probeinterface

Samuel Garcia

CONTENTS

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

The package handles the following items:

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

The `probeinterface` package also provide:

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

Here a schema for the naming used in the package:

orphan
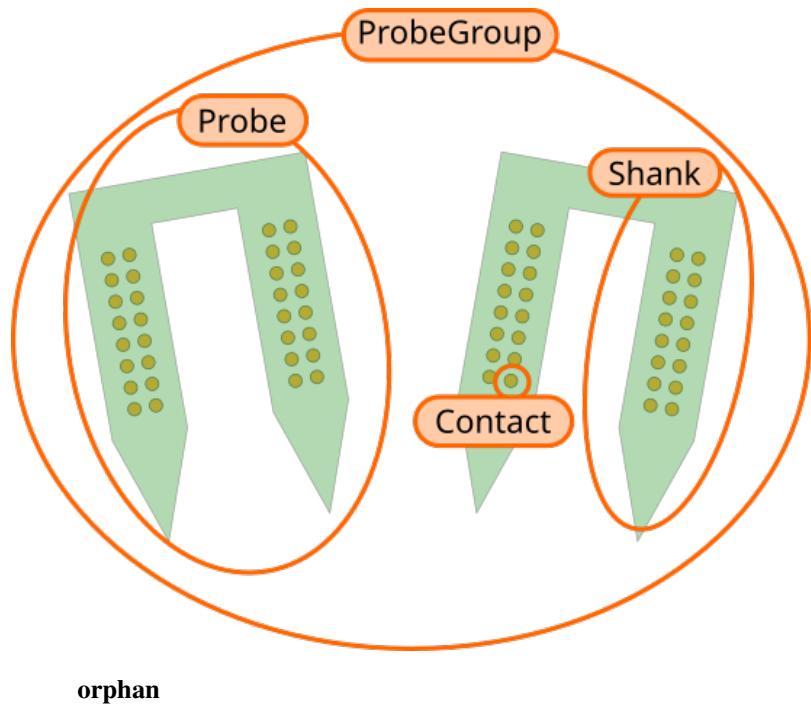
EXAMPLES

Start here with a tutorial showing probeinterface.

1.1 Generate a Probe from scratch

This example generates a probe from scratch.

```python
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 32-contact probe

```python
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 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.

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

Probe objects have fancy prints!

```python
print(probe)
```

Out:

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

In addition to contacts, we can crate 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)
```


Out:

```
(<matplotlib.collections.PolyCollection object at 0x7fff8a4043a0>, <matplotlib.
→collections.PolyCollection object at 0x7fff8a407ee0>)
```
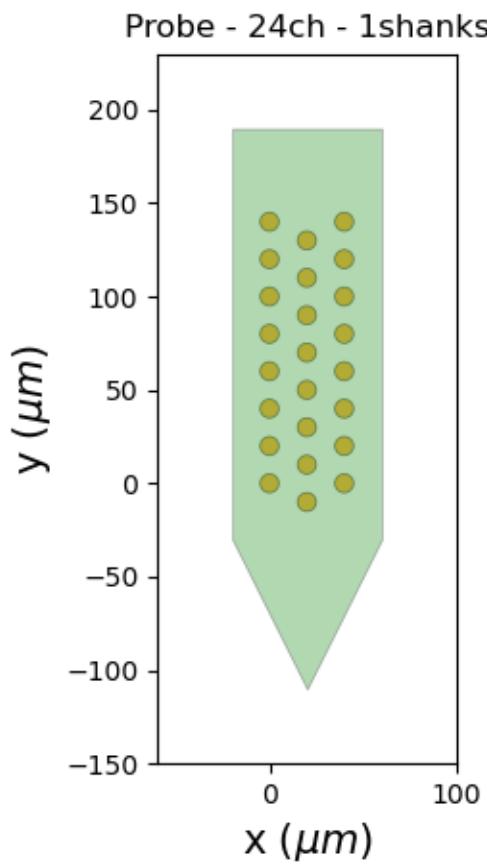
A 2d *Probe* can be transformed to a 3d *Probe* by indicating the *plane* on which contacts will lie (Here the 'y' coordinate will be 0 for all contacts):

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

```
plt.show()
```

Probe - 24ch - 1shanks
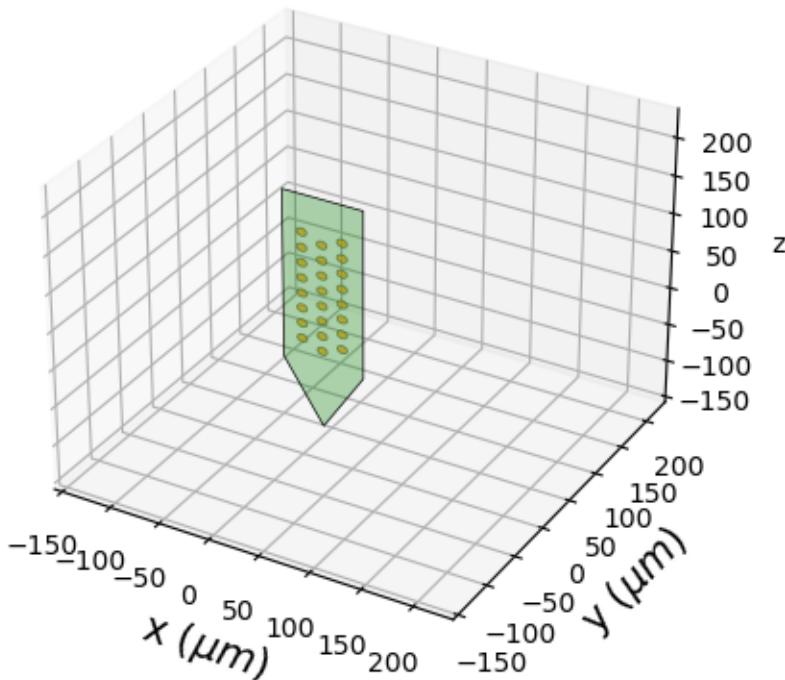
Total running time of the script: (0 minutes 0.924 seconds)

1.2 2d and 3d Probes

This example shows how to manipulate the 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 32 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

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 plane is '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(plane='xz')
print(probe_2d.contact_positions.shape)
print(probe_3d.contact_positions.shape)
```

Out:

```
(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)
```

Probe - 24ch - 1shanks

Out:
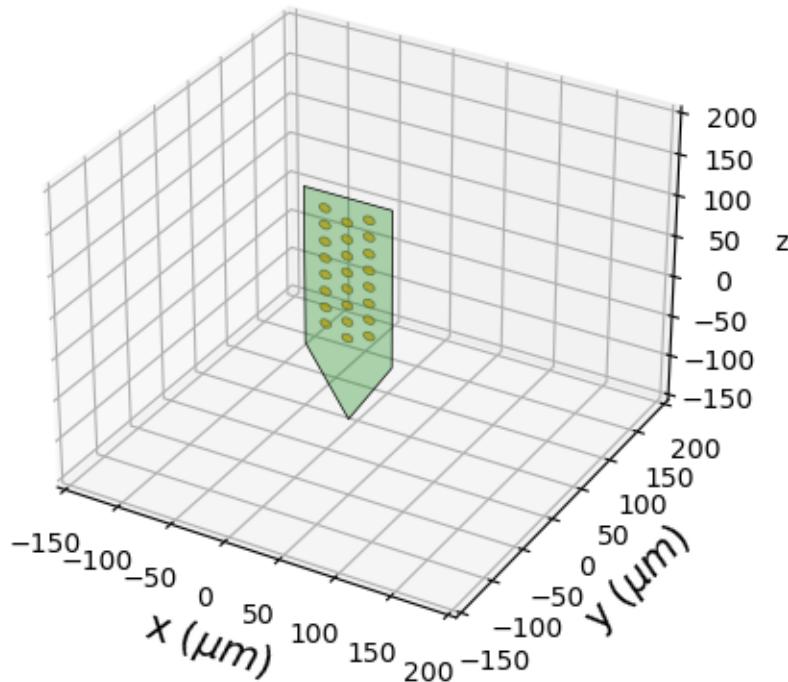
```
(<mpl_toolkits.mplot3d.art3d.Poly3DCollection object at 0x7fff8a4d23e0>, <mpl_toolkits.
→mplot3d.art3d.Poly3DCollection object at 0x7fff8a4d34f0>)
```

We can create another probe lying on another plane:

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

Probe - 24ch - 1shanks

Out:
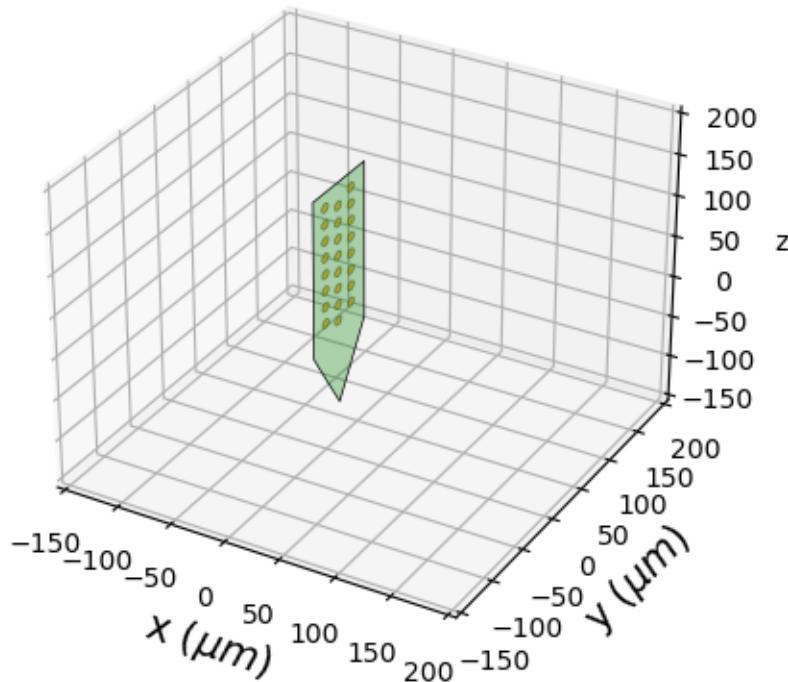
```
(<mpl_toolkits.mplot3d.art3d.Poly3DCollection object at 0x7ffef93bce50>, <mpl_toolkits.
→mplot3d.art3d.Poly3DCollection object at 0x7ffef93bd780>)
```

