|  |
| --- |
| ◆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](https://authors.yodosha.com/readme.html)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。

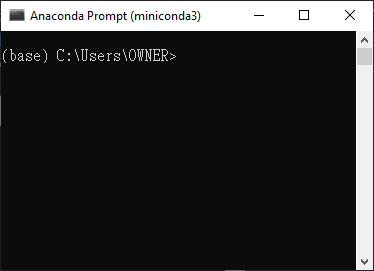


図１: Anaconda Promptのスタート画面

“(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。

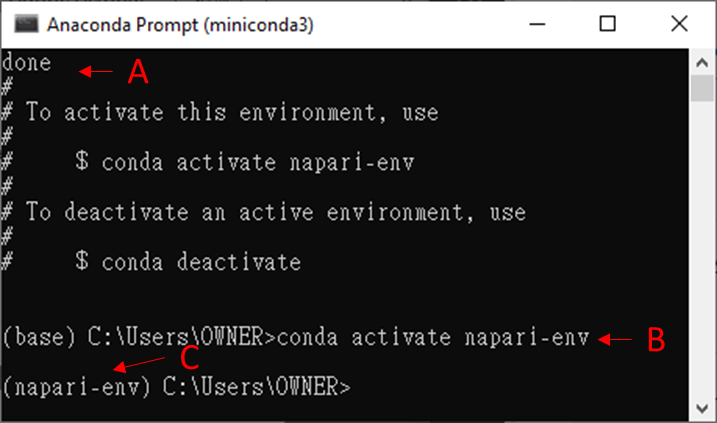


図2：Napari-env 仮想環境に入る

### 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。

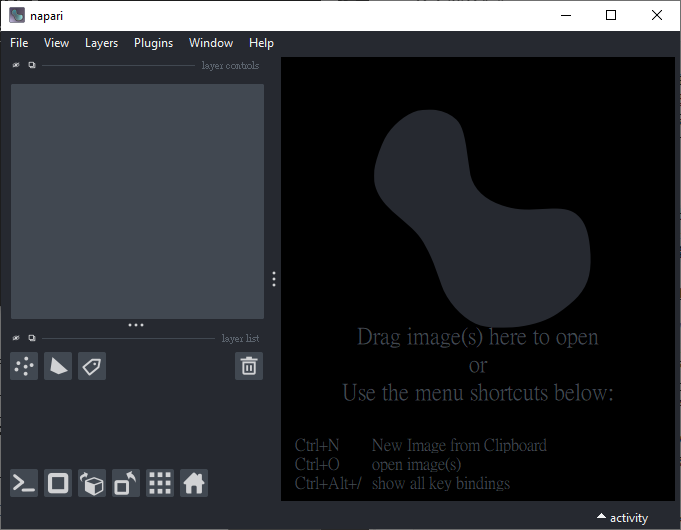


図３:Napari スタート画面

## 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。

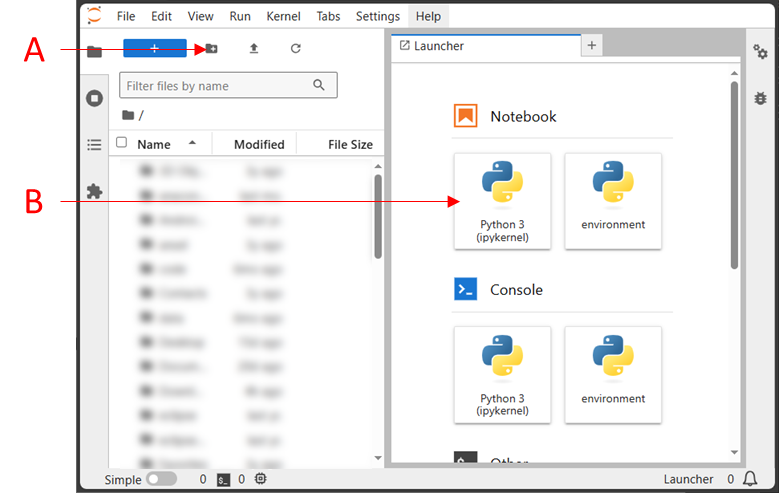


図4:Jupyter Lab スタートアップページ

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。

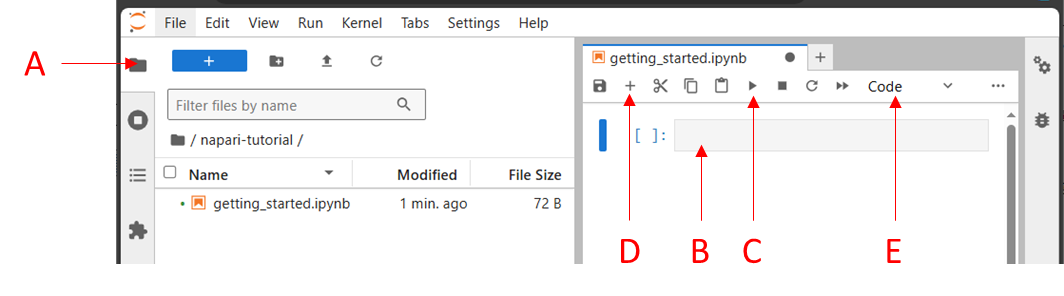


図５: Jupyter エディター

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５E)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。

