magpylib Documentation

Release 3.0.2

Michael Ortner

Jun 27, 2021

Content:

1 When can you use Magpylib ? 2

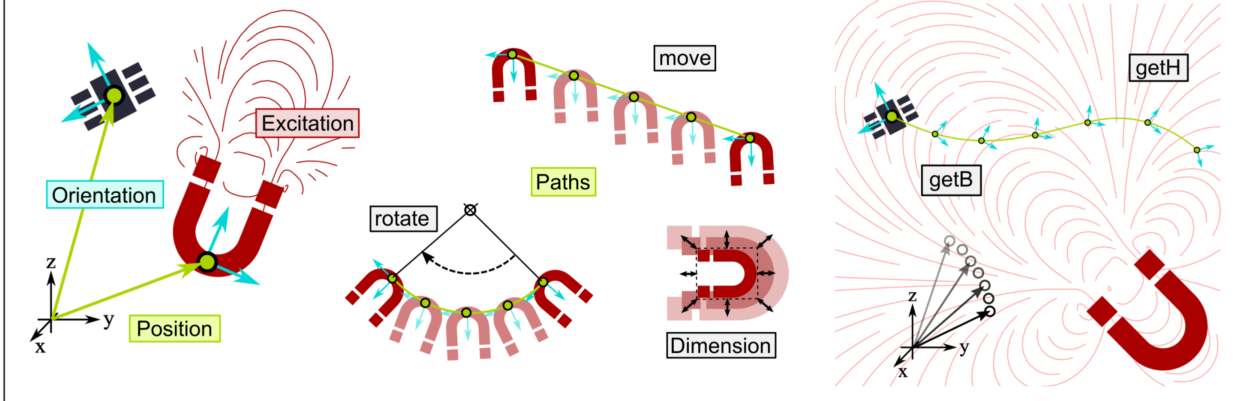
2 Quickstart 3

3 Index 100

Python Module Index 101

Index 102

- Python package for calculating 3D static magnetic fields of magnets (e.g. Cuboids, Cylinders, Spheres, ...), line currents (e.g. straight or loops) and other sources (e.g. Dipoles).
- The fields are computed using analytical solutions which makes the computation very fast (fully vectorized)
- The field computation is coupled to a geometry interface (position, orientation, paths) which makes it convenient to determine relative motion between sources and observers.



CHAPTER 1

When can you use Magpylib ?

The analytical solutions are exact when there is no material response. In permanent magnets, when (remanent) permeabilities are below $\mu_r < 1.1$ the error is typically below 1% (long magnet shapes are better, large distance from magnet is better). For more details check out the physics section.

Magpylib is at its best when dealing with air-coils (no eddy currents) and high grade permanent magnet assemblies (Ferrite, NdFeB, SmCo or similar materials).

CHAPTER 2

Quickstart

Install Magpylib with pip or conda:

`> pip install magpylib.`

`> conda install magpylib.`

This **Example code** calculates the magnetic field of a cylindrical magnet.

```
import magpylib as mag3
s = mag3.magnet.Cylinder(magnetization=(0,0,350), dimension=(4,5))
observer_pos = (4,4,4)
print(s.getB(observer_pos))

# Output: [ 5.08641867  5.08641867 -0.60532983]
```

A cylinder shaped permanent magnet with diameter and height of 4 and 5 millimeter, respectively, is created in a global coordinate system with cylinder axis parallel to the z-axis and geometric magnet center in the origin. The magnetization is homogeneous and points in z-direction with an amplitude of 350 millitesla. The magnetic field is calculated in units of millitesla at the observer position (4,4,4) in units of millimeter.

2.1 Documentation v3.0.0

Note: Magpylib v3 documentation is work in progress. For now, please refer to the library docstrings and the old documentation.

2.2 Installation

2.2.1 Dependencies

Magpylib works with Python 3.7 or later ! The following packeges will be automatically installed, or updated. See Git Hub for respective versions. Packages will never be downgraded.

- numpy
- matplotlib
- scipy (.spatial.transform, .special)

2.2.2 Using a package manager

Magpylib works with PyPI and conda-forge repositories.

Install with pip,

```
pip install magpylib
```

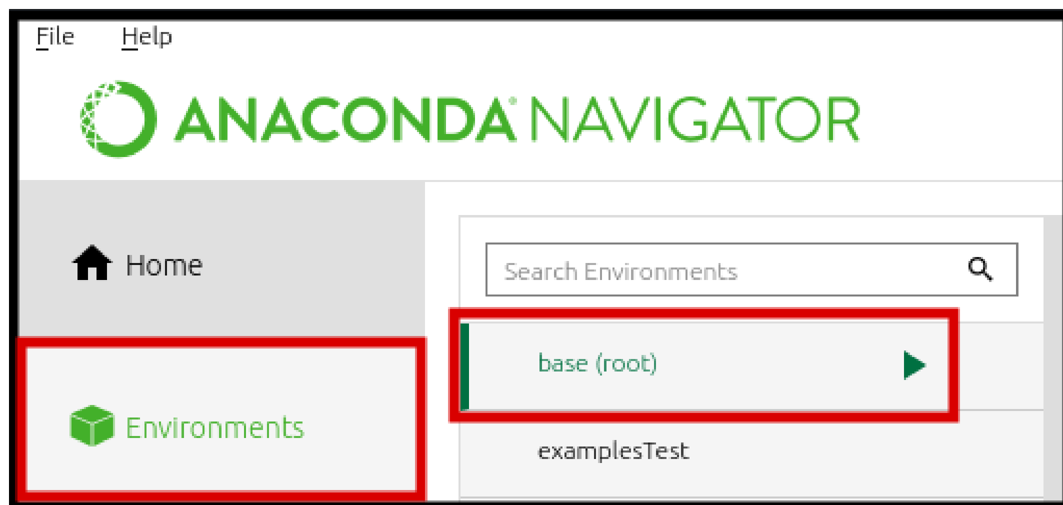
or with conda

```
conda install magpylib
```

2.2.3 Using Anaconda

Or if you have little experience with Python we recommand using Anaconda.

1. Download & install Anaconda3
2. Start Anaconda Navigator
3. On the interface, go to *Environments* and choose the environment you wish to install magpylib in. For this example, we will use the base environment:

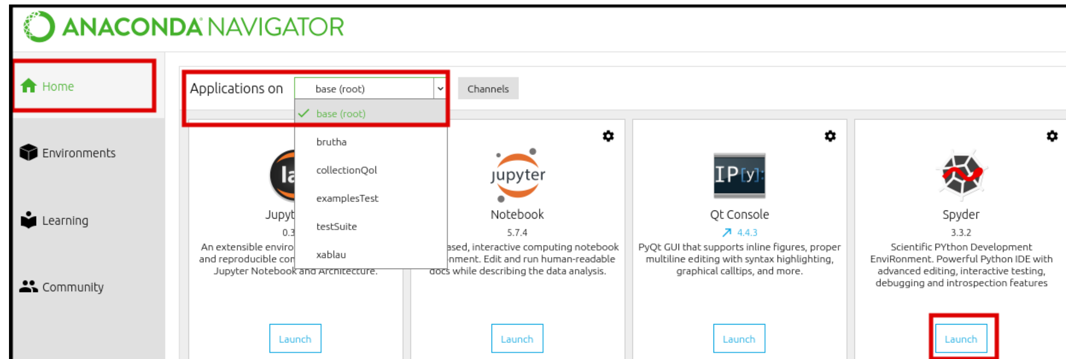


4. Click the arrow, and open the conda terminal

5. Input the following to install from conda-forge:

```
conda install -c conda-forge magpylib
```

6. Dont forget to select the proper environment in your IDE.



2.2.4 Download Sites

Currently magpylib is hosted at:

- Conda Cloud
- Python Package Index
- GitHub repository

2.3 Example Codes

This section includes a few code examples that show how the library can be used and what i can be used for. A detailed technical library documentation can be found in the *Documentation v3.0.0*.

Note: Magpylib v3 examples are WIP - for now, please refer to library docstrings and old example codes.

2.4 MATLAB Integration

Note: MATLAB does not support Tkinter, which disables matplotlib. This means that `displaySystem()` will not generate a display and might interrupt the program.

2.4.1 Setting Python Interpreter

As of version R2015b, MATLAB allows you to call libraries from other programming languages, including Python, which enables users to run magpylib from the MATLAB interface. The following guide intends to provide a digest of

the Official MATLAB documentation with a focus on utilizing this interface with magpylib.

Running `>>> pyversion` following line in the MATLAB console tells you which Python environment (user space interpreter) is connected to your MATLAB interface.

If magpylib is already installed in this environment you can directly call it, as shown in the *Example* below. If not please follow the *Installation* instructions and install magpylib.

If you choose to install magpylib in a different environment than the one that is currently connected to your MATLAB interpreter, use the following command in the MATLAB console to connect the new environment instead (choose correct path pointing at your Python interpreter).

```
>>> pyversion C:\Users\...\AppData\Local\Continuum\anaconda3\envs\magpy\python.exe
```

2.4.2 Example

The following MATLAB 2019 script showcases most functionalities.

```
%%%%%%%%%%%%%%%%%% magpytest.m %%%%%%%%%%%%%%
%% Showcase Python + MATLAB Interoperability.
%% Define and calculate the field of a
%% Cuboid magnet inside a Collection.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Import the library
py.importlib.import_module("magpylib")

%% Define Python types for input
vec3 = py.list({1,2,3})
scalar = py.int(90)

%% Define input
mag = vec3
dim = vec3
angle = scalar
sensorPos = vec3

%% Execute Python
% 2 positional and 1 keyword argument in Box
box = py.magpylib.source.magnet.Box(mag,dim,pyargs('angle',angle))
col = py.magpylib.Collection(box)
pythonResult = col.getB(sensorPos)

%% Convert Python Result to MATLAB data format
matlabResult = double(pythonResult)
```

Note: With old versions of Matlab the *double(pythonResult)* type conversion might give an error message.

2.5 Credits, Contribution & Citation

2.5.1 Maintainers & Contact

Michael Ortner - Concept, Physics, Coding.

- michael.ortner@silicon-austria.com
- Silicon Austria Labs, Sensors division, 9500 Villach, Austria

Lucas Gabriel Coliado Bandeira - Software engineering

- lucascoliado@hotmail.com

Jaka Pribosek - Concepts.

Alexandre Boisselet - Python code layout.

2.5.2 Credits

We want to thank a lot of ppl who have helped to realize and advance this project over the years. The project was supported by CTR-AG and is now supported by the Silicon Austria Labs public research center.

2.5.3 Contributions

We welcome any feedback (Bug reports, feature requests, comments, really anything) via email magpylib@gmail.com or through gitHub channels.

2.5.4 Citation

We are thankful for any reference and citation through the original publication.

A valid bibtex entry would be

```
@Article{magpylib2020,
title = {Magpylib: A free Python package for magnetic field computation},
author  ={Ortner, Michael and Coliado Bandeira, Lucas Gabriel},
year    = {2020},
journal = {SoftwareX},
publisher = {Elsevier},
doi = {10.1016/j.softx.2020.100466}
}
```

2.5.5 License

Magpylib is published under the open source AGPL v3 license. If you are interested in a non-disclosure private license, please contact us at magpylib@gmail.com.

2.6 Physics & Computation

2.6.1 The analytical solutions

Permanent Magnets

Magnetic field computations in Magpylib are based on known analytical solutions (formulas) to permanent magnet and current problems. For Magpylib we have used the following references:

- Field of cuboid magnets: [1999Yang, 2005Engel-Herbert, 2013Camacho]

- Field of cylindrical magnets: [1994Furlani, 2009Derby]
- Field of facet bodies: [2009Janssen, 2013Rubeck]
- Field of circular line current: [1950Smythe, 2001Simpson, 2017Ortner]
- all others derived by hand

A short reflection on how these formulas can be achieved: In magnetostatics (no currents) the magnetic field becomes conservative (Maxwell: $\nabla \times \mathbf{H} = 0$) and can thus be expressed through the magnetic scalar potential Φ_m:

$$\mathbf{H} = -\nabla \cdot \Phi_m$$

The solution to this equation can be expressed by an integral over the magnetization distribution $\mathbf{M}(\mathbf{r})$ as

$$\Phi_m(\mathbf{r}) = \frac{1}{4\pi} \int_{V'} \frac{\nabla' \cdot \mathbf{M}(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} dV' + \frac{1}{4\pi} \oint_{S'} \frac{\mathbf{n}' \cdot \mathbf{M}(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} dS'$$

where $\mathbf{r}$ denotes the position, V is the magnetized volume with surface S and normal vector $\mathbf{n}$ onto the surface. This solution is derived in detail e.g. in [1999Jackson].

Currents

The fields of currents are directly derived using the law of Biot-Savart with the current distribution $\mathbf{J}(\mathbf{r})$:

$$\mathbf{B}(\mathbf{r}) = \frac{\mu_0}{4\pi} \int_{V'} \mathbf{J}(\mathbf{r}') \times \frac{\mathbf{r} - \mathbf{r}'}{|\mathbf{r} - \mathbf{r}'|^3} dV'$$

In some special cases (simple shapes, homogeneous magnetizations and current distributions) the above integrals can be worked out directly to give analytical formulas (or simple, fast converging series). The derivations can be found in the respective references. A noteworthy comparison between the Coulombian approach and the Amperian current model is given in [2009Ravaud].

References

- [1999Yang] Z. J. Yang et al., “Potential and force between a magnet and a bulk Y1Ba2Cu3O7-d superconductor studied by a mechanical pendulum”, Superconductor Science and Technology 3(12):591, 1999
- [2005 Engel-Herbert] R. Engel-Herbert et al., Journal of Applied Physics 97(7):074504 - 074504-4 (2005)
- [2013 Camacho] J.M. Camacho and V. Sosa, “Alternative method to calculate the magnetic field of permanent magnets with azimuthal symmetry”, Revista Mexicana de Fisica E 59 8–17, 2013
- [1994Furlani] E. P. Furlani, S. Reanik and W. Janson, “A Three-Dimensional Field Solution for Bipolar Cylinders”, IEEE Transaction on Magnetics, VOL. 30, NO. 5, 1994
- [2009Derby] N. Derby, “Cylindrical Magnets and Ideal Solenoids”, arXiv:0909.3880v1, 2009
- [1950Smythe] W.B. Smythe, “Static and dynamic electricity” McGraw-Hill New York, 1950, vol. 3.
- [2001Simpson] J. Simplson et al., “Simple analytic expressions for the magnetic field of a circular current loop,” 2001.
- [2017Ortner] M. Ortner et al., “Feedback of Eddy Currents in Layered Materials for Magnetic Speed Sensing”, IEEE Transactions on Magnetics (Volume: 53, Issue: 8, Aug. 2017)
- [2009Janssen] J.L.G. Janssen, J.J.H. Paulides and E.A. Lomonova, “3D ANALYTICAL FIELD CALCULATION USING TRIANGULAR MAGNET SEGMENTS APPLIED TO A SKEWED LINEAR PERMANENT MAGNET ACTUATOR”, ISEF 2009 - XIV International Symposium on Electromagnetic Fields in Mechatronics, Electrical and Electronic Engineering Arras, France, September 10-12, 2009
- [2013Rubeck] C. Rubeck et al., “Analytical Calculation of Magnet Systems: Magnetic Field Created by Charged Triangles and Polyhedra”, IEEE Transactions on Magnetics, VOL. 49, NO. 1, 2013

- [1999Jackson] J. D. Jackson, "Classical Electrodynamics", 1999 Wiley, New York
- [2009Ravaud] R. Ravaud and G. Lamarquand, "Comparison of the coulombian and amperian current models for calculating the magnetic field produced by radially magnetized arc-shaped permanent magnets", HAL Id: hal-00412346

2.6.2 Accuracy of the Solutions and Demagnetization

Line currents:

The magnetic field of a wire carrying a homogeneous current density is similar (ON THE OUTSIDE ONLY) to the field of a line current in the center of the wire, which carries the total current of the wire. Current distributions become inhomogeneous at bends of the wire or when eddy currents (finite frequencies) are involved.

Magnets and Demagnetization

The anayltical solutions are exact when bodies have a homogeneous magnetization. However, real materials always have a material response which results in an inhomogeneous magnetization even when the initial magnetization is perfectly homogeneous. There is a lot of literature on such demagnetization effects.

Modern high grade permanent magnets (NdFeB, SmCo, Ferrite) have a very weak material responses (local slope of the magnetization curve, remanent permeability) of the order of $\mu_r \approx 1.05$. In this case the analyical solutions provide an excellent approximation with less than 1% error even at close distance from the magnet surface. A detailed error analysis and discussion is presented in the appendix of [2020Malago].

Soft-Magnetic Materials

Soft-magnetic materials like iron or steel with large permeabilities $\mu_r \sim 1000$ can in principle not be modeled with Magpylib. However, when the body is static, when there is no strong local interaction with an adjacent magnet and when the body is mostly conformal one can approximate the field using the Magpylib solutions and some empirical magnetization that depends on the shape of the body, the material response and the strength of the magnetizing field.

An example would be the magnetization of a soft-magnetic metal piece in the earth magnetic field. However, even in such a case it is probably more efficient to use a simple dipole approximation.

Convergence of the diametral Cylinder solution

The diametral Cylinder solution is based on a convering series. 50 iterations are probaby ok and also set as standard. If you want to be precise increase iterations and observer the convergence behavior. Change the setting to `x` with

```
magpylib.Config.ITER_CYLINDER = x
```

References

[2020Malago] P. Malagò et al., Magnetic Position System Design Method Applied to Three-Axis Joystick Motion Tracking. Sensors, 2020, 20. Jg., Nr. 23, S. 6873.

2.6.3 Computation

Magpylib code is fully vectorized, written almost completly in numpy native. Magpylib automatically vectorizes the computation with complex inputs (many sources, many observers, paths) and never falls back on using loops.