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()
```

Probe - 24ch - 1shanks
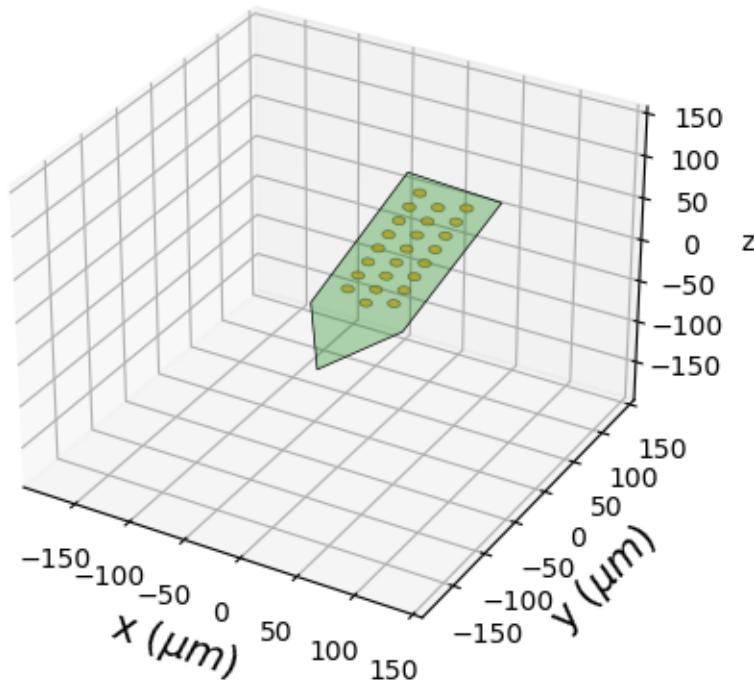
Total running time of the script: (0 minutes 0.451 seconds)

1.3 Generate a ProbeGroup

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

Import

```python
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:

```python
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_channel_count()', probegroup.get_channel_count())
```

Out:

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

We can now plot all probes in the same axis:

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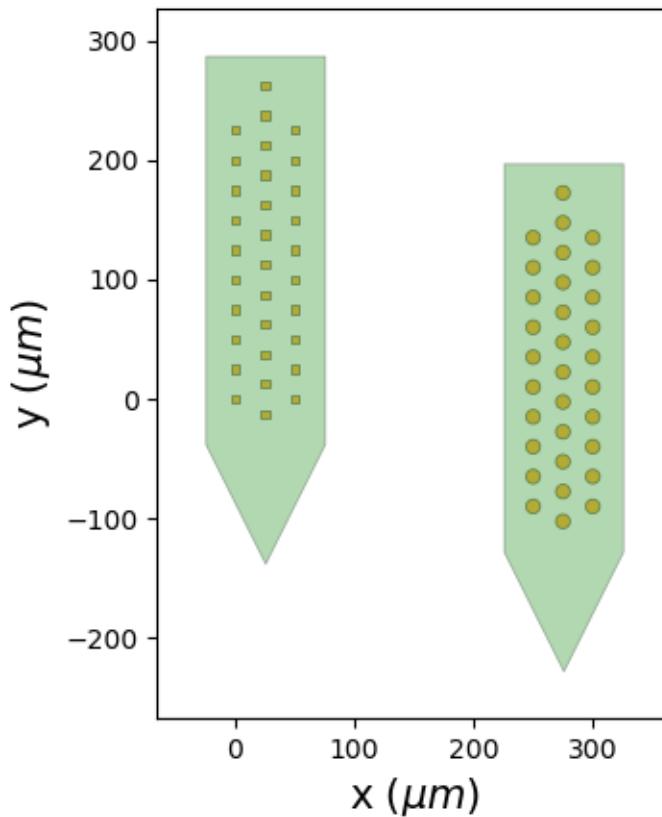

or in separate axes:

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

plt.show()
```
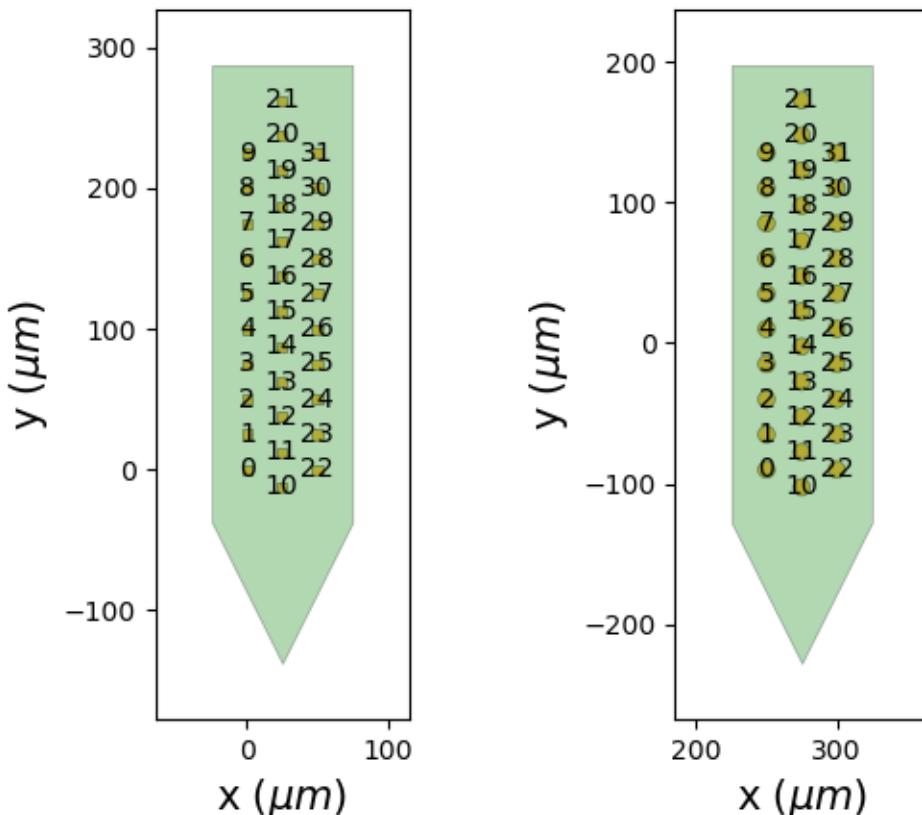
Total running time of the script: (0 minutes 0.327 seconds)

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

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

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)
```


Out:
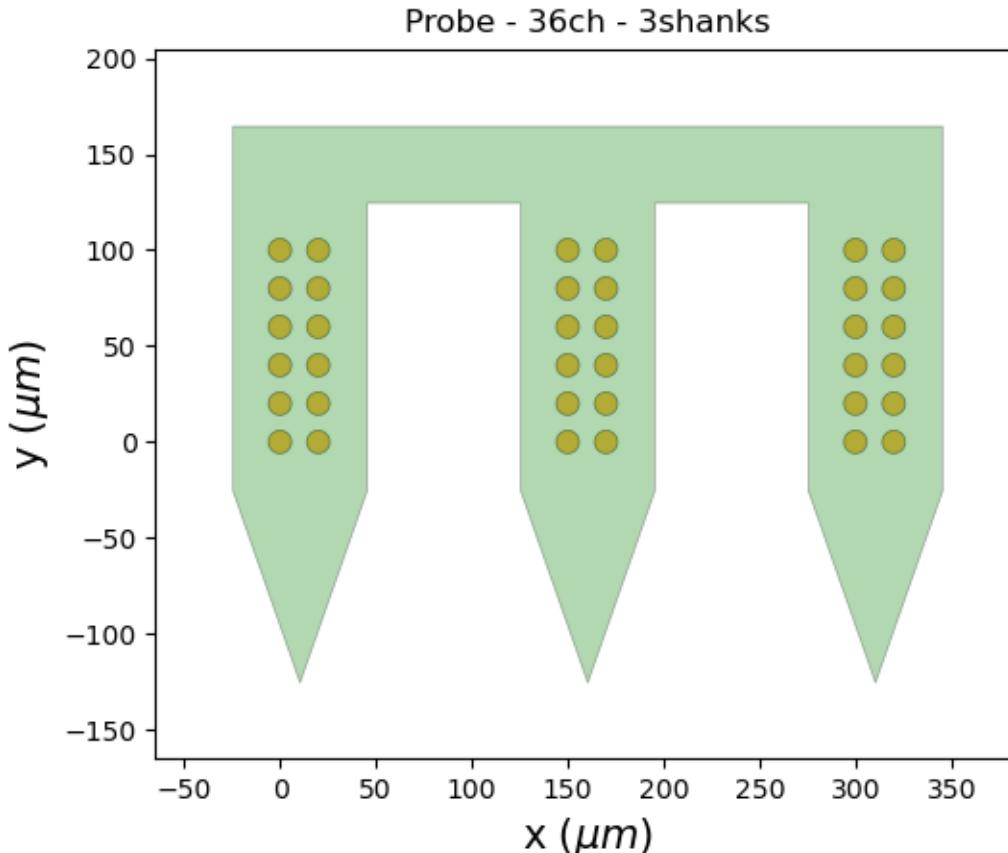
```
(<matplotlib.collections.PolyCollection object at 0x7ffef92d0160>, <matplotlib.
→collections.PolyCollection object at 0x7fff8a4d00d0>)
```

