◆ご執筆に際し、ご注意いただきたいこと◆

☐ 文章・図表には，出典が明記されていますか？
☐ 著作権を侵害していませんか？
☐ 写真・動画の使用許諾は得られていますか？
☐ ご原稿の新規性は問題ございませんか？
☐ ご原稿は最新の情報に則っていますか？

詳しくは詳しくはこちらのウェブページまたは
別紙「ご執筆時のお願い」をご参照ください

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓以下、テンプレートとなります↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

速習 napari ―基礎知識と書き方

黄承宇

Cheng-Yu Hung

ケンブリッジ大学大学院生理学、発生及び神経生物研究科

Department of Physiology, Development and Neuroscience, University of Cambridge

近年、生物画像解析の分野は急速に進歩しており、多次元データに対応した画像可視化ツールの需要が高まっている。Napari は、多次元データに対応し新しく開発された画像可視化ツールであり、画像（Image）、点群（Point）、ラベル（Label）、形（Shape）、トラック（Track）などのデータを表示することができる。また、Napari は Python 上に構築されたオープンソースソフトウェアであり、使いやすくカスタマイズ可能で、多くの科学的ライブラリが利用可能な Python エコシステムの利点を活かしている。このチュートリアルでは、Napari を動かす Python 環境を conda を通じてセットアップする方法から始め、Python コーディングのインターフェースである Jupyter ノートブックを紹介し、データサイエンスでよく使われるパッケージを紹介しながら、Napari の基本知識と使い方を手を動かしながら学んでいく。

＝総説形式＝＝
本文（解説文＋実行コード：25,000 文字）

はじめの一歩

Python 環境のセットアップ

Anaconda/Miniconda のインストール

Python パッケージ管理ツールである conda(https://docs.anaconda.com/)を使用して Python 環境を設定する。conda には Anaconda と Miniconda の二つのバージョンがあり、どちらも本章および後の章で必要な Python 環境を提供する。違いとして、Anaconda にはパッケージ管理のための GUI（グラフィカルユーザーインターフェース）が付属しているが、Miniconda にはない。Miniconda では（Anaconda でも可能だが）コマンドラインでパッケージを管理する。Anaconda にはプログラミング初心者向けの追加機能が多く含まれているため、Anaconda は Miniconda より多くのストレージを必要とする（約 4.4 GB 対約 480 MB）。詳細についてはこちら(https://docs.anaconda.com/distro-or-miniconda/)を参照されたい。本チュートリアルでは、すべてをコマンドラインで行うため、Miniconda で十分である。ダウンロードリンクはこちら(https://docs.anaconda.com/miniconda/#miniconda-latest-installer-links)である。

Miniconda で仮想環境(venv)を作成する

次に、conda を使って仮想環境（virtual environment, venv）を作成する。venv は、異なるプロジェクトごとに必要なパッケージを分けて管理するためのツールである。これにより、各プロジェクトに独立したワークベンチを作成し、プロジェクトごとの依存関係を管理することができる。多くの場合、異なるプロジェクトには異なるパッケージが必要であるため、プロジェクト間のパッケージコンフリクトを防ぎ、パッケージをプロジェクト単位で管理するために venv を使用する。さらに、Python 環境が破損した場合でも、環境を削除して再作成することが可能である。

では、コマンドラインを使用して環境を作成する。Python 環境のインストールに関する詳細なチュートリアルはこちら(https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html)で見つけることができる。まず、エクスプローラーで “anaconda prompt (miniconda3)” を見つけてクリックする。図 1 のような画面が現れる。

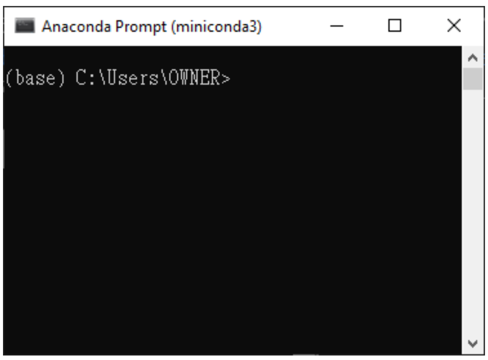


図 1：Anaconda Prompt のスタート画面

“(base)” は、ベース、デフォルトの環境にいることを意味している。次に、napari-env という名前の環境を作成する。環境を作成するには、次のコマンドを実行する：

```
conda create -y -n napari-env -c conda-forge python=3.11
```

ここでは、Python 3.11 の環境を作成する。conda が続行するかどうか尋ねる場合(proceed ([y]/n)? は、`y` を入力する。インストールが完了したら、`done` と表示される（図２A)。次に、環境を `conda activate` コマンドでアクセスする（図２B)。コマンドラインの始まりが`(base)`から `(napari-env)`に変わる（C)。これは、`napari-env`の仮想環境に入ったことを意味する。

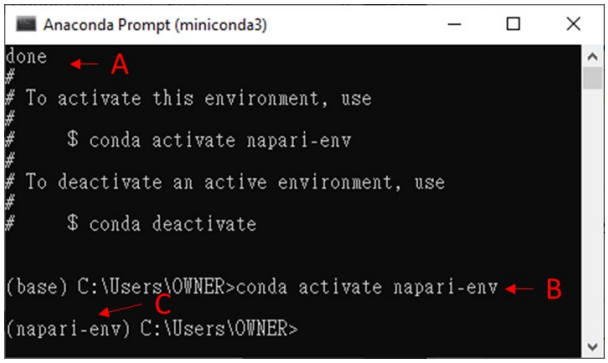


図２：Napari-env 仮想環境に入る

napari と Jupyter ノートブックおよびその他の便利なパッケージのインストール

`napari-env`に入ったら、まず napari をインストールする：

```
python -m pip install "napari[all]"
```

インストールが終われば、図２C の “(napari-env) … >” が再び表示される。

次に、他の画像解析や画像処理に頻繁に使われるパッケージをインストールする。これには、コーディングや napari とのインターフェースに使用する jupyter、画像処理やマトリックス計算を含む数値演算のための numpy、作図のための matplotlib、科学計算のための scipy、画像処理に使われる多くのアルゴリズムを含む scikit-image、及び表形式データの操作のための pandas が含まれている。これらのパッケージをインストールするには、次のコマンドを実行する：

```
pip3 install numpy matplotlib scipy scikit-image jupyter pandas
```

`proceed ([y]/n)?` と表示されたら、`y` を入力する。
以上で環境インストール手順は終了である。

ここで紹介している方法以外にも、yaml ファイル（環境設定ファイル）を使用して環境をインストールすることも可能である。その方法については、後の章で紹介される。

次回 napari-env 環境に戻る方法

コマンドウィンドウを閉じて、再度前回作成した環境に戻りたい場合がある。しかし、コマンドプロンプトを再起動するたびに、常に`(base)`環境に戻ってしまう。`(napari-env)`環境に戻るには、次のコマンドを入力し、エンターキーを押す：

```
conda activate napari-env
```

環境名を異なる名前にした場合や、作成した仮想環境の一覧を確認したい場合は、次のコマンドを入力してエンターキーを押す：

```
conda env list
```

現在の環境には`*`が表示される。

他にも多くの conda コマンドがあり、仮想環境の情報表示や削除などに使用できる。詳細については、こちら(https://docs.conda.io/projects/conda/en/4.6.0/user-guide/tasks/manage-environments.html)のウェブサイトを参照してほしい。

Napari を起動する

このセクションでは、Napari の基本的な使い方を紹介する。

方法１ - コマンドラインから起動する

Napari を起動する方法はいくつかある。一つ目の方法はコマンドラインで`napari`と入力してエンターキーを押すことである。次のコマンドを入力してエンターキーを押す：

```
napari
```

これにより、図３のように Napari ビューアーが開く。画像を読み込むには、トップバーの File メニューをクリックし、Open File(s)を選択する。画像を選択し、Open をクリックすると、画像が表示される。また、別の方法として、画像を直接ドラッグアンドドロップすることも可能である。ビューアーの各部分の機能について次のセクションで説明する。

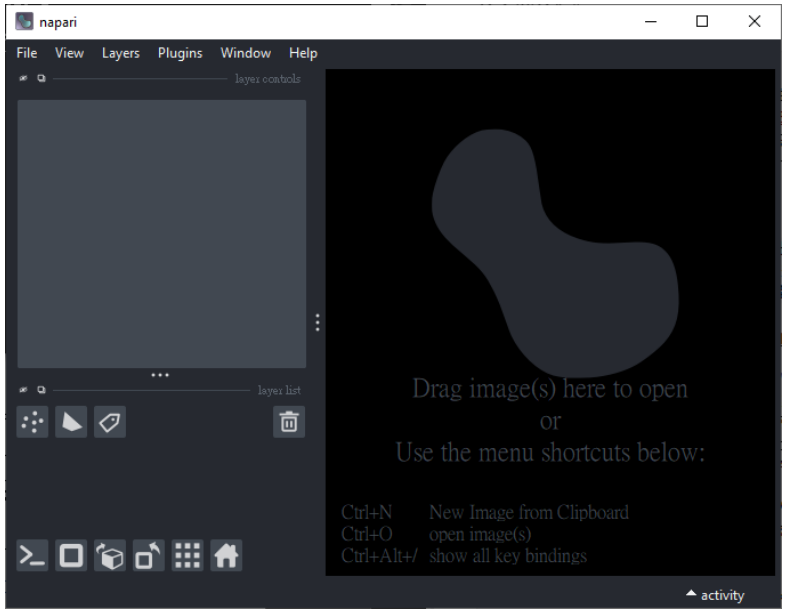


図３:Napari スタート画面

方法２ - Jupyter ノートブックから Python スクリプトで起動する

Napari は Python スクリプトからも起動できる。Python の IDE（統合開発環境）である Jupyter ノートブックを使用して、Napari を起動する方法について説明する。Jupyter では、コードとその結果を同じ場所に表示することができるため、データの可視化に適している。また、コードとドキュメンテーションの両方を含むノートブックを作成することができるため、コードの再利用性が高まり、他の人との共有や科学文献のコードシェアリングにも便利である。

このセクションでは、まず Jupyter ノートブックの基本的な使い方を紹介する。このツールは後の章でも使用するため、役立つであろう。最後に、Napari から画像を呼び出す方法の例を示す。

Jupyter ノートブックを起動する

Jupyter Lab 内で Jupyter Notebook を実行する。Jupyter Lab を起動するには、次のコマンドを実行する：

```
jupyter lab
```

