1\mu s$, then we observe oscilitory behavior after the field is turned on. This is a toy model, so more detailed discussion of results will be saved for other notebooks, but you can see that viewing this behavior is very straightforward once the system has been solved.

3.5.1 Initial Conditions

You might reasonably ask where the steady-state that is on at $t = 0$ comes from in the above example. If the initial condition of the time solve is unspecified, `rydiqule` will solve the steady-state problem *without* any of the time-dependent fields active and use that as the initial condition for the solver. However, if you wish to start the system with a different initial density matrix, you can specifiy the optional `init_cond` argument in `solve_time()`. As a note, the shape of the initial condition must match what the shape of the steady-state solution is (if you wish to use different initial conditions for each set of parameters), or just the shape of the density matrix (to use the same initial condition for all cases). For this system, they are the same since there is only one value for each laser parameter, but in general they need to match.

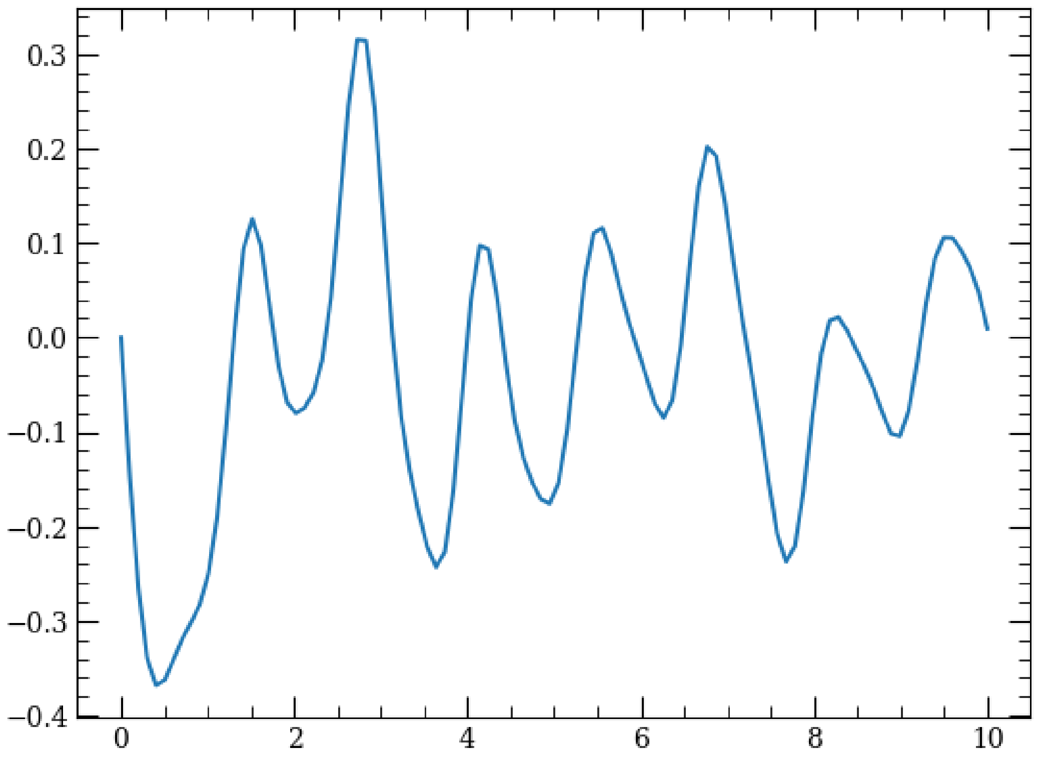
Let us see what it looks like to set the initial condition to all the population staying in the ground state. This corresponds to $\rho_{00} = 1$, and $\rho_{ij} = 0$ for $i, j \neq 0, 0$. Since we know `rydiqule` discards the ground state when it solves, this corresponds to an array of all zeros.

```
sol_dim = sensor_time.basis_size**2 - 1
ic = np.zeros(sol_dim)
print(f"Initial Condition Shape: {ic.shape}")
```

```
Initial Condition Shape: (15,)
```

Notice that we also accessed the size of the basis via `Sensor.basis_size`. This is the expected size for a solution, so lets see what the solution looks like for the absorption element ρ_{01}.

```
solution_time_from_0 = rq.solve_time(sensor_time, end_time, num_pts, init_cond=ic)
absorption_time_from_0 = rq.get_rho_ij(solution_time_from_0.rho, 0, 1).imag

plt.plot(solution_time_from_0.t, absorption_time_from_0);
```



As you might expect, we see an abrupt transient from zero, which has a small jump at $1\mu s$, followed by some oscillitory behavior that is at least similar to the plot above.

3.5.2 Multi-value Parameters in a time solve

Just like `solve steady`, `solve_time` supports coupling parameter definitions that are list-like. It works more or less as you might expect based on the behavior of steady-state solving, but it is worth briefly discussing for the sake of demonstrating what the solution might look like.

We start by creating a sensor identical to the one above, but with detunings defined as lists:

```
basis_size = 4
time_sensor_sweep = rq.Sensor(basis_size)

detunings = 2*np.pi*np.linspace(-10,10,51) #51 values between -10 and 10 MHz
probe = {"states":(0,1), "detuning": detunings, "rabi_frequency": 3, "label":"probe
↪"}
coupling = {"states":(1,2), "detuning": detunings, "rabi_frequency": 5, "label":
↪"coupling"}
rf = {"states":(2,3), "detuning": 0, "rabi_frequency":7, "label": "rf", "time_
↪dependence":turn_on_field}

time_sensor_sweep.add_couplings(probe, coupling, rf)

gamma = np.zeros((basis_size, basis_size))
gamma[1:,0] = 0.1
time_sensor_sweep.set_gamma_matrix(gamma)
```

Then, we solve with `solve_time` just as above. We have reduced the number of detunings, but remember that `rydiqule` is still solving about 2,500 equations here, so we should expect it to be a little slow. We can use the `%%time` jupyter magic to see how slow.

```
%%time
end_time = 10 #microseconds
num_pts = 100

solution_time_sweep = rq.solve_time(time_sensor_sweep, end_time, num_pts)
```

```
CPU times: total: 1.8 s
Wall time: 22.7 s
```

On my machine it runs in about 7 seconds, but the time depends on several things, including how powerful your machine is and library version/build issues we are working on. If it takes you several times longer, consider trying with `use_nkode=True`. We can now inspect and try and make sense of the shape of the solution.

```
print(f"Shape: {solution_time_sweep.rho.shape}")
```

```
Shape: (51, 51, 100, 15)
```

So what does this mean? As you likely can figure out, the last axis of the solution is still the elements of the density matrix with the expected 15 elements. Then, the second-to-last axis is the time axis. This holds true in general for time solutions regardless of how many other axes there are. The last is always density matrix elements and the second-to-last is always time. We can then get the other axes from a call to `Sensor.axis_labels()`, which works just as above to help with the rest of the axes.

```
print(time_sensor_sweep.axis_labels())
```

```
['probe_detuning', 'coupling_detuning']
```

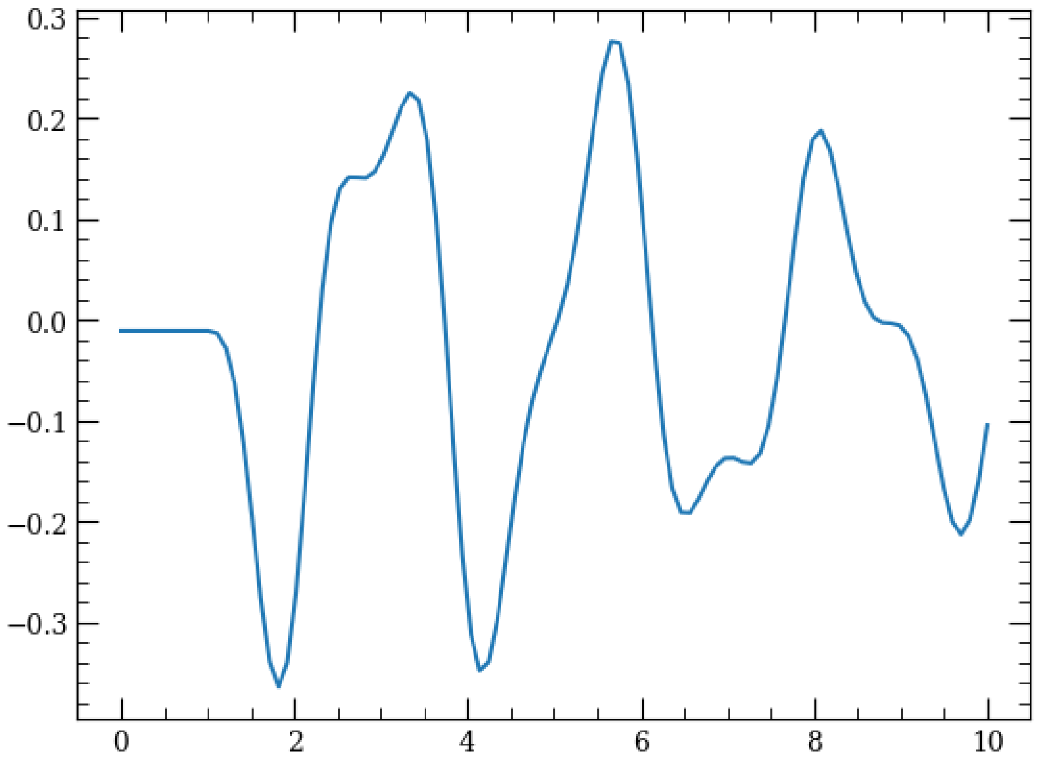
`get_rho_ij()` also works as expected for stacked time solutions:

```
absorption = rq.get_rho_ij(solution_time_sweep.rho, 0, 1).imag
print(absorption.shape)
```

```
(51, 51, 100)
```

So suppose we wanted to plot ρ_{01} vs t for the on-resonance case. On-resonance (detuning zero) corresponds to the middle of the detuning list defined above, element 25. We can then plot that vs time.

```
absorption_resonance = absorption[25, 25]
plt.plot(solution_time_sweep.t, absorption_resonance);
```



3.6 4. Simulating Doppler broadening

One final key piece of functionality in `rydiqule` is the ability to simulate a doppler-broadened system. Internally, this is handled by breaking a doppler velocity profile into some number of slices, then applying the corresponding detunings for each class to the unshifted solution. This produces many sets of equations of motion, which are then solved, and then a weighted average performed to get the doppler-broadened solution.

To set up a system with doppler broadening, we use the `"kvec"` keyword the couplings definition. A `kvec` is a 3-element spatial vector in the propagation direction of the field, with magnitude equal to the standard deviation of the doppler velocity profile component in that direction.

```
basis_size = 4
sensor_doppler = rq.Sensor(basis_size)

width = 2*np.pi*0.1 #Mrad/s
k_direction = np.array([1,0,0])

detunings = 2*np.pi*np.linspace(-10,10,201) #201 values between -10 and 10 MHz
probe = {"states":(0,1), "detuning": detunings, "rabi_frequency": 3, 'kvec': k_
↪direction*width}
coupling = {"states":(1,2), "detuning": 0, "rabi_frequency": 5, 'kvec': k_
↪direction*width}
rf = {"states":(2,3), "detuning": 0, "rabi_frequency":7}

sensor_doppler.add_couplings(probe, coupling, rf)

gamma = np.zeros((basis_size, basis_size))
gamma[1:,0] = 0.1
sensor_doppler.set_gamma_matrix(gamma)
```

Again, this is a cartoon example, but it illustrates how to set up doppler broadening. In this case, we apply identical 100 Mhz doppler broadening in the x direction on both the coupling and probe lasers. With the lasers configured, we can now call the `solve_steady_state()` function to solve as before, this time with the optional argument `doppler=True`.

```
solution_doppler = rq.solve_steady_state(sensor_doppler, doppler=True)
print(f"Solution shape: {solution_doppler.rho.shape}")
```

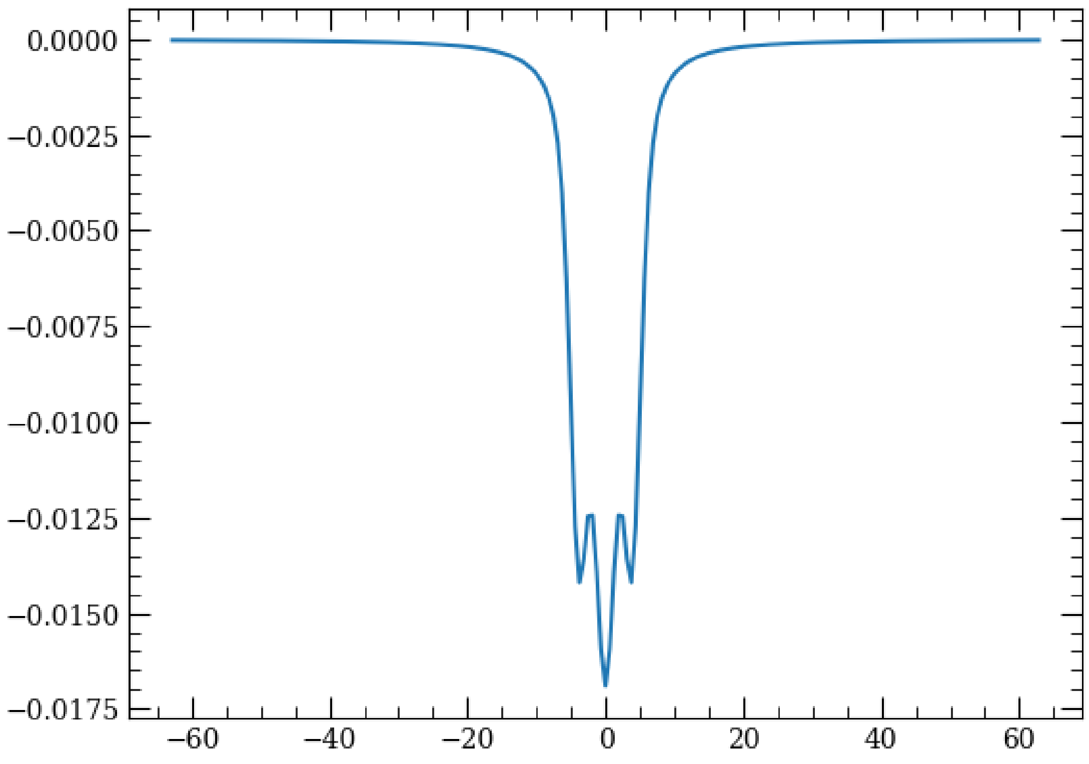
```
Solution shape: (201, 15)
```

Note that by default, the solution shape is the same as it was without doppler. `rydiqule` performs a weighted average of the solutions for each doppler slice. This behavior can be disabled with the optional `sum_doppler=False` flag. In this case, the wighted solution to each doppler class will be returned so they can be inspected.

```
absorption_doppler = rq.get_rho_ij(solution_doppler.rho,0,1).imag
print(f"Absorption_shape: {absorption_doppler.shape}")

plt.plot(detunings, absorption_doppler)
```

```
Absorption_shape: (201,)
```

```
[<matplotlib.lines.Line2D at 0x13b6ebf79a0>]
```



Notice that apart from the doppler broadening, this system is identical to the first one we solved over a range of detunings. Now, with the doppler broadening, its features have been smeared out a little as expected with a doppler distribution.

It is also worth mentioning that the same argument works identically in the `solve_time()` function.

3.7 5. `Cell` and real atoms

So far all of the functionality shown so far have used the `Sensor` object, providing the barebones physics needed to run simulations. However, in an applied setting, we would like to avoid manually filling all these parameters in with the properties of real atoms. Thus, we have created the `Cell` class, which inherites `Sensor`. This inheritence structure means that a `Cell` uses all the same functions as `Sensor`, in our case with some extra bells and whistles. First, and most importantly, we can specify a physical atom to associate with the sensor. This (behind the scenes) automatically fills in things like state lifetimes and transition frequencies.

The first argument one can specify a particluar Rydberg atom isotope via one of the following strings: `['H', 'Li6', 'Li7', 'Na', 'K39', 'K40', 'K41', 'Rb85', 'Rb87', 'Cs']`, correspoonding respecively to Hydrogen-1, Lithium-6, Lithium-7, Sodium-23, Potassium-39, Potassium-40, Potassium-41, Rubidium-85, Rubidium-87, and Caesium-133. After the atom, the next arguments are sequence of atomic states formatted with the Quantum numbers `[n, l, j, m]`. If we wanted to create a `Cell` with the D1 line of the Rubidium atom for example, we might do the followingL

```
Rb_Cell = rq.Cell('Rb85', [5, 0, 1/2, 1/2], [5, 1, 1/2, 1/2])
```

Of course, it might get tediout to add common transitions like D1 and D2, so Rydiqule adds a shorthand for calculating them, `D1_states` and `D2_states`. These can be specified either with a principle quantum number `n`, or with one of the strings we use to specify the atom (additionally, the non-isotope-specified chemical symbol can be used, such as `"Rb"`).

```
print(rq.D1_states('Rb85'))
print(rq.D2_states(5))
```

```
([5, 0, 0.5, 0.5], [5, 1, 0.5, 0.5])
([5, 0, 0.5, 0.5], [5, 1, 1.5, 0.5])
```

Using this method, the cell can be created as follows:

```
atom = 'Rb85'
Rb_Cell = rq.Cell(atom, *rq.D1_states(atom))
```

So we have created out cell, but how can we be sure what states are in the system? We can use the `Cell.states_list()` function.

```
print(Rb_Cell.states_list())
```

```
[[5, 0, 0.5, 0.5], [5, 1, 0.5, 0.5]]
```

These are the 2 states we are considering in our `Cell`, formatted with the quantum numbers `[n, l, j, m]`. As you can see, they do indeed correspond to the D1 line of Rubidium-85. But what if we wanted to add more states? Additional states can be defined as part of the system, but must be declared when the `Cell` is created. Let's add something in a higher state $n = 20$.

```
atom = 'Rb85'
new_state = [20, 1, 1.5, 1,5]
Rb_Cell = rq.Cell(atom, *rq.D1_states(atom), new_state)
```

Importantly, while the constructor is a little, different, it is still a `Sensor` under the hood, we just no longer need to specify transition frequencies and decoherences manually (although decoherences can still be added further if desired). As such we can call all the usual sensor functions, and all the information is still stored on the `couplings` atribute of `Cell`.

Additional self-broadening terms can be added to the decoherence matrix using the usual methods. Note that to make a new decoherence additive rather than overriding something, give it a unique label.

```
gamma_collisional = 2*np.pi*1.0 #radMHz
Rb_Cell.add_self_broadening(1, gamma_collisional, label="collisional")
print(Rb_Cell.decoherence_matrix())
```

```
[[ 0.41172855  0.          0.        ]
 [36.33703603  6.28318531  0.        ]
 [ 0.48781242  0.          0.        ]]
```

So we now have a 3-level system with the expected states, so far so good. And it even generated a gamma matrix based on state lifetimes! From here we can add couplings as before.

Be aware that when using `Cell`, if `detuning` is not specified, the Hamiltonian will be written in the laboratory frame and the frequency of the transition will be calculated automatically. This can lead to very large frequencies in the Hamiltonian,

```
detunings = 2*np.pi*np.linspace(-10,10,201) #201 values between -10 and 10 MHz
laser_01 = {"states": (0,1), "detuning": detunings, "rabi_frequency": 3}
laser_12 = {"states": (1,2), "detuning": 2, "rabi_frequency": 5}
Rb_Cell.add_couplings(laser_01, laser_12)
```

To drive home that this really is just a sensor, let's have a look at the `couplings` attribute.

```
print(Rb_Cell.couplings.nodes(data=True))
print(Rb_Cell.couplings.edges(data=True))
```

```
[(0, {'qnums': [5, 0, 0.5, 0.5], 'energy': 0, 'gamma_lifetime': 0}), (1, {'qnums':␣
↪[5, 1, 0.5, 0.5], 'energy': 2369435838.3044744, 'gamma_lifetime': 35.
↪92530747427563}), (2, {'qnums': [20, 1, 1.5, 1, 5], 'energy': 6277562673.179859,
↪'gamma_lifetime': 0.07608386623561031})]
[(0, 0, {'gamma_transit': 0.41172855461658464, 'label': '(0,0)'}), (0, 1, {'rabi_
↪frequency': 3, 'detuning': array([-62.83185307, -62.20353454, -61.57521601, -60.
↪94689748,
       -60.31857895, -59.69026042, -59.06194189, -58.43362336,
       -57.80530483, -57.1769863 , -56.54866776, -55.92034923,
       -55.2920307 , -54.66371217, -54.03539364, -53.40707511,
       -52.77875658, -52.15043805, -51.52211952, -50.89380099,
       -50.26548246, -49.63716393, -49.0088454 , -48.38052687,
       -47.75220833, -47.1238898 , -46.49557127, -45.86725274,
       -45.23893421, -44.61061568, -43.98229715, -43.35397862,
       -42.72566009, -42.09734156, -41.46902303, -40.8407045 ,
       -40.21238597, -39.58406744, -38.9557489 , -38.32743037,
       -37.69911184, -37.07079331, -36.44247478, -35.81415625,
       -35.18583772, -34.55751919, -33.92920066, -33.30088213,
       -32.6725636 , -32.04424507, -31.41592654, -30.78760801,
       -30.15928947, -29.53097094, -28.90265241, -28.27433388,
       -27.64601535, -27.01769682, -26.38937829, -25.76105976,
       -25.13274123, -24.5044227 , -23.87610417, -23.24778564,
       -22.61946711, -21.99114858, -21.36283004, -20.73451151,
       -20.10619298, -19.47787445, -18.84955592, -18.22123739,
       -17.59291886, -16.96460033, -16.3362818 , -15.70796327,
       -15.07964474, -14.45132621, -13.82300768, -13.19468915,
       -12.56637061, -11.93805208, -11.30973355, -10.68141502,
       -10.05309649,  -9.42477796,  -8.79645943,  -8.1681409 ,
        -7.53982237,  -6.91150384,  -6.28318531,  -5.65486678,
        -5.02654825,  -4.39822972,  -3.76991118,  -3.14159265,
        -2.51327412,  -1.88495559,  -1.25663706,  -0.62831853,
         0.        ,   0.62831853,   1.25663706,   1.88495559,
         2.51327412,   3.14159265,   3.76991118,   4.39822972,
```

(continues on next page)

(continued from previous page)

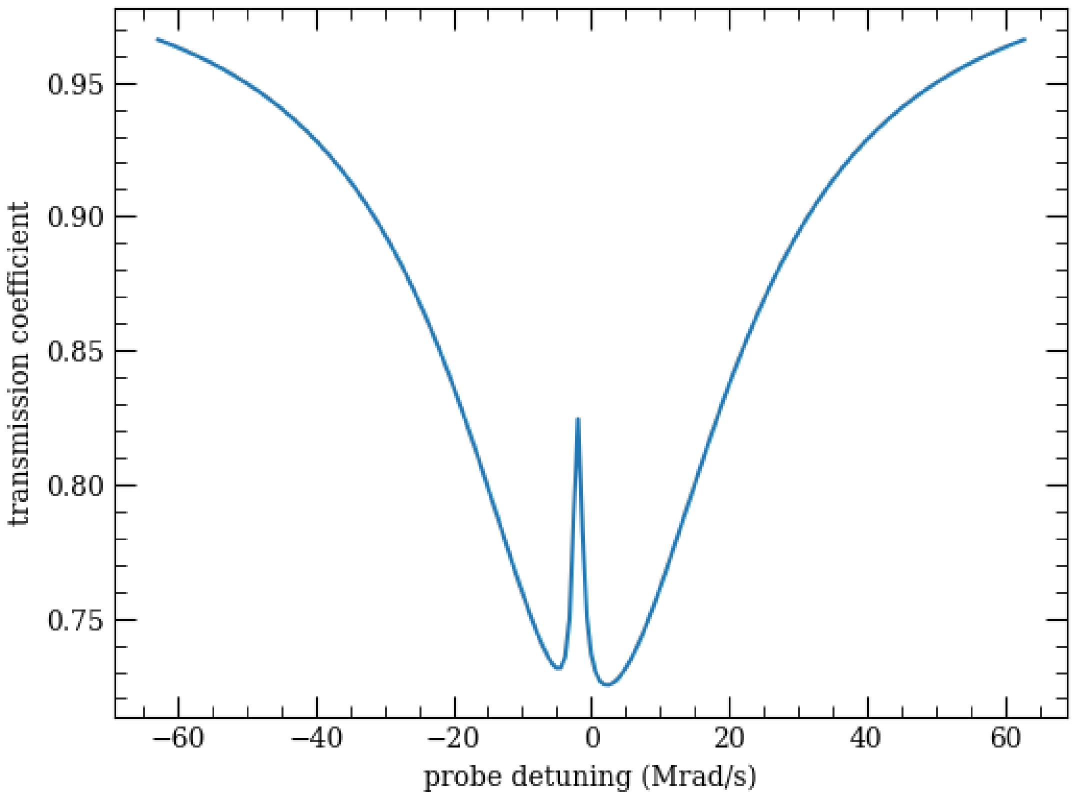
```
          5.02654825,   5.65486678,   6.28318531,   6.91150384,
          7.53982237,   8.1681409 ,   8.79645943,   9.42477796,
         10.05309649,  10.68141502,  11.30973355,  11.93805208,
         12.56637061,  13.19468915,  13.82300768,  14.45132621,
         15.07964474,  15.70796327,  16.3362818 ,  16.96460033,
         17.59291886,  18.22123739,  18.84955592,  19.47787445,
         20.10619298,  20.73451151,  21.36283004,  21.99114858,
         22.61946711,  23.24778564,  23.87610417,  24.5044227 ,
         25.13274123,  25.76105976,  26.38937829,  27.01769682,
         27.64601535,  28.27433388,  28.90265241,  29.53097094,
         30.15928947,  30.78760801,  31.41592654,  32.04424507,
         32.6725636 ,  33.30088213,  33.92920066,  34.55751919,
         35.18583772,  35.81415625,  36.44247478,  37.07079331,
         37.69911184,  38.32743037,  38.9557489 ,  39.58406744,
         40.21238597,  40.8407045 ,  41.46902303,  42.09734156,
         42.72566009,  43.35397862,  43.98229715,  44.61061568,
         45.23893421,  45.86725274,  46.49557127,  47.1238898 ,
         47.75220833,  48.38052687,  49.0088454 ,  49.63716393,
         50.26548246,  50.89380099,  51.52211952,  52.15043805,
         52.77875658,  53.40707511,  54.03539364,  54.66371217,
         55.2920307 ,  55.92034923,  56.54866776,  57.1769863 ,
         57.80530483,  58.43362336,  59.06194189,  59.69026042,
         60.31857895,  60.94689748,  61.57521601,  62.20353454,
         62.83185307]), 'phase': 0, 'kvec': (0, 0, 0), 'no_rwa_warning': False,
↪'label': '(0,1)'}), (1, 0, {'gamma_transition': 35.92530747427563, 'label': '(1,
↪0)', 'gamma_transit': 0.41172855461658464}), (1, 1, {'gamma_collisional': 6.
↪283185307179586, 'label': '(1,1)'}), (1, 2, {'rabi_frequency': 5, 'detuning': 2,
↪'phase': 0, 'kvec': (0, 0, 0), 'no_rwa_warning': False, 'label': '(1,2)'}), (2,␣
↪0, {'gamma_transition': 0.006131399733258158, 'label': '(2,0)', 'gamma_transit':␣
↪0.41172855461658464})]
```

There is a **lot** more information, but the structure is identical. The only difference is that values like decohereces from natural state lifetimes have been added automatically. If we added states without the RWA, transition frequencies would be added automatically as well, since "absolute energy" (energy difference with the ground state) is stored on the nodes. Given that it is the same as any other `Sensor`, let's show that by solving the system as we wold have before, then we can call it a day!

IN this case, we can easily call a convenience function, `rq.get_transmission_coef()` to return the cell transmission coefficient

```
sol = rq.solve_steady_state(Rb_Cell)
absorption_cell = rq.get_transmission_coef(sol, Rb_Cell, cell_length = .001,
                                           probe_tuple = (0, 1))
plt.plot(detunings, absorption_cell)
plt.xlabel('probe detuning (Mrad/s)')
plt.ylabel('transmission coefficient')
```

```
Text(0, 0.5, 'transmission coefficient')
```



This is a very minimal example introducing the concept of the `Cell` class as an extension of `Sensor`. For a more detailed example of its use, other notebooks make heavy use of the `Cell` to do meaningful physics. It also illustrates that `Sensor` is extensible, which actually has some very helpful implications for what you can do with `rydiqule`. `Sensor`, or even `Cell`, can be extended further to model very specific experimental setups so they can be re-created easily, specifying only the parameters that are likely to change. If you are comfortable with object-oriented programming in python, the sky is the limit in terms of what you can store on the `couplings` graph and what you can calculate automatically. From here, feel free to play around with other notebooks, and use `rydiqule` for your own problems! We are constantly trying to improve the library, so feedback is welcome.

3.8 Acknowledgments

The authors recognize financial support from the US Army and Defense Advanced Research Projects Agency (DARPA). Rydiqule has been approved for unlimited public release by DEVCOM Army Research Laboratory and DARPA. This software is released under the xx licence through the University of Maryland Quantum Technology Center. The views, opinions and/or findings expressed here are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

CHAPTER

FOUR

CHANGELOG

4.1 v1.0.0rc1

4.1.1 Improvements

- Added a warning in cell if `add_coupling` is called a dipole-forbidden transition.
- The zip_parameters function can now be called on parameters of different types (e.g. detuning with rabi_frequency)
- The time solver now can call ivp solvers outside its own module. This allows for more quickly using different backend solvers for time-dependent problems.
- Implement timesolver backends based on CyRK's cython and numba ode solvers
- Optimize scipy backend of the timesolver for smaller dimensional problems

4.1.2 Bug Fixes

- Fixed issue where solvers would save doppler axes labels and values even when they are summed over to the solution object
- Fixed a bug where energy level diagrams broke when decochernce rates were scanned.
- Fixed issue where compiled timesolvers could not solve doppler averaged problems.
- Fixed issue where certain doppler solves could not be sliced correctly

4.1.3 Deprecations

4.2 v0.5.0

4.2.1 Improvements

- Add isometric-population meshing option to `doppler_mesh`
- Allow `get_rho_ij` to accept a `Solution` object directly, in addition to solution numpy arrays
- Add `get_rho_populations` helper function to efficiently get the trace of density matrix solutions
- Allow `beam_power` or `beam_waist` to be scanned parameters in a `Cell` coupling
- Add more information to `Solution` objects returned by the solvers
- Allow dephasings to be scannable parameters.
- Updated the framework for scanning parameters to generate relevant lists on the fly

- Note: This changes the order of axes in a stack. Previously, the axes would be ordered based on the order they were added to the system. They are now ordered based on python's `sort()` applied to a tuple of ((low_state, high_state), parameter_name). As a result, they will be ordered first by lower state, then by upper state, then alphabetically by parameter name (e.g. "detuning", "rabi_frequency") In cases where the code was being used for simulations, this may affect cases where axes were defined specifically by number, and these may need to be updated.

- Added a distinction between stack shapes in steady-state vs time-dependent. For example, a steady-state hamiltonian stack may have shape `(10,1,3,3)` while the time dependent portion may have shape `(1,25,3,3)`.
- Renamed the `ham_slice` function to `matrix_slice` and allowed it to iterate over any number of matrices. - Updated internals of solver functions to use this framework.
- `zip_parameters` function no longer enforces parameters be the same type.

4.2.2 Bug Fixes

- Fixed several issues with parameter zipping functionality producing errors when sensor methods were called multiple times.
- Fixed issue where `get_rho_ij` incorrectly calculated the `rho_00` element
- Allow `Cell.add_coupling` to accept a list of e-field values
- Fixed an bug where specifying a list of `rabi_frequency` in a coupling with `time-dependence` would raise an error when solved
- Fixed issue with dephasing broadcasting preventing hamiltonian slices for large solves

4.2.3 Deprecations

- Removed all `sensor_management` functionality as too difficult to maintain generally and securely.
- Removed the internal `_variable_couplings`, `_variable_parameters`, and `_variable_values` attributes from sensor.

4.3 v0.4.0

4.3.1 Improvements

- Changed the handling of decoherent transitions to be stored on graph edges rather than as a separate attribute.
  - Gamma matrix is now calculated on the fly with the `decoherence__matrix()` method.
  - Decoherent transitions are now added with with the `add_decoherence()` function in `Sensor`.
  - `Cell` now calculates tranistion frequencies and decay rates automatically and places them on the appropriate graph edges.
- Changed the `Sensor.couplings` attribute from a `nx.Graph` to an `nx.DiGraph`. This has multiple advantages:
  - A less vague definition of detuning convention.
  - Precise definition of energy ordering: couplings now always point from lower to higher absolute energy.
  - More flexibility in decoherence. Decoherent transions now point "from" one state "to" another rather than just "between" 2 states. This fixes a limitation where gamma matrices no longer must be lower triangular.
- `get_snr()` function in `rq.experiments` now takes `kappa` and `eta` as optional arguments to allow for running on any `Sensor` object. They can still be inferred from a `Sensor` subclass that has them as attributes if unspecified.

- time solver now properly handles complex time dependences in the rotating wave approximation
- Added type hints to code base that can be used to static type check with mypy
- Added functions `rq.calc_kappa` and `rq.calc_eta` to properly calculate kappa and eta constants for experimental parameters.
- Added function `rq.get_OD` that calculates the optical depth of a solution
- Improved accuracy of the solver memory estimates
- Increased input validation unit test coverage
- Generalized handling of transit broadening to allow for multiple repopulation states with varying branching ratios

4.3.2 Bug Fixes

- Fixed an issue with time dependence in the probe laser
- Modified solver to allow for complex time dependence
- Fixed non-hermitian hamiltonians in time solver
- Fixed error with multiple time-dependences in time solver
- Added functionality to solver error with complex time dependences
- Modified experimental return functions (`get_transmission_coef()`, `get_phase_shift()`, and `get_susceptibility()`\`) to allow scanning of probe rabi frequency
- Fixed `get_rho_ij` so that it correctly calculates the `(0,0)` population element
- Fix error in `test_sensor_management` which fails if temporary directory does not exist.
- Tighten `test_decoherences` tolerances to the 2pi*100Hz level to catch errors in decoherence matrix generation.
- Fixed issue where `get_snr` ignored the optical path length input parameter
- Fixed issue where calling `solve_steady_state` with `sum_doppler=False` would double memory footprint.
- Fixed issue where `solve_steady_state` could be called with `weight_doppler=False` and `sum_doppler=True`.