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

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

Out:

```
['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 link to a *Probe* object and can also retrieve positions, contact shapes, etc.:

```python
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)
```

Out:

```
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 then into a single *Probe* object

```python
# 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])
```

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

Out:

```
['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']
```

```python
plot_probe(multi_shank)

plt.show()
```
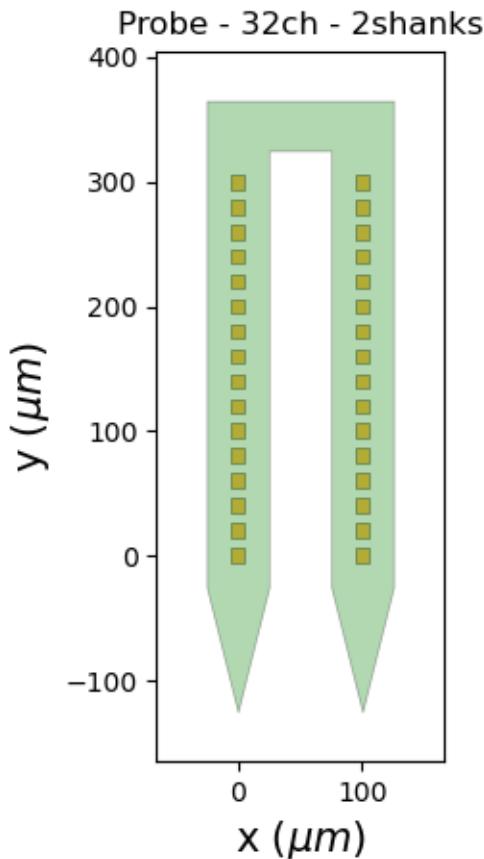
Total running time of the script: (0 minutes 0.241 seconds)

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

```python
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],
                                     xpitch=75, ypitch=75, y_shift_per_column=[0, -37.5,
→0],
                                     contact_shapes='circle', contact_shape_params={
→'radius': 12})

plot_probe(probe, with_channel_index=True)
```


Out:

```
(<matplotlib.collections.PolyCollection object at 0x7ffef9370640>, <matplotlib.
→collections.PolyCollection object at 0x7ffef9210310>)
```
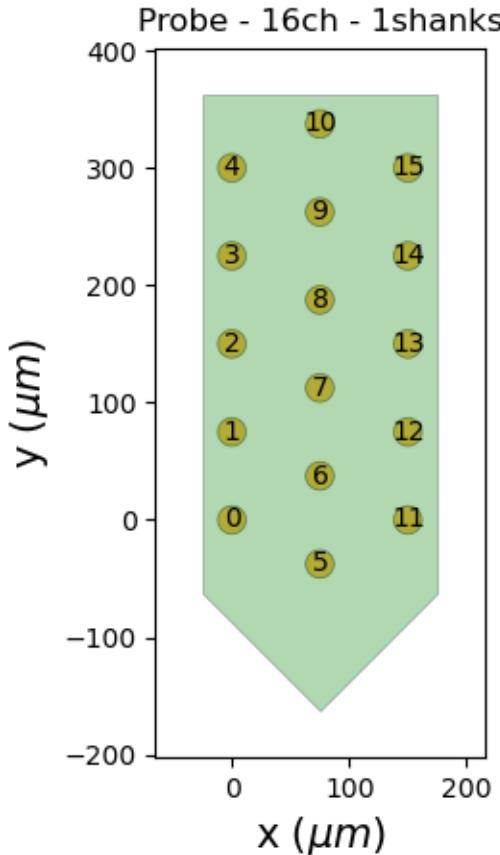
The Probe is not connected to any device yet:

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

Out:

```
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 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)
```

Out:

```
[ 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_channel_index=True, with_device_index=True)
```


Out:
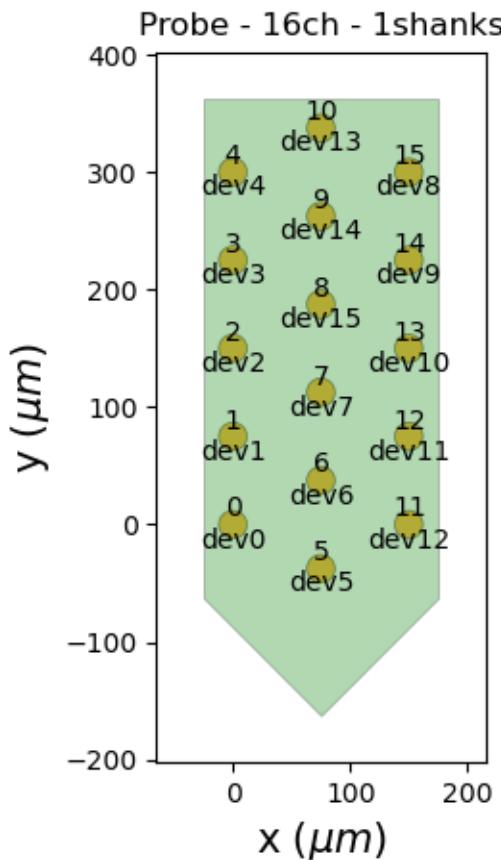
```
(<matplotlib.collections.PolyCollection object at 0x7fff8a43ee90>, <matplotlib.
→collections.PolyCollection object at 0x7fff8a4058a0>)
```

Very often we have several probes on the device and this can lead to even more complex channel indices. *Probe-Group.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()

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())
```

Out:

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

The indices of the probe group can also be plotted:

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

plt.show()
```
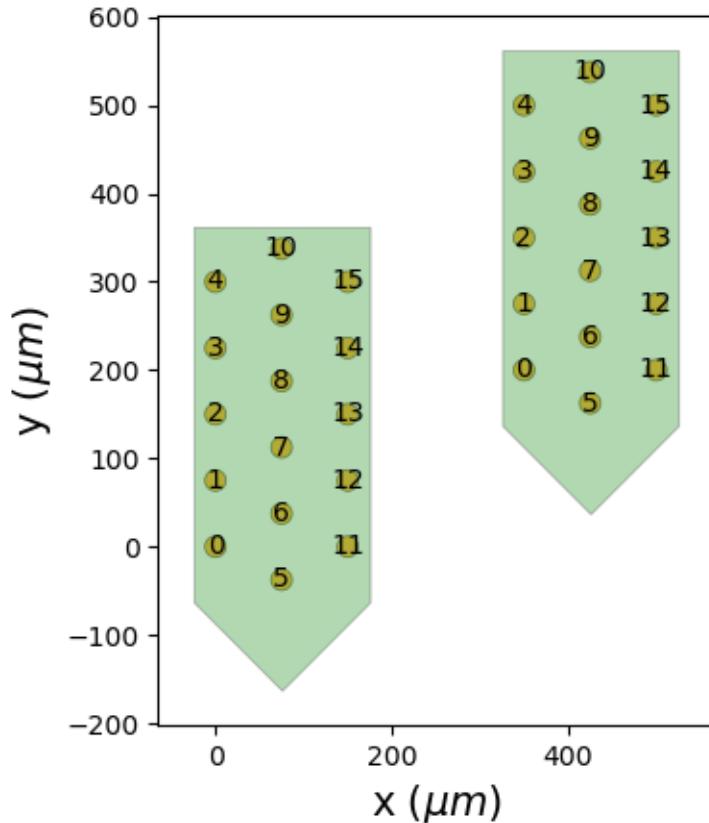
Total running time of the script: (0 minutes 0.383 seconds)

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 theses formats:

* PRB (.prb) : used by klusta/spyking-circus/tridesclous

* CVS (.csv): 2 or 3 columns locations in text file

* mearec (.h5) : mearec handle the geometry

* spikeglx (.meta) : spikeglx handle the handle also the geometry

Import

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


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

```python
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

```python
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:

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

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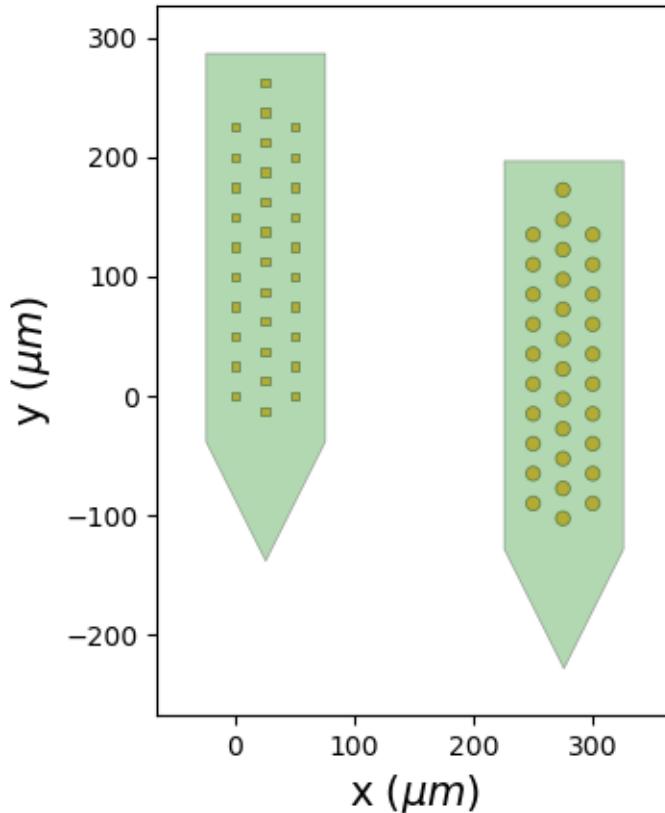