Note: Maximal performance is achieved when `.getB(sources, observers)` is called only a single time in your program. Try not to use loops.

Of course the objective oriented interface (sensors and sources) comes with an overhead. If you want to achieve maximal performance this overhead can be avoided through direct access to the vectorized field functions with the top level function `magpylib.getBv`.

2.7 magpylib package

2.7.1 Welcome to Magpylib !

Magpylib provides static 3D magnetic field computation for permanent magnets, currents and other sources using (semi-) analytical formulas from the literature.

2.7.2 Resources

Documentation on Read-the-docs:

https://magpylib.readthedocs.io/en/latest/

Github repository:

https://github.com/magpylib/magpylib

Original software publication (version 2):

https://www.sciencedirect.com/science/article/pii/S2352711020300170

2.7.3 Introduction

Magpylib uses units of

- [mT]: for the B-field and the magnetization (mu0*M).
- [kA/m]: for the H-field.
- [mm]: for all position inputs.
- [deg]: for angle inputs by default.
- [A]: for current inputs.

Magpylib objects represent magnetic field sources and sensors with various attributes

```
>>> import magpylib as mag3
>>>
>>> # magnets
>>> src1 = mag3.magnet.Box(magnetization=(0,0,1000), dimension=(1,2,3))
>>> src2 = mag3.magnet.Cylinder(magnetization=(0,1000,0), dimension=(1,2))
>>> src3 = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>>
>>> # currents
>>> src4 = mag3.current.Circular(current=15, diameter=3)
>>> src5 = mag3.current.Line(current=15, vertices=[(0,0,0), (1,2,3)])
>>>
>>> # misc
>>> src6 = mag3.misc.Dipole(moment=(100,200,300))
>>>
>>> # sensor
```

(continues on next page)

(continued from previous page)

```
>>> sens = mag3.Sensor()
>>>
>>> for obj in [src1, src2, src3, src4, src5, src6, sens]:
>>>     print(obj)
Box(id=1792490441024)
Cylinder(id=1792490439680)
Sphere(id=1792491053792)
Circular(id=1792491053456)
Line(id=1792492457312)
Dipole(id=1792492479728)
Sensor(id=1792492480784)
```

All Magpylib objects are endowed with `position` and `orientation` attributes that describe their state in a global coordinate system. By default they are set to zero and unit-rotation respectively.

```
>>> import magpylib as mag3
>>> sens = mag3.Sensor()
>>> print(sens.position)
>>> print(sens.orientation.as_quat())
[0. 0. 0.]
[0. 0. 0. 1.]
```

Manipulate position and orientation attributes directly through source attributes, or by using built-in `move` and `rotate` methods.

```
>>> import magpylib as mag3
>>> sens = mag3.Sensor(position=(1,1,1))
>>> print(sens.position)
>>> sens.move((1,1,1))
>>> print(sens.position)
[1. 1. 1.]
[2. 2. 2.]
```

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sens = mag3.Sensor(orientation=R.from_rotvec((.1,.1,.1)))
>>> print(sens.orientation.as_rotvec())
>>> sens.rotate(R.from_rotvec((.1,.1,.1)))
>>> print(sens.orientation.as_rotvec())
>>> sens.rotate_from_angax(angle=.1, axis=(1,1,1), degree=False)
>>> print(sens.orientation.as_rotvec())
[0.1 0.1 0.1]
[0.2 0.2 0.2]
[0.25773503 0.25773503 0.25773503]
```

Source position and rotation attributes can also represent complete source paths in the global coordinate system. Such paths can be generated conveniently using the `move` and `rotate` methods.

```
>>> import magpylib as mag3
>>> src = mag3.magnet.Box(magnetization=(1,2,3), dimension=(1,2,3))
>>> src.move([(1,1,1),(2,2,2),(3,3,3),(4,4,4)], start='append')
>>> print(src.position)
[[0. 0. 0.]  [1. 1. 1.]  [2. 2. 2.]  [3. 3. 3.]  [4. 4. 4.]]
```

2.7.4 Grouping objects

Use the Collection class to group objects for common manipulation. All object methods can also be applied to complete Collections.

```
>>> import magpylib as mag3
>>> src1 = mag3.magnet.Box(magnetization=(0,0,1000), dimension=(1,2,3))
>>> src2 = mag3.magnet.Cylinder(magnetization=(0,1000,0), dimension=(1,2))
>>> col = src1 + src2
>>> col.move((1,2,3))
>>> for src in col:
>>>     print(src.pos)
[1. 2. 3.]
[1. 2. 3.]
```

2.7.5 Field computation

The magnetic field generated by `sources` at `observers` can be computed through the top level functions `getB` and `getH`. Sources are magpylib source objects like Circular or Dipole. Observers are magpylib Sensor objects or simply sets (list, tuple, ndarray) of positions. The result will be an array of all possible source-observer-path combinations

```
>>> import magpylib as mag3
>>> src1 = mag3.current.Circular(current=15, diameter=2)
>>> src2 = mag3.misc.Dipole(moment=(100,100,100))
>>> sens = mag3.Sensor(position=(1,1,1))
>>> obs_pos = (1,2,3)
>>> B = mag3.getB(sources=[src1,src2], observers=[sens,obs_pos])
>>> print(B)
[[[0.93539608 0.93539608 0.40046672]
  [0.05387784 0.10775569 0.0872515 ]]
 [[3.06293831 3.06293831 3.06293831]
  [0.04340403 0.23872216 0.43404028]]]
```

Field computation is also directly accessible in the form of object methods:

```
>>> import magpylib as mag3
>>> src = mag3.misc.Dipole(moment=(100,100,100))
>>> sens = mag3.Sensor(position=(1,1,1))
>>> pos_obs = (1,2,3)
>>> print(src.getB(sens, pos_obs))
[[3.06293831 3.06293831 3.06293831]
 [0.04340403 0.23872216 0.43404028]]
```

Finally there is a direct (very fast) interface to the field computation formulas that avoids the object oriented Magpylib interface:

```
>>> import magpylib as mag3
>>> B = mag3.getBv(
>>>     source_type='Dipole',
>>>     moment=(100,100,100),
>>>     observer=[(1,1,1), (1,2,3)])
>>> print(B)
[[3.06293831 3.06293831 3.06293831]
 [0.04340403 0.23872216 0.43404028]]
```

2.7.6 Graphic output

Display sources, collections, paths and sensors using Matplotlib from top level functions,

```
>>> import magpylib as mag3
>>> src1 = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> src2 = mag3.current.Circular(current=15, diameter=3)
>>> mag3.display(src1, src2)
--> graphic output
```

or directly through object methods

```
>>> import magpylib as mag3
>>> src = mag3.current.Circular(current=15, diameter=3)
>>> src.display()
--> graphic output
```

`magpylib.getB`(*sources*, *observers*, *sumup=False*, *squeeze=True*, ***specs*)

Compute B-field in [mT] for given sources and observers.

Parameters

- **sources** (*source object, Collection or 1D list thereof*) – Sources can be a single source object, a Collection or a 1D list of L source objects and/or collections.
- **observers** (*array_like or Sensor or 1D list thereof*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated, can be a Sensor object with pixel shape (N1, N2, . . . , 3) or a 1D list of K Sensor objects with similar pixel shape. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single sensor or only a single source) are eliminated.

Returns **B-field** – B-field of each source (L) at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [mT]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(L, M, K, N1, N2, .., 3)

Notes

This function automatically joins all sensor and position inputs together and groups similar sources for optimal vectorization of the computation. For maximal performance call this function as little as possible and avoid using it in loops.

Examples

Compute the B-field of a spherical magnet at a sensor positioned at (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> B = mag3.getB(source, sensor)
>>> print(B)
[-0.62497314  0.34089444  0.51134166]
```

Compute the B-field of a spherical magnet at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> B = mag3.getB(source, (1,2,3))
>>> print(B)
[[-0.88894262  0.          0.        ]
 [-0.62497314 -0.34089444 -0.51134166]
 [-0.17483825 -0.41961181 -0.62941771]
 [ 0.09177028 -0.33037301 -0.49555952]
 [ 0.17480239 -0.22080302 -0.33120453]]
```

Compute the B-field of two sources at two observer positions, with and without sumup:

```
>>> import magpylib as mag3
>>> src1 = mag3.current.Circular(current=15, diameter=2)
>>> src2 = mag3.misc.Dipole(moment=(100,100,100))
>>> obs_pos = [(1,1,1), (1,2,3)]
>>> B = mag3.getB([src1,src2], obs_pos)
>>> print(B)
[[[0.93539608 0.93539608 0.40046672]
  [0.05387784 0.10775569 0.0872515 ]]
 [[3.06293831 3.06293831 3.06293831]
  [0.04340403 0.23872216 0.43404028]]]
>>> B = mag3.getB([src1,src2], obs_pos, sumup=True)
>>> print(B)
[[3.99833439 3.99833439 3.46340502]
 [0.09728187 0.34647784 0.52129178]]
```

`magpylib.getH`(*sources*, *observers*, *sumup=False*, *squeeze=True*, ***specs*)

Compute H-field in [kA/m] for given sources and observers.

Parameters

- **sources** (*source object, Collection or 1D list thereof*) – Sources can be a single source object, a Collection or a 1D list of L source objects and/or collections.
- **observers** (*array_like or Sensor or 1D list thereof*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated, can be a Sensor object with pixel shape (N1, N2, . . . , 3) or a 1D list of K Sensor objects with similar pixel shape. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single sensor or only a single source) are eliminated.

Returns **H-field** – H-field of each source (L) at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [kA/m]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(L, M, K, N1, N2, .., 3)

Notes

This function automatically joins all sensor and position inputs together and groups similar sources for optimal vectorization of the computation. For maximal performance call this function as little as possible and avoid

using it in loops.

Examples

Compute the H-field of a spherical magnet at a sensor positioned at (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> H = mag3.getH(source, sensor)
>>> print(H)
[-0.49733782  0.27127518  0.40691277]
```

Compute the H-field of a spherical magnet at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> H = mag3.getH(source, (1,2,3))
>>> print(H)
[[-0.70739806  0.          0.        ]
 [-0.49733782 -0.27127518 -0.40691277]
 [-0.13913186 -0.33391647 -0.5008747 ]
 [ 0.07302847 -0.26290249 -0.39435373]
 [ 0.13910332 -0.17570946 -0.26356419]]
```

Compute the H-field of two sources at two observer positions, with and without sumup:

```
>>> import magpylib as mag3
>>> src1 = mag3.current.Circular(current=15, diameter=2)
>>> src2 = mag3.misc.Dipole(moment=(100,100,100))
>>> obs_pos = [(1,1,1), (1,2,3)]
>>> H = mag3.getH([src1,src2], obs_pos)
>>> print(H)
[[[0.74436455 0.74436455 0.31868129]
  [0.04287463 0.08574925 0.06943254]]
 [[2.43740886 2.43740886 2.43740886]
  [0.03453983 0.18996906 0.34539828]]]
>>> H = mag3.getH([src1,src2], obs_pos, sumup=True)
>>> print(H)
[[3.18177341 3.18177341 2.75609015]
 [0.07741445 0.27571831 0.41483082]]
```

magpylib.**getBv**(***kwargs*)

B-Field computation in units of [mT] from a dictionary of input vectors of length N.

This function avoids the object-oriented Magpylib interface and gives direct access to the field implementations. It is the fastet way to compute fields with Magpylib.

"Static" inputs of shape (x,) will automatically be tiled to shape (N,x) to fit with other inputs.

Required inputs depend on chosen source_type!

Parameters

- **source_type** (*string*) – Source type for computation. Must be either 'Box', 'Cylinder', 'Sphere', 'Dipole', 'Circular' or 'Line'. Expected input parameters depend on source_type.

- **position** (*array_like, shape (3,) or (N,3), default=(0,0,0)*) – Source positions in units of [mm].
- **orientation** (*scipy Rotation object, default=unit rotation*) – Source rotations relative to the initial state (see object docstrings).
- **observer** (*array_like, shape (3,) or (N,3)*) – Observer positions in units of [mm].
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output are eliminated.
- **magnetization** (*array_like, shape (3,) or (N,3)*) – Only *source_type in ('Box', 'Cylinder', 'Sphere')*! Magnetization vector (mu0*M) or remanence field of homogeneous magnet magnetization in units of [mT].
- **moment** (*array_like, shape (3,) or (N,3)*) – Only *source_type = 'Moment'*! Magnetic dipole moment in units of [mT*mm^3]. For homogeneous magnets the relation is moment = magnetization*volume.
- **current** (*array_like, shape (N,)*) – Only *source_type in ('Line', 'Circular')*! Current flowing in loop in units of [A].
- **dimension** (*array_like*) – Only *source_type in ('Box', 'Cylinder')*! Magnet dimension input in units of [mm].
- **diameter** (*array_like, shape (N)*) – Only *source_type in (Sphere, Circular)*! Diameter of source in units of [mm].
- **segment_start** (*array_like, shape (N,3)*) – Only *source_type = 'Line'*! Start positions of line current segments in units of [mm].
- **segment_end** (*array_like, shape (N,3)*) – Only *source_type = 'Line'*! End positions of line current segments in units of [mm].

Returns **B-field** – B-field generated by sources at observer positions in units of [mT].

Return type ndarray, shape (N,3)

Examples

Three-fold evaluation of the dipole field. For each computation the moment is (100,100,100).

```
>>> import magpylib as mag3
>>> B = mag3.getBv(
>>>     source_type='Dipole',
>>>     position=[(1,2,3), (2,3,4), (3,4,5)],
>>>     moment=(100,100,100),
>>>     observer=[(1,1,1), (2,2,2), (3,3,3)])
>>> print(B)
[[-0.71176254  0.56941003  1.85058261]
 [-0.71176254  0.56941003  1.85058261]
 [-0.71176254  0.56941003  1.85058261]]
```

Six-fold evaluation of a Cuboid magnet field with increasing size and magnetization of the magnet. Position and orientation are by default (0,0,0) and unit-orientation, respectively. The observer position is (1,2,3) for each evaluation.

```
>>> import numpy as np
>>> import magpylib as mag3
>>> B = mag3.getBv(
>>>     source_type='Box',
>>>     magnetization = [(0,0,m) for m in np.linspace(500,1000,6)],
>>>     dimension = [(a,a,a) for a in np.linspace(1,2,6)],
>>>     observer=(1,2,3))
>>> print(B)
[[ 0.48818967  0.97689261  0.70605984]
 [ 1.01203491  2.02636222  1.46575704]
 [ 1.87397714  3.756164    2.72063422]
 [ 3.19414311  6.41330652  4.65485356]
 [ 5.10909461 10.2855981   7.4881383 ]
 [ 7.76954697 15.70382556 11.48192812]]
```

magpylib.**getHv**(***kwargs*)

H-Field computation in units of [kA/m] from a dictionary of input vectors of length N.

This function avoids the object-oriented Magpylib interface and gives direct access to the field implementations. It is the fastet way to compute fields with Magpylib.

"Static" inputs of shape (x,) will automatically be tiled to shape (N,x) to fit with other inputs.

Required inputs depend on chosen source_type!

Parameters

- **source_type** (*string*) – Source type for computation. Must be either 'Box', 'Cylinder', 'Sphere', 'Dipole', 'Circular' or 'Line'. Expected input parameters depend on source_type.
- **position** (*array_like, shape (3,) or (N,3), default=(0,0,0)*) – Source positions in units of [mm].
- **orientation** (*scipy Rotation object, default=unit rotation*) – Source rotations relative to the initial state (see object docstrings).
- **observer** (*array_like, shape (3,) or (N,3)*) – Observer positions in units of [mm].
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output are eliminated.
- **magnetization** (*array_like, shape (3,) or (N,3)*) – Only *source_type in ('Box', 'Cylinder', 'Sphere')*! Magnetization vector (mu0*M) or remanence field of homogeneous magnet magnetization in units of [mT].
- **moment** (*array_like, shape (3,) or (N,3)*) – Only *source_type = 'Moment'*! Magnetic dipole moment in units of [mT*mm^3]. For homogeneous magnets the relation is moment = magnetization*volume.
- **current** (*array_like, shape (N,)*) – Only *source_type in ('Line', 'Circular')*! Current flowing in loop in units of [A].
- **dimension** (*array_like*) – Only *source_type in ('Box', 'Cylinder')*! Magnet dimension input in units of [mm].
- **diameter** (*array_like, shape (N)*) – Only *source_type in (Sphere, Circular)*! Diameter of source in units of [mm].
- **segment_start** (*array_like, shape (N,3)*) – Only *source_type = 'Line'*! Start positions of line current segments in units of [mm].

- **segment_end** (*array_like, shape (N,3)*) – Only *source_type* = *'Line'*! End positions of line current segments in units of [mm].

Returns **H-field** – H-field generated by sources at observer positions in units of [kA/m].

Return type ndarray, shape (N,3)

Examples

Three-fold evaluation of the dipole field. For each computation the moment is (100,100,100).

```
>>> import magpylib as mag3
>>> H = mag3.getHv(
>>>     source_type='Dipole',
>>>     position=[(1,2,3), (2,3,4), (3,4,5)],
>>>     moment=(100,100,100),
>>>     observer=[(1,1,1), (2,2,2), (3,3,3)])
>>> print(H)
[[-0.56640264  0.45312211  1.47264685]
 [-0.56640264  0.45312211  1.47264685]
 [-0.56640264  0.45312211  1.47264685]]
```

Six-fold evaluation of a Cuboid magnet field with increasing size and magnetization of the magnet. Position and orientation are (0,0,0) and unit-orientation, respectively, by default. The observer position is (1,2,3) for each evaluation.

