spinor-gpe

Release 0.0.1

Benjamin D. Smith

Dec 21, 2021

CONTENTS:

1 Introduction **3**

2 Installation **5**
- 2.1 Dependencies . . . 5
  - 2.1.1 Primary packages: . . . 5
  - 2.1.2 Other packages: . . . 5
- 2.2 Installing Dependencies . . . 5
- 2.3 CUDA Installation . . . 6
- 2.4 Getting Started . . . 6

3 Basic Operation **7**
- 3.1 1. Setup: Data path and PSpinor object . . . 7
- 3.2 2. Run: Begin Propagation . . . 7
- 3.3 3. Analyze: Plot the results . . . 8
- 3.4 4. Repeat . . . 8

4 Examples **9**
- 4.1 Example 1: Ground State . . . 9
  - 4.1.1 Physical Parameters . . . 9
  - 4.1.2 Numerical Parameters . . . 10
- 4.2 Example 2: Anisotropic Time-of-Flight . . . 13
  - 4.2.1 Physical Parameters . . . 13
  - 4.2.2 Numerical Parameters . . . 14
- 4.3 Example 3: Raman Rabi Flopping . . . 17
  - 4.3.1 Physical Parameters . . . 17
  - 4.3.2 Numerical Parameters . . . 18
- 4.4 Example 4: Raman Detuning Gradient Ground State . . . 21
  - 4.4.1 Physical Parameters . . . 21
  - 4.4.2 Numerical Parameters . . . 23

5 Benchmarks **27**
- 5.1 Physical Parameters . . . 27
- 5.2 Numerical Parameters . . . 28
  - 5.2.1 FFT and iFFT Time . . . 29
  - 5.2.2 Hadamard Time . . . 32
  - 5.2.3 Propagation Time . . . 34

6 pspinor package **37**
- 6.1 Submodules . . . 37
- 6.2 pspinor.pspinor module . . . 37
- 6.3 pspinor.tensor_propagator module . . . 44

6.4 pspinor.prop_result module 47
6.5 pspinor.tensor_tools module 49
6.6 pspinor.plotting_tools module 54
6.7 Module contents 56

7 Indices and tables **57**

Python Module Index **59**

Index **61**

`spinor-gpe` is high-level, object-oriented Python package for numerically solving the quasi-2D, psuedospinor (two component) Gross-Piteavskii equation (GPE), for both ground state solutions and real-time dynamics. This project grew out of a desire to make high-performance simulations of the GPE more accessible to the entering researcher.

While this package is primarily built on NumPy, the main computational heavy-lifting is performed using PyTorch, a deep neural network library commonly used in machine learning applications. PyTorch has a NumPy-like interface, but a backend that can run either on a conventional processor or a CUDA-enabled NVIDIA(R) graphics card. Accessing a CUDA device will provide a significant hardware acceleration of the simulations.

This package has been tested on Windows, Mac, and Linux systems.

View the documentation on ReadTheDocs

CHAPTER

ONE

INTRODUCTION

`spinor-gpe` is high-level, object-oriented Python package for numerically solving the quasi-2D, psuedospinor (two component) Gross-Piteavskii equation (GPE), for both ground state solutions and real-time dynamics. This project grew out of a desire to make high-performance simulations of the GPE more accessible to the entering researcher.

While this package is primarily built on NumPy, the main computational heavy-lifting is performed using PyTorch, a deep neural network library commonly used in machine learning applications. PyTorch has a NumPy-like interface, but a backend that can run either on a conventional processor or a CUDA-enabled NVIDIA(R) graphics card. Accessing a CUDA device will provide a significant hardware acceleration of the simulations.

This package has been tested on Windows, Mac, and Linux systems.

View the documentation on ReadTheDocs

CHAPTER

TWO

INSTALLATION

2.1 Dependencies

2.1.1 Primary packages:

1. PyTorch >= 1.8.0
2. cudatoolkit >= 11.1
3. NumPy

2.1.2 Other packages:

4. matplotlib (visualizing results)
5. tqdm (progress messages)
6. scikit-image (matrix signal processing)
7. ffmpeg = 4.3.1 (animation generation)

2.2 Installing Dependencies

The dependencies for `spinor-gpe` can be installed directly into the new `conda` virtual environment `spinor` using the *environment.yml* file included with the package:

```
conda env create --file environment.yml
```

This installation may take a while.

Note: The version of CUDA used in this package does not support macOS. Users on these computers may still install PyTorch and run the examples on their CPU. To install correctly on macOS, remove the - *cudatoolkit=11.1* line from the *environment.yml* file. After installation, you will need to modify the example code to run on the *cpu* device instead of the *cuda* device.

The above dependencies can also be installed manually using `conda` into a virtual environment:

```
conda activate <new_virt_env_name>
conda install pytorch torchvision torchaudio cudatoolkit=11.1 -c pytorch -c conda-forge
conda install numpy matplotlib tqdm scikit-image ffmpeg spyder
```

Note: For more information on installing PyTorch, see its installation instructions page.

To verify that Pytorch was installed correctly, you should be able to import it:

```
>>> import torch
>>> x = torch.rand(5, 3)
>>> print(x)
tensor([[0.2757, 0.3957, 0.9074],
        [0.6304, 0.1279, 0.7565],
        [0.0946, 0.7667, 0.2934],
        [0.9395, 0.4782, 0.9530],
        [0.2400, 0.0020, 0.9569]])
```

Also, if you have an NVIDIA GPU, you can test that it is available for GPU computing:

```
>>> torch.cuda.is_available()
True
```

2.3 CUDA Installation

CUDA is the API that interfaces with the computing resources on NVIDIA graphics cards, and it can be accessed through the PyTorch package. If your computer has an NVIDIA graphics card, start by verifying that it is CUDA-compatible. This page lists out the compute capability of many NVIDIA devices. (Note: yours may still be CUDA-compatible even if it is not listed here.)

Given that your graphics card can run CUDA, the following are the steps to install CUDA on a Windows computer:

1. Install the NVIDIA CUDA Toolkit. Go to the CUDA download page for the most recent version. Select the operating system options and installer type. Download the installer and install it via the wizard on the screen. This may take a while. For reference, here is the Windows CUDA Toolkit installation guide.

   To test that CUDA is installed, run *which nvcc*, and, if instlled correctly, will return the installation path. Also run *nvcc –version* to verify that the version of CUDA matches the PyTorch CUDA toolkit version (>=11.1).

2. Download the correct drivers for your NVIDIA device. Once the driver is installed, you will have the NVIDIA Control Panel installed on your computer.

2.4 Getting Started

1. Clone the repository.
2. Navigate to the `spinor_gpe/examples/` directory, and start to experiment with the examples there.

CHAPTER

THREE

BASIC OPERATION

This package has a simple, object-oriented interface for imaginary- and real-time propagations of the pseudospinor-GPE. While there are other operations and features to this package, all simulations will have the following basic structure:

3.1 1. Setup: Data path and PSpinor object

```
>>> import pspinor as spin
>>> DATA_PATH = '<project_name>/Trial_###'
>>> ps = spin.PSpinor(DATA_PATH)
```

The program will create a new directory `DATA_PATH`, in which the data and results from this simulation trial will be saved. If `DATA_PATH` is a relative path, as shown above, then the trial data will be located in the `/data/` folder. When working with multiple simulation projects, it can be helpful to specify a `<project_name>` directory; furthermore, the form `Trial_###` is convenient, but not strictly required.

3.2 2. Run: Begin Propagation

The example below demonstrates imaginary-time propagation. The method `PSpinor.imaginary` performs the propagation loop and returns a `PropResult` object. This object contains the results, including the final wavefunctions and populations, and analysis and plotting methods (described below).

```
>>> DT = 1/50
>>> N_STEPS = 1000
>>> DEVICE = 'cuda'
>>> res = ps.imaginary(DT, N_STEPS, DEVICE, is_sampling=True, n_samples=50)
```

For real-time propagation, use the method `PSpinor.real`.

3.3 3. Analyze: Plot the results

`PropResult` provides several methods for viewing and understanding the final results. The code block below demonstrates several of them:

```
>>> res.plot_spins()  # Plots the spin-dependent densities and phases.
>>> res.plot_total()  # Plots the total densities and phases.
>>> res.plot_pops()   # Plots the spin populations throughout the propagation.
>>> res.make_movie()  # Generates a movie from the sampled wavefunctions.
```

Note that `PSpinor` also exposes methods to plot the spin and total densities. These can be used independent of `PropResult`:

```
>>> ps.plot_spins()
```

3.4 4. Repeat

Likely you will want to repeat or chain together different segments of this structure. Demonstrations of this are shown in the `Examples` gallery.

CHAPTER

FOUR

EXAMPLES

These four examples showcase different features and capabilities of the `spinor-gpe` package, including imaginary- and real-time propagation, together with Raman coupling and detuning configurations.

4.1 Example 1: Ground State

Starting with the Thomas-Fermi solution, propagate in imaginary time to reach the ground state. Propagation smooths out the sharp edges on both components' densities.

4.1.1 Physical Parameters

Atom number

$N_{\mathrm{at}} = 100$

Atomic mass, Rubidium-87

$m = 1.4442 \times 10^{-25}$ [kg]

Trap frequencies

$(\omega_x, \omega_y, \omega_z) = 2\pi \times (50, 50, 2000)$ [Hz]

$(\omega_x, \omega_y, \omega_z) = \omega_x \times (1, \gamma, \eta) = (1, 1, 40)$ [ω_x]

Harmonic oscillator length, x-axis

$a_x = \sqrt{\hbar/m\omega_x} = 1.525$ [μm]

3D scattering length, Rubidium-87

$a = 5.313$ [nm]

$a_{\rm sc} = a/a_x = 0.00348$ [a_x]

Scattering 2D scale

$g_{\rm sc}^{\rm 2D} = \sqrt{8\pi\eta}\, a_{\rm sc} = 0.1105$ [$\omega_x a_x^2$]

Scattering coupling

$(g_{\rm uu}, g_{\rm dd}, g_{\rm ud}) = g_{\rm sc}^{\rm 2D} \times (1, 1, 1.04)$ [$\omega_x a_x^2$]

Chemical potential

$\mu = \sqrt{4N_{\rm at} a_{\rm sc} \gamma \sqrt{\eta/2\pi}} = 1.875$ [ω_x]

Thomas-Fermi radius

$R_{\rm TF} = \sqrt{2\mu} = 1.937$ [a_x]

Initial population fractions

$(p_0, p_1) = (0.5, 0.5)$

Raman wavelength

$\lambda_L = 790.1$ [nm]

4.1.2 Numerical Parameters

Number of grid points

$(N_x, N_y) = (64, 64)$

r-grid half-size

$(x^{\max}, y^{\max}) = (8, 8)\ [a_x]$

r-grid spacing

$(\Delta x, \Delta y) = (0.25, 0.25)\ [a_x]$

k-grid half-size

$(k_x^{\max}, k_y^{\max}) = \pi/(\Delta x, \Delta y)$

$(k_x^{\max}, k_y^{\max}) = (12.566, 12.566)\ [a_x^{-1}]$

k-grid spacing

$(\Delta k_x, \Delta k_y) = \pi/(x^{\max}, y^{\max})$

$(\Delta k_x, \Delta k_y) = (0.3927, 0.3927)\ [a_x^{-1}]$

Time scale

$\tau_0 = 1/\omega_x = 0.00318\ [\mathrm{s/rad}]$

$\tau_0 = 1\ [\omega_x^{-1}]$

Time step duration, imaginary

$\Delta\tau_{\mathrm{im}} = 1/50\ [-i\tau_0]$

Number of time steps, imaginary

$N_{\mathrm{im}} = 100$

```
import os
import sys
sys.path.insert(0, os.path.abspath('../..'))  # Adds project root to the PATH

import numpy as np

from spinor_gpe.pspinor import pspinor as spin


# 1. SETUP
```