The format looks like this:

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

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

Out:

```
{
    "specification": "probeinterface",
    "version": "0.2.7",
    "probes": [
        {
            "ndim": 2,
            "si_units": "um",
            "annotations": {
                "name": "",
                "manufacturer": "me"
            },
            "contact_annotations": {
                "quality": [
                    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 an example .prb file with 2 channel groups of 4 channels each. It can be easily loaded and plotted with *probeinterface*

```python
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_channel_index=True)

plt.show()
```
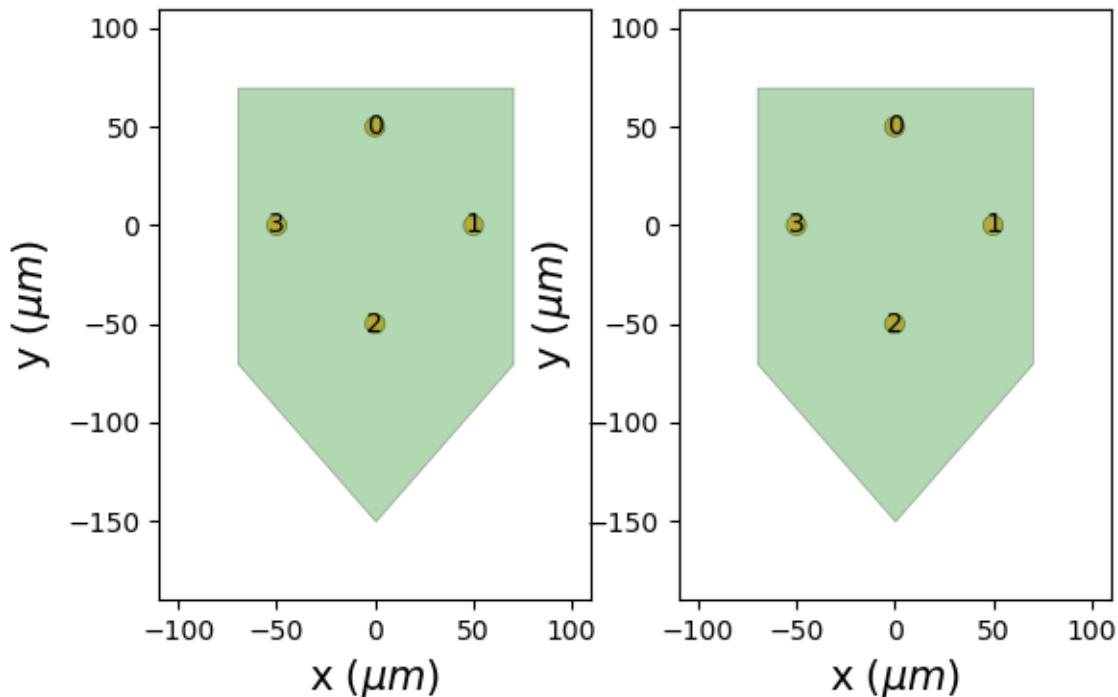

Total running time of the script: (0 minutes 0.273 seconds)

1.7 Probe generator

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

- tetrodes

- linear probes

- multi-column probes

Import

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

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

Generate 4 tetrodes:

```python
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_channel_index=True, same_axes=True)
```
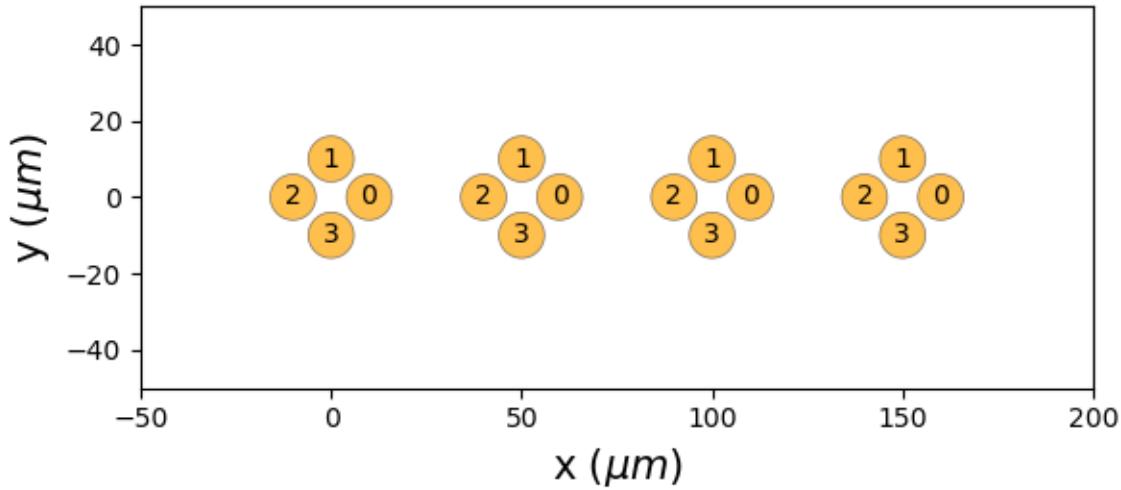
Generate a linear probe:

```
from probeinterface import generate_linear_probe

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

Out:

```
(<matplotlib.collections.PolyCollection object at 0x7ffef92d0d00>, <matplotlib.
↪collections.PolyCollection object at 0x7ffef9372560>)
```
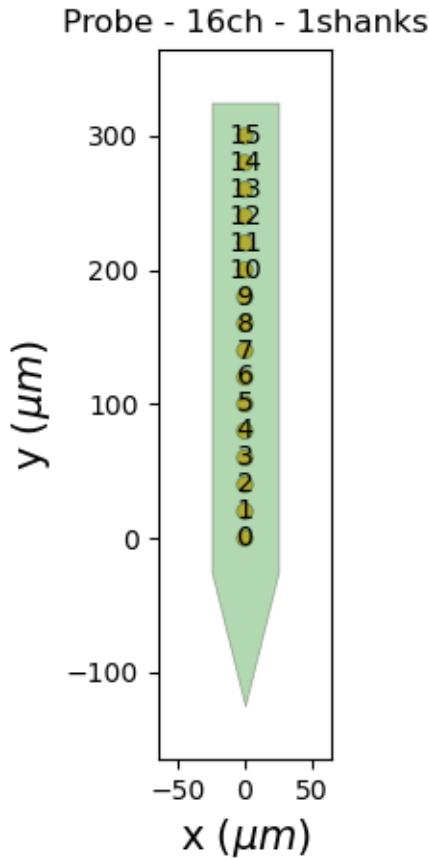
Generate a multi-column probe:

```python
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})
plot_probe(multi_columns, with_channel_index=True, )
```

Probe - 32ch - 1shanks

Out:
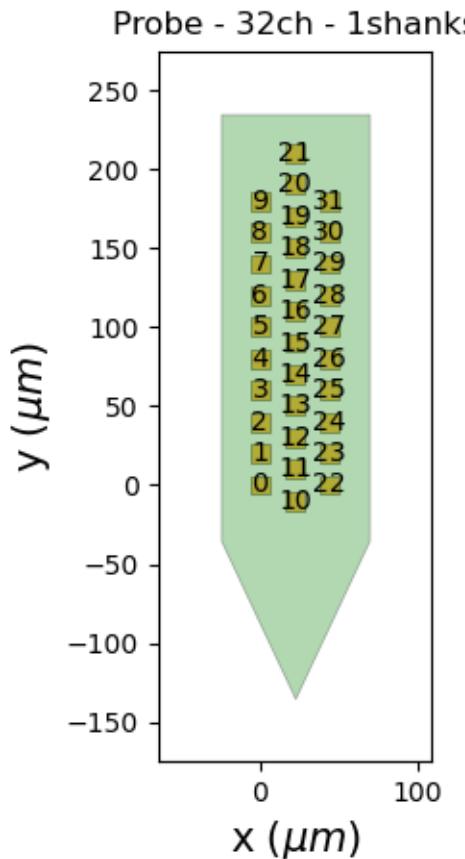
```
(<matplotlib.collections.PolyCollection object at 0x7ffef927e3e0>, <matplotlib.
↪collections.PolyCollection object at 0x7fff8a4bb3d0>)
```

