probeinterface

Samuel Garcia

Feb 07, 2025

CONTENTS

1 Examples **3**

1.1 Generate a Probe from scratch . . . 3
1.2 2d and 3d Probes . . . 5
1.3 Generate a ProbeGroup . . . 9
1.4 Multi shank probes . . . 11
1.5 Handle channel indices . . . 14
1.6 Import/export functions . . . 18
1.7 Probe generator . . . 22
1.8 More plotting examples . . . 26
1.9 More complicated probes . . . 29
1.10 Get probe from library . . . 33
1.11 Automatic wiring . . . 35
1.12 Plot values . . . 38
1.13 Overview . . . 41
1.14 Examples . . . 43
1.15 Format specifications . . . 43
1.16 Probeinterface public library . . . 50
1.17 API . . . 50
1.18 Release notes . . . 66

Python Module Index **73**

Index **75**

`probeinterface` is a Python package to handle probe layout, geometry and wiring to a device for neuroscience experiments.

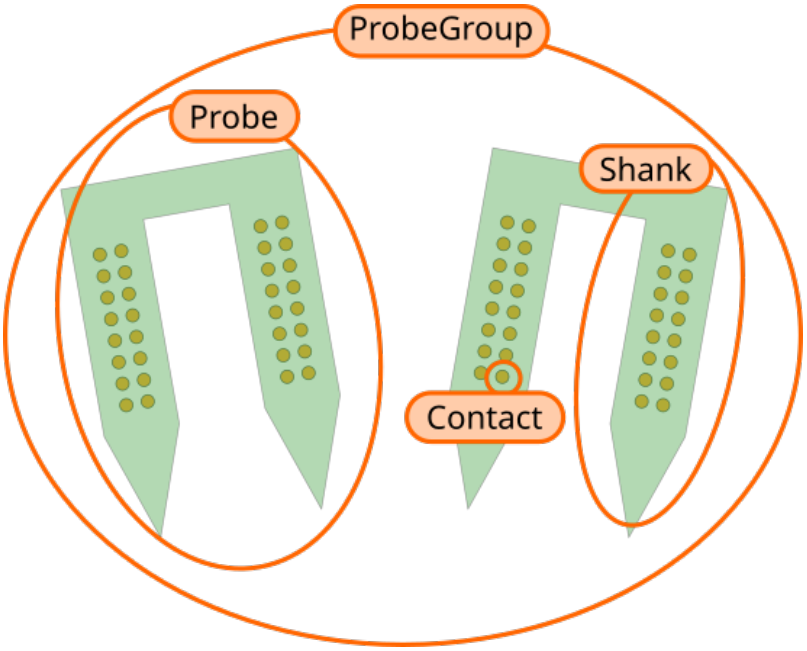
The package handles the following items:

- probe geometry (2D or 3D layout)
- probe shape (contours of the probe)
- shape and size of the shank
- probe wiring to the recording device
- combination of several probes: global geometry + global wiring

The `probeinterface` package also provides:

- basic plotting functions with matplotlib
- input/output functions to several formats (PRB, NWB, CSV, MEArec, SpikeGLX, …)

Here is a schema for the naming used in the package:



orphan

CHAPTER

ONE

EXAMPLES

Start here with a tutorial showing probeinterface.

1.1 Generate a Probe from scratch

This example generates a probe from scratch.

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

from probeinterface import Probe
from probeinterface.plotting import plot_probe
```

First, let's create dummy positions for a 24-contact probe

```
n = 24
positions = np.zeros((n, 2))
for i in range(n):
    x = i // 8
    y = i % 8
    positions[i] = x, y
positions *= 20
positions[8:16, 1] -= 10
```

Now we can create a *Probe* object and set the position and shape of each contact

The *ndim* argument indicates that the contact is 2d, so the positions have a (n_elec, 2) shape. We can also define a 3d probe with *ndim=3* and positions will have a (n_elec, 3) shape.

Note: *shapes* and *shape_params* could be arrays as well, indicating the shape for each contact separately.

```
probe = Probe(ndim=2, si_units='um')
probe.set_contacts(positions=positions, shapes='circle', shape_params={'radius': 5})
```

Probe objects have fancy prints!

```
print(probe)
```

```
Probe - 24ch - 1shanks
```

In addition to contacts, we can create the planar contour (polygon) of the probe

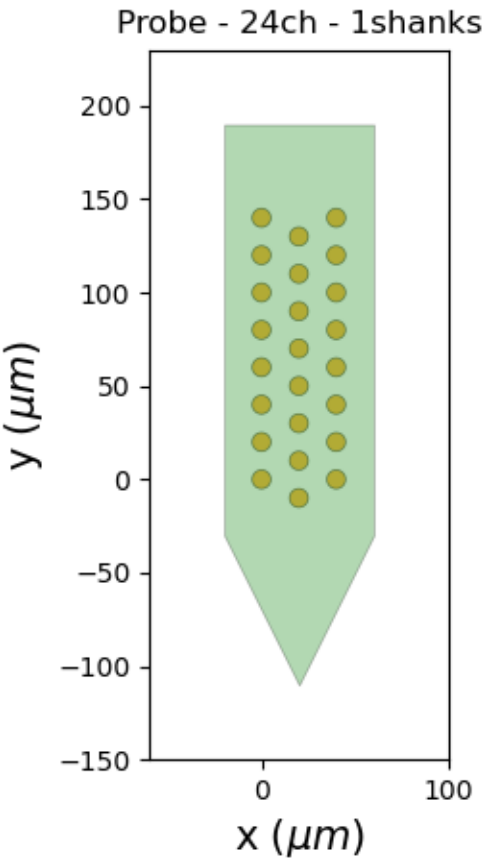
```
polygon = [(-20, -30), (20, -110), (60, -30), (60, 190), (-20, 190)]
probe.set_planar_contour(polygon)
```

If *pandas* is installed, the *Probe* object can be exported as a dataframe for a simpler view:

```
df = probe.to_dataframe()
df
```

If *matplotlib* is installed, the *Probe* can also be easily plotted:

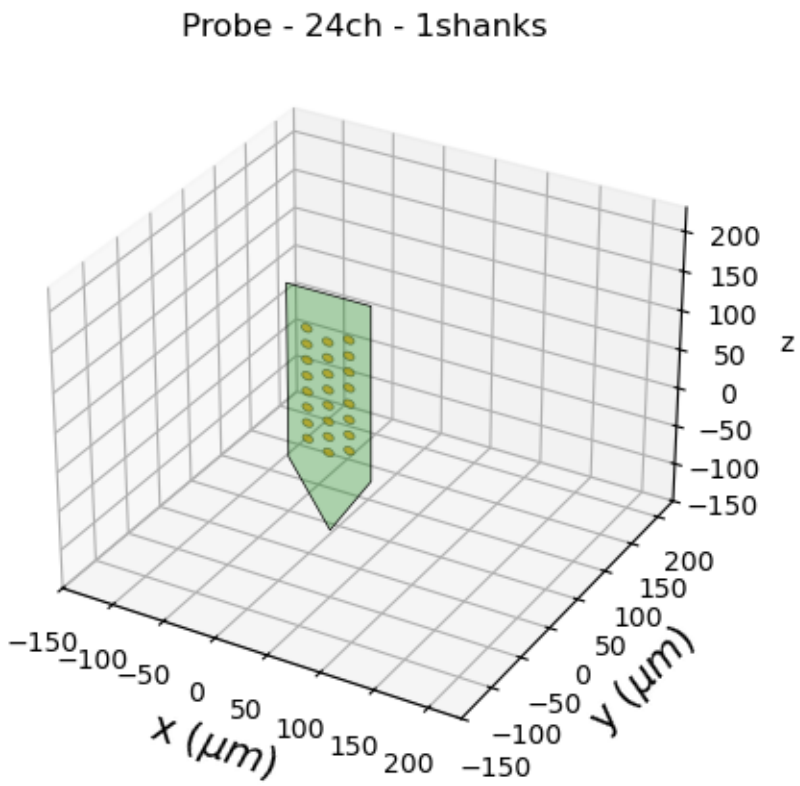
```
plot_probe(probe)
```




```
(<matplotlib.collections.PolyCollection object at 0x7fff56bdc440>, <matplotlib.
↪collections.PolyCollection object at 0x7fff56b42c10>)
```

A 2d *Probe* can be transformed into a 3d *Probe* by indicating the *axes* on which contacts will lie (Here the 'y' coordinate will be 0 for all contacts):

```
probe_3d = probe.to_3d(axes='xz')
plot_probe(probe_3d)

plt.show()
```



Total running time of the script: (0 minutes 0.584 seconds)

1.2 2d and 3d Probes

This example shows how to manipulate a probe in 2d or 3d.

Import

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

from probeinterface import Probe
from probeinterface.plotting import plot_probe
```

First, let's create one 2d probe with 24 contacts:

```
n = 24
positions = np.zeros((n, 2))
for i in range(n):
    x = i // 8
    y = i % 8
    positions[i] = x, y
positions *= 20
positions[8:16, 1] -= 10
```

(continues on next page)

(continued from previous page)

```
probe_2d = Probe(ndim=2, si_units='um')
probe_2d.set_contacts(positions=positions, shapes='circle', shape_params={'radius': 5}
↪)
probe_2d.create_auto_shape(probe_type='tip')
```

Let's transform it into a 3d probe.

Here the axes are 'xz' so y will be 0 for all contacts. The shape of probe_3d.contact_positions is now (n_elec, 3)

```
probe_3d = probe_2d.to_3d(axes='xz')
print(probe_2d.contact_positions.shape)
print(probe_3d.contact_positions.shape)
```

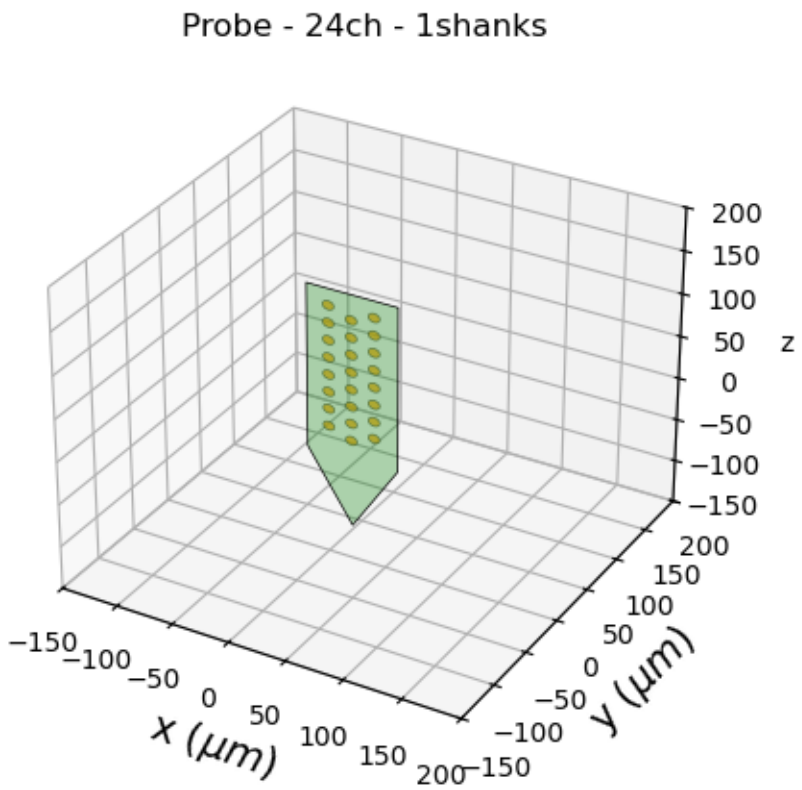
```
(24, 2)
(24, 3)
```

Note that all **"y"** coordinates are 0

```
df = probe_3d.to_dataframe()
df[['x', 'y', 'z']].head()
```

The plotting function automatically displays the *Probe* in 3d:

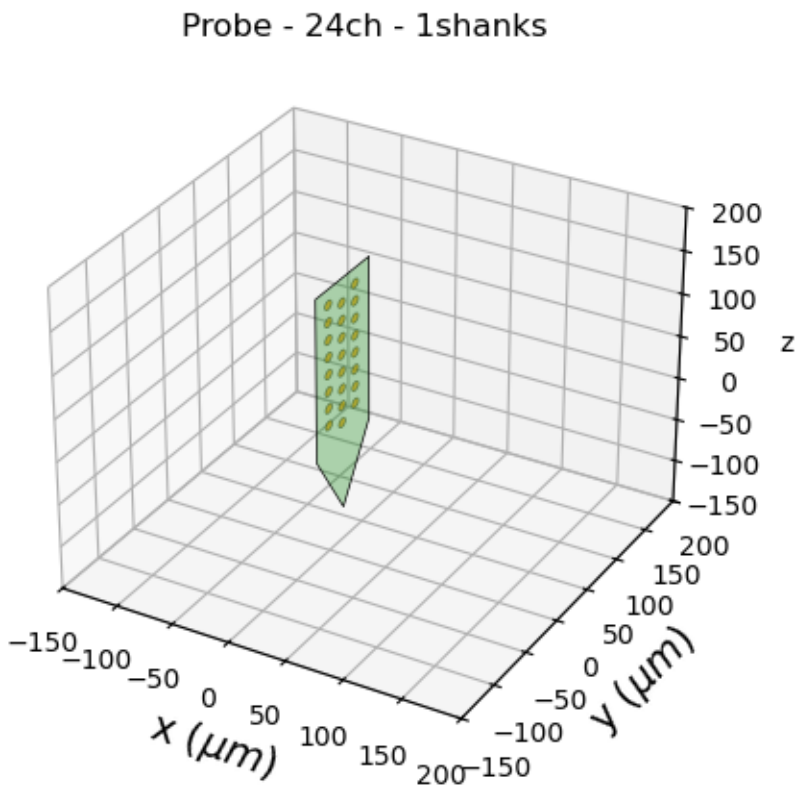
```
plot_probe(probe_3d)
```




```
(<mpl_toolkits.mplot3d.art3d.Poly3DCollection object at 0x7fff56c18050>, <mpl_
↪toolkits.mplot3d.art3d.Poly3DCollection object at 0x7fff56c18410>)
```

We can create another probe lying on another plane:

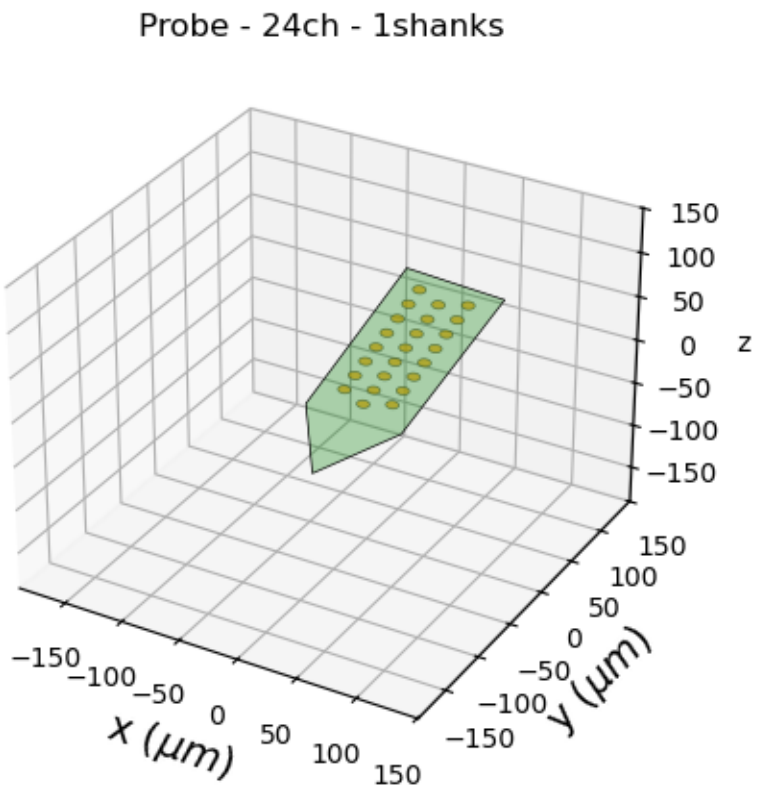
```
other_3d = probe_2d.to_3d(axes='yz')
plot_probe(other_3d)
```




```
(<mpl_toolkits.mplot3d.art3d.Poly3DCollection object at 0x7fff46896710>, <mpl_
↪toolkits.mplot3d.art3d.Poly3DCollection object at 0x7fff468965d0>)
```

Probe can be moved and rotated in 3d:

```
probe_3d.move([0, 30, -50])
probe_3d.rotate(theta=35, center=[0, 0, 0], axis=[0, 1, 1])

plot_probe(probe_3d)

plt.show()
```



Total running time of the script: (0 minutes 0.367 seconds)

1.3 Generate a ProbeGroup

This example shows how to assemble several Probe objects into a ProbeGroup object.