(continues on next page)

(continued from previous page)

```
DATA_PATH = 'examples/Trial_011'  # Default data path is in the /data/ folder

FREQ = 50
W = 2*np.pi*FREQ
Y_SCALE = 1
Z_SCALE = 40.0

ATOM_NUM = 1e2
OMEG = {'x': W, 'y': Y_SCALE * W, 'z': Z_SCALE * W}
G_SC = {'uu': 1, 'dd': 1, 'ud': 1.04}

ps = spin.PSpinor(DATA_PATH, overwrite=True,  # Initialize PSpinor object
                  atom_num=ATOM_NUM,
                  omeg=OMEG,
                  g_sc=G_SC,
                  pop_frac=(0.5, 0.5),
                  r_sizes=(8, 8),
                  mesh_points=(64, 64))

ps.coupling_setup(wavel=790.1e-9, kin_shift=False)

ZOOM = 4  # Zooms the momentum-space density plots by a constant factor

# Plot real- and momentum-space density & real-space phase of both components
ps.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM)


# 2. RUN (Imaginary-time)

DT = 1/50
N_STEPS = 100
DEVICE = 'cpu'
ps.rand_seed = 99999

# Run propagation loop:
# - Returns `PropResult` & `TensorPropagator` objects
res, prop = ps.imaginary(DT, N_STEPS, DEVICE, is_sampling=True, n_samples=50)


# 3. ANALYZE

res.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM)
res.plot_total(kscale=ps.kL_recoil, zoom=ZOOM)  # Plot total density & phase
res.plot_pops()  # Plot how the spins' populations evolves
res.make_movie(rscale=ps.rad_tf, kscale=ps.kL_recoil, play=True, zoom=ZOOM,
               norm_type='half')
```

Total running time of the script: (0 minutes 0.000 seconds)

4.2 Example 2: Anisotropic Time-of-Flight

Starts with the Thomas-Fermi solution for a highly anisotropic trap. Propagates in imaginary time tor reach the ground state. The trapping potential is suddenly removed and both components expand and experience an inversion of their aspect ratio throughout real time propagation.

4.2.1 Physical Parameters

Atom number

$N_{\mathrm{at}} = 10,000$

Atomic mass, Rubidium-87

$m = 1.4442 \times 10^{-25}$ [kg]

Trap frequencies

$(\omega_x, \omega_y, \omega_z) = 2\pi \times (50, 200, 2000)$ [Hz]

$(\omega_x, \omega_y, \omega_z) = \omega_x \times (1, \gamma, \eta) = (1, 4, 40)$ $[\omega_x]$

Harmonic oscillator length, x-axis

$a_x = \sqrt{\hbar/m\omega_x} = 1.525$ [μm]

3D scattering length, Rubidium-87

$a = 5.313$ [nm]

$a_{\mathrm{sc}} = a/a_x = 0.00348$ $[a_x]$

Scattering 2D scale

$g_{\mathrm{sc}}^{\mathrm{2D}} = \sqrt{8\pi\eta}\, a_{\mathrm{sc}} = 0.1105$ $[\omega_x a_x^2]$

Scattering coupling

$(g_{\mathrm{uu}}, g_{\mathrm{dd}}, g_{\mathrm{ud}}) = g_{\mathrm{sc}}^{\mathrm{2D}} \times (1, 1, 0.5)\ [\omega_x a_x^2]$

Chemical potential

$\mu = \sqrt{4 N_{\mathrm{at}} a_{\mathrm{sc}} \gamma \sqrt{\eta/2\pi}} = 37.508\ [\omega_x]$

Thomas-Fermi radius

$R_{\mathrm{TF}} = \sqrt{2\mu} = 8.661\ [a_x]$

Initial population fractions

$(p_0, p_1) = (0.5, 0.5)$

Raman wavelength

$\lambda_L = 790.1\ [\mathrm{nm}]$

4.2.2 Numerical Parameters

Number of grid points

$(N_x, N_y) = (512, 512)$

r-grid half-size

$(x^{\max}, y^{\max}) = (32, 32)\ [a_x]$

r-grid spacing

$(\Delta x, \Delta y) = (0.125, 0.125)\ [a_x]$

k-grid half-size

$(k_x^{\max}, k_y^{\max}) = \pi/(\Delta x, \Delta y)$
$(k_x^{\max}, k_y^{\max}) = (25.133, 25.133)\ [a_x^{-1}]$

k-grid spacing

$(\Delta k_x, \Delta k_y) = \pi/(x^{\max}, y^{\max})$

$(\Delta k_x, \Delta k_y) = (0.0982, 0.0982)\ [a_x^{-1}]$

Time scale

$\tau_0 = 1/\omega_x = 0.00318\ [\mathrm{s/rad}]$

$\tau_0 = 1\ [\omega_x^{-1}]$

Time step duration, imaginary

$\Delta\tau_{\mathrm{im}} = 1/50\ [-i\tau_0]$

Number of time steps, imaginary

$N_{\mathrm{im}} = 1000$

Time step duration, real

$\Delta\tau_{\mathrm{real}} = 1/500\ [\tau_0]$

Number of time steps, real

$N_{\mathrm{real}} = 1000$

```
import os
import sys
sys.path.insert(0, os.path.abspath('../..'))  # Adds project root to the PATH

import numpy as np

from spinor_gpe.pspinor import pspinor as spin


# 1. SETUP

DATA_PATH = 'examples/Trial_002'  # Default data path is in the /data/ folder

FREQ = 50
W = 2*np.pi*FREQ
Y_SCALE = 4
Z_SCALE = 40.0
```

(continues on next page)

(continued from previous page)

```
ATOM_NUM = 1e4
OMEG = {'x': W, 'y': Y_SCALE * W, 'z': Z_SCALE * W}
G_SC = {'uu': 1, 'dd': 1, 'ud': 0.5}

ps = spin.PSpinor(DATA_PATH, overwrite=True,
                  atom_num=ATOM_NUM,
                  omeg=OMEG,
                  g_sc=G_SC,
                  phase_factor=1,  # Complex unit phase factor on down spin
                  pop_frac=(0.5, 0.5),
                  r_sizes=(32, 32),
                  mesh_points=(512, 512))

ps.coupling_setup(wavel=790.1e-9)

ZOOM = 4  # Zooms the momentum-space density plots by a constant factor

# Plot real- and momentum-space density & real-space phase of both components
ps.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM)


# 2. RUN (Imaginary-time)

DT = 1/50
N_STEPS = 1000
DEVICE = 'cuda'
ps.rand_seed = 99999

# Run propagation loop:
# - Returns `PropResult` & `TensorPropagator` objects
res0, prop0 = ps.imaginary(DT, N_STEPS, DEVICE, is_sampling=False, n_samples=50)


# 3. ANALYZE

res0.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM)
res0.plot_total(kscale=ps.kL_recoil, zoom=ZOOM)
res0.plot_pops()


# 4. RUN (Real-time)

DT = 1/500
N_STEPS = 1000
ps.pot_eng = np.zeros_like(ps.pot_eng)  # Removes trapping potential

# Run propagation loop
res1, prop1 = ps.real(DT, N_STEPS, DEVICE, is_sampling=True, n_samples=50)


# 5. ANALYZE
```

(continues on next page)

(continued from previous page)

```
res1.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM/2)
res1.plot_total(kscale=ps.kL_recoil, zoom=ZOOM/2)
res1.plot_pops()
res1.make_movie(rscale=ps.rad_tf, kscale=ps.kL_recoil, play=True, zoom=ZOOM/2,
                norm_type='half')
```

Total running time of the script: (0 minutes 0.000 seconds)

4.3 Example 3: Raman Rabi Flopping

Starts with the Thomas-Fermi solution with all the population in one spin component. Propagates in imaginary time to the ground state. From here, configures a uniform Raman coupling which drives the population on resonance between the two components.

4.3.1 Physical Parameters

Atom number

$N_{\mathrm{at}} = 10,000$

Atomic mass, Rubidium-87

$m = 1.4442 \times 10^{-25}$ [kg]

Trap frequencies

$(\omega_x, \omega_y, \omega_z) = 2\pi \times (50, 200, 2000)$ [Hz]

$(\omega_x, \omega_y, \omega_z) = \omega_x \times (1, \gamma, \eta) = (1, 1, 40)$ [ω_x]

Harmonic oscillator length, x-axis

$a_x = \sqrt{\hbar/m\omega_x} = 1.525$ [μm]

3D scattering length, Rubidium-87

$a = 5.313$ [nm]

$a_{\mathrm{sc}} = a/a_x = 0.00348$ [a_x]

Scattering 2D scale

$g_{\rm sc}^{\rm 2D} = \sqrt{8\pi\eta}\, a_{\rm sc} = 0.1105\ [\omega_x a_x^2]$

Scattering coupling

$(g_{\rm uu}, g_{\rm dd}, g_{\rm ud}) = g_{\rm sc}^{\rm 2D} \times (1, 1, 0.0)\ [\omega_x a_x^2]$

Chemical potential

$\mu = \sqrt{4 N_{\rm at} a_{\rm sc} \gamma \sqrt{\eta/2\pi}} = 18.754\ [\omega_x]$

Thomas-Fermi radius

$R_{\rm TF} = \sqrt{2\mu} = 6.124\ [a_x]$

Initial population fractions

$(p_0, p_1) = (1.0, 0.0)$

Raman wavelength

$\lambda_L = 790.1\ [\rm nm]$

4.3.2 Numerical Parameters

Number of grid points

$(N_x, N_y) = (256, 256)$

r-grid half-size

$(x^{\rm max}, y^{\rm max}) = (16, 16)\ [a_x]$

r-grid spacing

$(\Delta x, \Delta y) = (0.125, 0.125)\ [a_x]$

k-grid half-size

$(k_x^{\max}, k_y^{\max}) = \pi/(\Delta x, \Delta y)$

$(k_x^{\max}, k_y^{\max}) = (25.133, 25.133)\ [a_x^{-1}]$

k-grid spacing

$(\Delta k_x, \Delta k_y) = \pi/(x^{\max}, y^{\max})$

$(\Delta k_x, \Delta k_y) = (0.1963, 0.1963)\ [a_x^{-1}]$

Time scale

$\tau_0 = 1/\omega_x = 0.00318\ [\mathrm{s/rad}]$

$\tau_0 = 1\ [\omega_x^{-1}]$

Time step duration, imaginary

$\Delta\tau_{\mathrm{im}} = 1/50\ [-i\tau_0]$

Number of time steps, imaginary

$N_{\mathrm{im}} = 1000$

Time step duration, real

$\Delta\tau_{\mathrm{real}} = 1/5000\ [\tau_0]$

Number of time steps, real

$N_{\mathrm{real}} = 2000$

```
import os
import sys
sys.path.insert(0, os.path.abspath('../..'))  # Adds project root to the PATH

import numpy as np
```

(continues on next page)