```
>>> import numpy as np
>>> import magpylib as mag3
>>> H = mag3.getHv(
>>>     source_type='Box',
>>>     magnetization = [(0,0,m) for m in np.linspace(500,1000,6)],
>>>     dimension = [(a,a,a) for a in np.linspace(1,2,6)],
>>>     observer=(1,2,3))
>>> print(H)
[[ 0.388489    0.77738644  0.56186457]
 [ 0.80535179  1.61252782  1.16641239]
 [ 1.49126363  2.98906034  2.16501192]
 [ 2.54181833  5.10354717  3.70421476]
 [ 4.06568831  8.1850189   5.95887113]
 [ 6.18280903 12.49670731  9.13702808]]
```

class magpylib.**Sensor**(*position=(0, 0, 0)*, *pixel=(0, 0, 0)*, *orientation=None*)

Bases: magpylib._lib.obj_classes.class_BaseGeo.BaseGeo, magpylib._lib.obj_classes.class_BaseDisplayRepr.BaseDisplayRepr

Magnetic field sensor. Can be used as observer input for magnetic field computation.

Local object coordinates: Sensor pixel (=sensing elements) are defined in the local object coordinate system. Local (Sensor) and global CS coincide when position=(0,0,0) and orientation=unit_rotation.

Parameters

- **position** (*array_like, shape (3,) or (M,3), default=(0,0,0)*) – Object position (local CS origin) in the global CS in units of [mm]. For M>1, the position represents a path. The position and orientation parameters must always be of the same length.
- **pixel** (*array_like, shape (3,) or (N1,N2,..,3), default=(0,0,0)*) – Sensor pixel positions (=sensing elements) in the local Sensor CS in units of [mm].

The magnetic field is evaluated at Sensor pixels.

- **orientation** (*scipy Rotation object with length 1 or M, default=unit rotation*) – Object orientation (local CS orientation) in the global CS. For M>1 orientation represents different values along a path. The position and orientation parameters must always be of the same length.

Returns **Sensor object**

Return type *Sensor*

Examples

By default a Sensor is initialized at position (0,0,0), with unit rotation and pixel (0,0,0):

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.pixel)
[0. 0. 0.]
>>> print(sensor.orientation.as_quat())
[0. 0. 0. 1.]
```

Sensors are observers for magnetic field computation. In this example we compute the H-field as seen by the sensor in the center of a circular current loop:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> loop = mag3.current.Circular(current=1, diameter=1)
>>> H = sensor.getH(loop)
>>> print(H)
[0. 0. 1.]
```

Field computation is performed at every pixel of a sensor. The above example is reproduced for a 2x2-pixel sensor:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor(pixel=[[(0,0,0), (0,0,1)],[(0,0,2), (0,0,3)]])
>>> loop = mag3.current.Circular(current=1, diameter=1)
>>> H = sensor.getH(loop)
>>> print(H.shape)
(2, 2, 3)
>>> print(H)
[[[0.         0.         1.        ]
  [0.         0.         0.08944272]]
 [[0.         0.         0.0142668 ]
  [0.         0.         0.00444322]]]
```

Compute the field of a sensor along a path. The path positions are chosen so that they coincide with the pixel positions of the previous example:

```
>>> import magpylib as mag3
>>> loop = mag3.current.Circular(current=1, diameter=1)
>>> sensor = mag3.Sensor()
>>> sensor.move([(0,0,1)]*3, start=1, increment=True)
>>> print(sensor.position)
```

(continues on next page)

(continued from previous page)

```
[[0. 0. 0.]
 [0. 0. 1.]
 [0. 0. 2.]
 [0. 0. 3.]]
>>> H = sensor.getH(loop)
>>> print(H)
[[0.         0.         1.        ]
 [0.         0.         0.08944272]
 [0.         0.         0.0142668 ]
 [0.         0.         0.00444322]]
```

display (*markers=[(0, 0, 0)], axis=None, show_direction=False, show_path=True, size_sensors=1, size_direction=1, size_dipoles=1*)

Display object graphically using matplotlib 3D plotting.

Parameters

- **markers** (*array_like, shape (N,3), default=[(0,0,0)]*) – Display position markers in the global CS. By default a marker is placed in the origin.
- **axis** (*pyplot.axis, default=None*) – Display graphical output in a given pyplot axis (must be 3D). By default a new pyplot figure is created and displayed.
- **show_direction** (*bool, default=False*) – Set True to show magnetization and current directions.
- **show_path** (*bool or int, default=True*) – Options True, False, positive int. By default object paths are shown. If show_path is a positive integer, objects will be displayed at multiple path positions along the path, in steps of show_path.
- **size_sensor** (*float, default=1*) – Adjust automatic display size of sensors.
- **size_direction** (*float, default=1*) – Adjust automatic display size of direction arrows.
- **size_dipoles** (*float, default=1*) – Adjust automatic display size of dipoles.

Returns None

Return type NoneType

Examples

Display Magpylib objects graphically using Matplotlib:

```
>>> import magpylib as mag3
>>> obj = mag3.magnet.Sphere(magnetization=(0,0,1), diameter=1)
>>> obj.move([(.2,0,0)]*50, increment=True)
>>> obj.rotate_from_angax(angle=[10]*50, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> obj.display(show_direction=True, show_path=10)
--> graphic output
```

Display figure on your own 3D Matplotlib axis:

```
>>> import matplotlib.pyplot as plt
>>> import magpylib as mag3
>>> my_axis = plt.axes(projection='3d')
```

(continues on next page)

(continued from previous page)

```
>>> obj = mag3.magnet.Box(magnetization=(0,0,1), dimension=(1,2,3))
>>> obj.move([(x,0,0) for x in [0,1,2,3,4,5]])
>>> obj.display(axis=my_axis)
>>> plt.show()
--> graphic output
```

getB (**sources*, *sumup=False*, *squeeze=True*)
Compute B-field in [mT] for given sources as seen by the Sensor.

Parameters

- **sources** (*source objects or Collections*) – Sources can be a mixture of L source objects or Collections.
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single sensor or only a single source) are eliminated.

Returns B-field – B-field of each source (L) at each path position (M) and each sensor pixel position (N1,N2,…) in units of [mT]. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(L, M, N1, N2, .., 3)

Examples

Sensors are observers for magnetic field computation. In this example we compute the B-field [mT] as seen by the sensor in the center of a circular current loop:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> loop = mag3.current.Circular(current=1, diameter=1)
>>> B = sensor.getB(loop)
>>> print(B)
[0.         0.         1.25663706]
```

Field computation is performed at every pixel of a sensor. The above example is reproduced for a 2x2-pixel sensor:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor(pixel=[[(0,0,0), (0,0,1)],[(0,0,2), (0,0,3)]])
>>> loop = mag3.current.Circular(current=1, diameter=1)
>>> B = sensor.getB(loop)
>>> print(B.shape)
(2, 2, 3)
>>> print(B)
[[[0.         0.         1.25663706]
  [0.         0.         0.11239704]]
 [[0.         0.         0.01792819]
  [0.         0.         0.00558351]]]
```

Compute the field of a sensor along a path. The path positions are chosen so that they coincide with the pixel positions in the previous example.

```
>>> import magpylib as mag3
>>> loop = mag3.current.Circular(current=1, diameter=1)
```

(continues on next page)

(continued from previous page)

```
>>> sensor = mag3.Sensor()
>>> sensor.move([(0,0,1)]*3, start=1, increment=True)
>>> print(sensor.position)
[[0. 0. 0.]
 [0. 0. 1.]
 [0. 0. 2.]
 [0. 0. 3.]]
>>> B = sensor.getB(loop)
>>> print(B)
[[0.         0.         1.25663706]
 [0.         0.         0.11239704]
 [0.         0.         0.01792819]
 [0.         0.         0.00558351]]
```

getH(**sources*, *sumup=False*, *squeeze=True*)

Compute H-field in [kA/m] for given sources as seen by the Sensor.

Parameters

- **sources** (*source objects or Collections*) – Sources can be a mixture of L source objects or Collections.
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single sensor or only a single source) are eliminated.

Returns H-field – H-field of each source (L) at each path position (M) and each sensor pixel position (N1,N2,. . .) in units of [kA/m]. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(L, M, N1, N2, .., 3)

Examples

Sensors are observers for magnetic field computation. In this example we compute the H-field as seen by the sensor in the center of a circular current loop:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> loop = mag3.current.Circular(current=1, diameter=1)
>>> H = sensor.getH(loop)
>>> print(H)
[0. 0. 1.]
```

Field computation is performed at every pixel of a sensor. The above example is reproduced for a 2x2-pixel sensor:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor(pixel=[[(0,0,0), (0,0,1)],[(0,0,2), (0,0,3)]])
>>> loop = mag3.current.Circular(current=1, diameter=1)
>>> H = sensor.getH(loop)
>>> print(H.shape)
(2, 2, 3)
>>> print(H)
[[[0.         0.         1.        ]
  [0.         0.         0.08944272]]
```

(continues on next page)

(continued from previous page)

```
 [[0.         0.         0.0142668 ]
  [0.         0.         0.00444322]]]
```

Compute the field of a sensor along a path. The path positions are chosen so that they coincide with the pixel positions in the previous example.

```
>>> import magpylib as mag3
>>> loop = mag3.current.Circular(current=1, diameter=1)
>>> sensor = mag3.Sensor()
>>> sensor.move([(0,0,1)]*3, start=1, increment=True)
>>> print(sensor.position)
[[0. 0. 0.]
 [0. 0. 1.]
 [0. 0. 2.]
 [0. 0. 3.]]
>>> H = sensor.getH(loop)
>>> print(H)
[[0.         0.         1.        ]
 [0.         0.         0.08944272]
 [0.         0.         0.0142668 ]
 [0.         0.         0.00444322]]
```

move (*displacement*, *start=-1*, *increment=False*)

Translates the object by the input displacement (can be a path).

This method uses vector addition to merge the input path given by displacement and the existing old path of an object. It keeps the old orientation. If the input path extends beyond the old path, the old path will be padded by its last entry before paths are added up.

Parameters

- **displacement** (*array_like, shape (3,) or (N,3)*) – Displacement vector shape=(3,) or path shape=(N,3) in units of [mm].
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input displacements are absolute. If *increment=True*, input displacements are interpreted as increments of each other. For example, an incremental input displacement of *[(2,0,0), (2,0,0), (2,0,0)]* corresponds to an absolute input displacement of *[(2,0,0), (4,0,0), (6,0,0)]*.

Returns self

Return type Magpylib object

Examples

With the `move` method Magpylib objects can be repositioned in the global coordinate system:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> sensor.move((1,1,1))
```

(continues on next page)

(continued from previous page)

```
>>> print(sensor.position)
[1. 1. 1.]
```

It is also a powerful tool for creating paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move((1,1,1), start='append')
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]]
>>> sensor.move([(.1,.1,.1)]*2, start='append')
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [1.1 1.1 1.1]
 [1.1 1.1 1.1]]
```

Complex paths can be generated with ease, by making use of the `increment` keyword and superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move([(1,1,1)]*4, start='append', increment=True)
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
>>> sensor.move([(.1,.1,.1)]*5, start=2)
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [2.1 2.1 2.1]
 [3.1 3.1 3.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]]
```

orientation
Object orientation attribute getter and setter.

pixel
Sensor pixel attribute getter and setter.

position
Object position attribute getter and setter.

reset_path()
Reset object path to position = (0,0,0) and orientation = unit rotation.

Returns self

Return type Magpylib object

Examples

Create an object with non-zero path

```
>>> import magpylib as mag3
>>> obj = mag3.Sensor(position=(1,2,3))
>>> print(obj.position)
[1. 2. 3.]
>>> obj.reset_path()
>>> print(obj.position)
[0. 0. 0.]
```

rotate(*rotation*, *anchor=None*, *start=-1*, *increment=False*)

Rotates the object in the global coordinate system by a given rotation input (can be a path).

This method applies given rotations to the original orientation. If the input path extends beyond the existing path, the old path will be padded by its last entry before paths are added up.

Parameters

- **rotation** (*scipy Rotation object*) – Rotation to be applied. The rotation object can feature a single rotation of shape (3,) or a set of rotations of shape (N,3) that correspond to a path.
- **anchor** (*None, 0 or array_like, shape (3,), default=None*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other.

Returns self

Return type Magpylib object

Examples

With the `rotate` method Magpylib objects can be rotated about their local coordinate system center:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> rotation_object = R.from_euler('x', 45, degrees=True)
>>> sensor.rotate(rotation_object)
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the anchor keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> rotation_object = R.from_euler('x', [10,20,30], degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the increment keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', [10]*3, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append',
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
```

(continues on next page)

(continued from previous page)

```
 [20.  0.  0.]
 [30.  0.  0.]]
>>> rotation_object = R.from_euler('z', [5]*4, degrees=True)
>>> sensor.rotate(rotation_object, anchor=0, start=0, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

rotate_from_angax (*angle*, *axis*, *anchor=None*, *start=-1*, *increment=False*, *degrees=True*)

Object rotation in the global coordinate system from angle-axis input.

This method applies given rotations to the original orientation. If the input path extends beyond the existingp path, the oldpath will be padded by its last entry before paths are added up.

Parameters

- **angle** (*int/float or array_like with shape (n,) unit [deg] (by default)*) – Angle of rotation, or a vector of n angles defining a rotation path in units of [deg] (by default).
- **axis** (*str or array_like, shape (3,)*) – The direction of the axis of rotation. Input can be a vector of shape (3,) or a string 'x', 'y' or 'z' to denote respective directions.
- **anchor** (*None or array_like, shape (3,), default=None, unit [mm]*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other. For example, the incremental angles [1,1,1,2,2] correspond to the absolute angles [1,2,3,5,7].
- **degrees** (*bool, default=True*) – By default angle is given in units of [deg]. If degrees=False, angle is given in units of [rad].

Returns self

Return type Magpylib object

Examples

With the `rotate_from_angax` method Magpylib objects can be rotated about their local coordinte system center:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> sensor.rotate_from_angax(angle=45, axis='x')
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis=(1,0,0), anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis='x', anchor=(0,1,0), start='append
↪')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[10,20,30], axis='x', anchor=(0,1,0),␣
↪start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax([10]*3, 'x', (0,1,0), start=1, increment=True)
>>> print(sensor.position)
```

(continues on next page)

(continued from previous page)

```
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[5]*4, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

class magpylib.**Collection**(**sources*)

Bases: magpylib._lib.obj_classes.class_BaseDisplayRepr.BaseDisplayRepr, magpylib._lib.obj_classes.class_BaseGetBH.BaseGetBH

Group multiple sources in one Collection for common manipulation.

Operations applied to a Collection are sequentially applied to all sources in the Collection. Collections do not allow duplicate sources (will be eliminated automatically).

Collections have the following dunders defined: __add__, __sub__, __iter__, __getitem__, __repr__

Parameters **sources** (*source objects, Collections or arbitrary lists thereof*) – Ordered list of sources in the Collection.

Returns Collection object

Return type *Collection*

Examples

Create Collections for common manipulation. All sources added to a Collection are stored in the `sources` attribute, which is an ordered set (list with unique elements only)

```
>>> import magpylib as mag3
>>> sphere = mag3.magnet.Sphere((1,2,3),1)
>>> loop = mag3.current.Circular(1,1)
>>> dipole = mag3.misc.Dipole((1,2,3))
>>> col = mag3.Collection(sphere, loop, dipole)
>>> print(col.sources)
[Sphere(id=1879891544384), Circular(id=1879891543040), Dipole(id=1879892157152)]
```

Cycle directly through the Collection `sources` attribute

```
>>> for src in col:
>>>     print(src)
```

(continues on next page)

(continued from previous page)

```
Sphere(id=1879891544384)
Circular(id=1879891543040)
Dipole(id=1879892157152)
```

and directly access objects from the Collection

```
>>> print(col[1])
Circular(id=1879891543040)
```

Add and subtract sources to form a Collection and to remove sources from a Collection.

```
>>> col = sphere + loop
>>> print(col.sources)
[Sphere(id=1879891544384), Circular(id=1879891543040)]
>>> col - sphere
>>> print(col.sources)
[Circular(id=1879891543040)]
```

Manipulate all objects in a Collection directly using `move` and `rotate` methods

```
>>> import magpylib as mag3
>>> sphere = mag3.magnet.Sphere((1,2,3),1)
>>> loop = mag3.current.Circular(1,1)
>>> col = sphere + loop
>>> col.move((1,1,1))
>>> print(sphere.position)
[1. 1. 1.]
```

and compute the total magnetic field generated by the Collection.