4.3.3 Deprecations

- `get_snr` no longer allows manually specifying `Sensor.eta` and `Sensor.kappa`, these values must be passed as args for Sensor input
- Removed unused `gamma_transit` argument from Sensor init
- Re-ordered argument list to `Cell.add_coupling` to match order of `Sensor.add_coupling`
- `Sensor.add_fields` has been fully removed and no longer works as a deprecated alias of `Sensor.add_couplings`

4.4 v0.3.0

4.4.1 Improvements

- Expanded documention
- Removed restrictions on ARC and numpy versions during installation.
- Vectorized equation of motion generation to support prepending axes to a hamiltonian
- Updated the internal mechanism for sensor handling fields of various type
  - Fields are now internally called couplings
  - Fields are specified as either having rabi_frequency or transition_frequency, corresponding to RWA or non-RWA fields
  - Fields are specified as either having detuning or transition_frequency, corresponding to steady-state or time-dependent fields
  - Fields with specific traits can be accessed with the `couplings_with()` function
- Added a feature to save/load sensors/cells
- Implemented NumbaKitODE which considerably speeds up solve_time. This feature can be enabled by setting parameter compile=True of solve_time.
- Improved logic for building diagonal terms of Hamiltonian using NetworkX graph library that allows for diagonal terms to be built from any set of values.
- Generalized doppler averaging to support prepended axes on hamiltonians.
- Improved time solver logic for improved modularity across doppler solving and multivalue parameters.
- Added a feature to draw level diagram
- Seamlessly generate all Hamiltonians from lists of parameters in sensor.
- Added ability to label couplings.
- Added capability to make any coupling time-dependent
- Sped up time solving considerably by simultaneously solving all equations rather than looping.
- Allow for user to specify fields by beam power, beam waist, and electric field, in the Cell framework.
- Solve functions now return a bunch-type object rather than a tuple.
- Added functionality that breaks equations into slices based on memory requirements
- Quantum numbers and absolute energies are now stored on the nodes of a Cell couplings graph
- Cell now adds decay rates and decoherences to the nodes and edges of the Cell couplings graph
- Cell now calculates the gamma matrix in an arbitrary way, and is no longer limited to two laser, ladder schemes
- Added function to calculate sensor SNR with repect to any varied sensor coupling parameter
- Added function to return sensor parameter mesh

4.4.2 Bug Fixes

- Fixed example notebook.
- Fixed issue where doppler averaging breaks if there are uncoupled levels.
- Fixed doppler averaging so that doppler shifts are applied with signs consistent with the hamiltonian.
- Fixed a bug where doppler averaging did not properly solve separately for each doppler class.
- Fixed issue where spatial dimension of doppler averaging is not introspected correctly in the presence of round-off errors.

4.4.3 Deprecations

- All "field" functionality are being deprecated in favor of "coupling"
- The `rf_couplings`, `target_state`, and `rf_dipole_matrix` arguments of `solve_time()`
- All functions relating to sensor.transtion_map are deprecated
- Cell now does not accept gamma_excited or gamma_Rydberg as these are always calculated or Sensor can be used with a given gamma matrix
- Cell now does not accept gamma_doppler as Doppler broadening width is given by mutiplying the most proable velocity and the laser k-vector

4.5 v0.2.0

Beta release. Contains very large number of backwards-incompatible changes over alpha release.

4.6 v0.1.0

Alpha release. Minimum viable product release that does basic modeling tasks slowly.

CHAPTER

FIVE

PHYSICS DOCUMENTATION

The following pages contain white-ups that explain the theoretical basis for the many operations performed by rydiqule.

5.1 Stacking Conventions

5.1.1 Introduction

For many probelms, Rydiqule is designed to implicitly handle multiple possible values for a single parameter. For example, sweeping over a range of detuning values is handled in Rydiqule simply by specifying the value of interest a list or array rather than a single value. This enables a tremendous amount of flexibility in the problems that Rydiqule can solve naturally, but there are some things worth noting about how Rydiqule specifically handles these problems, which are outlined in this document.

5.1.2 Numpy arrays

Typically, python lists are quite slow to perform operations on since they are dynamically sized and typed. This allows tremendous flexibility in what can be put into a list but some problems with how fast elements are accessed from that list and operating on. `Numpy arrays[https://numpy.org/doc/stable/user/whatisnumpy.html]_` were created to address this limitation and Rydiqule makes extensive use of them to make its calculations fast without losing the ease-of-use benefits of a Python interface. Fundamentally, a numpy `ndarray` is a grid of numbers that has dimensionality `m1` by `m2` by `m3` and so on. Numpy routines are written to operate on these arrays very quickls for large numbers of dimensions.

5.1.3 Stacking

While numpy's own way of handling arrays via matrix broadcasting is `well-documented[https://numpy.org/doc/]_`, and most of Rydiqule's own functions use the standard numpy conventions, there are some additional assumptions Rydiqule makes when performing these operations that are worth outlining. Fundamentally, Rydiqule thinks about these `ndarray` objects as groups of matrices, meaning that calculations are performed assuming, for example, that an array of shape `(25, 3, 3)` represents 25 3×3 matrices. This is the array that would be generated if a list of 25 values were provided for a detuning value in a 3-level `Sensor`, and that `Sensor`'s `get_hamiltonian` function were called. Rydiqule seamlessly handles all the work of generating those Hamiltonian matrices for each value, and returns a single array object as an output. Similarly, if 2 values are specified as lists of length 25, a single arary of shape `(25, 25, 3, 3)` would be returned, with a different 3×3 Hamiltonian matrix for every combination of parameter values, for a total of 625 Hamiltonian matrices. Rydiqule terms this array a "stack" of Hamiltonians, and the "stack shape" are the axes preceding the actual matrix value axes (in this case `(25, 25)`), and is typically, denoted in Rydiqule as `*l` to make clear that it could be any length of set of values depending on the problem.

Hamiltonian generations is created using this convention, and that carries through to generation of equations of motion, and any other quantities that may have a different matrix for each parameter value. A Hamiltonian stack of shape `(*l, 3, 3)` will generate an equation of motion (eom) stack of shape `(*l, 8, 8)`, with all stack demensions remaining consistent. Rydiqule's internals are, broadly, agnostic to exaclty what the dimensions `*l` represent, and work regardless, as long as the dimensions corresponding to the actual quantities are in the expected position at the end.

5.1.4 Parameter Ordering

Given that any number of parameters may be defined as a list, Rydiqule needs a convention to ensure, in the final result, the values represent what is expected and has not been turned around. It is important to Rydiqule's design philosophy that internal variables not by tracked opaquely, and that quantities are, to the extent possible, generated on the fly in a predictable and reproducible way. This begs the questions, which axis corresponds to which value? Suppose the coupling between states 0 and 1 is swept in detuning over 25 values, as is the coupling between states 1 and 2, the stack shape will be `(25, 25)`, but there are some uncertainties. One might assume that the first axis corresponds the the first laser, and the second axis corresponds to the second laser. However, this is not necessarily obvious, and it might be the other way around without a unifying convention. Rydiqule's solution to this problem turns out to be simple: python's `.sort()` function. Since it always orders things accordning to the same rules, there is a predictable outcome to which axis is which. The parameters are represent by tuples: `((0,1),"detuning")` and `((1,2),"detuning")`. `.sort()` will sort them first by the lower state of a transition, then by the upper state, then alphabetically by the string parameter name (in this case detuning for both).

With this simple convention, Rydiqule makes these arrays consistent accross functions. One can be sure that all values will be exaclty what is expected and line up properly for all quantities. Hamiltonians, equations of motion, and solutions will all use the same rules. To avoid figuring this out manually for every system, the `Sensor` module contains the `.axis_labels()` method, which returns a list of which axes are which in string form for results interpretation. Note that the internal functions which calculate these values don't actually care what the axis are, but they do keep them consistent between calculations.

5.1.5 Doppler

It is worth a quick note how Rydiqule handles doppler broadening, because it leverages the same conventions around stacking as other parameter scans, and it may be encountered and cause confusion if you use Rydiqule enough. If doppler is accounted for in a solve, that typically is not invoked until the relevant `solve` function is called. Given a case of `n_doppler` velocity classes in 1 dimension, a new axis will be prepended to the stack, resuling in a `n_doppler` new dopple-shifted Hamiltonian matrices matrix that was previously in the stack. Typically, this is done under the hood, and these other solutions are averaged over before a result is returned, but examining intermediate values may ultimately result in seeing these axes, even if they are not present in the solution that is returned. Importantly, the solver internals are still agnostic to what these preceding doppler axes represent, giving flexibility and allowing a single function to handle all cases. Again, this is an intermediate step that typically does not affect how results are interpreted, it just helps to understand the internals a little better.

5.2 Equations of Motion Generation

5.2.1 Introduction

This document details a few notes about the theoretical operations taking place under the hood in the Rydiqule Modelling package. In particular, we discuss the methods that Rydiqule uses to numerically solve differential equations for density matrices.

5.2.2 Hamiltonian and Rotating Wave Approximation

For a two level atom interacting with an electric field **E**, the dipole interaction Hamiltonian is,

$$H = \omega \left|e\right\rangle\left\langle e\right| - \mathbf{d}\cdot\mathbf{E}$$

where the Rabi frequency is defined as $\Omega = \mathbf{d}\cdot\mathbf{E}/\hbar$. The electric field is,

$$\begin{aligned}\mathbf{E} &= \mathbf{E}_0\cos(\omega t + \phi)\\ &= \frac{\mathbf{E}_0}{2}(e^{i\omega t} + e^{-i\omega t})\end{aligned}$$

The dipole operator can be written [1]

$$\mathbf{d} = \left\langle g\right|\mathbf{d}\left|e\right\rangle(\left|g\right\rangle\left\langle e\right| + \left|e\right\rangle\left\langle g\right|)$$

The operator $\left|g\right\rangle\left\langle e\right|$ evolves at frequency $e^{i\omega t}$ under the bare Hamiltonian, so we expand and take the slowing evolving terms (RWA, see [1]).

$$\begin{aligned}H_{\text{RWA}} &= \omega\left|e\right\rangle\left\langle e\right|\\ &- \left\langle g\right|\mathbf{d}\left|e\right\rangle\cdot\frac{\mathbf{E}_0}{2}(\sigma^+e^{-i\omega t} + \sigma^-e^{i\omega t})\end{aligned}$$

where $\sigma^+ = \left|g\right\rangle\left\langle e\right|$ and $\sigma^- = \left|e\right\rangle\left\langle g\right|$

5.2.3 Equations of Motion

The Master equation is,

$$\dot{\rho} = -i[H,\rho] - \mathcal{L}$$

We write this in summation notation,

$$\dot{\rho}_{ij} = -i(H_{ik}\rho_{kj} - \rho_{ik}H_{kj}) - L_{ij}$$

More, generally, we can re-write this equation as a matrix equation,

$$\dot{\rho}_{ij} = R_{ik}\rho_{kj}$$

By re-shaping these equations, using `numpy.reshape`, we can convert this into a linear set of differential equations in matrix form (see, for example `Sensor._hamiltonian_term()` and `Sensor._decoherence_term()` in Rydiqule). I am not going to focus on these right now. I will call the re-shaped density vector p

$$\dot{p}_l = M_{li}p_i$$

This is a linear set of equations we can easily solve with `linalg.solve`. As a note, our reshaping procedure produces a **basis that is**, for basis size **b**

$$l = b\times j + i$$

For example,

l	ij
0	00
1	10
2	20
3	01
4	11
5	21

For the programmatic code, we need knowledge of this relationship.

5.2.4 Removing the Ground State

The density vector (matrix) is physically constrained, so that the total population is one. This constraint is not included in the equations of motion. This leads to numerical instabilities. The best way to fix this instability is to algebraically remove one of the equations of motion (ie the ground state). To remove the ground state, we apply the constraint

$$\rho_{00} = 1 - \rho_{ii}.$$

Writing this in terms of ρ' gives,

$$p_0 = 1 - \sum_x p_{[(b+1)\times x]}$$

We use this to re-write Eq. \ref{eq:master},

$$\dot{p}_l = M_{li} p_i - M_{l0} p_0 + M_{l0}(1 - \sum_x p_{[(b+1)\times x]})$$

This is the equation we must implement to remove the ground state.

In the code, we can apply Eq. \ref{eq:groundRemoved} and then we can simply remove the first column of M_{il}. In the code, we implement this transformation by replacing the set of equations M_{li},

$$M_{li}\rho_i \rightarrow (M_{li} + M'_{li})\rho_i + c_l$$

The constant term c is equivalent to the first column of M_{li}.

$$c_l = M_{l0}$$

The term we need to add, M' is

$$M'_{li} = -M_{l0} \sum_x p_{[i=(b+1)\times x]}$$

This can be implemented as the tensor product of two vectors

$$M'_{li} = -M_{l0} \otimes p*$$

where M_{i0} is just $M[:,0]$ and $p* = p_{[j=(b+1)\times x]}$ is a vector of ones and zeros that is generated with list comprehension.

The end result is an equation where each ground state term of the density matrix $\rho_0 0$ is replaced by the sum of all excited states.

5.2.5 Making the Equations Real

Numerically, converting to a real set of equations is important, because it prohibits the buildup of “imaginary populations” in quantum states. In other words, some equations in the equations of motion are physically required to be real, and some are complex. Machine rounding errors causes leakage into the imaginary parts of the populations equation, which is unphysical. Under certain solving conditions the equations are not stable to this buildup. Converting all the equations to real solves the issue.

The equation we want to solve (for the density vector p) is,

$$\dot{p_c} = M_c \cdot p_c + c_c$$

where the $_c$ notation represents that each term is complex.

The change in basis that we implement is shown below in equation and table format,

$$\begin{aligned}\rho_{ii} &\rightarrow \rho_{ii}\\ \rho_{ij} &\rightarrow Re(\rho_{ij}),\ i > j\\ \rho_{ji} &\rightarrow Im(\rho_{ij}),\ i < j\end{aligned}$$

l	real ij	complex ij
0	ρ_{00}	ρ_{00}
1	ρ_{10}	$\mathrm{Re}(\rho_{10})$
2	ρ_{20}	$\mathrm{Re}(\rho_{20})$
3	ρ_{30}	$\mathrm{Re}(\rho_{30})$
4	ρ_{01}	$\mathrm{Im}(\rho_{10})$
5	ρ_{11}	ρ_{11}

We implement this with a transformation matrix U that is unitary up to a scale factor,

$$\begin{aligned} M_r &= U \cdot M_c \cdot U^{-1} \\ c_r &= U \cdot c_c \end{aligned}$$

This matrix is calculated in the `get_basis_transformation()` helper function and is subsequently used to transform between the complex and real bases.

[1]D. A. Steck, *Quantum and Atom Optics*, 0.13.15 ed. (2022).

CHAPTER

SIX

API DOCUMENATION

rydiqule	Parent computational module.

6.1 rydiqule

Parent computational module.

Modules

rydiqule.atom_utils	Utilities for interacting with atomic parameters and ARC.
rydiqule.cell	Physical Cell objects for use in solvers.
rydiqule.doppler_utils	Utilities for implementing Doppler averaging
rydiqule.energy_diagram	Energy profile diagram
rydiqule.experiments	Standard methods for converting results to physical values.
rydiqule.rydiqule_utils	General rydiqule package utilities
rydiqule.sensor	Sensor objects that control solvers.
rydiqule.sensor_solution	Bunch-like object use to store aspects of a solution when calling rydiule.solve() Adds essential keys with "None" entries
rydiqule.sensor_utils	Utilities used by the Sensor classes.
rydiqule.solvers	Steady-state solvers of the Optical Bloch Equations.
rydiqule.timesolvers	Solvers for time domain analysis with an arbitrary RF field

6.1.1 rydiqule.atom_utils

Utilities for interacting with atomic parameters and ARC.

Module Attributes

`ATOMS`	Alkali atoms defined by ARC that can be used with `Cell`.

rydiqule.atom_utils.ATOMS

`rydiqule.atom_utils.`**`ATOMS = {'Cs': <class 'arc.alkali_atom_data.Caesium'>, 'H': <class 'arc.alkali_atom_data.Hydrogen'>, 'K39': <class 'arc.alkali_atom_data.Potassium39'>, 'K40': <class 'arc.alkali_atom_data.Potassium40'>, 'K41': <class 'arc.alkali_atom_data.Potassium41'>, 'Li6': <class 'arc.alkali_atom_data.Lithium6'>, 'Li7': <class 'arc.alkali_atom_data.Lithium7'>, 'Na': <class 'arc.alkali_atom_data.Sodium'>, 'Rb85': <class 'arc.alkali_atom_data.Rubidium85'>, 'Rb87': <class 'arc.alkali_atom_data.Rubidium87'>}`**

Alkali atoms defined by ARC that can be used with `Cell`.

Functions

`D1_states`(n)	Retrieve the quantum numbers for the states corresponding to the D1 line for a given Rydberg atom or principle quantum number.
`D2_states`(n)	Retrieve the quantum numbers for the states corresponding to the D2 line for a given Rydberg atom or principle quantum number.

rydiqule.atom_utils.D1_states

`rydiqule.atom_utils.`**`D1_states`**(*n: Union[int, str]*)

Retrieve the quantum numbers for the states corresponding to the D1 line for a given Rydberg atom or principle quantum number.

Parameters

n (`int or str`) – Either the string flag of the atom or the principle quantum number n of an atom. If string, must be one of ['H', 'Li', 'Li6', 'Li7', 'Na', 'K', 'K39', 'K40', 'K41', 'Rb', 'Rb85', 'Rb87', 'Cs'].

Returns

- *list* – Quantum numbers [n, l, j, m] of the atoms ground state.
- *list* – Quantum numbers [n, l, j, m] of the first excited state corresponding to the D1 line of the Rydberg atom.

rydiqule.atom_utils.D2_states

`rydiqule.atom_utils.D2_states`(*n: Union[int, str]*)

Retrieve the quantum numbers for the states corresponding to the D2 line for a given Rydberg atom or principle quantum number.

Parameters

n (*int or str*) – Either the string flag of the atom or the principle quantum number n of an atom. If string, must be one of [‘H’, ‘Li’, ‘Li6’, ‘Li7’, ‘Na’, ‘K’, ‘K39’, ‘K40’, ‘K41’, ‘Rb’, ‘Rb85’, ‘Rb87’, ‘Cs’].

Returns

- *list* – Quantum numbers [n, l, j, m] of the atoms ground state.
- *list* – Quantum numbers [n, l, j, m] of the first excited state corresponding to the D2 line of the Rydberg atom.

6.1.2 rydiqule.cell

Physical Cell objects for use in solvers.

Classes

`Cell`(atom_flag, *atomic_states[, ...])	Subclass of `Sensor` that creates a Sensor with additional physical properties corresponding to a specific Rydberg atom.

rydiqule.cell.Cell

class `rydiqule.cell.Cell`(*atom_flag: Literal['H', 'Li6', 'Li7', 'Na', 'K39', 'K40', 'K41', 'Rb85', 'Rb87', 'Cs'], *atomic_states: Sequence, gamma_transit: Optional[float] = None, cell_length: float = 0, beam_area: float = 1e-06, beam_diam: Optional[float] = None, temp: float = 300.0*)

Bases: `Sensor`

Subclass of `Sensor` that creates a Sensor with additional physical properties corresponding to a specific Rydberg atom.

In addition to the core functionality of `~.Sensor`, this class allows for labelling of states with quantum numbers, calculating of state lifetimes and decoherences and tracking of of some physical laser parameters. A key distictinction between a `Cell` and a `Sensor` is that a cell supports (and requires) and absolute ordering of energy between states, which allows for implicit calculation of decay rates an transition frequencies.

`__init__`(*atom_flag: Literal['H', 'Li6', 'Li7', 'Na', 'K39', 'K40', 'K41', 'Rb85', 'Rb87', 'Cs'], *atomic_states: Sequence, gamma_transit: Optional[float] = None, cell_length: float = 0, beam_area: float = 1e-06, beam_diam: Optional[float] = None, temp: float = 300.0*) → None

Initialize the Rydberg cell from the given parameters.

Parameters

- **atom_flag** (*str*) – Which atom is used in the cell for calculating physical properties with ARC Rydberg. One of [‘H’, ‘Li6’, ‘Li7’, ‘Na’, ‘K39’, ‘K40’, ‘K41’, ‘Rb85’, ‘Rb87’, ‘Cs’].
- **atomic_states** (*list[list]*) – List of states to be added to the cell. Each state is an iterable whose elements are each a list of the form [n, l, j, m], represnting the Rydberg atomic quantum numbers of the state. At least two states must be added so that the system is nontrivial. The number of states will determine the basis size of the system.

- **`gamma_transit`** (*`float`*, *`optional`*) – Decoherence due to atom transit through the optical beams. Specified in units of Mrad/s. If `None`, will calculate based on value of `beam_area`. See *`add_transit_broadening()`* for details on how transit broadening is treated. Default is None.
- **`beam_area`** (*`float`*, *`optional`*) – Area of probing field cross-section in m^2. Used to calculate `kappa` and `gamma_transit`. Default is 1e-6.
- **`beam_diam`** (*`float`*, *`optonal`*) – Diameter of the probing field cross section in meters. Used to calculate `gamma_transit`. If `None`, it is calculated from `beam_area` assuming the beam cross-section is a circle. Default is `None`.
- **`temp`** (*`float`*, *`optional`*) – Temperature of the gas in Kelvin. Used in calculations of enery level lifetime. Default is 300 K.

Raises

- **`ValueError`** – If at least two atomic states are not provided.
- **`ValueError`** – If atom_flag is not one of ARC's supported alkali atoms.

Methods

`__init__`(atom_flag, *atomic_states[, ...])	Initialize the Rydberg cell from the given parameters.
`add_coupling`(states[, rabi_frequency, ...])	Overload of `add_coupling()` which allows for different specification of coupling fields which are more reflective of real experimental setups.
`add_couplings`(*couplings[, no_rwa_warning])	Add any number of couplings between pairs of states.
`add_decoherence`(states, gamma[, label])	Add decoherent coupling to the graph between two states.
`add_self_broadening`(node, gamma[, label])	Specify self-broadening (such as collisional broadening) of a level.
`add_states`()	Deprecated.
`add_transit_broadening`(gamma_transit[, repop])	Adds transit broadening by adding a decoherence from each node to ground.
`axis_labels`([collapse])	Get a list of axis labels for stacked hamiltonians.
`basis`()	Generate basis labels of density matrix components.
`collapse_dims`(stack)	Collapse dimensions of a stack accoring to the `_zipped_parameters` attribute.
`couplings_with`(*keys[, method])	Returns a version of self.couplings with only the keys specified.
`decoherence_matrix`()	Get the decoherence matrix for a system.
`get_couplings`()	Returns the couplings of the system as a dictionary
`get_doppler_shifts`()	Returns the Hamiltonian with only detunings set to the kvector values for each spatial dimension.
`get_hamiltonian`()	Creates the Hamiltonians from the couplings defined by the fields.
`get_hamiltonian_diagonal`(values[, no_stack])	Apply addition and subtraction logic corresponding to the direction of the couplings.
`get_parameter_mesh`([sparse])	Returns the parameter mesh of the sensor.
`get_time_couplings`()	Returns the list of matrices of all couplings in the system defined with a `time_dependence` key.
`get_time_dependence`()	Function which returns a list of the `time_dependence` functions.
`get_time_hamiltonians`()	Get the hamiltonians for the time solver.
`get_transition_frequencies`()	Gets an array of the diagonal elements of the Hamiltonian from the field detunings.
`get_value_dictionary`(key)	Get subset of dictionary coupling parameters.
`level_ordering`()	Return a list of the integer numbers of each state (*not* the quantum numbers) in descending order by energy order.
`set_gamma_matrix`(gamma_matrix)	Set the decoherence matrix for the system.
`spatial_dim`()	Returns the number of spatial dimensions doppler averaging will occur over.
`states_list`()	Returns a list of quantum numbers for all states in the cell.
`unzip_parameters`(*labels, **extra_kwargs)	Remove a set of zipped parameters from the internal zipped_parameters list.
`variable_parameters`([apply_mesh])	Property to retrieve the values of parameters that were stored on the graph as arrays.
`zip_parameters`(*labels)	Define 2 scannable parameters as "zipped" so they are scanned in parallel.

Attributes

eta	Noise density prefactor, in units of root(Hz).
kappa	Differential prefactor, in units of (Mrad/s)/m.

_add_coupling(*states: Tuple[int, ...], **field_params*) → None

Function for internal use which will ensure the supplied couplings is valid, add the field to self.couplings.

Exists to abstract away some of the internally necessary bookkeeping functionality from user-facing classes.

Parameters

- **states** (*tuple*) – The integer pair of states to be coupled.
- ****field_params** (*dict*) – The dictionry of couplings parameters. For details on the keys of the dictionry see `add_coupling()`.

_add_decay_to_graph() → None

Internal helper method to add population decay rates to the nodes in order to calculate gamma matrix.

1. add the state lifetime to each node
2. add the transition rate to each edge

_add_states(**states: Sequence*) → None

Internal method to add states to the system and update internal variables for energy and decay rates.

Quantum numbers and other states information are stored on the couplings graph nodes. The first state added to the system is treated as the ground state, and all "absolute energies" are calculated as a difference from ground. Should only be called in `__init__()`.

Parameters

***states** (*list[list[int or float]]*) – States that are added to the list of atomic states of interest for the cell. Arguments should be lists of the form `[n, l, j, m]`, where n, l, j, and m are the ususal quantum numbers describing the state: principal, orbital, total angular momentum, and magnetic quantum numbers respectively.

_coupling_with_label(*label: str*) → dict

Helper function to return the pair of states corresponding to a particular label string. For internal use.

_get_probe_info(*q_optical: Literal[- 1, 0, 1] = 0*) → None

Internal helper method to get information about the probing transition.

For this function, the probe transition is defined as the transition between the ground state and first excited state.

Parameters

q_optical (*int, optional*) – polarization of probing optical field in spherical basis. Must be -1, 0, 1. Defaults to 0 for linear polarization.

_remove_edge_data(*states: Tuple[int, ...], kind: str*)

Helper function to remove all data that was added with a `add_coupling()` call or `add_decoherence()` call. Needed to ensure that two nodes do not have coherent couplings pointing both ways and to invalidate existing zip parameter couplings.

Parameters

- **states** (*tuple*) – Edge from which to remove data.
- **kind** (*str*) – What type of data to remove. Valid options are `coherent` coherent couplings or the incoherent key to be cleared (must start with `gamma`).

Raises

ValueError – If `kind` is not `'coherent'` and doesn't begin with `'gamma'`

`_stack_shape`(*collapse=True*, *time_dependence='all'*) → Tuple[int, ...]

Internal function to get the shape of the tuple preceding the two hamiltonian axes in `get_hamiltonian()`

`_states_valid`(*states: Tuple[int, ...]*) → Tuple[int, ...]

Confirms that the provided states are in a valid format.

Typically used internally to validate states added. If provided as a form other than a tuple, first casts to a tuple for consistent indexing.

Checks that `states` contains 2 elements, can be interpreted as a tuple, and that both states lie inside the basis.

Parameters

states (*iterable*) – iterable of to validate. Should be a pair of integers that can be cast to a tuple.

Returns

Length 2 tuple of validated state labels.

Return type

tuple

Raises

- **ValueError** – If `states` has more than two elements.
- **TypeError** – If `states` cannot be converted to a tuple.
- **ValueError** – If either state in `states` is outside the basis.

`add_coupling`(*states: Tuple[int, ...]*, *rabi_frequency: Optional[Union[float, List[float], ndarray]] = None*, *detuning: Optional[Union[float, List[float], ndarray]] = None*, *transition_frequency: Optional[float] = None*, *phase: Optional[Union[float, List[float], ndarray]] = 0*, *kvec: Tuple[float, float, float] = (0, 0, 0)*, *time_dependence: Optional[Callable[[float], float]] = None*, *label: Optional[str] = None*, *e_field: Optional[Union[float, List[float], ndarray]] = None*, *beam_power: Optional[float] = None*, *beam_waist: Optional[float] = None*, *suppress_dipole_warn: Optional[float] = False*, *q: Literal[- 1, 0, 1] = 0*, ***extra_kwargs*) → None

Overload of `add_coupling()` which allows for different specification of coupling fields which are more reflective of real experimental setups.

Rabi frequency is a mandatory argument in `Sensor` but in `Cell`, there are 3 options for laser power specification:

1. Explicit rabi-frequency definition identical to `Sensor`.
2. Electric field strength, in V/m.
3. Specification of both beam power and beam waist.

Any one of these options can be used in place of the standard`rabi_frequency` argument of `add_coupling()`.

As in `Sensor`, if `detuning` is specified, the coupling is assumed to act under the rotating-wave approximation (RWA), and `transition_frequency` can not be specified. However, unlike in a `Sensor`, if `detuning` is not specified, in a `Cell`, `transition_frequency` will be calculated automatically based on atomic properties rather than taken as an argument.

Parameters

- **states** (*sequence*) – Length-2 list-like object (list or tuple) of integers corresponding to the numbered states of the cell.
- **rabi_frequency** (*float, optional*) – The rabi frequency, in Mrad/s, of the coupling field. If specified, `e_field`, `beam_power`, and `beam_waist` cannot be specified.

- **detuning** (*float, optional*) – Field detuning, in Mrad/s, of a coupling in the RWA. If specified, RWA is assumed, otherwise RWA not assumed, and transition frequency will be calculated based on atomic properties.
- **phase** (*float, optional*) – The relative phase of the field in radians. Defaults to zero.
- **kvec** (*sequence, optional*) – A three-element iterable that defines the atomic doppler shift on a particular coupling field. It should have magntiude equal to the doppler shift (in the units of Mrad/s) of an atom moving at the Maxwell-Boltzmann distribution most probable speed, `vP=np.sqrt(2*kB*T/m)`. I.E. `np.linalg.norm(kvec)=2*np.pi/lambda*vP`. If equal to `(0,0,0)`, solvers will ignore doppler shifts on this field. Defaults to `(0,0,0)`.
- **time_dependence** (*scalar function, optional*) – A scalar function that specifies a time-dependent field. The time dependence function is defined as a funtion that returns a unitless value as a function of time that is multiplied by the `rabi_frequency` parameter.
- **label** (*str, optional*) – The user-defined name of the coupling. This does not change any calculations, but can be used to track individual couplings, and will be reflected in the output of *axis_labels()* Default None results in using the states tuple as the label.
- **e_field** (*float, optional*) – Electric field strenth of the coupling in Volts/meter. If specified, `rabi_frequency`, `beam_power`, and `beam_waist` cannot be specified.
- **beam_power** (*float, optional*) – Beam power in Watts. If specified, `beam_waist` must also be supplied, and `rabi_frequency` and `e_field` cannot be specified. `beam_power` and `beam_waist` cannot be scanned simultaneously.
- **beam_waist** (*float, optional*) – 1/e^2 Beam waist (radius) in units of meters. Only necessary when specifying `beam_power`.
- **q** (*int, optional*) – Coupling polarization in spherical basis. Valid values are -1, 0, 1 for $-\sigma$, linear, $+\sigma$. Default is 0 for linear.

Raises

- **ValueError** – If `states` is not a list-like of 2 integers.
- **ValueError** – If an invalid combination of `rabi_frequency`, `e_field`, `beam_power`, and `beam_waist` is provided.
- **ValueError** – If `tranistion_frequency` is passed as an argument (it is calculated from atomic properties).
- **ValueError** – If `beam_power` and `beam_waist` are both sequences.

Notes

Note: Note that while this function can be used directly just as in *Sensor*, it will often be called implicitly via *add_couplings()* which `Cell` inherits. While they are equivalent, the second of these options is often the more clear approach.

Note: Specifying the beam power by beam parameters or electric field still computes the `rabi_frequency` and adds that quantity to the `Cell` to maintain consistency across `rydiqule`'s other calculations. In other words, `beam_power`, `beam_waist`, and `e_field` will never appear as quantities on the graph of a `Cell`.

Examples

In the simplest case, it is clear physical properties are calculated automatically in a `Cell` All the familiar quantities are present, as well as many more.

```
>>> cell = rq.Cell("Rb85", *rq.D2_states("Rb85"))
>>> cell.add_coupling(states=(0,1), detuning=1, rabi_frequency=2)
>>> print(dict(cell.couplings.edges))
{(0, 0): {'gamma_transit': 0.41172855461658464, 'label': '(0,0)'}, (0, 1):
↪{'rabi_frequency': 2, 'detuning': 1,
'phase': 0, 'kvec': (0, 0, 0), 'label': '(0,1)'}, (1, 0): {'gamma_
↪transition': 37.829349995476726,
'label': '(1,0)', 'gamma_transit': 0.41172855461658464}}
```

Here we see implicitly calling this overloaded function through *`add_couplings()`*.

```
>>> cell = rq.Cell("Rb85", *rq.D2_states("Rb85"))
>>> c = {"states":(0,1), "detuning":1, "rabi_frequency":2}
>>> cell.add_couplings(c)
>>> print(dict(cell.couplings.edges))
{(0, 0): {'gamma_transit': 0.41172855461658464, 'label': '(0,0)'}, (0, 1):
↪{'rabi_frequency': 2, 'detuning': 1,
'phase': 0, 'kvec': (0, 0, 0), 'label': '(0,1)'}, (1, 0): {'gamma_
↪transition': 37.829349995476726,
'label': '(1,0)', 'gamma_transit': 0.41172855461658464}}
```

`e_field` can be specified in stead of `rabi_frequency`, but a `rabi_frequency` will still be added to the system based on the `e_field`, rather than `e_field` directly.

```
>>> cell = rq.Cell("Rb85", *rq.D2_states("Rb85"))
>>> c = {"states":(0,1), "detuning":1, "e_field":6}
>>> cell.add_couplings(c)
>>> print(cell.couplings.edges(data='e_field'))
>>> print(cell.couplings.edges(data='rabi_frequency'))
[(0, 0, None), (0, 1, None), (1, 0, None)]
[(0, 0, None), (0, 1, -1.172912676105507), (1, 0, None)]
```

As can `beam_power` and `beam_waist`, with similar behavior regarding how information is stored.

```
>>> cell = rq.Cell("Rb85", *rq.D2_states("Rb85"))
>>> c = {"states":(0,1), "detuning":1, "beam_power":1, "beam_waist":1}
>>> cell.add_couplings(c)
>>> print(cell.couplings.edges(data='beam_power'))
>>> print(cell.couplings.edges(data='rabi_frequency'))
[(0, 0, None), (0, 1, None), (1, 0, None)]
[(0, 0, None), (0, 1, 4.28138982322625), (1, 0, None)]
```

`add_couplings`(**couplings: Dict*, *no_rwa_warning: Optional[bool] = False*) → None

Add any number of couplings between pairs of states.

Acts as an alternative to calling *`add_coupling()`* individually for each pair of states. Can be used interchangably up to preference, and all of keyword *`add_coupling()`* are supported dictionary keys for dictionaries passed to this function.

Parameters

- **`couplings`** (*`tuple of dicts`*) – Any number of dictionaries, each specifying the parameters of a single field coupling 2 states. For more details on the keys of each dictionry see the arguments for *`add_coupling()`*. Equivalent to passing each dictiories keys and values to *`add_coupling()`* individually.
- **`no_rwa_warning`** (*`bool`*) – Whether to supress the warning that appears when adding a coupling with a sufficiently transition frequency that solving the system may

take prohibitively long without the rotating wave approximation.

Raises

`ValueError` – If the `states` parameter is missing.

Examples

```
>>> s = rq.Sensor(3)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":2}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":4}
>>> s.add_couplings(blue, red)
>>> print(s.couplings.edges(data=True))
[(0, 1, {'rabi_frequency': 1, 'detuning': 2, 'phase': 0, 'kvec': (0, 0, 0)}
↪), (1, 2, {'rabi_frequency': 3, 'detuning': 4, 'phase': 0, 'kvec': (0, 0,
↪ 0)})]
```

add_decoherence(*states: Tuple[int, ...], gamma: Union[float, List[float], ndarray], label: Optional[str] = None*) → None

Add decoherent coupling to the graph between two states.

If `gamma` is list-like, the sensor will scan over the values, solving the system for each different gamma.