(continued from previous page)

```
from spinor_gpe.pspinor import pspinor as spin


# 1. SETUP

DATA_PATH = 'examples/Trial_003'  # Default data path is in the /data/ folder

FREQ = 50
W = 2*np.pi*FREQ
Y_SCALE = 1
Z_SCALE = 40.0

ATOM_NUM = 1e4
OMEG = {'x': W, 'y': Y_SCALE * W, 'z': Z_SCALE * W}
G_SC = {'uu': 1, 'dd': 1, 'ud': 0.0}
POP_FRAC = (1.0, 0.0)

ps = spin.PSpinor(DATA_PATH, overwrite=True,
                  atom_num=ATOM_NUM,
                  omeg=OMEG,
                  g_sc=G_SC,
                  pop_frac=POP_FRAC,
                  r_sizes=(16, 16),
                  mesh_points=(256, 256))

ps.coupling_setup(wavel=790.1e-9, kin_shift=True)

# Shifts the k-space density momentum peaks by `kshift_val` [`kL_recoil` units]
ps.shift_momentum(scale=1.0, frac=(0, 1.0))

# Selects the form of the coupling operator in the non-rotated reference frame
ps.rot_coupling = False

ZOOM = 2  # Zooms the momentum-space density plots by a constant factor

ps.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM)


# 2. RUN (Imaginary-time)

DT = 1/50
N_STEPS = 1000
DEVICE = 'cuda'
ps.rand_seed = 99999

res0, prop0 = ps.imaginary(DT, N_STEPS, DEVICE, is_sampling=True, n_samples=50)


# 3. ANALYZE

res0.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM)
```

(continues on next page)

(continued from previous page)

```
res0.plot_total(kscale=ps.kL_recoil, zoom=ZOOM)
res0.plot_pops()
# res0.make_movie(rscale=ps.rad_tf, kscale=ps.kL_recoil, play=True, zoom=ZOOM)
print(f'\nFinal energy: {res0.eng_final[0]} [hbar * omeg]')


# 4. RUN (Real-time)

# Initializes a uniform Raman coupling (scaled in `EL_recoil` units)
ps.coupling_uniform(1.0 * ps.EL_recoil)

DT = 1/5000
N_STEPS = 2000
res1, prop1 = ps.real(DT, N_STEPS, DEVICE, is_sampling=True, n_samples=100)


# 5. ANALYZE

res1.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM/2)
res1.plot_total(kscale=ps.kL_recoil, zoom=ZOOM/2)
res1.plot_pops()
res1.make_movie(rscale=ps.rad_tf, kscale=ps.kL_recoil, play=True, zoom=ZOOM/2)
print(f'\nFinal energy: {res1.eng_final[0]} [hbar * omeg]')
```

Total running time of the script: (0 minutes 0.000 seconds)

4.4 Example 4: Raman Detuning Gradient Ground State

Starts with the Thomas-Fermi solution. Configures a uniform Raman coupling and a linear gradient in the Raman detuning. Propagates in imaginary time, before reaching the ground state of this configuration. The detuning gradient separates the two components vertically, and the line where they interfere is a row of vortices.

4.4.1 Physical Parameters

Atom number

$N_{\mathrm{at}} = 10,000$

Atomic mass, Rubidium-87

$m = 1.4442 \times 10^{-25}$ [kg]

Trap frequencies

$(\omega_x, \omega_y, \omega_z) = 2\pi \times (50, 50, 2000)$ [Hz]

$(\omega_x, \omega_y, \omega_z) = \omega_x \times (1, \gamma, \eta) = (1, 1, 40)\ [\omega_x]$

Harmonic oscillator length, x-axis

$a_x = \sqrt{\hbar/m\omega_x} = 1.525\ [\mu\mathrm{m}]$

3D scattering length, Rubidium-87

$a = 5.313\ [\mathrm{nm}]$

$a_{\mathrm{sc}} = a/a_x = 0.00348\ [a_x]$

Scattering 2D scale

$g_{\mathrm{sc}}^{\mathrm{2D}} = \sqrt{8\pi\eta}\, a_{\mathrm{sc}} = 0.1105\ [\omega_x a_x^2]$

Scattering coupling

$(g_{\mathrm{uu}}, g_{\mathrm{dd}}, g_{\mathrm{ud}}) = g_{\mathrm{sc}}^{\mathrm{2D}} \times (1, 0.995, 0.995)\ [\omega_x a_x^2]$

Chemical potential

$\mu = \sqrt{4N_{\mathrm{at}} a_{\mathrm{sc}} \gamma \sqrt{\eta/2\pi}} = 18.754\ [\omega_x]$

Thomas-Fermi radius

$R_{\mathrm{TF}} = \sqrt{2\mu} = 6.124\ [a_x]$

Initial population fractions

$(p_0, p_1) = (0.5, 0.5)$

Raman wavelength

$\lambda_L = 790.1\ [\mathrm{nm}]$

4.4.2 Numerical Parameters

Number of grid points

$(N_x, N_y) = (256, 256)$

r-grid half-size

$(x^{\max}, y^{\max}) = (16, 16)\ [a_x]$

r-grid spacing

$(\Delta x, \Delta y) = (0.125, 0.125)\ [a_x]$

k-grid half-size

$(k_x^{\max}, k_y^{\max}) = \pi/(\Delta x, \Delta y)$
$(k_x^{\max}, k_y^{\max}) = (25.133, 25.133)\ [a_x^{-1}]$

k-grid spacing

$(\Delta k_x, \Delta k_y) = \pi/(x^{\max}, y^{\max})$
$(\Delta k_x, \Delta k_y) = (0.1963, 0.1963)\ [a_x^{-1}]$

Time scale

$\tau_0 = 1/\omega_x = 0.00318\ [\mathrm{s/rad}]$
$\tau_0 = 1\ [\omega_x^{-1}]$

Time step duration, imaginary

$\Delta\tau_{\mathrm{im}} = 1/50\ [-i\tau_0]$

Number of time steps, imaginary

$N_{\mathrm{im}} = 1000$

```
import os
import sys
```

(continues on next page)

(continued from previous page)

```
sys.path.insert(0, os.path.abspath('../..'))  # Adds project root to the PATH

import numpy as np

from spinor_gpe.pspinor import pspinor as spin

# 1. SETUP

DATA_PATH = 'examples/Trial_008'  # Default data path is in the /data/ folder

FREQ = 50
W = 2*np.pi*FREQ
Y_SCALE = 1
Z_SCALE = 40.0

ATOM_NUM = 1e4
OMEG = {'x': W, 'y': Y_SCALE * W, 'z': Z_SCALE * W}
G_SC = {'uu': 1, 'dd': 0.995, 'ud': 0.995}

ps = spin.PSpinor(DATA_PATH, overwrite=True,  # Initialize PSpinor object
                  atom_num=ATOM_NUM,
                  omeg=OMEG,
                  g_sc=G_SC,
                  pop_frac=(0.5, 0.5),
                  r_sizes=(16, 16),
                  mesh_points=(256, 256))

ps.coupling_setup(wavel=804e-9, kin_shift=True)

# Shifts the k-space density momentum peaks by `kshift_val` [`kL_recoil` units]
ps.shift_momentum(scale=0.6, frac=(0.5, 0.5))
ps.coupling_uniform(5 * ps.EL_recoil)
ps.detuning_grad(-12)

# Selects the form of the coupling operator in the rotated reference frame
ps.rot_coupling = True

ZOOM = 2  # Zooms the momentum-space density plots by a constant factor

ps.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM)


# 2. RUN (Imaginary-time)

DT = 1/50
N_STEPS = 1000
DEVICE = 'cuda'
ps.rand_seed = 99999

res, prop = ps.imaginary(DT, N_STEPS, DEVICE, is_sampling=True, n_samples=50)

# 3. ANALYZE
```

(continues on next page)

(continued from previous page)

```
res.plot_spins(rscale=ps.rad_tf, kscale=ps.kL_recoil, zoom=ZOOM)
res.plot_total(kscale=ps.kL_recoil, zoom=ZOOM)
res.plot_pops()
res.make_movie(rscale=ps.rad_tf, kscale=ps.kL_recoil, play=True, zoom=ZOOM,
               norm_type='half')
print(f'\nFinal energy: {res.eng_final[0]} [hbar * omeg]')
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER

FIVE

BENCHMARKS

These three scripts profile the execution time of certain functions within the spinor-gpe package. The first one measures the time for the main time-evolution propagation function *TensorPropagator.full_step()*. The second and third measure the execution times for the forward and inverse 2D FFT functions and the Hadamard product, respectively.

The scripts share the following physical and numerical parameters. Individual differeces in these parameters are noted in each file.

5.1 Physical Parameters

Atom number

$N_{\mathrm{at}} = 100$

Atomic mass, Rubidium-87

$m = 1.4442 \times 10^{-25}$ [kg]

Trap frequencies

$(\omega_x, \omega_y, \omega_z) = 2\pi \times (50, 50, 2000)$ [Hz]

$(\omega_x, \omega_y, \omega_z) = \omega_x \times (1, \gamma, \eta) = (1, 1, 40)$ [ω_x]

Harmonic oscillator length, x-axis

$a_x = \sqrt{\hbar/m\omega_x} = 1.525$ [μm]

3D scattering length, Rubidium-87

$a = 5.313$ [nm]

$a_{\rm sc} = a/a_x = 0.00348\ [a_x]$

Scattering 2D scale

$g_{\rm sc}^{\rm 2D} = \sqrt{8\pi\eta}\, a_{\rm sc} = 0.1105\ [\omega_x a_x^2]$

Scattering coupling

$(g_{\rm uu}, g_{\rm dd}, g_{\rm ud}) = g_{\rm sc}^{\rm 2D} \times (1, 1, 1.04)\ [\omega_x a_x^2]$

Chemical potential

$\mu = \sqrt{4 N_{\rm at} a_{\rm sc} \gamma \sqrt{\eta/2\pi}} = 1.875\ [\omega_x]$

Thomas-Fermi radius

$R_{\rm TF} = \sqrt{2\mu} = 1.937\ [a_x]$

Initial population fractions

$(p_0, p_1) = (0.5, 0.5)$

Raman wavelength

$\lambda_L = 790.1\ [\rm nm]$

5.2 Numerical Parameters

Number of grid points

$(N_x^{(\rm min)}, N_y^{(\rm min)}) = (64, 64)$ to $(4096, 4096)$

r-grid half-size

$(x^{\rm max}, y^{\rm max}) = (8, 8)\ [a_x]$

r-grid spacing

$(\Delta x, \Delta y) = (0.25, 0.25)$ to $(0.003906, 0.003906)$ $[a_x]$

k-grid half-size

$(k_x^{\max}, k_y^{\max}) = \pi/(\Delta x, \Delta y)$

$(k_x^{\max}, k_y^{\max}) = (12.566, 12.566)$ to $(804.25, 804.25)$ $[a_x^{-1}]$

k-grid spacing

$(\Delta k_x, \Delta k_y) = \pi/(x^{\max}, y^{\max})$

$(\Delta k_x, \Delta k_y) = (0.3927, 0.3927)$ $[a_x^{-1}]$

Time scale

$\tau_0 = 1/\omega_x = 0.00318$ [s/rad]

$\tau_0 = 1$ $[\omega_x^{-1}]$

Time step duration, imaginary

$\Delta\tau_{\text{im}} = 1/50$ $[-i\tau_0]$

Number of time steps, imaginary

$N_{\text{im}} = 1$

5.2.1 FFT and iFFT Time

On a given system and hardware configuration, times the FFT and iFFT function calls for increasing mesh grid sizes.