```
>>> B = col.getB((1,2,3))
>>> print(B)
[-0.00372678  0.01820438  0.03423079]
```

add(**sources*)

Add arbitrary sources or Collections.

Parameters sources (*src objects, Collections or arbitrary lists thereof*) – Add arbitrary sequences of sources and Collections to the Collection. The new sources will be added at the end of self.sources. Duplicates will be eliminated.

Returns self

Return type *Collection*

Examples

Add sources to a Collection:

```
>>> import magpylib as mag3
>>> src = mag3.current.Circular(1,1)
>>> col = mag3.Collection()
>>> col.add(src)
>>> print(col.sources)
[Circular(id=2519738714432)]
```

copy()

Returns a copy of the Collection.

Returns self

Return type *Collection*

Examples

Create a copy of a Collection object:

```
>>> import magpylib as mag3
>>> col = mag3.Collection()
>>> print(id(col))
2221754911040
>>> col2 = col.copy()
>>> print(id(col2))
2221760504160
```

display(*markers=[(0, 0, 0)], axis=None, show_direction=False, show_path=True, size_sensors=1, size_direction=1, size_dipoles=1*)

Display object graphically using matplotlib 3D plotting.

Parameters

- **markers** (*array_like, shape (N,3), default=[(0,0,0)]*) – Display position markers in the global CS. By default a marker is placed in the origin.
- **axis** (*pyplot.axis, default=None*) – Display graphical output in a given pyplot axis (must be 3D). By default a new pyplot figure is created and displayed.
- **show_direction** (*bool, default=False*) – Set True to show magnetization and current directions.
- **show_path** (*bool or int, default=True*) – Options True, False, positive int. By default object paths are shown. If show_path is a positive integer, objects will be displayed at multiple path positions along the path, in steps of show_path.
- **size_sensor** (*float, default=1*) – Adjust automatic display size of sensors.
- **size_direction** (*float, default=1*) – Adjust automatic display size of direction arrows.
- **size_dipoles** (*float, default=1*) – Adjust automatic display size of dipoles.

Returns None

Return type NoneType

Examples

Display Magpylib objects graphically using Matplotlib:

```
>>> import magpylib as mag3
>>> obj = mag3.magnet.Sphere(magnetization=(0,0,1), diameter=1)
>>> obj.move([(.2,0,0)]*50, increment=True)
>>> obj.rotate_from_angax(angle=[10]*50, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> obj.display(show_direction=True, show_path=10)
--> graphic output
```

Display figure on your own 3D Matplotlib axis:

```
>>> import matplotlib.pyplot as plt
>>> import magpylib as mag3
>>> my_axis = plt.axes(projection='3d')
>>> obj = mag3.magnet.Box(magnetization=(0,0,1), dimension=(1,2,3))
>>> obj.move([(x,0,0) for x in [0,1,2,3,4,5]])
>>> obj.display(axis=my_axis)
>>> plt.show()
--> graphic output
```

getB(**observers*, *squeeze=True*)

Compute B-field in units of [mT] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **B-field** – B-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [mT]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the B-field [mT] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> B = source.getB(sensor)
>>> print(B)
[-0.62497314  0.34089444  0.51134166]
```

Compute the B-field [mT] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> B = source.getB((1,2,3))
>>> print(B)
[[-0.88894262  0.          0.        ]
[-0.62497314 -0.34089444 -0.51134166]
[-0.17483825 -0.41961181 -0.62941771]
[ 0.09177028 -0.33037301 -0.49555952]
[ 0.17480239 -0.22080302 -0.33120453]]
```

Compute the B-field [mT] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> B = source.getB(sens1, sens2)
>>> print(B)
[[0.01421427 0.02842853 0.02114728]
[0.00621368 0.00932052 0.00501254]]
```

getH(**observers*, *squeeze=True*)

Compute H-field in units of [kA/m] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, ..., 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, ..., 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **H-field** – H-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,...) in units of [kA/m]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the H-field [kA/m] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> H = source.getH(sensor)
>>> print(H)
[-0.49733782  0.27127518  0.40691277]
```

Compute the H-field [kA/m] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> H = source.getH((1,2,3))
>>> print(H)
[[-0.70739806  0.          0.        ]
[-0.49733782 -0.27127518 -0.40691277]
[-0.13913186 -0.33391647 -0.5008747 ]
[ 0.07302847 -0.26290249 -0.39435373]
[ 0.13910332 -0.17570946 -0.26356419]]
```

Compute the H-field [kA/m] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> H = source.getH(sens1, sens2)
>>> print(H)
[[0.01131135 0.02262271 0.01682847]
[0.00494469 0.00741704 0.00398885]]
```

move (*displacement*, *start=-1*, *increment=False*)

Translates each object in the Collection individually by the input displacement (can be a path).

This method uses vector addition to merge the input path given by displacement and the existing old path of an object. It keeps the old orientation. If the input path extends beyond the old path, the old path will be padded by its last entry before paths are added up.

Parameters

- **displacement** (*array_like, shape (3,) or (N,3)*) – Displacement vector shape=(3,) or path shape=(N,3) in units of [mm].
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input displacements are absolute. If *increment=True*, input displacements are interpreted as increments of each other. For example, an incremental input displacement of *[(2,0,0), (2,0,0), (2,0,0)]* corresponds to an absolute input displacement of *[(2,0,0), (4,0,0), (6,0,0)]*.

Returns self

Return type *Collection*

Examples

This method will apply the `move` operation to each Collection object individually.

```
>>> import magpylib as mag3
>>> dipole = mag3.misc.Dipole((1,2,3))
>>> loop = mag3.current.Circular(1,1)
>>> col = loop + dipole
>>> col.move([(1,1,1), (2,2,2)])
>>> for src in col:
>>>     print(src.position)
[[1. 1. 1.]  [2. 2. 2.]]
[[1. 1. 1.]  [2. 2. 2.]]
```

But manipulating individual objects keeps them in the Collection

```
>>> dipole.move((1,1,1))
>>> for src in col:
>>>     print(src.position)
[[1. 1. 1.]  [2. 2. 2.]]
[[1. 1. 1.]  [3. 3. 3.]]
```

remove (*source*)

Remove a specific source from the Collection.

Parameters **source** (*source object*) – Remove the given source from the Collection.

Returns self

Return type *Collection*

Examples

Remove a specific source from a Collection:

```
>>> import magpylib as mag3
>>> src1 = mag3.current.Circular(1,1)
>>> src2 = mag3.current.Circular(1,1)
>>> col = src1 + src2
>>> print(col.sources)
[Circular(id=2405009623360), Circular(id=2405010235504)]
>>> col.remove(src1)
>>> print(col.sources)
[Circular(id=2405010235504)]
```

reset_path()

Reset all object paths to position = (0,0,0) and orientation = unit rotation.

Returns self

Return type *Collection*

Examples

Create a collection with non-zero paths

```
>>> import magpylib as mag3
>>> dipole = mag3.misc.Dipole((1,2,3), position=(1,2,3))
>>> loop = mag3.current.Circular(1,1, position=[(1,1,1)]*2)
>>> col = loop + dipole
>>> for src in col:
>>>     print(src.position)
[[1. 1. 1.]  [1. 1. 1.]]
[1. 2. 3.]
>>> col.reset_path()
>>> for src in col:
>>>     print(src.position)
[0. 0. 0.]
[0. 0. 0.]
```

rotate(*rot*, *anchor=None*, *start=-1*, *increment=False*)

Rotates each object in the Collection individually.

This method applies given rotations to the original orientation. If the input path extends beyond the existing path, the old path will be padded by its last entry before paths are added up.

Parameters

- **rotation** (*scipy Rotation object*) – Rotation to be applied. The rotation object can feature a single rotation of shape (3,) or a set of rotations of shape (N,3) that correspond to a path.

- **anchor** (*None, 0 or array_like, shape (3,), default=None*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other.

Returns self

Return type *Collection*

Examples

This method will apply the `rotate` operation to each Collection object individually.

```
>>> from scipy.spatial.transform import Rotation as R
>>> import magpylib as mag3
>>> dipole = mag3.misc.Dipole((1,2,3))
>>> loop = mag3.current.Circular(1,1)
>>> col = loop + dipole
>>> col.rotate(R.from_euler('x', [45,90], degrees=True))
>>> for src in col:
>>>     print(src.orientation.as_euler('xyz', degrees=True))
[[45.  0.  0.]  [90.  0.  0.]]
[[45.  0.  0.]  [90.  0.  0.]]
```

Manipulating individual objects keeps them in the Collection

```
>>> dipole.rotate(R.from_euler('x', [45], degrees=True))
>>> for src in col:
>>>     print(src.orientation.as_euler('xyz', degrees=True))
[[45.  0.  0.]  [ 90.  0.  0.]]
[[45.  0.  0.]  [135.  0.  0.]]
```

rotate_from_angax (*angle, axis, anchor=None, start=-1, increment=False, degrees=True*)

Rotates each object in the Collection individually from angle-axis input.

This method applies given rotations to the original orientation. If the input path extends beyond the existingp path, the oldpath will be padded by its last entry before paths are added up.

Parameters

- **angle** (*int/float or array_like with shape (n,) unit [deg] (by default)*) – Angle of rotation, or a vector of n angles defining a rotation path in units of [deg] (by default).
- **axis** (*str or array_like, shape (3,)*) – The direction of the axis of rotation. Input can be a vector of shape (3,) or a string 'x', 'y' or 'z' to denote respective directions.
- **anchor** (*None or array_like, shape (3,), default=None, unit [mm]*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).

- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other. For example, the incremental angles [1,1,1,2,2] correspond to the absolute angles [1,2,3,5,7].
- **degrees** (*bool, default=True*) – By default angle is given in units of [deg]. If degrees=False, angle is given in units of [rad].

Returns self

Return type *Collection*

Examples

This method will apply the `rotate_from_angax` operation to each Collection object individually.

```
>>> import magpylib as mag3
>>> dipole = mag3.misc.Dipole((1,2,3))
>>> loop = mag3.current.Circular(1,1)
>>> col = loop + dipole
>>> col.rotate_from_angax([45,90], 'x')
>>> for src in col:
>>>     print(src.orientation.as_euler('xyz', degrees=True))
[[45.  0.  0.]  [90.  0.  0.]]
[[45.  0.  0.]  [90.  0.  0.]]
```

Manipulating individual objects keeps them in the Collection

```
>>> dipole.rotate_from_angax(45, 'x')
>>> for src in col:
>>>     print(src.orientation.as_euler('xyz', degrees=True))
[[45.  0.  0.]  [ 90.  0.  0.]]
[[45.  0.  0.]  [135.  0.  0.]]
```

sources
Collection sources attribute getter and setter.

magpylib.**display**(**objects, markers=[(0, 0, 0)], axis=None, show_direction=False, show_path=True, size_sensors=1, size_direction=1, size_dipoles=1*)
Display objects and paths graphically using matplotlib 3D plotting.

Parameters

- **objects** (*sources, collections or sensors*) – Objects to be displayed.
- **markers** (*array_like, shape (N,3), default=[(0,0,0)]*) – Display position markers in the global CS. By default a marker is placed in the origin.
- **axis** (*pyplot.axis, default=None*) – Display graphical output in a given pyplot axis (must be 3D). By default a new pyplot figure is created and displayed.
- **show_direction** (*bool, default=False*) – Set True to show magnetization and current directions.
- **show_path** (*bool or int, default=True*) – Options True, False, positive int. By default object paths are shown. If show_path is a positive integer, objects will be displayed at multiple path positions along the path, in steps of show_path.

- **size_sensor** (*float, default=1*) – Adjust automatic display size of sensors.
- **size_direction** (*float, default=1*) – Adjust automatic display size of direction arrows.
- **size_dipoles** (*float, default=1*) – Adjust automatic display size of dipoles.

Returns None

Return type NoneType

Examples

Display multiple objects, object paths, markers in 3D using Matplotlib:

```
>>> import magpylib as mag3
>>> col = mag3.Collection(
    [mag3.magnet.Sphere(magnetization=(0,0,1), diameter=1) for _ in range(3)])
>>> for displ,src in zip([(.1414,0,0),(-.1,-.1,0),(-.1,.1,0)], col):
>>>     src.move([displ]*50, increment=True)
>>>     src.rotate_from_angax(angle=[10]*50, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> ts = [-.6,-.4,-.2,0,.2,.4,.6]
>>> sens = mag3.Sensor(position=(0,0,2), pixel=[(x,y,0) for x in ts for y in ts])
>>> mag3.display(col, sens)
--> graphic output
```

Display figure on your own 3D Matplotlib axis:

```
>>> import matplotlib.pyplot as plt
>>> import magpylib as mag3
>>> my_axis = plt.axes(projection='3d')
>>> magnet = mag3.magnet.Box(magnetization=(0,0,1), dimension=(1,2,3))
>>> sens = mag3.Sensor(position=(0,0,3))
>>> mag3.display(magnet, sens, axis=my_axis)
>>> plt.show()
--> graphic output
```

class magpylib.**Config**

Bases: object

Library default settings

Parameters

- **CHECK_INPUTS** (*bool, default=True*) – Check user input types, shapes at various stages and raise errors when they are not within designated parameters.
- **EDGESIZE** (*float, default=1e-14*) – getBand getH return 0 on edge, formulas often show singularities there and undefined forms. EDGESIZE defines how close to the edge 0 will be returned to avoid running into numerical instabilities.
- **ITER_CYLINDER** (*int, default=50*) – Cylinder with diametral magnetization uses Simpsons iterative formula to compute the integral. More iterations increase precision but slow down the computation.

Examples

Compute the field very close to a line current:

```
>>> import magpylib as mag3
>>> current = mag3.current.Line(current=1, vertices=[(-.1,0,0),(.1,0,0)])
>>> B_close = current.getB((0,0,1e-11))
>>> print(B_close)
[ 0.e+00 -2.e+10  0.e+00]
```

Change the edgesize setting so that the position is now inside of the cut-off region

```
>>> mag3.Config.EDGESIZE=1e-10
>>> B_close = current.getB((0,0,1e-11))
>>> print(B_close)
[0. 0. 0.]
```

Reset the Config to original values:

```
>>> mag3.Config.reset()
>>> B_close = current.getB((0,0,1e-11))
>>> print(B_close)
[ 0.e+00 -2.e+10  0.e+00]
```

classmethod reset()

Reset Config to default values.

Returns None

Return type NoneType

2.7.7 Subpackages

magpylib.current package

This subpackage contains all electric current classes. Currents are modeled as line-currents, input is the current in units of Ampere [A]. Field computation formulas are obtained via the law of Biot-Savardt.

class magpylib.current.**Circular**(*current=None*, *diameter=None*, *position=(0, 0, 0)*, *orientation=None*)

Bases: magpylib._lib.obj_classes.class_BaseGeo.BaseGeo, magpylib._lib.obj_classes.class_BaseDisplayRepr.BaseDisplayRepr, magpylib._lib.obj_classes.class_BaseGetBH.BaseGetBH, magpylib._lib.obj_classes.class_BaseExcitations.BaseCurrent

Circular current loop.

Local object coordinates: The Circular current loop lies in the x-y plane of the local object coordinate system, with its center in the origin. Local (Circular) and global CS coincide when position=(0,0,0) and orientation=unit_rotation.

Parameters

- **current** (*float*) – Electrical current in units of [A].
- **diameter** (*float*) – Diameter of the loop in units of [mm].
- **position** (*array_like, shape (3,) or (M,3), default=(0,0,0)*) – Object position (local CS origin) in the global CS in units of [mm]. For M>1, the position represents a path. The position and orientation parameters must always be of the same length.

- **orientation** (*scipy Rotation object with length 1 or M, default=unit rotation*) – Object orientation (local CS orientation) in the global CS. For M>1 orientation represents different values along a path. The position and orientation parameters must always be of the same length.

Returns **Circular object**

Return type *Circular*

Examples

By default a Circular is initialized at position (0,0,0), with unit rotation:

```
>>> import magpylib as mag3
>>> magnet = mag3.current.Circular(current=100, diameter=2)
>>> print(magnet.position)
[0. 0. 0.]
>>> print(magnet.orientation.as_quat())
[0. 0. 0. 1.]
```

Circulars are magnetic field sources. Below we compute the H-field [kA/m] of the above Circular at the observer position (1,1,1),

```
>>> H = magnet.getH((1,1,1))
>>> print(H)
[4.96243034 4.96243034 2.12454191]
```

or at a set of observer positions:

```
>>> H = magnet.getH([(1,1,1), (2,2,2), (3,3,3)])
>>> print(H)
[[4.96243034 4.96243034 2.12454191]
 [0.61894364 0.61894364 0.06167939]
 [0.18075829 0.18075829 0.00789697]]
```

The same result is obtained when the Circular moves along a path, away from the observer:

```
>>> magnet.move([(-1,-1,-1), (-2,-2,-2)], start=1)
>>> H = magnet.getH((1,1,1))
>>> print(H)
[[4.96243034 4.96243034 2.12454191]
 [0.61894364 0.61894364 0.06167939]
 [0.18075829 0.18075829 0.00789697]]
```

current
Object current attribute getter and setter.

diameter
Object diameter attribute getter and setter.

display (*markers=[(0, 0, 0)], axis=None, show_direction=False, show_path=True, size_sensors=1, size_direction=1, size_dipoles=1*)
Display object graphically using matplotlib 3D plotting.

Parameters

- **markers** (*array_like, shape (N,3), default=[(0,0,0)]*) – Display position markers in the global CS. By default a marker is placed in the origin.

- **axis** (*pyplot.axis, default=None*) – Display graphical output in a given pyplot axis (must be 3D). By default a new pyplot figure is created and displayed.
- **show_direction** (*bool, default=False*) – Set True to show magnetization and current directions.
- **show_path** (*bool or int, default=True*) – Options True, False, positive int. By default object paths are shown. If show_path is a positive integer, objects will be displayed at multiple path positions along the path, in steps of show_path.
- **size_sensor** (*float, default=1*) – Adjust automatic display size of sensors.
- **size_direction** (*float, default=1*) – Adjust automatic display size of direction arrows.
- **size_dipoles** (*float, default=1*) – Adjust automatic display size of dipoles.

Returns None

Return type NoneType

Examples

Display Magpylib objects graphically using Matplotlib:

```
>>> import magpylib as mag3
>>> obj = mag3.magnet.Sphere(magnetization=(0,0,1), diameter=1)
>>> obj.move([(.2,0,0)]*50, increment=True)
>>> obj.rotate_from_angax(angle=[10]*50, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> obj.display(show_direction=True, show_path=10)
--> graphic output
```

Display figure on your own 3D Matplotlib axis:

```
>>> import matplotlib.pyplot as plt
>>> import magpylib as mag3
>>> my_axis = plt.axes(projection='3d')
>>> obj = mag3.magnet.Box(magnetization=(0,0,1), dimension=(1,2,3))
>>> obj.move([(x,0,0) for x in [0,1,2,3,4,5]])
>>> obj.display(axis=my_axis)
>>> plt.show()
--> graphic output
```

getB (**observers, squeeze=True*)

Compute B-field in units of [mT] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **B-field** – B-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [mT]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the B-field [mT] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> B = source.getB(sensor)
>>> print(B)
[-0.62497314  0.34089444  0.51134166]
```