Parameters

- **states** (*tuple of ints*) – Length-2 tuple of integers corresponding to the two states. The first value is the number of state out of which population decays, and the second is the number of the state into which population decays.
- **gamma** (*float or sequence*) – The decay rate, in Mrad/s.
- **label** (*str or None, optional*) – Optional label for the decay. If `None`, decay will be stored on the graph edge as `"gamma"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`.

Notes

Note: Adding a decoherece with a particular label (including `None`) will override an existing decoherent transition with that label.

Examples

s = rq.Sensor(3) >>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1) >>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1) >>> s.add_decoherence((2,0), 0.1, label=”misc”) >>> print(s.decoherence_matrix()) [[0. 0. 0.] [0. 0. 0.] [0.1 0. 0.]]

Decoherence values can also be scanned. Here decoherece from states 2->0 is scanned between 0 and 0.5 for 11 values. We can also see how the Hamiltonian shape accounts for this to allow for clean broadcasting, indicating that the hamiltonian is identical accross all decoherence values.

```
>>> s = rq.Sensor(3)
>>> gamma = np.linspace(0,0.5,11)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_decoherence((2,0), gamma)
>>> print(s.decoherence_matrix().shape)
(11, 3, 3)
>>> print(s.get_hamiltonian().shape)
(11, 3, 3)
```

add_self_broadening(*node: int*, *gamma: Union[float, List[float], ndarray]*, *label: Optional[str] = 'self'*) → None

Specify self-broadening (such as collisional broadening) of a level.

Equivalent to calling *add_decoherence()* and specifying both states to be the same, with the "self" label. For more complicated systems, it may be useful to further specify the source of self-broadening as, for example, "collisional" for easier bookkeeping and to ensure no values are overwritten.

Parameters

- **node** (*int*) – The integer number of the state node to which the broadening will be added. The integer corresponds to the state's position in the graph.
- **gamma** (*float or sequence*) – The broadening width to be added in Mrad/s.
- **label** (*str, optional*) – Optional label for the state. If `None`, decay will be stored on the graph edge as `"gamma"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`

Notes

Note: Just as with the *add_decoherence()* function, adding a decoherence value with a label that already exists will overwrite an existing decoherent transition with that label. The "self" label is applied to this function automatically to help avoid an unwanted overwrite.

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_self_broadening(1, 0.1)
>>> print(s.couplings.edges(data=True))
>>> print(s.decoherence_matrix())
[(1, 1, {'gamma_self': 0.1, 'label': '(1,1)'})]
[[0.  0.  0. ]
[0.  0.1 0. ]
[0.  0.  0. ]]
```

add_states()

Deprecated.

Use "atomic_state" keyword argument of the constructor instead.

add_transit_broadening(*gamma_transit: Union[float, List[float], ndarray]*, *repop: Dict[int, float] = {0: 1.0}*) → None

Adds transit broadening by adding a decoherence from each node to ground.

For each graph node n, adds a decoherent transition from n the specified state (0 by default) using the *add_decoherence()* method with the `"transit"` label. See *add_decoherence()* for more details on labeling.

If an array of transit values are provided, they will be automatically zipped together into a single scanning element.

Parameters

- **gamma_transit** (*float or sequence*) – The transit broadening rate in Mrad/s.
- **repop** (*dict, optional*) – Dictionary of states for transit to repopulate in to. The keys represent the state labels. The values represent the fractional amount that goes to

that state. If the sum of values does not equal 1, population will not be conserved. Default is to repopulate everything into state 0.

Warns

- **If the values of the `repop` parameter do not sum to 1, thus meaning**
- **population will not be conserved.**

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_transit_broadening(0.1)
>>> print(s.couplings.edges(data=True))
>>> print(s.decoherence_matrix())
[(0, 0, {'gamma_transit': 0.1}), (1, 0, {'gamma_transit': 0.1}), (2, 0, {
↪'gamma_transit': 0.1})]
[[0.1 0.  0. ]
[0.1 0.  0. ]
[0.1 0.  0. ]]
```

axis_labels(*collapse: bool = True*) → List[str]

Get a list of axis labels for stacked hamiltonians.

The axes of a hamiltonian stack are defined as the axes preceding the usual hamiltonian, which are always the last 2. These axes only exist if one of the parametes used to define a Hamiltonian are lists.

Be default, labels which have been zipped using *zip_parameters()* will be combined into a single label, as this is how *get_hamiltonian()* treats these axes.

Returns
Strings corresponding to the label of each axis on a stack of multiple hamiltonians.

Return type
list of str

Examples

There are no preceding axes if there are no list-like parameters.

```
>>> s = rq.Sensor(3)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":2}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":4}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape())
>>> print(s.axis_labels())
(3,3)
[]
```

Adding list-like parameters expands the hamiltonian

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-10, 10, 11)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":det, "label":
↪"blue"}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape)
>>> print(s.axis_labels())
(11, 11, 3, 3)
['blue_detuning', '(1, 2)_detuning']
```

Zipping parameters combines their corresponding labels, since their Hamiltonians now lie on a single axis of the stack. Here the axis of length 7 (axis 0) corresponds to the rabi frequencies and the axis of shape 11 (axis 1) corresponds to the zipped detunings

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=np.arange(11), rabi_frequency=np.
↪linspace(-3, 3, 7))
>>> s.add_coupling(states=(1,2), detuning=0.5*np.arange(11), rabi_
↪frequency=1)
>>> s.zip_parameters("(0,1)_detuning", "(1,2)_detuning")
>>> print(s.get_hamiltonian().shape)
>>> print(s.axis_labels())
(7, 11, 3, 3)
['(0,1)_rabi_frequency', '(0,1)_detuning|(1,2)_detuning']
```

basis() → ndarray

Generate basis labels of density matrix components.

The basis corresponds to the elements in the solution. This is not the complex basis of the sensor class, but rather the real basis of a solution after calling one of `rydiqule`'s solvers. This means that the ground state population has been removed and it has been transformed to the real basis.

Returns
Array of string labels corresponding to the solving basis. Is a 1-D array of length `n**2-1`.

Return type
numpy.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> print(s.basis())
['01_real' '02_real' '01_imag' '11_real' '12_real' '02_imag' '12_imag'
'22_real']
```

collapse_dims(*stack: ndarray*) → ndarray

Collapse dimensions of a stack accoring to the `_zipped_parameters` attribute.

Designed primarily for internal use, but can be called externally or in custom overloads of *`get_hamiltonian()`*. Effectively performs a `numpy.einsum` operation to replicate `numpy.diag` along the appropriate dimensions.

Parameters
stack (*`numpy.ndarray`*) – Array with shape matching uncollapsed system hamiltonians.

Returns
Input array with appropriate dimensions removed via a numpy.diag-like operation.

Return type
numpy.ndarray

couplings_with(**keys: str, method: Literal['all', 'any', 'not any'] = 'all'*) → Dict[Tuple[int, ...], Dict]

Returns a version of self.couplings with only the keys specified.

Can be specified with a several criteria, including all, none, or any of the keys specified.

Parameters

- **str)** (*keys (tuple of*) – parameter names for a state. See *`add_coupling()`* for which names are valid for a Sensor object.

- **method** (*{'all','any', 'not any'}*) – Method to see if a given field matches the keys given. Choosing "all" will return couplings which have keys matching all of the values provided in the keys argument, while coosing "any", will return all couplings with keys matching at least one of the values specified by keys. For example, `sensor.couplings_with("rabi_frequency")` returns a dictionary of all couplings for which a rabi_frequency was specified. `sensor.couplings_with("rabi_frequency", "detuning", method="all")` returns all couplings for which both rabi_frequency and detuning are specified. 'sensor.couplings_with("rabi_frequency", "detuning", method="any")` returns all couplings for which either rabi_frequency or detuning are specified. Defaults to "all".

Returns
A copy of the `sensor.couplings` dictionary with only couplings containing the specified parameter keys.

Return type
dict

Examples

Can be used, for example, to return couplings in the roating wave approximation.

```
>>> s = rq.Sensor(3)
>>> sinusoid = lambda t: 0 if t<1 else sin(100*t)
>>> f2 = {"states": (0,1), "detuning": 1, "rabi_frequency":2}
>>> f1 = {"states": (1,2), "transition_frequency":100, "rabi_frequency":1,
↪"time_dependence": sinusoid}
>>> s.add_couplings(f1, f2)
>>> gamma = np.array([[.2,0,0],
...                   [.1,0,0],
...                   [0.05,0,0]])
>>> s.set_gamma_matrix(gamma)
>>> print(s.couplings_with("detuning"))
{(0, 1): {'rabi_frequency': 2, 'detuning': 1, 'phase': 0, 'kvec': (0, 0,␣
↪0), 'no_rwa_warning': False, 'label': '(0,1)'}}
```

decoherence_matrix() → ndarray

Get the decoherence matrix for a system.

This overload differs from `decoherence_matrix()` by including state lifetimes and decay rates calculated from arc rydberg without any explicit definition of decoherence terms. In other words, it ensures that the calculated decoherence terms match what is expected for a particular real-world atom.

Returns
The decoherence matrix stack of the system.

Return type
numpy.ndarray

eta: **float** = **None**

Noise density prefactor, in units of root(Hz). Must be specified when using `Sensor`. Automatically calculated when using *`Cell`*.

get_couplings() → Dict[Tuple[int, ...], Dict]

Returns the couplings of the system as a dictionary

Deprecating in favor of calling the couplings.edges attribute directly.

Returns
A dictionary of key-value pairs with the keys corresponding to levels of transition, and the values being dictionaries of coupling attributes.

Return type
dict

get_doppler_shifts() → ndarray

Returns the Hamiltonian with only detunings set to the kvector values for each spatial dimension.

Determining if a float should be treated as zero is done using `numpy.isclose`, which has default absolute tolerance of `1e-08`.

Returns
Array of shape (used_spatial_dim,n,n), Hamiltonians with only the doppler shifts present along each non-zero spatial dimension specified by the fields' "kvec" parameter.

Return type
numpy.ndarray

get_hamiltonian() → ndarray

Creates the Hamiltonians from the couplings defined by the fields.

They will only be the steady state hamiltonians, i.e. will only contain terms which do not vary with time. Implicitly creates hamiltonians in "stacks" by creating a grid of all supported coupling parameters which are lists. This grid of parameters will not contain rabi-frequency parameters which vary with time and are defined as list-like. Rather, the associated axis will be of length 1, with the scanning over this value handled by the `get_time_couplings()` function.

For m list-like parameters x1,x2,…,xm with shapes N1,N2,…,Nm, and basis size n, the output will be shape `(N1,N2,...,Nm, n, n)`. The dimensions N1,N2,…Nm are labeled by the output of *`axis_labels()`*.

If any parameters have been zipped with the `_zip_parameters()` method, those parameters will share an axis in the final hamiltonian stack. In this case, if axis N1 and N2 above are the same shape and zipped, the final Hamiltonian will be of shape `(N1,...,Nm, n, n)`

See rydiqule's conventions for matrix stacking for more details.

Returns
The complex hamiltonian stack for the sensor.

Return type
np.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":det}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(red, blue)
>>> print(s.get_hamiltonian().shape)
(11, 11, 3, 3)
```

Time dependent couplings are handled separately. The axis that contains array-like parameters with time dependence is length 1 in the steady-state Hamiltonian.

```
>>> s = rq.Sensor(3)
>>> rabi = np.linspace(-1,1,11)
>>> step = lambda t: 0 if t<1 else 1
>>> blue = {"states":(0,1), "rabi_frequency":rabi, "detuning":1}
>>> red = {"states":(1,2), "rabi_frequency":rabi, "detuning":0, 'time_
↪dependence': step}
>>> s.add_couplings(red, blue)
>>> print(s.get_hamiltonian().shape)
(11, 1, 3, 3)
```

Zipping parameters means they share an axis in the Hamiltonian.

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=np.arange(11), rabi_frequency=2)
>>> s.add_coupling(states=(1,2), detuning=0.5*np.arange(11), rabi_
↪frequency=1)
>>> s.zip_parameters("(0,1)_detuning", "(1,2)_detuning")
>>> H = s.get_hamiltonian()
>>> print(H.shape)
(11, 3, 3)
```

get_hamiltonian_diagonal(*values: dict*, *no_stack: bool = False*) → ndarray

Apply addition and subtraction logic corresponding to the direction of the couplings.

For a given state `n`, the path from ground will be traced to `n`. For each edge along this path, values will be added where the path direction and coupling direction match, and subtracting values where they do not. The sum of all such values along the path is the `n` th term in the output array.

Primarily for internal functions which help generate hamiltonians. Most commonly used to calculate total detunings for ranges of couplings under the RWA

Parameters

- **values** (*dict*) – Key-value pairs where the keys correspond to transitions (agnostic to ordering of states) and values corresponding to the values to which the logic will be applied.
- **no_stack** (*bool, optional*) – Whether to ignore variable parameters in the system and use only basic math operations rather than reshape the output. Typically only `True` for calculating doppler shifts.

Returns

The digonal of the hamiltonian of the system of shape `(*l,n)`, where `l` is the shape of the hamiltonian stack for the sensor.

Return type

numpy.ndarray

get_parameter_mesh(*sparse: bool = True*, ***kwargs*) → Tuple[ndarray, ...]

Returns the parameter mesh of the sensor.

The parameter mesh is the flattened grid of variable parameters in all the couplings of a sensor. Wraps `numpy.meshgrid` by passing arguments as the variable parameters in a sensor. The `indexing` argument is always `"ij"` for matrix indexing. For full documention of all arguments, see `numpy.meshgrid` documentation. Argument documentation copied from `numpy.meshgrid`.

Parameters

- **sparse** (*bool, optional*) – If `True`, the shape of the returned coordinate array for dimension i is reduced from `(N1, ..., Ni, ... Nn)` to `(1, ..., 1, Ni, 1, ..., 1)`. These sparse coordinate grids are intended to be used in a way consistent with numpy's broadcasting conventions. When all coordinates are used in an expression, broadcasting still leads to a fully-dimensonal result array. Recommended to keep as `True`. Default is `True`.
- **kwargs** (*dict, optional*) – Additional keyword arguments to pass to `numpy.meshgrid`. All arguments are passed directly to `numpy.meshgrid` with the exception of `indexing`, which is fixed to `"ij"` and cannot be specified.

Examples

```
>>> s = rq.Sensor(3)
>>> rabi1 = np.linspace(-1,1,11)
>>> rabi2 = np.linspace(-2,2,21)
>>> s.add_coupling(states=(0,1), rabi_frequency=rabi1, detuning=1)
>>> s.add_coupling(states=(1,2), rabi_frequency=rabi2, detuning=1)
>>> for p in s.get_parameter_mesh():
...     print(p.shape)
(11, 1)
(1, 21)
```

get_time_couplings() → Tuple[ndarray, ndarray]

Returns the list of matrices of all couplings in the system defined with a `time_dependence` key.

The ouput will be two lists of matricies representing which terms of the hamiltonian are dependent on each time-dependent coupling. The lists will be of length M and shape `(*l_time, n, n)`, where M is the number of time-dependent couplings, `l_time` is time-dependent stack shape (possibly all ones), and `n` is the basis size. Each matrix will have terms equal to the rabi frequency (or half the rabi frequency under RWA) in positions that correspond to the associated transition. For example, in the case where there is a `time_dependence` function defined for the `(2,3)` transition with a rabi frequency of 1, the associated time coupling matrix will be all zeros, with a 1 in the `(2,3)` and `(3,2)` positions.

Typically, this function is called internally and multiplied by the output of the *`get_time_dependence()`* function.

Returns

- *list of numpy.ndarray* – The list of M `(*l,n,n)` matrices representing the real-valued time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.
- *list of numpy.ndarray* – The list of M `(*l,n,n)` matrices representing the imaginary-valued time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.

Examples

```
>>> s = rq.Sensor(3)
>>> step = lambda t: 0 if t<1 else 1
>>> wave = lambda t: np.sin(2000*np.pi*t)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1,
↪"time_dependence":wave}
>>> f2 = {"states": (1,2), "transition_frequency":10, "rabi_frequency": 2,
↪"time_dependence":step}
>>> s.add_couplings(f1, f2)
>>> time_hams, time_hams_i = s.get_time_couplings()
>>> for H in time_hams:
...     print(H)
[[0.+0.j 1.+0.j 0.+0.j]
[1.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j]]
[[0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 2.+0.j]
[0.+0.j 2.+0.j 0.+0.j]]
```

To handle stacking across the steady-state and time hamiltonians, the dimensions are matched in a way that broadcasting works in a numpy-friendly way

```
>>> s = rq.Sensor(3)
>>> rabi = np.linspace(-1,1,11)
>>> step = lambda t: 0 if t<1 else 1
>>> blue = {"states":(0,1), "rabi_frequency":rabi, "detuning":1}
>>> red = {"states":(1,2), "rabi_frequency":rabi, "detuning":0, 'time_
↪dependence': step}
>>> s.add_couplings(red, blue)
>>> time_hams, time_hams_i = s.get_time_couplings()
>>> print(s.get_hamiltonian().shape)
>>> print(time_hams[0].shape)
>>> print(time_hams_i[0].shape)
(1, 11, 3, 3)
(11, 1, 3, 3)
(11, 1, 3, 3)
```

get_time_dependence() → List[Callable[[float], float]]

Function which returns a list of the `time_dependence` functions.

The list is returned with in the order that matches with the time hamiltonians from *`get_time_couplings()`* such that the ith element of of the return of this functions corresponds with the ith Hamiltonian terms returned by that function.

Returns
List of scalar functions, representing all couplings specified with a `time_dependence`.

Return type
list

Examples

```
>>> s = rq.Sensor(3)
>>> step = lambda t: 0 if t<1 else 1
>>> wave = lambda t: np.sin(2000*np.pi*t)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1,
↪"time_dependence":wave}
>>> f2 = {"states": (1,2), "transition_frequency":10, "rabi_frequency": 2,
↪"time_dependence":step}
>>> s.add_couplings(f1, f2)
>>> print(s.get_time_dependence())
[<function <lambda> at 0x7fb310edd9d0>, <function <lambda> at␣
↪0x7fb37c0c81f0>]
```

get_time_hamiltonians() → Tuple[ndarray, ndarray, ndarray]

Get the hamiltonians for the time solver.

Get both the steady state hamiltonian (as returned by *`get_hamiltonian()`*) and the time_dependent hamiltonians (as returned by *`get_time_couplings()`*). The time dependent hamiltonians give 2 terms, the hamiltonian corresponding to the real part of the coupling and the hamiltonian corresponding to the imaginary part.

Returns

- **hamiltonian_base** (*np.ndarray*) – The `(*l,n,n)` shape base hamiltonian of the system containing all elements that do not depend on time, where `n` is the basis size of the sensor.
- **dipole_matrix_real** (*np.ndarray*) – The `(M,n,n)` shape array of matrices representing the real time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.

- **dipole_matrix_imag** (*nd.ndarray*) – The `(M,n,n)` shape array of matrices representing the imaginary time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.

Examples

```
>>> s = rq.Sensor(2)
>>> step = lambda t: 0. if t<1 else 1.
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1, time_
↪dependence=step)
>>> H_base, H_time_real, H_time_imaginary = s.get_time_hamiltonians()
>>> print(H_base)
>>> print(H_time_real)
>>> print(H_time_imaginary)
[[0.+0.j 0.+0.j]
[0.+0.j 1.+0.j]]
[array([[0. +0.j, 0.5+0.j],
    [0.5+0.j, 0. +0.j]])]
[array([[0.+0.j , 0.+0.5j],
    [0.-0.5j, 0.+0.j ]])]
```

`get_transition_frequencies``()` → ndarray

Gets an array of the diagonal elements of the Hamiltonian from the field detunings.

Wraps the *`get_hamiltonian_diagonal()`* function using both transition frequencies and detunings. Primarily for internal use.

Returns

N-D array of the hamiltonian diagonal. For an n-level system with stack shape `*l`, will be shape `(*l, n)`

Return type

numpy.ndarray

`get_value_dictionary``(`*key: str*`)` → dict

Get subset of dictionary coupling parameters.

Return a dictionary of key value pairs where the keys are couplings added to the system and the values are the value of the parameter specified by key. Produces an output that can be passed directly to *`get_hamiltonian_diagonal()`*. Only couplings whose parameter dictionaries contain “key” will be in the returned dictionary.

Parameters

key (`str`) – String value of the parameter name to build the dictionary. For example, `get_value_dictionary("detuning")` will return a dictionary with keys corresponding to transitions and values corresponding to detuning for each transition which has a detuning.

Returns

Coupling dictionary with couplings as keys and corresponding values set by input key.

Return type

dict

Examples

```
>>> s = rq.Sensor(4)
>>> f1 = {"states": (0,1), "detuning": 2, "rabi_frequency": 1}
>>> f2 = {"states": (1,2), "detuning": 3, "rabi_frequency": 2}
>>> f3 = {"states": (2,3), "rabi_frequency": 3, "transition_frequency": 3}
>>> s.add_couplings(f1, f2, f3)
>>> print(s.get_value_dictionary("detuning"))
{(0,1): 2, (1,2): 3}
```

kappa: float = None

Differential prefactor, in units of (Mrad/s)/m. Must be specified when using `Sensor`. Automatically calculated when using *`Cell`*.

level_ordering() → List[int]

Return a list of the integer numbers of each state (*not* the quantum numbers) in descending order by energy order.

All energies are calculated with respect to the ground state energy, which is defined as 0. Ground state is determined by the first state added to the system (state 0). Thus, state 0 will always be last in the list.

Returns
The level numbers of the states in order of decending energy relative to the ground state (state 0).

Return type
list[int]

Examples

For the following example, states are added to the cell in ascending energy order, so the return reflects that, with the highest-energy state first.

```
>>> state1 = [50, 2, 2.5, 2.5]
>>> state2 = [51, 2, 2.5, 2.5]
>>> cell = rq.Cell("Rb85", *rq.D2_states("Rb85"), state1, state2) #uses␣
↪the D2 line of Rb85
>>> print(cell.states_list())
>>> print(cell.level_ordering())
[[5, 0, 0.5, 0.5], [5, 1, 1.5, 0.5], [50, 2, 2.5, 2.5], [51, 2, 2.5, 2.5]]
[3, 2, 1, 0]
```

If we add `state1` and `state2` in the opposite order (thus switching their positions in the states list), the level ordering will change since `state2` is still a higher energy.

```
>>> state1 = [50, 2, 2.5, 2.5]
>>> state2 = [51, 2, 2.5, 2.5]
>>> cell = rq.Cell("Rb85", *rq.D2_states("Rb85"), state2, state1) #uses␣
↪the D2 line of Rb85
>>> print(cell.states_list())
>>> print(cell.level_ordering())
[[5, 0, 0.5, 0.5], [5, 1, 1.5, 0.5], [51, 2, 2.5, 2.5], [50, 2, 2.5, 2.5]]
[2, 3, 1, 0]
```

set_gamma_matrix(*gamma_matrix: ndarray*) → None

Set the decoherence matrix for the system.

Works by first removing all existing decoherent data from graph edges, then individually adding all nonzero terms of a provided gamma matrix to the corresponding graph edges. Can be used to set all decoherence attributes to edges simultaneously, but *`add_decoherence()`* is preferred.

Unlike *add_decoherence()*, does not support scanning multiple decoherence values, rather should be used to set the decoherences of the system to individual static values.

Parameters

gamma_matrix (*numpy.ndarray*) – Array of shape (basis_size, basis_size). Element (i,j) describes the decoherence rate, in Mrad/s, from state i to state j.

Raises

- **TypeError** – If gamma_matrix is not a numpy array.
- **ValueError** – If gamma_matrix is not a square matrix of the appropriate size
- **ValueError** – If the shape of gamma_matrix is not compatible with self.basis_size.

Examples

```
>>> s = rq.Sensor(2)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1}
>>> s.add_couplings(f1)
>>> gamma = np.array([[.1,0],[.1,0]])
>>> s.set_gamma_matrix(gamma)
>>> print(s.decoherence_matrix())
[[0.1 0. ]
 [0.1 0. ]]
```

spatial_dim() → int

Returns the number of spatial dimensions doppler averaging will occur over.

Determining if a float should be treated as zero is done using numpy.isclose, which has default absolute tolerance of 1e-08.

Returns

Number of dimensions, between 0 and 3, where 0 means no doppler averaging kvectors have been specified or are too small to be calculates.

Return type

int

Examples

No spatial dimesions specified

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_freqency=1)
>>> print(s.spatial_dim())
0
```

One spatial dimension specified

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_freqency=1, kvec=(0,0,1))
>>> print(s.spatial_dim())
1
```

Multiple spatial dimensions can exist in a single coupling or across multiple couplings

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_freqency=1, kvec=(1,0,1))
>>> print(s.spatial_dim())
2
```

```
>>> s = rq.Sensor(3)
>>> s.add_coupling((0,1), detuning = 1, rabi_freqency=1, kvec=(1,0,1))
>>> s.add_coupling((1,2), detuning = 2, rabi_freqency=2, kvec=(0,1,0))
>>> print(s.spatial_dim())
3
```

states_list() → List[Sequence]

Returns a list of quantum numbers for all states in the cell.

States position in the list will be in the order they were added, and correspond to the density matrix values numbered with the same index.

Returns
List of quantum states of the form [n, l, j, m] that are stored on graph nodes in the cell.

Return type
list[list]

Examples

```
>>> state1 = [50, 2, 2.5, 2.5] # states are written with quantum numbers␣
↪[n, l, j, m]
>>> state2 = [51, 2, 2.5, 2.5]
>>> cell = rq.Cell("Rb85", *rq.D2_states(5), state1, state2) #D2 states␣
↪gets the states for the Rubidium 85 D2 line
>>> print(cell.states_list())
[[5, 0, 0.5, 0.5], [5, 1, 1.5, 0.5], [50, 2, 2.5, 2.5], [51, 2, 2.5, 2.5]]
```

unzip_parameters(**labels: str*, ***extra_kwargs*)

Remove a set of zipped parameters from the internal zipped_parameters list.

If an element of the internal `_zipped_parameters` array matches ALL labels provided, removes it from the internal `zipped_parameters` method. If no such element is in `_zipped_parameters`, does nothing.

Parameters

- **labels** (*str*) – Any number of string labels matching ALL the names of a set of parameters that has already been zipped.
- ****extra_kwargs** (*dict*) –

  Optional Extra keyword-only arguments. Supported argument is

  – verbose (str): Whether to print if there are no parameters matching `labels` in `_zipped_parameters`. Defaults to `True`.

Notes

Note: This function should always be used rather than modifying the `_zipped_parameters` attribute directly.

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label=
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters("probe_detuning", "(1,2)_detuning")
>>> print(s._zipped_parameters) #NOT modifying directly
>>> s.unzip_parameters('(1,2)_detuning', 'probe_detuning')
>>> print(s._zipped_parameters) #NOT modifying directly
[['(1,2)_detuning', 'probe_detuning']]
[]
```

If the labels provided are not a match, a message is printed and nothing is altered.

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label=
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters("probe_detuning", "(1,2)_detuning")
>>> print(s._zipped_parameters) #NOT modifying directly
>>> s.unzip_parameters('green_detuning', 'probe_detuning')
>>> print(s._zipped_parameters) #NOT modifying directly
[['(1,2)_detuning', 'probe_detuning']]
No zipped parameters matching ['green_detuning', 'probe_detuning']
[['(1,2)_detuning', 'probe_detuning']]
```

`variable_parameters`(*apply_mesh=False*)

Property to retrieve the values of parameters that were stored on the graph as arrays.

Values are returned as a list of tuples in the standard order of pythons default sorting, applied first to the tuple indicating states and then to the key of the parameter itself. This means that couplings are sorted first by lower state, then by upper state, then alphabetically by the name of the parameter.

Returns

A list of tuples corresponding to the parameters of the systems that are variable (i.e. stored as an array). They are ordered accordning to states, then according to variable name. Tuple entries of the list take the form `(states, param_name, value)`

Return type

list of tuples

Examples

```
>>> s = rq.Sensor(3)
>>> vals = np.linspace(-1,2,3)
>>> s.add_coupling(states=(1,2), rabi_frequency=vals, detuning=1)
>>> s.add_coupling(states=(0,1), rabi_frequency=vals, detuning=vals)
>>> for states, key, value in s.variable_parameters():
...     print(f"{states}: {key}={value}")
(0, 1): detuning=[-1.   0.5  2. ]
(0, 1): rabi_frequency=[-1.   0.5  2. ]
(1, 2): rabi_frequency=[-1.   0.5  2. ]
```

zip_parameters(**labels: str*) → None

Define 2 scannable parameters as "zipped" so they are scanned in parallel.

Zipped parameters will share an axis when quantities relevant to the equations of motion, such as the `gamma_matrix` and `hamiltonian` are generated. Note that calling this function does not affect internal quanties directly, but adds their labels together in the internal `self._zipped_parameters` list, and they are zipped at calculation time for `hamiltonian` and `decoherence_matrix`.

Parameters

labels (*str*) – Parameter labels to scan together. Parameter labels are strings of the form `"<coupling_label>_<parameter_name>"`, such as `"(0, 1)_detuning"`. Must be at least 2 labels to zip.

Raises

- **ValueError** – If fewer than 2 labels are provided.
- **ValueError** – If any of the 2 labels are the same.
- **ValueError** – If any elements of `labels` are not labels of couplings in the sensor.
- **ValueError** – If any of the parameters specified by labels are already zipped.
- **ValueError** – If any of the parameters specified are not list-like.
- **ValueError** – If all list-like parameters are not the same length.

Notes

Note: This function should be called last after all Sensor couplings and dephasings have been added. Changing a coupling that has already been zipped removes it from the `self.zipped_parameters` list.

Note: Modifying the `Sensor.zipped_parameters` attribute directly can break some functionality and should be avoided. Use this function or *`unzip_parameters()`* instead.

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label=
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters("probe_detuning", "(1,2)_detuning")
>>> print(s._zipped_parameters) #NOT modifying directly
[['(1,2)_detuning', 'probe_detuning']]
```

6.1.3 rydiqule.doppler_utils

Utilities for implementing Doppler averaging

Functions

apply_doppler_weights(sols, velocities, volumes)	Calculates and applies the weight for each doppler class given unweighted solutions to doppler-shifted equations.
doppler_classes([method])	Defines which velocity classes to sample for doppler averaging.
doppler_mesh(doppler_velocities, spatial_dim)	Creates meshgrids of evaluation points and point "volumes" for doppler averaging.
gaussian3d(Vs)	Evaluate a multi-dimensional gaussian, with sigma=1, for the given detunings.
generate_doppler_shift_eom(doppler_hamiltonians)	doppler_mesh_method Generates the EOMs for the supplied doppler shifts.
get_doppler_equations(base_eoms, ...)	Returns the equations for each slice of the doppler profile.

rydiqule.doppler_utils.apply_doppler_weights

`rydiqule.doppler_utils.`**`apply_doppler_weights`**(*sols: ndarray*, *velocities: ndarray*, *volumes: ndarray*) → ndarray

Calculates and applies the weight for each doppler class given unweighted solutions to doppler-shifted equations.

Works for both time-domain and stead-states solutions.

Parameters

- **`sols`** (*`numpy.ndarray`*) – The array of solutions over which to calculate weights.
- **`velocities`** (*`numpy.ndarray`*) – Array of shape `(n_dim, *n_dop)` where n_dim is the number of dimensions over which doppler shifts are being considered and `*n_dop` is a number of axes equal to n_dim with length equal to the number of doppler velocity classes which are being considered. The values correspond the velocity class in units of sigma.
- **`volumes`** (*`numpy.ndarray`*) – Array of shape equal to `velocities`. The values correspond to the spacings between doppler classes on each axis.

Returns

The weighted solution array of shape equal to that of `sols`.

Return type

numpy.ndarray

Raises

`ValueError` – If the shapes of `velocities` and `volumes` do not match.

rydiqule.doppler_utils.doppler_classes

`rydiqule.doppler_utils.`**`doppler_classes`**(*method: Optional[Union[*UniformMethod, SplitMethod, DirectMethod*]] = None*) → ndarray

Defines which velocity classes to sample for doppler averaging.

These are defined in units of the gaussian width sigma of the Maxwell-Boltzmann distribution.

Note: To avoid issues, optical detunings should not leave densely sampled velocity classes. To avoid artifacts, the density of points should provide >~10 points over the narrowest absorptive feature. The default is a decent first guess, but for many problems the sampling mesh should be adjusted.

Parameters

`method` (*`dict`*) – Specifies method to use and any control parameters. Must contain the key `"method"` with one of the following options. Each method has suboptions that also need to be specified. Valid options are:

- `"uniform"`: Defines a uniformly spaced, dense grid. Configuration parameters include:
  - `"width_doppler"`: Float that specifies one-sided width of gaussian distributionto average over, in units of sigma. Defaults to 2.0.
  - `"n_uniform"`: Int that specifies how many points to use. Defaults to 1601.
- `"isopop"`: Defines a grid with uniform population in each interval. This method highly emphasises physics happening near the 0 velocity class. If stuff is happening for non-zero velocity classes, it is likely to alias it unless `n_isopop` is large. See Ref[1] for details. Configuration parameters include:
  - `"n_isopop"`: Int that specifies how many points to use. Defaults to 400.
- `"split"`: Defines a grid with a dense central spacing and wide spacing wings. This method provides a decent compromise between uniform and isopop. It uses fewer points than uniform, but also works well for non-zero velocity class models (like Autler-Townes splittings). This is the default meshing method. Configuration parameters include:
  - `"width_doppler"`: Float that specifies one-sided width of coarse grided portion of the gaussian distribution. Units are in sigma. Defaults to 2.0.
  - `"width_coherent"`: Float that specifies one-sided width of fine grided portion of gaussian distribution. Units are in sigma. Defaults to 0.4.
  - `"n_doppler"`: Int that specifies how many points to use for the coarse grid. Note that points of the coarse grid that fall within the fine grid are dropped. Default is 201.
  - `"n_coherent"`: Int that specifies how many points to use for the fine grid. Default is 401.

  Note: Note that for the "split" method, a union of 2 samplings is taken, so the number of total points will not necessary be equal to the sum of `"n_coherent"` and `"n_doppler"`.

- `"direct"`: Use the supplied 1-D numpy array to build the mesh.

[1] Andrew P. Rotunno, et. al. Inverse Transform Sampling for Efficient Doppler-Averaged Spectroscopy Simulation, arXiv:2304.12468 (2023) http://arxiv.org/abs/2304.12468

- "doppler_velocities": Mandatory parameter that holds the 1-D numpy array to use when building the mesh grids.

Returns
1-D array of velocities to be sampled.

Return type
numpy.ndarray

Examples

The defaults will sample more densely near the center of the distribution, (the "split" method) with a total of 561 classes.

```
>>> classes = rq.doppler_classes() #use the default values
>>> print(classes.shape)
(561,)
```

Specifying "uniform" with no additional arguments produces 1601 evenly spaced classes by default.

```
>>> m = {"method":"uniform"}
>>> classes = rq.doppler_classes(method=m)
>>> print(classes.shape)
(1601,)
```

Further specifying the number of points allows more dense or sparse sampling of the velocity distribution.