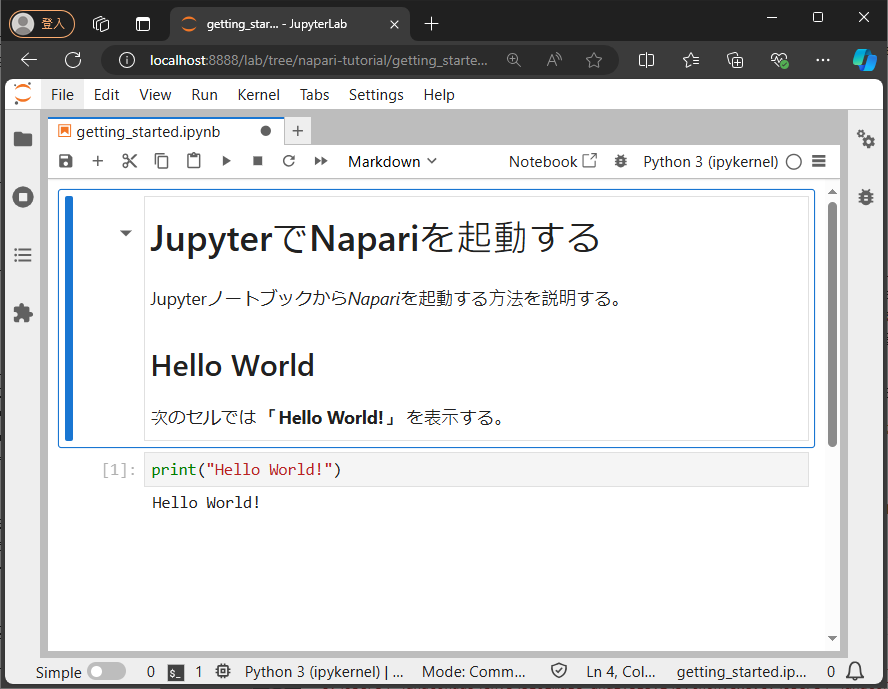


図6: Jupyter でのマークダウン編集

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。

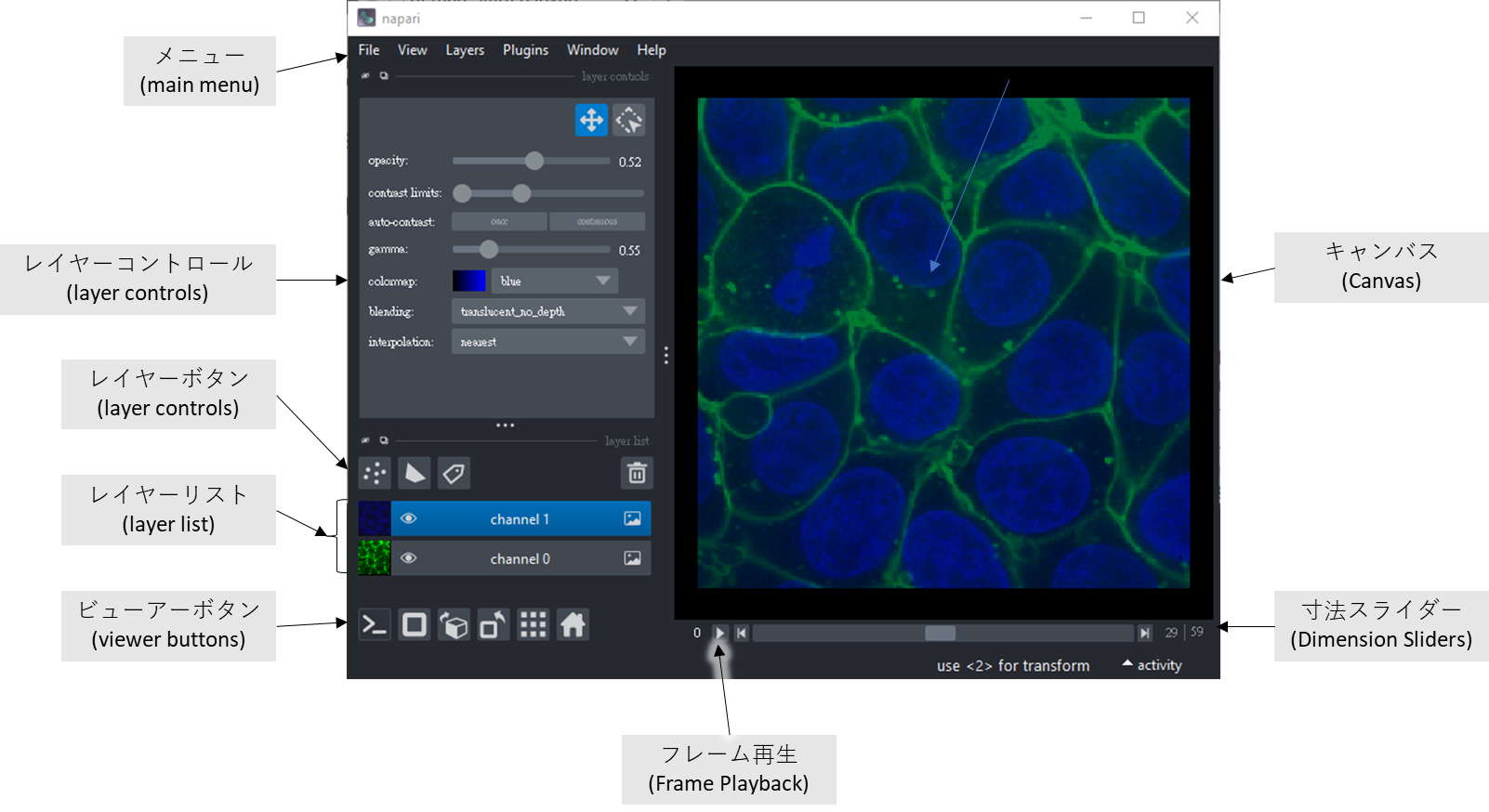


図7：Napariビューアー

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。

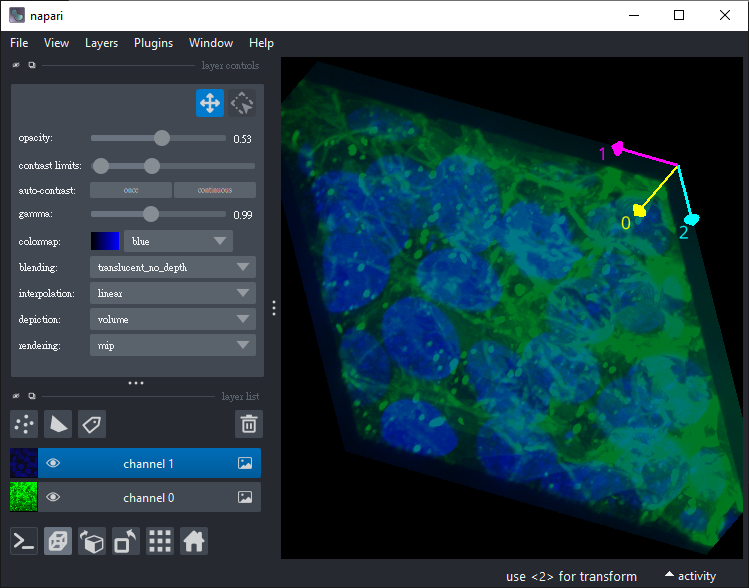