Compute the B-field [mT] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> B = source.getB((1,2,3))
>>> print(B)
[[-0.88894262  0.          0.        ]
[-0.62497314 -0.34089444 -0.51134166]
[-0.17483825 -0.41961181 -0.62941771]
[ 0.09177028 -0.33037301 -0.49555952]
[ 0.17480239 -0.22080302 -0.33120453]]
```

Compute the B-field [mT] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> B = source.getB(sens1, sens2)
>>> print(B)
[[0.01421427 0.02842853 0.02114728]
[0.00621368 0.00932052 0.00501254]]
```

getH(**observers*, *squeeze=True*)

Compute H-field in units of [kA/m] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **H-field** – H-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [kA/m]. Sensor pixel positions are equivalent to simple

observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the H-field [kA/m] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> H = source.getH(sensor)
>>> print(H)
[-0.49733782  0.27127518  0.40691277]
```

Compute the H-field [kA/m] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> H = source.getH((1,2,3))
>>> print(H)
[[-0.70739806  0.          0.        ]
[-0.49733782 -0.27127518 -0.40691277]
[-0.13913186 -0.33391647 -0.5008747 ]
[ 0.07302847 -0.26290249 -0.39435373]
[ 0.13910332 -0.17570946 -0.26356419]]
```

Compute the H-field [kA/m] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> H = source.getH(sens1, sens2)
>>> print(H)
[[0.01131135 0.02262271 0.01682847]
[0.00494469 0.00741704 0.00398885]]
```

move (*displacement*, *start=-1*, *increment=False*)

Translates the object by the input displacement (can be a path).

This method uses vector addition to merge the input path given by displacement and the existing old path of an object. It keeps the old orientation. If the input path extends beyond the old path, the old path will be padded by its last entry before paths are added up.

Parameters

- **displacement** (*array_like, shape (3,) or (N,3)*) – Displacement vector shape=(3,) or path shape=(N,3) in units of [mm].
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input displacements are absolute. If *increment=True*, input displacements are interpreted as increments of

each other. For example, an incremental input displacement of *[(2,0,0), (2,0,0), (2,0,0)]* corresponds to an absolute input displacement of *[(2,0,0), (4,0,0), (6,0,0)]*.

Returns self

Return type Magpylib object

Examples

With the `move` method Magpylib objects can be repositioned in the global coordinate system:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> sensor.move((1,1,1))
>>> print(sensor.position)
[1. 1. 1.]
```

It is also a powerful tool for creating paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move((1,1,1), start='append')
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]]
>>> sensor.move([(.1,.1,.1)]*2, start='append')
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [1.1 1.1 1.1]
 [1.1 1.1 1.1]]
```

Complex paths can be generated with ease, by making use of the `increment` keyword and superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move([(1,1,1)]*4, start='append', increment=True)
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
>>> sensor.move([(.1,.1,.1)]*5, start=2)
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [2.1 2.1 2.1]
 [3.1 3.1 3.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]]
```

`orientation`
Object orientation attribute getter and setter.

position
Object position attribute getter and setter.

reset_path()
Reset object path to position = (0,0,0) and orientation = unit rotation.

Returns self

Return type Magpylib object

Examples

Create an object with non-zero path

```
>>> import magpylib as mag3
>>> obj = mag3.Sensor(position=(1,2,3))
>>> print(obj.position)
[1. 2. 3.]
>>> obj.reset_path()
>>> print(obj.position)
[0. 0. 0.]
```

rotate(*rotation*, *anchor=None*, *start=-1*, *increment=False*)
Rotates the object in the global coordinate system by a given rotation input (can be a path).

This method applies given rotations to the original orientation. If the input path extends beyond the existing path, the old path will be padded by its last entry before paths are added up.

Parameters

- **rotation** (*scipy Rotation object*) – Rotation to be applied. The rotation object can feature a single rotation of shape (3,) or a set of rotations of shape (N,3) that correspond to a path.
- **anchor** (*None, 0 or array_like, shape (3,), default=None*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other.

Returns self

Return type Magpylib object

Examples

With the `rotate` method Magpylib objects can be rotated about their local coordinate system center:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
```

(continues on next page)

(continued from previous page)

```
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> rotation_object = R.from_euler('x', 45, degrees=True)
>>> sensor.rotate(rotation_object)
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> rotation_object = R.from_euler('x', [10,20,30], degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
```

(continues on next page)

(continued from previous page)

```
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', [10]*3, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append',␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> rotation_object = R.from_euler('z', [5]*4, degrees=True)
>>> sensor.rotate(rotation_object, anchor=0, start=0, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

rotate_from_angax(*angle*, *axis*, *anchor=None*, *start=-1*, *increment=False*, *degrees=True*)

Object rotation in the global coordinate system from angle-axis input.

This method applies given rotations to the original orientation. If the input path extends beyond the existingp path, the oldpath will be padded by its last entry before paths are added up.

Parameters

- **angle** (*int/float or array_like with shape (n,) unit [deg] (by default)*) – Angle of rotation, or a vector of n angles defining a rotation path in units of [deg] (by default).
- **axis** (*str or array_like, shape (3,)*) – The direction of the axis of rotation. Input can be a vector of shape (3,) or a string 'x', 'y' or 'z' to denote respective directions.
- **anchor** (*None or array_like, shape (3,), default=None, unit [mm]*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other. For example, the incremental angles [1,1,1,2,2] correspond to the absolute angles [1,2,3,5,7].
- **degrees** (*bool, default=True*) – By default angle is given in units of [deg]. If degrees=False, angle is given in units of [rad].

Returns self

Return type Magpylib object

Examples

With the `rotate_from_angax` method Magpylib objects can be rotated about their local coordinte system center:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> sensor.rotate_from_angax(angle=45, axis='x')
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis=(1,0,0), anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis='x', anchor=(0,1,0), start='append
↪')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[10,20,30], axis='x', anchor=(0,1,0),␣
↪start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
```

(continues on next page)

(continued from previous page)

```
 [110.    0.    0.]
 [120.    0.    0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax([10]*3, 'x', (0,1,0), start=1, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[5]*4, axis='z', anchor=0, start=0,
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

class `magpylib.current.`**`Line`**(*current=None, vertices=[None, None], position=(0, 0, 0), orientation=None*)

Bases: `magpylib._lib.obj_classes.class_BaseGeo.BaseGeo`, `magpylib._lib.obj_classes.class_BaseDisplayRepr.BaseDisplayRepr`, `magpylib._lib.obj_classes.class_BaseGetBH.BaseGetBH`, `magpylib._lib.obj_classes.class_BaseExcitations.BaseCurrent`

Current flowing in straight lines from vertex to vertex.

Local object coordinates: The Line current vertices are defined in the local object coordinate system. Local (Line) and global CS coincide when position=(0,0,0) and orientation=unit_rotation.

Parameters

- **current** (*float*) – Electrical current in units of [A].
- **vertices** (*array_like, shape (N,3)*) – The current flows along the vertices which are given in units of [mm] in the local CS of the Line object.
- **position** (*array_like, shape (3,) or (M,3), default=(0,0,0)*) – Object position (local CS origin) in the global CS in units of [mm]. For M>1, the position represents a path. The position and orientation parameters must always be of the same length.
- **orientation** (*scipy Rotation object with length 1 or M, default=unit rotation*) – Object orientation (local CS orientation) in the

global CS. For M>1 orientation represents different values along a path. The position and orientation parameters must always be of the same length.

Returns Line object

Return type *Line*

Examples

By default a Line is initialized at position (0,0,0), with unit rotation:

```
>>> import magpylib as mag3
>>> magnet = mag3.current.Line(current=100, vertices=[(-1,0,0),(1,0,0)])
>>> print(magnet.position)
[0. 0. 0.]
>>> print(magnet.orientation.as_quat())
[0. 0. 0. 1.]
```

Lines are magnetic field sources. Below we compute the H-field [kA/m] of the above Line at the observer position (1,1,1),

```
>>> H = magnet.getH((1,1,1))
>>> print(H)
[ 0.          -3.24873667  3.24873667]
```

or at a set of observer positions:

```
>>> H = magnet.getH([(1,1,1), (2,2,2), (3,3,3)])
>>> print(H)
[[ 0.          -3.24873667  3.24873667]
 [ 0.          -0.78438229  0.78438229]
 [ 0.          -0.34429579  0.34429579]]
```

The same result is obtained when the Line moves along a path, away from the observer:

```
>>> magnet.move([(-1,-1,-1), (-2,-2,-2)], start=1)
>>> H = magnet.getH((1,1,1))
>>> print(H)
[[ 0.          -3.24873667  3.24873667]
 [ 0.          -0.78438229  0.78438229]
 [ 0.          -0.34429579  0.34429579]]
```

current
Object current attribute getter and setter.

display (*markers=[(0, 0, 0)], axis=None, show_direction=False, show_path=True, size_sensors=1, size_direction=1, size_dipoles=1*)
Display object graphically using matplotlib 3D plotting.

Parameters

- **markers** (*array_like, shape (N,3), default=[(0,0,0)]*) – Display position markers in the global CS. By default a marker is placed in the origin.
- **axis** (*pyplot.axis, default=None*) – Display graphical output in a given pyplot axis (must be 3D). By default a new pyplot figure is created and displayed.
- **show_direction** (*bool, default=False*) – Set True to show magnetization and current directions.

- **show_path** (*bool or int, default=True*) – Options True, False, positive int. By default object paths are shown. If show_path is a positive integer, objects will be displayed at multiple path positions along the path, in steps of show_path.
- **size_sensor** (*float, default=1*) – Adjust automatic display size of sensors.
- **size_direction** (*float, default=1*) – Adjust automatic display size of direction arrows.
- **size_dipoles** (*float, default=1*) – Adjust automatic display size of dipoles.

Returns None

Return type NoneType

Examples

Display Magpylib objects graphically using Matplotlib:

```
>>> import magpylib as mag3
>>> obj = mag3.magnet.Sphere(magnetization=(0,0,1), diameter=1)
>>> obj.move([(.2,0,0)]*50, increment=True)
>>> obj.rotate_from_angax(angle=[10]*50, axis='z', anchor=0, start=0,
↪increment=True)
>>> obj.display(show_direction=True, show_path=10)
--> graphic output
```

Display figure on your own 3D Matplotlib axis:

```
>>> import matplotlib.pyplot as plt
>>> import magpylib as mag3
>>> my_axis = plt.axes(projection='3d')
>>> obj = mag3.magnet.Box(magnetization=(0,0,1), dimension=(1,2,3))
>>> obj.move([(x,0,0) for x in [0,1,2,3,4,5]])
>>> obj.display(axis=my_axis)
>>> plt.show()
--> graphic output
```

getB (**observers, squeeze=True*)

Compute B-field in units of [mT] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **B-field** – B-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [mT]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the B-field [mT] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> B = source.getB(sensor)
>>> print(B)
[-0.62497314  0.34089444  0.51134166]
```

Compute the B-field [mT] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> B = source.getB((1,2,3))
>>> print(B)
[[-0.88894262  0.          0.        ]
[-0.62497314 -0.34089444 -0.51134166]
[-0.17483825 -0.41961181 -0.62941771]
[ 0.09177028 -0.33037301 -0.49555952]
[ 0.17480239 -0.22080302 -0.33120453]]
```

Compute the B-field [mT] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> B = source.getB(sens1, sens2)
>>> print(B)
[[0.01421427 0.02842853 0.02114728]
[0.00621368 0.00932052 0.00501254]]
```

getH(**observers*, *squeeze=True*)

Compute H-field in units of [kA/m] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **H-field** – H-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [kA/m]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the H-field [kA/m] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> H = source.getH(sensor)
>>> print(H)
[-0.49733782  0.27127518  0.40691277]
```

Compute the H-field [kA/m] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> H = source.getH((1,2,3))
>>> print(H)
[[-0.70739806  0.          0.        ]
[-0.49733782 -0.27127518 -0.40691277]
[-0.13913186 -0.33391647 -0.5008747 ]
[ 0.07302847 -0.26290249 -0.39435373]
[ 0.13910332 -0.17570946 -0.26356419]]
```

Compute the H-field [kA/m] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> H = source.getH(sens1, sens2)
>>> print(H)
[[0.01131135 0.02262271 0.01682847]
[0.00494469 0.00741704 0.00398885]]
```

move (*displacement*, *start=-1*, *increment=False*)

Translates the object by the input displacement (can be a path).

This method uses vector addition to merge the input path given by displacement and the existing old path of an object. It keeps the old orientation. If the input path extends beyond the old path, the old path will be padded by its last entry before paths are added up.

Parameters

- **displacement** (*array_like, shape (3,) or (N,3)*) – Displacement vector shape=(3,) or path shape=(N,3) in units of [mm].
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input displacements are absolute. If *increment=True*, input displacements are interpreted as increments of each other. For example, an incremental input displacement of *[(2,0,0), (2,0,0), (2,0,0)]* corresponds to an absolute input displacement of *[(2,0,0), (4,0,0), (6,0,0)]*.

Returns self

Return type Magpylib object

Examples

With the `move` method Magpylib objects can be repositioned in the global coordinate system:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> sensor.move((1,1,1))
>>> print(sensor.position)
[1. 1. 1.]
```

It is also a powerful tool for creating paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move((1,1,1), start='append')
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]]
>>> sensor.move([(.1,.1,.1)]*2, start='append')
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [1.1 1.1 1.1]
 [1.1 1.1 1.1]]
```

Complex paths can be generated with ease, by making use of the `increment` keyword and superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move([(1,1,1)]*4, start='append', increment=True)
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
>>> sensor.move([(.1,.1,.1)]*5, start=2)
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [2.1 2.1 2.1]
 [3.1 3.1 3.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]]
```

orientation
Object orientation attribute getter and setter.

position
Object position attribute getter and setter.

reset_path()
Reset object path to position = (0,0,0) and orientation = unit rotation.

Returns self

Return type Magpylib object

Examples

Create an object with non-zero path

```
>>> import magpylib as mag3
>>> obj = mag3.Sensor(position=(1,2,3))
>>> print(obj.position)
[1. 2. 3.]
>>> obj.reset_path()
>>> print(obj.position)
[0. 0. 0.]
```

rotate (*rotation*, *anchor=None*, *start=-1*, *increment=False*)

Rotates the object in the global coordinate system by a given rotation input (can be a path).

This method applies given rotations to the original orientation. If the input path extends beyond the existing path, the old path will be padded by its last entry before paths are added up.

Parameters

- **rotation** (*scipy Rotation object*) – Rotation to be applied. The rotation object can feature a single rotation of shape (3,) or a set of rotations of shape (N,3) that correspond to a path.
- **anchor** (*None, 0 or array_like, shape (3,), default=None*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other.

Returns self

Return type Magpylib object

Examples

With the `rotate` method Magpylib objects can be rotated about their local coordinate system center:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> rotation_object = R.from_euler('x', 45, degrees=True)
>>> sensor.rotate(rotation_object)
>>> print(sensor.position)
[0. 0. 0.]
```

(continues on next page)

(continued from previous page)

```
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the anchor keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> rotation_object = R.from_euler('x', [10,20,30], degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the increment keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', [10]*3, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append',
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
```

(continues on next page)

(continued from previous page)

```
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> rotation_object = R.from_euler('z', [5]*4, degrees=True)
>>> sensor.rotate(rotation_object, anchor=0, start=0, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

rotate_from_angax (*angle*, *axis*, *anchor=None*, *start=-1*, *increment=False*, *degrees=True*)

Object rotation in the global coordinate system from angle-axis input.

This method applies given rotations to the original orientation. If the input path extends beyond the existingp path, the oldpath will be padded by its last entry before paths are added up.

Parameters

- **angle** (*int/float or array_like with shape (n,) unit [deg] (by default)*) – Angle of rotation, or a vector of n angles defining a rotation path in units of [deg] (by default).
- **axis** (*str or array_like, shape (3,)*) – The direction of the axis of rotation. Input can be a vector of shape (3,) or a string 'x', 'y' or 'z' to denote respective directions.
- **anchor** (*None or array_like, shape (3,), default=None, unit [mm]*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other. For example, the incremental angles [1,1,1,2,2] correspond to the absolute angles [1,2,3,5,7].
- **degrees** (*bool, default=True*) – By default angle is given in units of [deg]. If degrees=False, angle is given in units of [rad].

Returns self

Return type Magpylib object

Examples

With the `rotate_from_angax` method Magpylib objects can be rotated about their local coordinte system center:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> sensor.rotate_from_angax(angle=45, axis='x')
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis=(1,0,0), anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis='x', anchor=(0,1,0), start='append
↪')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[10,20,30], axis='x', anchor=(0,1,0),␣
↪start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax([10]*3, 'x', (0,1,0), start=1, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[5]*4, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

vertices
Object vertices attribute getter and setter.

magpylib.magnet package

This subpackage contains all magnet classes. Magnets are modeled with homogeneous magnetization given in units of millitesla [mT] through mu0*M. See documentation for details on field computations.

class magpylib.magnet.**Box**(*magnetization=(None, None, None)*, *dimension=(None, None, None)*, *position=(0, 0, 0)*, *orientation=None*)

Bases: `magpylib._lib.obj_classes.class_BaseGeo.BaseGeo`, `magpylib._lib.obj_classes.class_BaseDisplayRepr.BaseDisplayRepr`, `magpylib._lib.obj_classes.class_BaseGetBH.BaseGetBH`, `magpylib._lib.obj_classes.class_BaseExcitations.BaseHomMag`

Cuboid magnet with homogeneous magnetization.

Local object coordinates: The geometric center of the Box is located in the origin of the local object coordinate system. Box sides are parallel to the local basis vectors. Local (Box) and global CS coincide when position=(0,0,0) and orientation=unit_rotation.

Parameters

- **magnetization** (*array_like, shape (3,)*) – Magnetization vector (mu0*M, remanence field) in units of [mT] given in the local CS of the Box object.
- **dimension** (*array_like, shape (3,)*) – Dimension/Size of the Box with sides [a,b,c] in units of [mm].
- **position** (*array_like, shape (3,) or (M,3), default=(0,0,0)*) – Object position (local CS origin) in the global CS in units of [mm]. For M>1, the position represents a path. The position and orientation parameters must always be of the same

length.

- **orientation** (*scipy Rotation object with length 1 or M, default=unit rotation*) – Object orientation (local CS orientation) in the global CS. For M>1 orientation represents different values along a path. The position and orientation parameters must always be of the same length.

Returns Box object

Return type *Box*

Examples

By default a Box is initialized at position (0,0,0), with unit rotation:

```
>>> import magpylib as mag3
>>> magnet = mag3.magnet.Box(magnetization=(100,100,100), dimension=(1,1,1))
>>> print(magnet.position)
[0. 0. 0.]
>>> print(magnet.orientation.as_quat())
[0. 0. 0. 1.]
```

Boxs are magnetic field sources. Below we compute the H-field [kA/m] of the above Box at the observer position (1,1,1),

```
>>> H = magnet.getH((1,1,1))
>>> print(H)
[2.4844679 2.4844679 2.4844679]
```

or at a set of observer positions:

```
>>> H = magnet.getH([(1,1,1), (2,2,2), (3,3,3)])
>>> print(H)
[[2.4844679  2.4844679  2.4844679 ]
 [0.30499798 0.30499798 0.30499798]
 [0.0902928  0.0902928  0.0902928 ]]
```

The same result is obtained when the Box moves along a path, away from the observer:

```
>>> magnet.move([(-1,-1,-1), (-2,-2,-2)], start=1)
>>> H = magnet.getH((1,1,1))
>>> print(H)
[[2.4844679  2.4844679  2.4844679 ]
 [0.30499798 0.30499798 0.30499798]
 [0.0902928  0.0902928  0.0902928 ]]
```

dimension
Object dimension attribute getter and setter.

display(*markers=[(0, 0, 0)], axis=None, show_direction=False, show_path=True, size_sensors=1, size_direction=1, size_dipoles=1*)
Display object graphically using matplotlib 3D plotting.

Parameters

- **markers** (*array_like, shape (N,3), default=[(0,0,0)]*) – Display position markers in the global CS. By default a marker is placed in the origin.
- **axis** (*pyplot.axis, default=None*) – Display graphical output in a given pyplot axis (must be 3D). By default a new pyplot figure is created and displayed.

- **show_direction** (*bool, default=False*) – Set True to show magnetization and current directions.
- **show_path** (*bool or int, default=True*) – Options True, False, positive int. By default object paths are shown. If show_path is a positive integer, objects will be displayed at multiple path positions along the path, in steps of show_path.
- **size_sensor** (*float, default=1*) – Adjust automatic display size of sensors.
- **size_direction** (*float, default=1*) – Adjust automatic display size of direction arrows.
- **size_dipoles** (*float, default=1*) – Adjust automatic display size of dipoles.

Returns None

Return type NoneType

Examples

Display Magpylib objects graphically using Matplotlib:

```
>>> import magpylib as mag3
>>> obj = mag3.magnet.Sphere(magnetization=(0,0,1), diameter=1)
>>> obj.move([(.2,0,0)]*50, increment=True)
>>> obj.rotate_from_angax(angle=[10]*50, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> obj.display(show_direction=True, show_path=10)
--> graphic output
```

Display figure on your own 3D Matplotlib axis:

```
>>> import matplotlib.pyplot as plt
>>> import magpylib as mag3
>>> my_axis = plt.axes(projection='3d')
>>> obj = mag3.magnet.Box(magnetization=(0,0,1), dimension=(1,2,3))
>>> obj.move([(x,0,0) for x in [0,1,2,3,4,5]])
>>> obj.display(axis=my_axis)
>>> plt.show()
--> graphic output
```

getB (**observers, squeeze=True*)

Compute B-field in units of [mT] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns B-field – B-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [mT]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the B-field [mT] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> B = source.getB(sensor)
>>> print(B)
[-0.62497314  0.34089444  0.51134166]
```

Compute the B-field [mT] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> B = source.getB((1,2,3))
>>> print(B)
[[-0.88894262  0.          0.        ]
[-0.62497314 -0.34089444 -0.51134166]
[-0.17483825 -0.41961181 -0.62941771]
[ 0.09177028 -0.33037301 -0.49555952]
[ 0.17480239 -0.22080302 -0.33120453]]
```

Compute the B-field [mT] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> B = source.getB(sens1, sens2)
>>> print(B)
[[0.01421427 0.02842853 0.02114728]
[0.00621368 0.00932052 0.00501254]]
```

getH(**observers*, *squeeze=True*)

Compute H-field in units of [kA/m] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, …, 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, …, 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **H-field** – H-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,…) in units of [kA/m]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the H-field [kA/m] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> H = source.getH(sensor)
>>> print(H)
[-0.49733782  0.27127518  0.40691277]
```

Compute the H-field [kA/m] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> H = source.getH((1,2,3))
>>> print(H)
[[-0.70739806  0.          0.        ]
[-0.49733782 -0.27127518 -0.40691277]
[-0.13913186 -0.33391647 -0.5008747 ]
[ 0.07302847 -0.26290249 -0.39435373]
[ 0.13910332 -0.17570946 -0.26356419]]
```

Compute the H-field [kA/m] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> H = source.getH(sens1, sens2)
>>> print(H)
[[0.01131135 0.02262271 0.01682847]
[0.00494469 0.00741704 0.00398885]]
```

magnetization

Object magnetization attribute getter and setter.

move (*displacement*, *start=-1*, *increment=False*)

Translates the object by the input displacement (can be a path).

This method uses vector addition to merge the input path given by displacement and the existing old path of an object. It keeps the old orientation. If the input path extends beyond the old path, the old path will be padded by its last entry before paths are added up.

Parameters

- **displacement** (*array_like, shape (3,) or (N,3)*) – Displacement vector shape=(3,) or path shape=(N,3) in units of [mm].
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input displacements are absolute. If *increment=True*, input displacements are interpreted as increments of each other. For example, an incremental input displacement of *[(2,0,0), (2,0,0), (2,0,0)]* corresponds to an absolute input displacement of *[(2,0,0), (4,0,0), (6,0,0)]*.

Returns self

Return type Magpylib object

Examples

With the `move` method Magpylib objects can be repositioned in the global coordinate system:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> sensor.move((1,1,1))
>>> print(sensor.position)
[1. 1. 1.]
```

It is also a powerful tool for creating paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move((1,1,1), start='append')
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]]
>>> sensor.move([(.1,.1,.1)]*2, start='append')
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [1.1 1.1 1.1]
 [1.1 1.1 1.1]]
```

Complex paths can be generated with ease, by making use of the `increment` keyword and superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move([(1,1,1)]*4, start='append', increment=True)
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
>>> sensor.move([(.1,.1,.1)]*5, start=2)
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [2.1 2.1 2.1]
 [3.1 3.1 3.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]]
```

`orientation`
Object orientation attribute getter and setter.

`position`
Object position attribute getter and setter.

reset_path()

Reset object path to position = (0,0,0) and orientation = unit rotation.

Returns self

Return type Magpylib object

Examples

Create an object with non-zero path

```
>>> import magpylib as mag3
>>> obj = mag3.Sensor(position=(1,2,3))
>>> print(obj.position)
[1. 2. 3.]
>>> obj.reset_path()
>>> print(obj.position)
[0. 0. 0.]
```

rotate(*rotation*, *anchor=None*, *start=-1*, *increment=False*)

Rotates the object in the global coordinate system by a given rotation input (can be a path).

This method applies given rotations to the original orientation. If the input path extends beyond the existing path, the old path will be padded by its last entry before paths are added up.

Parameters

- **rotation** (*scipy Rotation object*) – Rotation to be applied. The rotation object can feature a single rotation of shape (3,) or a set of rotations of shape (N,3) that correspond to a path.
- **anchor** (*None, 0 or array_like, shape (3,), default=None*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other.

Returns self

Return type Magpylib object

Examples

With the `rotate` method Magpylib objects can be rotated about their local coordinate system center:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
```

(continues on next page)

(continued from previous page)

```
[0. 0. 0.]
>>> rotation_object = R.from_euler('x', 45, degrees=True)
>>> sensor.rotate(rotation_object)
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> rotation_object = R.from_euler('x', [10,20,30], degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', [10]*3, degrees=True)
```

(continues on next page)

(continued from previous page)

```
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append',␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> rotation_object = R.from_euler('z', [5]*4, degrees=True)
>>> sensor.rotate(rotation_object, anchor=0, start=0, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

rotate_from_angax (*angle*, *axis*, *anchor=None*, *start=-1*, *increment=False*, *degrees=True*)

Object rotation in the global coordinate system from angle-axis input.

This method applies given rotations to the original orientation. If the input path extends beyond the existingp path, the oldpath will be padded by its last entry before paths are added up.

Parameters

- **angle** (*int/float or array_like with shape (n,) unit [deg] (by default)*) – Angle of rotation, or a vector of n angles defining a rotation path in units of [deg] (by default).
- **axis** (*str or array_like, shape (3,)*) – The direction of the axis of rotation. Input can be a vector of shape (3,) or a string 'x', 'y' or 'z' to denote respective directions.
- **anchor** (*None or array_like, shape (3,), default=None, unit [mm]*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other. For example, the incremental angles [1,1,1,2,2] correspond to the absolute angles [1,2,3,5,7].
- **degrees** (*bool, default=True*) – By default angle is given in units of [deg]. If degrees=False, angle is given in units of [rad].

Returns self

Return type Magpylib object

Examples

With the `rotate_from_angax` method Magpylib objects can be rotated about their local coordinte system center:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> sensor.rotate_from_angax(angle=45, axis='x')
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis=(1,0,0), anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis='x', anchor=(0,1,0), start='append
↪')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[10,20,30], axis='x', anchor=(0,1,0),␣
↪start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax([10]*3, 'x', (0,1,0), start=1, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[5]*4, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

class magpylib.magnet.**Cylinder**(*magnetization=(None, None, None)*, *dimension=(None, None)*, *position=(0, 0, 0)*, *orientation=None*)

Bases: `magpylib._lib.obj_classes.class_BaseGeo.BaseGeo`, `magpylib._lib.obj_classes.class_BaseDisplayRepr.BaseDisplayRepr`, `magpylib._lib.obj_classes.class_BaseGetBH.BaseGetBH`, `magpylib._lib.obj_classes.class_BaseExcitations.BaseHomMag`

Cylinder magnet with homogeneous magnetization.

Local object coordinates: The geometric center of the Cylinder is located in the origin of the local object coordinate system. The Cylinder axis conincides with the local CS z-axis. Local (Cylinder) and global CS coincide when position=(0,0,0) and orientation=unit_rotation.

By default ITER_CYLINDER=50 for iteration of diametral magnetization computation. Use `magpylib.Config.ITER_CYLINDER=X` to change this setting.

Parameters

- **magnetization** (*array_like, shape (3,)*) – Magnetization vector (mu0*M, remanence field) in units of [mT] given in the local CS of the Cylinder object.
- **dimension** (*array_like, shape (2,)*) – Dimension/Size of the Cylinder with diameter/height (d,h) in units of [mm].
- **position** (*array_like, shape (3,) or (M,3), default=(0,0,0)*) – Object position (local CS origin) in the global CS in units of [mm]. For M>1, the position represents a path. The position and orientation parameters must always be of the same length.
- **orientation** (*scipy Rotation object with length 1 or M, default=unit rotation*) – Object orientation (local CS orientation) in the global CS. For M>1 orientation represents different values along a path. The position and orientation parameters must always be of the same length.

Returns Cylinder object

Return type *Cylinder*

Examples

By default a Cylinder is initialized at position (0,0,0), with unit rotation:

```
>>> import magpylib as mag3
>>> magnet = mag3.magnet.Cylinder(magnetization=(100,100,100), dimension=(1,1))
>>> print(magnet.position)
[0. 0. 0.]
>>> print(magnet.orientation.as_quat())
[0. 0. 0. 1.]
```

Cylinders are magnetic field sources. Below we compute the H-field [kA/m] of the above Cylinder at the observer position (1,1,1),

```
>>> H = magnet.getH((1,1,1))
>>> print(H)
[1.95851744 1.95851744 1.8657571 ]
```

or at a set of observer positions:

```
>>> H = magnet.getH([(1,1,1), (2,2,2), (3,3,3)])
>>> print(H)
[[1.95851744 1.95851744 1.8657571 ]
 [0.24025917 0.24025917 0.23767364]
 [0.07101874 0.07101874 0.07068512]]
```

The same result is obtained when the Cylinder moves along a path, away from the observer:

```
>>> magnet.move([(-1,-1,-1), (-2,-2,-2)], start=1)
>>> H = magnet.getH((1,1,1))
>>> print(H)
[[1.95851744 1.95851744 1.8657571 ]
 [0.24025917 0.24025917 0.23767364]
 [0.07101874 0.07101874 0.07068512]]
```

dimension
Object dimension attribute getter and setter.

display (*markers=[(0, 0, 0)], axis=None, show_direction=False, show_path=True, size_sensors=1, size_direction=1, size_dipoles=1*)
Display object graphically using matplotlib 3D plotting.

Parameters

- **markers** (*array_like, shape (N,3), default=[(0,0,0)]*) – Display position markers in the global CS. By default a marker is placed in the origin.
- **axis** (*pyplot.axis, default=None*) – Display graphical output in a given pyplot axis (must be 3D). By default a new pyplot figure is created and displayed.
- **show_direction** (*bool, default=False*) – Set True to show magnetization and current directions.
- **show_path** (*bool or int, default=True*) – Options True, False, positive int. By default object paths are shown. If show_path is a positive integer, objects will be displayed at multiple path positions along the path, in steps of show_path.

- **size_sensor** (*float, default=1*) – Adjust automatic display size of sensors.
- **size_direction** (*float, default=1*) – Adjust automatic display size of direction arrows.
- **size_dipoles** (*float, default=1*) – Adjust automatic display size of dipoles.

Returns None

Return type NoneType

Examples

Display Magpylib objects graphically using Matplotlib:

```
>>> import magpylib as mag3
>>> obj = mag3.magnet.Sphere(magnetization=(0,0,1), diameter=1)
>>> obj.move([(.2,0,0)]*50, increment=True)
>>> obj.rotate_from_angax(angle=[10]*50, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> obj.display(show_direction=True, show_path=10)
--> graphic output
```

Display figure on your own 3D Matplotlib axis:

```
>>> import matplotlib.pyplot as plt
>>> import magpylib as mag3
>>> my_axis = plt.axes(projection='3d')
>>> obj = mag3.magnet.Box(magnetization=(0,0,1), dimension=(1,2,3))
>>> obj.move([(x,0,0) for x in [0,1,2,3,4,5]])
>>> obj.display(axis=my_axis)
>>> plt.show()
--> graphic output
```

getB(**observers, squeeze=True*)

Compute B-field in units of [mT] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **B-field** – B-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [mT]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the B-field [mT] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> B = source.getB(sensor)
>>> print(B)
[-0.62497314  0.34089444  0.51134166]
```

Compute the B-field [mT] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> B = source.getB((1,2,3))
>>> print(B)
[[-0.88894262  0.          0.        ]
[-0.62497314 -0.34089444 -0.51134166]
[-0.17483825 -0.41961181 -0.62941771]
[ 0.09177028 -0.33037301 -0.49555952]
[ 0.17480239 -0.22080302 -0.33120453]]
```

Compute the B-field [mT] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> B = source.getB(sens1, sens2)
>>> print(B)
[[0.01421427 0.02842853 0.02114728]
[0.00621368 0.00932052 0.00501254]]
```

getH(**observers, squeeze=True*)

Compute H-field in units of [kA/m] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns H-field – H-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [kA/m]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the H-field [kA/m] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> H = source.getH(sensor)
>>> print(H)
[-0.49733782  0.27127518  0.40691277]
```

Compute the H-field [kA/m] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> H = source.getH((1,2,3))
>>> print(H)
[[-0.70739806  0.          0.        ]
[-0.49733782 -0.27127518 -0.40691277]
[-0.13913186 -0.33391647 -0.5008747 ]
[ 0.07302847 -0.26290249 -0.39435373]
[ 0.13910332 -0.17570946 -0.26356419]]
```

Compute the H-field [kA/m] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> H = source.getH(sens1, sens2)
>>> print(H)
[[0.01131135 0.02262271 0.01682847]
[0.00494469 0.00741704 0.00398885]]
```

magnetization
Object magnetization attribute getter and setter.

move (*displacement*, *start=-1*, *increment=False*)
Translates the object by the input displacement (can be a path).

This method uses vector addition to merge the input path given by displacement and the existing old path of an object. It keeps the old orientation. If the input path extends beyond the old path, the old path will be padded by its last entry before paths are added up.

Parameters

- **displacement** (*array_like, shape (3,) or (N,3)*) – Displacement vector shape=(3,) or path shape=(N,3) in units of [mm].
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input displacements are absolute. If *increment=True*, input displacements are interpreted as increments of each other. For example, an incremental input displacement of *[(2,0,0), (2,0,0), (2,0,0)]* corresponds to an absolute input displacement of *[(2,0,0), (4,0,0), (6,0,0)]*.