```
>>> m = {"method":"uniform", "n_uniform":801}
>>> classes = rq.doppler_classes(method=m)
>>> print(classes.shape)
(801,)
```

The "split" method also has further specifications

```
>>> m = {"method":"split", "n_coherent":301, "n_doppler":501}
>>> classes = rq.doppler_classes(method=m)
>>> print(classes.shape)
(701,)
```

References

rydiqule.doppler_utils.doppler_mesh

`rydiqule.doppler_utils.`**`doppler_mesh`**(*doppler_velocities: ndarray, spatial_dim: int*) → Tuple[ndarray, ndarray]

Creates meshgrids of evaluation points and point "volumes" for doppler averaging.

Parameters

- **`dop_velocities`** (*`numpy.ndarray`*) – A 1-D array of velocities to evaluate over. These should be normalized to the most probable velocity used by *`gaussian3d()`*.
- **`spatial_dim`** (*`int`*) – Number of spatial dimensions to grid over.

Returns

- **Vs** (*numpy.ndarray*) – Velocity evaluation points array of shape `(spatial_dim, spatial_dim*[len(dop_vel)])`.
- **Vols** (*numpy.ndarray*) – "Volume" of each meshpoint. Has same shape as `Vs`.

Examples

```
>>> m = {"method":"uniform", "n_uniform":801}
>>> classes = rq.doppler_classes(method=m)
>>> mesh, vols = rq.doppler_mesh(classes, 2)
>>> print(type(mesh), type(vols))
>>> mesh_np = np.array(mesh)
>>> vols_np = np.array(vols)
>>> print(mesh_np.shape, vols_np.shape)
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
(2, 801, 801) (2, 801, 801)
```

rydiqule.doppler_utils.gaussian3d

`rydiqule.doppler_utils.`**`gaussian3d`**`(`*Vs: ndarray*`)` → ndarray

Evaluate a multi-dimensional gaussian, with sigma=1, for the given detunings.

Parameters

`Vs` (*`numpy.ndarray`*) – Array of normalized velocity classes for which to get the gaussian weighting.

Returns

Gaussian weights for the velocity classes. Has same shape as `Vs`.

Return type

numpy.ndarray

rydiqule.doppler_utils.generate_doppler_shift_eom

`rydiqule.doppler_utils.`**`generate_doppler_shift_eom`**`(`*doppler_hamiltonians: ndarray*`)` → ndarray

doppler_mesh_method Generates the EOMs for the supplied doppler shifts.

Multiply the output by the velocity in each dimension, then add to the normal EOMs to get the full Doppler shifted EOMs.

Parameters

`doppler_hamiltonians` (*`numpy.ndarray`*) – Hamiltonians of only the doppler shifts, one for each spatial dimension to be averaged over.

Returns

Corresponding LHS EOMs with ground removed and in the real basis.

Return type

numpy.ndarray

rydiqule.doppler_utils.get_doppler_equations

`rydiqule.doppler_utils.`**`get_doppler_equations`**`(`*base_eoms: ndarray*, *doppler_hamiltonians: ndarray*, *Vs: ndarray*`)` → ndarray

Returns the equations for each slice of the doppler profile.

A new axes corresponding to these slices are appended to the beginning. For example, if equations are of shape `(m,m)` and there are `n_doppler` doppler values being sampled, the return will be of shape `(n_doppler, m, m)`.

Parameters

- **`base_eoms`** (*`numpy.ndarray`*) – Stacked square arrays representing the unshifted equations, i.e. the theoretical equations for an ensemble of atoms with zero momentum.

- **doppler_hamiltonians** (*numpy.ndarray*) – Arrays of hamiltonians with only doppler shifts present. One for each spatial dimension needed. See *get_doppler_shifts()* for details.
- **Vs** (*numpy.ndarray*) – Mesh of velocity classes to sample, with same spatial dimensions as `dop_ham`. See *doppler_mesh()* for details.

Returns
An array of shape `(*Vs.shape[1:], *base_eoms.shape)` which is a, potentially multi-dimensional, stack of individual equations of shape `(m, m)`. Each slice of this stack is an equation of shape `(m, m)` with the corresponding doppler shifts applied.

Return type
numpy.ndarray

Note: Each doppler shift is equal to `k_i*vP*det_i`, in units of Mrad/s, where `i` denotes the einstein summation along the spatial dimensions. `det` is the normalized velocity class, with `vP*det_i=v_i` giving the velocity. `vP` is the most probable speed from the Maxwell-Boltzmann distribution: sqrt(2*kB*T/m). `k_i` is the k-vector of the field along the same axis as `det_i`. `doppler_hamiltonians` provides `k_i*vP`, `Vs` provides `det_i`.

Classes

DirectMethod(*args, **kwargs)	
IsoPopMethod(*args, **kwargs)	
SplitMethod(*args, **kwargs)	
UniformMethod(*args, **kwargs)	

rydiqule.doppler_utils.DirectMethod

class rydiqule.doppler_utils.**DirectMethod**(**args*, ***kwargs*)

Bases: `dict`

__init__(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`clear`()	
`copy`()	
`fromkeys`([value])	Create a new dictionary with keys from iterable and values set to value.
`get`(key[, default])	Return the value for key if key is in the dictionary, else default.
`items`()	
`keys`()	
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`()	Remove and return a (key, value) pair as a 2-tuple.
`setdefault`(key[, default])	Insert key with a value of default if key is not in the dictionary.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

Attributes

`method`
`doppler_velocities`

clear() → None. Remove all items from D.

copy() → a shallow copy of D

doppler_velocities: Union[ndarray, Sequence]

fromkeys(*value=None*, /)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, /)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

method: Literal['direct']

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, /)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

rydiqule.doppler_utils.IsoPopMethod

class rydiqule.doppler_utils.**IsoPopMethod**(**args*, ***kwargs*)

Bases: dict

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

method
n_isopop

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

method: Literal['isopop']

n_isopop: int

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

rydiqule.doppler_utils.SplitMethod

class rydiqule.doppler_utils.**SplitMethod**(**args*, ***kwargs*)

Bases: dict

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

method
width_doppler
n_doppler
width_coherent
n_coherent

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

method: Literal['split']

n_coherent: int

n_doppler: int

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

width_coherent: float

width_doppler: float

rydiqule.doppler_utils.UniformMethod

class rydiqule.doppler_utils.**UniformMethod**(**args*, ***kwargs*)

Bases: dict

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

method
width_doppler
n_uniform

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

method: Literal['uniform']

n_uniform: int

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

width_doppler: float

6.1.4 rydiqule.energy_diagram

Energy profile diagram

This is a simple script to plot energy profile diagram using matplotlib.

```
E|          4__
n|   2__    /  \
e|1__/  \__/5   \
r|  3\__/       6\__
g|
y|
```

Original author is Giacomo Marchioro. The following is modified from https://github.com/giacomomarchioro/PyEnergyDiagrams

Module Attributes

COLORS	Seaborn colorblind palette, for use with coupling arrows.
COLORS_W	Reversed Seaborn colorblind palette, for use with wavy arrows.

rydiqule.energy_diagram.COLORS

rydiqule.energy_diagram.**COLORS = ['#0173b2', '#de8f05', '#029e73', '#d55e00', '#cc78bc', '#ca9161', '#fbafe4', '#949494', '#ece133', '#56b4e9']**

Seaborn colorblind palette, for use with coupling arrows.

rydiqule.energy_diagram.COLORS_W

rydiqule.energy_diagram.**COLORS_W = ['#006374', '#b8850a', '#3c3c3c', '#a23582', '#592f0d', '#591e71', '#8c0800', '#12711c', '#b1400d', '#001c7f']**

Reversed Seaborn colorblind palette, for use with wavy arrows.

Reversing the order helps limit color overlap when using default ordering.

Functions

draw_wiggly_arrow(ax, start, stop[, amp, ...])	Helper funtion that draws a wavy arrow between two points on a plot.

rydiqule.energy_diagram.draw_wiggly_arrow

rydiqule.energy_diagram.**draw_wiggly_arrow**(*ax: Axes*, *start: Collection*, *stop: Collection*, *amp: float = 0.07*, *nhalfwaves: int = 8*, *arrow_size: float = 0.06*, *linestyle: str = 'solid'*, *color: str = 'k'*, *alpha: float = 1.0*) → None

Helper funtion that draws a wavy arrow between two points on a plot.

Parameters

- **ax** (matplotlib.axes.Axes) – Axes to add the wiggly arrow to.
- **start** (*tuple*) – Arrow start point in axes units
- **stop** (*tuple*) – Arrow stop point in axes units
- **amp** (*float, optional*) – Amplitude of the wave in axes units. Default is 0.07.
- **nhalfwaves** (*int, optional*) – Number of half-waves to wave the arrow. Default is 8.
- **arrow_size** (*float, optional*) – Size of the arrow in axes units. Default is 0.06.
- **linestyle** (*str, optional*) – Matplotlib linestyle definition. Default is 'solid'.
- **color** (*str, optional*) – Matplotlib color specification. Default is 'k'.
- **alpha** (*float, optional*) – Matplotlib alpha specification. Default is 1.

Classes

ED([aspect])	Energy diagram class

rydiqule.energy_diagram.ED

class rydiqule.energy_diagram.**ED**(*aspect: str = 'equal'*)

Bases: `object`

Energy diagram class

__init__(*aspect: str = 'equal'*) → None

Constructor for an energy diagram.

This class contains methods to add energy levels and connections between them. It uses matplotlib to generate the figure, so standard manipulations via matplotlib function calls (including saving of the figure) can be done.

Call `plot()` to actually create the plot.

Parameters

aspect (*str, optional*) – Kwarg passed to `fig.add_subplot()`. Default is `'equal'`.

Note: Calling `plot()` in a jupyter notebook will automatically show the generated figure thanks to jupyter notebook magic in handling matplotlib figures. If you wish to see the plot outside a jupyter notebook, you will need to call `plt.show()` just like any other matplotlib figure.

Methods

__init__([aspect])	Constructor for an energy diagram.
add_arrow(start_level_id, end_level_id, ...)	Add a arrow between two energy levels using IDs of the level.
add_level(energy[, bottom_text, position, ...])	This method add a new energy level to the plot.
add_link(start_level_id, end_level_id[, ...])	Add a link between two energy levels using IDs of the level.
add_wiggly_arrow(start_level_id, ...)	Add a wiggly arrow between two energy levels using IDs of the level.
plot([show_IDs, ylabel, ax])	Plot the energy diagram.

__auto_adjust() → None

Sets the ratio to the best dimension and space between the levels.

add_arrow(*start_level_id: int, end_level_id: int, linestyle: str*) → None

Add a arrow between two energy levels using IDs of the level.

Use self.plot(show_index=True) to show the IDs of the levels.

Parameters

- **start_level_id** (*int*) – Starting level ID
- **end_level_id** (*int*) – Ending level ID
- **linestyle** (*str*) – matplotlib linestyle string

add_level(*energy: float, bottom_text: str = '', position: Optional[Union[str, float]] = None, color: str = 'k', top_text: str = 'Energy', right_text: str = '', left_text: str = '', linestyle: str = 'solid'*) → None

This method add a new energy level to the plot.

Parameters

- **energy** (*int*) – The energy of the level in Kcal mol-1

- **bottom_text** (*str*) – The text on the bottom of the level (label of the level) (default “)
- **position** (*str*) – The position of the level in the plot. Keep it empty to add the level on the right of the previous level use ‘last’ as argument for adding the level to the last position used for the level before. An integer can be used for adding the level to an arbitrary position. (default None)
- **color** (*str*) – matplotlib color specification of the level (default ‘k’)
- **top_text** (*str*) – Text on the top of the level. By default it will print the energy of the level. (default ‘Energy’)
- **right_text** (*str*) – Text at the right of the level. (default “)
- **left_text** (*str*) – Text at the left of the level. (default “)
- **linestyle** (*str*) – The linestyle of the level, one of the following values: ‘solid’, ‘dashed’, ‘dashdot’, ‘dotted’ (default ‘solid’)

add_link (*start_level_id: int, end_level_id: int, color: str = 'k', ls: str = '--', linewidth: float = 1*) → None

Add a link between two energy levels using IDs of the level.

Use self.plot(show_index=True) to show the IDs of the levels.

Parameters

- **start_level_id** (*int*) – Starting level ID
- **end_level_id** (*int*) – Ending level ID
- **color** (*str*) – matplotlib color specification of the line
- **ls** (*str*) – matplotlib line style e.g. – , ..
- **linewidth** (*float*) – line width

add_wiggly_arrow (*start_level_id: int, end_level_id: int, ls_dict: dict*) → None

Add a wiggly arrow between two energy levels using IDs of the level.

Use self.plot(show_index=True) to show the IDs of the levels.

Parameters

- **start_level_id** (*int*) – Starting level ID
- **end_level_id** (*int*) – Ending level ID
- **ls_dict** (*dict*) – Dictionary of linestyle parameters. Passed as kwargs to `draw_wiggly_arrow()`.

plot (*show_IDs: bool = False, ylabel: str = 'Energy / $kcal$ mol^{-1}', ax: Optional[Axes] = None*) → Tuple[Figure, Axes]

Plot the energy diagram.

Use show_IDs=True for showing the IDs of the energy levels and allowing an easy linking.

```
E|          4__
n|   2__    /  \
e|1__/  \__/5   \
r|  3\__/       6\__
g|
y|
```

Parameters

- **show_IDs** (*bool*) – show the IDs of the energy levels
- **ylabel** (*str*) – The label to use on the left-side axis. “Energy / $kcal$ mol^{-1}” by default.

- **ax** (`matplotlib.axes.Axes`) – The axes to plot onto. If not specified, a Figure and Axes will be created for you.

Returns

- **fig** (`matplotlib.figure.Figure`) – Figure handle for the generated figure.
- **ax** (`matplotlib.axes.Axes`) – Axes handle for the generated figure.

6.1.5 rydiqule.experiments

Standard methods for converting results to physical values.

Functions

get_OD(rho_probe, cell_length, probe_rabi, kappa)	Calculates the optical depth from the solution.
get_phase_shift(sol, cell, cell_length[, ...])	Extract the phase shift from a solution.
get_snr(sensor, optical_path_length, param_label)	Calculate a Sensor's signal-to-noise ratio, in a 1Hz bandwidth, to a specified signal parameter.
get_solution_element(sols, idx)	Return a slice of an n_dimensional matrix of solutions of shape (...,n^2-1), where n is the basis size of the quantum system.
get_susceptibility(sol, cell[, probe_tuple])	For a given density matrix solution and cell, return the atomic susceptibility on the probe transition.
get_transmission_coef(sol, cell, cell_length)	Extract the transmission term from a solution.

rydiqule.experiments.get_OD

`rydiqule.experiments.`**`get_OD`**(*rho_probe: ndarray*, *cell_length: float*, *probe_rabi: Union[float, ndarray]*, *kappa: float*) → ndarray

Calculates the optical depth from the solution.

Assumes the optically-thin approximation is valid. If a calculated OD for a solution exceeds 1, this approximation is likely invalid.

Parameters

- **`rho_probe`** (*`numpy.ndarray`*) – Array of matrix elements for the probing transition.
- **`cell_length`** (*`float`*) – Optical path length of the cell, in meters.
- **`probe_rabi`** (*`float or numpy.ndarray`*) – Probe Rabi frequency, in Mrad/s
- **`kappa`** (*`float`*) – kappa constant, in units of MHz/m

Returns

OD – Optical depth of the sample

Return type

numpy.ndarray

Warns

UserWarning – If any OD exceeds 1, which indicates the optically-thin approximation is likely invalid.

Examples

```
>>> c = rq.Cell('Rb85', *rq.D2_states('Rb85'))
>>> c.add_coupling(states=(0,1), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
>>> OD = rq.experiments.get_OD(rq.get_rho_ij(sols, 1, 0), 1e-3, 1, c.kappa)
>>> print(OD)
>>> OD2 = rq.experiments.get_OD(rq.get_rho_ij(sols, 1, 0), 1e-2, 1, c.kappa)
>>> print(OD2)
(3,)
0.7412112017002291
7.412112017002292
~/src/Rydiqule/src/rydiqule/experiments.py:103: UserWarning: At least one␣
↪solution has optical depth greater than 1.
Integrated results are likely invalid.
```

rydiqule.experiments.get_phase_shift

rydiqule.experiments.**get_phase_shift**(*sol:* Solution, *cell:* Cell, *cell_length: float*, *probe_tuple: Tuple[int, ...] = (0, 1)*) → ndarray

Extract the phase shift from a solution.

Assumes the optically-thin approximation is valid.

Parameters

- **sol** (*Solution*) – Solution object to extract phase shift from.
- **cell** (*Cell*) – The cell used to generate the solution. Used to get physical information needed for the calculation.
- **cell_length** (*float*) – Optical path length of the cell, in meters.
- **probe_tuple** (*tuple of int, optional*) – Tuple of probing coupling. Defaults to `(0,1)`.

Returns
Probe phase in radians.

Return type
numpy.ndarray

Examples

```
>>> c = rq.Cell('Rb85', *rq.D2_states('Rb85'))
>>> c.add_coupling(states=(0,1), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
>>> phase_shift = rq.get_phase_shift(sols, c, 1e-3)
>>> print(phase_shift)
(3,)
-0.03807271078849609
```

rydiqule.experiments.get_snr

`rydiqule.experiments.get_snr`(*sensor:* Sensor, *optical_path_length: float*, *param_label: str*, *probe_tuple: Tuple[int, int] = (0, 1)*, *phase_quadrature: bool = False*, *kappa: Optional[float] = None*, *eta: Optional[float] = None*) → Tuple[ndarray, Tuple[ndarray, ...]]

Calculate a Sensor's signal-to-noise ratio, in a 1Hz bandwidth, to a specified signal parameter.

SNR is calculated with respect to the signal parameter, relative to the inital value of the signal parameter. The returned mesh is similarly transformed from the typical sensor mesh, by replacing the total value of the signal parameter with the deviation in the signal parameter (relative to it's initial array value).

The conventions used follow that of[1].

Parameters

- **sensor** (`Sensor`) – sensor for which SNR should be calculated. The definition of sensor.couplings should contain at least one coupling with a list-like parameter. For the list-like parameter, the first array element is the "base" against which SNR for each other value is calculated.
- **optical_path_length** (*int*) – Sensor optical path length in meters.
- **param_label** (*str*) – Label of the axis with respect to which SNR is calculated. See `Sensor.axis_labels()` for more details on axis labeling. The value corresponding to this label should be the list-like parameter with respect to which SNR should be calculated. This parameter list must have at least two elements, and SNR is calculated relative to the first element in the list for all other elements in the list.
- **probe_tuple** (*tuple*) – Two-integer tuple specifying the probing transition.
- **phase_quadrature** (`bool`, optional) – Whether the sensor is measured in the phase quadrature of the probe laser. False denotes measurement in the amplitude quadrature. Default is False.
- **kappa** (*float, optional*) – Differential prefactor, in units of (Mrad/s)/m. Must be specified when using `Sensor`.
- **eta** (*float, optional*) – Noise density prefactor, in units of root(Hz) Must be specified when using `Sensor`.

Returns

- **snrs** (*numpy.ndarray*) – Array of SNRs for the sensor with respect to the change in the signal parameter. Calculated in units of amplitude relative to noise standard deviation.
- **mesh** (*tuple(numpy.ndarray)*) – Numpy meshgrid of the coupling parameters that yield each snr. The signal parameter axis now shows the signal change.

Raises

ValueError – If the specified param_label is not in `Sensor.axis_labels()`

[1] D. H. Meyer, C. O'Brien, D. P. Fahey, K. C. Cox, and P. D. Kunz, "Optimal atomic quantum sensing using electromagnetically-induced-transparency readout," Phys. Rev. A, vol. 104, p. 043103, 2021.

Examples

```
>>> c = rq.Sensor()
>>> c.add_coupling(states=(0,1), rabi_frequency=np.linspace(0, 1, 5),␣
↪detuning=1)
>>> snr, mesh = rq.get_snr(c, 0.01, '(0, 1)_rabi_frequency')
>>> print(snr)
>>> print(mesh)
[     0. 137024. 273980. 410796. 547405.]
[array([0., 0., 0., 1., 1.])]
```

References

rydiqule.experiments.get_solution_element

`rydiqule.experiments.`**`get_solution_element`**(*sols:* Solution, *idx: int*) → ndarray

Return a slice of an n_dimensional matrix of solutions of shape (…,n^2-1), where n is the basis size of the quantum system.

Parameters

- **`sols`** (*`numpy.ndarray`*) – An N-Dimensional numpy array representing the final density matrix, with ground state removed, and written in the totally real equations basis. Can have arbitrary axes preceding density matrix axis.
- **`idx`** (*`int`*) – Solution index to slice.

Returns

Slice of solutions corresponding to index idx. For example, if sols has shape (…, n^2-1), sol_slice will have shape (…).

Return type

numpy.ndarray

Raises

`IndexError` – If `idx` in not within the shape determined by basis size.

Examples

```
>>> c = rq.Cell('Rb85', *rq.D2_states('Rb85'))
>>> c.add_coupling(states=(0,1), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
>>> rho_01 = rq.get_solution_element(sols, 0)
>>> print(rho_01)
(3,)
-0.0013139903428765695
```

rydiqule.experiments.get_susceptibility

`rydiqule.experiments.`**`get_susceptibility`**(*sol:* Solution, *cell:* Cell, *probe_tuple: Tuple[int, ...] = (0, 1)*) → ndarray

For a given density matrix solution and cell, return the atomic susceptibility on the probe transition.

Parameters

- **sol** (*Solution*) – Solution object to extract susceptibility from.
- **cell** (*Cell*) – The cell used to generate the solution. Used to get physical information needed for the calculation.
- **probe_tuple** (*tuple of int, optional*) – Tuple of probing coupling. Defaults to `(0,1)`.

Returns

Susceptibility of the density matrix solution.

Return type

numpy.ndarray

Examples

```
>>> c = rq.Cell('Rb85', *rq.D2_states('Rb85'))
>>> c.add_coupling(states=(0,1), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
>>> sus = rq.get_susceptibility(sols, c)
>>> print(f"{sus:.2f}")
(3,)
-24.40+474.99j
```

rydiqule.experiments.get_transmission_coef

`rydiqule.experiments.`**`get_transmission_coef`**(*sol:* Solution, *cell:* Cell, *cell_length: float*, *probe_tuple: Tuple[int, ...] = (0, 1)*) → ndarray

Extract the transmission term from a solution.

Assumes the optically-thin approximation is valid.

Parameters

- **sol** (*Solution*) – A Solution object containing at least the `rho` attribute. Typically created as the return of `solve_steady_state()` or `~.solve_time`.
- **cell** (*Cell*) – The cell used to generate the solution. Used to get physical information needed for the calculation.
- **cell_length** (*float*) – Optical path length of the cell, in meters.
- **probe_tuple** (*tuple of int, optional*) – Tuple of probing coupling. Defaults to `(0,1)`.

Returns

Numerical value of the probe absorption in fractional units (P_out/P_in).

Return type

numpy.ndarray

Examples

```
>>> c = rq.Cell('Rb85', *rq.D2_states('Rb85'))
>>> c.add_coupling(states=(0,1), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
>>> t = rq.get_transmission_coef(sols, c, 1e-3)
>>> print(t)
(3,)
0.47653638415943955
```

6.1.6 rydiqule.rydiqule_utils

General rydiqule package utilities

Functions

about()	About box describing Rydiqule and its core dependencies.

rydiqule.rydiqule_utils.about

`rydiqule.rydiqule_utils.`**`about`**`()`

About box describing Rydiqule and its core dependencies.

Prints human readable strings of information about the system.

Examples

```
>>> import rydiqule as rq
>>> rq.about()

        Rydiqule
    ================

Rydiqule Version:     0.4.0
Installation Path:    C:\~\rydiqule\src\rydiqule

    Dependencies
    ================

NumPy Version:        1.21.5
SciPy Version:        1.7.3
Matplotlib Version:   3.5.2
ARC Version:          3.2.1
Python Version:       3.9.12
Python Install Path:  C:\~\miniconda3\envs\arc
Platform Info:        Windows (AMD64)
CPU Count:            16
Total System Memory:  256 GB
```

6.1.7 rydiqule.sensor

Sensor objects that control solvers.

Classes

`Sensor`(basis_size, *couplings)	Class that contains minimum information necessary to run the solvers.

rydiqule.sensor.Sensor

class `rydiqule.sensor.`**`Sensor`**(*basis_size: int*, **couplings: Dict*)

Bases: `object`

Class that contains minimum information necessary to run the solvers.

Consider this class the theorist's interface to the solvers. It requires nearly complete, explicit specification of inputs. This allows for very fine control of the solvers, including the ability to solve systems that are not entirely physical.

`__init__`(*basis_size: int*, **couplings: Dict*) → None

Initializes the Sensor of the specified basis size.

Parameters

- **`basis_size`** (*int*) – Number of states in the basis.
- **`*couplings`** (*tuple(dict)*) – Couplings dictionaries to pass to *`add_couplings()`* on sensor construction.

Raises

`TypeError` – If `basis_size` is not an integer.

Methods

__init__(basis_size, *couplings)	Initializes the Sensor of the specified basis size.
add_coupling(states[, rabi_frequency, ...])	Adds a single coupling of states to the system.
add_couplings(*couplings[, no_rwa_warning])	Add any number of couplings between pairs of states.
add_decoherence(states, gamma[, label])	Add decoherent coupling to the graph between two states.
add_self_broadening(node, gamma[, label])	Specify self-broadening (such as collisional broadening) of a level.
add_transit_broadening(gamma_transit[, repop])	Adds transit broadening by adding a decoherence from each node to ground.
axis_labels([collapse])	Get a list of axis labels for stacked hamiltonians.
basis()	Generate basis labels of density matrix components.
collapse_dims(stack)	Collapse dimensions of a stack accoring to the `_zipped_parameters` attribute.
couplings_with(*keys[, method])	Returns a version of self.couplings with only the keys specified.
decoherence_matrix()	Build a decoherence matrix out of the decoherence terms of the graph.
get_couplings()	Returns the couplings of the system as a dictionary
get_doppler_shifts()	Returns the Hamiltonian with only detunings set to the kvector values for each spatial dimension.
get_hamiltonian()	Creates the Hamiltonians from the couplings defined by the fields.
get_hamiltonian_diagonal(values[, no_stack])	Apply addition and subtraction logic corresponding to the direction of the couplings.
get_parameter_mesh([sparse])	Returns the parameter mesh of the sensor.
get_time_couplings()	Returns the list of matrices of all couplings in the system defined with a `time_dependence` key.
get_time_dependence()	Function which returns a list of the `time_dependence` functions.
get_time_hamiltonians()	Get the hamiltonians for the time solver.
get_transition_frequencies()	Gets an array of the diagonal elements of the Hamiltonian from the field detunings.
get_value_dictionary(key)	Get subset of dictionary coupling parameters.
set_gamma_matrix(gamma_matrix)	Set the decoherence matrix for the system.
spatial_dim()	Returns the number of spatial dimensions doppler averaging will occur over.
unzip_parameters(*labels, **extra_kwargs)	Remove a set of zipped parameters from the internal zipped_parameters list.
variable_parameters([apply_mesh])	Property to retrieve the values of parameters that were stored on the graph as arrays.
zip_parameters(*labels)	Define 2 scannable parameters as "zipped" so they are scanned in parallel.

Attributes

eta	Noise density prefactor, in units of root(Hz).
kappa	Differential prefactor, in units of (Mrad/s)/m.

_add_coupling(*states: Tuple[int, ...]*, ***field_params*) → None

Function for internal use which will ensure the supplied couplings is valid, add the field to self.couplings.

Exists to abstract away some of the internally necessary bookkeeping functionality from user-facing classes.

Parameters

- **states** (*tuple*) – The integer pair of states to be coupled.
- ****field_params** (*dict*) – The dictionry of couplings parameters. For details on the keys of the dictionry see `add_coupling()`.

_coupling_with_label(*label: str*) → dict

Helper function to return the pair of states corresponding to a particular label string. For internal use.

_remove_edge_data(*states: Tuple[int, ...]*, *kind: str*)

Helper function to remove all data that was added with a `add_coupling()` call or `add_decoherence()` call. Needed to ensure that two nodes do not have coherent couplings pointing both ways and to invalidate existing zip parameter couplings.

Parameters

- **states** (*tuple*) – Edge from which to remove data.
- **kind** (*str*) – What type of data to remove. Valid options are `coherent` coherent couplings or the incoherent key to be cleared (must start with `gamma`).

Raises

ValueError – If `kind` is not `'coherent'` and doesn't begin with `'gamma'`

_stack_shape(*collapse=True*, *time_dependence='all'*) → Tuple[int, ...]

Internal function to get the shape of the tuple preceding the two hamiltonian axes in `get_hamiltonian()`

_states_valid(*states: Tuple[int, ...]*) → Tuple[int, ...]

Confirms that the provided states are in a valid format.

Typically used internally to validate states added. If provided as a form other than a tuple, first casts to a tuple for consistent indexing.

Checks that `states` contains 2 elements, can be interpreted as a tuple, and that both states lie inside the basis.

Parameters

states (*iterable*) – iterable of to validate. Should be a pair of integers that can be cast to a tuple.

Returns

Length 2 tuple of validated state labels.

Return type

tuple

Raises

- **ValueError** – If `states` has more than two elements.
- **TypeError** – If `states` cannot be converted to a tuple.
- **ValueError** – If either state in `states` is outside the basis.

add_coupling(*states: Tuple[int, ...], rabi_frequency: Optional[Union[float, List[float], ndarray]] = None, detuning: Optional[Union[float, List[float], ndarray]] = None, transition_frequency: Optional[float] = None, phase: Optional[Union[float, List[float], ndarray]] = 0, kvec: Tuple[float, float, float] = (0, 0, 0), time_dependence: Optional[Callable[[float], float]] = None, label: Optional[str] = None, **extra_kwargs*) → None

Adds a single coupling of states to the system.

One or more of these paramters can be a list or array-like of values to represent a laser that can take on a set of discrete values during a field scan. Designed to be a user-facing wrapper for `_add_coupling()` with arguments for states and coupling parameters.

Parameters

- **states** (*tuple of ints of length 2*) – The pair of states of the sensor which the state couples Must be a tuple of intergers of length 2, and the integers must be unique indicies within the basis.
- **rabi_frequency** (*float or complex, or list-like of float or complex*) – The rabi frequency of the field being added. Defined in units of Mrad/s. List-like values will invoke Rydiqule's stacking convention when relevant quantities are calculated.
- **detuning** (*float or list-like of floats or None, optional*) – The frequency difference between the transition frequency and the field frequency in units of Mrad/s. List-like values will invoke Rydiqule's stacking convention when relevant quantities are calculated. If specified, the coupling is treated with the rotating-wave approximation rather than in the lab frame, and `transition_frequency` is ignored if present.
- **transition_frequency** (*float or list-like of floats or None, optional*) – The transition frequency between a particular pair of states. List-like values will invoke Rydiqule's stacking convention when relevant quantities are calculated. Only used directly in calculations if `detuning` is `None`, ignored otherwise. Note that on its own, it only defines the spacing between two energy levels and not the field itsself. To define a field, the `time_dependence` argument must be specified, or else the off-diagonal terms to drive transitions will not be generated in the Hamiltonian matrix.
- **phase** (*float, optional*) – The relative phase of the field in radians. Defaults to zero.
- **kvec** (*iterable, optional*) – A three-element iterable that defines the atomic doppler shift on a particular coupling field. It should have magntiude equal to the doppler shift (in the units of Mrad/s) of a n atom moving at the Maxwell-Boltzmann distribution most probable speed, `vP=np.sqrt(2*kB*T/m)`. I.E. `np.linalg.norm(kvec)=2*np.pi/lambda*vP`. If equal to `(0,0,0)`, solvers will ignore doppler shifts on this field. Defaults to `(0,0,0)`.
- **time_dependence** (*scalar function, optional*) – A scalar function specifying a time-dependent field. The time dependence function is defined as a python funtion that returns a unitless value as a function of time (in microseconds) that is multiplied by the `rabi_frequency` parameter to get a field strength scaled to units of Mrad/s.
- **label** (*str or None, optional*) – Name of the coupling. This does not change any calculations, but can be used to help track individual couplings, and will be reflected in the output of `axis_labels()`, and to specify zipping for `zip_couplings()`. If `None`, the label is generated as the value of `states` cast to a string with whitespace removed. Defaults to `None`.

Raises

- **ValueError** – If `states` cannot be interpreted as a tuple.

- **ValueError** – If `states` does not have a length of 2.
- **ValueError** – If the state numbers specified by `states` are beyond the basis size. For example, calling this function with `states=(3,4)` will raise this error if the basis size is equal to 3.
- **ValueError** – If both `rabi_frequency` and `dipole_moment` are specified or if neither are specified.
- **ValueError** – If both detuning and transition_frequency are specified or if neither are specified.

Examples

```
>>> s = rq.Sensor(2)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=2)
```

```
>>> s = rq.Sensor(2)
>>> s.add_coupling(states=(0,1), detuning=np.linspace(-10, 10, 101), rabi_
↪frequency=2, label="laser")
```

```
>>> s = rq.Sensor(2)
>>> step = lambda t: 1 if t>=1 else 0
>>> s.add_coupling(states=(0,1), transition_frequency=1000, rabi_
↪frequency=2, time_dependence=step)
```

```
>>> s = rq.Sensor(2)
>>> kp = 250*np.array([1,0,0])
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=2, kvec=kp)
```

add_couplings(**couplings: Dict*, *no_rwa_warning: Optional[bool] = False*) → None

Add any number of couplings between pairs of states.

Acts as an alternative to calling *add_coupling()* individually for each pair of states. Can be used interchangably up to preference, and all of keyword *add_coupling()* are supported dictionary keys for dictionaries passed to this function.

Parameters

- **couplings** (*tuple of dicts*) – Any number of dictionaries, each specifying the parameters of a single field coupling 2 states. For more details on the keys of each dictionry see the arguments for *add_coupling()*. Equivalent to passing each dictiories keys and values to *add_coupling()* individually.
- **no_rwa_warning** (*bool*) – Whether to supress the warning that appears when adding a coupling with a sufficiently transition frequency that solving the system may take prohibitively long without the rotating wave approximation.

Raises

ValueError – If the `states` parameter is missing.

Examples

```
>>> s = rq.Sensor(3)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":2}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":4}
>>> s.add_couplings(blue, red)
>>> print(s.couplings.edges(data=True))
[(0, 1, {'rabi_frequency': 1, 'detuning': 2, 'phase': 0, 'kvec': (0, 0,␣
↪0)}), (1, 2, {'rabi_frequency': 3, 'detuning': 4, 'phase': 0, 'kvec':␣
↪(0, 0, 0)})]
```

add_decoherence(*states: Tuple[int, ...], gamma: Union[float, List[float], ndarray], label: Optional[str] = None*) → None

Add decoherent coupling to the graph between two states.

If `gamma` is list-like, the sensor will scan over the values, solving the system for each different gamma.

Parameters

- **states** (*tuple of ints*) – Length-2 tuple of integers corresponding to the two states. The first value is the number of state out of which population decays, and the second is the number of the state into which population decays.
- **gamma** (*float or sequence*) – The decay rate, in Mrad/s.
- **label** (*str or None, optional*) – Optional label for the decay. If `None`, decay will be stored on the graph edge as `"gamma"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`.

Notes

Note: Adding a decoherece with a particular label (including `None`) will override an existing decoherent transition with that label.

Examples

s = rq.Sensor(3) >>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1) >>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1) >>> s.add_decoherence((2,0), 0.1, label=”misc”) >>> print(s.decoherence_matrix()) [[0. 0. 0.] [0. 0. 0.] [0.1 0. 0.]]

Decoherence values can also be scanned. Here decoherece from states 2->0 is scanned between 0 and 0.5 for 11 values. We can also see how the Hamiltonian shape accounts for this to allow for clean broadcasting, indicating that the hamiltonian is identical accross all decoherence values.