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()
```

Probe - 144ch - 1shanks
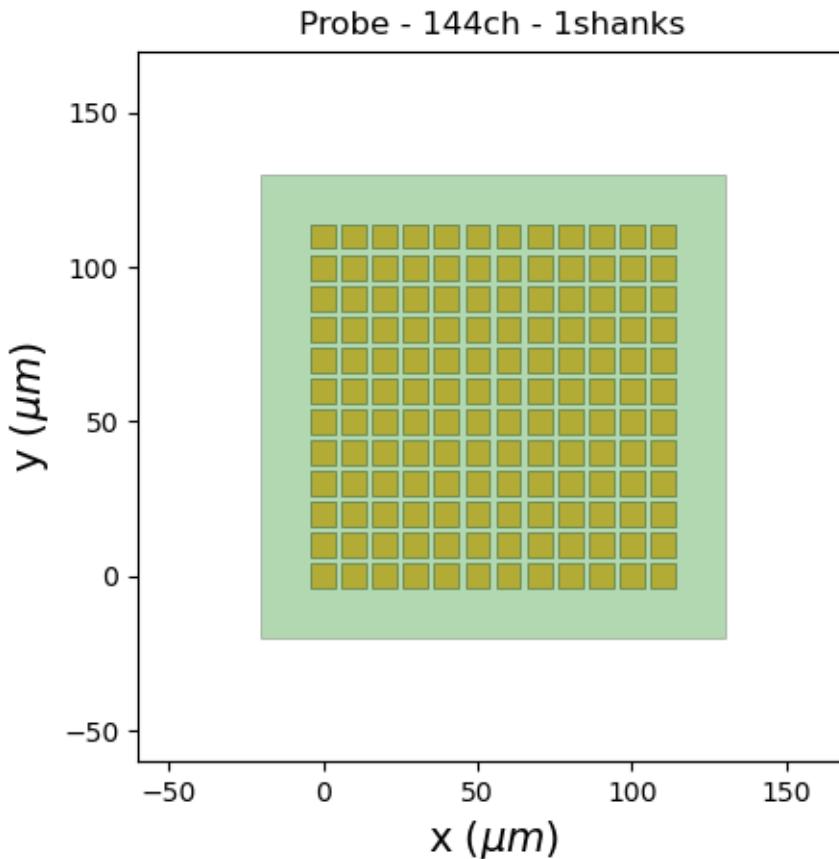
Total running time of the script: (0 minutes 0.474 seconds)

1.8 More plotting examples

Here some examples to showcase several plotting scenarios.

Import

```python
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

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

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

# make some colors for each probe
```

```python
probe1 = generate_linear_probe(num_elec=9)
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')
```

Probe - 4ch - 1shanks
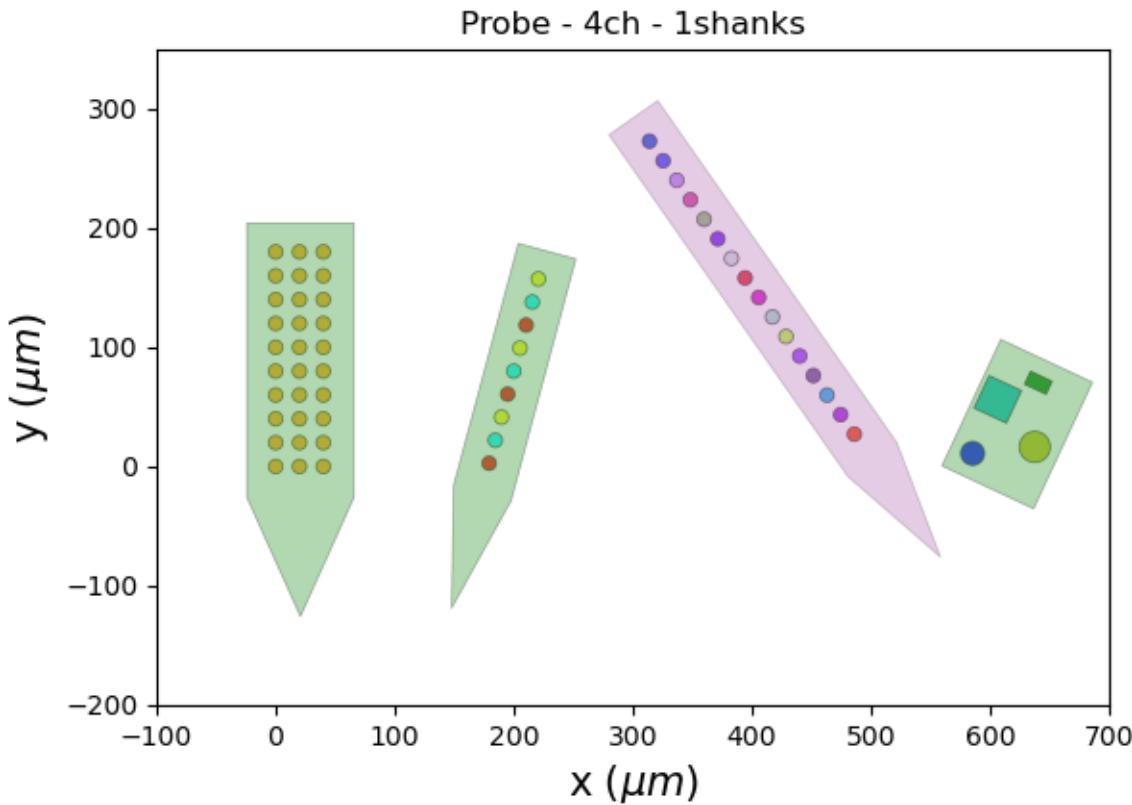
Some example in 3d for romantic who like flowers. . .

```python
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
```

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

plt.show()
```


Probe - 24ch - 1shanks
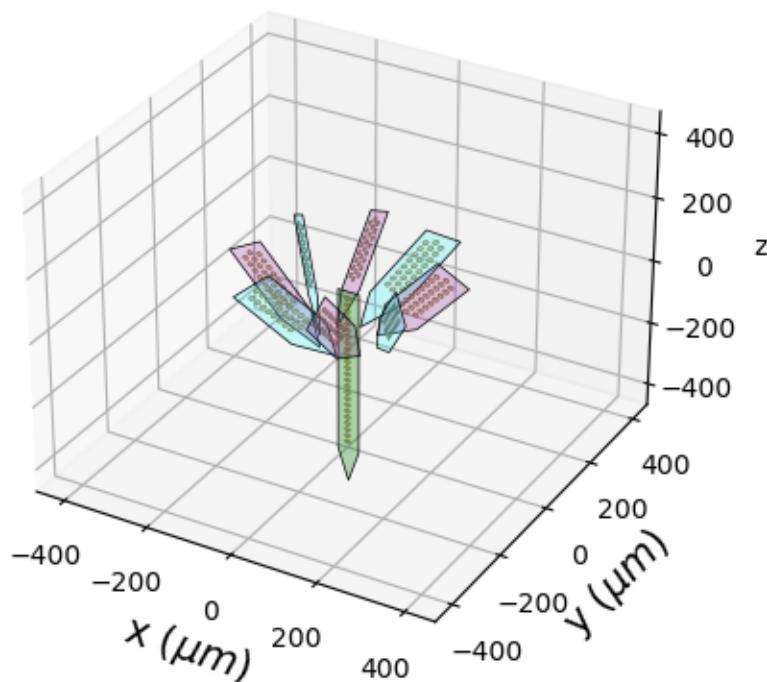
Total running time of the script: (0 minutes 0.418 seconds)

1.9 More complicated probes

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

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

from probeinterface import Probe
from probeinterface.plotting import 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)
```

Probe - 24ch - 1shanks

Out:

```
(<matplotlib.collections.PolyCollection object at 0x7fff8a4d3d60>, <matplotlib.
→collections.PolyCollection object at 0x7fff8a4baad0>)
```
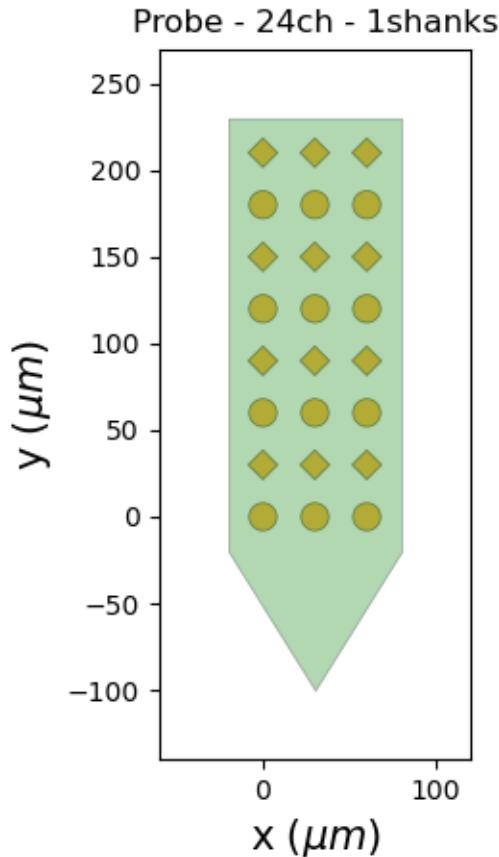
We can also use the *rotate_contacts* to make contact-wise rotations:

```python
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)
```

1.9. More complicated probes   **31**

Probe - 24ch - 1shanks

Out:
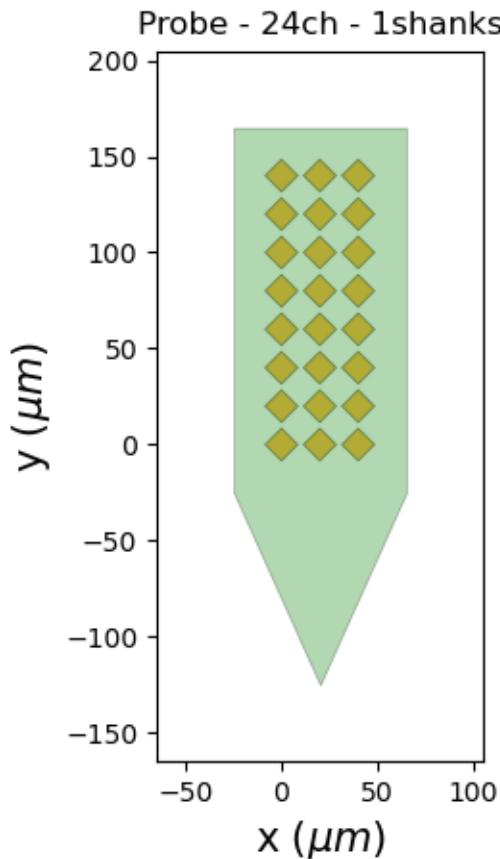
```
(<matplotlib.collections.PolyCollection object at 0x7ffef93bf280>, <matplotlib.
↪collections.PolyCollection object at 0x7ffef92108e0>)
```