Returns self

Return type Magpylib object

Examples

With the `move` method Magpylib objects can be repositioned in the global coordinate system:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> sensor.move((1,1,1))
>>> print(sensor.position)
[1. 1. 1.]
```

It is also a powerful tool for creating paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move((1,1,1), start='append')
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]]
>>> sensor.move([(.1,.1,.1)]*2, start='append')
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [1.1 1.1 1.1]
 [1.1 1.1 1.1]]
```

Complex paths can be generated with ease, by making use of the `increment` keyword and superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move([(1,1,1)]*4, start='append', increment=True)
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
>>> sensor.move([(.1,.1,.1)]*5, start=2)
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [2.1 2.1 2.1]
 [3.1 3.1 3.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]]
```

`orientation`
Object orientation attribute getter and setter.

`position`
Object position attribute getter and setter.

`reset_path()`
Reset object path to position = (0,0,0) and orientation = unit rotation.

Returns self

Return type Magpylib object

Examples

Create an object with non-zero path

```
>>> import magpylib as mag3
>>> obj = mag3.Sensor(position=(1,2,3))
>>> print(obj.position)
[1. 2. 3.]
>>> obj.reset_path()
>>> print(obj.position)
[0. 0. 0.]
```

rotate (*rotation*, *anchor=None*, *start=-1*, *increment=False*)

Rotates the object in the global coordinate system by a given rotation input (can be a path).

This method applies given rotations to the original orientation. If the input path extends beyond the existing path, the old path will be padded by its last entry before paths are added up.

Parameters

- **rotation** (*scipy Rotation object*) – Rotation to be applied. The rotation object can feature a single rotation of shape (3,) or a set of rotations of shape (N,3) that correspond to a path.
- **anchor** (*None, 0 or array_like, shape (3,), default=None*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other.

Returns self

Return type Magpylib object

Examples

With the `rotate` method Magpylib objects can be rotated about their local coordinate system center:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> rotation_object = R.from_euler('x', 45, degrees=True)
>>> sensor.rotate(rotation_object)
>>> print(sensor.position)
[0. 0. 0.]
```

(continues on next page)

(continued from previous page)

```
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> rotation_object = R.from_euler('x', [10,20,30], degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', [10]*3, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append',␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
```

(continues on next page)

(continued from previous page)

```
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> rotation_object = R.from_euler('z', [5]*4, degrees=True)
>>> sensor.rotate(rotation_object, anchor=0, start=0, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

rotate_from_angax (*angle*, *axis*, *anchor=None*, *start=-1*, *increment=False*, *degrees=True*)

Object rotation in the global coordinate system from angle-axis input.

This method applies given rotations to the original orientation. If the input path extends beyond the existingp path, the oldpath will be padded by its last entry before paths are added up.

Parameters

- **angle** (*int/float or array_like with shape (n,) unit [deg] (by default)*) – Angle of rotation, or a vector of n angles defining a rotation path in units of [deg] (by default).
- **axis** (*str or array_like, shape (3,)*) – The direction of the axis of rotation. Input can be a vector of shape (3,) or a string ‘x’, ‘y’ or ‘z’ to denote respective directions.
- **anchor** (*None or array_like, shape (3,), default=None, unit [mm]*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other. For example, the incremental angles [1,1,1,2,2] correspond to the absolute angles [1,2,3,5,7].
- **degrees** (*bool, default=True*) – By default angle is given in units of [deg]. If degrees=False, angle is given in units of [rad].

Returns self

Return type Magpylib object

Examples

With the `rotate_from_angax` method Magpylib objects can be rotated about their local coordinte system center:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> sensor.rotate_from_angax(angle=45, axis='x')
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis=(1,0,0), anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis='x', anchor=(0,1,0), start='append
↪')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[10,20,30], axis='x', anchor=(0,1,0),␣
↪start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax([10]*3, 'x', (0,1,0), start=1, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[5]*4, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

class `magpylib.magnet.`**`Sphere`**(*magnetization=(None, None, None), diameter=None, position=(0, 0, 0), orientation=None*)

Bases: `magpylib._lib.obj_classes.class_BaseGeo.BaseGeo`, `magpylib._lib.obj_classes.class_BaseDisplayRepr.BaseDisplayRepr`, `magpylib._lib.obj_classes.class_BaseGetBH.BaseGetBH`, `magpylib._lib.obj_classes.class_BaseExcitations.BaseHomMag`

Spherical magnet with homogeneous magnetization.

Local object coordinates: The Sphere center is located in the origin of the local object coordinate system. Local (Sphere) and global CS coincide when position=(0,0,0) and orientation=unit_rotation.

Parameters

- **magnetization** (*array_like, shape (3,)*) – Magnetization vector (mu0*M, remanence field) in units of [mT] given in the local CS of the Sphere object.
- **diameter** (*float*) – Diameter of the Sphere in units of [mm].
- **position** (*array_like, shape (3,) or (M,3), default=(0,0,0)*) – Object position (local CS origin) in the global CS in units of [mm]. For M>1, the position represents a path. The position and orientation parameters must always be of the same length.
- **orientation** (*scipy Rotation object with length 1 or M, default=unit rotation*) – Object orientation (local CS orientation) in the global CS. For M>1 orientation represents different values along a path. The position and orientation parameters must always be of the same length.

Returns Sphere object

Return type *Sphere*

Examples

By default a Sphere is initialized at position (0,0,0), with unit rotation:

```
>>> import magpylib as mag3
>>> magnet = mag3.magnet.Sphere(magnetization=(100,100,100), diameter=1)
>>> print(magnet.position)
[0. 0. 0.]
>>> print(magnet.orientation.as_quat())
[0. 0. 0. 1.]
```

Spheres are magnetic field sources. Below we compute the H-field [kA/m] of the above Sphere at the observer position (1,1,1),

```
>>> H = magnet.getH((1,1,1))
>>> print(H)
[1.27622429 1.27622429 1.27622429]
```

or at a set of observer positions:

```
>>> H = magnet.getH([(1,1,1), (2,2,2), (3,3,3)])
>>> print(H)
[[1.27622429 1.27622429 1.27622429]
 [0.15952804 0.15952804 0.15952804]
 [0.04726757 0.04726757 0.04726757]]
```

The same result is obtained when the Sphere object moves along a path, away from the observer:

```
>>> magnet.move([(-1,-1,-1), (-2,-2,-2)], start=1)
>>> H = magnet.getH((1,1,1))
>>> print(H)
[[1.27622429 1.27622429 1.27622429]
 [0.15952804 0.15952804 0.15952804]
 [0.04726757 0.04726757 0.04726757]]
```

diameter
Object diameter attribute getter and setter.

display (*markers=[(0, 0, 0)], axis=None, show_direction=False, show_path=True, size_sensors=1, size_direction=1, size_dipoles=1*)
Display object graphically using matplotlib 3D plotting.

Parameters

- **markers** (*array_like, shape (N,3), default=[(0,0,0)]*) – Display position markers in the global CS. By default a marker is placed in the origin.
- **axis** (*pyplot.axis, default=None*) – Display graphical output in a given pyplot axis (must be 3D). By default a new pyplot figure is created and displayed.
- **show_direction** (*bool, default=False*) – Set True to show magnetization and current directions.
- **show_path** (*bool or int, default=True*) – Options True, False, positive int. By default object paths are shown. If show_path is a positive integer, objects will be displayed at multiple path positions along the path, in steps of show_path.
- **size_sensor** (*float, default=1*) – Adjust automatic display size of sensors.
- **size_direction** (*float, default=1*) – Adjust automatic display size of direction arrows.

- **size_dipoles** (*float, default=1*) – Adjust automatic display size of dipoles.

Returns None

Return type NoneType

Examples

Display Magpylib objects graphically using Matplotlib:

```
>>> import magpylib as mag3
>>> obj = mag3.magnet.Sphere(magnetization=(0,0,1), diameter=1)
>>> obj.move([(.2,0,0)]*50, increment=True)
>>> obj.rotate_from_angax(angle=[10]*50, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> obj.display(show_direction=True, show_path=10)
--> graphic output
```

Display figure on your own 3D Matplotlib axis:

```
>>> import matplotlib.pyplot as plt
>>> import magpylib as mag3
>>> my_axis = plt.axes(projection='3d')
>>> obj = mag3.magnet.Box(magnetization=(0,0,1), dimension=(1,2,3))
>>> obj.move([(x,0,0) for x in [0,1,2,3,4,5]])
>>> obj.display(axis=my_axis)
>>> plt.show()
--> graphic output
```

getB(**observers, squeeze=True*)

Compute B-field in units of [mT] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **B-field** – B-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [mT]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the B-field [mT] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
```

(continues on next page)

(continued from previous page)

```
>>> B = source.getB(sensor)
>>> print(B)
[-0.62497314  0.34089444  0.51134166]
```

Compute the B-field [mT] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> B = source.getB((1,2,3))
>>> print(B)
[[-0.88894262  0.          0.        ]
[-0.62497314 -0.34089444 -0.51134166]
[-0.17483825 -0.41961181 -0.62941771]
[ 0.09177028 -0.33037301 -0.49555952]
[ 0.17480239 -0.22080302 -0.33120453]]
```

Compute the B-field [mT] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> B = source.getB(sens1, sens2)
>>> print(B)
[[0.01421427 0.02842853 0.02114728]
[0.00621368 0.00932052 0.00501254]]
```

getH(**observers*, *squeeze=True*)

Compute H-field in units of [kA/m] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **H-field** – H-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [kA/m]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the H-field [kA/m] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
```

(continues on next page)

(continued from previous page)

```
>>> H = source.getH(sensor)
>>> print(H)
[-0.49733782  0.27127518  0.40691277]
```

Compute the H-field [kA/m] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> H = source.getH((1,2,3))
>>> print(H)
[[-0.70739806  0.          0.        ]
[-0.49733782 -0.27127518 -0.40691277]
[-0.13913186 -0.33391647 -0.5008747 ]
[ 0.07302847 -0.26290249 -0.39435373]
[ 0.13910332 -0.17570946 -0.26356419]]
```

Compute the H-field [kA/m] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> H = source.getH(sens1, sens2)
>>> print(H)
[[0.01131135 0.02262271 0.01682847]
[0.00494469 0.00741704 0.00398885]]
```

magnetization
Object magnetization attribute getter and setter.

move (*displacement*, *start=-1*, *increment=False*)
Translates the object by the input displacement (can be a path).

This method uses vector addition to merge the input path given by displacement and the existing old path of an object. It keeps the old orientation. If the input path extends beyond the old path, the old path will be padded by its last entry before paths are added up.

Parameters

- **displacement** (*array_like, shape (3,) or (N,3)*) – Displacement vector shape=(3,) or path shape=(N,3) in units of [mm].
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input displacements are absolute. If *increment=True*, input displacements are interpreted as increments of each other. For example, an incremental input displacement of *[(2,0,0), (2,0,0), (2,0,0)]* corresponds to an absolute input displacement of *[(2,0,0), (4,0,0), (6,0,0)]*.

Returns self

Return type Magpylib object

Examples

With the `move` method Magpylib objects can be repositioned in the global coordinate system:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> sensor.move((1,1,1))
>>> print(sensor.position)
[1. 1. 1.]
```

It is also a powerful tool for creating paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move((1,1,1), start='append')
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]]
>>> sensor.move([(.1,.1,.1)]*2, start='append')
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [1.1 1.1 1.1]
 [1.1 1.1 1.1]]
```

Complex paths can be generated with ease, by making use of the `increment` keyword and superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move([(1,1,1)]*4, start='append', increment=True)
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
>>> sensor.move([(.1,.1,.1)]*5, start=2)
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [2.1 2.1 2.1]
 [3.1 3.1 3.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]]
```

orientation
Object orientation attribute getter and setter.

position
Object position attribute getter and setter.

reset_path()
Reset object path to position = (0,0,0) and orientation = unit rotation.

Returns self

Return type Magpylib object

Examples

Create an object with non-zero path

```
>>> import magpylib as mag3
>>> obj = mag3.Sensor(position=(1,2,3))
>>> print(obj.position)
[1. 2. 3.]
>>> obj.reset_path()
>>> print(obj.position)
[0. 0. 0.]
```

rotate(*rotation*, *anchor=None*, *start=-1*, *increment=False*)

Rotates the object in the global coordinate system by a given rotation input (can be a path).

This method applies given rotations to the original orientation. If the input path extends beyond the existing path, the old path will be padded by its last entry before paths are added up.

Parameters

- **rotation** (*scipy Rotation object*) – Rotation to be applied. The rotation object can feature a single rotation of shape (3,) or a set of rotations of shape (N,3) that correspond to a path.
- **anchor** (*None, 0 or array_like, shape (3,), default=None*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other.

Returns self

Return type Magpylib object

Examples

With the `rotate` method Magpylib objects can be rotated about their local coordinate system center:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> rotation_object = R.from_euler('x', 45, degrees=True)
>>> sensor.rotate(rotation_object)
>>> print(sensor.position)
[0. 0. 0.]
```

(continues on next page)

(continued from previous page)

```
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> rotation_object = R.from_euler('x', [10,20,30], degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', [10]*3, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append',
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
```

(continues on next page)

(continued from previous page)

```
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> rotation_object = R.from_euler('z', [5]*4, degrees=True)
>>> sensor.rotate(rotation_object, anchor=0, start=0, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

rotate_from_angax(*angle*, *axis*, *anchor=None*, *start=-1*, *increment=False*, *degrees=True*)

Object rotation in the global coordinate system from angle-axis input.

This method applies given rotations to the original orientation. If the input path extends beyond the existingp path, the oldpath will be padded by its last entry before paths are added up.

Parameters

- **angle** (*int/float or array_like with shape (n,) unit [deg] (by default)*) – Angle of rotation, or a vector of n angles defining a rotation path in units of [deg] (by default).
- **axis** (*str or array_like, shape (3,)*) – The direction of the axis of rotation. Input can be a vector of shape (3,) or a string 'x', 'y' or 'z' to denote respective directions.
- **anchor** (*None or array_like, shape (3,), default=None, unit [mm]*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other. For example, the incremental angles [1,1,1,2,2] correspond to the absolute angles [1,2,3,5,7].
- **degrees** (*bool, default=True*) – By default angle is given in units of [deg]. If degrees=False, angle is given in units of [rad].

Returns self

Return type Magpylib object

Examples

With the `rotate_from_angax` method Magpylib objects can be rotated about their local coordinte system center:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> sensor.rotate_from_angax(angle=45, axis='x')
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis=(1,0,0), anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis='x', anchor=(0,1,0), start='append
↪')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[10,20,30], axis='x', anchor=(0,1,0),␣
↪start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax([10]*3, 'x', (0,1,0), start=1, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[5]*4, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

magpylib.misc package

This sub-package contains miscellaneous source objects.

class `magpylib.misc.`**`Dipole`**(*moment=(None, None, None)*, *position=(0, 0, 0)*, *orientation=None*)

Bases: `magpylib._lib.obj_classes.class_BaseGeo.BaseGeo`, `magpylib._lib.obj_classes.class_BaseDisplayRepr.BaseDisplayRepr`, `magpylib._lib.obj_classes.class_BaseGetBH.BaseGetBH`

Magnetic dipole moment.

Local object coordinates: The Dipole is located in the origin of the local object coordinate system. Local (Dipole) and global CS coincide when position=(0,0,0) and orientation=unit_rotation.

Parameters

- **moment** (*array_like, shape (3,), unit [mT*mm^3]*) – Magnetic dipole moment in units of [mT*mm^3] given in the local CS. For homogeneous magnets there is a relation moment=magnetization*volume.
- **position** (*array_like, shape (3,) or (M,3), default=(0,0,0)*) – Object position (local CS origin) in the global CS in units of [mm]. For M>1, the position represents a path. The position and orientation parameters must always be of the same length.
- **orientation** (*scipy Rotation object with length 1 or M, default=unit rotation*) – Object orientation (local CS orientation) in the global CS. For M>1 orientation represents different values along a path. The position and orientation parameters must always be of the same length.

Returns Dipole object

Return type *Dipole*

Examples

By default a Dipole is initialized at position (0,0,0), with unit rotation:

```
>>> import magpylib as mag3
>>> dipole = mag3.misc.Dipole(moment=(100,100,100))
>>> print(dipole.position)
[0. 0. 0.]
>>> print(dipole.orientation.as_quat())
[0. 0. 0. 1.]
```

Dipoles are magnetic field sources. Below we compute the H-field [kA/m] of the above Dipole at an observer position (1,1,1),

```
>>> H = dipole.getH((1,1,1))
>>> print(H)
[2.43740886 2.43740886 2.43740886]
```

or at a set of observer positions:

```
>>> H = dipole.getH([(1,1,1), (2,2,2), (3,3,3)])
>>> print(H)
[[2.43740886 2.43740886 2.43740886]
 [0.30467611 0.30467611 0.30467611]
 [0.0902744  0.0902744  0.0902744 ]]
```

The same result is obtained when the Dipole object moves along a path, away from the observer:

```
>>> dipole.move([(-1,-1,-1), (-2,-2,-2)], start=1)
>>> H = dipole.getH((1,1,1))
>>> print(H)
[[2.43740886 2.43740886 2.43740886]
 [0.30467611 0.30467611 0.30467611]
 [0.0902744  0.0902744  0.0902744 ]]
```

display (*markers=[(0, 0, 0)], axis=None, show_direction=False, show_path=True, size_sensors=1, size_direction=1, size_dipoles=1*)

Display object graphically using matplotlib 3D plotting.