図４のようなページが表示される。

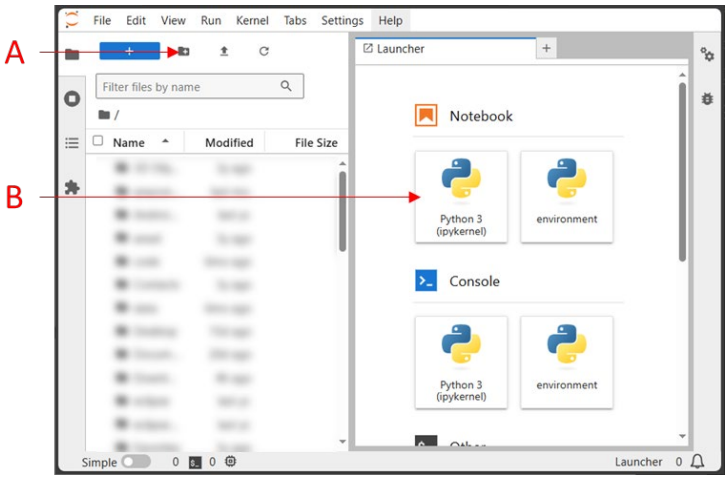


図 4:Jupyter Lab スタートアップページ

次に、この章で作成するスクリプトを保存するフォルダを作成する。まず、図４A をクリックして新しいフォルダを作成し、フォルダ名を napari-tutorial とする。フォルダをクリックして開く。次に、図４B をクリックして新しい Jupyter ノートブックを作成する。作成された新しい .ipynb ファイルを右クリックして、名前を getting_started.ipynb に変更する。これで、Jupyter Lab のウィンドウは図５のようになる。

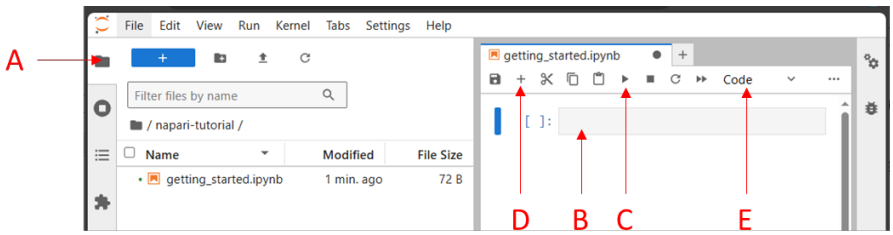


図５: Jupyter エディター

図５A をクリックして左のパネルを閉じ、コーディングを始める。

Jupyter での Hello World

前の章で、いくつかの簡単な Python プログラミングの構文について学んだ。ここでは、それを Jupyter で試してみる。

図５B に次のコードを入力する：

```
print("Hello World!")
```

そして、図５C をクリックするか、ショートカット `Ctrl + Enter` を押すと、コードが実行され、期待通り “Hello World!” が表示される：

Jupyter ノートブックでは、コードはブロックに分割されており、それぞれのブロックを「セル」と呼ぶ。さらにコードの「セル」を追加するには、現在のセルを選択して上部バーの ＋ 記号（図５D）

をクリックするか、ショートカットキー `b` を押す。もし `a` を押すと、新しいセルが現在のセルの上に作成される。

Jupyter でのドキュメンテーション作成

前述のように、Jupyter ノートブックの利点の一つは、ドキュメントとコードを一つのファイルにまとめることができる点である。先ほどは、Python のコードを入力するためのコードセルの作成方法を紹介した。ここでは、Jupyter で「マークダウン（Markdown）」セルを作成する方法を説明する。

Markdown は、テキストを簡単にフォーマット作成できる軽量マークアップ言語である。Markdown を使うことで、見出し、リスト、リンク、画像、表などを簡単な構文で追加できる。ドキュメント、ウェブページ、Jupyter ノートブックなどで広く使われている。

マークダウンセルを作成するには、はじめに一つのセルを選択それか作成し、上部バーのドロップダウンメニュー(図 5 E)をクリックし、「Markdown」を選択するか、ショートカット`m` を押してそのセルをマークダウンに変更する（コードセルに戻すには `y` を押す)。

ノートブックにタイトルをつけてみよう。新しいセルを今のセルの上に作り(ショートカット`a`)、マークダウンセルに変え(ショートカット`m`)、次の内容を入力する：

```
# Jupyter で Napari を起動する
Jupyter ノートブックから*Napari*を起動する方法を説明する。
## Hello World
次のセルでは **「Hello World!」** を表示する。
```

実行ボタンをクリックするか、`Ctrl + Enter` を押す。図 6 のように表示される。

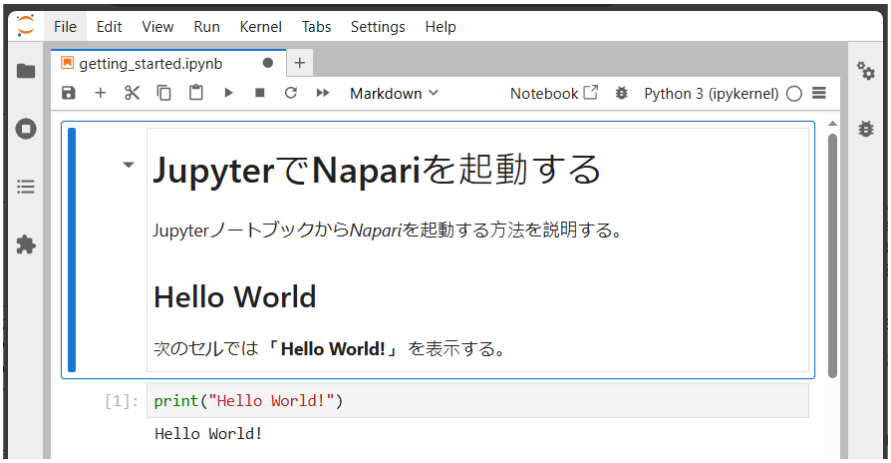


図 6: Jupyter でのマークダウン編集

ご覧の通り、テキストはタイトル（#)、サブタイトル（##）としてフォーマットされている。さらに、* でテキストを囲むことで斜体、** で囲むことで太字にすることができる。これ以外にも、### を使用してサブサブタイトルを作成することができる。また、* を使って箇条書きリストを、1. を使って番号付きリストを、--- を使って水平線を作成することができる。Markdown の構文について詳しくは、このチートシート(https://www.markdownguide.org/cheat-sheet/)を参照してほし

い。このチュートリアルをフォローする共に、各セクションのコードの前にマークダウンで少なくともサブタイトルを入れるようにお勧めする。

ここまで、いくつかの Jupyter ノートブックでのショートカットを学んできた。表 1 によく使う Jupyter ショートカットを整理した。さらにショートカットを知りたい場合は、上部バーの[Help > Show Keyboard Shortcuts]を参照すること。すべてのショートカットを覚える必要はないが、その存在を知っておくと便利である。Jupyter ノートブックを頻繁に使うようになると、自然に身についてくるだろう。

ショートカット	意味
a	セルを上に挿入する
b	セルを下に挿入する
ダブル d	選択したセルを削除する
Shift + Enter	セルを実行して次のセルに移動する
Ctrl + Enter	セルを実行する
y	セルをコードセルに変える(python)
m	セルをマークダウンセルに変える(markdown)

表 1:よく使う Jupyter ショートカット

演習　Hello World のセルの下に新しいマークダウンセルを作成し、セルに「Napari での画像表示」というサブタイトルを書いてみよう。

Jupyter で Napari を起動する

次に、Jupyter ノートブックから Napari を起動する。まず、skimage.data モジュールからサンプル画像を読み込む。新しいセルを作成し、次のコードを入力する:

```
# skimage から 3D 画像を読み込む
from skimage.data import cells3d

# 画像のサイズを取得
shape = cells3d().shape
print(f'画像のサイズ: {shape}')
```

次の出力が表示される：

```
画像のサイズ: (60, 2, 256, 256)
```

この出力から、画像が 4 次元の配列であることがわかる。最初の次元（dim 0）は 60 で、これはスタック内に 60 枚の画像があることを意味する。第 2 の次元は 2 で、各画像に 2 つのチャンネルがあることを示している。第 3 および第 4 の次元は 256 で、各画像のサイズが 256x256 ピクセルであることを示している。

このまま Napari で画像を表示することもできるが、画像を個々のチャンネルに分けて別々に表示する方が便利である。次のセルコードを入力する:

```
# 画像を個々のチャンネルに分離
cell3d_ch0 = cells3d()[:, 0, :, :]
```

```
cell3d_ch1 = cells3d()[:, 1, :, :]

# 画像のサイズを取得
shape = cell3d_ch0.shape
print(f'画像のサイズ: {shape}')
```

次に、Napari を起動し、二つのチャンネルに分けた画像を Napari に追加してみる。次のセルに下のコードを入力してみよう。

```
# napari をインポート
import napari

# napari のビューアを作成
viewer = napari.Viewer()

# 画像を追加
channel_0_img = viewer.add_image(cell3d_ch0, colormap='green',
name='channel 0')
channel_1_img= viewer.add_image(cell3d_ch1, colormap='blue',
name='channel 1')
```

Napari のビューアが表示される。`channel 0` と `channel 1` レイヤーの透明度（Opacity）、コントラスト（Contrast Limit）、およびガンマ（Gamma）を調整してみて、図 7 のような画像が見えるように確認してみよう。また、画像の拡大・縮小にはマウスのホイールを使用してみると良い。

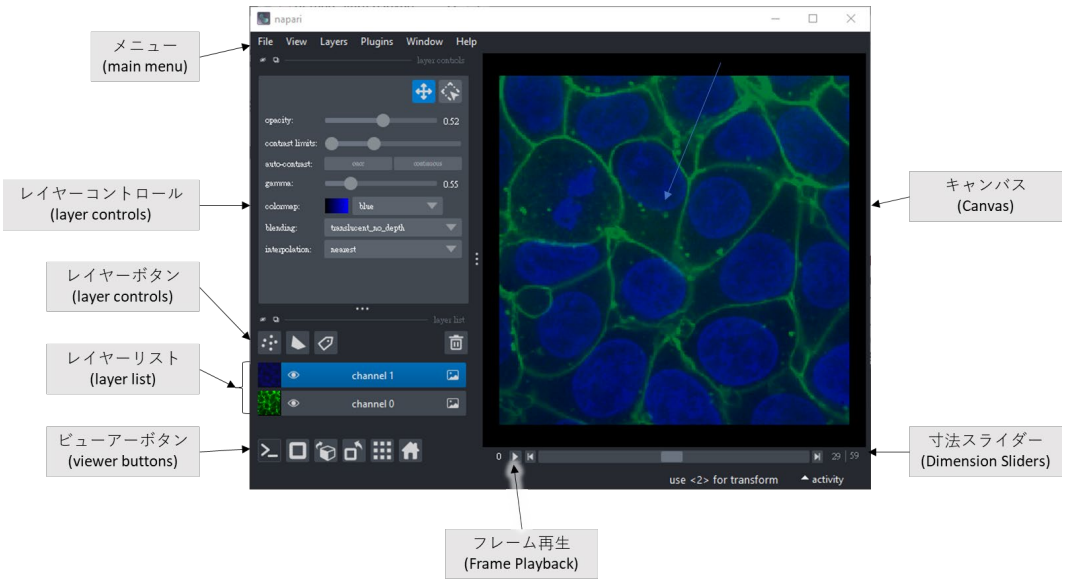


図 7：Napari ビューアー

上の図では、ビューアの各エリアに対応する名前がラベル付けされている。先ほどの調整で、Napari ビューアのキャンバス（Canvas）、レイヤーリスト（Layer List）、レイヤーコントロール（Layer Control）の機能を探索した。ウィンドウの各部分に関する詳細は、こちら(https://napari.org/stable/tutorials/fundamentals/viewer.html#layout-of-the-viewer)で確認でき