Import

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

from probeinterface import Probe, ProbeGroup
from probeinterface.plotting import plot_probe_group
from probeinterface import generate_dummy_probe
```

Generate 2 dummy *Probe* objects with the utils function:

```
probe0 = generate_dummy_probe(elec_shapes='square')
probe1 = generate_dummy_probe(elec_shapes='circle')
probe1.move([250, -90])
```

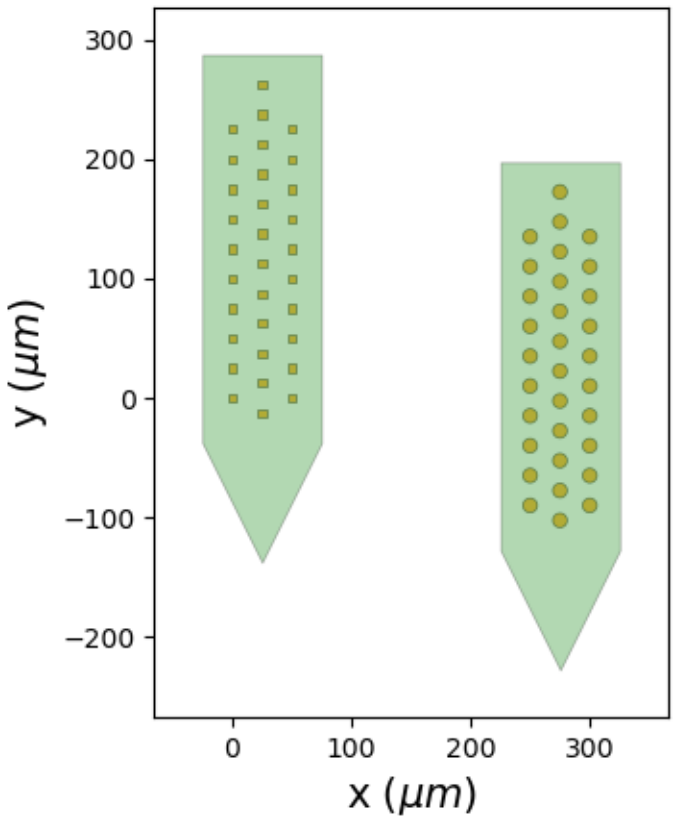
Let's create a *ProbeGroup* and add the *Probe* objects into it:

```
probegroup = ProbeGroup()
probegroup.add_probe(probe0)
probegroup.add_probe(probe1)

print('probe0.get_contact_count()', probe0.get_contact_count())
print('probe1.get_contact_count()', probe1.get_contact_count())
print('probegroup.get_contact_count()', probegroup.get_contact_count())
```

```
probe0.get_contact_count() 32
probe1.get_contact_count() 32
probegroup.get_contact_count() 64
```

We can now plot all probes in the same axis:

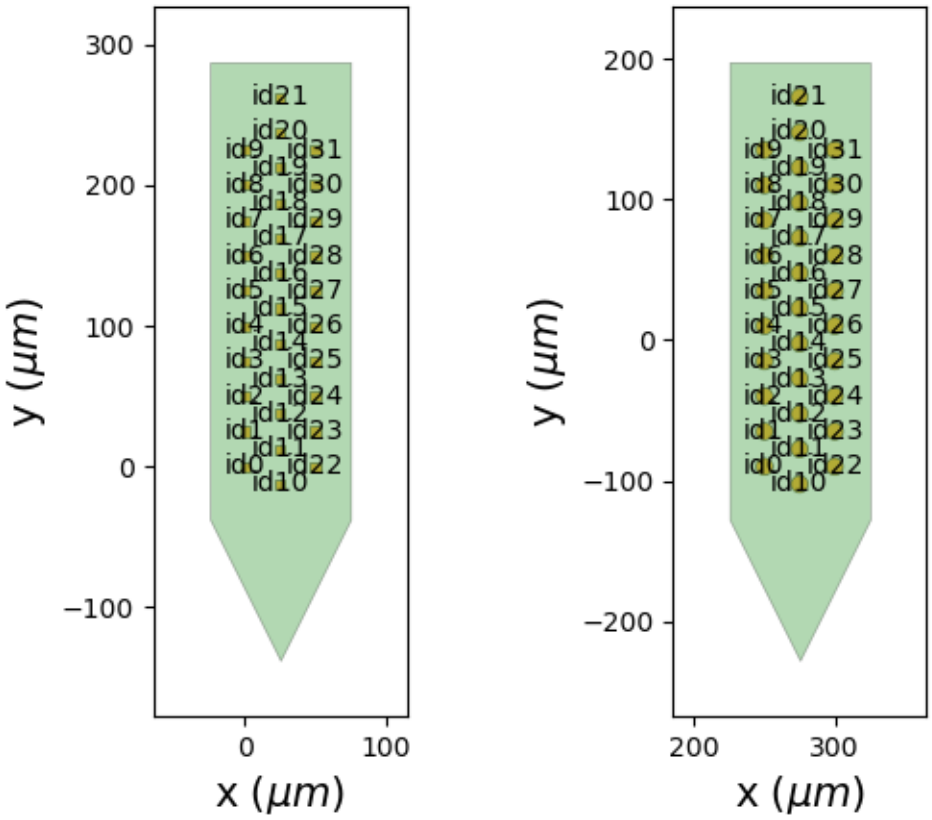
```
plot_probe_group(probegroup, same_axes=True)
```




```
/builddir/build/BUILD/python-probeinterface-0.2.25-build/probeinterface-0.2.25/
↪examples/ex_03_generate_probe_group.py:42: DeprecationWarning: `plot_probe_group`␣
↪is deprecated and will be removed in 2.23. Use plot_probegroup instead
  plot_probe_group(probegroup, same_axes=True)
```

or in separate axes:

```
plot_probe_group(probegroup, same_axes=False, with_contact_id=True)

plt.show()
```




```
/builddir/build/BUILD/python-probeinterface-0.2.25-build/probeinterface-0.2.25/
↪examples/ex_03_generate_probe_group.py:47: DeprecationWarning: `plot_probe_group`␣
↪is deprecated and will be removed in 2.23. Use plot_probegroup instead
  plot_probe_group(probegroup, same_axes=False, with_contact_id=True)
```

Total running time of the script: (0 minutes 0.252 seconds)

1.4 Multi shank probes

This example shows how to deal with multi-shank probes.

In *probeinterface* this can be done with a *Probe* object, but internally each probe handles a *shank_ids* vector to carry information about which contacts belong to which shanks.

Optionally, a `Probe` object can be rendered split into `Shank`.

Import

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

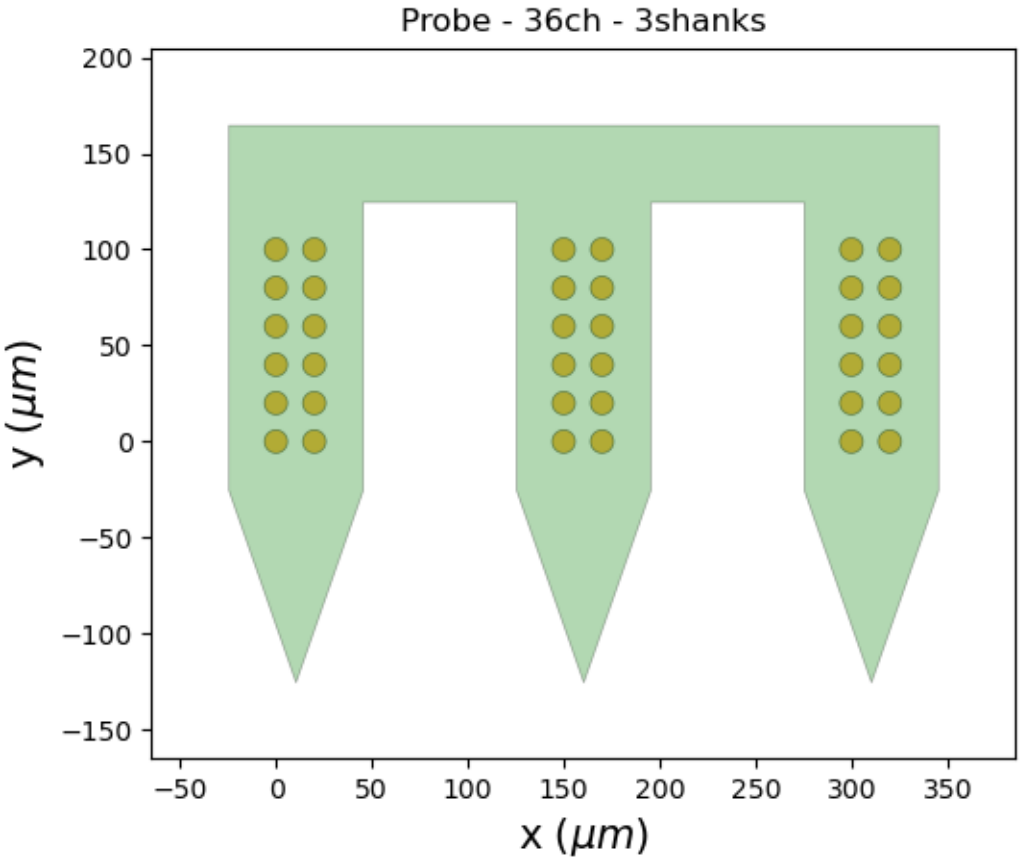
(continues on next page)

(continued from previous page)

```
from probeinterface import Probe, ProbeGroup
from probeinterface import generate_linear_probe, generate_multi_shank
from probeinterface import combine_probes
from probeinterface.plotting import plot_probe
```

Let's use a generator to create a multi-shank probe:

```
multi_shank = generate_multi_shank(num_shank=3, num_columns=2, num_contact_per_
↪column=6)
plot_probe(multi_shank)
```




```
(<matplotlib.collections.PolyCollection object at 0x7fff46972ad0>, <matplotlib.
↪collections.PolyCollection object at 0x7fff469711d0>)
```

multi_shank is one *probe* object, but internally the *Probe.shank_ids* vector handles the shank ids.

```
print(multi_shank.shank_ids)
```

```
['0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '1' '1' '1' '1' '1' '1'
 '1' '1' '1' '1' '1' '1' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2']
```

The dataframe displays the *shank_ids* column:

```
df = multi_shank.to_dataframe()
df
```

We can iterate over a multi-shank probe and get `Shank` objects. A *Shank* is linked to a *Probe* object and can also retrieve positions, contact shapes, etc.:

```
for i, shank in enumerate(multi_shank.get_shanks()):
    print('shank', i)
    print(shank.__class__)
    print(shank.get_contact_count())
    print(shank.contact_positions.shape)
```

```
shank 0
<class 'probeinterface.shank.Shank'>
12
(12, 2)
shank 1
<class 'probeinterface.shank.Shank'>
12
(12, 2)
shank 2
<class 'probeinterface.shank.Shank'>
12
(12, 2)
```

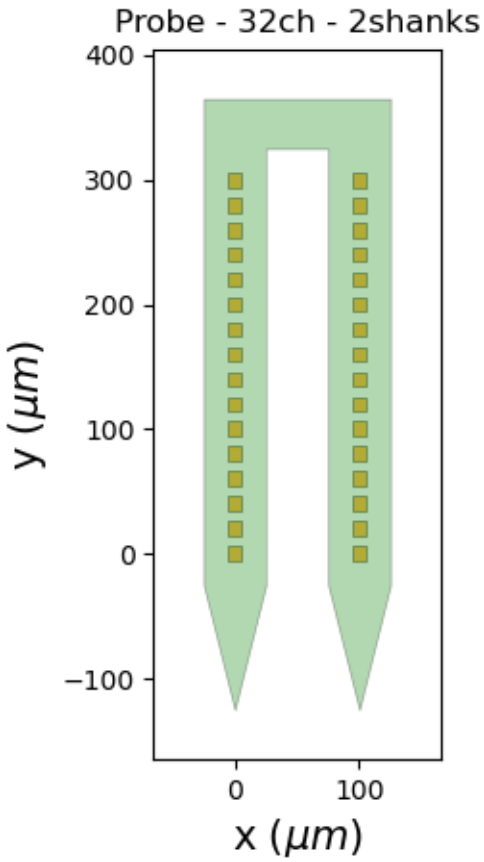
Another option to create multi-shank probes is to create several *Shank* objects as separate probes and then combine them into a single *Probe* object

```
# generate a 2 shanks linear
probe0 = generate_linear_probe(num_elec=16, ypitch=20,
                               contact_shapes='square',
                               contact_shape_params={'width': 12})
probe1 = probe0.copy()
probe1.move([100, 0])

multi_shank = combine_probes([probe0, probe1])
```

```
print(multi_shank.shank_ids)
```

```
['0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '1' '1'
 '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1']
```

```
plot_probe(multi_shank)

plt.show()
```



Total running time of the script: (0 minutes 0.189 seconds)

1.5 Handle channel indices

Probes can have a complex contacts indexing system due to the probe layout. When they are plugged into a recording device like an Open Ephys with an Intan headstage, the channel order can be mixed again. So the physical contact channel index is rarely the channel index on the device.

This is why the *Probe* object can handle separate *device_channel_indices*.

Import

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

from probeinterface import Probe, ProbeGroup
from probeinterface.plotting import plot_probe, plot_probe_group
from probeinterface import generate_multi_columns_probe
```

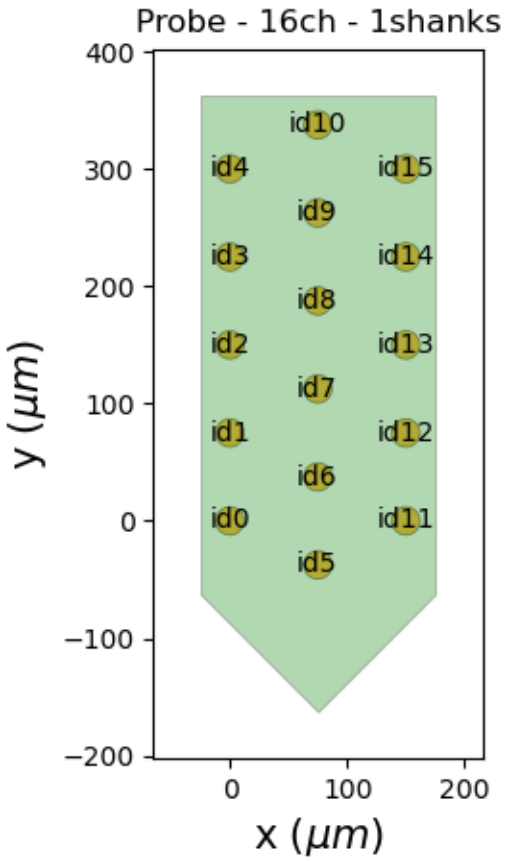
Let's first generate a probe. By default, the wiring is not complicated: each column increments the contact index from the bottom to the top of the probe:

```
probe = generate_multi_columns_probe(num_columns=3,
                                     num_contact_per_column=[5, 6, 5],
```

(continues on next page)

(continued from previous page)

```
                                        xpitch=75, ypitch=75, y_shift_per_column=[0, -37.
↪5, 0],
                                        contact_shapes='circle', contact_shape_params={
↪'radius': 12})

plot_probe(probe, with_contact_id=True)
```




```
(<matplotlib.collections.PolyCollection object at 0x7fff56bd8690>, <matplotlib.
↪collections.PolyCollection object at 0x7fff56bd8550>)
```

The Probe is not connected to any device yet:

```
print(probe.device_channel_indices)
```

```
None
```

Let's imagine we have a headstage with the following wiring: the first half of the channels have natural indices, but the order of the other half is reversed:

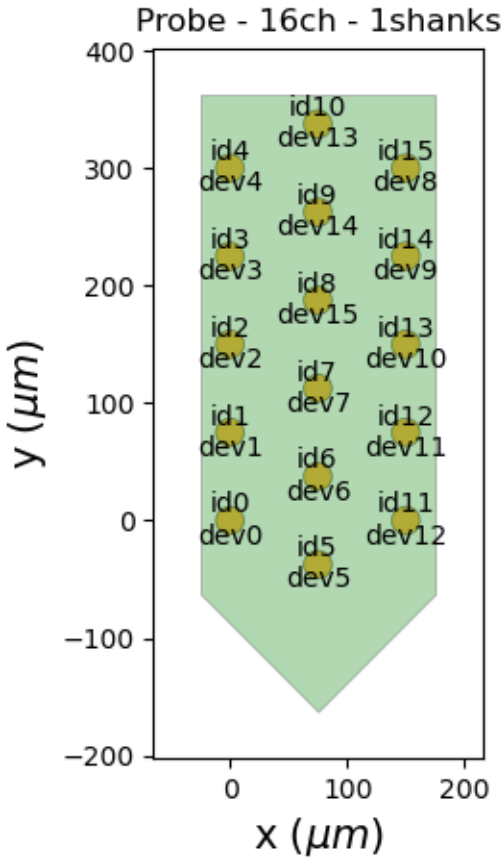
```
channel_indices = np.arange(16)
channel_indices[8:16] = channel_indices[8:16][::-1]
probe.set_device_channel_indices(channel_indices)
print(probe.device_channel_indices)
```

```
[ 0  1  2  3  4  5  6  7 15 14 13 12 11 10  9  8]
```