```
>>> s = rq.Sensor(3)
>>> gamma = np.linspace(0,0.5,11)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_decoherence((2,0), gamma)
>>> print(s.decoherence_matrix().shape)
(11, 3, 3)
>>> print(s.get_hamiltonian().shape)
(11, 3, 3)
```

add_self_broadening(*node: int, gamma: Union[float, List[float], ndarray], label: Optional[str] = 'self'*) → None

Specify self-broadening (such as collisional broadening) of a level.

Equivalent to calling *add_decoherence()* and specifying both states to be the same, with the "self" label. For more complicated systems, it may be useful to further specify the source of self-broadening as, for example, "collisional" for easier bookkeeping and to ensure no values are overwritten.

Parameters

- **node** (*int*) – The integer number of the state node to which the broadening will be added. The integer corresponds to the state's position in the graph.
- **gamma** (*float or sequence*) – The broadening width to be added in Mrad/s.
- **label** (*str, optional*) – Optional label for the state. If `None`, decay will be stored on the graph edge as `"gamma"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`

Notes

Note: Just as with the *add_decoherence()* function, adding a decoherence value with a label that already exists will overwrite an existing decoherent transition with that label. The "self" label is applied to this function automatically to help avoid an unwanted overwrite.

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_self_broadening(1, 0.1)
>>> print(s.couplings.edges(data=True))
>>> print(s.decoherence_matrix())
[(1, 1, {'gamma_self': 0.1, 'label': '(1,1)'})]
[[0.  0.  0. ]
[0.  0.1 0. ]
[0.  0.  0. ]]
```

add_transit_broadening(*gamma_transit: Union[float, List[float], ndarray], repop: Dict[int, float] = {0: 1.0}*) → None

Adds transit broadening by adding a decoherence from each node to ground.

For each graph node n, adds a decoherent transition from n the specified state (0 by default) using the *add_decoherence()* method with the `"transit"` label. See *add_decoherence()* for more details on labeling.

If an array of transit values are provided, they will be automatically zipped together into a single scanning element.

Parameters

- **gamma_transit** (*float or sequence*) – The transit broadening rate in Mrad/s.
- **repop** (*dict, optional*) – Dictionary of states for transit to repopulate in to. The keys represent the state labels. The values represent the fractional amount that goes to that state. If the sum of values does not equal 1, population will not be conserved. Default is to repopulate everything into state 0.

Warns

- **If the values of the `repop` parameter do not sum to 1, thus meaning**
- **population will not be conserved.**

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_transit_broadening(0.1)
>>> print(s.couplings.edges(data=True))
>>> print(s.decoherence_matrix())
[(0, 0, {'gamma_transit': 0.1}), (1, 0, {'gamma_transit': 0.1}), (2, 0, {
↪'gamma_transit': 0.1})]
[[0.1 0.  0. ]
[0.1 0.  0. ]
[0.1 0.  0. ]]
```

axis_labels(*collapse: bool = True*) → List[str]

Get a list of axis labels for stacked hamiltonians.

The axes of a hamiltonian stack are defined as the axes preceding the usual hamiltonian, which are always the last 2. These axes only exist if one of the parametes used to define a Hamiltonian are lists.

Be default, labels which have been zipped using *zip_parameters()* will be combined into a single label, as this is how *get_hamiltonian()* treats these axes.

Returns
Strings corresponding to the label of each axis on a stack of multiple hamiltonians.

Return type
list of str

Examples

There are no preceding axes if there are no list-like parameters.

```
>>> s = rq.Sensor(3)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":2}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":4}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape())
>>> print(s.axis_labels())
(3,3)
[]
```

Adding list-like parameters expands the hamiltonian

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-10, 10, 11)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":det, "label":
↪"blue"}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape)
>>> print(s.axis_labels())
(11, 11, 3, 3)
['blue_detuning', '(1, 2)_detuning']
```

Zipping parameters combines their corresponding labels, since their Hamiltonians now lie on a single axis of the stack. Here the axis of length 7 (axis 0) corresponds to the rabi frequencies and the axis of shape 11 (axis 1) corresponds to the zipped detunings

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=np.arange(11), rabi_
↪frequency=np.linspace(-3, 3, 7))
>>> s.add_coupling(states=(1,2), detuning=0.5*np.arange(11), rabi_
↪frequency=1)
```

(continues on next page)

(continued from previous page)

```
>>> s.zip_parameters("(0,1)_detuning", "(1,2)_detuning")
>>> print(s.get_hamiltonian().shape)
>>> print(s.axis_labels())
(7, 11, 3, 3)
['(0,1)_rabi_frequency', '(0,1)_detuning|(1,2)_detuning']
```

basis() → ndarray

Generate basis labels of density matrix components.

The basis corresponds to the elements in the solution. This is not the complex basis of the sensor class, but rather the real basis of a solution after calling one of `rydiqule`'s solvers. This means that the ground state population has been removed and it has been transformed to the real basis.

Returns

Array of string labels corresponding to the solving basis. Is a 1-D array of length `n**2-1`.

Return type

numpy.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> print(s.basis())
['01_real' '02_real' '01_imag' '11_real' '12_real' '02_imag' '12_imag'
'22_real']
```

collapse_dims(*stack: ndarray*) → ndarray

Collapse dimensions of a stack accoring to the `_zipped_parameters` attribute.

Designed primarily for internal use, but can be called externally or in custom overloads of *`get_hamiltonian()`*. Effectively performs a `numpy.einsum` operation to replicate `numpy.diag` along the appropriate dimensions.

Parameters

stack (*numpy.ndarray*) – Array with shape matching uncollapsed system hamiltonians.

Returns

Input array with appropriate dimensions removed via a numpy.diag-like operation.

Return type

numpy.ndarray

couplings_with(**keys: str*, *method: Literal['all', 'any', 'not any'] = 'all'*) → Dict[Tuple[int, ...], Dict]

Returns a version of self.couplings with only the keys specified.

Can be specified with a several criteria, including all, none, or any of the keys specified.

Parameters

- **str)** (*keys (tuple of)*) – parameter names for a state. See *`add_coupling()`* for which names are valid for a Sensor object.
- **method** (*{'all', 'any', 'not any'}*) – Method to see if a given field matches the keys given. Choosing "all" will return couplings which have keys matching all of the values provided in the keys argument, while coosing "any", will return all couplings with keys matching at least one of the values specified by keys. For example, `sensor.couplings_with("rabi_frequency")` returns a dictionary of all couplings for which a rabi_frequency was specified. `sensor.couplings_with("rabi_frequency", "detuning", method="all")` returns all couplings for which both rabi_frequency and

detuning are specified. ‘sensor.couplings_with(“rabi_frequency”, “detuning”, method=”any”)` returns all couplings for which either rabi_frequency or detuning are specified. Defaults to “all”.

Returns
A copy of the `sensor.couplings` dictionary with only couplings containing the specified parameter keys.

Return type
dict

Examples

Can be used, for example, to return couplings in the roating wave approximation.

```
>>> s = rq.Sensor(3)
>>> sinusoid = lambda t: 0 if t<1 else sin(100*t)
>>> f2 = {"states": (0,1), "detuning": 1, "rabi_frequency":2}
>>> f1 = {"states": (1,2), "transition_frequency":100, "rabi_frequency":1,
↪ "time_dependence": sinusoid}
>>> s.add_couplings(f1, f2)
>>> gamma = np.array([[.2,0,0],
...                   [.1,0,0],
...                   [0.05,0,0]])
>>> s.set_gamma_matrix(gamma)
>>> print(s.couplings_with("detuning"))
{(0, 1): {'rabi_frequency': 2, 'detuning': 1, 'phase': 0, 'kvec': (0, 0,␣
↪0), 'no_rwa_warning': False, 'label': '(0,1)'}}
```

decoherence_matrix() → ndarray

Build a decoherence matrix out of the decoherence terms of the graph.

For each edge, sums all parameters with a key that begins with “gamma”, and places it on the appropriate location in an adjacency matrix for the `couplings` graph.

Returns
The decoherence matrix stack of the system.

Return type
numpy.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_decoherence((1,0), 0.2, label="foo")
>>> s.add_decoherence((1,0), 0.1, label="bar")
>>> s.add_decoherence((2,0), 0.05)
>>> s.add_decoherence((2,1), 0.05)
>>> print(s.couplings.edges(data=True))
>>> print(s.decoherence_matrix())
[(1, 0, {'gamma_foo': 0.2, 'label': '(1,0)', 'gamma_bar': 0.1}), (2, 0, {
↪'gamma': 0.05, 'label': '(2,0)'}), (2, 1, {'gamma': 0.05, 'label': '(2,
↪1)'})]
[[0.   0.   0.  ]
[0.3  0.   0.  ]
[0.05 0.05 0.  ]]
```

Decoherences can be stacked just like any parameters of the Hamiltonian:

```
>>> s = rq.Sensor(3)
>>> gamma = np.linspace(0,0.5, 11)
>>> s.add_decoherence((1,0), gamma)
>>> print(s.decoherence_matrix().shape)
(11,3,3)
```

eta: Optional[float] = None

Noise density prefactor, in units of root(Hz). Must be specified when using *Sensor*. Automatically calculated when using `Cell`.

get_couplings() → Dict[Tuple[int, ...], Dict]

Returns the couplings of the system as a dictionary

Deprecating in favor of calling the couplings.edges attribute directly.

Returns
A dictionary of key-value pairs with the keys corresponding to levels of transition, and the values being dictionaries of coupling attributes.

Return type
dict

get_doppler_shifts() → ndarray

Returns the Hamiltonian with only detunings set to the kvector values for each spatial dimension.

Determining if a float should be treated as zero is done using `numpy.isclose`, which has default absolute tolerance of `1e-08`.

Returns
Array of shape (used_spatial_dim,n,n), Hamiltonians with only the doppler shifts present along each non-zero spatial dimension specified by the fields' "kvec" parameter.

Return type
numpy.ndarray

get_hamiltonian() → ndarray

Creates the Hamiltonians from the couplings defined by the fields.

They will only be the steady state hamiltonians, i.e. will only contain terms which do not vary with time. Implicitly creates hamiltonians in "stacks" by creating a grid of all supported coupling parameters which are lists. This grid of parameters will not contain rabi-frequency parameters which vary with time and are defined as list-like. Rather, the associated axis will be of length 1, with the scanning over this value handled by the *get_time_couplings()* function.

For m list-like parameters x1,x2,…,xm with shapes N1,N2,…,Nm, and basis size n, the output will be shape `(N1,N2,...,Nm, n, n)`. The dimensions N1,N2,…Nm are labeled by the output of *axis_labels()*.

If any parameters have been zipped with the `_zip_parameters()` method, those parameters will share an axis in the final hamiltonian stack. In this case, if axis N1 and N2 above are the same shape and zipped, the final Hamiltonian will be of shape `(N1,...,Nm, n, n)`

See rydiqule's conventions for matrix stacking for more details.

Returns
The complex hamiltonian stack for the sensor.

Return type
np.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":det}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(red, blue)
>>> print(s.get_hamiltonian().shape)
(11, 11, 3, 3)
```

Time dependent couplings are handled separately. The axis that contains array-like parameters with time dependence is length 1 in the steady-state Hamiltonian.

```
>>> s = rq.Sensor(3)
>>> rabi = np.linspace(-1,1,11)
>>> step = lambda t: 0 if t<1 else 1
>>> blue = {"states":(0,1), "rabi_frequency":rabi, "detuning":1}
>>> red = {"states":(1,2), "rabi_frequency":rabi, "detuning":0, 'time_
↪dependence': step}
>>> s.add_couplings(red, blue)
>>> print(s.get_hamiltonian().shape)
(11, 1, 3, 3)
```

Zipping parameters means they share an axis in the Hamiltonian.

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=np.arange(11), rabi_frequency=2)
>>> s.add_coupling(states=(1,2), detuning=0.5*np.arange(11), rabi_
↪frequency=1)
>>> s.zip_parameters("(0,1)_detuning", "(1,2)_detuning")
>>> H = s.get_hamiltonian()
>>> print(H.shape)
(11, 3, 3)
```

get_hamiltonian_diagonal(*values: dict*, *no_stack: bool = False*) → ndarray

Apply addition and subtraction logic corresponding to the direction of the couplings.

For a given state `n`, the path from ground will be traced to `n`. For each edge along this path, values will be added where the path direction and coupling direction match, and subtracting values where they do not. The sum of all such values along the path is the `n` th term in the output array.

Primarily for internal functions which help generate hamiltonians. Most commonly used to calculate total detunings for ranges of couplings under the RWA

Parameters

- **values** (*dict*) – Key-value pairs where the keys correspond to transitions (agnostic to ordering of states) and values corresponding to the values to which the logic will be applied.
- **no_stack** (*bool, optional*) – Whether to ignore variable parameters in the system and use only basic math operations rather than reshape the output. Typically only `True` for calculating doppler shifts.

Returns

The digonal of the hamiltonian of the system of shape `(*l,n)`, where `l` is the shape of the hamiltonian stack for the sensor.

Return type

numpy.ndarray

get_parameter_mesh(*sparse: bool = True*, ***kwargs*) → Tuple[ndarray, ...]

Returns the parameter mesh of the sensor.

The parameter mesh is the flattened grid of variable parameters in all the couplings of a sensor. Wraps `numpy.meshgrid` by passing arguments as the variable parameters in a sensor. The `indexing` argument is always `"ij"` for matrix indexing. For full documention of all arguments, see `numpy.meshgrid` documentation. Argument documentation copied from `numpy.meshgrid`.

Parameters

- **sparse** (*bool, optional*) – If `True`, the shape of the returned coordinate array for dimension i is reduced from `(N1, ..., Ni, ... Nn)` to `(1, ..., 1, Ni, 1, ..., 1)`. These sparse coordinate grids are intended to be used in a way consistent with numpy's broadcasting conventions. When all coordinates are used in an expression, broadcasting still leads to a fully-dimensonal result array. Recommended to keep as `True`. Default is `True`.
- **kwargs** (*dict, optional*) – Additional keyword arguments to pass to `numpy.meshgrid`. All arguments are passed directly to `numpy.meshgrid` with the exception of `indexing`, which is fixed to `"ij"` and cannot be specified.

Examples

```
>>> s = rq.Sensor(3)
>>> rabi1 = np.linspace(-1,1,11)
>>> rabi2 = np.linspace(-2,2,21)
>>> s.add_coupling(states=(0,1), rabi_frequency=rabi1, detuning=1)
>>> s.add_coupling(states=(1,2), rabi_frequency=rabi2, detuning=1)
>>> for p in s.get_parameter_mesh():
...     print(p.shape)
(11, 1)
(1, 21)
```

get_time_couplings() → Tuple[ndarray, ndarray]

Returns the list of matrices of all couplings in the system defined with a `time_dependence` key.

The ouput will be two lists of matricies representing which terms of the hamiltonian are dependent on each time-dependent coupling. The lists will be of length M and shape `(*l_time, n, n)`, where M is the number of time-dependent couplings, `l_time` is time-dependent stack shape (possibly all ones), and `n` is the basis size. Each matrix will have terms equal to the rabi frequency (or half the rabi frequency under RWA) in positions that correspond to the associated transition. For example, in the case where there is a `time_dependence` function defined for the `(2,3)` transition with a rabi frequency of 1, the associated time coupling matrix will be all zeros, with a 1 in the `(2,3)` and `(3,2)` positions.

Typically, this function is called internally and multiplied by the output of the *`get_time_dependence()`* function.

Returns

- *list of numpy.ndarray* – The list of M `(*l,n,n)` matrices representing the real-valued time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.
- *list of numpy.ndarray* – The list of M `(*l,n,n)` matrices representing the imaginary-valued time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.

Examples

```
>>> s = rq.Sensor(3)
>>> step = lambda t: 0 if t<1 else 1
>>> wave = lambda t: np.sin(2000*np.pi*t)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1,
↪ "time_dependence":wave}
>>> f2 = {"states": (1,2), "transition_frequency":10, "rabi_frequency": 2,
↪ "time_dependence":step}
>>> s.add_couplings(f1, f2)
>>> time_hams, time_hams_i = s.get_time_couplings()
>>> for H in time_hams:
...     print(H)
[[0.+0.j 1.+0.j 0.+0.j]
[1.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j]]
[[0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 2.+0.j]
[0.+0.j 2.+0.j 0.+0.j]]
```

To handle stacking across the steady-state and time hamiltonians, the dimensions are matched in a way that broadcasting works in a numpy-friendly way

```
>>> s = rq.Sensor(3)
>>> rabi = np.linspace(-1,1,11)
>>> step = lambda t: 0 if t<1 else 1
>>> blue = {"states":(0,1), "rabi_frequency":rabi, "detuning":1}
>>> red = {"states":(1,2), "rabi_frequency":rabi, "detuning":0, 'time_
↪dependence': step}
>>> s.add_couplings(red, blue)
>>> time_hams, time_hams_i = s.get_time_couplings()
>>> print(s.get_hamiltonian().shape)
>>> print(time_hams[0].shape)
>>> print(time_hams_i[0].shape)
(1, 11, 3, 3)
(11, 1, 3, 3)
(11, 1, 3, 3)
```

get_time_dependence() → List[Callable[[float], float]]

Function which returns a list of the `time_dependence` functions.

The list is returned with in the order that matches with the time hamiltonians from *get_time_couplings()* such that the ith element of of the return of this functions corresponds with the ith Hamiltonian terms returned by that function.

Returns

List of scalar functions, representing all couplings specified with a `time_dependence`.

Return type

list

Examples

```
>>> s = rq.Sensor(3)
>>> step = lambda t: 0 if t<1 else 1
>>> wave = lambda t: np.sin(2000*np.pi*t)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1,
↪ "time_dependence":wave}
>>> f2 = {"states": (1,2), "transition_frequency":10, "rabi_frequency": 2,
↪ "time_dependence":step}
>>> s.add_couplings(f1, f2)
>>> print(s.get_time_dependence())
[<function <lambda> at 0x7fb310edd9d0>, <function <lambda> at␣
↪0x7fb37c0c81f0>]
```

get_time_hamiltonians() → Tuple[ndarray, ndarray, ndarray]

Get the hamiltonians for the time solver.

Get both the steady state hamiltonian (as returned by *get_hamiltonian()*) and the time_dependent hamiltonians (as returned by *get_time_couplings()*). The time dependent hamiltonians give 2 terms, the hamiltonian corresponding to the real part of the coupling and the hamiltonian corresponding to the imaginary part.

Returns

- **hamiltonian_base** (*np.ndarray*) – The `(*l,n,n)` shape base hamiltonian of the system containing all elements that do not depend on time, where `n` is the basis size of the sensor.
- **dipole_matrix_real** (*np.ndarray*) – The `(M,n,n)` shape array of matrices representing the real time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.
- **dipole_matrix_imag** (*nd.ndarray*) – The `(M,n,n)` shape array of matrices representing the imaginary time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.

Examples

```
>>> s = rq.Sensor(2)
>>> step = lambda t: 0. if t<1 else 1.
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1, time_
↪dependence=step)
>>> H_base, H_time_real, H_time_imaginary = s.get_time_hamiltonians()
>>> print(H_base)
>>> print(H_time_real)
>>> print(H_time_imaginary)
[[0.+0.j 0.+0.j]
[0.+0.j 1.+0.j]]
[array([[0. +0.j, 0.5+0.j],
    [0.5+0.j, 0. +0.j]])]
[array([[0.+0.j , 0.+0.5j],
    [0.-0.5j, 0.+0.j ]])]
```

get_transition_frequencies() → ndarray

Gets an array of the diagonal elements of the Hamiltonian from the field detunings.

Wraps the *get_hamiltonian_diagonal()* function using both transition frequencies and detunings. Primarily for internal use.

Returns
N-D array of the hamiltonian diagonal. For an n-level system with stack shape `*l`, will be shape `(*l, n)`

Return type
numpy.ndarray

get_value_dictionary(*key: str*) → dict

Get subset of dictionary coupling parameters.

Return a dictionary of key value pairs where the keys are couplings added to the system and the values are the value of the parameter specified by key. Produces an output that can be passed directly to *get_hamiltonian_diagonal()*. Only couplings whose parameter dictionaries contain "key" will be in the returned dictionary.

Parameters
key (*str*) – String value of the parameter name to build the dictionary. For example, `get_value_dictionary("detuning")` will return a dictionary with keys corresponding to transitions and values corresponding to detuning for each transition which has a detuning.

Returns
Coupling dictionary with couplings as keys and corresponding values set by input key.

Return type
dict

Examples

```
>>> s = rq.Sensor(4)
>>> f1 = {"states": (0,1), "detuning": 2, "rabi_frequency": 1}
>>> f2 = {"states": (1,2), "detuning": 3, "rabi_frequency": 2}
>>> f3 = {"states": (2,3), "rabi_frequency": 3, "transition_frequency": 3}
>>> s.add_couplings(f1, f2, f3)
>>> print(s.get_value_dictionary("detuning"))
{(0,1): 2, (1,2): 3}
```

kappa: Optional[float] = None

Differential prefactor, in units of (Mrad/s)/m. Must be specified when using *Sensor*. Automatically calculated when using `Cell`.

set_gamma_matrix(*gamma_matrix: ndarray*) → None

Set the decoherence matrix for the system.

Works by first removing all existing decoherent data from graph edges, then individually adding all nonzero terms of a provided gamma matrix to the corresponding graph edges. Can be used to set all decoherence attributes to edges simultaneously, but *add_decoherence()* is preferred.

Unlike *add_decoherence()*, does not support scanning multiple decoherence values, rather should be used to set the decoherences of the system to individual static values.

Parameters
gamma_matrix (*numpy.ndarray*) – Array of shape `(basis_size, basis_size)`. Element `(i,j)` describes the decoherence rate, in Mrad/s, from state `i` to state `j`.

Raises

- **TypeError** – If `gamma_matrix` is not a numpy array.
- **ValueError** – If `gamma_matrix` is not a square matrix of the appropriate size
- **ValueError** – If the shape of `gamma_matrix` is not compatible with `self.basis_size`.

Examples

```
>>> s = rq.Sensor(2)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1}
>>> s.add_couplings(f1)
>>> gamma = np.array([[.1,0],[.1,0]])
>>> s.set_gamma_matrix(gamma)
>>> print(s.decoherence_matrix())
[[0.1 0. ]
 [0.1 0. ]]
```

spatial_dim() → int

Returns the number of spatial dimensions doppler averaging will occur over.

Determining if a float should be treated as zero is done using `numpy.isclose`, which has default absolute tolerance of `1e-08`.

Returns

Number of dimensions, between 0 and 3, where 0 means no doppler averaging kvectors have been specified or are too small to be calculates.

Return type

int

Examples

No spatial dimesions specified

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_freqency=1)
>>> print(s.spatial_dim())
0
```

One spatial dimension specified

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_freqency=1, kvec=(0,0,1))
>>> print(s.spatial_dim())
1
```

Multiple spatial dimensions can exist in a single coupling or across multiple couplings

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_freqency=1, kvec=(1,0,1))
>>> print(s.spatial_dim())
2
```

```
>>> s = rq.Sensor(3)
>>> s.add_coupling((0,1), detuning = 1, rabi_freqency=1, kvec=(1,0,1))
>>> s.add_coupling((1,2), detuning = 2, rabi_freqency=2, kvec=(0,1,0))
>>> print(s.spatial_dim())
3
```

unzip_parameters(**labels: str*, ***extra_kwargs*)

Remove a set of zipped parameters from the internal zipped_parameters list.

If an element of the internal `_zipped_parameters` array matches ALL labels provided, removes it from the internal `zipped_parameters` method. If no such element is in `_zipped_parameters`, does nothing.

Parameters

- **labels** (*str*) – Any number of string labels matching ALL the names of a set of parameters that has already been zipped.
- ****extra_kwargs** (*dict*) –

  Optional Extra keyword-only arguments. Supported argument is

  – verbose (str): Whether to print if there are no parameters matching `labels` in `_zipped_parameters`. Defaults to `True`.

Notes

Note: This function should always be used rather than modifying the `_zipped_parameters` attribute directly.

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label=
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters("probe_detuning", "(1,2)_detuning")
>>> print(s._zipped_parameters) #NOT modifying directly
>>> s.unzip_parameters('(1,2)_detuning', 'probe_detuning')
>>> print(s._zipped_parameters) #NOT modifying directly
[['(1,2)_detuning', 'probe_detuning']]
[]
```

If the labels provided are not a match, a message is printed and nothing is altered.

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label=
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters("probe_detuning", "(1,2)_detuning")
>>> print(s._zipped_parameters) #NOT modifying directly
>>> s.unzip_parameters('green_detuning', 'probe_detuning')
>>> print(s._zipped_parameters) #NOT modifying directly
[['(1,2)_detuning', 'probe_detuning']]
No zipped parameters matching ['green_detuning', 'probe_detuning']
[['(1,2)_detuning', 'probe_detuning']]
```

variable_parameters(*apply_mesh=False*)

Property to retrieve the values of parameters that were stored on the graph as arrays.

Values are returned as a list of tuples in the standard order of pythons default sorting, applied first to the tuple indicating states and then to the key of the parameter itself. This means that couplings are sorted first by lower state, then by upper state, then alphabetically by the name of the parameter.

Returns
A list of tuples corresponding to the parameters of the systems that are variable (i.e. stored as an array). They are ordered accordning to states, then according to variable name. Tuple entries of the list take the form `(states, param_name, value)`

Return type
list of tuples

Examples

```
>>> s = rq.Sensor(3)
>>> vals = np.linspace(-1,2,3)
>>> s.add_coupling(states=(1,2), rabi_frequency=vals, detuning=1)
>>> s.add_coupling(states=(0,1), rabi_frequency=vals, detuning=vals)
>>> for states, key, value in s.variable_parameters():
...     print(f"{states}: {key}={value}")
(0, 1): detuning=[-1.   0.5  2. ]
(0, 1): rabi_frequency=[-1.   0.5  2. ]
(1, 2): rabi_frequency=[-1.   0.5  2. ]
```

zip_parameters(**labels: str*) → None

Define 2 scannable parameters as "zipped" so they are scanned in parallel.

Zipped parameters will share an axis when quantities relevant to the equations of motion, such as the `gamma_matrix` and `hamiltonian` are generated. Note that calling this function does not affect internal quanties directly, but adds their labels together in the internal `self._zipped_parameters` list, and they are zipped at calculation time for `hamiltonian` and `decoherence_matrix`.

Parameters

labels (*str*) – Parameter labels to scan together. Parameter labels are strings of the form `"<coupling_label>_<parameter_name>"`, such as `"(0, 1)_detuning"`. Must be at least 2 labels to zip.

Raises

- **ValueError** – If fewer than 2 labels are provided.
- **ValueError** – If any of the 2 labels are the same.
- **ValueError** – If any elements of `labels` are not labels of couplings in the sensor.
- **ValueError** – If any of the parameters specified by labels are already zipped.
- **ValueError** – If any of the parameters specified are not list-like.
- **ValueError** – If all list-like parameters are not the same length.

Notes

Note: This function should be called last after all Sensor couplings and dephasings have been added. Changing a coupling that has already been zipped removes it from the `self.zipped_parameters` list.

Note: Modifying the `Sensor.zipped_parameters` attribute directly can break some functionality and should be avoided. Use this function or *unzip_parameters()* instead.

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label=
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters("probe_detuning", "(1,2)_detuning")
>>> print(s._zipped_parameters) #NOT modifying directly
[['(1,2)_detuning', 'probe_detuning']]
```

6.1.8 rydiqule.sensor_solution

Bunch-like object use to store aspects of a solution when calling rydiule.solve() Adds essential keys with "None" entries

Classes

Solution(**kwargs)	Manual implementation of a bunch object which fuctions as a dictionary with the ability to access elements.

rydiqule.sensor_solution.Solution

class rydiqule.sensor_solution.**Solution**(***kwargs*)

Bases: dict

Manual implementation of a bunch object which fuctions as a dictionary with the ability to access elements.

For now, little additional funcitonality exists on top of this, but some may be added in the future.

__init__(***kwargs*)

Methods

__init__(**kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	Remove and return a (key, value) pair as a 2-tuple.
rho_ij(i, j)	Gets the i,j element(s) of the density matrix solutions.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

rho	Solutions returned by the solver.
eta	Eta constant from the Cell.
kappa	Kappa constant from the Cell.
couplings	Dictionary of the couplings.
axis_labels	Labels for the axes of scanned parameters.
axis_values	Value arrays corresponding to each axis.
rq_version	Version of rydiqule that created the Solution.
doppler_classes	Doppler classes used to perform the doppler average.
t	Times the solution is returned at, when using the time solver.
init_cond	Initial conditions, when using the time solver.

axis_labels: list[str]

Labels for the axes of scanned parameters. If doppler averaging but not summing, doppler dimensions are prepended.

Type
list of str

axis_values: list

Value arrays corresponding to each axis. If doppler averaging but not summing, doppler classes in internal units are added.

Type
list

clear() → None. Remove all items from D.

copy() → a shallow copy of D

couplings: dict

Dictionary of the couplings.

Type
dict

doppler_classes: Optional[np.ndarray]

Doppler classes used to perform the doppler average. Will be None if doppler averaging was not used.

Type
numpy.ndarray, optional

eta: Optional[float]

Eta constant from the Cell. Not generally defined when using a Sensor.

Type
float, optional

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

init_cond: np.ndarray

Initial conditions, when using the time solver. Undefined otherwise.

Type
numpy.ndarray

items() → a set-like object providing a view on D's items

kappa: Optional[float]

Kappa constant from the Cell. Not generally defined when using a Sensor.

Type
float, optional

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

rho: np.ndarray

Solutions returned by the solver.

Type
numpy.ndarray

rho_ij(*i: int*, *j: int*) → ndarray

Gets the i,j element(s) of the density matrix solutions.

See `get_rho_ij()` for details.

Parameters

- **i** (`int`) – density matrix element `i`
- **j** (`int`) – density matrix element `j`

Returns
`[i,j]` elments of the density matrix

Return type
numpy.ndarray

rq_version: str

Version of rydiqule that created the Solution.

Type
str

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

t: np.ndarray

Times the solution is returned at, when using the time solver. Undefined otherwise.

Type
numpy.ndarray

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

6.1.9 rydiqule.sensor_utils

Utilities used by the Sensor classes.

Functions

calc_eta(probe_freq, probe_elem, beam_area)	Calculate eta according to Eq.
calc_kappa(probe_freq, probe_elem, density)	Calculate kappa according to Eq.
draw_diagram(sensor[, include_dephasing])	Draw a matplotlib plot that shows the energy level diagram, couplings, and dephasing paths.
generate_eom(hamiltonian, gamma_matrix[, ...])	Create the optical bloch equations for a hamiltonian and decoherence matrix using the Lindblad master equation.
get_basis_transform(basis_size)	Function that defines the basis transformation matrix u and its inverse u_i, between the real and complex basis.
get_rho_ij(sols, i, j)	For a given density matrix solution, retrieve a specific element of the density matrix.
get_rho_populations(sols)	For a given density matrix solution, return the diagonal populations.
make_real(equations, constant[, ground_removed])	Converts equations of motion from complex basis to real basis.
remove_ground(equations)	Remove the ground state from the equations of motion using population conservation.
scale_dipole(dipole)	Scale a dipole matrix from units of a0*e to Mrad/s when multiplied by a field in V/m.

rydiqule.sensor_utils.calc_eta

`rydiqule.sensor_utils.`**`calc_eta`**(*probe_freq: float*, *probe_elem: float*, *beam_area: float*) → float

Calculate eta according to Eq. 7 of Meyer et. al. PRA 104, 043103 (2021)

Parameters

- **`probe_freq`** (*`float`*) – Frequency of the probing field, in Hz
- **`probe_elem`** (*`float`*) – Dipole moment of the probing transition, in units of a_0*e
- **`beam_area`** (*`float`*) – Area of the probing field, in m^2

Returns

eta – in units of sqrt(MHz)

Return type

float

rydiqule.sensor_utils.calc_kappa

`rydiqule.sensor_utils.`**`calc_kappa`**(*probe_freq: float*, *probe_elem: float*, *density: float*) → float

Calculate kappa according to Eq. 5 of Meyer et. al. PRA 104, 043103 (2021)

Parameters

- **`probe_freq`** (*`float`*) – Frequency of the probing field, in Hz
- **`probe_elem`** (*`float`*) – Dipole moment of the probing transition, in units of a_0*e
- **`density`** (*`float`*) – Atomic density, in m^(-3)

Returns

kappa – in units of MHz/m

Return type

float

rydiqule.sensor_utils.draw_diagram

`rydiqule.sensor_utils.`**`draw_diagram`**(*sensor:* Sensor, *include_dephasing: bool = True*) → *ED*

Draw a matplotlib plot that shows the energy level diagram, couplings, and dephasing paths.

To show the plot, call `plt.show()`. If in a jupyter notebook, this is handled automatically.

Diagram has horizontal lines for the energy levels (spacing not to scale). Integer labels refer to the internal indexing for each state. If sensor is of type *`Cell`*, will also add text labels to each state of the quantum numbers.

Solid arrows between states are couplings defined with a non-zero Rabi frequency. Dashed arrows between states are couplings defined with a dipole moment.

Wiggly arrows between states denote a dephasing pathway. Opacity represents strength of dephasing relative to the largest specified dephasing, where fully opaque is the largest dephasing.

Parameters

- **`sensor`** (*`Sensor`*) – Sensor object to diagram.
- **`include_dephasing`** (*`bool, optional`*) – Whether to plot dephasing paths. Default is `True`.

Returns

Diagram handle

Return type

`ED`

rydiqule.sensor_utils.generate_eom

`rydiqule.sensor_utils.`**`generate_eom`**(*hamiltonian: ndarray*, *gamma_matrix: ndarray*, *remove_ground_state: bool = True*, *real_eom: bool = True*) → Tuple[ndarray, ndarray]

Create the optical bloch equations for a hamiltonian and decoherence matrix using the Lindblad master equation.

Parameters

- **`hamiltonian`** (*`numpy.ndarray`*) – Complex array representing the Hamiltonian matrix of the system, the matrix should be of shape `(*l, n, n)`, where n is the basis size and l is the shape of the stack of hamiltonians. For example, if the hamiltonian varies in 2 parameters l might be `(10, 10)`.
- **`gamma_matrix`** (*`numpy.ndarray`*) – Complex array representing the decoherence matrix of the system, the matrix should be of size `(n, n)`, where n is the basis size.
- **`remove_ground_state`** (*`bool, optional`*) – Remove the ground state from the equations of motion using population conservation. Setting to `False` is intended for internal use only and is not officially supported. See *`remove_ground()`* for details.
- **`real_eom`** (*`bool, optional`*) – Transform the equations of motion from the complex basis to the real basis. Setting to `False` is intended for internal use only and is not officially supported Seee *`make_real()`* for details.

Returns

- **equations** (*numpy.ndarray*) – The array representing the Optical Bloch Equations (OBEs) of the system. The shape will be `(*l, n^2-1, n^2-1)` if `remove_ground_state` is `True` and `(*l, n^2, n^2)` otherwise. The datatype will be `np.float64` if `real_eom` is `True` and `np.complex128` otherwise.
- **const** (*numpy.ndarray*) – Array of which defines the constant term in the linear OBEs. The shape will be `(*l, n^2-1)` if `remove_ground_state` is `True` and `(*l, n^2)` otherwise. The datatype will be `np.float64` if `real_eom` is `True` and `np.complex128` otherwise.

Raises

`ValueError` – If the shapes of gamma_matrix and hamiltonian are not matching: or not square in the last 2 dimensions

Examples