る。

コードをもう一度確認してほしい。Napari のビューアがオブジェクト指向プログラミングの概念を利用していることに注意する必要がある（前の章を参照）。そのため、画像や画像関連のデータは**レイヤー**(Layer)として作成されたビューアオブジェクトに追加されている。ここでは、viewer.add_image() メソッドを使って画像レイヤーを追加した。次のセクションでは、他のレイヤーの追加方法についても説明する。

顕微鏡の設計と光学的のリミットにより、3D 画像のほとんどは異方性(anisotropic)であり、x、y、z の各軸のピクセルサイズが異なることが多い。この画像もその一例だ。skimage.data.cell3d のドキュメンテーション(https://scikit-image.org/docs/stable/api/skimage.data.html#skimage.data.cells3d)を確認すると、z、y、x 軸のピクセルサイズ（それぞれ 0.29、0.26、0.26 µm）がわかる。これらのサイズを視覚化するため、z、y、x 軸のスケーリングを下のコードで行う。

```
viewer.layers['channel 0'].scale = [0.29, 0.26, 0.26]
viewer.layers['channel 1'].scale = [0.29, 0.26, 0.26]
```

演習: 図 7 の各部分のスライダーやボタンをそれぞれクリックして、その機能を試してみよう。画像を三次元モードで表示し、下の図のように画像を回転させてみよう。

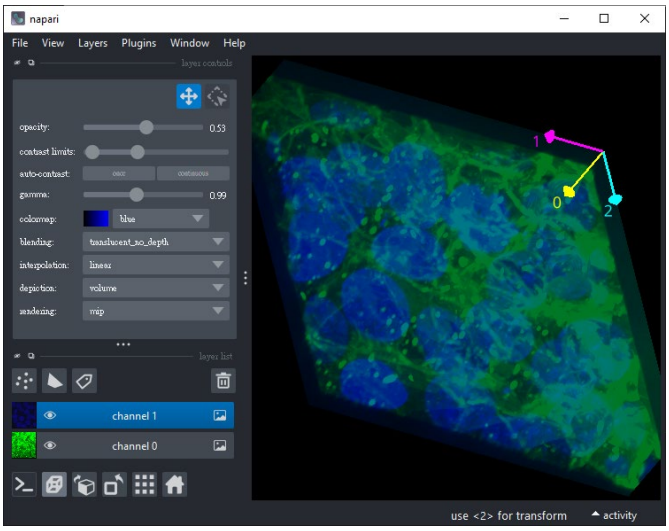


図 8

作業が完了したら、以下のコマンドを実行して Napari ビューアを閉じる。

```
# Napari ビューアを閉じる
viewer.close()
```

ついでに: matplotlib での画像表示

次のセクションに進む前に、matplotlib を使った画像の表示方法をカバーしたい。matplotlib は Python の作図用パッケージで、データの可視化に広く使用されている。プロット、ヒストグラム、パワースペクトル、棒グラフ、エラーチャート、散布図などを作成することができる。Napari が開

発される前は、matplotlib が画像の可視化に最も人気のあるライブラリの一つであり、現在でも他のアプリケーションで広く使用されている。ここでは、matplotlib を使って画像を表示する方法を紹介する。

前のセルの下に新しいセルを作成し、次のコードを入力しよう（何をしているのか説明するために、マークダウンセルを追加してもよいだろう):

```
# skimage から cell 画像を読み込む
from skimage.data import cell

# matplotlib.pyplot を plt としてインポート
from matplotlib import pyplot as plt

# cell 画像を表示
plt.imshow(cell())

# 画像のタイトルを設定
plt.title('cell')
```

下の図のような画像が現れる。

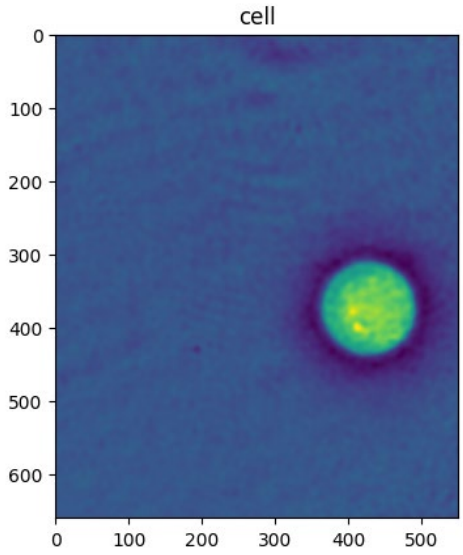


図 9

以前使用した二つのチャンネルの画像を、matplotlib.pyplot.imshow を使って表示することもできる。

```
# 2D に圧縮するためには、最大値投影（max intensity projection）を用いする。これは、3D データの各スライスに対して、
# 指定した軸方向の最大値を取ることで、2D 画像に圧縮する方法。
import numpy as np

cell2d_ch0 = np.max(cell3d_ch0, axis=0)
cell2d_ch1 = np.max(cell3d_ch1, axis=0)
```

```
# 画像の表示
plt.subplot(1, 2, 1)
plt.imshow(cell2d_ch0)
plt.title('Channel 0')
plt.subplot(1, 2, 2)
plt.imshow(cell2d_ch1)
plt.title('Channel 1')
```

図１０のような画像が現れる。

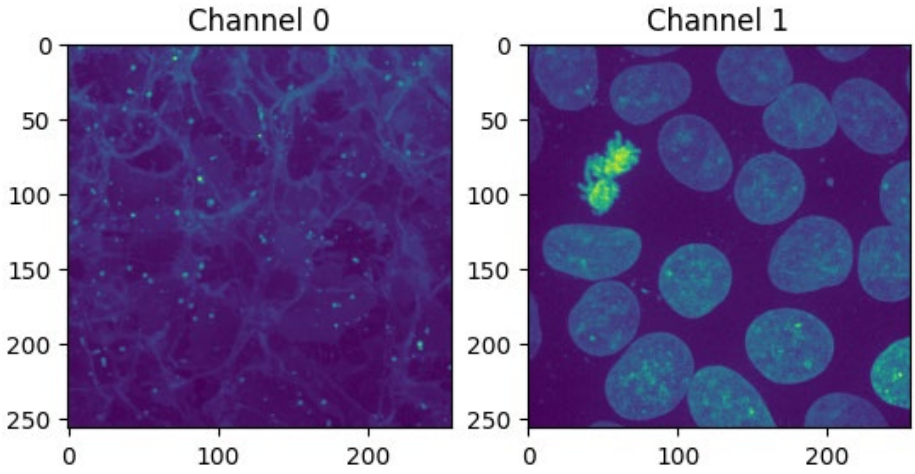


図１０

Napari が登場する前は、Matplotlib が画像可視化の最も人気のあるライブラリであった。しかし、Matplotlib では 3D 可視化が不可能であり、また、複数の画像を重ねて表示することも難しい。この点が、Napari の画像可視化における優位性を際立たせている。だが、2D 画像の可視化や作図などでは Matplotlib は今でも主流ですので言及しておきたい。

次のセクションでは、Napari のもう一つの大きな利点として、Napari ビューアにさまざまなタイプのデータ - レイヤーを追加する方法について学ぶ。

Napari Layer(レイヤー)の紹介

次に、Napari ビューアのレイヤー機能を探っていく。先述の通り、Napari ビューアはオブジェクト指向ビューアであり、データはレイヤーとしてビューアに追加される。前のセクションでは、viewer.add_image()メソッドを使用して画像レイヤーをビューアに追加した。このセクションでは、ラベル、ポイント、トラック、シェイプなど、他のさまざまなタイプのレイヤーをビューアに追加する方法について詳しく見ていく。

Labels Layer(ラベルレイヤー)

Napari に Labels Layer を追加する

ラベルは画像に含まれたピクセルを注記するために使用される。ラベルは整数配列として表現され、各整数が画像の異なる領域を表す。例えば、あるグループのピクセルがみんな同じ細胞核を表してい

たら、そのピクセルたちは同じ数字にラベリングされる。画像をラベルする過程をセグメンテーション(Segmentation)と呼ぶ。Napari では、ラベルは`viewer.add_labels()`メソッドを使ってレイヤーとしてビューアに追加される。次の例では、前のセクションの画像を閾値処理(Thresholding)でセグメンテーションして、得られたラベルをビューアに追加する。ワークフローは図１１のようになる。セグメンテーションワークフローのデザインについては、後の章(実践編 型１)で深く説明される。