We can visualize the two sets of indices:

- the prbXX is the contact index ordered from 0 to N
- the devXX is the channel index on the device (with the second half reversed)

```
plot_probe(probe, with_contact_id=True, with_device_index=True)
```




```
(<matplotlib.collections.PolyCollection object at 0x7fff56c1bc50>, <matplotlib.
↪collections.PolyCollection object at 0x7fff56c187d0>)
```

Very often we have several probes on the device and this can lead to even more complex channel indices. *ProbeGroup.get_global_device_channel_indices()* gives an overview of the device wiring.

```
probe0 = generate_multi_columns_probe(num_columns=3,
                                      num_contact_per_column=[5, 6, 5],
                                      xpitch=75, ypitch=75, y_shift_per_column=[0, -
↪37.5, 0],
                                      contact_shapes='circle', contact_shape_params={
↪'radius': 12})
probe1 = probe0.copy()
```

(continues on next page)

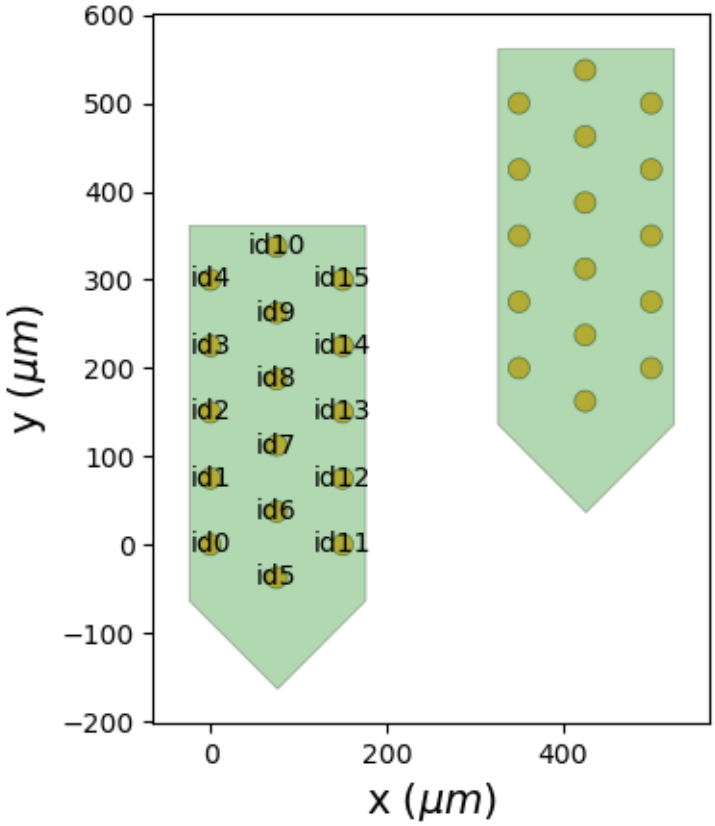
(continued from previous page)

```
probe1.move([350, 200])
probegroup = ProbeGroup()
probegroup.add_probe(probe0)
probegroup.add_probe(probe1)

# wire probe0 0 to 31 and shuffle
channel_indices0 = np.arange(16)
np.random.shuffle(channel_indices0)
probe0.set_device_channel_indices(channel_indices0)

# wire probe0 32 to 63 and shuffle
channel_indices1 = np.arange(16, 32)
np.random.shuffle(channel_indices1)
probe1.set_device_channel_indices(channel_indices1)

print(probegroup.get_global_device_channel_indices())
```

```
[(0,  3) (0,  5) (0,  8) (0,  7) (0,  4) (0,  0) (0, 13) (0, 11) (0, 10)
 (0,  2) (0,  6) (0,  1) (0, 14) (0,  9) (0, 15) (0, 12) (1, 31) (1, 29)
 (1, 16) (1, 27) (1, 18) (1, 22) (1, 30) (1, 21) (1, 17) (1, 20) (1, 25)
 (1, 24) (1, 26) (1, 23) (1, 19) (1, 28)]
```

The indices of the probe group can also be plotted:

```
fig, ax = plt.subplots()
plot_probe_group(probegroup, with_contact_id=True, same_axes=True, ax=ax)

plt.show()
```




```
/builddir/build/BUILD/python-probeinterface-0.2.25-build/probeinterface-0.2.25/
↪examples/ex_05_device_channel_indices.py:88: DeprecationWarning: `plot_probe_group`␣
↪is deprecated and will be removed in 2.23. Use plot_probegroup instead
  plot_probe_group(probegroup, with_contact_id=True, same_axes=True, ax=ax)
```

Total running time of the script: (0 minutes 0.311 seconds)

1.6 Import/export functions

probeinterface has its own format based on JSON. The format can handle several probes in one file. It has a 'probes' key that can contain a list of probes.

Each probe field in the json format contains the *Probe* class attributes.

It also supports reading (and sometimes writing) from these formats:

- PRB (.prb) : used by klusta/spyking-circus/tridesclous
- CSV (.csv): 2 or 3 columns locations in text file
- mearec (.h5) : mearec handles the geometry
- spikeglx (.meta) : spikeglx also handles the geometry

Import

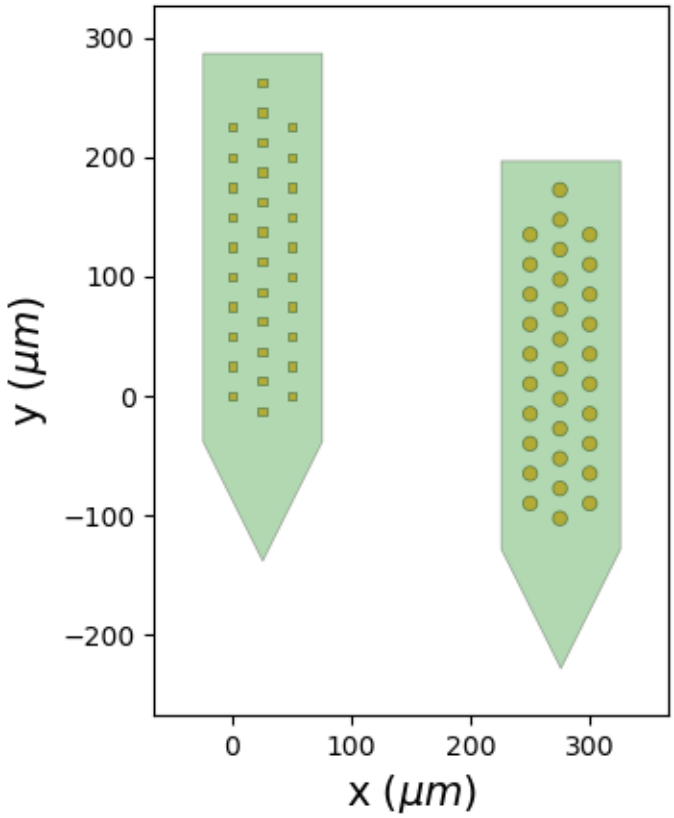
```
import numpy as np
import matplotlib.pyplot as plt

from probeinterface import Probe, ProbeGroup
from probeinterface.plotting import plot_probe, plot_probe_group
from probeinterface import generate_dummy_probe
from probeinterface import write_probeinterface, read_probeinterface
from probeinterface import write_prb, read_prb
```

Let's first generate 2 dummy probes and combine them into a ProbeGroup

```
probe0 = generate_dummy_probe(elec_shapes='square')
probe1 = generate_dummy_probe(elec_shapes='circle')
probe1.move([250, -90])

probegroup = ProbeGroup()
probegroup.add_probe(probe0)
probegroup.add_probe(probe1)
```

With the *write_probeinterface* and *read_probeinterface* functions we can write to and read from the json-based probeinterface format:

```
write_probeinterface('my_two_probe_setup.json', probegroup)

probegroup2 = read_probeinterface('my_two_probe_setup.json')
plot_probe_group(probegroup2)
```




```
/builddir/build/BUILD/python-probeinterface-0.2.25-build/probeinterface-0.2.25/
↪examples/ex_06_import_export_to_file.py:51: DeprecationWarning: `plot_probe_group`␣
↪is deprecated and will be removed in 2.23. Use plot_probegroup instead
  plot_probe_group(probegroup2)
```

The format looks like this:

```
with open('my_two_probe_setup.json', mode='r') as f:
    txt = f.read()

print(txt[:600], '...')
```

```
{
    "specification": "probeinterface",
    "version": "0.2.25",
    "probes": [
        {
            "ndim": 2,
            "si_units": "um",
            "annotations": {
                "manufacturer": "me"
            },
            "contact_annotations": {
```

(continues on next page)

(continued from previous page)

```
                "quality": [
                    1000.0,
                    1000.0,
                    1000.0,
                    1000.0,
                    1000.0,
                    1000.0,
                    1000.0,
                    1000.0,
                    1000.0,
                    1000.0,
                    1000.0,
 ...
```

PRB is an historical format introduced by the Klusta team and it is also used by SpikeInterface, Spyking-circus, and Tridesclous. The format is in fact a python script that describes a dictionary. This format handles:

- multiple groups (multi-shank or multi-probe)
- contact_positions with 'geometry'
- device_channel_indices with 'channels'

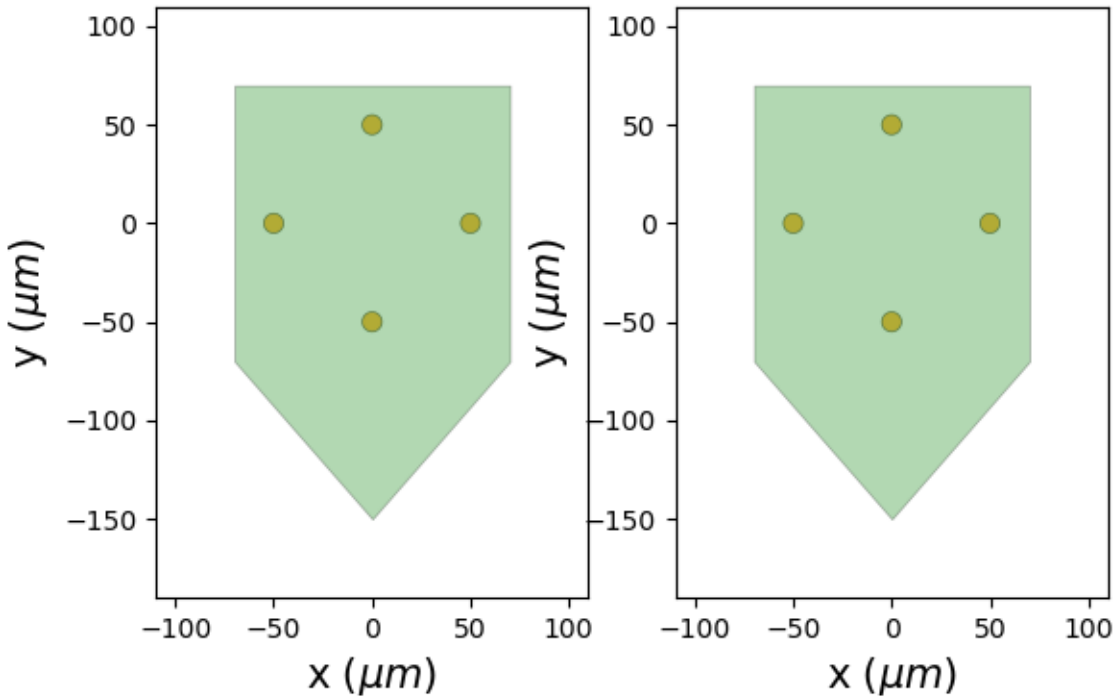
Here is an example of a .prb file with 2 channel groups of 4 channels each. It can be easily loaded and plotted with *probeinterface*

```
prb_two_tetrodes = """
channel_groups = {
    0: {
            'channels' : [0,1,2,3],
            'geometry': {
                0: [0, 50],
                1: [50, 0],
                2: [0, -50],
                3: [-50, 0],
            }
    },
    1: {
            'channels' : [4,5,6,7],
            'geometry': {
                4: [0, 50],
                5: [50, 0],
                6: [0, -50],
                7: [-50, 0],
            }
    }
}
"""

with open('two_tetrodes.prb', 'w') as f:
    f.write(prb_two_tetrodes)

two_tetrode = read_prb('two_tetrodes.prb')
plot_probe_group(two_tetrode, same_axes=False, with_contact_id=True)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




```
/builddir/build/BUILD/python-probeinterface-0.2.25-build/probeinterface-0.2.25/
↪examples/ex_06_import_export_to_file.py:101: DeprecationWarning: `plot_probe_group`␣
↪is deprecated and will be removed in 2.23. Use plot_probegroup instead
  plot_probe_group(two_tetrode, same_axes=False, with_contact_id=True)
```

Total running time of the script: (0 minutes 0.201 seconds)

1.7 Probe generator

probeinterface have also basic function to generate simple contact layouts like:

- tetrodes
- linear probes
- multi-column probes

Import

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

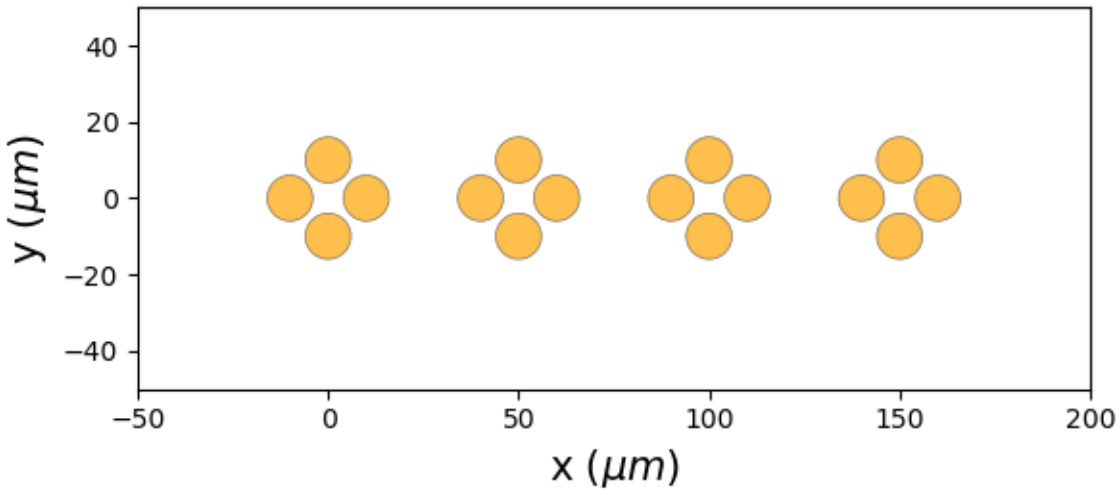
(continues on next page)

(continued from previous page)

```
from probeinterface import Probe, ProbeGroup
from probeinterface.plotting import plot_probe, plot_probe_group
```

Generate 4 tetrodes:

```
from probeinterface import generate_tetrode

probegroup = ProbeGroup()
for i in range(4):
    tetrode = generate_tetrode()
    tetrode.move([i * 50, 0])
    probegroup.add_probe(tetrode)
probegroup.set_global_device_channel_indices(np.arange(16))

df = probegroup.to_dataframe()
df

plot_probe_group(probegroup, with_contact_id=True, same_axes=True)
```




```
/builddir/build/BUILD/python-probeinterface-0.2.25-build/probeinterface-0.2.25/
↪examples/ex_07_probe_generator.py:38: DeprecationWarning: `plot_probe_group` is␣
```

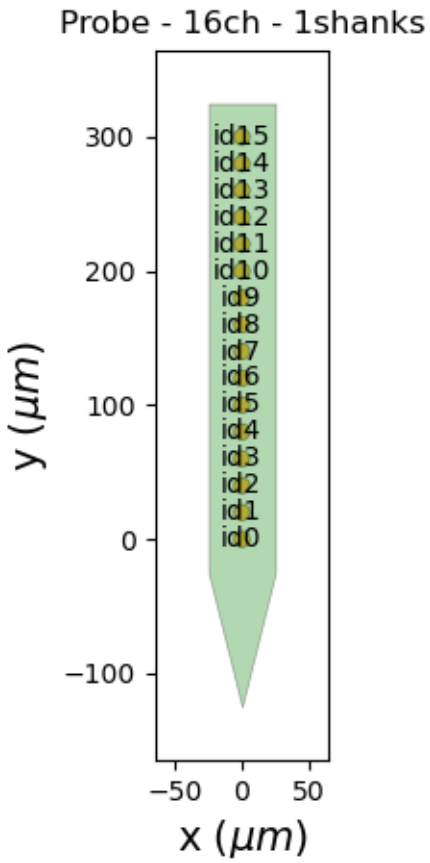
(continues on next page)

(continued from previous page)

```
↪deprecated and will be removed in 2.23. Use plot_probegroup instead
  plot_probe_group(probegroup, with_contact_id=True, same_axes=True)
```

Generate a linear probe:

```
from probeinterface import generate_linear_probe

linear_probe = generate_linear_probe(num_elec=16, ypitch=20)
plot_probe(linear_probe, with_contact_id=True)
```




```
(<matplotlib.collections.PolyCollection object at 0x7fff56c19bd0>, <matplotlib.
↪collections.PolyCollection object at 0x7fff56c19d10>)
```

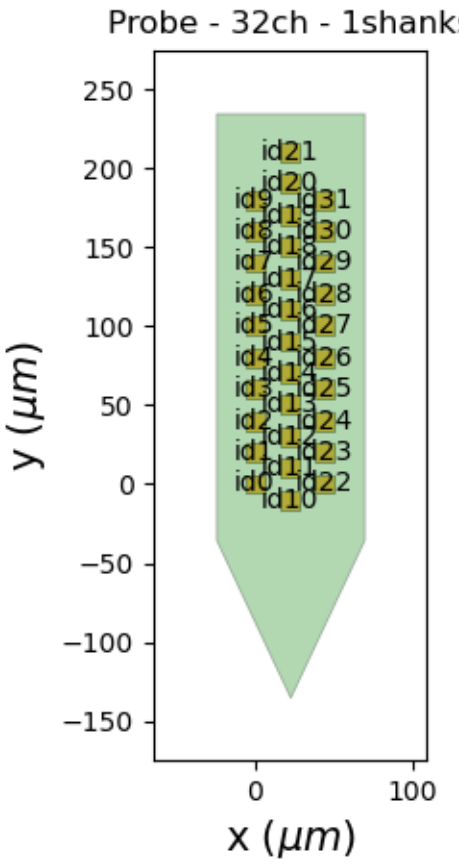
Generate a multi-column probe:

```
from probeinterface import generate_multi_columns_probe

multi_columns = generate_multi_columns_probe(num_columns=3,
                                             num_contact_per_column=[10, 12, 10],
                                             xpitch=22, ypitch=20,
                                             y_shift_per_column=[0, -10, 0],
                                             contact_shapes='square', contact_shape_
↪params={'width': 12})
```

(continues on next page)

(continued from previous page)

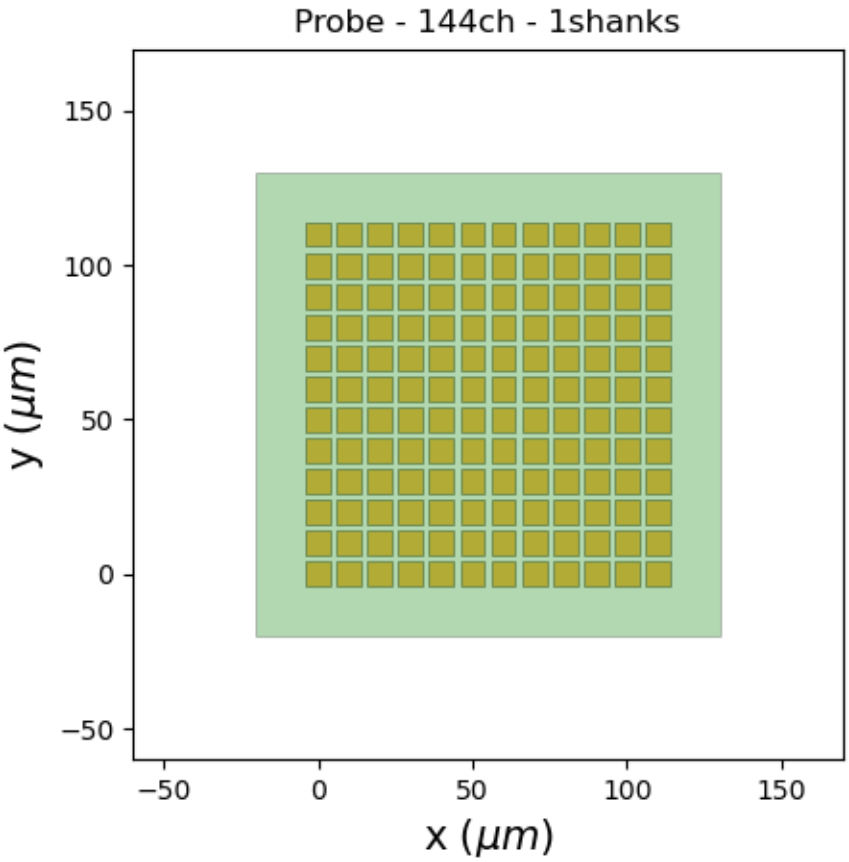
```
plot_probe(multi_columns, with_contact_id=True, )
```




```
(<matplotlib.collections.PolyCollection object at 0x7fff46829bd0>, <matplotlib.
↪collections.PolyCollection object at 0x7fff4682a210>)
```

Generate a square probe:

```
square_probe = generate_multi_columns_probe(num_columns=12,
                                            num_contact_per_column=12,
                                            xpitch=10, ypitch=10,
                                            contact_shapes='square', contact_shape_
↪params={'width': 8})
square_probe.create_auto_shape('rect')
plot_probe(square_probe)

plt.show()
```



Total running time of the script: (0 minutes 0.377 seconds)

1.8 More plotting examples

Here are some examples to showcase several plotting scenarios.

Import

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

from probeinterface import Probe, ProbeGroup
from probeinterface.plotting import plot_probe, plot_probe_group
from probeinterface import generate_multi_columns_probe, generate_linear_probe
```

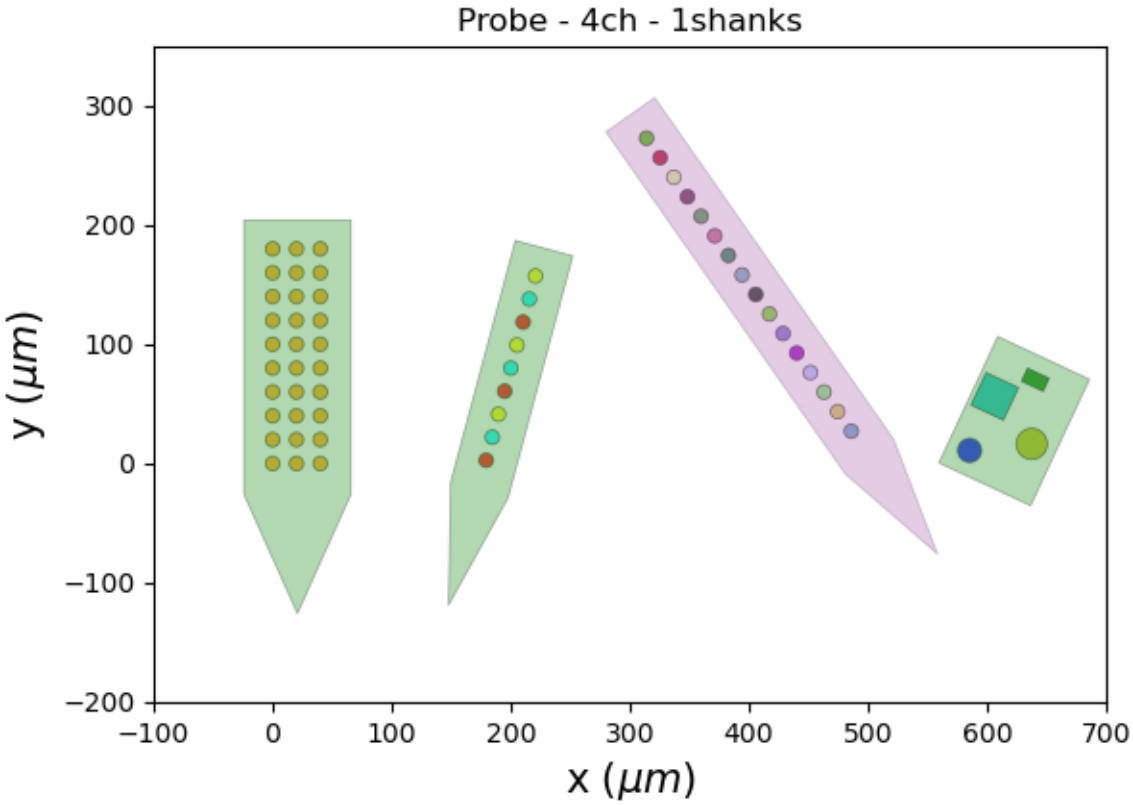
Some examples in 2d

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

probe0 = generate_multi_columns_probe()
plot_probe(probe0, ax=ax)

# give each probe a different color
probe1 = generate_linear_probe(num_elec=9)
```

(continues on next page)

(continued from previous page)

```
probe1.rotate(theta=15)
probe1.move([200, 0])
plot_probe(probe1, ax=ax,
           contacts_colors=['red', 'cyan', 'yellow'] * 3)

# prepare yourself for carnival!
probe2 = generate_linear_probe()
probe2.rotate(theta=-35)
probe2.move([400, 0])
n = probe2.get_contact_count()
rand_colors = np.random.rand(n, 3)
plot_probe(probe2, ax=ax, contacts_colors=rand_colors,
           probe_shape_kwargs={'facecolor': 'purple', 'edgecolor': 'k', 'lw': 0.5,
↪'alpha': 0.2})

# and make some alien probes
probe3 = Probe()
positions = [[0, 0], [0, 50], [25, 77], [45, 27]]
shapes = ['circle', 'square', 'rect', 'circle']
params = [{'radius': 10}, {'width': 30}, {'width': 20, 'height': 12}, {'radius': 13}]
probe3.set_contacts(positions=positions, shapes=shapes,
                    shape_params=params)
probe3.create_auto_shape(probe_type='rect')
probe3.rotate(theta=25)
probe3.move([600, 0])
plot_probe(probe3, ax=ax, contacts_colors=['b', 'c', 'g', 'y'])

ax.set_xlim(-100, 700)
ax.set_ylim(-200, 350)

ax.set_aspect('equal')
```



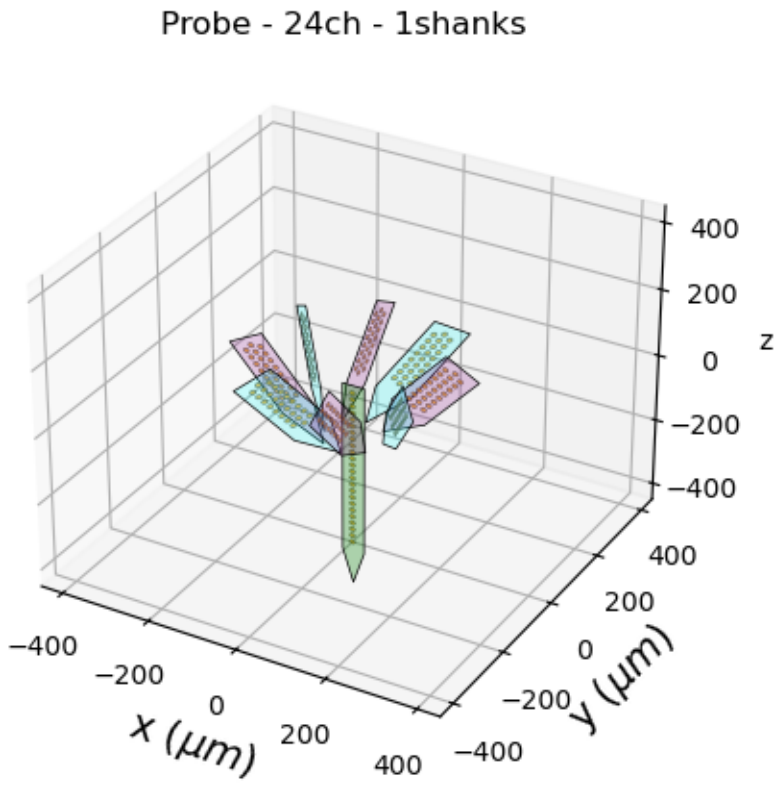
Some examples in 3d for the romantic who likes flowers…

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection='3d')

n = 8
for i in range(n):
    probe = generate_multi_columns_probe(num_columns=3,
                                         num_contact_per_column=[8, 9, 8],
                                         xpitch=20, ypitch=20,
                                         y_shift_per_column=[0, -10, 0]).to_3d()
    probe.rotate(theta=35, center=[0, 0, 0], axis=[0, 1, 0])
    probe.move([100, 50, 0])
    probe.rotate(theta=i * 360 / n, center=[0, 0, 0], axis=[0, 0, 1])
    plot_probe(probe, ax=ax,
               probe_shape_kwargs={'facecolor': ['purple', 'cyan'][i % 2], 'edgecolor
↪': 'k', 'lw': 0.5, 'alpha': 0.2})

probe = generate_linear_probe(num_elec=24, ypitch=20).to_3d()

probe.move([0, 0, -450])
plot_probe(probe, ax=ax)

lims = -450, 450
```

(continues on next page)

(continued from previous page)

```
ax.set_xlim(*lims)
ax.set_ylim(*lims)
ax.set_zlim(*lims)

plt.show()
```



Total running time of the script: (0 minutes 0.354 seconds)

1.9 More complicated probes

This example demonstrates how to generate a more complicated probe with hybrid contacts shape and contact rotations within the *contact_plane_axes* attribute.

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

from probeinterface import Probe
from probeinterface.plotting import plot_probe
```

Let's first set the positions of the contacts

```
n = 24
positions = np.zeros((n, 2))
for i in range(3):
    positions[i * 8: (i + 1) * 8, 0] = i * 30
    positions[i * 8: (i + 1) * 8, 1] = np.arange(0, 240, 30)
```

Electrode shapes can be arrays to handle hybrid shape contacts

```
shapes = np.array(['circle', 'square'] * 12)
shape_params = np.array([{'radius': 8}, {'width': 12}] * 12)
```

The *plane_axes* argument handles the axis for each contact.

It can be used for contact-wise rotations.

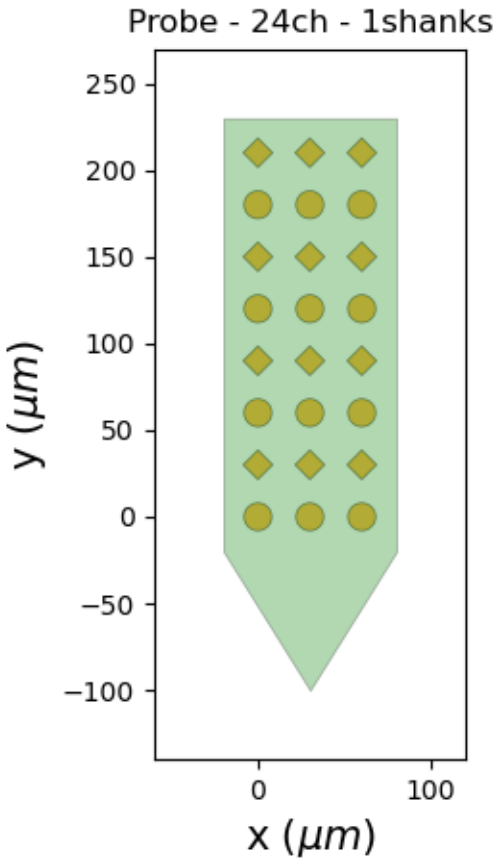
plane_axes has a shape of (num_elec, 2, ndim)

```
plane_axes = [[[1 / np.sqrt(2), 1 / np.sqrt(2)], [-1 / np.sqrt(2), 1 / np.sqrt(2)]]␣
↪* n
plane_axes = np.array(plane_axes)
```

Create the probe

```
probe = Probe(ndim=2, si_units='um')
probe.set_contacts(positions=positions, plane_axes=plane_axes,
                   shapes=shapes, shape_params=shape_params)
probe.create_auto_shape()
```

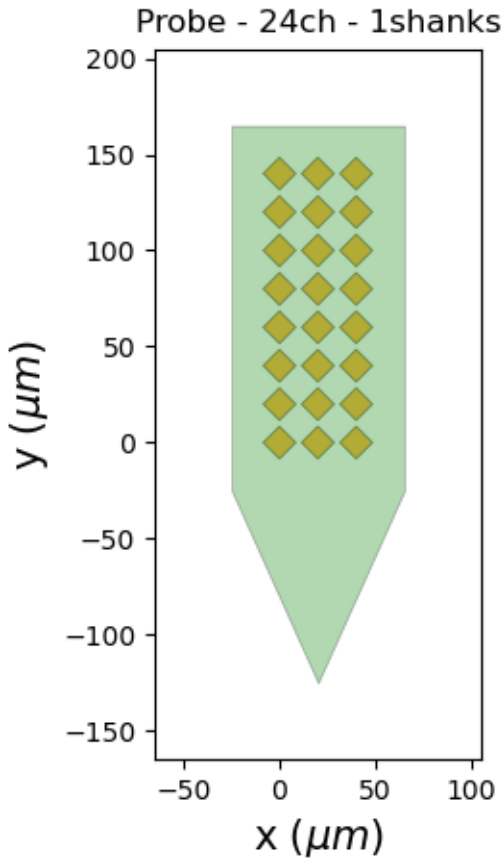
```
plot_probe(probe)
```