図８

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。

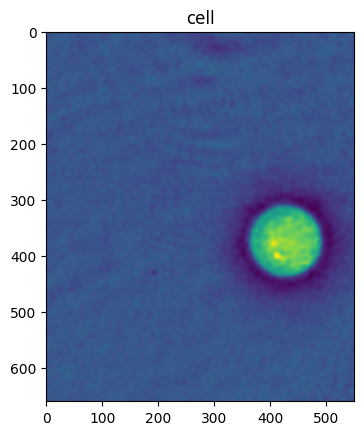


図９

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。

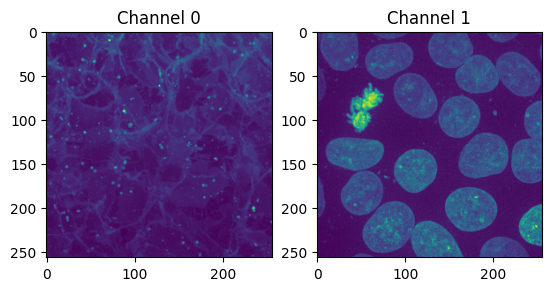


図１０

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。

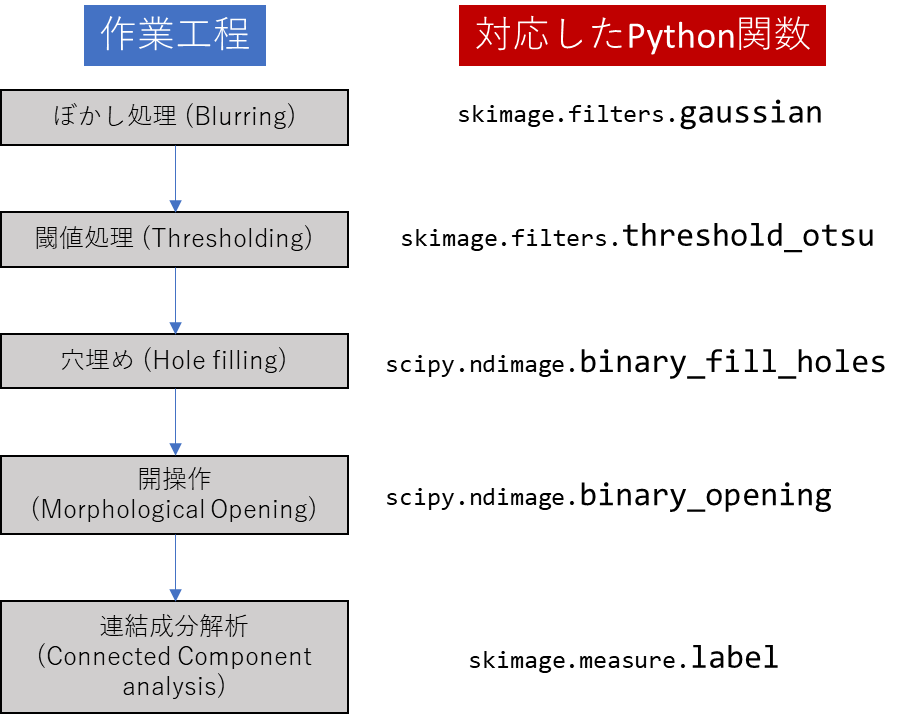


図１１:セグメンテーションワークフロー

|  |
| --- |
| # 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。

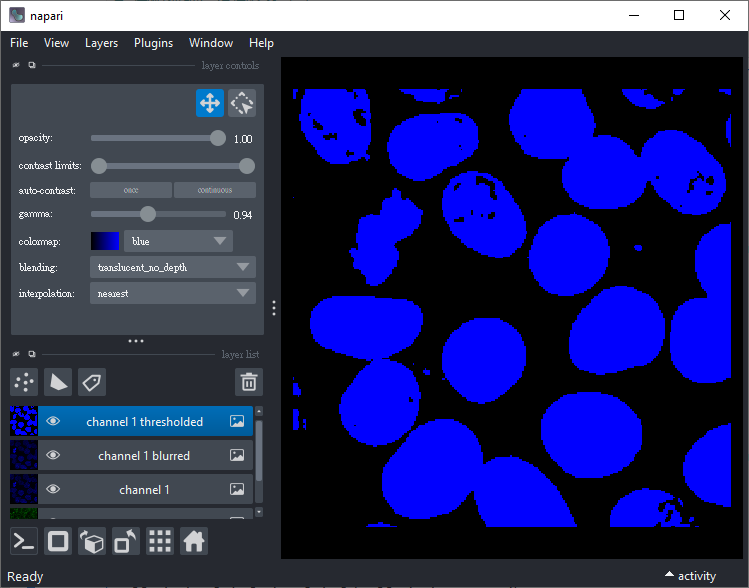


図１２

|  |
| --- |
| # 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。