```
import os
import sys
import timeit
sys.path.insert(0, os.path.abspath('../..'))  # Adds project root to the PATH

import numpy as np
import torch
from scipy.stats import median_abs_deviation as mad

from spinor_gpe.pspinor import pspinor as spin
```

(continues on next page)

(continued from previous page)

```
from spinor_gpe.pspinor import tensor_tools as ttools

torch.cuda.empty_cache()

grids = [(64, 64),
         (64, 128),
         (128, 128),
         (128, 256),
         (256, 256),
         (256, 512),
         (512, 512),
         (512, 1024),
         (1024, 1024),
         (1024, 2048),
         (2048, 2048),
         (2048, 4096),
         (4096, 4096)]
n_grids = len(grids)
meas_times = [[0] for i in range(n_grids)]
repeats = np.zeros(n_grids)
size = np.zeros(n_grids)


DATA_PATH = 'benchmarks/Bench_001'  # Default data path is in the /data/ folder

W = 2 * np.pi * 50
ATOM_NUM = 1e2
OMEG = {'x': W, 'y': W, 'z': 40 * W}
G_SC = {'uu': 1, 'dd': 1, 'ud': 1.04}

DEVICE = 'cuda'
COMPUTER = 'Acer Aspire'

for i, grid in enumerate(grids):
    print(i)
    try:
        ps = spin.PSpinor(DATA_PATH, overwrite=True,
                          atom_num=ATOM_NUM, omeg=OMEG, g_sc=G_SC,
                          pop_frac=(0.5, 0.5), r_sizes=(8, 8),
                          mesh_points=grid)

        ps.coupling_setup(wavel=790.1e-9, kin_shift=False)

        res, prop = ps.imaginary(1/50, 1, DEVICE, is_sampling=False)

        stmt = """ttools.fft_2d(prop.psik, prop.space['dr'])"""

        timer = timeit.Timer(stmt=stmt, globals=globals())

        N = timer.autorange()[0]
        if N < 10:
            N *= 10
```

(continues on next page)

(continued from previous page)

```
        vals = timer.repeat(N, 1)
        meas_times[i] = vals
        repeats[i] = N
        size[i] = np.log2(np.prod(grid))

        torch.cuda.empty_cache()
    except RuntimeError as ex:
        print(ex)
        break

median = np.array([np.median(times) for times in meas_times])
med_ab_dev = np.array([mad(times, scale='normal') for times in meas_times])

tag = 'fft\\' + COMPUTER + '_' + DEVICE + '_fft'
np.savez(ps.paths['data'] + '..\\' + tag, computer=COMPUTER, device=DEVICE,
         size=size, n_repeats=repeats, med=median, mad=med_ab_dev)

np.save(ps.paths['data'] + '..\\' + tag, np.array(meas_times, dtype='object'))
```

```
for i, grid in enumerate(grids):
    print(i)
    try:
        ps = spin.PSpinor(DATA_PATH, overwrite=True,
                          atom_num=ATOM_NUM, omeg=OMEG, g_sc=G_SC,
                          pop_frac=(0.5, 0.5), r_sizes=(8, 8),
                          mesh_points=grid)

        ps.coupling_setup(wavel=790.1e-9, kin_shift=False)

        res, prop = ps.imaginary(1/50, 1, DEVICE, is_sampling=False)

        stmt = """ttools.ifft_2d(prop.psik, prop.space['dr'])"""

        timer = timeit.Timer(stmt=stmt, globals=globals())

        N = timer.autorange()[0] * 10
        vals = timer.repeat(N, 1)
        meas_times[i] = vals
        repeats[i] = N
        size[i] = np.log2(np.prod(grid))

        torch.cuda.empty_cache()
    except RuntimeError as ex:
        print(ex)
        break

median = np.array([np.median(times) for times in meas_times])
med_ab_dev = np.array([mad(times, scale='normal') for times in meas_times])

tag = COMPUTER + '_' + DEVICE + '_ifft'
np.savez('data\\' + tag, computer=COMPUTER, device=DEVICE,
         size=size, n_repeats=repeats, med=median, mad=med_ab_dev)
```

(continues on next page)

(continued from previous page)

```
np.save(ps.paths['data'] + '..\\' + tag, np.array(meas_times, dtype='object'))
```

Total running time of the script: (0 minutes 0.000 seconds)

5.2.2 Hadamard Time

On a given system and hardware configuration, times Hadamard product for increasing mesh grid sizes.

```
import os
import sys
import timeit
import math
sys.path.insert(0, os.path.abspath('../..'))  # Adds project root to the PATH

import numpy as np
import torch
from scipy.stats import median_abs_deviation as mad
from tqdm import tqdm

from spinor_gpe.pspinor import pspinor as spin
from spinor_gpe.pspinor import tensor_tools as ttools

torch.cuda.empty_cache()


def closestDivisors(n):
    """Return the two largest divisors that are closest together."""
    b = round(math.sqrt(n))
    while n % b > 0:
        b -= 1
    return b, n // b


closest = np.vectorize(closestDivisors)

a = np.logspace(12, 24, 25, base=2.0, dtype=int)
print(a)
print(closest(a))
close = closest(a)
grids = [(a, b) for a, b in zip(close[0], close[1])]
```

```
n_grids = len(grids)
meas_times = [[0] for i in range(n_grids)]
repeats = np.zeros(n_grids)
size = np.zeros(n_grids)


DATA_PATH = 'benchmarks/Bench_001'  # Default data path is in the /data/ folder

W = 2 * np.pi * 50
```

(continues on next page)

(continued from previous page)

```
ATOM_NUM = 1e2
OMEG = {'x': W, 'y': W, 'z': 40 * W}
G_SC = {'uu': 1, 'dd': 1, 'ud': 1.04}

DEVICE = 'cuda'
COMPUTER = 'Acer Aspire'

ps = spin.PSpinor(DATA_PATH, overwrite=True,
                  atom_num=ATOM_NUM, omeg=OMEG, g_sc=G_SC,
                  pop_frac=(0.5, 0.5), r_sizes=(8, 8),
                  mesh_points=grids[0])

for i, grid in tqdm(enumerate(grids)):
    print(f'{i}/{len(grids)}')
    try:
        psi = [torch.rand(grid, device=DEVICE, dtype=torch.complex128),
               torch.rand(grid, device=DEVICE, dtype=torch.complex128)]
        op = [torch.rand(grid, device=DEVICE, dtype=torch.float64),
              torch.rand(grid, device=DEVICE, dtype=torch.float64)]

        stmt = """x = [o * p for o, p in zip(op, psi)]"""

        timer = timeit.Timer(stmt=stmt, globals=globals())

        N = timer.autorange()[0]
        if N < 10:
            N *= 10
        vals = timer.repeat(N, 1)
        meas_times[i] = vals
        repeats[i] = N
        size[i] = np.log2(np.prod(grid))

        torch.cuda.empty_cache()
    except RuntimeError as ex:
        print(ex)
        break

median = np.array([np.median(times) for times in meas_times])
med_ab_dev = np.array([mad(times, scale='normal') for times in meas_times])

tag = COMPUTER + '_' + DEVICE + '_had'
np.savez('data\\' + tag, computer=COMPUTER, device=DEVICE,
         size=size, n_repeats=repeats, med=median, mad=med_ab_dev)

np.save(ps.paths['data'] + '..\\' + tag, np.array(meas_times, dtype='object'))
```

Total running time of the script: (0 minutes 0.000 seconds)

5.2.3 Propagation Time

For a given system and hardware configuration (CPU or GPU), this script generates *PSpinor* objects of increasing mesh grid size in a for-loop, starting from (64, 64) up to (4096, 4096), or until the memory limit is reached. After generating the *PSpinor*, it performs a brief propagation in imaginary time, returning a *TensorPropagator* object *prop*. The script then measures the time for a single function call of *prop.full_step()*, and repeats that N times. With the measured times for each grid size, saves the median and median absolute deviation.

```
import os
import sys
import timeit

import numpy as np
import torch
from scipy.stats import median_abs_deviation as mad

sys.path.insert(0, os.path.abspath('../../..'))  # Adds project root to PATH

from spinor_gpe.pspinor import pspinor as spin

torch.cuda.empty_cache()
grids = [(64, 64),
         (64, 128),
         (128, 128),
         (128, 256),
         (256, 256),
         (256, 512),
         (512, 512),
         (512, 1024),
         (1024, 1024),
         (1024, 2048),
         (2048, 2048),
         (2048, 4096),
         (4096, 4096)]
grids = [grids[0]]
n_grids = len(grids)
meas_times = [[0] for i in range(n_grids)]
repeats = np.zeros(n_grids)
size = np.zeros(n_grids)


DATA_PATH = 'benchmarks/Bench_001'  # Default data path is in the /data/ folder

W = 2 * np.pi * 50
ATOM_NUM = 1e2
OMEG = {'x': W, 'y': W, 'z': 40 * W}
G_SC = {'uu': 1, 'dd': 1, 'ud': 1.04}

DEVICE = 'cuda'
COMPUTER = 'Acer Aspire'

for i, grid in enumerate(grids):
    print(i)
```

(continues on next page)

(continued from previous page)

```
    try:
        # Create a PSpinor object
        ps = spin.PSpinor(DATA_PATH, overwrite=True,
                          atom_num=ATOM_NUM, omeg=OMEG, g_sc=G_SC,
                          pop_frac=(0.5, 0.5), r_sizes=(8, 8),
                          mesh_points=grid)
        ps.coupling_setup(wavel=790.1e-9, kin_shift=False)
        res, prop = ps.imaginary(1/50, 1, DEVICE, is_sampling=False)

        stmt = """prop.full_step()"""  # A full time step function call.

        # Create a code timing object.
        timer = timeit.Timer(stmt=stmt, globals=globals())

        # Estimate the number of timing repetitions to make
        N = timer.autorange()[0]
        if N < 10:
            N *= 10

        vals = timer.repeat(N, 1)  # Time and repeat N times.

        # Save timing values.
        meas_times[i] = vals
        repeats[i] = N
        size[i] = np.log2(np.prod(grid))
        torch.cuda.empty_cache()
    except RuntimeError as ex:
        print(ex)
        break
```

```
median = np.array([np.median(times) for times in meas_times])
med_ab_dev = np.array([mad(times, scale='normal') for times in meas_times])

tag = COMPUTER + '_' + DEVICE
np.savez('bench_data\\' + tag, computer=COMPUTER, device=DEVICE,
         size=size, n_repeats=repeats, med=median, mad=med_ab_dev)

np.save(ps.paths['data'] + '..\\' + tag, np.array(meas_times, dtype='object'))
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER

SIX

PSPINOR PACKAGE

6.1 Submodules

6.2 pspinor.pspinor module

class spinor_gpe.pspinor.pspinor.**PSpinor**(*path*, *omeg=None*, *g_sc=None*, *mesh_points=(256, 256)*, *r_sizes=(16, 16)*, *atom_num=10000.0*, *pop_frac=(0.5, 0.5)*, ***kwargs*)

Bases: `object`

A GPU-compatible simulator of the pseudospin-1/2 GPE.

Contains the functionality to run a real- or imaginary-time propataion of the pseudospin-1/2 Gross-Pitaevskii equation. Contains methods to generate the required energy and spatial grids. Also has methods to generate the grids for the momentum-(in)dependent coupling between spin components, corresponding to an (RF) Raman coupling interaction.