では、コーディングをはじめよう。次のセルに、下のコードを入力しよう。

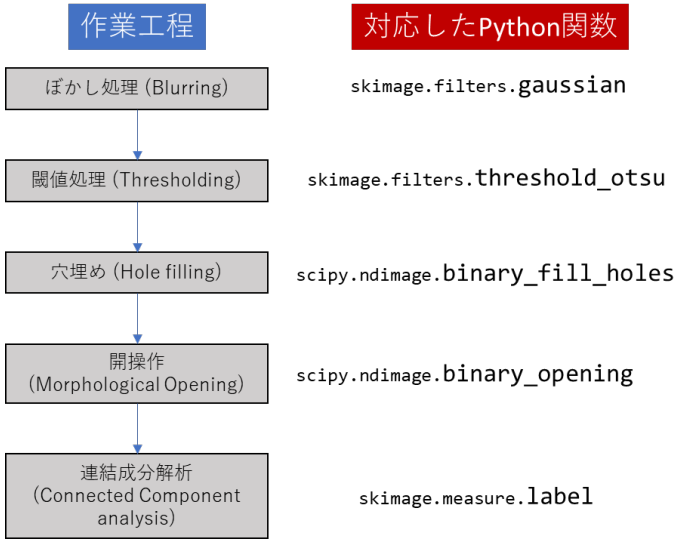


図１１:セグメンテーションワークフロー

```
# napari をインポート
import napari

# napari のビューアを作成
viewer = napari.Viewer()

# 画像を追加 (2D 画像を使用する。)
image_layer = viewer.add_image(cell2d_ch0, colormap='green', name='channel 0')
image_layer = viewer.add_image(cell2d_ch1, colormap='blue', name='channel 1')

# cell2d_ch1 画像に対してガウスぼかしを実行する
from skimage.filters import gaussian

# ガウスフィルターを適用
cell2d_ch1_blurred = gaussian(cell2d_ch1, sigma=1)

# ぼかした画像を表示
image_layer = viewer.add_image(cell2d_ch1_blurred, colormap='blue',
name='channel 1 blurred')
```

```
# cell2d_ch1 画像に対して大津閾値処理を実行する
from skimage.filters import threshold_otsu

# 大津閾値処理を適用
thresh = threshold_otsu(cell2d_ch1_blurred)
cell2d_ch1_thresholded = cell2d_ch1_blurred > thresh

# 閾値処理した画像を表示
image_layer = viewer.add_image(cell2d_ch1_thresholded, colormap='blue',
name='channel 1 thresholded')
```

演習: 図１２のように`cell2d_ch1_thresholded` には、多くの細胞中の穴やデブリがセグメントされている。この画像を使って、次のセルにモルフォロジー操作（ここでは、穴埋めと開操作）を適用して細胞の形を整え、その後、ラベルを作成して `viewer.add_labels()` で Napari ビューアに追加しよう。

ヒント: `skimage.morphology`モジュールの `binary_fill_holes()` と `binary_opening()` メソッドを使用する。ラベリングは、`skimage.measure`モジュールの `label()` メソッドを使用すること。

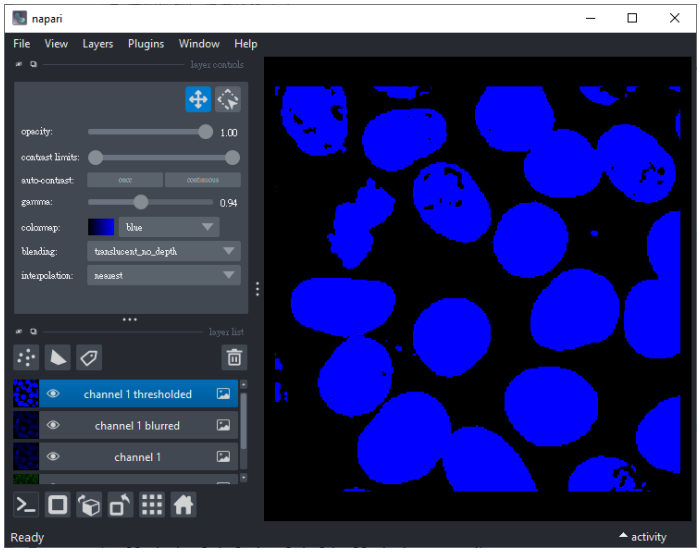


図１２

```
# cell2d_ch1_thresholded 画像に対して穴埋め操作を行う
from scipy.ndimage import binary_fill_holes

# 穴埋め関数を適用する
cell2d_ch1_filled = binary_fill_holes(cell2d_ch1_thresholded)
```

```
# 穴埋め後の画像を表示する
image_layer = viewer.add_image(cell2d_ch1_filled, colormap='blue',
name='channel 1 filled')

# cell2d_ch1_filled 画像に対して開操作を行う
from skimage.morphology import opening, disk

# 開操作を適用する
cell2d_ch1_opened = opening(cell2d_ch1_filled, disk(5))

# 開操作後の画像を表示する
image_layer = viewer.add_image(cell2d_ch1_opened, colormap='blue',
name='channel 1 opened')

# cell2d_ch1_opened 画像に対して連結成分解析を行う
from skimage.measure import label

# 連結成分解析を適用する
cell2d_ch1_labeled = label(cell2d_ch1_opened)

# ラベルを表示する
label_layer = viewer.add_labels(cell2d_ch1_labeled, name='channel 1 labeled')
```

図１３のような画像が表示される。ラベルは異なる色で表示され、異なる領域を区別している。ラベルの透明度とコントラストを調整して、画像を見てみよう。Napari では、レイヤーのプロパティによって、異なるコントロールツールが表示される。ラベルの場合、Eraser、Fill などのツールが表示される。これらのツールを使って、ラベルを編集することができる。

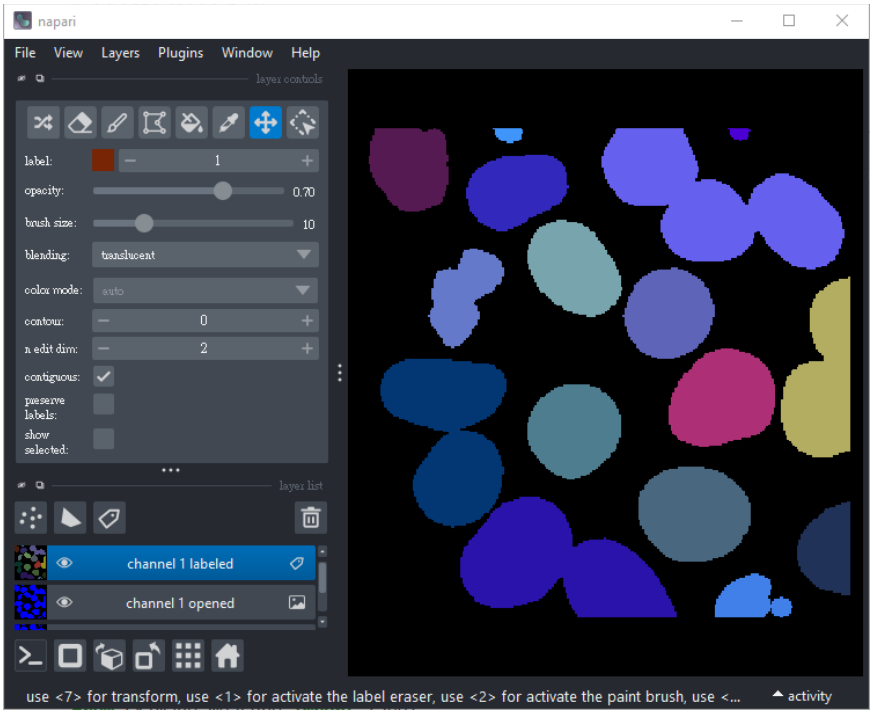


図１３：ラベリングの結果

演習: ラベルを注意深く見ると、一部のラベルが正しくないことに気づくかもしれない。レイヤーコントロールツールを使って、各細胞が異なるラベル（色）を持つようにラベルを編集してみよう(ヒント 1)。また、以下(図１4)に示すようにラベルのボーダーを表示する方法を探してみてほしい(ヒント 2)。最後に、各処理ごとの結果をタイルで示してみよう(ヒント 3)。

*ヒント 1: こちらのサイト（https://napari.org/stable/howtos/layers/labels.html#editing-using-the-tools-in-the-gui）を参照すると良いでしょう。
** ヒント 2: contour の数値を変更してみてください。
*** ヒント３：ビューアボタンのそれぞれを試してみてください。

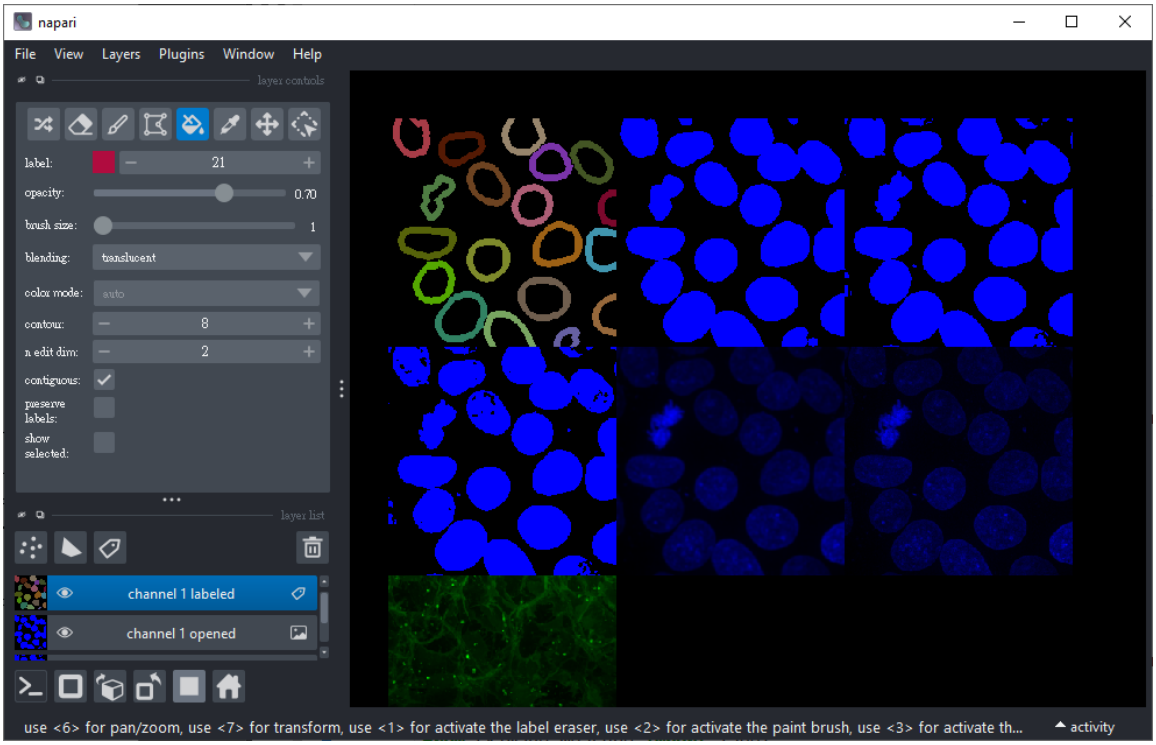


図 1 4

ラベルを保存する

手動でラベルを編集した後、後での分析のためにラベルを保存したい場合、次のコードを実行して、Napari ビューアからノートブックにラベル変数を取り戻すことができる。

```
# ラベルレイヤーからラベルを保存する
labeled_image = label_layer.data

# Napari ビューアを閉じる
viewer.close()

# ユニークなラベルを表示する
unique_labels = np.unique(labeled_image)
print(f'リラベリング前のユニークなラベル: {unique_labels}')

# いくつかのラベルを削除したので、画像を再ラベリングする必要があります
from skimage.measure import label

# 画像を再ラベリングする
cell2d_ch1_relabel = label(labeled_image)
unique_labels = np.unique(cell2d_ch1_relabel)
print(f'リラベリング後のユニークなラベル: {unique_labels}')

# 再ラベリングされた画像を matplotlib で表示する
```

```
plt.imshow(cell2d_ch1_relabel)
plt.colorbar()
plt.title('Channel 1 Relabeled')

# 再ラベリングされた画像を保存する
from skimage.io import imsave

imsave('cell2d_ch1_relabel.tif', cell2d_ch1_relabel)
```