```
>>> ham = np.diag([1,-1])
>>> gamma = np.array([[.1, 0],[.1,0]])
>>> print(ham.shape)
>>> eom, const = rq.generate_eom(ham, gamma)
>>> print(eom)
>>> print(const.shape)
(2, 2)
[[-0.1  2.   0. ]
 [-2.  -0.1  0. ]
 [ 0.   0.  -0.1]]
(3,)
```

This also works with a “stack” of multiple hamiltonians:

```
>>> ham_base = np.diag([1,-1])
>>> ham_full = np.array([ham_base for _ in range(10)])
>>> gamma = np.array([[.1, 0],[.1,0]])
>>> print(ham_full.shape)
>>> eom, const = rq.generate_eom(ham_full, gamma)
>>> print(eom.shape)
>>> print(const.shape)
(10, 2, 2)
(10, 3, 3)
(10, 3)
```

rydiqule.sensor_utils.get_basis_transform

`rydiqule.sensor_utils.`**`get_basis_transform`**(*basis_size: int*) → Tuple[ndarray, ndarray]

Function that defines the basis transformation matrix u and its inverse u_i, between the real and complex basis.

This matrix u implements that the $\rho[j,i] \rightarrow Re(\rho[j,i])$ and $\rho[i,j] \rightarrow Im(\rho[j,i])$.

The transformation is not quite unitary, due to the asymmetry of the factors of 1/2.

Parameters

`basis_size` (*`int`*) – Size of the basis to generate transformations for.

Returns

- **u** (*numpy.ndarray*) – Forward transformation matrix.
- **u_inv** (*numpy.ndarray*) – Inverse transformation matrix.

Raises

`ValueError` – If `basis_size` does not match current basis.

rydiqule.sensor_utils.get_rho_ij

`rydiqule.sensor_utils.`**`get_rho_ij`**(*sols: Union[ndarray,* Solution*], i: int, j: int*) → ndarray

For a given density matrix solution, retrieve a specific element of the density matrix.

Assumes the ground state of the solution is eliminated (as per *`remove_ground()`*), and assumes Rydiqule's nominal state ordering of the Density Vector (per *`make_real()`*).

Parameters

- **`sols`** (numpy.ndarray or *`Solution`*) – Solutions to extract the matrix element for. Can be either the solution object returned by the solve or an N-D array representing density vectors, with ground state removed, and written in the totally real equations.
- **`i`** (*`int`*) – density matrix index i
- **`j`** (*`int`*) – density matrix index j

Returns

Array of rho_ij values. Will be of type float when `i==j`. Will be of type complex128 when `i!=j`.

Return type

numpy.ndarray

Examples

```
>>> sols = np.arange(180).reshape((4,5,3,3))
>>> print(sols.shape)
>>> rho_01 = rq.get_rho_ij(sols, 0,1)
>>> print(rho_01.shape)
>>> print(rho_01[0,0])
(4, 5, 3, 3)
(4, 5, 3)
[0.-1.j 3.-4.j 6.-7.j]
```

rydiqule.sensor_utils.get_rho_populations

`rydiqule.sensor_utils.`**`get_rho_populations`**`(`*sols: Union[ndarray,* Solution*]*`)` → ndarray

For a given density matrix solution, return the diagonal populations.

Note that rydiqule's convention for removing the ground state forces population conservation, ie the sum of these populations will be 1.

Parameters

`sols` (numpy.ndarray or *`Solution`*) – Solutions to extract the matrix element for. Can be either the solution object returned by the solve or an N-D array representing density vectors, with ground state removed, and written in the totally real equations.

Returns

Populations of the density matrices. Will have same shape as input solutions, with the last dimension reduced to the basis size.

Return type

numpy.ndarray

rydiqule.sensor_utils.make_real

`rydiqule.sensor_utils.`**`make_real`**`(`*equations: ndarray*, *constant: ndarray*, *ground_removed: bool = True*`)` → Tuple[ndarray, ndarray]

Converts equations of motion from complex basis to real basis.

Changes the density vector equation for p_ij into the Re[p_ij] equation and changing the density vector equation for p_ji into the equation for Im[p_ij].

Parameters

- **`equations`** (*`numpy.ndarray`*) – Complex equations of motion.
- **`constant`** (*`numpy.ndarray`*) – RHS of the equations of motion.
- **`ground_removed`** (*`bool, optional`*) – Indicates if `equations` has had the ground state removed. Default is `True`.

Returns

- **real_eqns** (*numpy.ndarray*) – EOMs in real basis.
- **real_const** (*numpy.ndarray*) – RHS of EOMs in real basis.

rydiqule.sensor_utils.remove_ground

`rydiqule.sensor_utils.`**`remove_ground`**(*equations: ndarray*) → Tuple[ndarray, ndarray]

Remove the ground state from the equations of motion using population conservation.

Population conservation enforces

$$\rho_{(0,0)} = 1 - \sum_{i=1}^{n-1} \rho_{(i,i)}$$

We use this equation to remove the EOM for rho_00 and enforce population conservation in the steady state.

Parameters

`equations` (*`numpy.ndarray`*) – array of shape (n^2, n^2) representing the equations of motion of the system, where n is the number of basis states.

Returns

The modified equations of shape (n^2-1, n^2-1)

Return type

numpy.ndarray

rydiqule.sensor_utils.scale_dipole

`rydiqule.sensor_utils.`**`scale_dipole`**(*dipole: Union[float, ndarray]*) → Union[float, ndarray]

Scale a dipole matrix from units of a0*e to Mrad/s when multiplied by a field in V/m.

Parameters

`dipole` (*`float or numpy.ndarray`*) – Array of dipole moments in units of a0*e. These are the default units used by ARC.

Returns

Scaled array in units of (Mrad/s)/(V/m)

Return type

numpy.ndarray

6.1.10 rydiqule.solvers

Steady-state solvers of the Optical Bloch Equations.

Functions

`solve_steady_state`(sensor[, doppler, ...])	Finds the steady state solution for a system characterized by a sensor.
`steady_state_solve_stack`(eom, const)	Helper function which returns the solution to the given equations of motion

rydiqule.solvers.solve_steady_state

`rydiqule.solvers.`**`solve_steady_state`**(*sensor:* Sensor, *doppler:* bool *= False*, *doppler_mesh_method:* Optional*[*Union*[*UniformMethod, SplitMethod, DirectMethod*]] = None*, *sum_doppler:* bool *= True*, *weight_doppler:* bool *= True*, *n_slices:* Optional*[*int*] = None*) → *Solution*

Finds the steady state solution for a system characterized by a sensor.

If insuffucent system memory is available to solve the system in a single call, system is broken into "slices" of manageable memory footprint which are solved indivudually. This slicing behavior does not affect the result. Can be performed with or without doppler averging.

Parameters

- **`sensor`** (*`Sensor`*) – The sensor for which the solution will be calculated.
- **`doppler`** (*`bool, optional`*) – Whether to calculate the solution for a doppler-broadened gas. If `True`, only uses dopper brodening defined by `kvec` parameters for couplings in the `sensoe`, so setting this `True` without `kvec` definitions will have no effect. Default is `False`.
- **`(dict`** (*`doppler_mesh_method`*) – If not `None`, should be a dictionary of meshing parameters to be passed to *`doppler_classes()`*. See *`doppler_classes()`* for more information on supported methods and arguments. If `None, uses the default doppler meshing. Default is `None`.
- **`optional)`** – If not `None`, should be a dictionary of meshing parameters to be passed to *`doppler_classes()`*. See *`doppler_classes()`* for more information on supported methods and arguments. If `None, uses the default doppler meshing. Default is `None`.
- **`sum_doppler`** (*`bool`*) – Whether to average over doppler classes after the solve is complete. Setting to `False` will not perform the sum, allowing viewing of the weighted results of the solve for each doppler class. In this case, an axis will be prepended to the solution for each axis along which doppler broadening is computed. Ignored if `doppler=False`. Default is `True`.
- **`weight_doppler`** (*`bool`*) – Whether to apply weights to doppler solution to perform averaging. If `False`, will **not** apply weights or perform a doppler_average, regardless of the value of `sum_doppler`. Changing from default intended only for internal use. Ignored if `doppler=False` or `sum_doppler=False`. Default is `True`.
- **`n_slices`** (*`int or None, optional`*) – How many sets of equations to break the full equations into. The actual number of slices will be the largest between this value and the minumum number of slices to solve the system without a memory error. If `None`, uses the minimum number of slices to solve the system without a memory error. Detailed information about slicing behavior can be found in `matrix_slice()`. Default is `None`.

Notes

Note: If decoherence values are not sufficiently populated in the sensor, the resulting equations may be singular, resulting in an error in `numpy.linalg`. This error is not caught for flexibility, but is likely the culprit for `numpy.linalg` errors encountered in steady-state solves.

Note: The solution produced by this function will be expressed using rydiqule's convention of converting a density matrix into the real basis and removing the ground state to improve numerical stability.

Returns
A bunch-type object contining information about the solution. Presently, only attribute "rho" is added to the solution, corresponding to the density matrix of the steady state solution. Will include solutions to all parameter value combinations if array-like parameters are specified.

Return type
Solution

Examples

A basic solve for a 3-level system would have a "density matrix" solution of size 8 (3^2-1)

```
>>> s = rq.Sensor(3)
>>> s.add_coupling((0,1), detuning = 1, rabi_freqency=1)
>>> s.add_coupling((1,2), detuning = 2, rabi_freqency=2)
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_steady_state(s)
>>> print(type(sol))
>>> print(type(sol.rho))
>>> print(sol.rho.shape)
<class 'rydiqule.sensor_solution.Solution'>
<class 'numpy.ndarray'>
(8,)
```

Defining an array-like parameter will automatically calculate the density matrix solution for every value. Here we use 11 values, resulting in 11 density matrices. The `axis_labels` attribute of the solution can clarify which axes are which.

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((0,1), detuning = det, rabi_freqency=1)
>>> s.add_coupling((1,2), detuning = 2, rabi_freqency=2)
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_steady_state(s)
>>> print(sol.rho.shape)
>>> print(sol.axis_labels)
(11, 8)
['(0,1)_detuning']
```

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((0,1), detuning = det, rabi_freqency=1)
>>> s.add_coupling((1,2), detuning = det, rabi_freqency=2)
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_steady_state(s)
>>> print(sol.rho.shape)
>>> print(sol.axis_labels)
```

(continues on next page)

(continued from previous page)

```
(11, 11, 8)
['(0,1)_detuning', '(1,2)_detuning']
```

If the solve uses doppler broadening, but not averaging for doppler is specified, there will be a solution axis corresponding to doppler classes.

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((0,1), detuning = det, rabi_freqency=1)
>>> s.add_coupling((1,2), detuning = 2, rabi_freqency=2, kvec=(1,0,0))
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_steady_state(s, doppler=True, sum_doppler=False)
>>> print(sol.rho.shape)
>>> print(sol.axis_labels)
(561, 11, 8)
['doppler_0', '(0,1)_detuning']
```

rydiqule.solvers.steady_state_solve_stack

`rydiqule.solvers.`**`steady_state_solve_stack`**(*eom: ndarray*, *const: ndarray*) → ndarray

Helper function which returns the solution to the given equations of motion

Solves an equation of the form $\dot{x} = Ax + b$, or a set of such equations arranged into stacks. Essentially just wraps numpy.linalg.solve(), but included as its own function for modularity if another solver is found to be worth invesitigating.

Parameters

- **eom** (*numpy.ndarray*) – An square array of shape `(*l,n,n)` representing the differential equations to be solved. The matrix (or matrices) A in the above formula.
- **const** (*numpy.ndarray*) – An array or shape `(*l,n)` representing the constant in the matrix form of the differential equation. The constant b in the above formula. Stack shape `*l` must be consistent with that in the `eom` argument

Returns

A 1xn array representing the steady-state solution of the differential equation

Return type

numpy.ndarray

6.1.11 rydiqule.timesolvers

Solvers for time domain analysis with an arbitrary RF field

Functions

generate_eom_time(hamiltonians_time)	Generates the Optical Bloch Equations for just the rf terms.
solve_eom_stack(eoms_base, const, ...)	Solve a stack of equations of motion with shape `(*l, n, n)` in the time domain.
solve_time(sensor, end_time, num_pts[, ...])	Solves the response of the optical sensor in the time domain given the its time-dependent inputs

rydiqule.timesolvers.generate_eom_time

`rydiqule.timesolvers.`**`generate_eom_time`**(*hamiltonians_time: ndarray*) → Tuple[ndarray, ndarray]

Generates the Optical Bloch Equations for just the rf terms. Uses the convention of the hamiltonian_rf return of the get_time_hamiltonian function. The equations of motion returned are assumed to be used in conjunction with an electric field.

Parameters

`hamiltonians_time` (*`numpy.ndarray`*) – A matrix of shape (basis_size, basis_size), where the off-diagonal terms (i,j) are the dipole matrix elements in e a_b of the transition coupling state i to state j.

Returns

- **numpy.ndarray** (*Part of the Optical Bloch Equations corresponding to time_dependent couplings.*) – To produce equations to solve, these values must be multiplied by an electric field in V/m.
- **numpy.ndarray** (*Constant term of the time-dependent portion of the equations*) – of motion. Same units as the equations themselves.

rydiqule.timesolvers.solve_eom_stack

`rydiqule.timesolvers.`**`solve_eom_stack`**(*eoms_base: ndarray*, *const: ndarray*, *eom_time_r: ndarray*, *const_r: ndarray*, *eom_time_i: ndarray*, *const_i: ndarray*, *time_inputs: List[Callable[[float], float]]*, *t_eval: ndarray*, *init_cond: ndarray*, *solver*, ***kwargs*) → ndarray

Solve a stack of equations of motion with shape `(*l, n, n)` in the time domain.

Companion function to *`solve_time()`*, but can be invoked on its for equations already formatted.

Parameters

- **`eoms_base`** (*`numpy.ndarray`*) – Array of shape `(*l, n, n)` represnting the part of equations of motion of the system which do not respond to external fields.
- **`const`** (*`numpy.ndarray`*) – constant term of shape (n,) added in differential equations. Typically generated by *`generate_eom()`*.
- **`eoms_time_r`** (*`list[numpy.ndarray]`*) – list of arrays of shape `(basis_size^2-1, basis_size^2-1)` representing the parts of the OBEs with a real-valued time-dependence. In the solver, this array will be multiplied by a time-dependent rabi frequency. Typically a matrix of mostly zeros, with non-zero terms corresponding to a particular time-dependent coupling
- **`const_r`** (*`numpy.ndarray`*) – Constant term of shape (n,) added in a real time-dependent portion of differential equations. Typically generated by `generate_eom_time()`.
- **`eoms_time_i`** (*`numpy.ndarray`*) – list of arrays of shape `(basis_size^2-1, basis_size^2-1)` representing the parts of the OBEs with an imaginary-valued time-dependence. In the solver, this array will be multiplied by a time-dependent rabi frequency.
- **`const_i`** (*`numpy.ndarray`*) – constant term of shape (n,) added in an imaginary time-dependent portion of differential equations. Typically generated by `generate_eom_time()`.
- **`t_eval`** (*`numpy.ndarray`*) – 1-D array of times, in microseconds, at which to evaluate the solution. Does not affect evaluations in the solve.

- **time_inputs** (*list[function float->float]*) – List of functions which represent the rabi frequency of a field as a function of time. list length should be identical to the length of obes_time. In the solver, the *i* th time input will be evaluated at time *t* and multiplied by the *i* th entry of obes_time.
- **tuple(float)** (*time_range*) – Pair of values represent the start and end time, in microseconds, of the simulation.
- **init_cond** (numpy.ndarray or `None`, optional) – Density matrix representing the initial state of the system. If specified, the shape should be either `(n)` in the case of a single initial condition for all parameter values, or should be of shape `(*l, n)` matching the output shape of a steady state solve if the initial condition may be different for different combinations of parameters. If `None`, will solve the problem in the steady state with all time-dependent fields "off" and use the solution as the initial condition for the time behavior. Other possible manual options might include a matrix populated by zeros representing the entire population in the ground state. Defaults to `None`.

Returns
Flattened solution array corresponding to time points.

Return type
numpy.ndarray

rydiqule.timesolvers.solve_time

`rydiqule.timesolvers.`**`solve_time`**(*sensor:* Sensor, *end_time: float*, *num_pts: int*, *init_cond: Optional[ndarray] = None*, *doppler: bool = False*, *doppler_mesh_method: Optional[Union[*UniformMethod, SplitMethod, DirectMethod*]] = None*, *sum_doppler: bool = True*, *n_slices: Optional[int] = None*, *solver: Union[Callable, Literal['scipy', 'nbkode', 'cyrk', 'nbrk']] = 'scipy'*, ***kwargs*) → *Solution*

Solves the response of the optical sensor in the time domain given the its time-dependent inputs

If insuffucent system memory is available to solve the system all at once, system is broken into "slices" of manageable memory footprint which are solved indivudually. This slicing behavior does not affect the result. All couplings that include a "time_dependence" argument will be solved in the time domain.

A number of solver backends work with rydiqule, but the default `"scipy"` ivp solver is the is recommended backend in almost all cases, as it is the most fully-featured and documented. Advanced users have the ablity to define their own solver backends by creating a function that follows the call signature for rydiqule timesolver backends. Additional arguments to the solver backend can be supplied with `**kwargs`.

Parameters

- **sensor** (*Sensor*) – The sensor object representing the atomic/laser arrangement of the system.
- **end_time** (*float*) – Amount of time, in microseconds, for which to simulate the system
- **num_pts** (*int*) – The number of points along the range `(0, end_time)` for which the solution is evaluated. This does not affect the number of funtion evaluations during the solve, rather the spacing of the points in the reported solution.
- **init_cond** (numpy.ndarray or `None`, optional) – Density matrix representing the initial state of the system. If specified, the shape should be either `(n)` in the case of a single initial condition for all parameter values, or should be of shape `(*l, n)` matching the output shape of a steady state solve if the initial condition may be different for different combinations of parameters. If `None`, will solve the problem in the steady state with all time-dependent fields "off" and use the solution as the initial condition for the time behavior. Other possible manual options might include a matrix populated by zeros representing the entire population in the ground state. Defaults to `None`.

- **doppler** (*bool, optional*) – Whether to account for doppler shift among moving atoms in the gas. If True, the solver will implicitly define a velocity distribution for particles in the cell, solve the problem for each velocity class, and return a weighted average of the results. Note that solving in this manner carries a substantial performance penalty, as each doppler velocity class is solved as its own problem. If solved with doppler, only axis specified by a `"kvec"' argument in one of the sensor couplings will be average over. The time solver currently supports doppler averaging in any number of spatial dimensions, up to the limit of 3 imposed by the macroscopic physical world. Defaults to `False`.
- **doppler_mesh_method** (*dict, optional*) – Dictionary that controls the doppler meshing method. Exact details of this are found in the documentation of `doppler_classes()`. Ignored if `doppler=False`. Default is `None`.
- **sum_doppler** (*bool, optional*) – Whether to average over doppler classes after the solve is complete. Setting to false will not perform the sum, allowing viewing of the weighted results of the solve for each doppler class. Ignored if `doppler=False`. Default is `True`.
- **n_slices** (*int or None, optional*) – How many sets of equations to break the full equations into. The actual number of slices will be the largest between this value and the minumum number of slices to solve the system without a memory error. If `None`, solver uses the minimum number of slices required to solve without a `memoryError`. Defaults to None.
- **solver** (*{"scipy", "nbkode", "cyrk", "nbrk"} or callable*) – The backend solver used to solve the ivp generated by the sensor. All string values correspond to backend solvers built in to rydiqule. Valid string values are:
  - **"scipy": Solves equations with `scipy.integrate.solve_ivp()`.**
    The default, most stable, and well-supported option.
  - **"nbkode": Solves equations with the jit-compiled Runge-Kutta-45 solver in numbakit-ode.**
    May be faster for very long time solves with many timesteps.
  - **"cyrk": Solves jit-compiled equations with a cython compiled RK solver from `CyRK`.**
    Due to some jit compilation, only faster for moderate length problems (ie problems with a moderate number of required time steps).
  - **"nbrk": Solves equations with the jit-compiled RK solver from `CyRK`.**
    Due to extensive jit compilation, only faster for very long solves (ie problems with a large number of required time steps).

  Additionally, can be specified with a callable that matches rydiqule's time-solver convention, enabling using a custom solver backend.

  Note: Unless otherwise noted, backends other than scipy are considered experimental. Issues with their use are considered features not fully implemented rather than bugs.

- ****kwargs** (*Additional keyword arguments passed to the backend solver. See documentation of the*) – relevant solver (i.e. `scipy.integrate.solve_ivp()`) for details and supported arguments.

Returns

A bunch-type object contining information about the solution. Timesolver specific attributes are `t` and `init_cond`, corresponding respectively to the times at which the solution is sampled and the initial conditions used for the solve.

Return type

Solution

CHAPTER

SEVEN

DEVELOPER DOCUMENTATION

These pages contain documentation relevant to the development of rydiqule. If you with to work on the source code of rydiqule, details relating to policies and tools can be found here.

7.1 Unit Tests

Rydiqule comes bundled with a suite of unit tests that confirm basic functionality of the various components and checks for robustness to erroneous or unexpected arguments. We strive to follow the testing methodology and practices employed by numpy. We agree with the stipulation made there, that

> "Long experience has shown that by far the best time to write the tests is before you write or change the code - this is test-driven development"

7.1.1 pytest

Rydiqule takes advantage of the `pytest` testing framework to run unit and integration tests of the code base. The full test suite is run from the project base directory with the command

```
pytest tests/
```

This command will run all tests in the `tests/` subdirectory. These tests cover a wide range of functionality as well as a number of representative integrations that demonstrate how the code can be used to generate end results.

Marks

The tests are marked based on the type of test that is being performed, and `pytest` can be told to only run certain tests. For example, this command will only run tests relating to the steady state solving functionality:

```
pytest -m steady_state
```

You can also exclude a specific group of tests. For example, this command will exlcude tests marked as slow.

```
pytest -m "not slow"
```

Marks specifications can be combined using standard boolean keywords as well. The following will run all the time tests that are not slow.

```
pytest -m "time and not slow"
```

The available marks can be listed using `pytest --markers`. The markers we use are

Table 7.1: Markers

Marker	Description
slow	Marks a test as taking a long time to run
high_memory	Marks a test needing a lot of RAM
steady_state	Marks a test as using the steady-state solver
time	Marks a test as using the time solver
doppler	Marks a test that incorporates Doppler averaging.
exp	Marks a test that represents a full experiment.
util	Marks a test of the ancillary utilties.
structure	Marks a test of the definition of the atomic system.
dev	Used to temporarily mark a single test that is being developed so it can run independently.

Coverage

If you install the `pytest-cov` plugin, you can check code coverage of the tests by modifying the command to read.

```
pytest --cov=rydiqule tests/
```

Durations

If you want to see which tests take the longest to complete, you can use the `--durations=n` flag to give the `n` longest time tests:

```
pytest --durations=3 tests/
```

Settings the `durations` flag to 0 will cause pytest to report the time taken for all tests run.

7.2 Type Hinting

Rydiqule employs the optional type hinting capabilities of python. These type annotations are not checked or enforced at runtime by python itself. Rather, they provide hints to fellow programmers and users about the types of function arguments, return types, and class variables.

We use the mypy static type checking library to read these hints and catch type errors within the code base. To run this check locally, install the `mypy` python package and run the following command from the package root directory.

```
mypy
```

This command will automatically read configuration options set in the `mypy.ini` file. Further optional flags can be passed to the command to override or add optional behaviors. Initial run of the mypy takes some time, however subsequent runs take advantage of local caching to increase analysis speed. Using the mypy daemon mode can further increase analysis speed if necessary.

An html report of the mypy coverage can be generated using the following command.

```
mypy --html-report .mypy_report
```

This command will store the html pages in the specified directory `.mypy_report`. Note that this command takes a long time to run every time, as it cannot use the cache.

7.3 Linting

We use the `flake8` linting package to help enforce code style and consistent readability. It can be run locally from the project root folder by calling the command `flake8`. It pulls options for running the command from the `.flake8` configuration file.

If you intend to work on the rydiqule codebase, it is good practice to incorporate automatic linting within your code editor.

Linting can be disabled for a single line by using the `# noqa` tag at the end of the line.

7.4 Building the Documentation

The Rydiqule documentation can be built locally from the source repository using `sphinx`. To do so, you will need to install the `sphinx` and `sphinx-rtd-theme` packages.

7.4.1 html

An html webpage version of the documentation formatted in the read-the-docs style can be made by running the following command from the `docs/` subdirectory.

```
make html
```

The output will be located in the `docs/build/html/` subdirectory. The home page is `index.html`. The html documentation has the best formatting by default and is the easiest to use.

7.4.2 latexpdf

A pdf version of the documentation can be built using

```
make latexpdf
```

The output will be located in `docs/build/latex` and is called `rydiqule.pdf`. Note that building the pdf requires `perl` and a functioning latex installation with the `latexmk` package. You will also require the GNU FreeFont collection. On Windows, these can be installed manually at the system level or via the MikTeX package `gnu-freefont`. This build also requires a great many other latex packages in addition to `latexmk`. It is easiest to install these packages on the fly as needed, if your latex distribution supports that.

Given the difficulty of building this type of documentation, we attempt to include an updated pdf with each relase. It is locaed in the `docsbuildlatex` directory.

7.4.3 epub

There is also the ability to build the documentation in the EPUB format, if desired.

```
make epub
```

```
%matplotlib inline
```

```
%load_ext autoreload
%autoreload 2
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.transform import Rotation as R
```

```
import rydiqule as rq
from rydiqule.sensor_utils import get_rho_ij
```

CHAPTER

EIGHT

3-PHOTON RYDBERG EIT

We demonstrate three photon coherent excitations using the system studied in Taicharoen et. al. PRA 063427 (2019). This is a rubidium vapor with a $5S_{1/2} \rightarrow 5P_{3/2} \rightarrow 5D_{5/2} \rightarrow 28F_{7/2}$ excitation pathway, with corresponding optical fields of 780 nm, 776 nm, and 1260 nm. These fields are labelled probe, dressing, and coupling, respectively.

Here we demonstrate using rydiqule to solve this system under three conditions:

1) Cold atoms
2) Warm atoms, colinear optical beams
3) Warm atoms, doppler-free angles

Because there are three optical fields, the basic coherent feature observed, on resonance, is expected to the absorptive (rather than transmissive like EIT). Going to warm atoms, this feature broadens signficantly, and other coherent features can arise at non-zero detunings of the fields. Going to a Doppler-free excitation in warm atoms, we find that a transmissive feature is observed on resonance. This feature is significantly narrower than those observed in the colinear case.

8.1 Doppler-free, 3 photon excitation

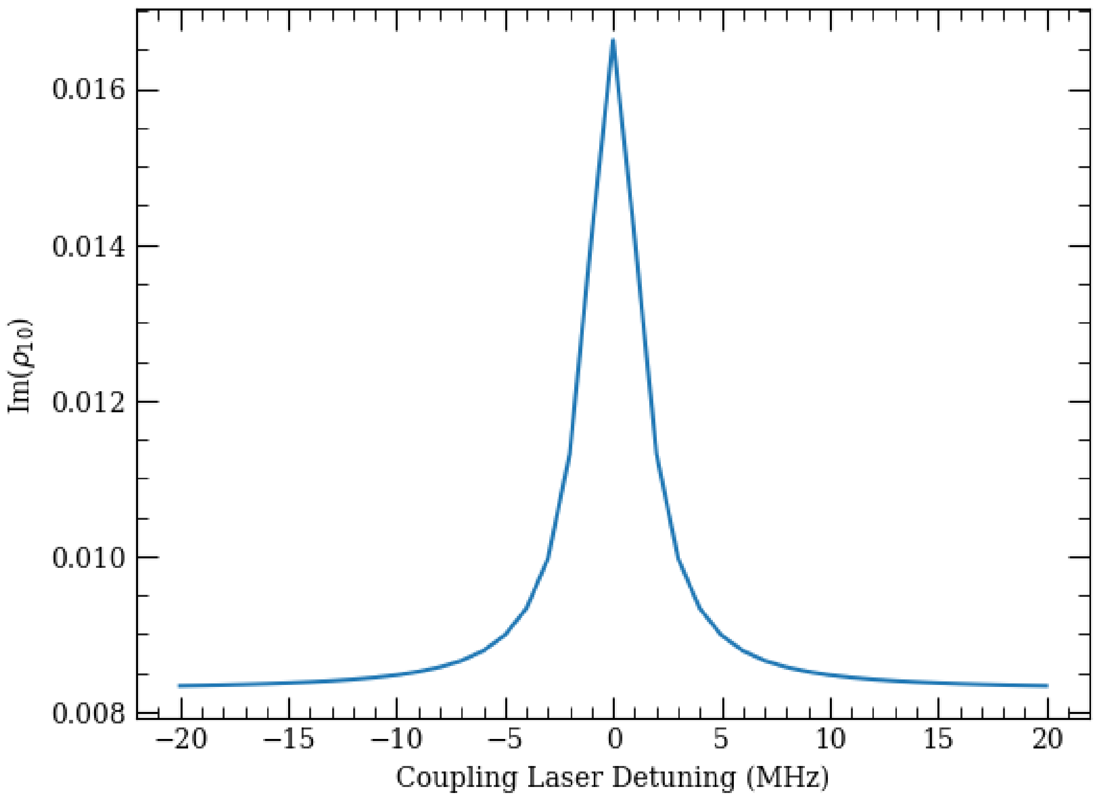
With all three fields resonant, we expect to see Electromagnetically-Induced-Absorption (EIA) instead of EIT.

```
detunings = np.linspace(-20,20,41)

probe = {'states':(0,1), 'rabi_frequency':2*np.pi*0.1,'detuning':2*np.pi*0}
dress = {'states':(1,2), 'rabi_frequency':2*np.pi*2}
couple = {'states':(2,3), 'rabi_frequency':2*np.pi*2, 'detuning':2*np.pi*detunings}

basis_size = 4
gam = np.zeros((basis_size,basis_size),dtype=np.float64)
gam[1,0] = 6
gam[2,1] = 0.66
gam[3,2] = 10e-3
gamma_matrix = 2*np.pi*gam

sensor = rq.Sensor(basis_size)
sensor.add_couplings(probe,couple)
sensor.set_gamma_matrix(gamma_matrix)
```

```
dress['detuning'] = 2*np.pi*0
sensor.add_couplings(dress)
sols = rq.solve_steady_state(sensor)
```

```
fig, ax = plt.subplots()
ax.plot(detunings, get_rho_ij(sols.rho,1,0).imag)
ax.set_xlabel("Coupling Laser Detuning (MHz)")
ax.set_ylabel(r"Im($\rho_{10}$)")
```

```
Text(0, 0.5, 'Im($\rho_{10}$)')
```



8.2 Colinear 3-photon Excitation with Doppler Averaging

We can take our three fields and configure them in the (+,-,-) configuration of Taicharoen (2019). We will need to do Doppler averaging along the colinear axis to get the result. The magnitude of a field's kvector is defined such that it is the magnitude of the Doppler shift associated with the most probable speed of the Maxwell-Boltzmann distribution ($v_P \equiv \sqrt{2k_BT/m}$, where k_B is Boltzmann's constant, T is the gas temperature, and m is the atomic mass). It should have units of Mrad/s like all other specifies quantities.

The following reproduces Figure 2b from Taicharoen et. al. PRA 063427 (2019). Note that having all fields resonant results in an EIA feature, but it now much broader than the Doppler-free case above. If the dressing field is detuning, EIT features are observed.

```
detunings = np.linspace(-200,200,201)

kp = 2*np.pi/780e-3*np.array([1,0,0])
kd = 2*np.pi/776e-3*np.array([-1,0,0])
kc = 2*np.pi/1260e-3*np.array([-1,0,0])
vP = 242.387 # m/s

###
probe = {'states':(0,1), 'rabi_frequency':2*np.pi*10, 'kvec':vP*kp, 'detuning': 0}
dress = {'states':(1,2), 'rabi_frequency':2*np.pi*25, 'kvec':vP*kd}
couple = {'states':(2,3), 'rabi_frequency':2*np.pi*18, 'detuning':2*np.
↪pi*detunings, 'kvec':vP*kc}
###

n = 4
sensor = rq.Sensor(n)
sensor.add_decoherence((1,0), 2*np.pi*6)
```

(continues on next page)

(continued from previous page)

```
sensor.add_decoherence((2,1), 2*np.pi*0.66)
sensor.add_decoherence((3,2), 2*np.pi*10e-3)

sensor.add_couplings(probe,couple)
```

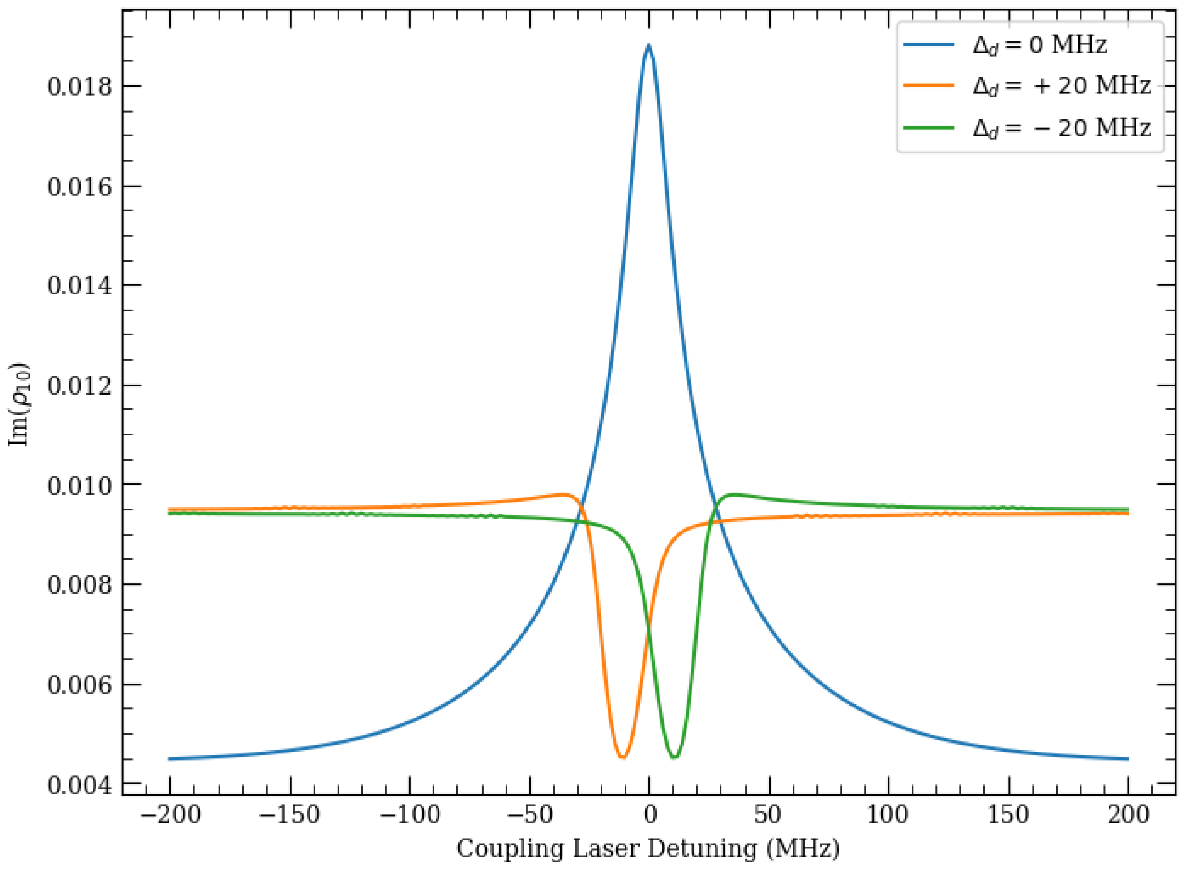
```
dress['detuning'] = 2*np.pi*0
sensor.add_couplings(dress)
sols0 = rq.solve_steady_state(sensor,doppler=True)
```

```
dress['detuning'] = 2*np.pi*20
sensor.add_couplings(dress)
solsp20 = rq.solve_steady_state(sensor,doppler=True)
```

```
dress['detuning'] = -2*np.pi*20
sensor.add_couplings(dress)
solsm20 = rq.solve_steady_state(sensor,doppler=True)
```

```
fig, ax = plt.subplots(figsize=(8,6))
ax.plot(detunings, get_rho_ij(sols0.rho,1,0).imag, label="$\\Delta_d= 0$ MHz")
ax.plot(detunings, get_rho_ij(solsp20.rho,1,0).imag, label="$\\Delta_d= +20$ MHz")
ax.plot(detunings, get_rho_ij(solsm20.rho,1,0).imag, label="$\\Delta_d= -20$ MHz")
ax.set_xlabel("Coupling Laser Detuning (MHz)")
ax.set_ylabel(r"Im($\rho_{10}$)")
ax.legend()
```

```
<matplotlib.legend.Legend at 0x25c82005b50>
```



8.3 Doppler-Free Angles

We can take the same co-propagating system, and instead change the angles of the three beams such that $k_p + k_d + k_c \approx 0$.