```
(<matplotlib.collections.PolyCollection object at 0x7fff46843890>, <matplotlib.
↪collections.PolyCollection object at 0x7fff468439d0>)
```

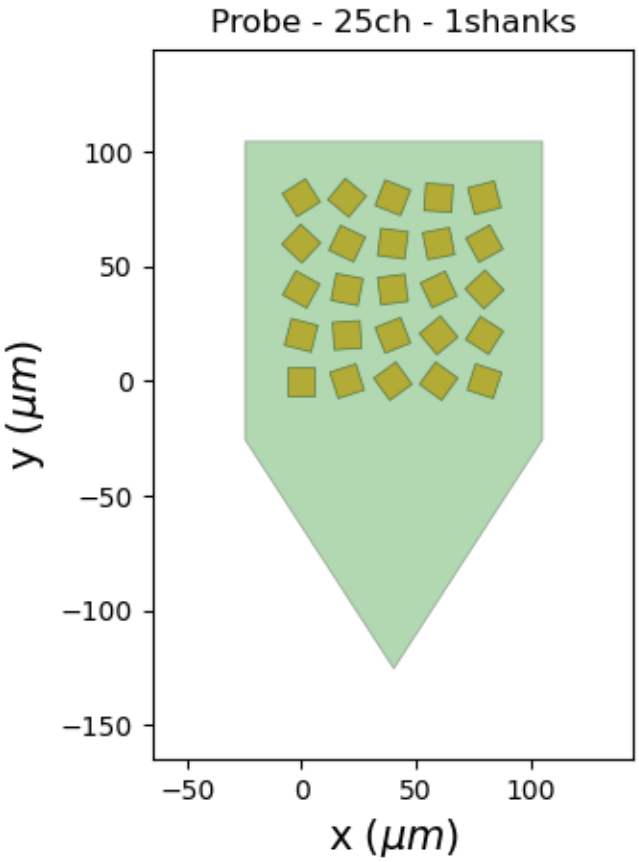
We can also use the *rotate_contacts* to make contact-wise rotations:

```
from probeinterface import generate_multi_columns_probe

probe = generate_multi_columns_probe(num_columns=3,
                                     num_contact_per_column=8, xpitch=20, ypitch=20,
                                     contact_shapes='square', contact_shape_params={
↪'width': 12})
probe.rotate_contacts(45)
plot_probe(probe)
```




```
(<matplotlib.collections.PolyCollection object at 0x7fff56bda990>, <matplotlib.
↪collections.PolyCollection object at 0x7fff56bd8f50>)
```

```
probe = generate_multi_columns_probe(num_columns=5,
                                     num_contact_per_column=5, xpitch=20, ypitch=20,
                                     contact_shapes='square', contact_shape_params={
↪'width': 12})
thetas = np.arange(25) * 360 / 25
probe.rotate_contacts(thetas)
plot_probe(probe)

plt.show()
```



Total running time of the script: (0 minutes 0.254 seconds)

1.10 Get probe from library

probeinterface provides a library of probes from several manufacturers on the GitHub platform: https://github.com/SpikeInterface/probeinterface_library

Users and manufacturers are welcome to contribute to it.

The Python module provide a function to download and cache files locally in the *probeinterface* json-based format.

```
from pprint import pprint

import numpy as np
import matplotlib.pyplot as plt

from probeinterface import Probe, get_probe
from probeinterface.plotting import plot_probe
```

Download one probe:

```
manufacturer = 'neuronexus'
probe_name = 'A1x32-Poly3-10mm-50-177'
```

(continues on next page)

(continued from previous page)

```
probe = get_probe(manufacturer, probe_name)
print(probe)
```

```
Probe - neuronexus - A1x32-Poly3-10mm-50-177 - 32ch - 1shanks
```

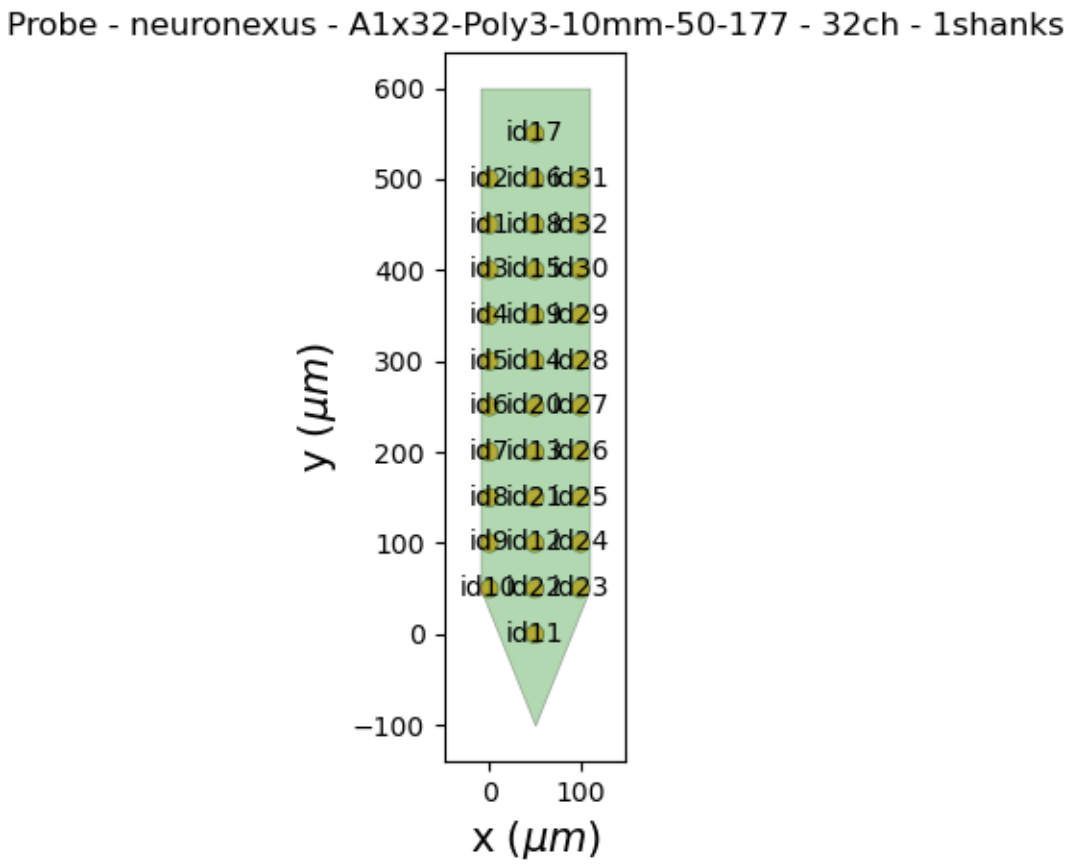
Files from the library also contain annotations specific to manufacturers. We can see here that Neuronexus probes have contact indices starting at "1" (one-based)

```
pprint(probe.annotations)
```

```
{'first_index': 1,
 'manufacturer': 'neuronexus',
 'model_name': 'A1x32-Poly3-10mm-50-177'}
```

When plotting, the channel indices are automatically displayed with one-based notation (even if internally everything is still zero based):

```
plot_probe(probe, with_contact_id=True)
```




```
(<matplotlib.collections.PolyCollection object at 0x7fff5de525d0>, <matplotlib.
↪collections.PolyCollection object at 0x7fff5de52710>)
```

```
plt.show()
```

Total running time of the script: (0 minutes 0.129 seconds)

1.11 Automatic wiring

Here is an example on how to handle the wiring automatically and to get the device_channel_indices.

```
from pprint import pprint

import numpy as np
import matplotlib.pyplot as plt

from probeinterface import Probe, get_probe
from probeinterface.plotting import plot_probe
```

Download one probe:

```
manufacturer = 'neuronexus'
probe_name = 'A1x32-Poly3-10mm-50-177'

probe = get_probe(manufacturer, probe_name)
print(probe)
```

```
Probe - neuronexus - A1x32-Poly3-10mm-50-177 - 32ch - 1shanks
```

We can "wire" this probe to a recording device. Imagine we connect this Neuronexus probe with an Omnetic to an Intan RHD headstage.

Using the wiring documentation from these two sites: https://www.neuronexus.com/files/wiringconfiguration/Wiring_H32.pdf http://intantech.com/RHD_headstages.html?tabSelect=RHD32ch&yPos=0

After a long headache we can figure out the wiring to the device manually and set it using the *probe.set_device_channel_indices()* function:

```
device_channel_indices = [
    16, 17, 18, 20, 21, 22, 31, 30, 29, 27, 26, 25, 24, 28, 23, 19,
    12, 8, 3, 7, 6, 5, 4, 2, 1, 0, 9, 10, 11, 13, 14, 15]
probe.set_device_channel_indices(device_channel_indices)
```

In order to ease this process, *probeinterface* also includes some commonly used wirings based on standard connectors. In our case, we can simply use:

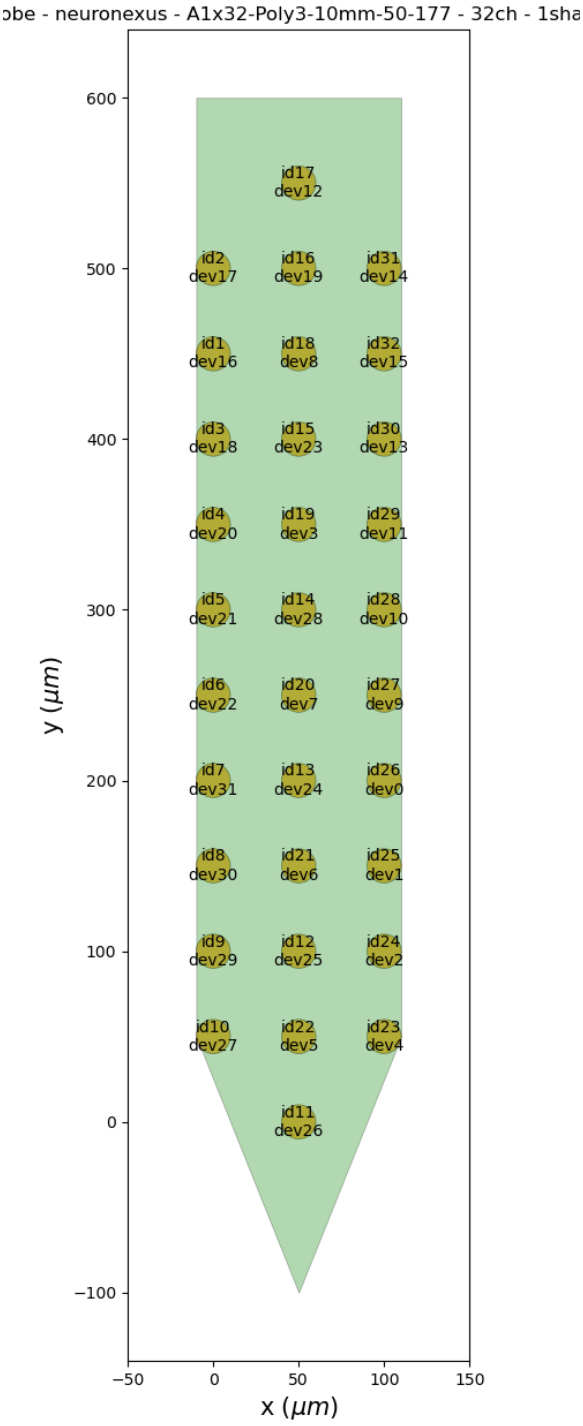
```
probe.wiring_to_device('H32>RHD2132')
print(probe.device_channel_indices)
```

```
[16 17 18 20 21 22 31 30 29 27 26 25 24 28 23 19 12  8  3  7  6  5  4  2
  1  0  9 10 11 13 14 15]
```

In this figure we have 2 numbers for each contact:

- the upper number "prbXX" is the Neuronexus index (one-based)
- the lower "devXX" is the channel on the Intan device (zero-based)

```
fig, ax = plt.subplots(figsize=(5, 15))
plot_probe(probe, with_contact_id=True, with_device_index=True, ax=ax)


plt.show()

"""
Available wiring "pathways"
---------------------------

The available pathways can be found in the `probeinterface.wiring <>`_ module.

The following pathways are available:
"""

from probeinterface import get_available_pathways

print(get_available_pathways())
```


obe - neuronexus - A1x32-Poly3-10mm-50-177 - 32ch - 1sha
id17
dev12
id2
dev17
id16
dev19
id31
dev14
id1
dev16
id18
dev8
id32
dev15
id3
dev18
id15
dev23
id30
dev13
id4
dev20
id19
dev3
id29
dev11
id5
dev21
id14
dev28
id28
dev10
id6
dev22
id20
dev7
id27
dev9
id7
dev31
id13
dev24
id26
dev0
id8
dev30
id21
dev6
id25
dev1
id9
dev29
id12
dev25
id24
dev2
id10
dev27
id22
dev5
id23
dev4
id11
dev26
600
500
400
300
200
100
0
−100
−50
0
50
100
150
y (μm)
x (μm)

```
['H32>RHD2132', 'ASSY-156>RHD2164', 'ASSY-116>RHD2132', 'ASSY-77>Adpt.A64-Om32_2x-sm-
↪NN>RHD2164', 'ASSY-77>Adpt.A64-Om32_2x-sm-NN>two_RHD2132', 'ASSY-77>Adpt.A64-Om32_
↪2x-sm-cambridgeneurotech>RHD2164', 'cambridgeneurotech_mini-amp-64']
```

Total running time of the script: (0 minutes 0.171 seconds)

1.12 Plot values

Here is an example of how to plot values with color scales. And also to plot an interpolated image.

```
from pprint import pprint

import numpy as np
import matplotlib.pyplot as plt

from probeinterface import Probe, get_probe
from probeinterface.plotting import plot_probe
```

Download one probe:

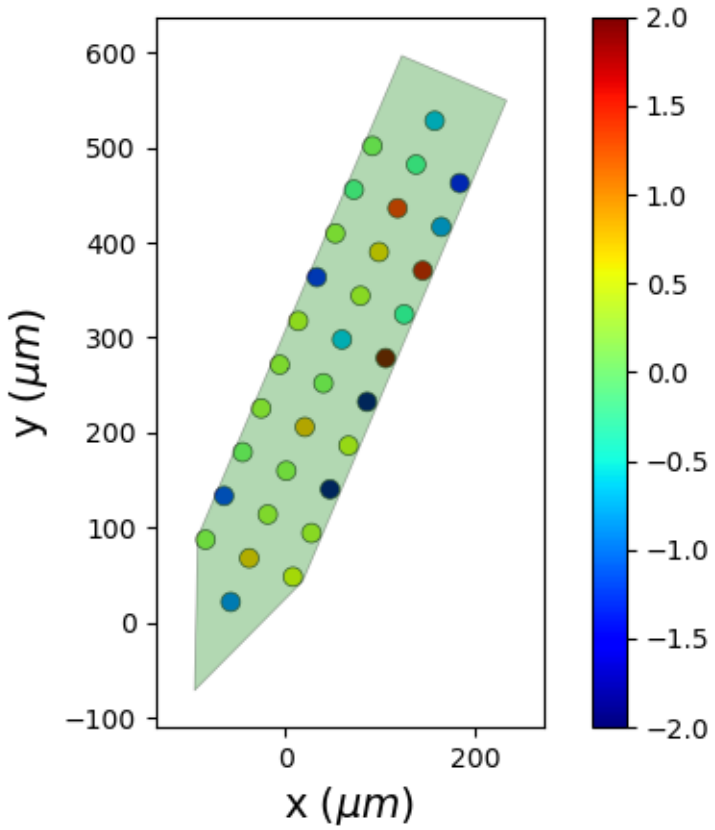
```
manufacturer = 'neuronexus'
probe_name = 'A1x32-Poly3-10mm-50-177'

probe = get_probe(manufacturer, probe_name)
probe.rotate(23)
```

fake values

```
values = np.random.randn(32)
```

plot with values

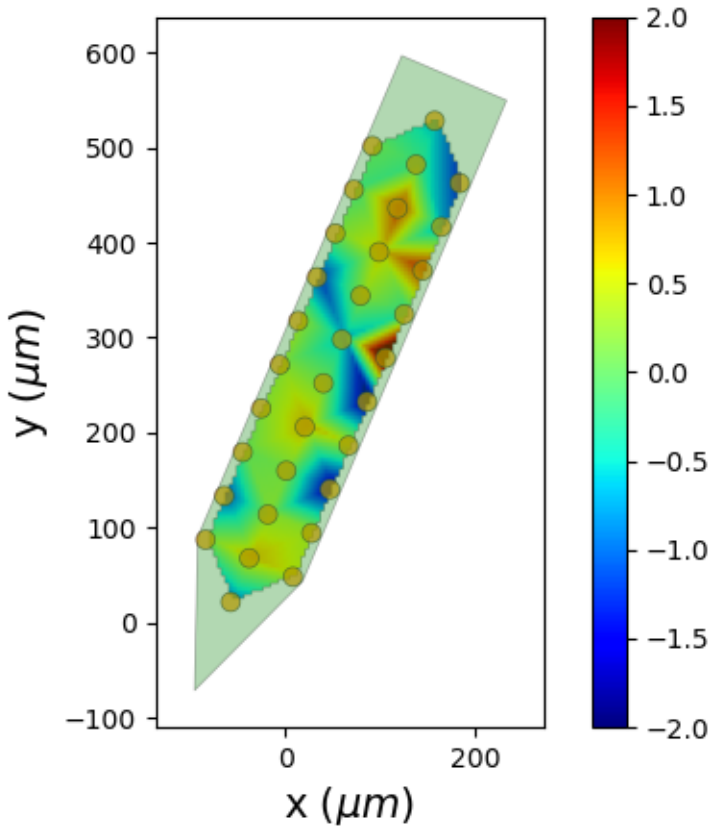
```
fig, ax = plt.subplots()
poly, poly_contour = plot_probe(probe, contacts_values=values,
            cmap='jet', ax=ax, contacts_kargs={'alpha' : 1},  title=False)
poly.set_clim(-2, 2)
fig.colorbar(poly)
```