The dominant length scale is in terms of the harmonic oscillator length along the x-direction *a_x*. The dominant energy scale is the harmonic trapping energy along the x-direction [hbar * *omeg['x']*].

paths
Essential paths and directory names for a given trial.

- 'folder' : The name of the topmost trial directory.
- 'data' : Path to the simulation data & results.
- 'code' : Path to the subdirectory containing the trial code.
- 'trial' : Path to the subdirectory containing raw trial data.

Type `dict`

atom_num
Total atom number.

Type `float`

space
Spatial arrays, meshes, spacings, volume elements, and sizes:

KEYS				
Arrays:	‘x’	‘y’	‘kx’	‘ky’
Meshes:	‘x_mesh’	‘y_mesh’	‘kx_mesh’	‘ky_mesh’
Spacings:	‘dr’		‘dk’	
Vol. elem.:	‘dv_r’		‘dv_k’	
Sizes:	‘r_sizes’		‘k_sizes’	
Other:	‘mesh_points’			

Type dict

pop_frac
The initial population fraction in each spin component.

Type iterable

omeg
Angular trapping frequencies, {‘x’, ‘y’, ‘z’}. omeg[‘x’] multiplied by \hbar is the characteristic energy scale.

Type dict

g_sc
Relative scattering interaction strengths, {‘uu’, ‘dd’, ‘ud’}.

Type dict

a_x
Harmonic oscillator length along the x-axis; this is the characteristic length scale.

Type float

chem_pot
Chemical potential, [\hbar * omeg[‘x’]].

Type float

rad_tf
Thomas-Fermi radius along the x-axis, [a_x].

Type float

time_scale
Inverse x-axis trapping frequency - the characteristic time scale, [1 / omeg[‘x’]].

Type float

pot_eng_spin
A list of 2D potential energy grids for each spin component, [\hbar * omeg[‘x’]].

Type list of array

kin_eng_spin
A list of 2D kinetic energy grids for each spin component, [\hbar * omeg[‘x’]].

Type list of array

psi
A list of 2D real-space spin component wavefunctions; generally complex.

Type list of array

psik
A list of 2D momentum-space spin component wavefunctions; generally complex.

Type list of array

is_coupling
Signals the presence of direct coupling between spin components.

Type bool

kL_recoil
The value of the single-photon recoil momentum, [1 / a_x].

Type float

EL_recoil
The energy scale corresponding to the single-photon recoil momentum, [1 / omeg['x']].

Type float

rand_seed
Value to seed the pseudorandom number generator.

Type int

prop
Object containing the results of propagation, along with analysis methods.

Type PropResult

rot_coupling
Option to place coupling in a rotating reference frame, i.e. no momentum shift on the coupling operation.

Type bool, default=True

compute_energy_grids()
Compute the initial potential and kinetic energy grids.

Assumes that the BEC is in a harmonic trap. This harmonic potential determines the initial 'Thomas-Fermi' density profile of the BEC. pot_eng can be modified prior to progation to have any arbitrary potential energy landscape.

Also assumes that the BEC has a simple free-particle kinetic energy dispersion. If using a momentum-dependent spin coupling, this grid will be modified later.

compute_spatial_grids(*mesh_points=(256, 256)*, *r_sizes=(16, 16)*)
Compute the real and momentum space grids.

Stored in the dict *space* are the real- and momentum-space mesh grids, mesh sizes, mesh spacings, volume elements, and the corresponding linear arrays.

Parameters

- **mesh_points** (iterable of int, default=(256, 256)) – The number of grid points along the x- and y-axes, respectively.
- **r_sizes** (iterable of int, default=(16, 16)) – The half size of the grid along the real x- and y-axes, respectively,in units of [a_x].

compute_tf_params(*species='Rb87'*)
Compute parameters and scales for the Thomas-Fermi solution.

Parameters **sepecies** (str, default='Rb87') – Designates the atomic species and corresponding physical data used in the simulations.

compute_tf_psi(*phase_factor=1.0*)
Compute the intial pseudospinor wavefunction *psi* and FFT *psik*.

The psuedospinor wavefunction *psi* that is generated is a `list` of 2D NumPy arrays. The Thomas-Fermi solution is real and has the form of an inverted parabaloid. *psik* is a `list` of the 2D FFT of *psi*'s components.

Parameters **phase_factor** (`complex`, default=1.0) – Unit complex number; initial relative phase factor between the two spin components.

property coupling

Get the *coupling* attribute.

2D coupling array [\hbar * omeg['x']].

coupling_grad(*slope*, *offset*, *axis=1*)

Generate a linear gradient of the interspin coupling strength.

Convenience function for generating linear gradients of the coupling. *coupling* can also be set to any arbitrary NumPy array directly:

```
>>> ps = PSpinor()
>>> ps.coupling_setup()
>>> ps.coupling = np.exp(-ps.x_mesh**2 / 2)  # Gaussian function
```

Note: When working with Raman recoil units [E_L], they will first need to be converted to [hbar*omeg_x] units.

Parameters

- **slope** (`float`) – The slope of the coupling gradient, in [hbar*omeg_x/a_x].
- **offset** (`float`) – The origin offset of the coupling gradient, in [hbar*omeg_x].
- **axis** (`int`, optional) – The axis along which the coupling gradient runs.

coupling_setup(*wavel=7.901e-07*, *scale=1.0*, *kin_shift=False*)

Calculate parameters for the momentum-(in)dependent coupling.

If `kin_shift=True`, the kinetic energy grid receives a spin-dependent shift in momentum.

Parameters

- **wavel** (`float`, default=790.1e-9) – Wavelength of Raman coupling. Note that you must specify the wavelength in meters.
- **scale** (`float`, default=1.0) – The relative scale of recoil momentum. Interesting physics may be simulated by considering a recoil momentum that is hypothetically much larger or much smaller than the native wavelength recoil momentum.
- **kin_shift** (`bool`, default=False) – Option for a momentum-(in)dependent coupling.

coupling_uniform(*value*)

Generate a uniform interspin coupling strength.

Convenience function for generating unirom gradients of the coupling. *coupling* can also be set to any arbitrary NumPy array directly.

Parameters **value** (`float`) – The value of the coupling, in [hbar*omega_x].

See also:

coupling_grad Coupling gradient

property detuning
Get the *detuning* attribute.

2D detuning array [\hbar * omeg['x']].

detuning_grad(*slope*, *offset=0.0*, *axis=1*)
Generate a linear gradient of the interspin coupling strength.

Convenience function for generating linear gradients of the coupling. *detuning* can also be set to any arbitrary NumPy array directly:

```
>>> ps = PSpinor()
>>> ps.coupling_setup()
>>> ps.detuning = np.sin(2 * np.pi * ps.x_mesh)  # Sin function
```

Note: when working with Raman recoil units [E_L], they will first need to be converted to [hbar*omeg_x] units.

Parameters

- **slope** (float) – The slope of the detuning gradient, in [hbar*omeg_x/a_x].
- **offset** (float) – The origin offset of the detuning gradient, in [hbar*omeg_x].
- **axis** (int, optional) – The axis along which the detuning gradient runs.

See also:

coupling_grad Coupling gradient

detuning_uniform(*value*)
Generate a uniform coupling detuning.

Convenience function for generating unirom gradients of the coupling. *detuning* can also be set to any arbitrary NumPy array directly.

Parameters **value** (float) – The value of the coupling, in [hbar*omega_x].

See also:

coupling_grad Coupling gradient

imaginary(*t_step*, *n_steps=1000*, *device='cpu'*, *is_sampling=False*, *n_samples=1*)
Perform imaginary-time propagation.

Propagation is carried out in a *TensorPropagator* object. The results are stored and returned in the *PropResult* object for further analysis.

Parameters

- **t_step** (float) – The propagation time step.
- **n_steps** (int, optional) – The total number of propagation steps.
- **device** (str, optional) – {'cpu', 'cuda'}
- **is_sampling** (bool, optional) – Option to sample wavefunctions throughout the propagation.
- **n_samples** (int, optional) – The number of samples to collect.

property kin_eng
Get the *kin_eng* attribute.

2D kinetic energy grid, [\hbar * omeg['x']]0

no_coupling_setup()
Provide the default parameters for no coupling.

plot_kdens(*psik=None*, *spin=None*, *cmap='viridis'*, *scale=1.0*)
Plot the momentum-space density of the wavefunction.

Shows the momentum-space density of either the up (*spin=0*), down (*spin=1*), or both (*spin=None*) spin components.

Parameters

- **psik** (list of Numpy array, optional.) – The wavefunction to plot. If no *psik* is supplied, then it uses the object attribute *self.psik*.
- **spin** (int or *None*, optional) – Which spin to plot. *None* plots both spins. 0 or 1 plots only the up or down spin, respectively.
- **cmap** (str, default='viridis') – The matplotlib colormap to use for the plots.
- **scale** (float, default=1.0) – A factor by which to scale the spatial dimensions, e.g. Thomas- Fermi radius.

See also:

plotting_tools.plot_dens Density plots.

plot_rdens(*psi=None*, *spin=None*, *cmap='viridis'*, *scale=1.0*)
Plot the real-space density of the wavefunction.

Shows the real-space density of either the up (*spin=0*), down (*spin=1*), or both (*spin=None*) spin components.

Parameters

- **psi** (list of Numpy array, optional.) – The wavefunction to plot. If no *psi* is supplied, then it uses the object attribute *self.psi*.
- **spin** (int or *None*, optional) – Which spin to plot. *None* plots both spins. 0 or 1 plots only the up or down spin, respectively.
- **cmap** (str, default='viridis') – The matplotlib colormap to use for the plots.
- **scale** (float, default=1.0) – A factor by which to scale the spatial dimensions, e.g. Thomas- Fermi radius.

See also:

plotting_tools.plot_dens Density plots.

plot_rphase(*psi=None*, *spin=None*, *cmap='twilight_shifted'*, *scale=1.0*)
Plot the real-space phase of the wavefunction.

Shows the real-space phase of either the up (*spin=0*), down (*spin=1*), or both (*spin=None*) spin components.

Parameters

- **psi** (list of Numpy array, optional.) – The wavefunction to plot. If no *psi* is supplied, then it uses the object attribute *self.psi*.

- **spin** (int or *None*, optional) – Which spin to plot. *None* plots both spins. 0 or 1 plots only the up or down spin, respectively.
- **cmap** (str, default='twilight_shifted') – The matplotlib colormap to use for the plots.
- **scale** (float, default=1.0) – A factor by which to scale the spatial dimensions, e.g. Thomas- Fermi radius.

See also:

plotting_tools.plot_phase Phase plots.

plot_spins(*rscale=1.0*, *kscale=1.0*, *cmap='viridis'*, *save=True*, *ext='.pdf'*, *zoom=1.0*)
Plot the densities (both real & k) and phases of spin components.

Parameters

- **rscale** (float, optional) – Real-space length scale. The default of 1.0 corresponds to the naturatl harmonic length scale along the x-axis.
- **kscale** (float, optional) – Momentum-space length scale. The default of 1.0 corresponds to the inverse harmonic length scale along the x-axis.
- **cmap** (str, optional) – Color map name for the real- and momentum-space density plots.
- **save** (bool, optional) – Saves the figure as a .pdf file (default). The filename has the format "*/data_path/*pop_evolution%s-*trial_name*.pdf".
- **ext** (str, optional) – Saved plot image file extension.
- **zoom** (float, optional) – A zoom factor for the k-space density plot.

property pot_eng
Get the *pot_eng* attribute.