```
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()
```

Probe - 25ch - 1shanks
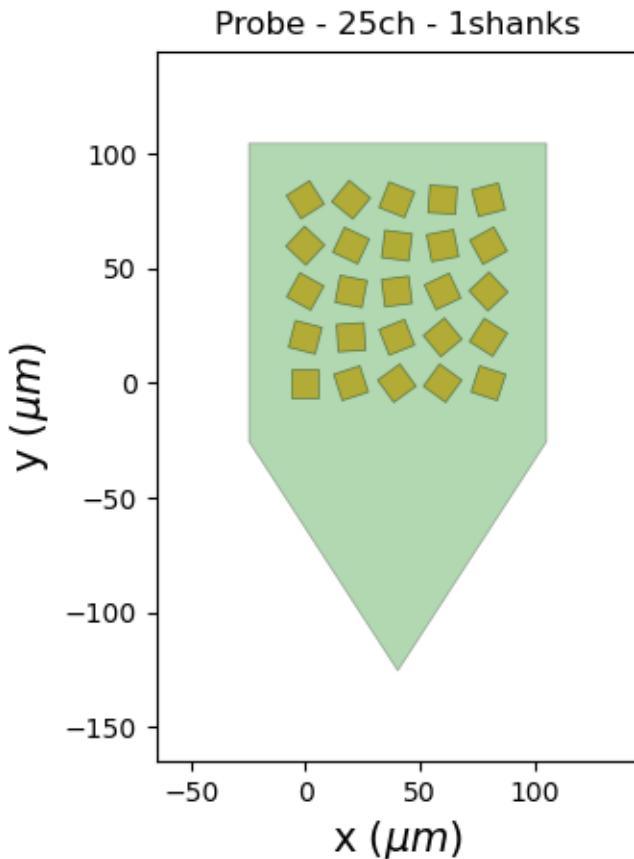
Total running time of the script: (0 minutes 0.301 seconds)

1.10 Get probe from library

probeinterface provides a library of probes from several manufacturers on the gin platform: https://gin.g-node.org/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.

```python
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:

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

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

Out:

```
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)
```

Out:

```
{'first_index': 1,
 'manufacturer': 'neuronexus',
 '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_channel_index=True)
```

Out:
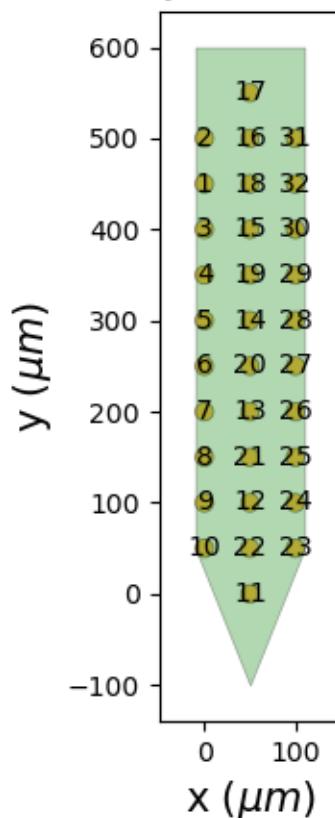
```
(<matplotlib.collections.PolyCollection object at 0x7fff8a4bb610>, <matplotlib.
→collections.PolyCollection object at 0x7ffef93bcc10>)
```

```
plt.show()
```

Total running time of the script: (0 minutes 0.146 seconds)

1.11 Automatic wiring

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

```python
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:

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

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

Out:

```
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 this 2 wiring documentation https://neuronexus.com/wp-content/uploads/2018/09/Wiring_H32.pdf http://intantech.com/RHD_headstages.html?tabSelect=RHD32ch&yPos=0

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

```python
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 wiring based on standard connectors. In our case, we can simply use:

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

Out:

```
[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_channel_index=True, with_device_index=True, ax=ax)


plt.show()
```

neuronexus - A1x32-Poly3-10mm-50-177 - 32ch - 1shanks
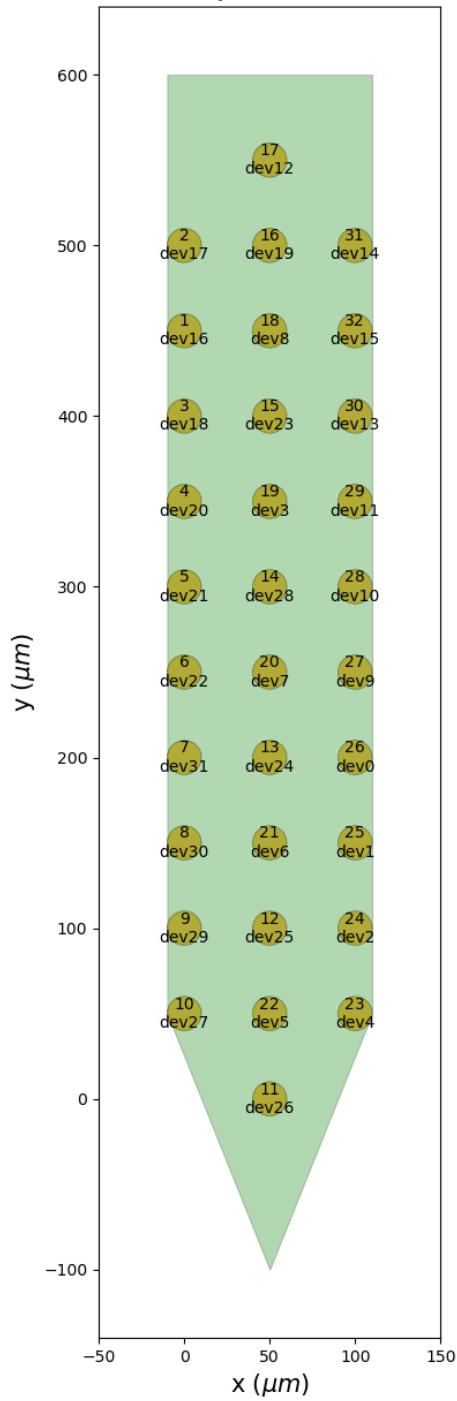
Total running time of the script: (0 minutes 0.210 seconds)

1.12 Plot values

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

```python
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:

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

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

fake values

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

plot with value

```python
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)
```
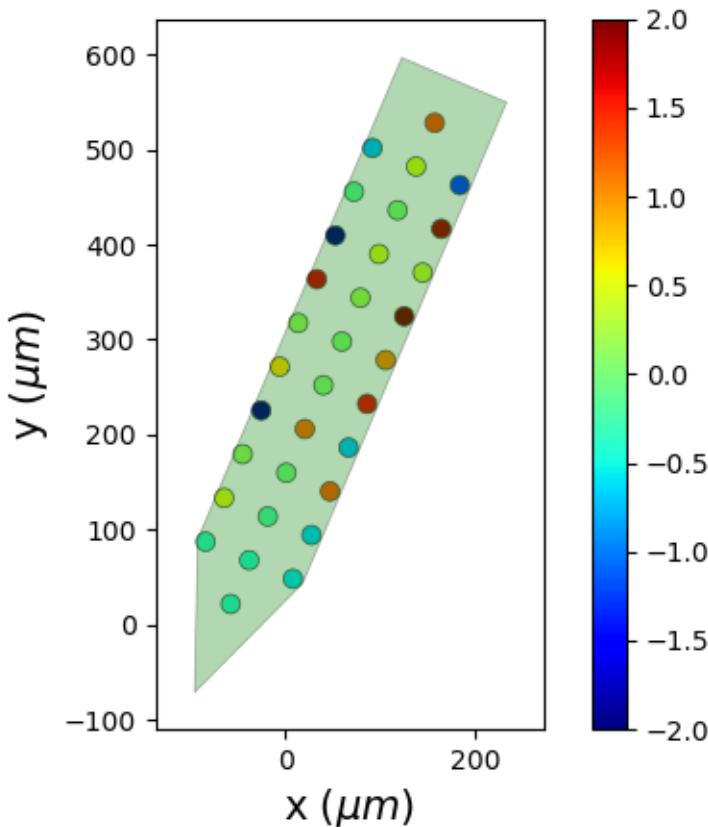
Out:

```
<matplotlib.colorbar.Colorbar object at 0x7ffef92542b0>
```

generated an interpolated image and plot it on top

```python
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)
```
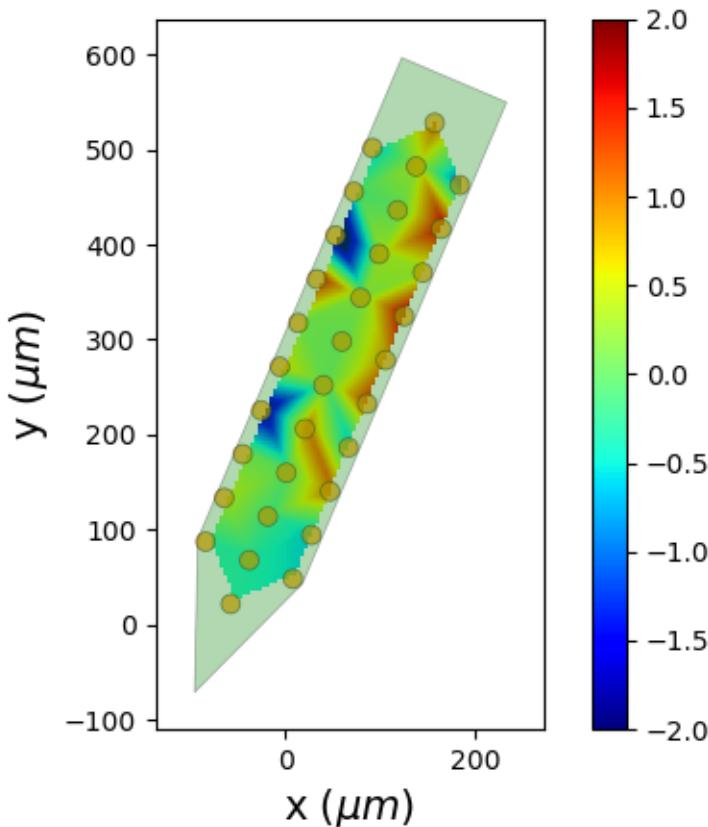
Out:

```
(127, 67)

<matplotlib.colorbar.Colorbar object at 0x7ffef78dc130>
```