```
<matplotlib.colorbar.Colorbar object at 0x7fff56b31010>
```

generated an interpolated image and plot it on top

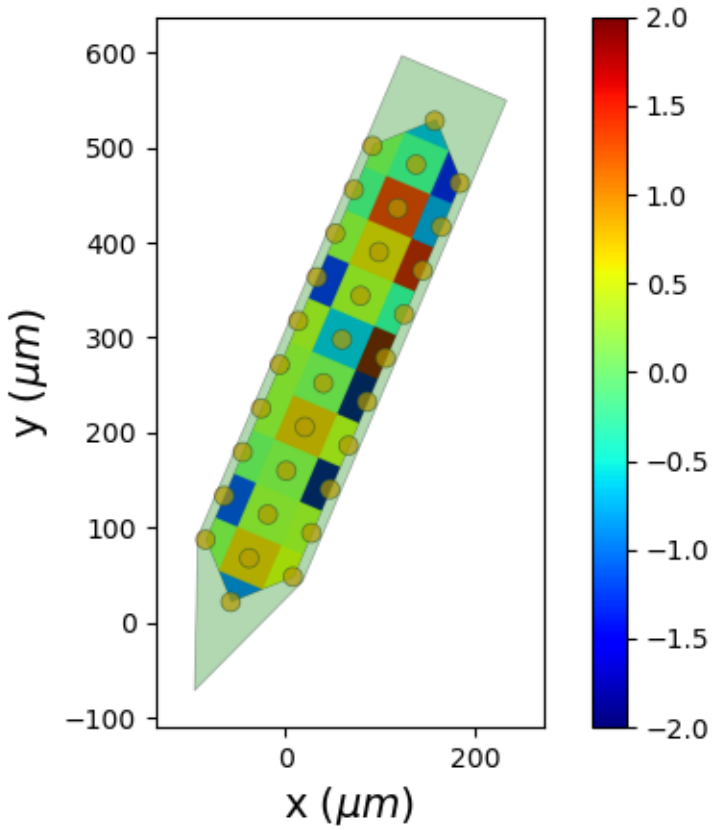
```
image, xlims, ylims = probe.to_image(values, pixel_size=4, method='linear')

print(image.shape)

fig, ax = plt.subplots()
plot_probe(probe, ax=ax, title=False)
im = ax.imshow(image, extent=xlims+ylims, origin='lower', cmap='jet')
im.set_clim(-2,2)
fig.colorbar(im)
```




```
(127, 67)

<matplotlib.colorbar.Colorbar object at 0x7fff4373a850>
```

works with several interpolation methods

```
image, xlims, ylims = probe.to_image(values, num_pixel=1000, method='nearest')

fig, ax = plt.subplots()
plot_probe(probe, ax=ax, title=False)
im = ax.imshow(image, extent=xlims+ylims, origin='lower', cmap='jet')
im.set_clim(-2,2)
fig.colorbar(im)



plt.show()
```



Total running time of the script: (0 minutes 1.989 seconds)

1.13 Overview

1.13.1 Introduction

To record neural electrical signals, extracellular neural probes are inserted into nervous tissues (e.g. brain, spinal cord). Neural probes are (usually) multi-channel arrays able to record from multiple contacts simultaneously, spanning from a few channels (e.g. tetrodes) to high-density silicon probes (e.g. Neuropixels - with up to 384 recording channels).

These probes (especially silicon probes) generally have a complex layout (or geometry) and can be connected to the recording system in multiple ways (wiring). To connect a neural probe to a recording device (e.g. Open Ephys, Blackrock, Ripple, Plexon, Intan, Multi-channel System) a headstage is used that is connected to the main recording device.

The complexity of the probe wiring and device wiring leads to the difficult task of directly linking the **physical contacts on the probe** and the **logical channel indices on the device**.

Recent *spike sorting* (i.e. methods to extract single neurons' activity from the extracellular recordings) algorithms strongly rely on the probe geometry to exploit the spatial distribution of the contacts and improve their performance.

Therefore, there is a need to correctly handle probe geometry and the wiring to the recording device in an easy-to-use and standardized way.

As an example, imagine you have:

- a **Neuronexus A1x32-Poly2** probe

- with the **intan RHD2132** headstage using the **omnetics 1315** connector
- connected on the **port B of an Open Ephys board**

What would be your final channel mapping be?

Of course one can sit down in the lab and try to figure it out… The goal of `probeinterface` is to make this time-consuming and error-prone process easier and standardized.

1.13.2 Scope

The scope of this project is to handle one (or several) Probe with three simple python classes:

- `Shank`
- *`Probe`*
- *`ProbeGroup`*.

These classes handle:

- probe geometry (2D or 3D contact layout)
- probe planar contours (polygon)
- shape and size of the contacts
- probe wiring to the recording device
- combination of several probes: global geometry + global wiring

This package also provide:

- read/write to a common format (JSON based)
- read/write function to other existing formats (PRB, NWB, CSV, MEArec, SpikeGLX, …)
- plotting routines
- generator functions to create user-defined probes

1.13.3 Goal 1

This common interface could be used by several projects for spike sorting and electrophysiology analysis:

- SpikeInterface: integrate this into spikeextractors to handle channel location and wiring
- NEO: handle array_annotations for AnalogSignal
- SpikeForest: use this package for plotting probe activity
- Phy: integrate for probe display
- SpyKING Circus: handle probe with this package
- Kilosort: handle probe with this package
- …and more

1.13.4 Goal 2

Implement and maintain a collection of widely used probes in Neuroscience, for example:

- Neuronexus
- Cambridge Neurotech

We have started a work-in-progess repo with a probe library

1.13.5 Existing projects

probeinterface is not the first attempt to build a library of available probes. Here is a list of available resources:

- JRClust probe library - Matlab format
- Klusta probe library - PRB format
- SpyKING Circus probe library - PRB format
- Justin Kiggins did some script for neuronexus mapping

All of these projects only describe the contact positions. Furthermore there is a strong ambiguity for users between the **contact index on the probe** and the **channel index on device**. This could lead to a wrong interpretation of the wiring.

With probeinterface we try to provide a unified framework for probe description, handling, and a comprehensive probe library.

1.13.6 Acknowledgements

The probeinterface is inspired on the MEAutility package, written by Alessio Buccino.

While the general idea of having an enhanced probe description is present, the MEAutility package mainly focuses on handling probes for modeling purposes, hence missing the wiring concept, and it can only handle a single probe at a time.

With probeinterface the focus is also to combine several Probes and to handle complex wiring for experimental description.

1.14 Examples

Start here with a tutorial showing probeinterface.

1.15 Format specifications

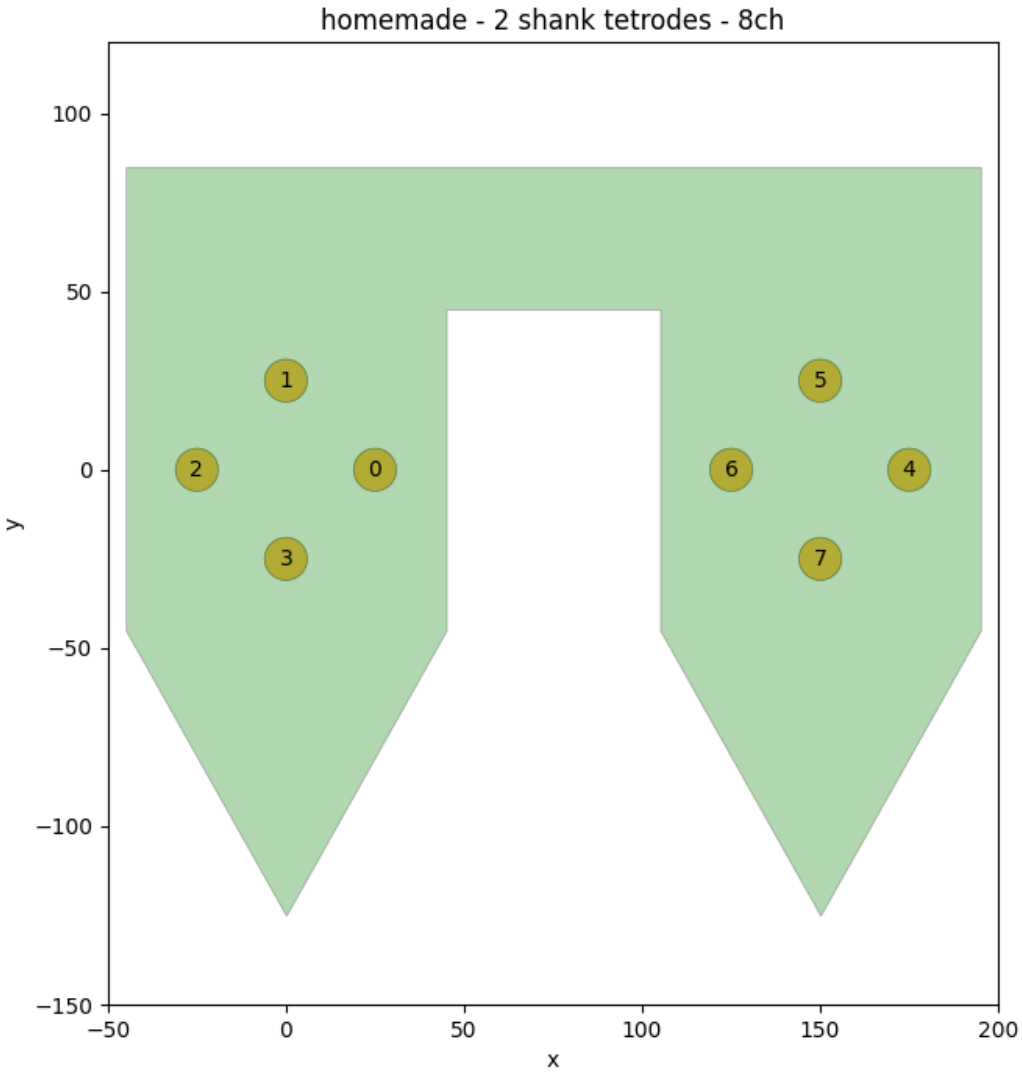
With probeinterface we introduce a simple format based on the JSON format. The format is a trivial json-serialisation of a Python dictionary. The dictionary maps every attribute of the Probe class.

In fact, the format itself describes a ProbeGroup, so it can include several probes. The format can describe a simple unique probe with its geometry and wiring, as well as a full experimental setup with several probes and their wiring to the recording device.

Here is a description of the fields in the json file.

Let's imagine we want to describe a probe with:

- 8 channels
- 2 shanks (one tetrode on each shank)



The first part contains fields about the `probeinterface` version and a list of probes:

```
{
  "specification": "probeinterface",
  "version": "0.1.0",
  "probes": [
    {
      ...
    }
  ]
}
```

Then each probe will be a sub-dictionary in the `probes` list:

```
{
    "ndim": 2,
    "si_units": "um",
    "annotations": {
        "name": "2 shank tetrodes",
        "manufacturer": "homemade"
    },
    "contact_positions": [
...
```

The probe dictionary contains all necessary fields and optional fields.

Necessary:

- ndim
- si_units
- annotations
- contact_positions
- contact_shapes
- contact_shape_params

Optional:

- contact_plane_axes
- probe_planar_contour
- device_channel_indices
- shank_ids

An example of a full json file:

```
{
    "specification": "probeinterface",
    "version": "0.1.0",
    "probes": [
        {
            "ndim": 2,
            "si_units": "um",
            "annotations": {
                "name": "2 shank tetrodes",
                "manufacturer": "homemade"
            },
            "contact_positions": [
                [
                    25.0,
                    0.0
                ],
                [
                    0.0,
                    25.0
                ],
                [
```

(continues on next page)

(continued from previous page)

```
                    -25.0,
                    0.0
                ],
                [
                    0.0,
                    -25.0
                ],
                [
                    175.0,
                    0.0
                ],
                [
                    150.0,
                    25.0
                ],
                [
                    125.0,
                    0.0
                ],
                [
                    150.0,
                    -25.0
                ]
            ],
            "contact_plane_axes": [
                [
                    [
                        1.0,
                        0.0
                    ],
                    [
                        0.0,
                        1.0
                    ]
                ],
                [
                    [
                        1.0,
                        0.0
                    ],
                    [
                        0.0,
                        1.0
                    ]
                ],
                [
                    [
                        1.0,
                        0.0
                    ],
                    [
                        0.0,
```

(continues on next page)

(continued from previous page)

```
                    1.0
                ]
            ],
            [
                [
                    1.0,
                    0.0
                ],
                [
                    0.0,
                    1.0
                ]
            ],
            [
                [
                    1.0,
                    0.0
                ],
                [
                    0.0,
                    1.0
                ]
            ],
            [
                [
                    1.0,
                    0.0
                ],
                [
                    0.0,
                    1.0
                ]
            ],
            [
                [
                    1.0,
                    0.0
                ],
                [
                    0.0,
                    1.0
                ]
            ],
            [
                [
                    1.0,
                    0.0
                ],
                [
                    0.0,
                    1.0
                ]
```

(continues on next page)

(continued from previous page)

```
            ]
        ],
        "contact_shapes": [
            "circle",
            "circle",
            "circle",
            "circle",
            "circle",
            "circle",
            "circle",
            "circle"
        ],
        "contact_shape_params": [
            {
                "radius": 6
            },
            {
                "radius": 6
            },
            {
                "radius": 6
            },
            {
                "radius": 6
            },
            {
                "radius": 6
            },
            {
                "radius": 6
            },
            {
                "radius": 6
            },
            {
                "radius": 6
            }
        ],
        "probe_planar_contour": [
            [
                -45.0,
                85.0
            ],
            [
                -45.0,
                45.0
            ],
            [
                -45.0,
                -45.0
            ],
            [
```

(continues on next page)

(continued from previous page)

```
                    0.0,
                    -125.0
                ],
                [
                    45.0,
                    -45.0
                ],
                [
                    45.0,
                    45.0
                ],
                [
                    105.0,
                    45.0
                ],
                [
                    105.0,
                    -45.0
                ],
                [
                    150.0,
                    -125.0
                ],
                [
                    195.0,
                    -45.0
                ],
                [
                    195.0,
                    45.0
                ],
                [
                    195.0,
                    85.0
                ]
            ],
            "shank_ids": [
                0,
                0,
                0,
                0,
                1,
                1,
                1,
                1
            ]
        }
    ]
}
```

1.16 Probeinterface public library

Probeinterface also handles a collection of probe descriptions on the GitHub platform

The python module has a simple function to download and cache locally by using *get_probe(…)*