図 15 と “リラベリング前と後のユニークなラベル”がアウトプットとしてでてくるだろう。前のコードセルでは、`label_layer = viewer.add_labels(...` を使ってラベルを表示しましたので、ここでは `label_layer.data` を使ってラベルをビューアオブジェクトから取得した。手動でラベリングを行う過程でラベルを削除したり、新しい ID でラベルを追加したため、「リラベリング前のユニークなラベル」の番号が連続していないことがわかる。理想的には、ラベル番号が 1, 2, 3 ... n で、n がラベルの数(ここでは細胞核の数)となるべきだ。これを実現するために、`skimage.measure` の `label` 関数を使用してラベル番号を再割り当てした。最後に、`plt.imshow` でラベル画像を表示し、`skimage.io.imsave` で画像を保存した。保存時に `UserWarning: cell2d_ch1_relabel.tif is a low contrast image` という警告が表示されることがあるが、これはラベル画像のグレースケールが通常の画像に比べて非常に少ないためである。警告を無視したい場合は、関数に `check_contrast=False` を追加して無効にすることができる。

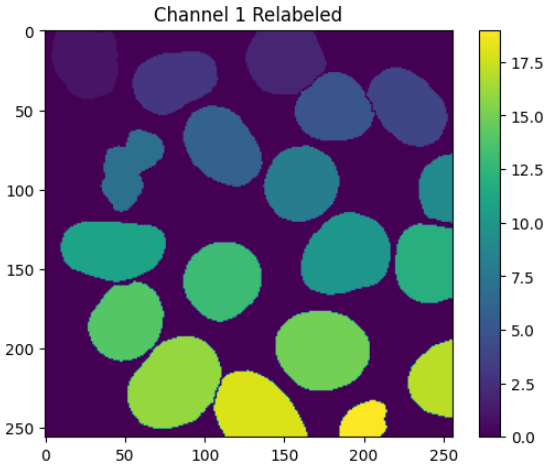


図 15

Regionprops の紹介

次に進む前に、skimage.measure.regionprops_table（または単に regionprops）について触れておきたい。この関数は、ラベルによって画像のプロパティを測定するのに役立つ。以下に画像特徴抽出の例コードを示す。regionprops と regionprops_table の詳細や測定可能な項目については、ドキュメンテーション(https://scikit-image.org/docs/stable/api/skimage.measure.html#skimage.measure.regionprops)を確認してほしい。

regionprops 以外に、ここではもう一つのライブラリである pandas も使用した。Pandas はテーブルデータの処理に広く使用されるライブラリであり、データサイエンティストにとって学ぶべき

最も重要なライブラリの一つである。Pandas は、コードの出力で示されるように、テーブルデータの可視化や、テーブル内のデータを簡単かつ迅速に修正するのにも役立つ。詳細については、Pandas (https://pandas.pydata.org/)のドキュメンテーションやネット上のチュートリアルを確認することをお勧めする。

```
# ラベル付き画像からプロパティを抽出するために regionprops を使用する
from skimage.measure import regionprops_table

# 抽出したいプロパティを定義する
properties = ['label', 'area', 'centroid', 'max_intensity', 'mean_intensity',
'min_intensity']

# プロパティを抽出する
props = regionprops_table(cell2d_ch1_relabel, cell2d_ch1,
properties=properties)

# DataFrame を作成するために pandas をインポートする
import pandas as pd

# props 辞書から DataFrame を作成する
props_df = pd.DataFrame(props)

# DataFrame の最初の数行を表示する
props_df.head()
```

図 16 のように Pandas の df.head()によって測定の結果がテーブルフォーマットで Jupyter ノートブックで示してくれる。

	label	area	centroid-0	centroid-1	max_intensity	mean_intensity	min_intensity
0	1	1544.0	19.823834	25.463083	45903.0	15984.369819	11049.0
1	2	1675.0	18.914627	150.336119	29258.0	16992.082388	12140.0
2	3	1597.0	31.976205	80.629931	34617.0	16980.803381	11855.0
3	4	1740.0	48.414943	227.582759	27219.0	16238.800000	10954.0
4	5	1611.0	49.557418	181.222843	36134.0	19216.277467	10290.0

Regionprops の測定結果で、図 17 のような簡単なグラフを作れる:

```
# 面積のヒストグラムを作成する
plt.hist(props_df['area'])
plt.xlabel('Area [pixels]')
plt.ylabel('Cell count')
plt.title('Cell Area Distribution')
```

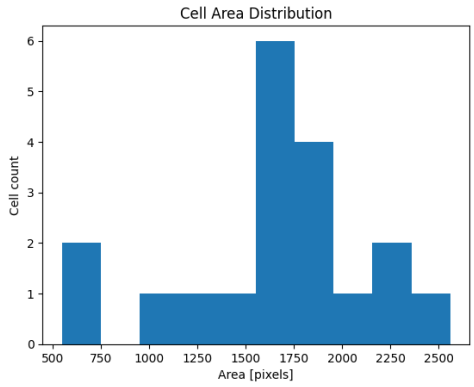


図１７:簡単な面積分布測定

Points Layer (点群レイヤー)

このセクションでは、粒子トラッキングの例を通じて、Napari のポイントレイヤーについて学ぶ。粒子の重心を検出し、これらの点を Napari で画像上に表示し、点を編集する方法を学ぶ。

このセクションおよび次のセクションで使用する画像は、後の章でトラッキング学習に使用するものである。ここでは、オブジェクトを手動でトラッキングする。その後、実践編・型２でトラッキングの原理と自動トラッキングソフトウェアの使用方法について詳しく学ぶ。

まず、このリンク(https://github.com/miura/jikken_igaku2023/tree/main/TrackMate)からデータをダウンロードし、Jupyter ノートブックと同じフォルダに配置する。直接 Jupyter の左半分(図５)のファイルエクスプローラーにドラッグアンドドロップできる。

次に、以下のコードを実行して、Napari で画像を開く。

```
# croped_sample.tif 画像を読み込む
from skimage.io import imread
import napari

croped_sample = imread('croped_sample.tif')

# croped_sample 画像の形状を表示する
print(f'The shape of the croped_sample image is: {croped_sample.shape}')

# croped_sample 画像を napari で表示する
viewer = napari.Viewer()
sample_image = viewer.add_image(croped_sample, name='sample_image')

# タイムラプスを視覚化するために時間の次元を誇張する
viewer.layers['sample_image'].scale = [15, 1, 1]
```

点の重心を探す

次に、粒子の重心を検出する。

演習：粒子の重心を検出するプログラムを書いてみよう。図 18 の例のワークフローを参考にしてほしい。

ヒント：このタスクには、前のセクションで示した大津の閾値処理方法と regionprops_table を使用できる。

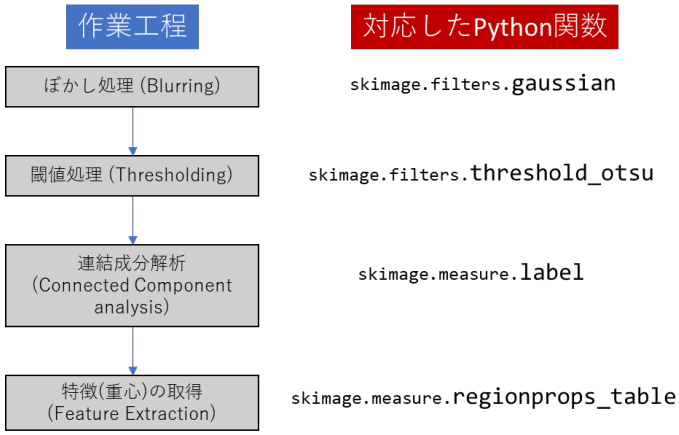


図 18：粒子重心を検出するワークフロー

```
# 各時間ポイントに対して croped_sample 画像に大津の閾値処理を適用する
from skimage.filters import threshold_otsu
import numpy as np
from skimage.filters import gaussian
from skimage.measure import label

# 閾値処理された画像を格納する空のリストを作成する
labeled_images = []

max_label_last_time_point = 0

# 各時間ポイントをループ処理する
for time_point in range(croped_sample.shape[0]):
    # 画像をスムージングするためにガウスフィルターを適用する
    smoothed_image = gaussian(croped_sample[time_point, :, :], sigma=1)
    # 大津の閾値処理を適用する
    thresh = threshold_otsu(smoothed_image)
    thresholded_image = smoothed_image > thresh
    # 連結成分解析（ラベリング）を適用する
    labeled_image = label(thresholded_image)
    # ラベルが時間ポイント間で一意になるように再ラベリングする
    labeled_image_unique = labeled_image + max_label_last_time_point
    labeled_image_unique[labeled_image == 0] = 0
```

```
    labeled_images.append(labeled_image_unique)
    # 最大ラベルを更新する
    max_label_last_time_point = np.max(labeled_image_unique)

labeled_images = np.array(labeled_images)

# napari でラベル付けされた画像を表示する
labeled_image = viewer.add_labels(labeled_images, name='labeled_images')
viewer.layers['labeled_images'].scale = [15, 1, 1]

# ラベル付けされた領域の重心を取得する
from skimage.measure import regionprops_table
import pandas as pd

# 抽出したいプロパティを定義する
properties = ['label', 'centroid']

props = regionprops_table(labeled_images, properties=properties)
df = pd.DataFrame(props)
# データフレームの最初の数行を表示する
df.head()
```

図 19 のような pandas テーブルが表示される。

	label	centroid-0	centroid-1	centroid-2
0	1	0.0	346.431776	234.282243
1	2	1.0	264.918466	199.986338
2	3	2.0	247.146011	168.451975
3	4	3.0	237.254724	153.732558
4	5	4.0	254.974448	118.499655