2D potential energy grid, [\hbar * omeg['x']].

real(*t_step*, *n_steps=1000*, *device='cpu'*, *is_sampling=False*, *n_samples=1*)
Perform real-time propagation.

Propagation is carried out in a *TensorPropagator* object. The results are stored and returned in the *PropResult* object for further analysis.

Parameters

- **t_step** (float) – The propagation time step.
- **n_steps** (int, optional) – The total number of propagation steps.
- **device** (str, optional) – {'cpu', 'cuda'}
- **is_sampling** (bool, optional) – Option to sample wavefunctions throughout the propagation.
- **n_samples** (int, optional) – The number of samples to collect.

seed_random_vortices(*N*)
Seed randomly-arranged vortices into the wavefunction.

seed_regular_vortices()
Seed regularly-arranged vortices into the wavefunction.

These seed-vortex functions might be moved to the ttools module.

seed_vortices(*positions*, *windings*)
Seeds vortices at the positions specified.

Pass a list of tuple coordinates

Parameters

- **positions** (list of tuple) – The positions at which to seed the vortices.
- **windings** (int or list of int) – Must have values of +1 or -1. If a single int is supplied, all vortices will have the same winding. If a list is supplied, it must have the same length as *positions*.
- **TODO** (*#*) –
- **component.** (*# in each spinor*) –
- **???** (*#*) –

setup_data_path(*path*, *overwrite*)
Create new data directory to store simulation data & results.

Parameters

- **path** (str) – The name of the directory to save the simulation. If *path* does not represent an absolute path, then the data is stored in spinor-gpe/data/*path*.
- **overwrite** (bool) – Gives the option to overwrite existing data sub-directories

shift_momentum(*psik=None*, *scale=1.0*, *frac=(0.5, 0.5)*)
Shifts momentum components of *psi* by a fracion of +/- kL_recoil.

The ground-state solutions of Raman-coupled spinor systems in general have spinor components with both left- and right-moving momentum peaks. Providing a manual shift on the momentum-space wavefunction components better approximates these solutions, i.e. faster convergence in imaginary time propagation.

Parameters

- **psik** (list of NumPy array, optional.) – The momentum-space pseudospinor wavefunction. If *psik* is not provided, then this function uses the current class attribute *self.psik*.
- **scale** (float, default=1.0) – By default, the function shifts the momentum peaks by a single unit of recoil momenta *kL_recoil*. *scale* gives the option of scaling the shift larger or smaller for faster convergence.
- **frac** (iterable, default=(0.5, 0.5)) – The fraction of each spinor component's momentum peak to shift in either direction. frac=(0.5, 0.5) splits into two equal peaks, while (0.0, 1.0) and (1.0, 0.0) move the entire peak one direction or the other.

6.3 pspinor.tensor_propagator module

class spinor_gpe.pspinor.tensor_propagator.**TensorPropagator**(*spin*, *t_step*, *n_steps*, *device='cpu'*, *time='imag'*, *is_sampling=False*, *n_samples=1*)

Bases: object

CPU- or GPU-compatible propagator of the GPE, with tensors.

n_steps
The total number of full time steps in propagation.

Type int

device
The computing device on which propagation is performed

Type str

paths
See pspinor.Pspinor.

Type dict

t_step
Duration of the full time step.

Type float or complex

dt_out
Duration of the outer time sub-step.

Type float or complex

dt_in
Duration of the inner time sub-step.

Type float or complex

rand_seed
See pspinor.Pspinor.

Type int

is_sampling
Option to sample the wavefunction periodically throughout propagation.

Type bool

atom_num
See pspinor.Pspinor.

Type float

is_coupling
See pspinor.Pspinor.

Type bool

g_sc
See *pspinor.Pspinor*.

Type dict of Tensor

kin_eng_spin
See pspinor.Pspinor.

Type list of Tensor

pot_eng_spin
See pspinor.Pspinor.

Type list of Tensor

psik
See *pspinor.Pspinor*.

Type list of Tensor

space

See *pspinor.Pspinor*. Contains only keys: {'dr', 'dk', 'x_mesh', 'y_mesh', 'dv_r', 'dv_k'}

Type dict of Tensor

coupling
See *pspinor.Pspinor*.

Type Tensor

kL_recoil
See pspinor.Pspinor.

Type float

expon
The exponential argument on the coupling operator off-diagonals. If the coupling is in a rotated reference frame, then `expon`=0.0.

Type Tensor

sample_rate
How often wavefunctions are sampled.

Type int

eng_out
Pre-computed energy evolution operators for the outer time sub-step.

Type dict of Tensor

eng_in
Pre-computed energy evolution operators for the inner time sub-step.

Type dict of Tensor

eng_expect(*psik*)
Compute the energy expectation value of the wavefunction.

To calculate the kinetic energy potion of the energy expectation value, spatial gradients of the phase and square root density must be obtained.

Parameters **psik** (list of NumPy array) – The k-space representation of wavefunction to evaluate.

Notes

While spatial gradients of the wavefunction's phase can be computed with PyTorch tensors, there is currently not an implementation of the 2D phase-unwrapping algorithm. For this this reason, the energy expectation value needs to be computed with NumPy arrays.

full_step()
Full step forward in real or imaginary time.

For accuracy, divide the full propagation step into three single steps using the magic gamma time steps.

prop_loop(*n_steps*)
Evaluate the propagation steps in a for-loop.

Saves the spin populations at every time step. If wavefunctions are sampled throughout the propagation, they are saved with the associated sampled times in *trial_data/psik_sampled%s_`folder_name*.npz.

Parameters **n_steps** (int) – The number of propagation steps.

Returns **result** – Contains the propagation results and analysis methods.

Return type `PropResult`

See also:

`spinor_gpe.prop_results` Propagation results

`single_step`(*t_step*, *eng*)

Single step forward in real or imaginary time with spectral method.

The kinetic, interaction, and coupling time-evolution operators are symmetrically split into two half-single steps around the full-single step potential energy operator.

Parameters

- **t_step** (`float`) – The sub-time step.
- **eng** (`dict`) – The kinetic and potential energy evolution operators corresponding to the given sub-time step.

6.4 pspinor.prop_result module

`class spinor_gpe.pspinor.prop_result.PropResult`(*psi_final*, *psik_final*, *eng_final*, *pops*, *sampled_path=None*)

Bases: `object`

The result of propagation, with plotting and analysis tools.

`psi`

The final real-space wavefunctions.

Type `list` of `array`

`psik`

The final momentum-space wavefunctions.

Type `list` of `array`

`eng_final`

The energy expectation values: [<total>, <kin.>, <pot.>, <int.>].

Type `list`

`pops`

Times and populations at every time step, {'times', 'vals'}.

Type `dict` of `array`

`sampled_path`

Path to the .npz file where the sampled wavefunctions and times are stored for this result.

Type `str`

`dens`

The final real-space densities.

Type `list` of `array`

`densk`

The final momentum-space densities.

Type `list` of `array`

phase
The final real-space phases.

Type list of array

paths
See pspinor.PSpinor.

Type dict

time_scale
See pspinor.PSpinor.

Type float

space
See tensor_propagator.TensorPropagator.

Type dict of array

analyze_vortex()
Compute the total vorticity in each spin component.

calc_separation()
Calculate the phase separation of the two spin components.

make_movie(*rscale=1.0*, *kscale=1.0*, *cmap='viridis'*, *play=False*, *zoom=1.0*, *norm_type='all'*)
Generate a movie of the wavefunctions' densities and phases.

Parameters

- **rscale** (float, optional) – Real-space length scale. The default of 1.0 corresponds to the naturatl harmonic length scale along the x-axis.
- **kscale** (float, optional) – Momentum-space length scale. The default of 1.0 corresponds to the inverse harmonic length scale along the x-axis.
- **cmap** (str, optional) – Color map name for the real- and momentum-space density plots.
- **play** (bool, default=False) – If True, the movie is opened in the computer's default media player after it is saved.
- **kzoom** (float, optional) – A zoom factor for the k-space density plot.
- **norm_type** (str, optional) – {'all', 'half'} Normalizes the colormaps to the full or half sum of the max densites. 'half' is useful for visualizing situations where the population is equally divided between the two spins.

plot_eng()
Plot the sampled energy expectation values.

plot_pops(*scaled=True*, *save=True*, *ext='.pdf'*)
Plot the spin populations as a function of propagation time.

Parameters

- **scaled** (bool, optional) – If *scaled* is True then the time-axis will be rescaled into proper time units. Otherwise, it's left in dimensionless time units.
- **save** (bool, optional) – Saves the figure as a .pdf file (default). The filename has the format "*/data_path*/pop_evolution%s-*trial_name*.pdf".
- **ext** (str, optional) – File extension for the saved plot image.

plot_spins(*rscale=1.0*, *kscale=1.0*, *cmap='viridis'*, *save=True*, *ext='.pdf'*, *show=True*, *zoom=1.0*)
Plot the densities (real & k) and phases of spin components.

Parameters

- **rscale** (float, default=1.0) – Real-space length scale. The default of 1.0 corresponds to the naturatl harmonic length scale along the x-axis.
- **kscale** (float, default=1.0) – Momentum-space length scale. The default of 1.0 corresponds to the inverse harmonic length scale along the x-axis.
- **cmap** (str, default='viridis') – Matplotlib color map name for the real- and momentum-space density plots.
- **save** (bool, default=True) – Saves the figure as a .pdf file (default). The filename has the format "*/data_path*/spin_dens_phase%s-*trial_name*.pdf".
- **ext** (str, default='.pdf') – File extension for the saved plot image.
- **zoom** (float, default=1.0) – A zoom factor for the k-space density plot.

plot_total(*rscale=1.0*, *kscale=1.0*, *cmap='viridis'*, *save=True*, *ext='.pdf'*, *show=True*, *zoom=1.0*)

Plot the total real-space density and phase of the wavefunction.

Parameters

- **rscale** (float, default=1.0) – Real-space length scale. The default of 1.0 corresponds to the naturatl harmonic length scale along the x-axis.
- **kscale** (float, default=1.0) – Momentum-space length scale. The default of 1.0 corresponds to the inverse harmonic length scale along the x-axis.
- **cmap** (str, default='viridis') – Color map name for the real- and momentum-space density plots.
- **save** (bool, default=True) – Saves the figure as a .pdf file (default). The filename has the format "*/data_path*/pop_evolution%s-*trial_name*.pdf".
- **ext** (str, default='.pdf') – File extension for the saved plot image.
- **show** (bool, default=True) – Option to display the generated image.
- **zoom** (float, default = 1.0) – A zoom factor for the k-space density plot.

rebin(*arr*, *new_shape=(256, 256)*)

Rebin a 2D *arr* to shape *new_shape* by averaging.

This may be used when generating movies of sampled wavefunctions. By down-sampling the density grids, the movie is generated much faster.

Parameters

- **arr** (2D list or NumPy array) – The input 2D array to rebin.
- **new_shape** (iterable, default=(256, 256)) – The target rebinned shape.

6.5 pspinor.tensor_tools module

tensor_tools.py module.

spinor_gpe.pspinor.tensor_tools.**calc_atoms**(*psi*, *vol_elem=1.0*)

Calculate the total number of atoms.

Parameters

- **psi** (list of 2D NumPy array or PyTorch Tensor) – The input spinor wavefunction.