```
from probeinterface import get_probe
probe = get_probe(manufacturer='neuronexus',
            probe_name='A1x32-Poly3-10mm-50-177')
```

We expect to build rapidly commonly used probes in this public repository.

1.16.1 How to contribute

TODO: explain with more details

1. **Generate the JSON file with probeinterface (or directly**
   with another language)
2. Generate an image of the probe with the *plot_probe* function in probeinterface
3. Clone the probeinterface_library repo
4. Put the JSON file and image into the correct folder or make a new folder (following the format of the repo)
5. Push to one of your branches with a git client
6. Make a pull request to the main repo

1.17 API

1.17.1 Probe

class probeinterface.**Probe**(*ndim: int = 2*, *si_units: str = 'um'*, *name: str | None = None*, *serial_number: str | None = None*, *model_name: str | None = None*, *manufacturer: str | None = None*)

Class to handle the geometry of one probe.

This class mainly handles contact positions, in 2D or 3D. Optionally, it can also handle the shape of the contacts and the shape of the probe.

add_probe_to_zarr_group(*group: zarr.Group*) → None

Serialize the probe's data and structure to a specified Zarr group.

This method is used to save the probe's attributes, annotations, and other related data into a Zarr group, facilitating integration into larger Zarr structures.

Parameters

group
[zarr.Group] The target Zarr group where the probe's data will be stored.

annotate(***kwargs*)

Annotates the probe object.

Parameters

**kwargs : list of keyword arguments to add to the annotations (e.g., brain_area="CA1")

annotate_contacts(***kwargs*)

Annotates the contacts of the probe.

Parameters

**kwargs : list of keyword arguments to add to the annotations (e.g., quality=["good", "bad", ...])

property contact_positions

The position of the center for each contact

copy()

Copy to another Probe instance.

Note: device_channel_indices are not copied and contact_ids are not copied

create_auto_shape(*probe_type: tip' | 'rect' | 'circular = 'tip'*, *margin: float = 20.0*)

Create a planar contour automatically based on probe contact positions.

This function generates a 2D polygon that outlines the shape of the probe, adjusted by a specified margin. The resulting contour is set as the planar contour of the probe.

Parameters

probe_type

[{"tip", "rect", "circular"}, default: "tip"] The type of probe used to collect contact data:

- "tip": Assumes a single-point contact probe. The generated contour is

a rectangle with a triangular "tip" extending downwards. * "rect": Assumes a rectangular contact probe. The generated contour is a rectangle. * "circular": Assumes a circular contact probe. The generated contour is a circle.

margin

[float, default: 20.0] The margin to add around the contact positions. The behavior varies by probe type:

- "tip": The margin is added around the rectangular portion of the contour

and to the base of the tip. The tip itself is extended downwards by four times the margin value. * "rect": The margin is added evenly around all sides of the rectangle. * "circular": The margin is added to the radius of the circle.

Notes

This function is designed for 2D data only. If you have 3D data, consider projecting it onto a plane before using this method.

static from_dataframe(*df: pandas.DataFrame*) → *Probe*

Create Probe from a pandas.DataFrame see Probe.to_dataframe()

Parameters

df

[pandas.DataFrame] The dataframe representation of the probe

Returns

probe
[Probe] The instantiated Probe object

static from_dict(*d: dict*) → *Probe*

Instantiate a Probe from a dictionary

Parameters

d
[dict] The dictionary representation of the probe

Returns

probe
[Probe] The instantiated Probe object

static from_numpy(*arr: ndarray*) → *Probe*

Create Probe from a complex numpy array see Probe.to_numpy()

Parameters

arr
[np.array] The structured np.array representation of the probe

Returns

probe
[Probe] The instantiated Probe object

static from_zarr(*folder_path: str | Path*) → *Probe*

Deserialize the Probe object from a Zarr file located at the given folder path.

Parameters

folder_path
[str | Path] The path to the folder where the Zarr file is located.

Returns

Probe
An instance of the Probe class initialized with data from the Zarr file.

static from_zarr_group(*group: zarr.Group*) → *Probe*

Load a probe instance from a given Zarr group.

Parameters

group
[zarr.Group] The Zarr group from which to load the probe.

Returns

Probe
An instance of the Probe class initialized with data from the Zarr group.

get_contact_count() → int
Return the number of contacts on the probe.

get_contact_vertices() → list
Return a list of contact vertices.

get_shank_count() → int
Return the number of shanks for this probe.

get_shanks()
Return the list of Shank objects for this Probe

get_slice(*selection: ndarray[bool | int]*)
Get a copy of the Probe with a sub selection of contacts.

Selection can be boolean or by index

Parameters

selection
[np.array of bool or int (for index)] Either an np.array of bool or for desired selection of contacts or the indices of the desired contacts

Returns

sliced_probe: Probe
The sliced probe

move(*translation_vector: np.array | list*)
Translate the probe in one direction.

Parameters

translation_vector
[list or array] The translation vector in shape 2D or 3D

rotate(*theta: float*, *center: list | np.ndarray | None = None*, *axis: 'xy' | 'yz' | 'xz' | None = None*)
Rotate the probe around a specified axis.

Parameters

theta
[float] In degrees, anticlockwise/counterclockwise

center
[array | list | None, default: None] Center of rotation. If None, the center of probe is used

axis
["xy" | "yz" | "xz" | None, default: None] Axis of rotation. It must be None for 2D probes It must be given for 3D probes

rotate_contacts(*thetas: float | np.array[float] | list[float]*)
Rotate each contact of the probe. Internally, it modifies the contact_plane_axes.

Parameters

thetas
[float | array[float] | list[float]] Rotation angle in degrees. If scalar, then it is applied to all contacts.

set_contact_ids(*contact_ids: np.array | list*)

Set contact ids. Channel ids are converted to strings. Contact ids must be **unique** for the **Probe** and also for the **ProbeGroup**

Parameters

contact_ids
[list or array] Array with contact ids. If contact_ids are int or float they are converted to str

set_contacts(*positions*, *shapes='circle'*, *shape_params={'radius': 10}*, *plane_axes=None*, *contact_ids=None*, *shank_ids=None*)

Sets contacts to a Probe.

This sets four attributes of the probe:
contact_positions, contact_shapes, contact_shape_params, _contact_plane_axes

Parameters

positions
[array (num_contacts, ndim)] Positions of contacts (2D or 3D depending on probe 'ndim').

shapes
["circle" | "square" | "rect" | array, default: "circle"] Shape of each contact ('circle'/'square'/'rect').

shape_params
[dict or list of dict, default: {"radius": 10}] Contains kwargs for shapes: * "radius" for circle * "width" for square, * "width/height" for rect

plane_axes
[np.array (num_contacts, 2, ndim) | None, default: None] Defines the two axes of the contact plane for each electrode. The third dimension corresponds to the probe *ndim* (2d or 3d).

contact_ids: array[str] | None, default: None
Defines the contact ids for the contacts. If None, contact ids are not assigned.

shank_ids
[array[str] | None, default: None] Defines the shank ids for the contacts. If None, then these are assigned to a unique Shank.

set_device_channel_indices(*channel_indices: np.array | list*)

Manually set the device channel indices.

If some channels are not connected or not recorded then channel should be set to "-1"

Parameters

channel_indices
[array[int] | list[int]] The device channel indices to set

set_planar_contour(*contour_polygon: list*)

Set the planar contour (the shape) of the probe.

Parameters

contour_polygon
[list] List of contour points (2D or 3D depending on ndim)

set_shank_ids(*shank_ids: np.array | list*)

Set shank ids.

Parameters

shank_ids
[list or array] Array with shank ids, if int or float converted to strings

to_2d(*axes: 'xy' | 'yz' | 'xz' = 'xy'*)

Transform 3d probe to 2d probe.

Note: device_channel_indices are not copied.

Parameters

plane
["xy" | "yz" | "xz", default: "xy"] The plane on which the 2D probe is defined.

to_3d(*axes: 'xy' | 'yz' | 'xz' = 'xz'*)

Transform 2d probe to 3d probe.

Note: device_channel_indices are not copied.

Parameters

axes
["xy" | "yz" | "xz", default: "xz"] The axes that define the plane on which the 2D probe is defined. 'xy', 'yz' ', xz'

to_dataframe(*complete: bool = False*) → pandas.DataFrame

Export the probe to a pandas dataframe

Parameters

complete
[bool, default: False] If True, export complete information about the probe, including the probe plane axis.

Returns

df
[pandas.DataFrame] The dataframe representation of the probe

to_dict(*array_as_list: bool = False*) → dict

Create a dictionary of all necessary attributes. Useful for dumping and saving to json.

Parameters

array_as_list
[bool, default: False] If True, arrays are converted to lists

Returns

d
[dict] The dictionary representation of the probe

`to_image`(*values: np.array | list*, *pixel_size: float = 0.5*, *num_pixel: Optional[int] = None*, *method: 'linear' | 'nearest' | 'cubic = 'linear'*, *xlims: Optional[tuple] = None*, *ylims: Optional[tuple] = None*) → tuple[np.ndarray, tuple, tuple]

Generated a 2d (image) from a values vector with an interpolation into a grid mesh.

Parameters

values
[np.ndarray | list] vector same size as contact number to be color plotted

pixel_size
[float, default: 0.5] size of one pixel in micrometers

num_pixel
[Optional[int] | None, default: None] alternative to pixel_size give pixel number of the image width

method
["linear" | "nearest" | "cubic", default: "linear"] Method of interpolation to generate a grid mesh

xlims
[Optional[tuple], default: None] Force image xlims

ylims
[Optional[tuple], default: None] Force image ylims

Returns

image
[2d array] The generated image

xlims
[tuple] The x limits

ylims
[tuple] The y limits

`to_numpy`(*complete: bool = False*) → array

Export the probe to a numpy structured array. This array handles all contact attributes.

Similar to the 'to_dataframe()' pandas function, but without pandas dependency.

The intended use is to attach this array to a recording object as a property ("contact vector")

Parameters

complete
[bool, default: False] If True, export complete information about the probe, including contact_plane_axes/si_units/device_channel_indices.

Returns

arr
[numpy.array] Structured array with the following dtype schema:

dtype = [
('x', 'float64'), ('y', 'float64'), ('z', 'float64', optional), ('contact_shapes', 'U64'), # Shape parameters ('shape_param_1', 'float64'), ('shape_param_2', 'float64'),

⋮⋮⋮

variable number of shape parameters … ('shank_ids', 'U64'), ('contact_ids', 'U64'),

The rest is added only if *complete=True* ('device_channel_indices', 'int64', optional), ('si_units', 'U64', optional), ('plane_axis_x_0', 'float64', optional), ('plane_axis_x_1', 'float64', optional), ('plane_axis_y_0', 'float64', optional), ('plane_axis_y_1', 'float64', optional), ('plane_axis_z_0', 'float64', optional), ('plane_axis_z_1', 'float64', optional), # Annotations ('annotation_name_1', 'dtype of annotation', optional), ('annotation_name_2', 'dtype of annotation', optional), ⋮

⋮⋮⋮

variable number of annotations …

]

to_zarr(*folder_path: str | Path*) → None

Serialize the Probe object to a Zarr file located at the specified folder path.

This method initializes a new Zarr group at the given folder path and calls *add_probe_to_zarr_group* to serialize the Probe's data into this group, effectively storing the entire Probe's state in a Zarr archive.

Parameters

folder_path
[str | Path] The path to the folder where the Zarr data structure will be created and where the serialized data will be stored. If the folder does not exist, it will be created.

wiring_to_device(*pathway: str*, *channel_offset: int = 0*)

Automatically set device_channel_indices based on a pathway.

See probeinterface.get_available_pathways()

Parameters

pathway
[str] The pathway. E.g. 'H32>RHD'

channel_offset: int, default: 0
An optional offset to add to the device_channel_indices

1.17.2 ProbeGroup

class probeinterface.**ProbeGroup**

Class to handle a group of Probe objects and the global wiring to a device.

Optionally, it can handle the location of different probes.

add_probe(*probe:* Probe)

Add an additional probe to the ProbeGroup

Parameters

probe: Probe
The probe to add to the ProbeGroup

auto_generate_contact_ids(**args*, ***kwargs*)
Annotate all contacts with unique contact_id values.

Parameters

*args: will be forwarded to *probeinterface.utils.generate_unique_ids* **kwargs: will be forwarded to *probeinterface.utils.generate_unique_ids*

auto_generate_probe_ids(**args*, ***kwargs*)
Annotate all probes with unique probe_id values.

Parameters

*args: will be forwarded to *probeinterface.utils.generate_unique_ids* **kwargs: will be forwarded to *probeinterface.utils.generate_unique_ids*

static from_dict(*d: dict*)
Instantiate a ProbeGroup from a dictionary

Parameters

d
[dict] The dictionary representation of the probegroup

Returns

probegroup
[ProbeGroup] The instantiated ProbeGroup object

static from_numpy(*arr: ndarray*) → *ProbeGroup*
Create ProbeGroup from a complex numpy array see ProbeGroup.to_numpy()

Parameters

arr
[np.array] The structured np.array representation of the probe

Returns

probegroup
[ProbeGroup] The instantiated ProbeGroup object

get_contact_count() → int
Total number of channels.

Returns

n: int
The total number of channels

get_global_contact_ids() → ndarray
Gets all contact ids concatenated across probes

Returns

contact_ids: np.ndarray
An array of the contaact ids across all probes

get_global_device_channel_indices() → ndarray
Gets the global device channels indices and returns as an array

Returns

channels: np.ndarray
a numpy array vector with 2 columns (probe_index, device_channel_indices)

Notes

If a channel within channels has a value of -1 this indicates that that channel is disconnected

set_global_device_channel_indices(*channels: np.array | list*)
Set global indices for all probes

Parameters

channels: np.array | list
The device channal indices to be set

to_dataframe(*complete: bool = False*) → pandas.DataFrame
Export the probegroup to a pandas dataframe

Parameters

complete
[bool, default: False] If True, export complete information about the probegroup, including the probe plane axis.

Returns

df
[pandas.DataFrame] The dataframe representation of the probegroup

to_dict(*array_as_list: bool = False*)
Create a dictionary of all necessary attributes.

Parameters

array_as_list
: [bool, default: False] If True, arrays are converted to lists, by default False

Returns

d
: [dict] The dictionary representation of the probegroup

to_numpy(*complete: bool = False*) → ndarray

Export all probes into a numpy array.

Parameters

complete: bool, default: False
: If True, export complete information about the probegroup including contact_plane_axes/si_units/device_channel_indices

1.17.3 Import/export to formats

Read/write probe info using a variety of formats:

- probeinterface (.json)
- PRB (.prb)
- CSV (.csv)
- mearec (.h5)
- ironclust/jrclust (.mat)
- Neurodata Without Borders (.nwb)

probeinterface.io.**read_probeinterface**(*file: str | Path*) → *ProbeGroup*

Read probeinterface JSON-based format.

Parameters

file: Path or str
: The file path

Returns

probegroup : ProbeGroup object

probeinterface.io.**write_probeinterface**(*file: str | Path*, *probe_or_probegroup:* Probe | ProbeGroup)

Write a probeinterface JSON file.

The format handles several probes in one file.

Parameters

file
: [Path or str] The file path

probe_or_probegroup
: [Probe or ProbeGroup object] If probe is given a probegroup is created anyway

probeinterface.io.**read_prb**(*file: str | Path*) → *ProbeGroup*

Read a PRB file and return a ProbeGroup object.

Since PRB does not handle contact shapes, contacts are set to be circle of 5um radius. Same for the probe shape, where an auto shape is created.

PRB format does not contain any information about the channel of the probe Only the channel index on device is given.

Parameters

file
[Path or str] The file path

Returns

probegroup : ProbeGroup object

probeinterface.io.**write_prb**(*file: str*, *probegroup:* ProbeGroup, *total_nb_channels: int | None = None*, *radius: float | None = None*, *group_mode: str = 'by_probe'*)

Write ProbeGroup into a prb file.

This format handles:

- multi Probe with channel group index key
- channel positions with "geometry"
- device_channel_indices with "channels "key

Note: much information is lost in the PRB format:

- contact shape
- shape
- channel index

Note:

- "total_nb_channels" is needed by spyking-circus
- "radius" is needed by spyking-circus
- "graph" is not handled

Parameters

file: str
The name of the file to be written

probegroup: ProbeGroup
The Probegroup to be used for writing

total_nb_channels: Optional[int], default None
***to do

radius: Optional[float], default None
*** to do

group_mode: str
One of "by_probe" or "by_shank

probeinterface.io.**read_csv**(*file: str | Path*)

Return a 2 or 3 columns csv file with contact positions

probeinterface.io.**write_csv**(*file*, *probe*)

Write contact postions into a 2 or 3 columns csv file