図 19

Napari に Points Layer を追加する

すべての ROI の重心が取得できたので、次のステップでは、Napari が受け入れる形式に変換します。

Napari が受け入れる形式は次の通りです

(https://napari.org/stable/howtos/layers/points.html を参考)：

```
points = np.array([[100, 100], [200, 200], [300, 100]])
```

見た通り、points 変数のサイズ(dimension)は NxD, N はポイントの数、D は次元数となる。

それでは、先ほど取得した Dataframe をこの形式に変換します

```
# データフレームを次の形式に再整形します：
# points = np.array([[0, 100, 100], [1, 200, 200], [2, 300, 100]])
centroid_0 = np.array(df['centroid-0'].to_list())
```

```
centroid_1 = np.array(df['centroid-1'].to_list())
centroid_2 = np.array(df['centroid-2'].to_list())

# セントロイド座標を結合
points = np.column_stack((centroid_0, centroid_1, centroid_2))
points
```

これで、Napari に Point Layer としてインポートする準備が整いた:

```
# ポイントを Napari ビューアにポイントレイヤーとして追加する
points_layer = viewer.add_points(points, size=10, name='centroids')

# ポイントレイヤーのスケールをイメージレイヤーと同じように調整する
viewer.layers['centroids'].scale = [15, 1, 1]
```

図 20 のような表示が確認できるはずです。ポイントがラベルに隠され見えるかもしれない。軸を表示するには、[View > Axis > Axis Visible] を選択する。

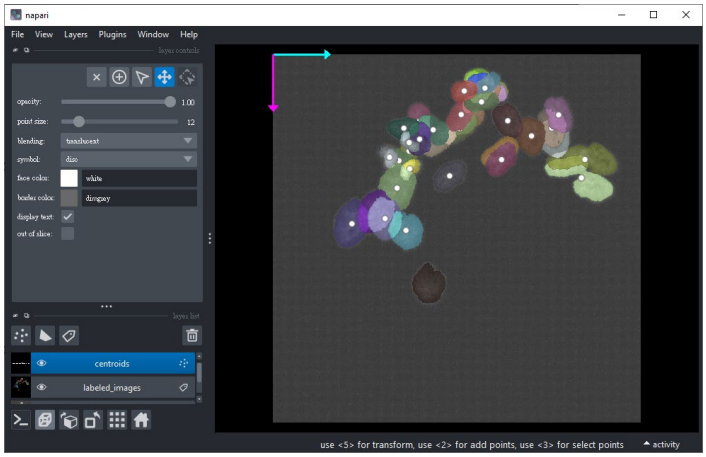


図 20:ポイントレイヤーの表示

演習: よく見れば、ROI が二つ重なってしまった場合がある。各粒子に対応するようにポイントを修正せよ。修正後、ポイントレイヤーのコントロールパネルで、ポイントの色を個々の粒子に対応させる（フレーム 6 以降に分裂したもの）。最終的には、図 21 のようになる。

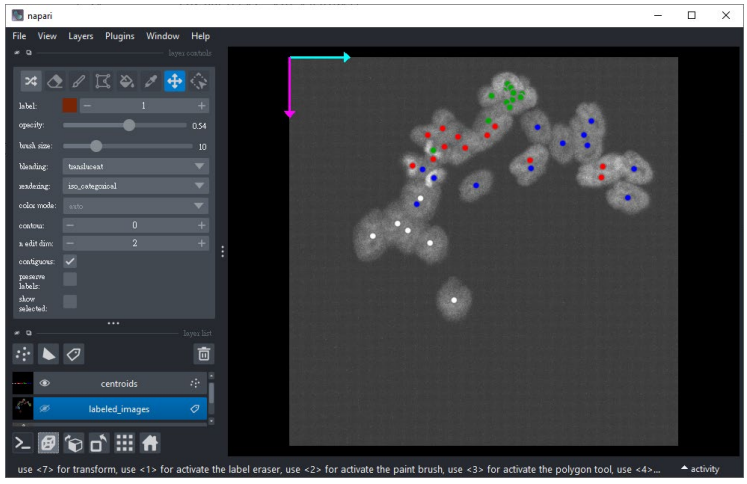


図２１：ポイントレイヤー修正後の結果

以下のスクリプトは、ポイントの色を抽出し、それを ID に変換する。

```
# ポイントとフェイスカラーを取得する
new_points = points_layer.data
points_face_color = points_layer.face_color

# ポイントフェイスカラー配列内のユニークな行を識別する
unique_colors = np.unique(points_face_color, axis=0)

# ポイントフェイスカラーを color_id に変更する
color_id = np.zeros(len(points_face_color), dtype=int)
for i, color in enumerate(unique_colors):
    color_id[(points_face_color == color).all(axis=1)] = i

# DataFrame に color_id を追加する
new_points_df = pd.DataFrame(new_points)
new_points_df['color_id'] = color_id

# Dim-0, 1, 2 を t,y,x に名前を変える
new_points_df.columns = ['t', 'y', 'x', 'color_id']

# データフレームの最初の数行を表示する
new_points_df.head()
```

図２２のような Pandas table が出てくる。選んだカラーによって、color_id が違うオーダーになっているかもしれませんが。

	t	y	x	color_id
0	0.0	346.431776	234.282243	3
1	1.0	264.918466	199.986338	3
2	2.0	247.146011	168.451975	3
3	3.0	237.254724	153.732558	3
4	4.0	254.974448	118.499655	3

図２２

Napari の Points Layer の解説はこれで終了である。Points Layer では、ポイントに「特徴」を追加するなど、他にもいくつかの操作が可能である（色を追加する方法に非常に似ている）。詳細については、Napari のドキュメントの点群レイヤーに関する部分を確認することを推奨する。

Napari のビューアはまだ閉じないでほしい。次のセクションでは、点群レイヤーを接続し、「トラッ

ク」レイヤーを導入する。

Tracks Layer (トラックレイヤー)

Tracks 変数の作成

以下では、先ほど作成した点を接続し、それをトラック(Tracks)に変換する。Napari では、Tracks Layer への入力データは、トラック ID と N 個の点を D 次元座標で含む NxD+1 の NumPy 配列またはリストでなければならない。Tracks データ管理の詳細は後の章で説明するが、現時点では、2D + 時間のトラックの場合、データは次のように配置する必要があることを覚えておいてほしい。

```
   track_id    t    y    x
0         1  ...  ...  ...
1         1  ...  ...  ...
2         2  ...  ...  ...
```

3D + 時間のトラックの場合、データは次のように配置する。

```
   track_id    t    y    x    z
0         1  ...  ...  ...  ...
1         1  ...  ...  ...  ...
2         2  ...  ...  ...  ...
```

それでは、データを再編成してトラック形式に変換する。

```
# Track Dataframe を編成: new_points_df の color_id 列を最初の列に移動し、列名を
'track_id'に変更
tracks_df = new_points_df[['color_id', 't', 'y', 'x']]
tracks_df.columns = ['track_id', 't', 'y', 'x']
tracks_df.head()
```

	track_id	x	y	z
0	3	0.0	346.431776	234.282243
1	3	1.0	264.918466	199.986338
2	3	2.0	247.146011	168.451975
3	3	3.0	237.254724	153.732558
4	3	4.0	254.974448	118.499655

図２３

Napari に Tracks Layer を追加する

Track 変数が準備できたら、`viewer.add_tracks`で Napari Viewer に入れる

```
# Track  Layer を追加
```

```
tracks = viewer.add_tracks(tracks_df, name='tracks')
# Track Layer のスケールを Image Layer と同じように調整する
viewer.layers['tracks'].scale = [15, 1, 1]
```

図２４のように、tracks が Napari Viewer に現れる。Tracks Layer コントロールパネルの各スライダーとボタン、ドロップダウンリストの機能を試してみよう。フレーム再生ボタン(図７を参考)も試してみるがよい。

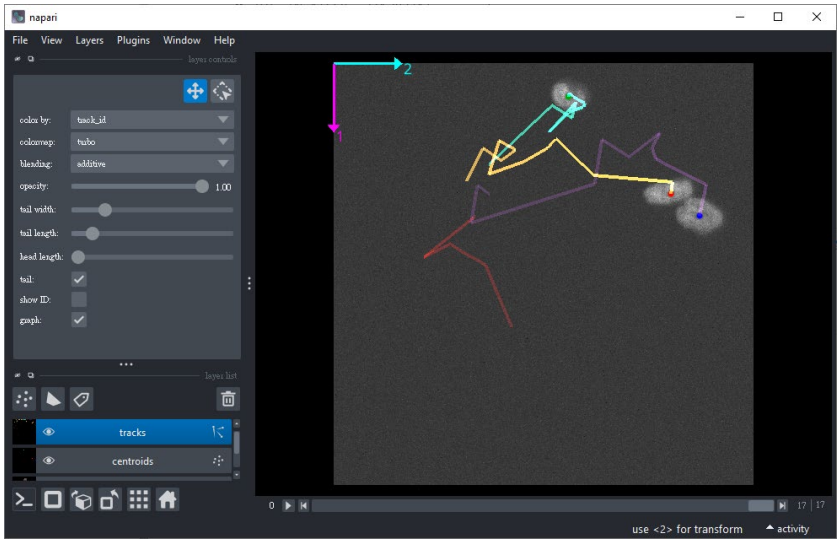


図２4:Tracks Layer の表示

Tracks の合流、分岐を管理する

Tracks の分岐点(フレーム６)が接続されていないことに気づくだろう。これは、Track データの分岐前後でそれぞれを別々のトラックとして扱っているためである。

Tracks の graph 引数を使用して、tracks 間の関係（例えば、合流や分岐）を定義することができる。graph は Python Dictionary として定義され、キーが track_id、値がその track の"親"の track_id となる。例えば、我々のケースでは、track3 が track0、1、2 に分岐する（track3 が track0、1、2 の”親”である)。詳細は track 章で学ぶが、ここでは次のように graph を定義する：

```
graph = {
    0: [3],
    1: [3],
    2: [3],
}
```

Graph を`viewer.add_tracks`のインプットとして扱う。

```
# グラフを Napari  add_tracks に追加
connected_tracks = viewer.add_tracks(tracks_df, graph=graph,
name='connected_tracks')
viewer.layers['connected_tracks'].scale = [15, 1, 1]
```