Parameters

- **markers** (*array_like, shape (N,3), default=[(0,0,0)]*) – Display position markers in the global CS. By default a marker is placed in the origin.
- **axis** (*pyplot.axis, default=None*) – Display graphical output in a given pyplot axis (must be 3D). By default a new pyplot figure is created and displayed.
- **show_direction** (*bool, default=False*) – Set True to show magnetization and current directions.
- **show_path** (*bool or int, default=True*) – Options True, False, positive int. By default object paths are shown. If show_path is a positive integer, objects will be displayed at multiple path positions along the path, in steps of show_path.
- **size_sensor** (*float, default=1*) – Adjust automatic display size of sensors.
- **size_direction** (*float, default=1*) – Adjust automatic display size of direction arrows.
- **size_dipoles** (*float, default=1*) – Adjust automatic display size of dipoles.

Returns None

Return type NoneType

Examples

Display Magpylib objects graphically using Matplotlib:

```
>>> import magpylib as mag3
>>> obj = mag3.magnet.Sphere(magnetization=(0,0,1), diameter=1)
>>> obj.move([(.2,0,0)]*50, increment=True)
>>> obj.rotate_from_angax(angle=[10]*50, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> obj.display(show_direction=True, show_path=10)
--> graphic output
```

Display figure on your own 3D Matplotlib axis:

```
>>> import matplotlib.pyplot as plt
>>> import magpylib as mag3
>>> my_axis = plt.axes(projection='3d')
>>> obj = mag3.magnet.Box(magnetization=(0,0,1), dimension=(1,2,3))
>>> obj.move([(x,0,0) for x in [0,1,2,3,4,5]])
>>> obj.display(axis=my_axis)
>>> plt.show()
--> graphic output
```

getB (**observers*, *squeeze=True*)

Compute B-field in units of [mT] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **B-field** – B-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [mT]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the B-field [mT] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> B = source.getB(sensor)
```

(continues on next page)

(continued from previous page)

```
>>> print(B)
[-0.62497314  0.34089444  0.51134166]
```

Compute the B-field [mT] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> B = source.getB((1,2,3))
>>> print(B)
[[-0.88894262  0.          0.        ]
[-0.62497314 -0.34089444 -0.51134166]
[-0.17483825 -0.41961181 -0.62941771]
[ 0.09177028 -0.33037301 -0.49555952]
[ 0.17480239 -0.22080302 -0.33120453]]
```

Compute the B-field [mT] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> B = source.getB(sens1, sens2)
>>> print(B)
[[0.01421427 0.02842853 0.02114728]
[0.00621368 0.00932052 0.00501254]]
```

getH(**observers*, *squeeze=True*)

Compute H-field in units of [kA/m] for given observers.

Parameters

- **observers** (*array_like or Sensors*) – Observers can be array_like positions of shape (N1, N2, . . . , 3) where the field should be evaluated or Sensor objects with pixel shape (N1, N2, . . . , 3). Pixel shapes (or observer positions) of all inputs must be the same. All positions are given in units of [mm].
- **sumup** (*bool, default=False*) – If True, the fields of all sources are summed up.
- **squeeze** (*bool, default=True*) – If True, the output is squeezed, i.e. all axes of length 1 in the output (e.g. only a single source) are eliminated.

Returns **H-field** – H-field at each path position (M) for each sensor (K) and each sensor pixel position (N1,N2,. . .) in units of [kA/m]. Sensor pixel positions are equivalent to simple observer positions. Paths of objects that are shorter than M will be considered as static beyond their end.

Return type ndarray, shape squeeze(M, K, N1, N2, .., 3)

Examples

Compute the H-field [kA/m] at a sensor directly through the source method:

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> sensor = mag3.Sensor(position=(1,2,3))
>>> H = source.getH(sensor)
```

(continues on next page)

(continued from previous page)

```
>>> print(H)
[-0.49733782  0.27127518  0.40691277]
```

Compute the H-field [kA/m] of a source at five path positions as seen by an observer at position (1,2,3):

```
>>> import magpylib as mag3
>>> source = mag3.magnet.Sphere(magnetization=(1000,0,0), diameter=1)
>>> source.move([(x,0,0) for x in [1,2,3,4,5]])
>>> H = source.getH((1,2,3))
>>> print(H)
[[-0.70739806  0.          0.        ]
[-0.49733782 -0.27127518 -0.40691277]
[-0.13913186 -0.33391647 -0.5008747 ]
[ 0.07302847 -0.26290249 -0.39435373]
[ 0.13910332 -0.17570946 -0.26356419]]
```

Compute the H-field [kA/m] of a source at two sensors:

```
>>> import magpylib as mag3
>>> source = mag3.current.Circular(current=15, diameter=1)
>>> sens1 = mag3.Sensor(position=(1,2,3))
>>> sens2 = mag3.Sensor(position=(2,3,4))
>>> H = source.getH(sens1, sens2)
>>> print(H)
[[0.01131135 0.02262271 0.01682847]
[0.00494469 0.00741704 0.00398885]]
```

moment
Object moment attributes getter and setter.

move (*displacement*, *start=-1*, *increment=False*)
Translates the object by the input displacement (can be a path).

This method uses vector addition to merge the input path given by displacement and the existing old path of an object. It keeps the old orientation. If the input path extends beyond the old path, the old path will be padded by its last entry before paths are added up.

Parameters

- **displacement** (*array_like, shape (3,) or (N,3)*) – Displacement vector shape=(3,) or path shape=(N,3) in units of [mm].
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input displacements are absolute. If *increment=True*, input displacements are interpreted as increments of each other. For example, an incremental input displacement of *[(2,0,0), (2,0,0), (2,0,0)]* corresponds to an absolute input displacement of *[(2,0,0), (4,0,0), (6,0,0)]*.

Returns self

Return type Magpylib object

Examples

With the `move` method Magpylib objects can be repositioned in the global coordinate system:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> sensor.move((1,1,1))
>>> print(sensor.position)
[1. 1. 1.]
```

It is also a powerful tool for creating paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move((1,1,1), start='append')
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]]
>>> sensor.move([(.1,.1,.1)]*2, start='append')
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [1.1 1.1 1.1]
 [1.1 1.1 1.1]]
```

Complex paths can be generated with ease, by making use of the `increment` keyword and superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.move([(1,1,1)]*4, start='append', increment=True)
>>> print(sensor.position)
[[0. 0. 0.]
 [1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
>>> sensor.move([(.1,.1,.1)]*5, start=2)
>>> print(sensor.position)
[[0.  0.  0. ]
 [1.  1.  1. ]
 [2.1 2.1 2.1]
 [3.1 3.1 3.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]
 [4.1 4.1 4.1]]
```

orientation
Object orientation attribute getter and setter.

position
Object position attribute getter and setter.

reset_path()
Reset object path to position = (0,0,0) and orientation = unit rotation.

Returns self

Return type Magpylib object

Examples

Create an object with non-zero path

```
>>> import magpylib as mag3
>>> obj = mag3.Sensor(position=(1,2,3))
>>> print(obj.position)
[1. 2. 3.]
>>> obj.reset_path()
>>> print(obj.position)
[0. 0. 0.]
```

rotate(*rotation*, *anchor=None*, *start=-1*, *increment=False*)

Rotates the object in the global coordinate system by a given rotation input (can be a path).

This method applies given rotations to the original orientation. If the input path extends beyond the existing path, the old path will be padded by its last entry before paths are added up.

Parameters

- **rotation** (*scipy Rotation object*) – Rotation to be applied. The rotation object can feature a single rotation of shape (3,) or a set of rotations of shape (N,3) that correspond to a path.
- **anchor** (*None, 0 or array_like, shape (3,), default=None*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other.

Returns self

Return type Magpylib object

Examples

With the `rotate` method Magpylib objects can be rotated about their local coordinate system center:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> rotation_object = R.from_euler('x', 45, degrees=True)
>>> sensor.rotate(rotation_object)
>>> print(sensor.position)
[0. 0. 0.]
```

(continues on next page)

(continued from previous page)

```
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the anchor keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', 90, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> rotation_object = R.from_euler('x', [10,20,30], degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the increment keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> from scipy.spatial.transform import Rotation as R
>>> sensor = mag3.Sensor()
>>> rotation_object = R.from_euler('x', [10]*3, degrees=True)
>>> sensor.rotate(rotation_object, anchor=(0,1,0), start='append',␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
```

(continues on next page)

(continued from previous page)

```
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> rotation_object = R.from_euler('z', [5]*4, degrees=True)
>>> sensor.rotate(rotation_object, anchor=0, start=0, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

rotate_from_angax (*angle*, *axis*, *anchor=None*, *start=-1*, *increment=False*, *degrees=True*)

Object rotation in the global coordinate system from angle-axis input.

This method applies given rotations to the original orientation. If the input path extends beyond the existingp path, the oldpath will be padded by its last entry before paths are added up.

Parameters

- **angle** (*int/float or array_like with shape (n,) unit [deg] (by default)*) – Angle of rotation, or a vector of n angles defining a rotation path in units of [deg] (by default).
- **axis** (*str or array_like, shape (3,)*) – The direction of the axis of rotation. Input can be a vector of shape (3,) or a string 'x', 'y' or 'z' to denote respective directions.
- **anchor** (*None or array_like, shape (3,), default=None, unit [mm]*) – The axis of rotation passes through the anchor point given in units of [mm]. By default (*anchor=None*) the object will rotate about its own center. *anchor=0* rotates the object about the origin (0,0,0).
- **start** (*int or str, default=-1*) – Choose at which index of the original object path, the input path will begin. If *start=-1*, inp_path will start at the last old_path position. If *start=0*, inp_path will start with the beginning of the old_path. If *start=len(old_path)* or *start='append'*, inp_path will be attached to the old_path.
- **increment** (*bool, default=False*) – If *increment=False*, input rotations are absolute. If *increment=True*, input rotations are interpreted as increments of each other. For example, the incremental angles [1,1,1,2,2] correspond to the absolute angles [1,2,3,5,7].
- **degrees** (*bool, default=True*) – By default angle is given in units of [deg]. If degrees=False, angle is given in units of [rad].

Returns self

Return type Magpylib object

Examples

With the `rotate_from_angax` method Magpylib objects can be rotated about their local coordinte system center:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz'))
[0. 0. 0.]
>>> sensor.rotate_from_angax(angle=45, axis='x')
>>> print(sensor.position)
[0. 0. 0.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[45.  0.  0.]
```

With the `anchor` keyword the object rotates about a designated axis that passes through the given anchor point:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis=(1,0,0), anchor=(0,1,0))
>>> print(sensor.position)
[ 0.  1. -1.]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[90.  0.  0.]
```

The method can also be used to generate paths, making use of scipy.Rotation object vector input:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax(angle=90, axis='x', anchor=(0,1,0), start='append
↪')
>>> print(sensor.position)
[[ 0.  0.  0.]
 [ 0.  1. -1.]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [90.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[10,20,30], axis='x', anchor=(0,1,0),␣
↪start='append')
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          1.         -1.        ]
 [ 0.          1.17364818 -0.98480775]
 [ 0.          1.34202014 -0.93969262]
 [ 0.          1.5        -0.8660254 ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[  0.   0.   0.]
 [ 90.   0.   0.]
 [100.   0.   0.]
 [110.   0.   0.]
 [120.   0.   0.]]
```

Complex paths can be generated by making use of the `increment` keyword and the superposition of subsequent paths:

```
>>> import magpylib as mag3
>>> sensor = mag3.Sensor()
>>> sensor.rotate_from_angax([10]*3, 'x', (0,1,0), start=1, increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [ 0.          0.01519225 -0.17364818]
 [ 0.          0.06030738 -0.34202014]
 [ 0.          0.1339746  -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  0.]
 [10.  0.  0.]
 [20.  0.  0.]
 [30.  0.  0.]]
>>> sensor.rotate_from_angax(angle=[5]*4, axis='z', anchor=0, start=0,␣
↪increment=True)
>>> print(sensor.position)
[[ 0.          0.          0.        ]
 [-0.00263811  0.01496144 -0.17364818]
 [-0.0156087   0.05825246 -0.34202014]
 [-0.04582201  0.12589494 -0.5       ]]
>>> print(sensor.orientation.as_euler('xyz', degrees=True))
[[ 0.  0.  5.]
 [10.  0. 10.]
 [20.  0. 15.]
 [30.  0. 20.]]
```

CHAPTER 3

Index

- genindex
- modindex

Python Module Index

m

magpylib, 10
magpylib.current, 39
magpylib.magnet, 59
magpylib.misc, 89

Index

A

add() (*magpylib.Collection method*), 30

B

Box (*class in magpylib.magnet*), 59

C

Circular (*class in magpylib.current*), 39
Collection (*class in magpylib*), 29
Config (*class in magpylib*), 38
copy() (*magpylib.Collection method*), 30
current (*magpylib.current.Circular attribute*), 40
current (*magpylib.current.Line attribute*), 50
Cylinder (*class in magpylib.magnet*), 69

D

diameter (*magpylib.current.Circular attribute*), 40
diameter (*magpylib.magnet.Sphere attribute*), 80
dimension (*magpylib.magnet.Box attribute*), 60
dimension (*magpylib.magnet.Cylinder attribute*), 70
Dipole (*class in magpylib.misc*), 89
display() (*in module magpylib*), 37
display() (*magpylib.Collection method*), 31
display() (*magpylib.current.Circular method*), 40
display() (*magpylib.current.Line method*), 50
display() (*magpylib.magnet.Box method*), 60
display() (*magpylib.magnet.Cylinder method*), 70
display() (*magpylib.magnet.Sphere method*), 80
display() (*magpylib.misc.Dipole method*), 90
display() (*magpylib.Sensor method*), 20

G

getB() (*in module magpylib*), 13
getB() (*magpylib.Collection method*), 32
getB() (*magpylib.current.Circular method*), 41
getB() (*magpylib.current.Line method*), 51
getB() (*magpylib.magnet.Box method*), 61
getB() (*magpylib.magnet.Cylinder method*), 71
getB() (*magpylib.magnet.Sphere method*), 81
getB() (*magpylib.misc.Dipole method*), 91
getB() (*magpylib.Sensor method*), 21
getBv() (*in module magpylib*), 15
getH() (*in module magpylib*), 14
getH() (*magpylib.Collection method*), 33
getH() (*magpylib.current.Circular method*), 42
getH() (*magpylib.current.Line method*), 52
getH() (*magpylib.magnet.Box method*), 62
getH() (*magpylib.magnet.Cylinder method*), 72
getH() (*magpylib.magnet.Sphere method*), 82
getH() (*magpylib.misc.Dipole method*), 92
getH() (*magpylib.Sensor method*), 22
getHv() (*in module magpylib*), 17

L

Line (*class in magpylib.current*), 49

M

magnetization (*magpylib.magnet.Box attribute*), 63
magnetization (*magpylib.magnet.Cylinder attribute*), 73
magnetization (*magpylib.magnet.Sphere attribute*), 83
magpylib (*module*), 10
magpylib.current (*module*), 39
magpylib.magnet (*module*), 59
magpylib.misc (*module*), 89
moment (*magpylib.misc.Dipole attribute*), 93
move() (*magpylib.Collection method*), 34
move() (*magpylib.current.Circular method*), 43
move() (*magpylib.current.Line method*), 53
move() (*magpylib.magnet.Box method*), 63
move() (*magpylib.magnet.Cylinder method*), 73
move() (*magpylib.magnet.Sphere method*), 83
move() (*magpylib.misc.Dipole method*), 93
move() (*magpylib.Sensor method*), 23

O

orientation (*magpylib.current.Circular attribute*), 44

orientation (*magpylib.current.Line attribute*), 54
orientation (*magpylib.magnet.Box attribute*), 64
orientation (*magpylib.magnet.Cylinder attribute*), 74
orientation (*magpylib.magnet.Sphere attribute*), 84
orientation (*magpylib.misc.Dipole attribute*), 94
orientation (*magpylib.Sensor attribute*), 24

P

pixel (*magpylib.Sensor attribute*), 24
position (*magpylib.current.Circular attribute*), 44
position (*magpylib.current.Line attribute*), 54
position (*magpylib.magnet.Box attribute*), 64
position (*magpylib.magnet.Cylinder attribute*), 74
position (*magpylib.magnet.Sphere attribute*), 84
position (*magpylib.misc.Dipole attribute*), 94
position (*magpylib.Sensor attribute*), 24

R

remove() (*magpylib.Collection method*), 34
reset() (*magpylib.Config class method*), 39
reset_path() (*magpylib.Collection method*), 35
reset_path() (*magpylib.current.Circular method*), 45
reset_path() (*magpylib.current.Line method*), 54
reset_path() (*magpylib.magnet.Box method*), 64
reset_path() (*magpylib.magnet.Cylinder method*), 74
reset_path() (*magpylib.magnet.Sphere method*), 84
reset_path() (*magpylib.misc.Dipole method*), 94
reset_path() (*magpylib.Sensor method*), 24
rotate() (*magpylib.Collection method*), 35
rotate() (*magpylib.current.Circular method*), 45
rotate() (*magpylib.current.Line method*), 55
rotate() (*magpylib.magnet.Box method*), 65
rotate() (*magpylib.magnet.Cylinder method*), 75
rotate() (*magpylib.magnet.Sphere method*), 85
rotate() (*magpylib.misc.Dipole method*), 95
rotate() (*magpylib.Sensor method*), 25
rotate_from_angax() (*magpylib.Collection method*), 36
rotate_from_angax() (*magpylib.current.Circular method*), 47
rotate_from_angax() (*magpylib.current.Line method*), 57
rotate_from_angax() (*magpylib.magnet.Box method*), 67
rotate_from_angax() (*magpylib.magnet.Cylinder method*), 77
rotate_from_angax() (*magpylib.magnet.Sphere method*), 87
rotate_from_angax() (*magpylib.misc.Dipole method*), 97
rotate_from_angax() (*magpylib.Sensor method*), 27

S

Sensor (*class in magpylib*), 18
sources (*magpylib.Collection attribute*), 37
Sphere (*class in magpylib.magnet*), 79

V

vertices (*magpylib.current.Line attribute*), 59