works with several interpolation method

```python
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()
```
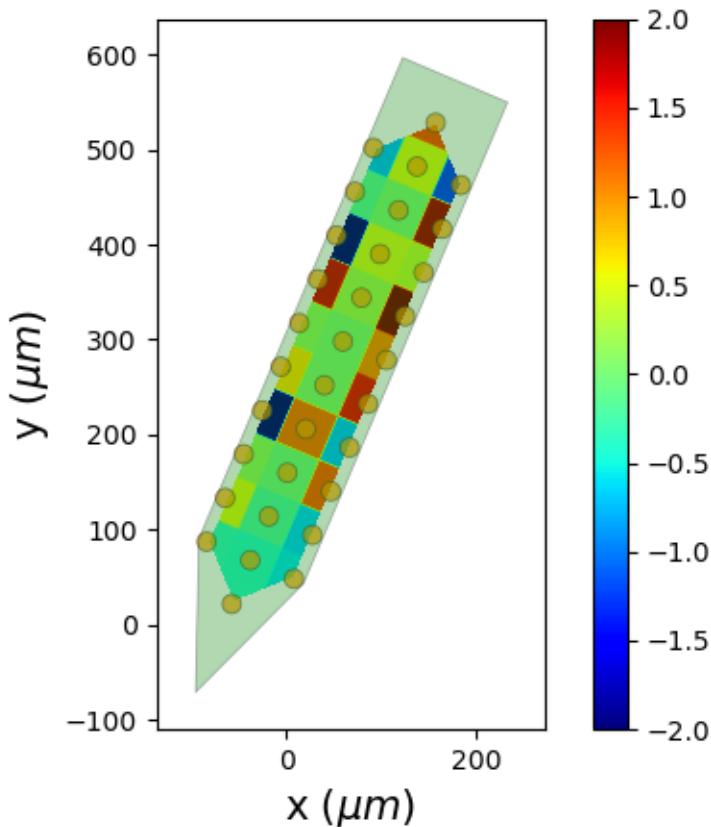
Total running time of the script: (0 minutes 2.036 seconds)

1.13 Overview

1.13.1 Introduction

To record neural electrical signals, extracellular neural probes are inserted in the brain. Neural probes are (usually) multi-channel arrays able to record from multiple contacts simultaneously, spanning from a few channels (e.g. tetrodes) to high-dense silicon probes (e.g. Neuropixels - with up to 384 recorded 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 to directly link the **physical contacts on the probe** and the **logical channel index on the device**.

Recent *spike sorting* (i.e. methods to extract single neurons' activity from the extracellular recordings) algorithms mainly 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 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 Open Ephys board**

What would be your final channel mapping?

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 step 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 contour (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 existing format (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

- SpiKING Circus: handle probe with this package

- Kilosort: handle probe with this package

- Tridesclous: 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
- IMEC
- 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 combinade 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 JSON format. The format is a trivial json-serialisation of a Python dictionary. The dictionary maps every attributes 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 recordingdevice.

Here 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)

homemade - 2 shank tetrodes - 8ch
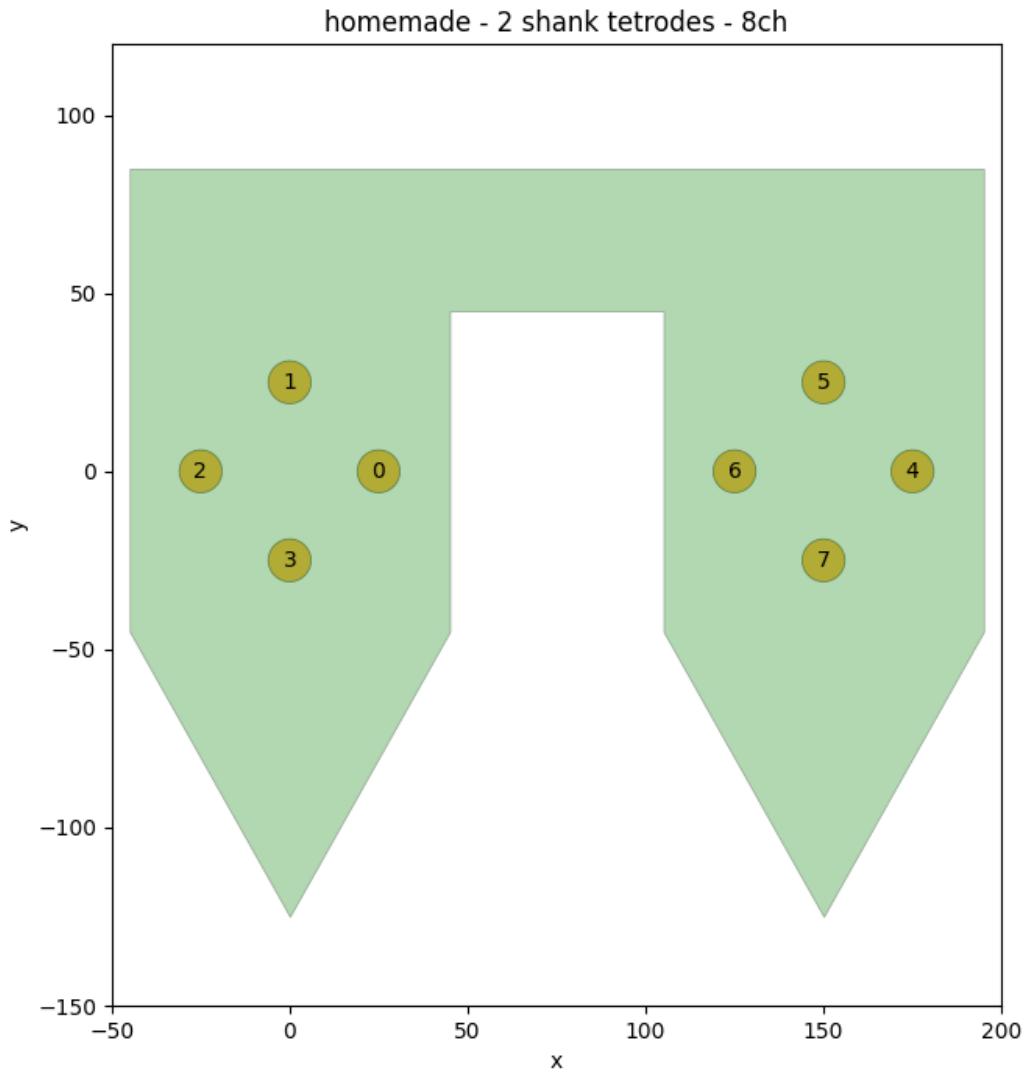
The first part contains fields about the `probeinterface` version and a list of probes:

```
{
  "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

The full json file looks as follows:

```
{
    "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
            ],
            [
                -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,
                        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
            ]
        ]
    ],
    "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
                ],
                [
                    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 handle a collection of probe description on the gin platform

The python module have simple function to download and chache locally *get_probe(...)*

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

The gin platform is a github like platform that make possible to handle "big files" with git annex. So user contribution in gin is as easy as a standard github pull request.

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

1.16.1 How to contribute

TODO: exclain with more details

1. **Genertae the JSON file with probeinterface function (or directly** with another language)

2. Generate animage with *plot_probe*

3. Clone with gin client the probeinterface_library repo

4. Put files at the good place.

5. Ask for an account

6. Push to a branch with gin client

7. Make a pull request on the gin portal (like a github PR)

1.17 API

1.17.1 Probe

class probeinterface.**Probe**(*ndim=2*, *si_units='um'*)
    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.

    annotate(***kwargs*)
        Annotates the probe object.

        **kwargs : list of kwyword arguments to add to the annotations

    copy()
        Copy to another Probe instance.

        Note: device_channel_indices is not copied and contact_ids is not copied

---

create_auto_shape(*probe_type='tip'*, *margin=20.0*)
Create planar contour automatically based on probe contact positions.

probe_type [str, optional] The probe type ('tip' or 'rect'), by default 'tip'

margin [float, optional] The margin to add to the contact positions, by default 20

static from_dataframe(*df*)
Create Probe from a pandas.DataFrame see Probe.to_dataframe()

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

probe [Probe] The instantiated Probe object

static from_dict(*d*)
Instantiate a Probe from a dictionary

d [dict] The dictionary representation of the probe

probe [Probe] The instantiated Probe object

static from_numpy(*arr*)
Create Probe from a complex numpy array see Probe.to_numpy()

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

probe [Probe] The instantiated Probe object

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

get_contact_vertices()
Return a list of contact vertices.

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

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

get_slice(*selection*)
Get a copy of the Probe with a sub selection of contacts.

Selection can be boolean or by index

selection : np.array of bool or int (for index)

move(*translation_vector*)
Translate the probe in one direction.