これでトラックが接続される。トラックを時間軸を 3 次元目の空間として可視化することもできる

(図２5)。

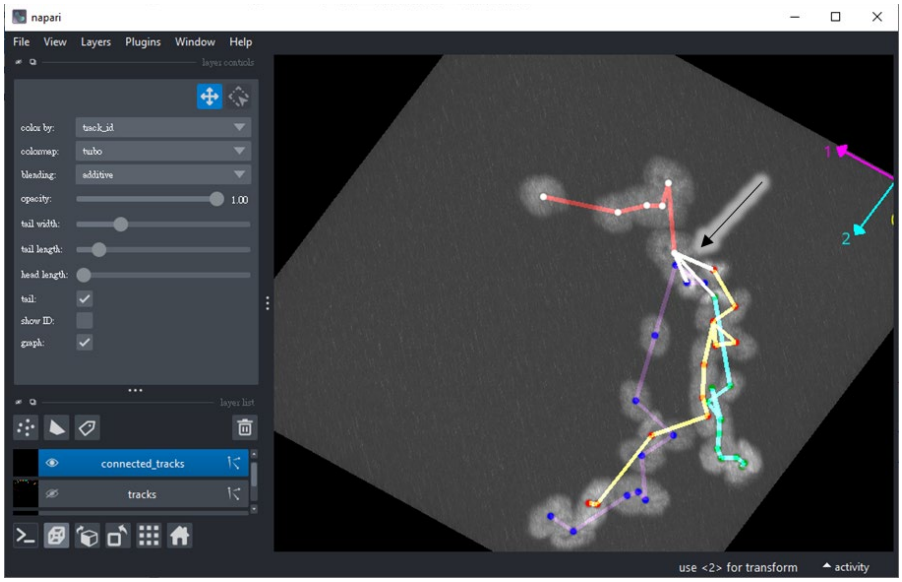


図２5:Tracks を 3D での表示。分岐している所が連結されていることをハイライトしている。

```
# Napari ビューアを閉じる
viewer.close()

# トラックデータフレームを保存する
tracks_df.to_csv('tracks.csv', index=False)
```

これで、Napari の Tracks Layer のウォークスルーは終了である。このセクションと前のセクションで、粒子追跡の例を通じて Napari の Points Layer と Tracks Layer について学んだ。ご覧のとおり、手動でのトラッキングは非常に面倒である。2024 年 8 月時点で、Napari コミュニティでは、手動でのトラックの注釈(Manual Track Annotation)を現状よりも容易にする方法を模索し始めており、今後の開発のアップデートに期待が持てる。また、多くの自動トラッキング方法が最近開発されており、後の章で紹介される。このセクションを通じて、その章を楽しみにしてほしい。次に、もう一つよく使われる Napari レイヤーである Shape Layer を紹介する。

Shape Layer (形レイヤー)

このチュートリアルで最後に紹介するレイヤーとして、Shapes Layer（形レイヤー）について説明する。その名の通り、このレイヤータイプを使って形状を作成することができる。以下のコードを使用して形状を作成できる(図２6)。

```
import napari
import numpy as np

# 三角形と長方形の頂点座標を定義
triangle = np.array([[10, 200], [50, 50], [200, 80]])
rectangle = np.array([[40, 40], [40, 80], [80, 80], [80, 40]])
```

```
# 三角形と長方形を napari ビューアに Shapes Layer として追加

viewer = napari.Viewer()

# 三角形を追加
triangle_layer = viewer.add_shapes([triangle], shape_type='polygon',
edge_color='red', face_color='blue', name='triangle')
# 長方形を追加
rectangle_layer = viewer.add_shapes([rectangle], shape_type='polygon',
edge_color='green', face_color='yellow', name='rectangle')
```

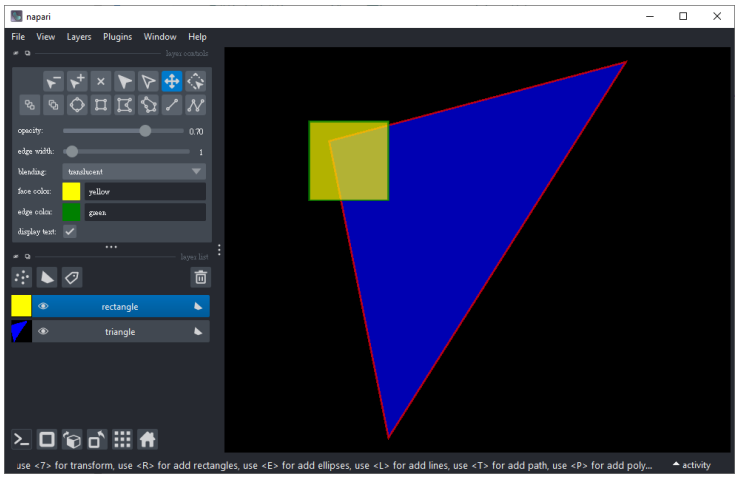


図 2 6: Napari Viewer の Shapes Layer

演習：Napari Viewer 内で形を作成できるか確認してください。これを行う方法については、Napari のドキュメンテーションを参照してください。

Napari の Shapes Layer では、形を作成するだけでなく、パス (Path) オブジェクトも作成できる。これにより、手動でデータをトレーシング(Tracing)したり、トレース(Trace)データを表示することが可能になる。例えば、神経の形態分析(例えば、*.swc ファイルの可視化)や、後の章、実践編・型 3 で説明される血管のトレース可視化にも利用できる。ここでは、実践編・型 3 で使用する血管画像を手動でトレーシングしてみよう。

https://bit.ly/jikken-igaku-kata から、画像をダウンロードして、前セクションと同じようにノートブックと同じフォルダーに画像を移動しよう。

以下のコードを使用してその画像を読み込もう:

```
# BloodVessels_small.tif を読み込む
from skimage.io import imread
import napari

blood_vessels = imread('BloodVessels_small.tif')
```

```
# 画像を表示する
viewer = napari.Viewer()
blood_vessels_image = viewer.add_image(blood_vessels, name='blood_vessels')
```

演習: Napari Viewer で Shapes Layer を作り Path 機能で血管をなぞってみよう。

以下のような結果が得られます:

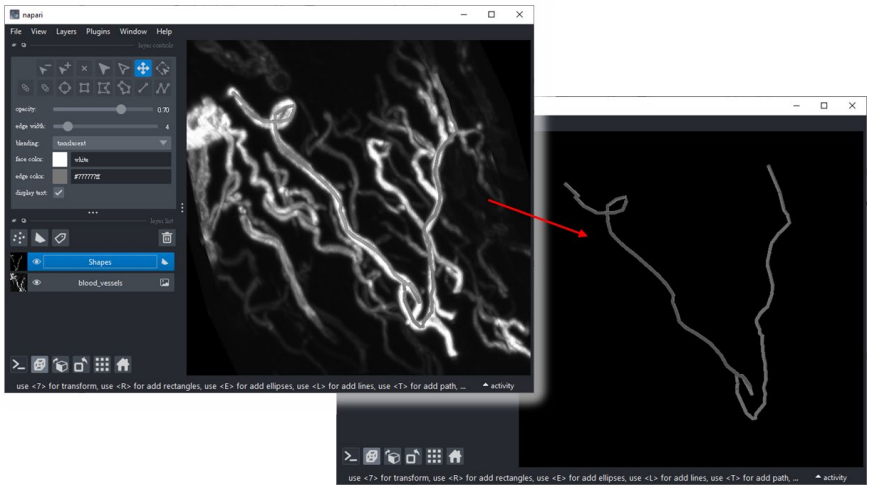


図２７:血管のトレーシングを Napari Shapes Layer で表示

Shape の名前を'blood_vessel_trace'に変えよう。これは、レイヤーリスト(図７を参考)からレイヤーをダブルクリックして簡単に変えられる。結果を Jupyter 環境に取り戻すには、次のコマンドを使用します:

```
# viewer.layers['blood_vessel_trace']からトレースデータを取得
trace_data = viewer.layers['blood_vessel_trace'].data
trace_data_np_array = np.array(trace_data)

# trace_data_np_array の形状を確認
print(f'trace_data_np_array の形状は: {trace_data_np_array.shape}')

# 次元のサイズが 1 のため、配列を「squeeze」します
trace_data_np_array_squeezed = np.squeeze(trace_data_np_array)

# trace_data_np_array_squeezed の形状を確認
print(f'trace_data_np_array_squeezed の形状は:
{trace_data_np_array_squeezed.shape}')

import pandas as pd

# trace_data_np_array_squeezed から DataFrame を作成
```

```
trace_data_df = pd.DataFrame(trace_data_np_array_squeezed, columns=['z', 'y',
'x'])
trace_data_df.head()
```

```
# DataFrame を csv ファイルとして保存
trace_data_df.to_csv('trace_data.csv', index=False)
```

試した通り、手動トレースは大変手間のかかる作業である。さらに、同じ人が同じ画像でトレースを行っても、毎回異なる結果になる可能性があり、再現性の問題がある。実践編・型 3 では、正確かつ自動的にトレースを行う方法について探る。

これまでに、Napari の Image Layer、Labels Layer、Points Layer、Track Layer、Shapes Layer について紹介した。これらは私の意見では、Napari で最もよく使用されるレイヤーである。さらに、Vectors Layer と Surfaces Layer の 2 種類もあり、データの視覚化や操作に利用できる。興味がある方は、Napari の公式ウェブサイトで詳細を確認してほしい。

Napari の Plugins(プラグイン)

Napari も Fiji-ImageJ のようにプラグインシステムを持っている。このセクションでは、Napari でプラグインをインストールする方法を紹介する。Napari にはプラグインをインストールする方法が二つある。一つは、Napari ビューアの GUI にある Napari プラグインインストーラーを使う方法、もう一つはコマンドラインを使う方法である。ここでは、両方の方法を使って、プレゼンテーションに使う動画を作るのに便利な napari-animation と、手動 3D トレーシングで便利な napari-filament-annotator をインストールする例を紹介する。

Napari プラグインマネージャーを使ったプラグインのインストール

Napari Viewer を開き、[Plugins > Install/Uninstall Plugins]に移動する。インストール可能なプラグインのリストが表示されるので、nnapari-animation プラグインを検索し、インストールボタンをクリックする。