- **vol_elem** (float) – 2D volume element of the space.

Returns **atom_num** – The total atom number in both spin components.

Return type float

spinor_gpe.pspinor.tensor_tools.**calc_pops**(*psi*, *vol_elem=1.0*)
Calculate the populations in each spin component.

Parameters

- **psi** (list of 2D NumPy array or PyTorch Tensor) – The input spinor wavefunction.
- **vol_elem** (float) – 2D volume element of the space.

Returns **pops** – The atom number in each spin component.

Return type list of float

spinor_gpe.pspinor.tensor_tools.**conj**(*psi*)
Complex conjugate of a complex tensor.

spinor_gpe.pspinor.tensor_tools.**conj_comp**(*psi_comp*)
Complex conjugate of a single wavefunction component.

spinor_gpe.pspinor.tensor_tools.**coupling_op**(*t_step*, *coupling=None*, *expon=tensor(0)*)
Compute the time-evolution operator for the coupling term.

Parameters

- **t_step** (float) – Sub-time step.
- **coupling** (2D PyTorch complex Tensor, optional.) – The coupling mesh. Default is none if *self.is_coupling = False*. In this case, the evolution operator returned will the identity matrix.
- **expon** (2D PyTorch real Tensor, optional.) – The exponential argument in the bare coupling term. If there is no coupling, then this is 0 by default.

Returns **coupl_ev_op**

Return type 2D PyTorch Tensor, optional.

spinor_gpe.pspinor.tensor_tools.**density**(*psi*)
Compute the density of a spinor wavefunction.

Parameters **psi** (list of 2D NumPy array or PyTorch Tensor) – The input spinor wavefunction.

Returns **dens** – The density of each component's wavefunction.

Return type NumPy array, PyTorch Tensor, or list thereof

spinor_gpe.pspinor.tensor_tools.**evolution_op**(*t_step*, *energy*)
Compute the unitary time-evolution operator for the given energy.

Parameters

- **energy** –
- **time_step** –

spinor_gpe.pspinor.tensor_tools.**expect_val**(*psi*)
Compute the expectation value of the supplied spatial operator.

spinor_gpe.pspinor.tensor_tools.**fft_1d**(*psi*, *delta_r=(1, 1)*, *axis=0*) → list
Compute the forward 1D FFT of *psi* along a single axis.

Parameters

- **psi** (list of NumPy array or PyTorch Tensor) – The input wavefunction.
- **delta_r** (NumPy array, default=(1,1)) – A two-element list of the real-sapce x- and y-mesh spacings, respectively.
- **axis** (int, default=0) – The axis along which to transform; note that 0 -> y-axis, and 1 -> x-axis.

Returns **psik_axis** – The FFT of psi along *axis*.

Return type list of NumPy array or PyTorch Tensor

spinor_gpe.pspinor.tensor_tools.**fft_2d**(*psi*, *delta_r=(1, 1)*) → list
Compute the forward 2D FFT of *psi*.

Parameters

- **psi** (list of NumPy array or PyTorch Tensor) – The input wavefunction.
- **delta_r** (NumPy array, default=(1,1)) – A two-element list of the real-space x- and y-mesh spacings, respectively. Typically, use *ps.space['dr']*.

Returns **psik** – The k-space FFT of the input wavefunction.

Return type list of NumPy array or PyTorch Tensor

spinor_gpe.pspinor.tensor_tools.**grad**(*psi*, *delta_r*)
Compute the spatial gradient of a wavefunction list.

Parameters

- **psi** –
- **delta_r** –

spinor_gpe.pspinor.tensor_tools.**grad_comp**(*psi_comp*, *delta_r*)
Spatial gradient of a single wavefunction component.

Raises **TypeError** – If *psi_comp* is neither an array or a Tensor of the correct shape.

spinor_gpe.pspinor.tensor_tools.**grad_sq**(*psi*, *delta_r*)
Compute the gradient squared of a wavefunction.

spinor_gpe.pspinor.tensor_tools.**grad_sq_comp**(*psi_comp*, *delta_r*)
Take a list of tensors or np arrays; checks type.

spinor_gpe.pspinor.tensor_tools.**ifft_1d**(*psik*, *delta_r=(1, 1)*, *axis=0*) → list
Compute the inverse 1D FFT of *psi* along a single axis.

Parameters

- **psik** (list of NumPy array or PyTorch Tensor) – The input wavefunction.
- **delta_r** (NumPy array, default=(1,1)) – A two-element list of the real-sapce x- and y-mesh spacings, respectively.
- **axis** (int, default=0) – The axis along which to transform; note that 0 -> x-axis, and 1 -> y-axis.

Returns **psi_axis** – The FFT of psi along *axis*.

Return type list of NumPy array or PyTorch Tensor

spinor_gpe.pspinor.tensor_tools.**ifft_2d**(*psik*, *delta_r=(1, 1)*) → list
Compute the inverse 2D FFT of *psik*.

Parameters

- **psik** (list of NumPy array or PyTorch Tensor) – The input wavefunction.
- **delta_r** (NumPy array, default=(1,1)) – A two-element list of the real-sapce x- and y-mesh spacings, respectively. Typically, use *ps.space['dr']*.

Returns **psi** – The real-space FFT of the input wavefunction.

Return type list of NumPy array or PyTorch Tensor

spinor_gpe.pspinor.tensor_tools.**inner_prod**()
Calculate the inner product of two wavefunctions.

spinor_gpe.pspinor.tensor_tools.**norm**(*psi*, *vol_elem*, *atom_num*, *pop_frac=None*)
Normalize spinor wavefunction to the expected atom numbers and populations.

This function normalizes to the total expected atom number *atom_num*, and to the expected population fractions *pop_frac*. Normalization is essential in processes where the total atom number is not conserved, (e.g. imaginary time propagation).

Parameters

- **psi** (list of NumPy arrays or PyTorch Tensors.) – The wavefunction to normalize.
- **vol_elem** (float) – Volume element for either real- or k-space.
- **atom_num** (int) – The total expected atom number.
- **pop_frac** (*array-like, optional*) – The expected population fractions in each spin component.

Returns

- **psi_norm** (list of NumPy arrays or PyTorch Tensors.) – The normalized wavefunction.
- **dens_norm** (list of NumPy arrays or PyTorch Tensors.) – The densities of the normalized wavefunction's components

spinor_gpe.pspinor.tensor_tools.**norm_sq**(*psi_comp*)
Compute the density (norm-squared) of a single wavefunction component.

Parameters **psi_comp** (NumPy array or PyTorch Tensor) – A single wavefunction component.

Returns **psi_sq** – The norm-square of the wavefunction.

Return type NumPy array or PyTorch Tensor

Raises **TypeError** – If *psi_comp* is neither an array or a Tensor of the correct shape.

spinor_gpe.pspinor.tensor_tools.**phase**(*psi*, *uwrap=False*, *dens=None*)
Compute the phase of a real-space spinor wavefunction.

Parameters **psi** (list of 2D NumPy array or PyTorch Tensor) – The input spinor wavefunction.

Returns **phase** – The phase of each component's wavefunction.

Return type NumPy array, PyTorch Tensor, or list thereof

spinor_gpe.pspinor.tensor_tools.**phase_comp**(*psi_comp*, *uwrap=False*, *dens=None*)
Compute the phase (angle) of a single complex wavefunction component.

Parameters **psi_comp** (NumPy array or PyTorch Tensor) – A single wavefunction component.

Returns **angle** – The phase (angle) of the component's wavefunction.

Return type NumPy array or PyTorch Tensor

spinor_gpe.pspinor.tensor_tools.**prod**(*factors*)
General function for multiplying the elements of a 1D data structure.

Operates similar to the *sum* function from the standard library.

spinor_gpe.pspinor.tensor_tools.**to_cpu**(*input_tens*)
Transfers *input_tens* from GPU to CPU memory.

Parameters **input_tens** (PyTorch `Tensor` or `list` of PyTorch `Tensor`) – Input tensor stored on GPU memory.

Returns **output_tens** – Output tensor stored on CPU memory.

Return type PyTorch `Tensor` or `list` of PyTorch `Tensor`

spinor_gpe.pspinor.tensor_tools.**to_gpu**(*input_tens*, *dev='cuda'*)
Transfers *input_tens* from CPU to GPU memory.

Parameters

- **input_tens** (PyTorch `Tensor` or `list` of PyTorch `Tensor`) – Input tensor stored on GPU memory.
- **dev** (`str`, default='cuda') – The name of the device on which to store the tensor, e.g. {'cuda', 'cuda:0'}

Returns **output_tens**

Return type PyTorch `Tensor` or `list` of PyTorch `Tensor`

spinor_gpe.pspinor.tensor_tools.**to_numpy**(*input_tens*)
Convert from PyTorch Tensor to NumPy arrays.

Accepts a single PyTorch Tensor, or a `list` of PyTorch Tensor, as in the wavefunction objects.

Parameters **input_tens** (PyTorch `Tensor`, or `list` of PyTorch `Tensor`) – Input tensor, or list of tensor, to be converted to `array`, on CPU memory.

Returns **output_arr** – Output array stored on CPU memory.

Return type NumPy `array` or `list` of NumPy `array`

spinor_gpe.pspinor.tensor_tools.**to_tensor**(*input_arr*, *dev='cpu'*, *dtype=64*)
Convert from NumPy arrays to Tensors.

Accepts a single NumPy array, or a `list` of NumPy arrays, as in the wavefunction objects.

Parameters

- **input_arr** (NumPy `array`, or `list` of NumPy `array`) – Input array, or list of arrays, to be converted to a `Tensor`, on either CPU or GPU memory.
- **dev** (`str`, default='cpu') – The name of the device on which to store the tensor, e.g. {'cpu', 'cuda', 'cuda:0'}
- **dtype** (`int`, default=64) – Designator for the torch dtype -
  - 32 : `torch.float32`;
  - 64 : `torch.float64`;
  - 128 : `torch.complex128`

Returns **output_tens** – Output tensor of *dtype* stored on *dev* memory.

Return type PyTorch `Tensor` or `list` of PyTorch `Tensor`

6.6 pspinor.plotting_tools module

plotting_tools.py module.

spinor_gpe.pspinor.plotting_tools.**next_available_path**(*file_name*, *trial_name*, *ext=''*)
Test for the next available path for a given file.

Parameters

- **file_name** (str) – The base file path to test.
- **trial_name** (str) – The name of the trial to append to the end of the file name.
- **ext** (str, default=”) – File extention.

Returns **test_path** – The file path with the next available index.

Return type str

spinor_gpe.pspinor.plotting_tools.**plot_dens**(*psi*, *spin=None*, *cmap='viridis'*, *scale=1.0*, *extent=None*)
Plot the real or k-space density of the wavefunction.

Based on the value of the *spin* parameter, this function will plot either the up (0), down (1), or both (None) spin components of the spinor wavefunction.

Parameters

- **psi** (list of Numpy array, optional.) – The wavefunction to plot. If no *psi* is supplied, then it uses the object attribute *self.psi*.
- **spin** (int or *None*, optional) – Which spin to plot. *None* plots both spins. 0 or 1 plots only the up or down spin, respectively.
- **cmap** (str, default=’viridis’) – The matplotlib colormap to use for the plots.
- **scale** (float, optional) – A factor to scale the spatial dimensions by, e.g. Thomas-Fermi radius.
- **extent** (*iterable*) – The spatial extent of the wavefunction components, in the format np.array([x_min, x_max, y_min, y_max]). Determines the natural spatial scale of the plot.

spinor_gpe.pspinor.plotting_tools.**plot_phase**(*psi*, *spin=None*, *cmap='twilight_shifted'*, *scale=1*, *extent=None*)
Plot the phase of the real wavefunction.