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

rotate(*theta*, *center=None*, *axis=None*)
Rotate the probe around a specified axis.

theta [float] In degrees, anticlockwise.

center [array] Center of rotation. If None, the center of probe is used

axis [None] Axis of rotation. It must be None for 2D probes and specified for 3D ones

rotate_contacts(*thetas*)
Rotate each contact of the probe. Internaly, it modifies the contact_plane_axes.

thetas [array of float] Rotation angle in degree. If scalar, then it is applied to all contacts.

set_contact_ids(*contact_ids*)

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

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*, *shank_ids=None*)

Sets contacts to a Probe.

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

shapes [str or array] Shape of each contact ('circle'/'square'/'rect').

shape_params [dict or list of dict] Contains kwargs for shapes ("radius" for circle, "width" for square, "width/height" for rect)

plane_axes [(num_contacts, 2, ndim)] Defines the axes of the contact plane (2d or 3d)

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

set_device_channel_indices(*channel_indices*)

Manually set the device channel indices.

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

channel_indices [array of int] The device channel indices to set

set_planar_contour(*contour_polygon*)

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

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

set_shank_ids(*shank_ids*)

Set shank ids.

shank_ids [list or array] Array with shank ids

to_3d(*plane='xz'*)

Transform 2d probe to 3d probe.

Note: device_channel_indices is not copied.

plane [str] The plane on which the 2D probe is defined. 'xy', 'yz ', xz'

to_dataframe(*complete=False*)

Export the probe to a pandas dataframe

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

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

to_dict(*array_as_list=False*)

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

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

d [dict] The dictionary representation of the probe

to_image(*values*, *pixel_size=0.5*, *num_pixel=None*, *method='linear'*, *xlims=None*, *ylims=None*)

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

> **values :** vector same size as contact number to be color plotted

> **pixel_size :** size of one pixel in micrometers

> **num_pixel :** alternative to pixel_size give pixel number of the image width

> method : 'linear' or 'nearest' or 'cubic' xlims : tuple or None

>> Force image xlims

> **ylims** [tuple or None] Force image ylims

> **image** [2d array] The generated image

> **xlims** [tuple] The x limits

> **ylims** [tuple] The y limits

to_numpy(*complete=False*)
> Export to a numpy vector (structured array). This vector handles all contact attributes.

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

> Very useful to export/slice/attach to a recording.

> **complete** [bool] If True, export complete information about the probe, including contact_plane_axes/si_units/device_channel_indices (default False)

> **arr** [numpy.array] With complex dtype

wiring_to_device(*pathway*, *channel_offset=0*)
> Automatically set device_channel_indices based on a pathway.

> See probeinterface.get_available_pathways()

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

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*)

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

>> *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.

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

>>> *probeinterface.utils.generate_unique_ids*

> **get_channel_count**()
>> Total number of channels.

> **get_global_contact_ids**()
>> get all contact ids concatenated across probes

> **get_global_device_channel_indices**()
>> return a numpy array vector with 2 columns (probe_index, device_channel_indices)

>> **Note:** channel -1 means not connected

> **set_global_device_channel_indices**(*channels*)
>> Set global indices for all probes

> **to_numpy**(*complete=False*)
>> Export all probes into a numpy array.

1.17.3 Import/export to formats

Read/write probe info using a variety of formats:

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

probeinterface.io.**read_probeinterface**(*file*)
> Read probeinterface JSON-based format.

> **file: Path or str** The file path

> probegroup : ProbeGroup object

probeinterface.io.**write_probeinterface**(*file*, *probe_or_probegroup*)
> Write a probeinterface JSON file.

> The format handles several probes in one file.

> **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*)
> 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.

> **file** [Path or str] The file path

> probegroup : ProbeGroup object

probeinterface.io.**write_prb**(*file*, *probegroup*, *total_nb_channels=None*, *radius=None*,
>> *group_mode='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

probeinterface.io.**read_csv**(*file*)

> 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_spikeglx**(*file*)

> Read probe position for the meta file generated by SpikeGLX
>
> See http://billkarsh.github.io/SpikeGLX/#metadata-guides for implementation.
>
> The x_pitch/y_pitch/width are set automatically depending the NP version.
>
> The shape is auto generated as a shank.
>
> **Now read:**
>
> > - NP1.0 (=phase3B2)
> >
> > - NP2.0 with 4 shank
>
> **file** [Path or str] The .meta file path
>
> probe : Probe object

probeinterface.io.**read_mearec**(*file*)

> 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.
>
> **file** [Path or str] The file path
>
> 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'*)
> Generate a dummy probe with 3 columns and 32 contacts. Mainly used for testing and examples.
>
> **elec_shapes** [str, optional] Shape of the electrodes, by default 'circle'
>
> **probe** [Probe] The generated probe

probeinterface.generator.**generate_dummy_probe_group**()
> Generate a ProbeGroup with 2 probes. Mainly used for testing and examples.
>
> **probe** [Probe] The generated probe

probeinterface.generator.**generate_tetrode**(*r=10*)
> Generate a tetrode Probe.
>
> **probe** [Probe] The generated probe

probeinterface.generator.**generate_multi_columns_probe**(*num_columns=3, num_contact_per_column=10, xpitch=20, ypitch=20, y_shift_per_column=None, contact_shapes='circle', contact_shape_params={'radius': 6}*)
> Generate a Probe with several columns.
>
> **num_columns** [int, optional] Number of columns, by default 3
>
> **num_contact_per_column** [int, optional] Number of contacts per column, by default 10
>
> **xpitch** [float, optional] Pitch in x direction, by default 20
>
> **ypitch** [float, optional] Pitch in y direction, by default 20
>
> **y_shift_per_column** [float, optional] Shift in y direction per column, by default None
>
> **contact_shapes** [str, optional] Shape of the contacts ('circle', 'rect', 'square'), by default 'circle'
>
> **contact_shape_params** [dict, optional] Parameters for the shape, by default {'radius': 6}
>
> **probe** [Probe] The generated probe

probeinterface.generator.**generate_linear_probe**(*num_elec=16, ypitch=20, contact_shapes='circle', contact_shape_params={'radius': 6}*)
> Generate a one-column linear probe.
>
> **num_elec** [int, optional] Number of electrodes, by default 16
>
> **ypitch** [float, optional] Pitch in y direction, by default 20
>
> **contact_shapes** [str, optional] Shape of the contacts ('circle', 'rect', 'square'), by default 'circle'
>
> **contact_shape_params** [dict, optional] Parameters for the shape, by default {'radius': 6}
>
> **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_channel_index=False*, *with_contact_id=False*, *with_device_index=False*, *first_index='auto'*, *contacts_values=None*, *cmap='viridis'*, *title=True*, *contacts_kargs={}*, *probe_shape_kwargs={}*, *xlims=None*, *ylims=None*, *zlims=None*, *show_channel_on_click=False*)

> Plot a Probe object. Generates a 2D or 3D axis, depending on Probe.ndim
>
> **probe** [Probe] The probe object
>
> **ax** [matplotlib.axis, optional] The axis to plot the probe on. If None, an axis is created, by default None
>
> **contacts_colors** [matplotlib color, optional] The color of the contacts, by default None
>
> **with_channel_index** [bool, optional] If True, channel indices are displayed on top of the channels, by default False
>
> **with_contact_id** [bool, optional] If True, channel ids are displayed on top of the channels, by default False
>
> **with_device_index** [bool, optional] If True, device channel indices are displayed on top of the channels, by default False
>
> **first_index** [str, optional] The first index of the contacts, by default 'auto' (taken from channel ids)
>
> **contacts_values** [np.array, optional] Values to color the contacts with, by default None
>
> **cmap** [str, optional] [description], by default 'viridis'
>
> **title** [bool, optional] If True, the axis title is set to the probe name, by default True
>
> **contacts_kargs** [dict, optional] Dict with kwargs for contacts (e.g. alpha, edgecolor, lw), by default {}
>
> **probe_shape_kwargs** [dict, optional] Dict with kwargs for probe shape (e.g. alpha, edgecolor, lw), by default {}
>
> **xlims** [tuple, optional] Limits for x dimension, by default None
>
> **ylims** [tuple, optional] Limits for y dimension, by default None
>
> **zlims** [tuple, optional] Limits for z dimension, by default None
>
> **show_channel_on_click** [bool, optional] If True, the channel information is shown upon click, by default False
>
> **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=True*, ***kargs*)

> Plot all probes from a ProbeGroup Can be in an existing set of axes or separate axes.
>
> **probegroup** [ProbeGroup] The ProbeGroup to plot
>
> **same_axes** [bool, optional] If True, the probes are plotted on the same axis, by default True

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*, *probe_name*)

> Get probe from ProbeInterface library
>
> **manufacturer** [str] The probe manufacturer (e.g. 'cambridgeneurotech')
>
> **probe_name** [str] The probe name
>
> probe : Probe object

1.18 Release notes

1.18.1 probeinterface 0.2.7

March, 1 2022

- add read_3brain to io
- annotate spikeGLX with probe version

1.18.2 probeinterface 0.2.6

November, 26 2021

- documention improvement
- spikeglx neuropixel2 integration
- plotting improvement

1.18.3 probeinterface 0.2.5

September, 14 2021

- vector annotations added to numpy representation
- add "electrode" to annotations from read_maxwell

1.18.4 probeinterface 0.2.4

July, 30 2021

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

1.18.5 probeinterface 0.2.3

May, 21 2021

- add a pathway
- show_channel_on_click
- debug read_mearec()

1.18.6 probeinterface 0.2.2

April, 4 2021

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

1.18.7 probeinterface 0.2.1

March, 24 2021

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

1.18.8 probeinterface 0.2.0

March, 2 2021

Format improvement with all ids in str.

1.18.9 probeinterface 0.1.0

11th jan 2021

Initial release.

PYTHON MODULE INDEX

p