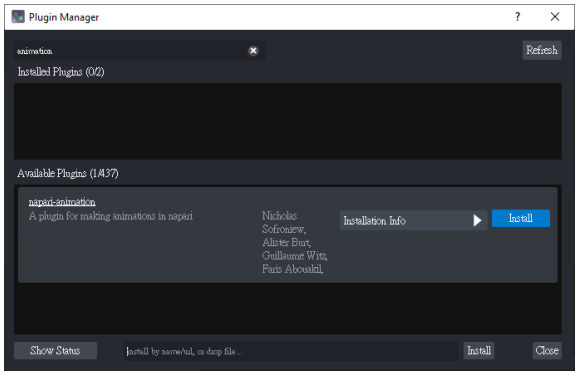
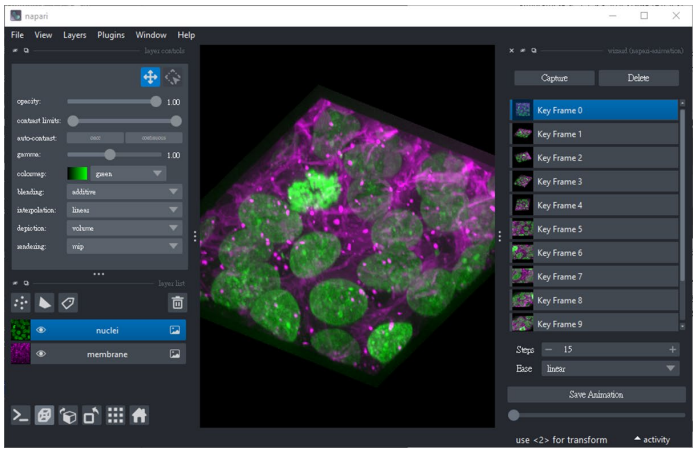


図 2 8:Napari の Plugin Installer

インストールが完了したら、Napari ビューアを再起動する。Napari のトップバーの[Plugin]から

napari-animation を探そう。最初のセクションで使用した画像(cells3d)を読み込み、ドキュメンテーションをチェックしながら動画を作ってみよう。Cells3d は Napari のサンプル画像であり、[File > Open Sample > Napari builtins > cells (3D +2Ch)] で開ける。

サンプルで作られた動画は、以下の QR コードで観れる。



サンプル動画

図 2 9:Napari-animation plugin のインターフェースとサンプル動画(QR コード)

コマンドラインを使ったプラグインのインストール

コマンドラインを使用してプラグインをインストールするには、プラグインの GitHub リポジトリまたは Napari Hub（https://www.napari-hub.org/）を確認する。Napari Hub では、Napari プラグインを共有・インストールできる。Napari Hub のウェブサイトに移動し、napari-plot-profile を検索する。

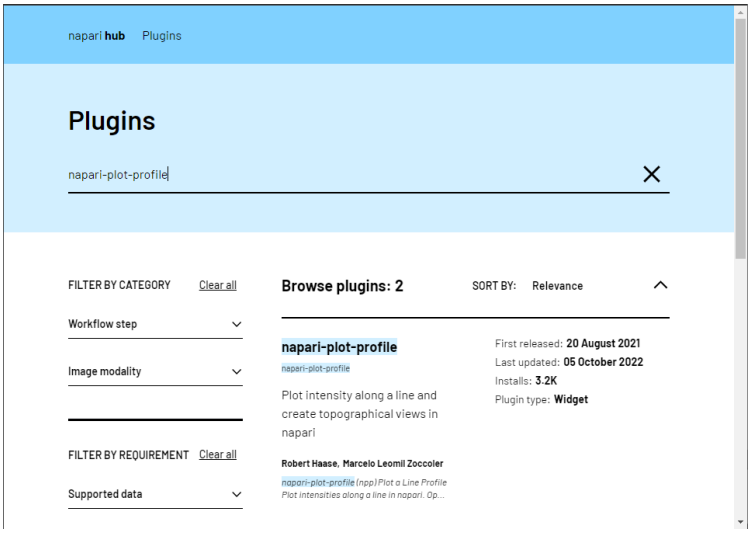


図 30:Napari-hub で napari-plot-profile を捜索

Napari-plot-profile のページ(https://www.napari-hub.org/plugins/napari-plot-profile)に入り、一番下に Installation Instruction がある。Pip でのインストールをお勧めしている:

```
pip install napari-plot-profile
```

このラインをコピーし、anaconda prompt を開け napari-env に入り(*Python 環境のセットアップ*セクションを参考)このコマンドを入力する。インストール後問題なければ Napari のトップバーの[Plugin]から napari-plot-profile が探せるはずだ。もう一回前のサンプル画像を呼び出し、Napari-hub 上のドキュメンテーションを見ながら簡単に画像の輝度の測定を行おう。

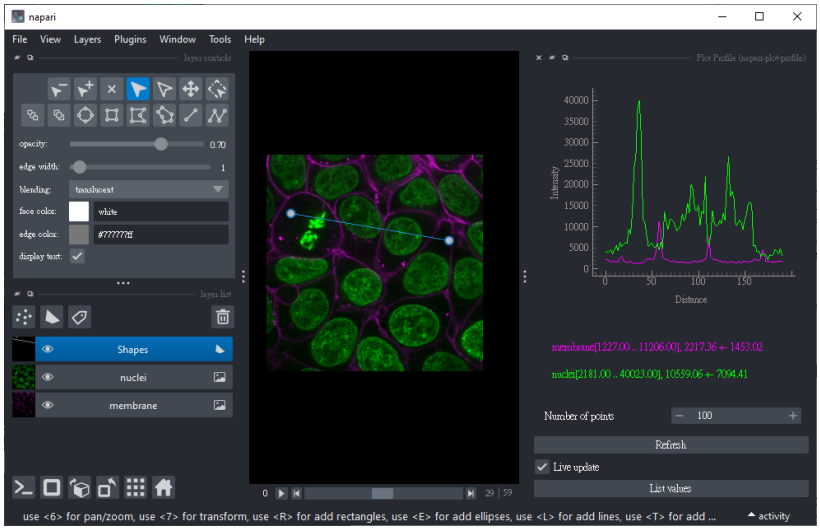


図 31: napari-plot-profile での輝度測定

その他の便利な Napari Plugins

この他にお勧めしたい Napari Plugin がいくつある。一般的な画像解析の使用に際しては、napari-assistant（https://www.napari-hub.org/plugins/napari-assistant）をお勧めする。これは、クリック＆トライ方式で画像解析ワークフローを構築するのに役立つ。Fiji-ImageJ の方法で画像処理に慣れている場合、非常に便利である。また、`napari-assistant`と併せて`napari-script-editor`（https://www.napari-hub.org/plugins/napari-script-editor）をインストールすると、ImageJ スクリプトレコーダーに相当する機能を実現できる。クリックした操作に対応する Python コードを生成してくれる。

napari-filament-annotator（https://www.napari-hub.org/plugins/napari-filament-annotator）は、フィラメントデータのラベリングやトレーシングに特化した便利なツールである。また、ディープラーニングベースのセグメンテーションやノイズ除去ツールも複数の Napari プラグインとして利用可能である。例えば、`cellpose-napari`（https://www.napari-hub.org/plugins/cellpose-napari）や`stardist-napari`(https://www.napari-hub.org/plugins/stardist-napari)、`napari-n2v`（https://www.napari-hub.org/plugins/napari-n2v）などがある。

これらのプラグインをぜひ自らで試してみてほしい。

もちろん、自分自身のプラグインを開発することも可能であり、そのためのガイドラインが Napari のウェブサイトに掲載されている（https://napari.org/stable/plugins/index.html）。ただし、これはより上級者向けの内容となるため、このチュートリアルではカバーしない。

エラーが発生した場合

Napari はまだ発展途上であり、バグや問題に遭遇することがあるかもしれない。もし問題が発生した場合は、Napari のドキュメントや GitHub リポジトリを確認してほしい。また、GitHub リポジトリや Image.sc で質問することもできる。Napari のコミュニティは非常に活発で助け合いの精神が強いので、必要な場合は遠慮せずに助けを求めてほしい。

おわりに

この章では、Napari の基本の使い方について説明した。Napari のインストール方法、コマンドラインや Jupyter ノートブックからの起動方法、Napari Viewer に異なる種類のデータ・レイヤーを追加する方法についても取り上げてきた。また、Napari にプラグインをインストールする方法についても説明した。

Napari は多次元データの視覚化に強力なツールであり、画像データの探索と解析に使いやすいインターフェースを提供してる。この章ではデータを手動でラベリング/トレーシングする方法に焦点を当てたが、後の章では、自動のワークフローを設計し、解析を行う方法について説明していく。この章が Napari の良い入門となり、これから Napari を使って自身のデータを探索し多次元ビューアで違ったデーターのアスペクトを探ったり、さらには Napari コミュニティへの貢献を始めるきっかけとなることを願っている。

↑↑↑↑本文ここまで（文字数のカウントはここまでです）

図表（10–15 点：目安です）

・誌面掲載時の図のサイズは 1 点あたり 150 mm× 80 mm を目安としています
・図の点数が超過する場合は、文字数（上記サイズで 600 文字）を適宜ご調整ください
・図は本 Word ファイルには埋め込まずに別データでご用意ください

図 1　タイトル

図 1　説明（200 字以内）○○．○○．○○○．○○○．（文献 1 より引用）

・誌面掲載時の図のサイズは 1 点あたり 150 mm × 80 mm を目安としています
・図の点数が超過する場合は、文字数（上記サイズで 600 文字）を適宜ご調整ください
・図は本 Word ファイルには埋め込まずに別データでご用意ください

図２　タイトル

図 2　説明（200 字以内）○○○．○○．○○．○○○．（文献 1 より引用）

・誌面掲載時の図のサイズは 1 点あたり 150 mm × 80 mm を目安としています
・図の点数が超過する場合は、文字数（上記サイズで 600 文字）を適宜ご調整ください
・図は本 Word ファイルには埋め込まずに別データでご用意ください

図３　タイトル

図３　説明（200 字以内）○○○．○○．○○．○○○．（文献 1 より引用）

・誌面掲載時の図のサイズは 1 点あたり 150 mm × 80 mm を目安としています
・図の点数が超過する場合は、文字数（上記サイズで 600 文字）を適宜ご調整ください
・図は本 Word ファイルには埋め込まずに別データでご用意ください

図４　タイトル

図４　説明（200 字以内）○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○．○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○．○○
○○○○○○○○．○○○
○○○○○○○．（文献 1 より引用）

・誌面掲載時の図のサイズは 1 点あたり 150 mm × 80 mm を目安としています
・図の点数が超過する場合は、文字数（上記サイズで 600 文字）を適宜ご調整ください
・図は本 Word ファイルには埋め込まずに別データでご用意ください

図５　タイトル

図５　説明（200 字以内）○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○．○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○．○○
○○○○○○○○．○○○
○○○○○○○．（文献 1 より引用）

用語解説（1 語あたり 50～150 字程度）

※1　ImageJ：○○○
○○○.

※2　二値化：○○
○○○.

文献（10～15 本：任意の文献スタイルでかまいません）

1） Honda M, et al：Exp Med, 800：800-804, 2016
2） Yamaguchi K：Exp Med, 800：800-804, 2016
3） Takagi R：Exp Med, 800：800-804, 2016
4）
5）
…

プログラムコードのライセンス

MIT ライセンス