Based on the value of the *spin* parameter, this function will plot either the up (0), down (1), or both (None) spin components of the spinor wavefunction.

Parameters

- **psi** (list of Numpy array, optional.) – The wavefunction to plot.
- **spin** (int or *None*, optional) – Which spin to plot. *None* plots both spins. 0 or 1 plots only the up or down spin, respectively.
- **cmap** (str, optional) – The colormap to use for the plots.
- **scale** (float, optional) – A factor to scale the spatial dimensions by, e.g. Thomas-Fermi radius.
- **extent** (*iterable*) – The spatial extent of the wavefunction components, in the format np.array([x_min, x_max, y_min, y_max]). Determines the natural spatial scale of the plot.

spinor_gpe.pspinor.plotting_tools.**plot_spins**(*psi*, *psik*, *extents*, *paths*, *cmap='viridis'*, *save=True*, *ext='.pdf'*, *show=True*, *zoom=1.0*)

Plot the densities (real & k) and phases of spin components.

In total, six subplots are generated. Each pair of axes are stored together in a list, which is returned in *all_plots*.

Parameters

- **psi** (list of Numpy array, optional.) – The real-space wavefunction to plot.
- **psik** (list of Numpy array, optional.) – The momentum-space wavefunction to plot.
- **extents** (dict of iterable) – The dictionary keys are {'r', 'k'}, and each value is a 4-element iterables giving the x- (kx-) and y- (ky-) spatial extents of the plot area, e.g. [x_min, x_max, y_min, y_max]
- **paths** (dict of str) – The dictionary keys contain {'data', 'folder'}, and the values are absolute paths to the saved data path and its containing folder.
- **cmap** (str, default='viridis') – Matplotlib color map name for the real- and momentum-space density plots.
- **save** (bool, default=True) – Saves the figure as a .pdf file (default). The filename has the format "*/data_path*/spin_dens_phase%s-*trial_name*.pdf".
- **ext** (str, default='.pdf') – File extension for the saved density plots.
- **zoom** (float, default=1.0) – A zoom factor for the k-space density plot.

Returns

- **fig** (plt.Figure) – The matplotlib figure for the plot.
- **all_plots** (dict of list) – The keys are {'r', 'ph', 'k'}. Each value is a pair of matplotlib.image.AxesImage for both spins.

spinor_gpe.pspinor.plotting_tools.**plot_total**(*psi*, *psik*, *extents*, *paths*, *cmap='viridis'*, *save=True*, *ext='.pdf'*, *show=True*, *zoom=1.0*)

Plot the total densities and phase of the wavefunction.

Parameters

- **psi** (list of Numpy array, optional.) – The real-space wavefunction to plot.
- **psik** (list of Numpy array, optional.) – The momentum-space wavefunction to plot.
- **extents** (dict of iterable) – The dictionary keys are {'r', 'k'}, and each value is a 4-element iterables giving the x- (kx-) and y- (ky-) spatial extents of the plot area, e.g. [x_min, x_max, y_min, y_max]
- **paths** (dict of str) – The dictionary keys contain {'data', 'folder'}, and the values are absolute paths to the saved data path and its containing folder.
- **cmap** (str, default='viridis') – Matplotlib color map name for the real- and momentum-space density plots.
- **save** (bool, default=True) – Saves the figure as a .pdf file (default). The filename has the format "*/data_path*/spin_dens_phase%s-*trial_name*.pdf".
- **ext** (str, default='.pdf') – File extension for the saved density plots.
- **zoom** (float, default=1.0) – A zoom factor for the k-space density plot.

Returns

- **fig** (plt.Figure) – The matplotlib figure for the plot.

- **all_plots** (dict of matplotlib.image.AxesImage) – The keys are {'r', 'ph', 'k'}. Each value is a separate matplotlib.image.AxesImage.

spinor_gpe.pspinor.plotting_tools.**progress_message**(*frame*, *n_total*)
Display an updating progress message while the animation is saving.

This function produces an output similar to what the tqdm package gives. This one works in situations where tqdm cannot be applied.

Parameters

- **frame** (int) – The current frame/index number in the loop.
- **n_total** (int) – The total number of frames in the loop.

spinor_gpe.pspinor.plotting_tools.**time_remaining**(*frame*, *n_total*, *its*)
Calculate completion time in the progress_message function.

Parameters

- **frame** (int) – The current frame/index number in the loop.
- **n_total** (int) – The total number of frames in the loop.
- **its** (float) – The time in seconds between successive iterations.

6.7 Module contents

Created on Fri Apr 9 13:35:19 2021

@author: benjamin

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

spinor_gpe.pspinor, 56
spinor_gpe.pspinor.plotting_tools, 54
spinor_gpe.pspinor.tensor_tools, 49

INDEX

A

a_x (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
analyze_vortex() (*spinor_gpe.pspinor.prop_result.PropResult method*), 48
atom_num (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 37
atom_num (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45

C

calc_atoms() (*in module spinor_gpe.pspinor.tensor_tools*), 49
calc_pops() (*in module spinor_gpe.pspinor.tensor_tools*), 50
calc_separation() (*spinor_gpe.pspinor.prop_result.PropResult method*), 48
chem_pot (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
compute_energy_grids() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 39
compute_spatial_grids() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 39
compute_tf_params() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 39
compute_tf_psi() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 39
conj() (*in module spinor_gpe.pspinor.tensor_tools*), 50
conj_comp() (*in module spinor_gpe.pspinor.tensor_tools*), 50
coupling (*spinor_gpe.pspinor.pspinor.PSpinor property*), 40
coupling (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 46
coupling_grad() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 40
coupling_op() (*in module spinor_gpe.pspinor.tensor_tools*), 50
coupling_setup() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 40
coupling_uniform() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 40

D

dens (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 47
density() (*in module spinor_gpe.pspinor.tensor_tools*), 50
densk (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 47
detuning (*spinor_gpe.pspinor.pspinor.PSpinor property*), 40
detuning_grad() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 41
detuning_uniform() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 41
device (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 44
dt_in (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
dt_out (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45

E

EL_recoil (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 39
eng_expect() (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator method*), 46
eng_final (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 47
eng_in (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 46
eng_out (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 46
evolution_op() (*in module spinor_gpe.pspinor.tensor_tools*), 50
expect_val() (*in module spinor_gpe.pspinor.tensor_tools*), 50
expon (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 46

F

fft_1d() (*in module spinor_gpe.pspinor.tensor_tools*), 50
fft_2d() (*in module spinor_gpe.pspinor.tensor_tools*), 51
full_step() (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator method*), 46

G

g_sc (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
g_sc (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
grad() (*in module spinor_gpe.pspinor.tensor_tools*), 51
grad_comp() (*in module spinor_gpe.pspinor.tensor_tools*), 51
grad_sq() (*in module spinor_gpe.pspinor.tensor_tools*), 51
grad_sq_comp() (*in module spinor_gpe.pspinor.tensor_tools*), 51

I

ifft_1d() (*in module spinor_gpe.pspinor.tensor_tools*), 51
ifft_2d() (*in module spinor_gpe.pspinor.tensor_tools*), 51
imaginary() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 41
inner_prod() (*in module spinor_gpe.pspinor.tensor_tools*), 52
is_coupling (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 39
is_coupling (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
is_sampling (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45

K

kin_eng (*spinor_gpe.pspinor.pspinor.PSpinor property*), 41
kin_eng_spin (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
kin_eng_spin (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
kL_recoil (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 39
kL_recoil (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 46

M

make_movie() (*spinor_gpe.pspinor.prop_result.PropResult method*), 48
module
    spinor_gpe.pspinor, 56
    spinor_gpe.pspinor.plotting_tools, 54
    spinor_gpe.pspinor.tensor_tools, 49

N

n_steps (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 44
next_available_path() (*in module spinor_gpe.pspinor.plotting_tools*), 54
no_coupling_setup() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 42
norm() (*in module spinor_gpe.pspinor.tensor_tools*), 52
norm_sq() (*in module spinor_gpe.pspinor.tensor_tools*), 52

O

omeg (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38

P

paths (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 48
paths (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 37
paths (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
phase (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 47
phase() (*in module spinor_gpe.pspinor.tensor_tools*), 52
phase_comp() (*in module spinor_gpe.pspinor.tensor_tools*), 52
plot_dens() (*in module spinor_gpe.pspinor.plotting_tools*), 54
plot_eng() (*spinor_gpe.pspinor.prop_result.PropResult method*), 48
plot_kdens() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 42
plot_phase() (*in module spinor_gpe.pspinor.plotting_tools*), 54
plot_pops() (*spinor_gpe.pspinor.prop_result.PropResult method*), 48
plot_rdens() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 42
plot_rphase() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 42
plot_spins() (*in module spinor_gpe.pspinor.plotting_tools*), 54
plot_spins() (*spinor_gpe.pspinor.prop_result.PropResult method*), 48
plot_spins() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 43
plot_total() (*in module spinor_gpe.pspinor.plotting_tools*), 55
plot_total() (*spinor_gpe.pspinor.prop_result.PropResult method*), 49

pop_frac (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
pops (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 47
pot_eng (*spinor_gpe.pspinor.pspinor.PSpinor property*), 43
pot_eng_spin (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
pot_eng_spin (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
prod() (*in module spinor_gpe.pspinor.tensor_tools*), 52
progress_message() (*in module spinor_gpe.pspinor.plotting_tools*), 56
prop (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 39
prop_loop() (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator method*), 46
PropResult (*class in spinor_gpe.pspinor.prop_result*), 47
psi (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 47
psi (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
psik (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 47
psik (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
psik (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
PSpinor (*class in spinor_gpe.pspinor.pspinor*), 37

R

rad_tf (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
rand_seed (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 39
rand_seed (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
real() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 43
rebin() (*spinor_gpe.pspinor.prop_result.PropResult method*), 49
rot_coupling (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 39

S

sample_rate (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 46
sampled_path (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 47
seed_random_vortices() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 43
seed_regular_vortices() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 43
seed_vortices() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 43
setup_data_path() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 44
shift_momentum() (*spinor_gpe.pspinor.pspinor.PSpinor method*), 44
single_step() (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator method*), 47
space_propagator (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 48
space (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 37
space (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
spinor_gpe.pspinor module, 56
spinor_gpe.pspinor.plotting_tools module, 54
spinor_gpe.pspinor.tensor_tools module, 49

T

t_step (*spinor_gpe.pspinor.tensor_propagator.TensorPropagator attribute*), 45
TensorPropagator (*class in spinor_gpe.pspinor.tensor_propagator*), 44
time_remaining() (*in module spinor_gpe.pspinor.plotting_tools*), 56
time_scale (*spinor_gpe.pspinor.prop_result.PropResult attribute*), 48
time_scale (*spinor_gpe.pspinor.pspinor.PSpinor attribute*), 38
to_cpu() (*in module spinor_gpe.pspinor.tensor_tools*), 53
to_gpu() (*in module spinor_gpe.pspinor.tensor_tools*), 53
to_numpy() (*in module spinor_gpe.pspinor.tensor_tools*), 53
to_tensor() (*in module spinor_gpe.pspinor.tensor_tools*), 53