We now need to do Doppler averaging in two orthogonal dimensions. Rydiqule automatically detects how many non-zero dimensions are present in the field kvectors and averages over the appropriate number of spatial dimensions.

```
detunings = np.linspace(-20,20,21)

kp = 2*np.pi/780e-3*np.array([1,0,0])
kd = 2*np.pi/776e-3*np.array([-1,0,0])
kc = 2*np.pi/1260e-3*np.array([-1,0,0])
vP = 242.387 # m/s

# rotate to doppler free angles
rd = R.from_euler('z',-35.964,degrees=True)
rc = R.from_euler('z',72.4718,degrees=True)
kdDF = rd.apply(kd)
kcDF = rc.apply(kc)

probe = {'states':(0,1), 'rabi_frequency':2*np.pi*10, 'kvec':vP*kp, 'detuning':0}
dress = {'states':(1,2), 'rabi_frequency':2*np.pi*25, 'kvec':vP*kdDF}
couple = {'states':(2,3), 'rabi_frequency':2*np.pi*18, 'detuning':2*np.
↪pi*detunings, 'kvec':vP*kcDF}

n = 4
sensor = rq.Sensor(n)
sensor.add_decoherence((1,0), 2*np.pi*6)
sensor.add_decoherence((2,1), 2*np.pi*0.66)
sensor.add_decoherence((3,2), 2*np.pi*10e-3)

sensor.add_couplings(probe,couple)
```

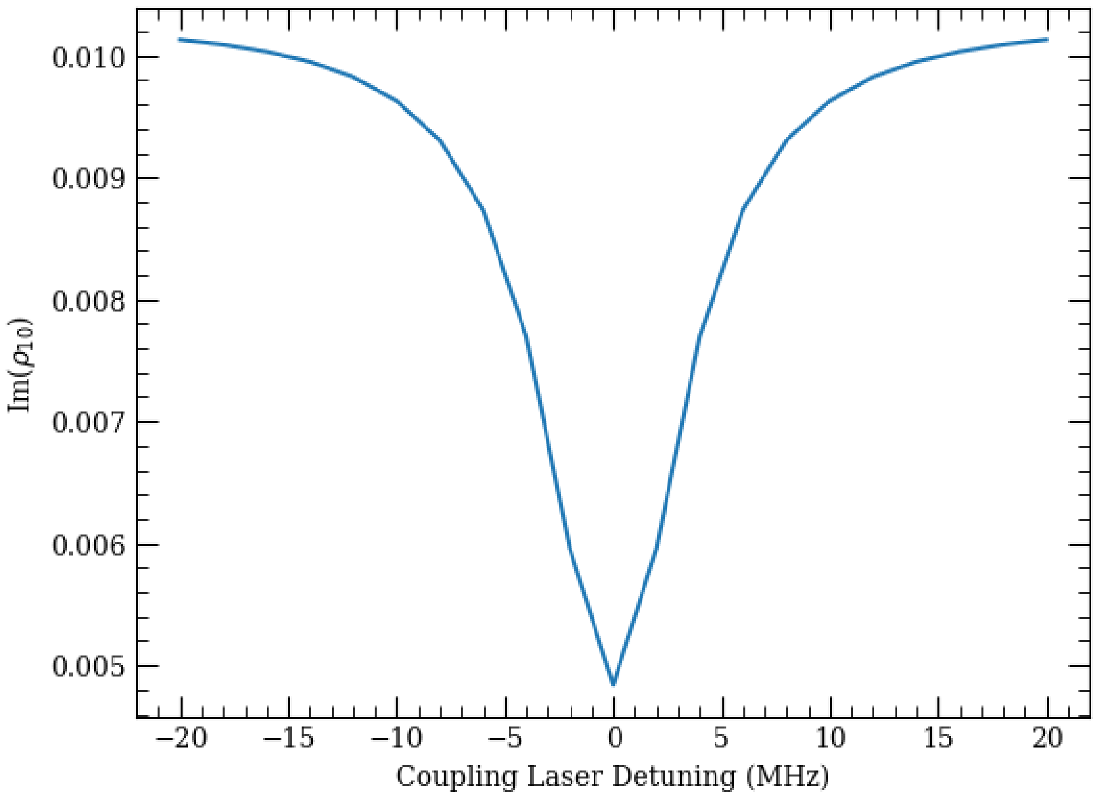
```
print('Residual fractional kvector sum due to round-off errors')
print((kp+kdDF+kcDF)/np.sqrt(kp.dot(kp)))
```

```
Residual fractional kvector sum due to round-off errors
[-1.41309045e-08 -4.99652620e-07  0.00000000e+00]
```

```
dress['detuning'] = 2*np.pi*0
sensor.add_couplings(dress)
sols0DF = rq.solve_steady_state(sensor,doppler=True)
```

```
fig, ax = plt.subplots()
ax.plot(detunings, get_rho_ij(sols0DF.rho,1,0).imag)
ax.set_xlabel("Coupling Laser Detuning (MHz)")
ax.set_ylabel(r"Im($\rho_{10}$)")
```

```
Text(0, 0.5, 'Im($\rho_{10}$)')
```



We observe a significantly narrower feature than in the collinear case, and it is now EIT instead of EIA.

The authors recognize financial support from the Defense Advanced Research Projects Agency (DARPA). Rydiqule has been approved for unlimited public release by DEVCOM Army Research Laboratory and DARPA. This software is released under the xx licence through the University of Maryland Quantum Technology Center. The views, opinions and/or findings expressed here are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

CHAPTER

NINE

CALCULATING SNR

Rydiqule contains the function `rydiqule.get_snr()` function that will take a Sensor or Cell and calculate the expected SNR for one axis of the solve. Below we demonstrate the use of this function to numerically confirm the analytic results of Meyer et. al. PRA 104, 043103 (2021) Eqs. 12 & 13:

$$\Omega_p^{(\mathrm{opt})} \approx \sqrt{\Gamma(2\gamma + \Gamma_r + \Gamma_c)}$$

$$\Omega_c^{(\mathrm{opt})} \approx \sqrt{2}\Omega_p^{(\mathrm{opt})}$$

These show the optical probe and coupling Rabi frequencies for resolving Rydberg state shifts in a 2-color Rydberg EIT measurement, in an optically-thin with no Doppler broadening.

```
import numpy as np
import rydiqule as rq
import matplotlib.pyplot as plt
```

```
%load_ext autoreload
%autoreload 2
```

Manually define representative kappa and eta constants for a Rb85 sensor. These are necessary to find the SNR in experimental units and must be supplied by the user when calculating using a Sensor. If using a Cell, these constants are automatically calculated and do not need to be passed to `get_snr`.

The definition of these numerical factors is found in Meyer et. al. PRA 104, 043103 (2021) Eqs. 5 & 7.

$$\kappa = \frac{\omega_p n \mu^2}{2c\epsilon_0 \hbar}$$

$$\eta = \sqrt{\frac{\omega \mu^2}{2c\epsilon_0 \hbar A}}$$

```
kappa = 28974.8787
eta = 0.00135882
```

9.1 1D Optimum

Here we demonstrate calculating the SNR for resolving a phase shift due to an RF Rydberg coupling vs probe Rabi frequency. We have chosen a far-detuned RF coupling to ensure Stark shifts are linear.

```
##Set up a simplified Rb Sensor
basis_size = 4
Rb_sensor = rq.Sensor(basis_size)

red_rabi = np.linspace(0.1,4,100)
blue_rabi = np.linspace(0.1,4,101)
```

(continues on next page)

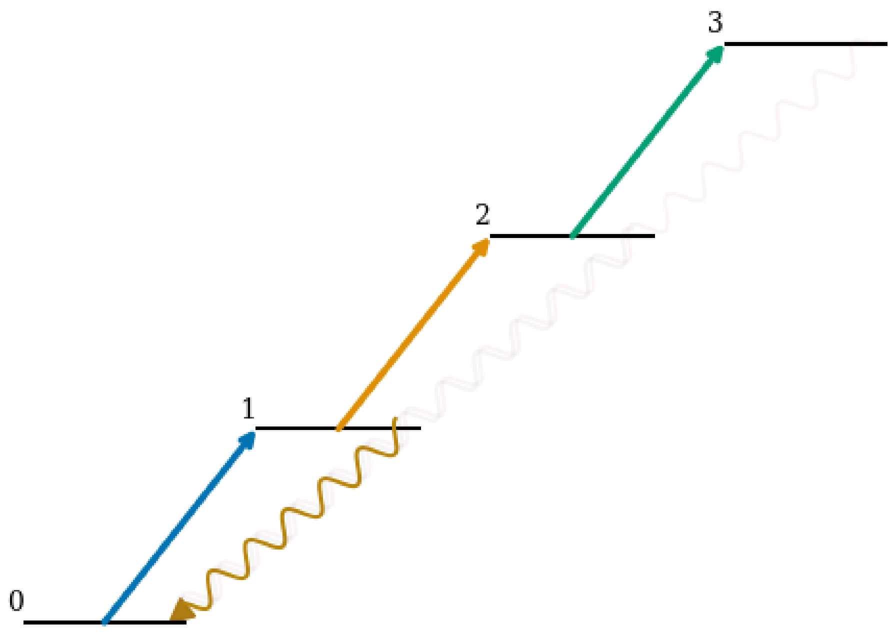
(continued from previous page)

```
blue_rabi_1 = 1
my_step = np.array([1, 1.1])
probe = {'states': (0,1), 'rabi_frequency': red_rabi, 'detuning': 0, 'label':
↪'probe'}
couple = {'states': (1,2), 'rabi_frequency':blue_rabi_1, 'detuning': 0, 'label':
↪'couple'}
rf = {'states': (2,3), 'rabi_frequency': my_step, 'detuning':20, 'label': 'rf'}

Rb_sensor.add_couplings(probe,couple, rf)

#simplify the gamma matrix to match predictions
gam = np.zeros((basis_size, basis_size))
gam[2,0] = 0.1
gam[3,0] = 0.1
gam[1,0] = 6.0
Rb_sensor.set_gamma_matrix(gam)
```

To calculate the SNR vs a specific parameter, that parameter must be list-like with at least two elements. So calculated vs RF Rabi frequency, we have specified two Rabi frequency values very close to each other to measure the local linear sensitivity. More values can be added to this list to see if sensitivity changes for larger changes in the parameter, which indicates nonlinear response.

```
rq.draw_diagram(Rb_sensor)
```

```
<rydiqule.energy_diagram.ED at 0x1e7117a4bb0>
```



We call `get_snr` with the Sensor to calculate with, the label of the swept parameter to calculate SNR against, the tuple of the probing transition to get measurable parameters from, which quadrature the probing field is being detected in, and the kappa and eta numerical factors.

```
snrs, param_mesh = rq.get_snr(Rb_sensor,optical_path_length= 0.0001, param_label =
↪'rf_rabi_frequency',
                              probe_tuple = (0,1), phase_quadrature = True,
                              kappa = kappa, eta = eta)
```

Using `Sensor.axis_labels()` we can identify which axis rydiqule has used for the swept parameters. This allows us to correctly index out the appropriate solutions for analysis. In particular, we need to index the sensitivity axis to get the sensitivity at the second RF Rabi frequency in the list (relative to the first).

```
#print the axis labels
Rb_sensor.axis_labels()
```

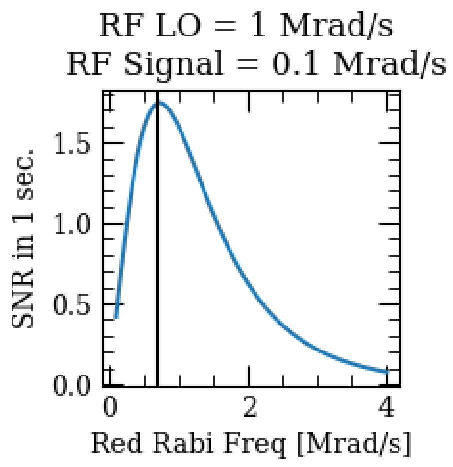
```
['probe_rabi_frequency', 'rf_rabi_frequency']
```

```
snrs_final = snrs[:,1]
param_mesh_final=np.array(param_mesh)[:,:,1]
```

We can plot the SNR as a function of probe rabi frequency. The vertical line represents the analytic optimum value for the probe Rabi frequency.

```
fix, ax = plt.subplots(figsize = (2,2))
ax.plot(param_mesh_final[0], snrs_final)
ax.set_xlabel("Red Rabi Freq [Mrad/s]")
ax.set_ylabel("SNR in 1 sec.")
ax.set_title('RF LO = 1 Mrad/s \n RF Signal = 0.1 Mrad/s')
ax.axvline(blue_rabi_1/np.sqrt(2),0,3, color = 'k')
```

```
<matplotlib.lines.Line2D at 0x1e7449efc70>
```



9.2 2D Optimum - Fnd Optimized Ω_p and Ω_c for best SNR

We can also calculate the SNR versus many different axis. Here we calculate versus both the probe and coupling Rabi frequencies.

```
couple = {'states': (1,2), 'rabi_frequency': blue_rabi, 'detuning': 0, 'label':
↪'couple'}

Rb_sensor.add_couplings(probe,couple, rf)
```

```
snrs, param_mesh = rq.get_snr(Rb_sensor, 100e-6, param_label = 'rf_rabi_frequency',
                              probe_tuple = (0,1), phase_quadrature = True,
                              kappa = kappa, eta = eta)
```

```
Rb_sensor.axis_labels()
```

```
['probe_rabi_frequency', 'couple_rabi_frequency', 'rf_rabi_frequency']
```

```
snrs_final = snrs[:,:,1]
param_mesh_final=np.array(param_mesh)[:,:,:,1]
```

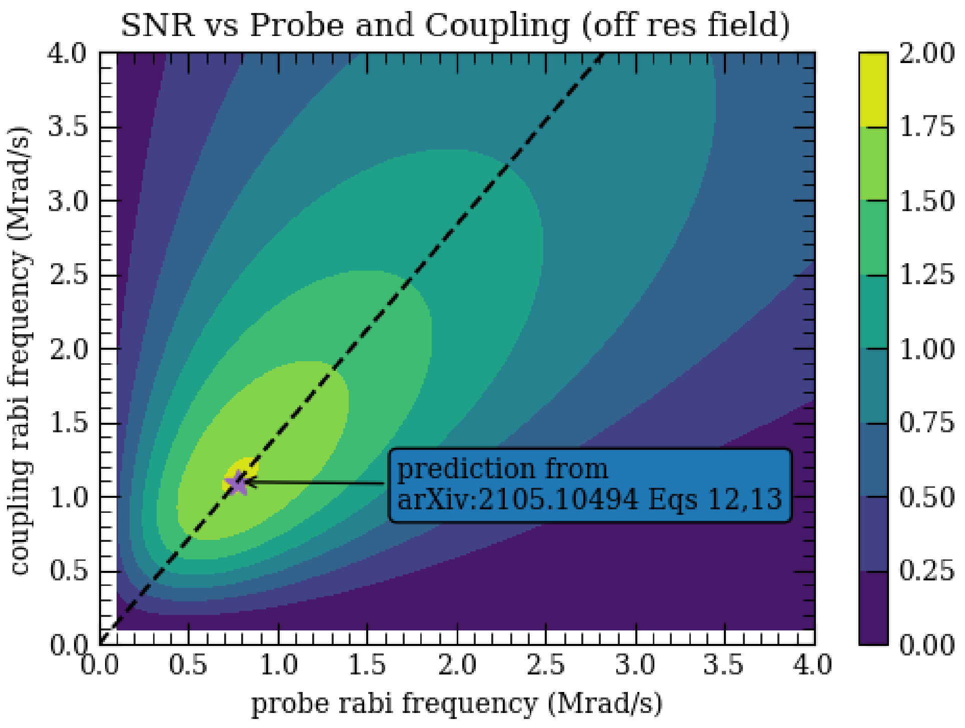
```
predictedOptimumProbe = np.sqrt(gam[1,0]*gam[2,0])
predictedOptimumCouple = np.sqrt(2*gam[1,0]*gam[2,0])
print(f'Predicted optimum probe Rabi frequency: {predictedOptimumProbe:.3f} Mrad/s
↪')
print(f'Predicted optimum coupling Rabi frequency: {predictedOptimumCouple:.3f}␣
↪Mrad/s')
```

```
Predicted optimum probe Rabi frequency: 0.775 Mrad/s
Predicted optimum coupling Rabi frequency: 1.095 Mrad/s
```

We plot the SNR versus both Rabi frequencies using a contour plot. We have overlaid the analytic predictions for the optimal SNR. Compare Figure 5(a) of Meyer et. al.

```
fig, ax = plt.subplots(figsize = (6,4))
CS = ax.contourf(param_mesh_final[0], param_mesh_final[1], snrs_final)
fig.colorbar(CS)
ax.set_xlabel('probe rabi frequency (Mrad/s)')
ax.set_ylabel('coupling rabi frequency (Mrad/s)')
ax.plot(predictedOptimumProbe, predictedOptimumCouple, '*', color = 'C4',␣
↪markersize = 10)
ax.plot([0,10,20], [0,np.sqrt(2)*10,np.sqrt(2)*20 ],'--', color = 'black')
ax.set_title("SNR vs Probe and Coupling (off res field)")
ax.set_ylim((0,4))
ax.set_xlim((0,4))

ax.annotate("prediction from\narXiv:2105.10494 Eqs 12,13",
            xy=(predictedOptimumProbe, predictedOptimumCouple), xycoords='data',
            xytext=(60,-10), textcoords='offset points',
            arrowprops=dict(arrowstyle='->'),
             bbox=dict(boxstyle='round')
            )
```

```
Text(60, -10, 'prediction fromnarXiv:2105.10494 Eqs 12,13')
```



The authors recognize financial support from the US Army and Defense Advanced Research Projects Agency (DARPA). Rydiqule has been approved for unlimited public release by DEVCOM Army Research Laboratory and DARPA. This software is released under the xx licence through the University of Maryland Quantum Technology Center. The views, opinions and/or findings expressed here are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

CHAPTER

TEN

RF HETERODYNE WITH DOPPLER EXAMPLE

This notebook demonstrates two-tone detection using a Rydberg sensor in the time domain with Doppler averaging. An RF local oscillator (LO) and signal (sig) are imposed on the Rydberg sensor. This is useful for RF phase detection, and can be used to linearize the detection, as shown below. The main results of this example showing how different levels of Doppler averaging affect the beat signal size of the sensor.

```
import datetime
from numba import vectorize, float64
##***LAST UPDATE***##
now = datetime.datetime.now()
print(now)
```

```
2023-05-05 21:17:07.596404
```

10.1 Imports

```
import numpy as np
import rydiqule as rq
import matplotlib.pyplot as plt
```

```
%load_ext autoreload
%autoreload 2
```

10.2 Define the Sensors

```
rf_rabi = 100 #Mrad/s
red_laser = {'states':(0,1), 'rabi_frequency':2*np.pi*5}  #fields are stored as␣
↪dictioniaries
blue_laser = {'states':(1,2), 'rabi_frequency':2*np.pi*7, 'detuning': 0}
LO_ss = {'states':(2,3), 'rabi_frequency':rf_rabi, 'detuning':0}

RydbergTargetState = [150, 2, 2.5, 0.5]  #states labeled n, l, j, m_j
RydbergExcitedState = [149, 3, 3.5, 0.5]

atom = "Rb85"
RbSensor_ss = rq.Cell(atom, *rq.D2_states(atom), RydbergTargetState,␣
↪RydbergExcitedState,
                      gamma_transit=2*np.pi*1)
RbSensor_time = rq.Cell(atom, *rq.D2_states(atom), RydbergTargetState,␣
↪RydbergExcitedState,
                      gamma_transit=2*np.pi*1)
```

```
state1 = RbSensor_time.states_list()[2]
state2 = RbSensor_time.states_list()[3]
print("1: ", state1)
print("2: ", state1)
dipoleMoment = RbSensor_time.atom.getDipoleMatrixElement(*state1,*state2, 0)

field = rf_rabi/rq.scale_dipole(dipoleMoment)

print("applied field, V/m:", field) #V/m
print("Rabi frequency, Mrad/s: ", field*rq.scale_dipole(dipoleMoment))
```

```
1:  [150, 2, 2.5, 0.5]
2:  [150, 2, 2.5, 0.5]
applied field, V/m: 0.08558532725336343
Rabi frequency, Mrad/s:  100.0
```

```
def sig_and_LO( delta, beta):
    def fun(t):
        return (1+beta*np.sin(delta*t))
    return fun
```

```
rf_freq = RbSensor_time.atom.getTransitionFrequency(*RydbergTargetState[:3],
↪*RydbergExcitedState[:3])*1E-6
rf_freq #MHz
```

```
658.5872652398125
```

10.3 Observe a heterodyne beat between the Signal and LO.

10.3.1 Define the RF LO and signal

```
sampleNum = 200
endTime = 10 # microseconds
rf = sig_and_LO( 5, .1)
```

10.3.2 Solve without Doppler averaging

Observe the beat between signal and LO fields.

```
red_laser = {'states':(0,1), 'rabi_frequency':2*np.pi*5, 'detuning':0}
blue_laser = {'states':(1,2), 'rabi_frequency':2*np.pi*7, 'detuning': 0}
rf = {'states':(2,3), "rabi_frequency": rf_rabi, 'detuning': 0, 'time_dependence':␣
↪sig_and_LO( 2*np.pi, .05)}

RbSensor_time.add_couplings(blue_laser, red_laser, rf)
```

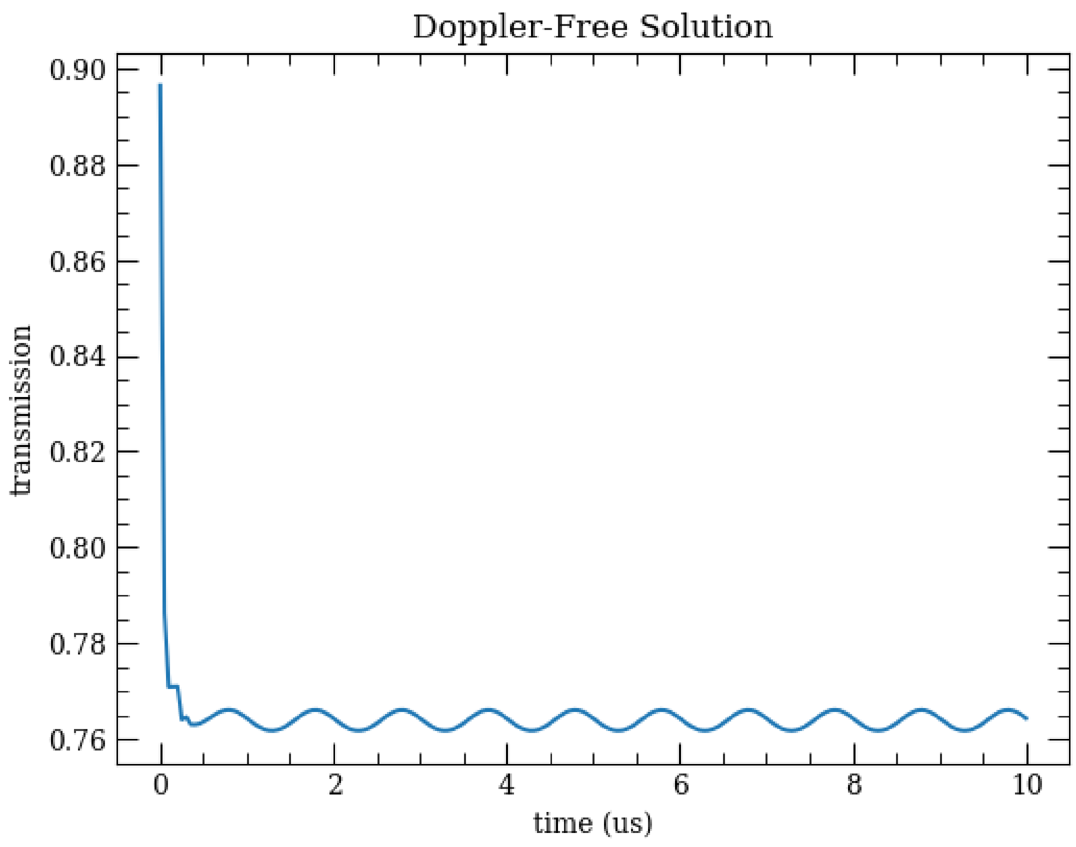
```
%%time
#Solve Without any doppler broadening

time_sol = rq.solve_time(RbSensor_time, endTime, sampleNum, atol=1e-6, rtol=1e-6)
```

```
CPU times: total: 78.1 ms
Wall time: 82.3 ms
```

```
transmission = rq.get_transmission_coef(time_sol, RbSensor_time, 1e-3)
```

```
fig, ax = plt.subplots()
ax.plot(time_sol.t, transmission)
ax.set_xlabel("time (us)")
ax.set_ylabel('transmission')
ax.set_title("Doppler-Free Solution")
```

```
Text(0.5, 1.0, 'Doppler-Free Solution')
```



10.3.3 Solve with Doppler averaging

Doppler averaged results require larger Rabi frequencies to observe similar sized signals.

```
red_laser = {'states':(0,1), 'rabi_frequency':2*np.pi*5, 'detuning':0, 'kvec':␣
↪500*(2*np.pi)*np.array([1,0,0])}
blue_laser = {'states':(1,2), 'rabi_frequency':2*np.pi*7, 'detuning': 0,'kvec':␣
↪308*(2*np.pi)*np.array([-1,0,0])}
#red_laser = {'states':(0,1), 'rabi_frequency':2*np.pi*1.0, 'detuning':0, 'kvec':␣
↪3*(2*np.pi)*np.array([1,0,0])}
#blue_laser = {'states':(1,2), 'rabi_frequency':2*np.pi*2.0, 'detuning': 0,'kvec':␣
↪1*(2*np.pi)*np.array([-1,0,0])}
rf = {'states':(2,3), "rabi_frequency":rf_rabi, 'detuning': 0, 'time_dependence':␣
↪sig_and_LO( 2*np.pi, .05)}

RbSensor_time.add_couplings(blue_laser, red_laser, rf)
```

```
%%time
#Solve with a doppler peak calculated from physical system properties
```

(continues on next page)

(continued from previous page)

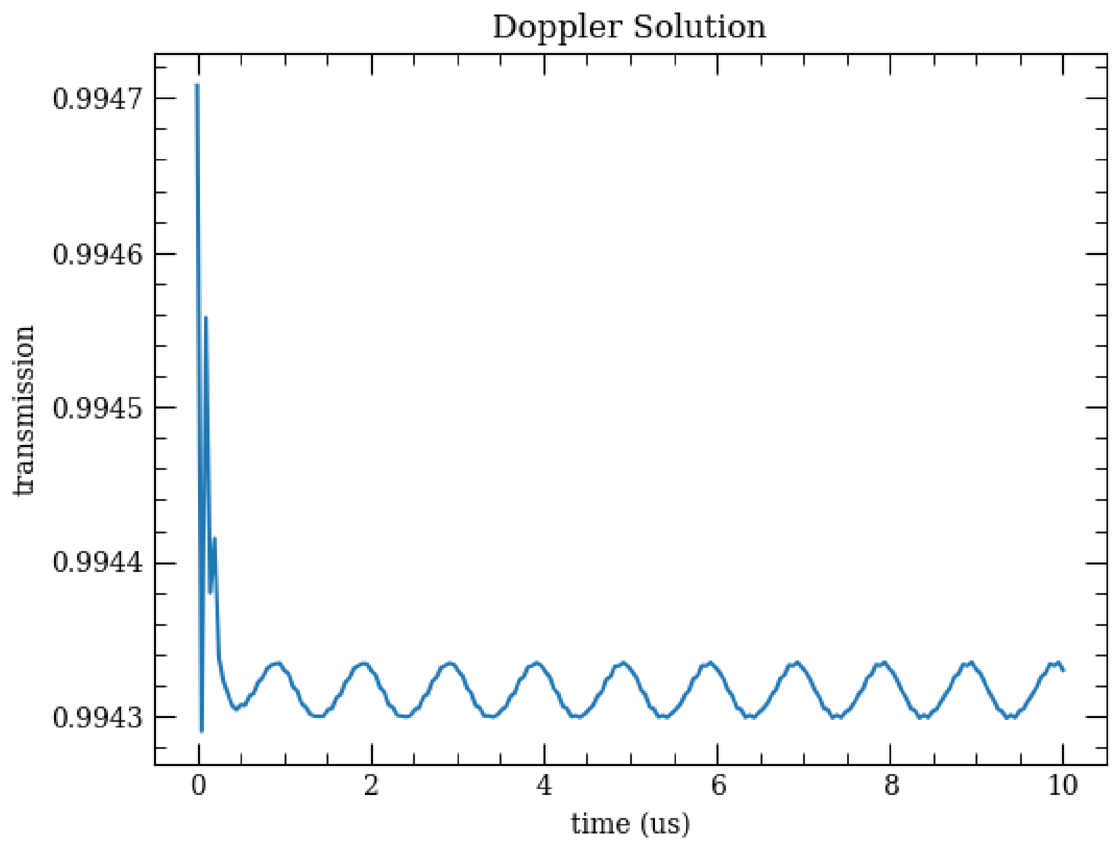
```
sampleNum = 200
endTime = 10
time_sol_doppler = rq.solve_time(RbSensor_time, endTime, sampleNum, doppler=True,␣
↪rtol = 1e-6, atol = 1e-6)
```

```
CPU times: total: 11.1 s
Wall time: 2min 46s
```

```
transmission_doppler = rq.get_transmission_coef(time_sol_doppler, RbSensor_time,␣
↪1e-3)
```

```
fig, ax = plt.subplots()
ax.plot(time_sol_doppler.t, transmission_doppler)
ax.set_xlabel("time (us)")
ax.set_ylabel('transmission')
ax.set_title("Doppler Solution")
```

```
Text(0.5, 1.0, 'Doppler Solution')
```



10.3.4 Solve with smaller Doppler Width

We can artificially reduce the amount of Doppler broadening. In this case, the default meshing of the velocity classes should be overridden to avoid excessive calculations.

```
red_laser = {'states':(0,1), 'rabi_frequency':2*np.pi*1.0, 'detuning':0, 'kvec':␣
↪1*(2*np.pi)*np.array([1,0,0])}
blue_laser = {'states':(1,2), 'rabi_frequency':2*np.pi*2.0, 'detuning': 0,'kvec':␣
↪0.6*(2*np.pi)*np.array([-1,0,0])}
rf = {'states':(2,3), "rabi_frequency":rf_rabi, 'detuning': 0, 'time_dependence':␣
↪sig_and_LO( 2*np.pi, .05)}

RbSensor_time.add_couplings(blue_laser, red_laser, rf)
```

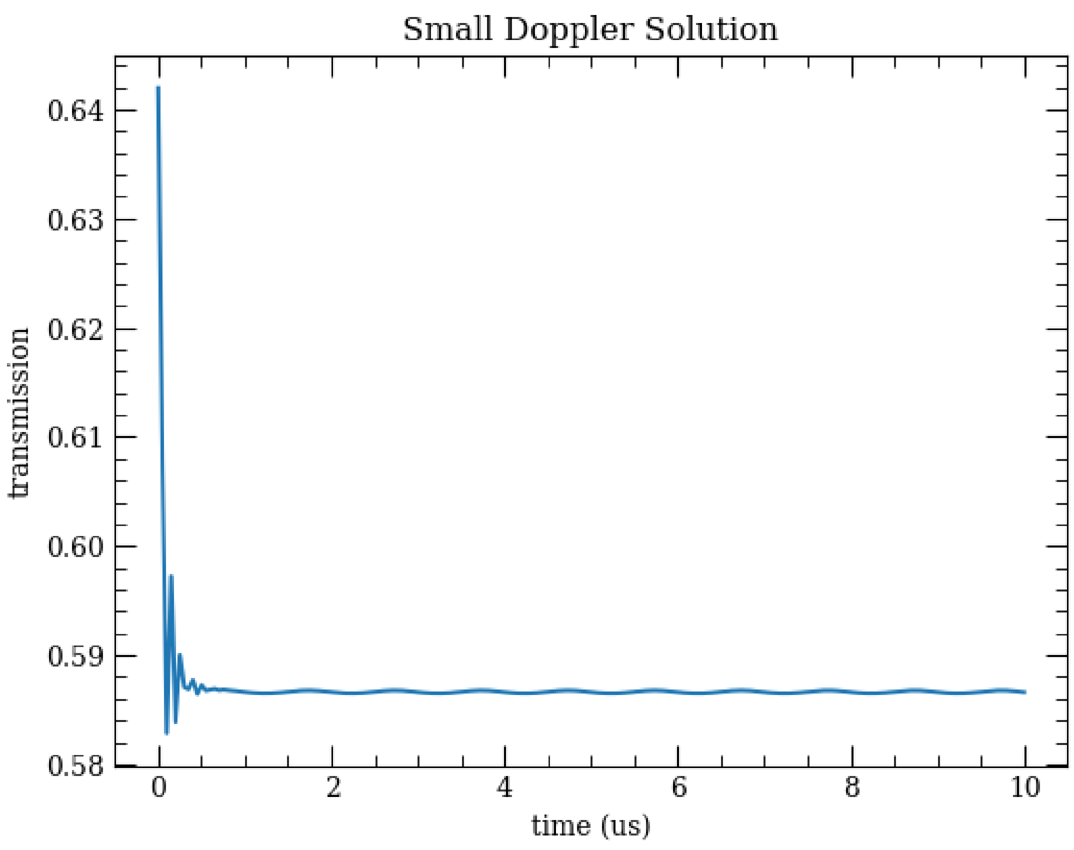
```
%%time
# Solve in the time domain with a 1MHz wide Doppler broadening
time_sol_doppler_narrow = rq.solve_time(RbSensor_time, endTime, sampleNum,
                                        doppler=True,
                                        doppler_mesh_method={'method':'uniform',
↪'width_doppler':2.5,'n_uniform':201},
                                        rtol = 1e-6, atol = 1e-6)
```

```
CPU times: total: 62.5 ms
Wall time: 1.1 s
```

```
transmission_doppler_narrow = rq.get_transmission_coef(time_sol_doppler_narrow,␣
↪RbSensor_time, 1e-3)
```

```
fig, ax = plt.subplots()
ax.plot(time_sol_doppler_narrow.t, transmission_doppler_narrow)
ax.set_xlabel("time (us)")
ax.set_ylabel('transmission')
ax.set_title("Small Doppler Solution")
```

```
Text(0.5, 1.0, 'Small Doppler Solution')
```