probeinterface.io.**read_mearec**(*file: str | Path*) → *Probe*

Read probe position, and contact shape from a MEArec file.

See https://mearec.readthedocs.io/en/latest/ and https://doi.org/10.1007/s12021-020-09467-7 for implementation.

Parameters

file
[Path or str] The file path

Returns

probe : Probe object

probeinterface.io.**read_nwb**(*file*)

Read probe position from an NWB file

1.17.4 Probe generators

This module contains useful helper functions for generating probes.

probeinterface.generator.**generate_dummy_probe**(*elec_shapes: circle' | 'square' | 'rect = 'circle'*) → *Probe*

Generate a dummy probe with 3 columns and 32 contacts. Mainly used for testing and examples.

Parameters

elec_shapes
["circle" | "square" | "rect", default: 'circle'] Shape of the electrodes

Returns

probe
[Probe] The generated probe

probeinterface.generator.**generate_dummy_probe_group**() → *ProbeGroup*

Generate a ProbeGroup with 2 probes. Mainly used for testing and examples.

Returns

probe
[Probe] The generated probe

probeinterface.generator.**generate_tetrode**(*r: float = 10.0*) → *Probe*

Generate a tetrode Probe.

Parameters

r: float, default: 10
The distance multiplier for the positions

Returns

probe
[Probe] The generated probe

probeinterface.generator.**generate_multi_columns_probe**(*num_columns: int = 3*, *num_contact_per_column: int = 10*, *xpitch: float = 20*, *ypitch: float = 20*, *y_shift_per_column: Optional[np.array | list] = None*, *contact_shapes: circle' | 'rect' | 'square = 'circle'*, *contact_shape_params: dict = {'radius': 6}*) → *Probe*

Generate a Probe with several columns.

Parameters

num_columns
[int, default: 3] Number of columns

num_contact_per_column
[int, default: 10] Number of contacts per column

xpitch
[float, default: 20] Pitch in x direction

ypitch
[float, default: 20] Pitch in y direction

y_shift_per_column
[Optional[array-like], default: None] Shift in y direction per column. It needs to have the same length as num_columns, by default None

contact_shapes
[“circle” | “rect” | “square”, default: “circle”] Shape of the contacts

contact_shape_params
[dict, default: {‘radius’: 6}] Parameters for the shape. For circle: {“radius”: float} For square: {“width”: float} For rectangle: {“width”: float, “height”: float}

Returns

probe
[Probe] The generated probe

probeinterface.generator.**generate_linear_probe**(*num_elec: int = 16*, *ypitch: float = 20*, *contact_shapes: circle' | 'rect' | 'square = 'circle'*, *contact_shape_params: dict = {'radius': 6}*) → *Probe*

Generate a one-column linear probe.

Parameters

num_elec
[int, default: 16] Number of electrodes

ypitch
[float, default: 20] Pitch in y direction

contact_shapes
["circle" | "rect" | "square", default 'circle'] Shape of the contacts

contact_shape_params
[dict, default: {'radius': 6}] Parameters for the shape. For circle: {"radius": float} For square: {"width": float} For rectangle: {"width": float, "height": float}

Returns

probe
[Probe] The generated probe

1.17.5 Plotting

A simple implementation for plotting a Probe or ProbeGroup using matplotlib.

Depending on Probe.ndim, the plotting is done in 2D or 3D

probeinterface.plotting.**plot_probe**(*probe*, *ax=None*, *contacts_colors=None*, *with_contact_id: bool = False*, *with_device_index: bool = False*, *text_on_contact: list | ndarray | None = None*, *contacts_values: ndarray | None = None*, *cmap: str = 'viridis'*, *title: bool = True*, *contacts_kargs: dict = {}*, *probe_shape_kwargs: dict = {}*, *xlims: tuple | None = None*, *ylims: tuple | None = None*, *zlims: tuple | None = None*, *show_channel_on_click: bool = False*)

Plot a Probe object. Generates a 2D or 3D axis, depending on Probe.ndim

Parameters

probe
[Probe] The probe object

ax
[matplotlib.axis | None, default: None] The axis to plot the probe on. If None, an axis is created

contacts_colors
[matplotlib color | None, default: None] The color of the contacts

with_contact_id
[bool, default: False] If True, channel ids are displayed on top of the channels

with_device_index
[bool, default: False] If True, device channel indices are displayed on top of the channels

text_on_contact: None | list | numpy.array, default: None
Addintional text to plot on each contact

contacts_values
[np.array, default: None] Values to color the contacts with

cmap
[a colormap color, default: "viridis"] A colormap color

title
[bool, default: True] If True, the axis title is set to the probe name

contacts_kargs
[dict, default: {}] Dict with kwargs for contacts (e.g. alpha, edgecolor, lw)

probe_shape_kwargs
[dict, default: {}] Dict with kwargs for probe shape (e.g. alpha, edgecolor, lw)

xlims
[tuple | None, default: None] Limits for x dimension

ylims
[tuple | None, default: None] Limits for y dimension

zlims
[tuple | None, default: None] Limits for z dimension

show_channel_on_click
[bool, default: False] If True, the channel information is shown upon click

Returns

poly
[PolyCollection] The polygon collection for contacts

poly_contour
[PolyCollection] The polygon collection for the probe shape

`probeinterface.plotting.`**`plot_probe_group`**(*probegroup*, *same_axes: bool = True*, ***kargs*)
This function is deprecated and will be removed in 0.2.23 Please use plot_probegroup instead

1.17.6 Library

Provides functions to download and cache pre-existing probe files from some manufacturers.

The library is hosted here: https://gin.g-node.org/spikeinterface/probeinterface_library

The gin platform enables contributions from users.

`probeinterface.library.`**`get_probe`**(*manufacturer: str*, *probe_name: str*, *name: str | None = None*) → *Probe*
Get probe from ProbeInterface library

Parameters

manufacturer
["cambridgeneurotech" | "neuronexus"] The probe manufacturer

probe_name
[str (see probeinterface_libary for options)] The probe name

name
[str | None, default: None] Optional name for the probe

Returns

probe : Probe object

1.18 Release notes

1.18.1 probeinterface 0.2.25

Feb, 6th 2025

Features

- Add cambridge neurotech adaptor to wiring > RHD2164 (#304)
- Refactor neuropixel in a separate file and generate the NP library (#316)
- Regenerate ASSY-1-P1 and ASSY-1-P2 probes (#317)

Bug fixes

- Fix OpenEphys interface issue with multiple probes of which some are disabled (s#308)
- Update schema to floats for radius, width, height (#296, #297)

Tests

- Add json validation to tests (#310)

Packaging

- Upper bound to zarr version (#314)

1.18.2 probeinterface 0.2.24

Sept, 6th 2024

Features

- Add docs to get_auto_lims (#290)
- Support Open Ephys 1.0 (#294)

Bug fixes

- Fix open-ephys load with serial number as custom probe name (#293)
- Update schema to floats for radius, width, height (#296, #297)

1.18.3 probeinterface 0.2.23

Jul, 17th 2024

Features

- Add part number *PRB_1_2_0480_2* for newly-manufactured Neuropixels 1.0 probes (#286)

1.18.4 probeinterface 0.2.22

Jul, 15th 2024

Bugs

- Fix broken link in *ex_11_automatic_wiring.py* (#277)
- Load Open Ephys probe when multiple signal chains are present (#275)

Features

- Add dtype information to *to_numpy* function docstring documentation (#278, #282)
- Add circular auto-shape option (#279)
- Add automatic probe layout for 3brain/biocam recordings (#274)
- Add NP1016 probe (same as 1015) (#268)
- Add json schema for probe json file (#265)
- Probe reader for Neuropixels 1.0 in SpikeGadgets .rec file. (#260)

Refactoring

- Move get_auto_lims function to utils (#281)
- Deprecate *plot_probe_group* in favor of *plot_probegroup* (#267)

1.18.5 probeinterface 0.2.21

Feb, 1st 2024

Features

- Add equality dunder method and test to *Probe* object (#248)
- Add save to zarr method to *Probe* object (#250)
- Fix *Probe* special properties setters (skip empty strings) (#252, #253)

1.18.6 probeinterface 0.2.20

Dec, 11th 2023

Features

- Fix Open Ephys recording state options (#239)
- Make docstrings-style compliant and add assert messaging (#241)
- Add missing NP2.4 and NP-Ultra probe part number (#243)

1.18.7 probeinterface 0.2.19

Nov, 2nd 2023

Features

- Unify NP reading with probe part number (#232)

1.18.8 probeinterface 0.2.18

Oct, 30th 2023

Features

- Extend probe constructor (name, serial_number, manufacturer, model_name) (#206)
- Extend available NP2 probe types to commercial types (20** series) (#217)
- Remove `with_channel_index` argument from `plot_probe` (#229)
- Remove checker for unique contact ids in probe group (#229)
- Unify usage of “contact” and remove “channel” notation (except for “device_channel_index”) (#229)

Bug fixes

- Fix shank_pitch to NP-2.4 in SpikeGLX (#205)
- Fix y_shift_per_column docs and add assertion (#212)
- Change np.in1d to np.isin as the former will be deprecated (#220)
- Fix contour of NP2.0 for Open Ephys Neuropixels (#224)

Docs

- Add available pathways in ‘Automatic wiring’ docs (#213)
- Add Typing and Update Docstrings (#214)
- Add more details to how to contribute (#222)

1.18.9 probeinterface 0.2.17

Thanks a lot to Ramon Heberto Mayorquin, who did the most the changes for this release.

June, 26th 2023

Packaging / Documentation / Testing

- Move probe libray from GIN to GitHub (#195)
- Restructure repo to follow `src/probeinterface` convention
- Black formatting anf pre-commit CI (#190-#191)
- Add safe import utils (#175)
- Add type hints in IO module (#173)
- Fix code coverage (#171)
- Cron job and manual trigger for CI tests (#170)
- Add badges and code coverage to actions (#168)
- Handle temporary files in the test suite (#164)
- Reorganize testing directory (#163)
- Update test workflow to use latest version of actions (#162)
- Add MEArec test data (#176)

- Add Maxwel and 3brain to tests (#172)
- Add test for shank (#181)
- Improve Documentation (#157 / 158)

Features

- Extended support for Neuropixels probes: * Fix CatGT parsin (#193) * Map NP1010 probe to the usual NP1 geometry (#188) * Open Ephys: Support subselection of channels in Record Node (#180) * Add NP-ultra probe testing data for spikeglx (#177) * Add IMRO tests (#179) * Consolidate Neuropixels information in one place (#174) * Refactor NP information (#166-#167) * Add NHP probe support for SpikeGLX (#169-#165-#160-#156)

Bug fixes

- Fix DeprecrationWarning: invalid escape sequence m (#183)
- Mearec description to string (#159)

Testing

- Add MEArec test data (#176)
- Add Maxwel and 3brain to tests (#172)
- Add test for shank (#181)

1.18.10 probeinterface 0.2.16

February, 9th 2023

- Fix for read_spikeglx() when not all channels are saved

1.18.11 probeinterface 0.2.15

December, 12th 2022

- Bug fix when parsing Open Ephys version from XML file (https://github.com/SpikeInterface/probeinterface/pull/146)
- Add test files and tests for Open Ephys reader

1.18.12 probeinterface 0.2.14

October, 27th 2022

- Fix a **important bug** in `read_spikeglx()` / `read_imro()` that was leading to wrong contact locations when the Imec Readout Table (aka imRo) was set with complex multi-bank patterns. The bug was introduced with version **0.2.10**, released on September 1st 2022, and it is also present in these versions: **0.2.10**, **0.2.11**, **0.2.12**, and **0.2.13**.

  If you used spikeinterface/probeinterface with SpikeGLX data using one of these versions, we recommend you to check your contact positions (if they are non-standard - using the probe tip) and re-run your spike-sorting analysis if they are wrong.

  A big thanks to Tom Bugnon and Graham Findlay for spotting the bug.

1.18.13 probeinterface 0.2.13

October, 20th 2022

- Fix install bug due to pyproject.toml
- Better handling of contact ids for unknown NP type in OpenEphy
- Including ability to read phase3A neuropixel arrays

1.18.14 probeinterface 0.2.12

October, 10th 2022

- New configuration files with pyproject.toml

1.18.15 probeinterface 0.2.11

September, 14th 2022

- do not rely on BASESTATION field to parse OpenEphys probe

1.18.16 probeinterface 0.2.10

September, 1st 2022

- fix read_openephys()
- fix read_spikeglx()
- regenerate cambridge neurotec
- implement read_imro() / write_imro()
- Add new wiring : 'ASSY-77>Adpt.A64-Om32_2x-sm>two_RHD2132'
- Handle OpenEphys NPIX with multiple probes
- Add cross-checked ASSY-116>RHD2132 mapping

1.18.17 probeinterface 0.2.9

April, 15th 2022

- openephys neuropixel
- fix examples

1.18.18 probeinterface 0.2.8

March, 23rd 2022

- wiring CambridgeNeurotec mini-amp-64
- expose function select_dimensions (2d>3d and 3d>2d)
- add to_dict/from_dict in ProbeGroup
- Add "text_on_contact" in plot_probe()
- Add read_openephys function for Neuropux-PXI plugin

1.18.19 probeinterface 0.2.7

March, 1 2022

- add read_3brain to io
- annotate spikeGLX with probe version

1.18.20 probeinterface 0.2.6

November, 26 2021

- documention improvement
- spikeglx neuropixel2 integration
- plotting improvement

1.18.21 probeinterface 0.2.5

September, 14 2021

- vector annotations added to numpy representation
- add “electrode” to annotations from read_maxwell

1.18.22 probeinterface 0.2.4

July, 30 2021

- expose read_maxwell function
- vector annotations
- changes to BIDS format

1.18.23 probeinterface 0.2.3

May, 21 2021

- add a pathway
- show_channel_on_click
- debug read_mearec()

1.18.24 probeinterface 0.2.2

April, 4 2021

- better plot_probe with index
- write_prb handle group_mode
- add wriring RDH2164
- doc improvement

1.18.25 probeinterface 0.2.1

March, 24 2021

- read/write to BIDS proposal.
- to_numpy()/from_numpy()
- to_dataframe()/from_dataframe()
- read_mearec

1.18.26 probeinterface 0.2.0

March, 2 2021

Format improvement with all ids in str.

1.18.27 probeinterface 0.1.0

11th jan 2021

Initial release.

PYTHON MODULE INDEX

p

probeinterface, 57
probeinterface.generator, 62
probeinterface.io, 60
probeinterface.library, 65
probeinterface.plotting, 64

INDEX

A

add_probe() (*probeinterface.ProbeGroup method*), 57
add_probe_to_zarr_group() (*probeinterface.Probe method*), 50
annotate() (*probeinterface.Probe method*), 50
annotate_contacts() (*probeinterface.Probe method*), 51
auto_generate_contact_ids() (*probeinterface.ProbeGroup method*), 58
auto_generate_probe_ids() (*probeinterface.ProbeGroup method*), 58

C

contact_positions (*probeinterface.Probe property*), 51
copy() (*probeinterface.Probe method*), 51
create_auto_shape() (*probeinterface.Probe method*), 51

F

from_dataframe() (*probeinterface.Probe static method*), 51
from_dict() (*probeinterface.Probe static method*), 52
from_dict() (*probeinterface.ProbeGroup static method*), 58
from_numpy() (*probeinterface.Probe static method*), 52
from_numpy() (*probeinterface.ProbeGroup static method*), 58
from_zarr() (*probeinterface.Probe static method*), 52
from_zarr_group() (*probeinterface.Probe static method*), 52

G

generate_dummy_probe() (*in module probeinterface.generator*), 62
generate_dummy_probe_group() (*in module probeinterface.generator*), 62
generate_linear_probe() (*in module probeinterface.generator*), 63
generate_multi_columns_probe() (*in module probeinterface.generator*), 63
generate_tetrode() (*in module probeinterface.generator*), 62
get_contact_count() (*probeinterface.Probe method*), 53
get_contact_count() (*probeinterface.ProbeGroup method*), 58
get_contact_vertices() (*probeinterface.Probe method*), 53
get_global_contact_ids() (*probeinterface.ProbeGroup method*), 59
get_global_device_channel_indices() (*probeinterface.ProbeGroup method*), 59
get_probe() (*in module probeinterface.library*), 65
get_shank_count() (*probeinterface.Probe method*), 53
get_shanks() (*probeinterface.Probe method*), 53
get_slice() (*probeinterface.Probe method*), 53

M

module
    probeinterface, 50, 57
    probeinterface.generator, 62
    probeinterface.io, 60
    probeinterface.library, 65
    probeinterface.plotting, 64
move() (*probeinterface.Probe method*), 53

P

plot_probe() (*in module probeinterface.plotting*), 64
plot_probe_group() (*in module probeinterface.plotting*), 65
Probe (*class in probeinterface*), 50
ProbeGroup (*class in probeinterface*), 57
probeinterface
    module, 50, 57
probeinterface.generator
    module, 62
probeinterface.io
    module, 60
probeinterface.library
    module, 65
probeinterface.plotting
    module, 64

R

read_csv() (*in module probeinterface.io*), 61
read_mearec() (*in module probeinterface.io*), 62
read_nwb() (*in module probeinterface.io*), 62
read_prb() (*in module probeinterface.io*), 60
read_probeinterface() (*in module probeinterface.io*), 60
rotate() (*probeinterface.Probe method*), 53
rotate_contacts() (*probeinterface.Probe method*), 53

S

set_contact_ids() (*probeinterface.Probe method*), 54
set_contacts() (*probeinterface.Probe method*), 54
set_device_channel_indices() (*probeinterface.Probe method*), 54
set_global_device_channel_indices() (*probeinterface.ProbeGroup method*), 59
set_planar_contour() (*probeinterface.Probe method*), 54
set_shank_ids() (*probeinterface.Probe method*), 55

T

to_2d() (*probeinterface.Probe method*), 55
to_3d() (*probeinterface.Probe method*), 55
to_dataframe() (*probeinterface.Probe method*), 55
to_dataframe() (*probeinterface.ProbeGroup method*), 59
to_dict() (*probeinterface.Probe method*), 55
to_dict() (*probeinterface.ProbeGroup method*), 59
to_image() (*probeinterface.Probe method*), 56
to_numpy() (*probeinterface.Probe method*), 56
to_numpy() (*probeinterface.ProbeGroup method*), 60
to_zarr() (*probeinterface.Probe method*), 57

W

wiring_to_device() (*probeinterface.Probe method*), 57
write_csv() (*in module probeinterface.io*), 62
write_prb() (*in module probeinterface.io*), 61
write_probeinterface() (*in module probeinterface.io*), 60