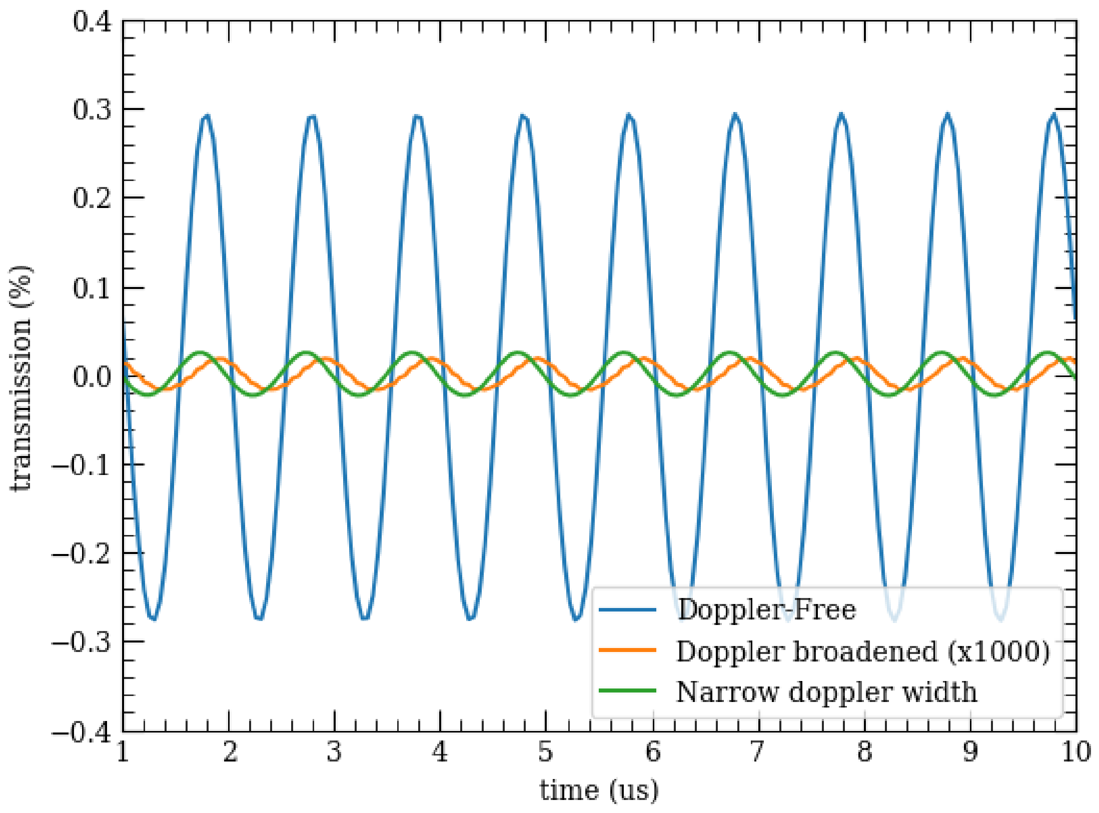
10.3.5 Compare the size of the beat signals

Here we ignore the starting transient, and normalize the beat signal. As the Doppler broadening is increased, the size of the beat is reduced (for the same optical depth).

```
def normalize_trace(trace,expand=1e2):
    ave = trace[100:].mean()
    return (trace - ave)/ave*expand
```

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

ax.plot(time_sol.t, normalize_trace(transmission), label='Doppler-Free')
ax.plot(time_sol_doppler.t, normalize_trace(transmission_doppler,1e3), label=
↪'Doppler broadened (x1000)')
ax.plot(time_sol_doppler_narrow.t, normalize_trace(transmission_doppler_narrow),␣
↪label='Narrow doppler width')
ax.set_xlim((1,10))
ax.set_ylim((-.4,.4))
ax.set_xlabel("time (us)")
ax.set_ylabel('transmission (%)')
ax.legend()
```

```
<matplotlib.legend.Legend at 0x21f036da850>
```



The authors recognize financial support from the US Army and Defense Advanced Research Projects Agency (DARPA). Rydiqule has been approved for unlimited public release by DEVCOM Army Research Laboratory and DARPA. This software is released under the xx licence through the University of Maryland Quantum Technology Center. The views, opinions and/or findings expressed here are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

rydiqule

CHAPTER

ELEVEN

RF HETERODYNE EXAMPLE

For a more thorough introduction to the core functionality of `rydiqule`, it may be helpful to look at the `Introduction_to_Rydiqule.ipynb` notebook before this one.

This notebook demonstrates two-tone detection using a Rydberg sensor in the time domain. An RF local oscillator (LO) and signal (sig) are imposed on the Rydberg sensor. This is useful for RF phase detection, and can be used to linearize the detection, as shown below. The main results of this example are:

1. we show that the time solver and steady-state solver approximately agree.
2. we plot an example of a time response due to RF heterodyne.
3. we find the optimum detuning of the Rydberg laser for RF heterodyne, for a given value of LO power.
4. we plot the linear dynamic of the RF heterodyning scheme, and show that it is limited by the LO power on the high end. The result is limited by the solver tolerance on the low end.

```
import datetime

##***LAST UPDATE***##
now = datetime.datetime.now()
print(now)
```

```
2023-05-05 22:11:43.987783
```

11.1 Imports

```
import numpy as np
import rydiqule as rq
import matplotlib.pyplot as plt
from tqdm import tqdm
```

```
%load_ext autoreload
%autoreload 2
```

11.2 Comparing the steady-state and time solver results

This example uses a `Cell` object, which inherits `Sensor`. The `Cell`'s purpose is to attach the bare pyhiscs calculations of a `Sensor` to a real physical atom (Rubidium-85) by default. This allows for specification of quantum numbers for states, meaning `rydiqule` can calculate things like transition frequencies (using ARC Rydberg) without needing to specify them in the object creation. The details will be discussed further down.

NOTE: The time solver runs more slowly for large transition frequencies, since it makes no rotating wave approximation. Therefore, it is advisable to debug calculations using a transition with a low frequency (ie, very large N). Once calculations are debugged and running well, they can be re-run with the appropriate n-level. Further, ARC calculates dipole moments for the chosen transition. For large n, this calculation is slow, due to the large amount of structure in the atomic wavefunction. However, ARC caches the results, so it only runs slowly the first time. It will likely take several minutes to calculate the add_states function, the first time, but then will run quickly.

11.2.1 The steady-state `Cell`

```
#states in cell are labeled by [n, l, j, m_j]
#A base sensor object does not support specification of quantum numbers for each␣
↪level
rydberg_target_state = [150, 2, 2.5, 0.5]
rydberg_excited_state = [149, 3, 3.5, 0.5]

atom = "Rb85"
RbSensor_ss = rq.Cell(atom, *rq.D2_states(atom), rydberg_target_state, rydberg_
↪excited_state,
                      gamma_transit=2*np.pi*1)
```

Define Transitions

Transitions are defined as dictionaries in a `Cell` in the same way as they are in a bas `Sensor`. For the steady-state case, nothing changes. For this example, will observer the response of the system over a series of 200 blue laser detunings.

```
rf_rabi = 25 #Mrad/s
n_det_ss = 200
detunings_ss = np.linspace(-150, 150, n_det_ss)

red_laser = {'states':(0,1), 'rabi_frequency':2*np.pi*0.6, 'detuning':0}
blue_laser = {'states':(1,2), 'rabi_frequency':2*np.pi*1.0, 'detuning':detunings_
↪ss}
local_oscillator_ss = {'states':(2,3), 'rabi_frequency':rf_rabi, 'detuning':0}

RbSensor_ss.add_couplings(red_laser, blue_laser, local_oscillator_ss)
```

Solve the steady state system

We solve a `Cell` in exaclty the same way we solve a `Sensor` object.

```
ss_solution = rq.solve_steady_state(RbSensor_ss)
print(ss_solution.rho.shape)
```

```
(200, 15)
```

11.2.2 The time solver `Cell`

We want to use the same decoherence values

```
rydberg_target_state = [150, 2, 2.5, 0.5]
rydberg_excited_state = [149, 3, 3.5, 0.5]

RbSensor_time = rq.Cell(atom, *rq.D2_states(atom), rydberg_target_state, rydberg_
↪excited_state,
                        gamma_transit=2*np.pi*1)
```

Define Couplings

```
[n,l,j,m] = RbSensor_ss.states_list()[2]
[n2, l2, j2, m2]  = RbSensor_ss.states_list()[3]
rf_freq = RbSensor_ss.atom.getTransitionFrequency(n2,l2,j2,n,l,j)*1E-6
def rf_carrier(t):
    return np.cos(2*np.pi*rf_freq*t)#extra factor of 2 to account for no RWA.
```

```
n_det = 20
detunings = np.linspace(-75, 75, n_det)

red_laser = {'states':(0,1), 'rabi_frequency':2*np.pi*0.6, 'detuning':0}
blue_laser = {'states':(1,2), 'rabi_frequency':2*np.pi*1.0, 'detuning':detunings}
rf_transition = {'states':(2,3), 'rabi_frequency':rf_rabi, 'time_dependence': rf_
↪carrier }

RbSensor_time.add_couplings(red_laser, blue_laser, rf_transition)
```

Defining the rf field

The `time_dependence` argument s expected to be a python function of a single variable (time in μs) that returns the field at that time. To match our steady state solution, which had a detuning of 0, we will explicitly define a single-tone field as a function of time that is resonant with our rf transition.

Solve in the time domain

Internally, the time solver will loop over all detuning values and output the full result. You can use the `verbose` flag to print its progress to `stdout`

```
%%time
end_time = 10 #microseconds
sample_num = 10

time_solution = rq.solve_time(RbSensor_time, end_time, sample_num, atol=1e-6,␣
↪rtol=1e-6)
```

```
CPU times: total: 8.92 s
Wall time: 13.8 s
```

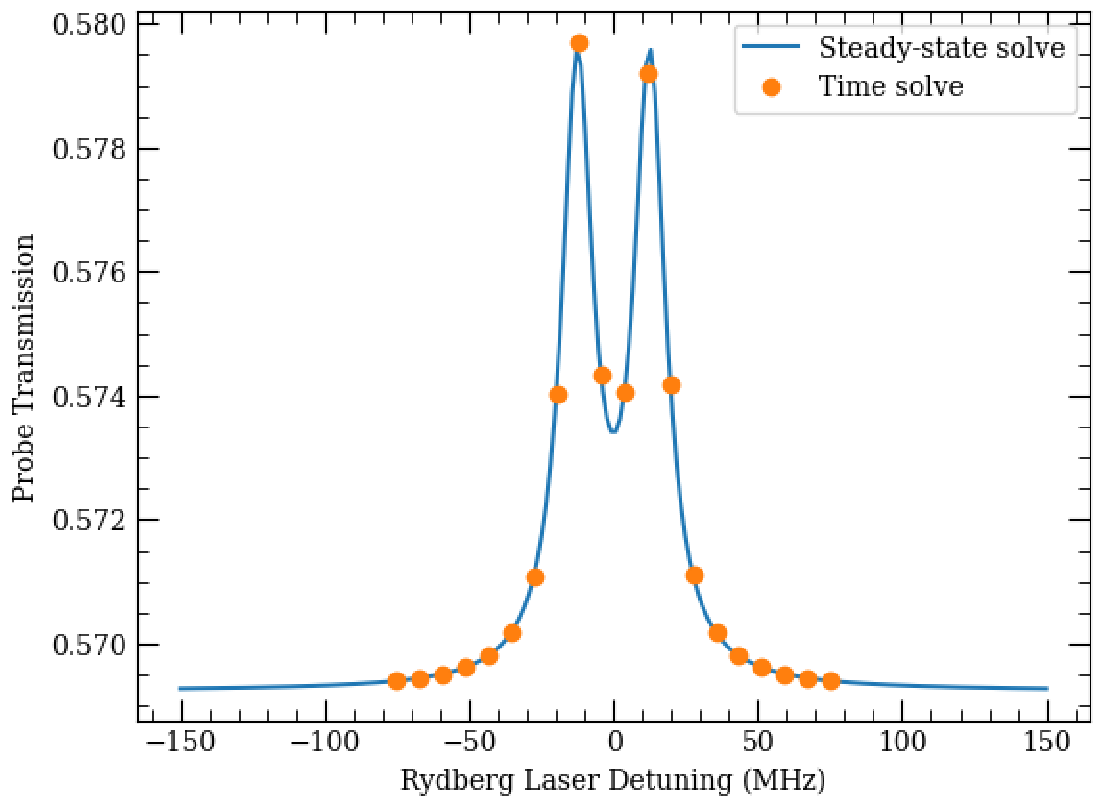
```
RbSensor_time.couplings.edges[2,3]
```

```
{'rabi_frequency': 25,
 'transition_frequency': -4138.025828450375,
 'phase': 0,
 'kvec': (0, 0, 0),
 'time_dependence': <function __main__.rf_carrier(t)>,
 'no_rwa_warning': False,
 'label': '(2,3)'}
```

11.2.3 Comparing results

Now, with results for both steady state and time simulations, we can compare them and see that they match. Note that because the time solution has an extra dimension, we only get the last (`[:,-1]`) element, since this is effectively the steady-state solution (assuming transient behavior has been damped out by 10 μs). Here we also use a function to get the transmission coefficient from the solution quickly by extracting the proper density matrix elements.

```
#Modify to include convenience functions and get physical parameters.

fig, ax = plt.subplots()
ax.plot(detunings_ss, rq.get_transmission_coef(ss_solution, RbSensor_ss, .001),␣
↪label="Steady-state solve")
ax.plot(detunings, rq.get_transmission_coef(time_solution, RbSensor_time, .001)[:,-
↪1], 'o', label="Time solve")
ax.set_xlabel("Rydberg Laser Detuning (MHz)")
ax.set_ylabel("Probe Transmission")
ax.legend();
```



11.3 Observe a heterodyne beat between the Signal and LO.

The goal of this section is to see the beating behavior in time between the signal and rf local oscillator. This will help see exactly how to observe behavior in the time domain with rydiqule. It will look a lot like other time solves, but we will look at the solution in a different way.

11.3.1 Define the RF LO and signal

Just like before, we need a function of time to input into the time solver. This time, instead of just an rf carrier signal, we will define a function that adds a local oscillator of frequency ω_0 with an "incoming" rf signal of frequency $\omega_0+\delta$. We will also define the time function as the return of another function, which will allow us to make changes to the function quickly if we want to experiment a little.

```
def sig_and_LO(omega_0, delta, beta):
    def fun(t):
        return np.sin(omega_0*t)+beta*np.sin((omega_0+delta)*t)
    return fun
```

```
omega_0 = 2*np.pi*rf_freq
delta = 5
beta = 0.05
```

11.3.2 The Sensor

We will use the same `RbSensor_time` sensor as before, but change the blue laser to be just a single value. This highlights an important aspect of `Sensor`. At present, it does not support multiple fields coupling the same pair of levels, but will override an old one with a new one when `add_coupling()` is called.

```
red_laser = {'states':(0,1), 'rabi_frequency':2*np.pi*0.6, 'detuning':0}
blue_laser = {'states':(1,2), 'rabi_frequency':6.0, 'detuning': 0}
rf_transition = {'states':(2,3), 'rabi_frequency':rf_rabi, 'time_dependence': sig_
↪and_LO(omega_0, delta, beta )}
RbSensor_time.add_couplings(red_laser,blue_laser, rf_transition)
```

11.3.3 Inital conditions

If the `init_cond` argument is not supplied to `rq.solve_time`, it will calculate the inital condition based on the steady-state solution of the supplied sensor **without** any of the time-dependant fields. Since we define our incoming field and LO in a single function, we will likely end up with a sizeable transient if we use this approach. This is now a great opportunity to demonstrate how to supply an inital condition manually. We calculate our inital condition using the solution to the **steady-state** sensor we defined above. Hopefully, this allows us to see the beat oscillation around the steady-state solution without a large transient from introducing our LO and rf field at the same time.

Solving is done the same way as before, this time solving for 250 points in 10 microseconds

```
RbSensor_ss.add_couplings(blue_laser)
sol_init = rq.solve_steady_state(RbSensor_ss)

sample_num=250
end_time = 10

time_sol_beat = rq.solve_time(RbSensor_time, end_time, sample_num,  init_cond=sol_
↪init.rho)

print(time_sol_beat.rho.shape)
```

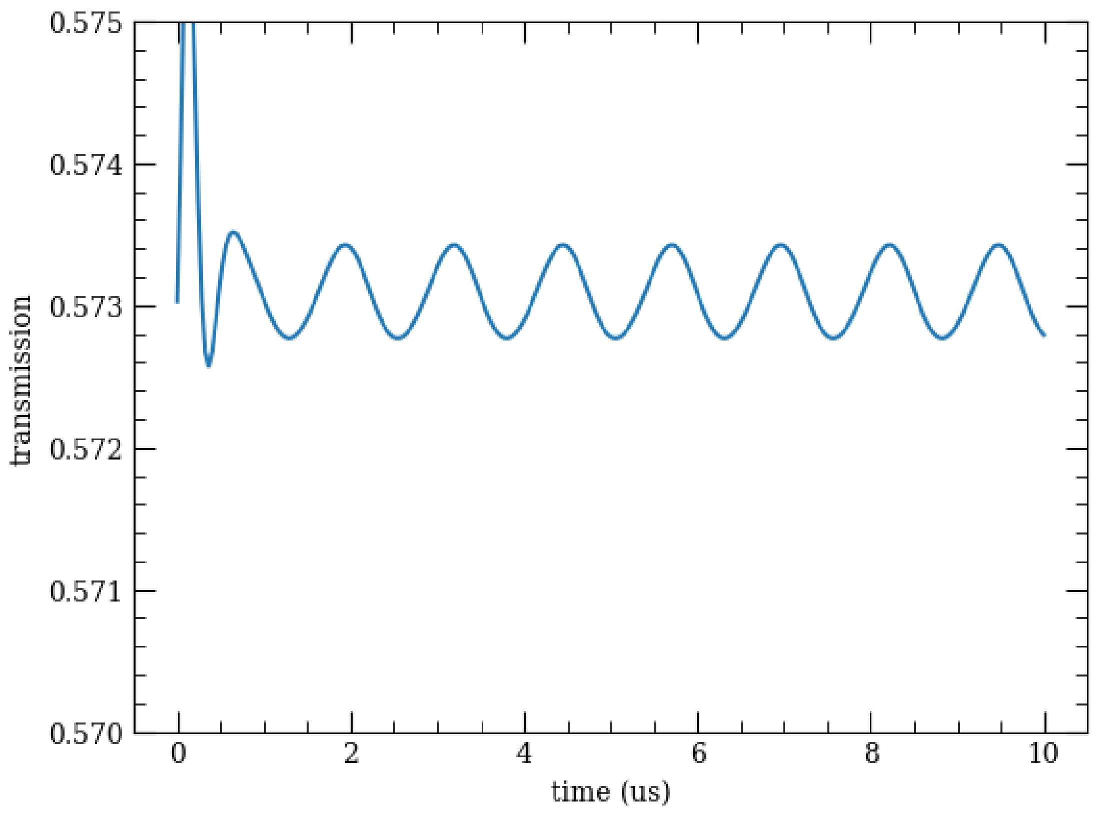
```
(250, 15)
```

11.3.4 Plotting the beat

We now use the `get_transmission_coef()` function again to extract the transmission from the `(250, 15)`-shaped solution to get a 250 element array we can plot against time. We can see that the system quickly settles into the expected beat frequency of 1 MHZ

```
fig, ax = plt.subplots()
transmission = time_sol_beat.rho[:,3]
ax.plot(time_sol_beat.t, rq.get_transmission_coef(time_sol_beat, RbSensor_time, .
↪001))
ax.set_xlabel("time (us)");
ax.set_ylabel('transmission');
ax.set_ylim(bottom = .57, top = .575)
```

```
(0.57, 0.575)
```



We do have a bit of transient behavior at the start, but we can clearly see the beat expected bead frequency of 5Mrad/s ≈ 800kHz.

11.4 RF Heterodyne in the Rotating Wave Approximation

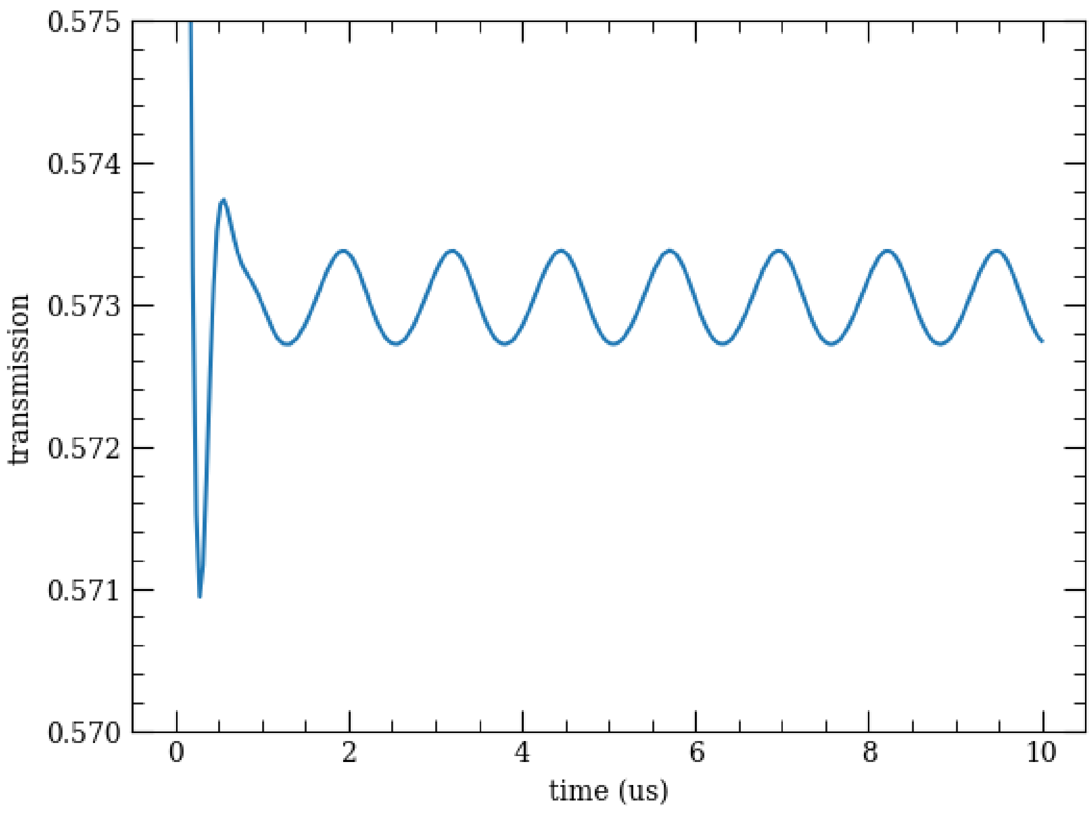
We move to a rotating frame by specifying an rf detuning, and writing the coupling in the complex rotating frame. This transformation will greatly speed up the time integration.

```
def sig_LO_RWA(det, beta):
    def fun(t):
        return 1+beta*np.exp(1j*det*t)
    return fun

red_laser = {'states':(0,1), 'rabi_frequency':2*np.pi*0.6, 'detuning':0}
blue_laser = {'states':(1,2), 'rabi_frequency':6.0, 'detuning': 0}
rf_transition = {'states':(2,3), 'rabi_frequency':rf_rabi, 'detuning':0, 'time_
↪dependence': sig_LO_RWA(delta, beta )}
RbSensor_time.add_couplings(red_laser, blue_laser, rf_transition)

sol_init = rq.solve_steady_state(RbSensor_ss)
```

```
sample_num=250
end_time = 10
time_sol_beat = rq.solve_time(RbSensor_time, end_time, sample_num)
```

```
fig, ax = plt.subplots()
ax.plot(time_sol_beat.t, rq.get_transmission_coef(time_sol_beat, RbSensor_time, .
↪001))
ax.set_xlabel("time (us)");
ax.set_ylabel('transmission');
ax.set_ylim(bottom = .57, top = .575)
```

```
(0.57, 0.575)
```



11.5 Find the optimum laser detuning with LO

We again use the same sensor to look at the sensitivity of the sensor as a function of blue laser detuning.

11.5.1 Set up and solve `Sensor`

```
num_dets = 75
detuning_list = np.linspace(-40,40,num_dets)
pk_to_pk_result = np.zeros(num_dets)
rf = sig_and_LO(2*np.pi*rf_freq, 5, 0.01)

sample_num = 300
end_time = 3
blue_laser = {'states':(1,2), 'rabi_frequency':6.0, 'detuning': detuning_list}
rf_transition = {'states':(2,3), 'rabi_frequency':rf_rabi, 'time_dependence': sig_
↪and_LO(rf_freq*2*np.pi, 2*np.pi, .05 )}
RbSensor_time.add_couplings(blue_laser, rf_transition) #this replaces the old␣
↪coupling

time_sol = rq.solve_time(RbSensor_time, end_time, sample_num)
```

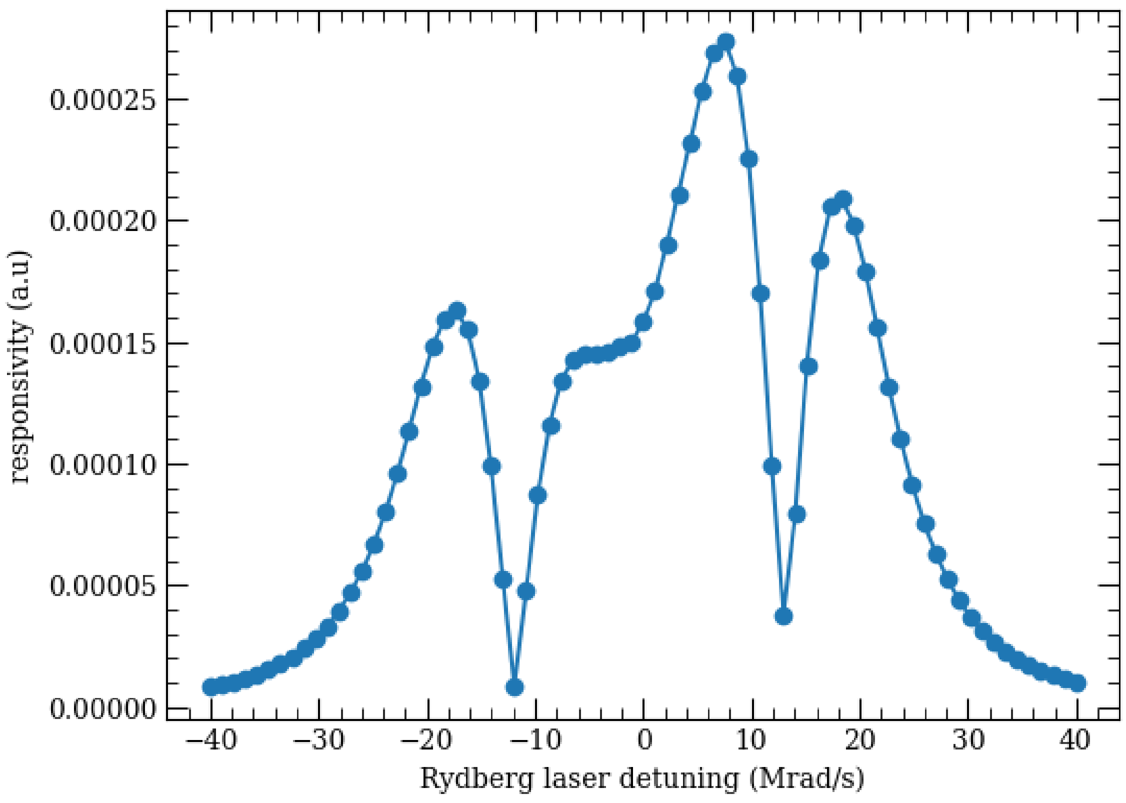
```
susceptibility = time_sol.rho[:,100:,3]
print(susceptibility.shape)
ptp_result = np.ptp(susceptibility, axis=-1)
```

```
(75, 200)
```

11.5.2 Plotting responsivity versus LO detuning

```
fig, ax = plt.subplots()
ax.plot(detuning_list, ptp_result,'o-')
ax.set_ylabel("responsivity (a.u)")
ax.set_xlabel("Rydberg laser detuning (Mrad/s)")
```

```
Text(0.5, 0, 'Rydberg laser detuning (Mrad/s)')
```



This plot shows that, for maximum responsivity, the Rydberg (or probe) laser must be tuned to the side of an Autler-Townes peak, for maximum sensitivity

11.6 Test the Linear Dynamic Range

Example testing the linear dynamic range in Heterodyne

11.6.1 Setting the laser parameters

We will set the blue laser detuning to -8 MRad/s, which was roughly the optimal value from the plot above

```
blue_laser = {'states':(1,2), 'rabi_frequency':6.0, 'detuning': -8}
RbSensor_time.add_couplings(blue_laser)
```

11.6.2 Solve parameters

With the detuning set, we can set up everything we need for our scan, namely the solver parameters and list of amplitudes.

```
num_Amps = 50
amp_list = np.logspace(-6,0.2,num_Amps)
sample_num = 300
end_time = 3
```

11.6.3 Running the loop

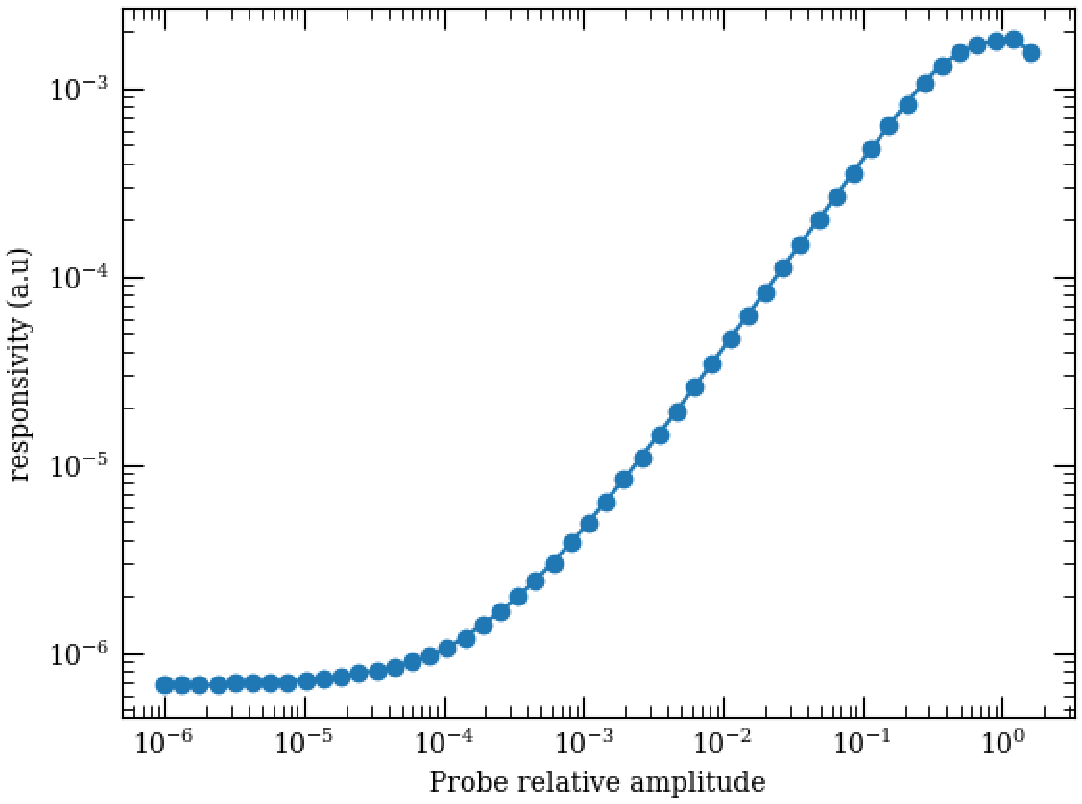
Now all that is left is get the value we want at each amplitude using a good old python `for` loop. This could be done with something like a `map()` function depending on your comfort level with python, but we will be explicit here.

```
pk_to_pk_result = np.zeros(num_Amps)

for idx, amp in enumerate(tqdm(amp_list)):
    #define and solve for rf input

    rf = sig_and_LO(2*np.pi*rf_freq, 5, amp)
    rf_transition = {'states':(2,3), 'rabi_frequency':rf_rabi, 'detuning':1, 'time_
↪dependence': sig_LO_RWA(1, amp)}
    RbSensor_time.add_couplings(rf_transition)
    time_sol = rq.solve_time(RbSensor_time, end_time, sample_num,  init_cond=sol_
↪init.rho, atol=1e-7, rtol=1e-7)

    #calculate responsivity
    pk_to_pk_signal = np.ptp(time_sol.rho[100:,3])
    pk_to_pk_result[idx] = pk_to_pk_signal
```

```
100
↪%|████████████████████████████████████████████████████████████████████████████████
↪50/50 [00:00<00:00, 53.58it/s]
```

11.6.4 Plotting the dynamic range

```
fig, ax = plt.subplots()
ax.plot(amp_list, pk_to_pk_result,'o-')
ax.set_ylabel("responsivity (a.u)")
ax.set_xlabel("Probe relative amplitude")
ax.set_xscale('log')
ax.set_yscale('log')
```



The authors recognize financial support from the US Army and Defense Advanced Research Projects Agency (DARPA). Rydiqule has been approved for unlimited public release by DEVCOM Army Research Laboratory and DARPA. This software is released under the xx licence through the University of Maryland Quantum Technology Center. The views, opinions and/or findings expressed here are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

PYTHON MODULE INDEX

r

rydiqule, ??
rydiqule.atom_utils, ??
rydiqule.cell, ??
rydiqule.doppler_utils, ??
rydiqule.energy_diagram, ??
rydiqule.experiments, ??
rydiqule.rydiqule_utils, ??
rydiqule.sensor, ??
rydiqule.sensor_solution, ??
rydiqule.sensor_utils, ??
rydiqule.solvers, ??
rydiqule.timesolvers, ??