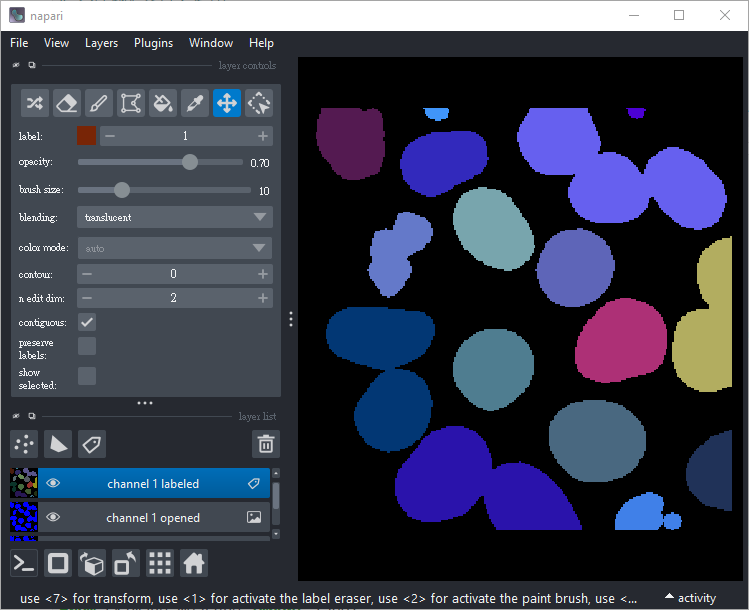


図１３：ラベリングの結果

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。

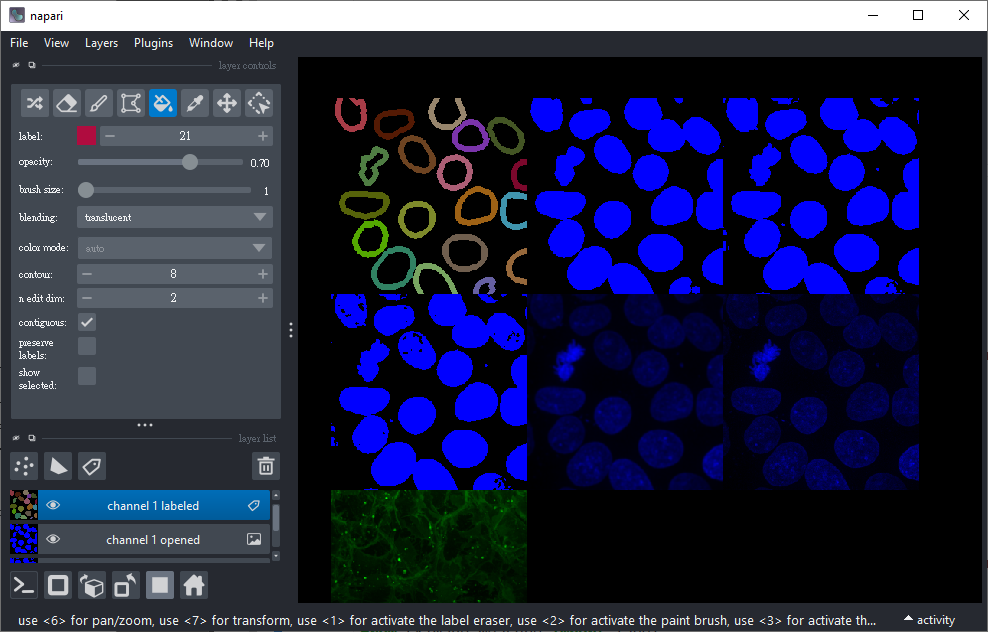


図１4

### 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。

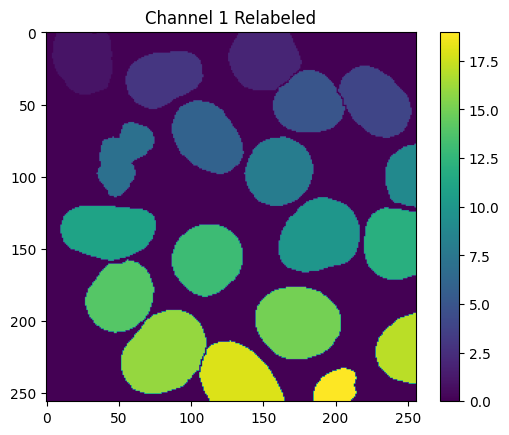


図１5

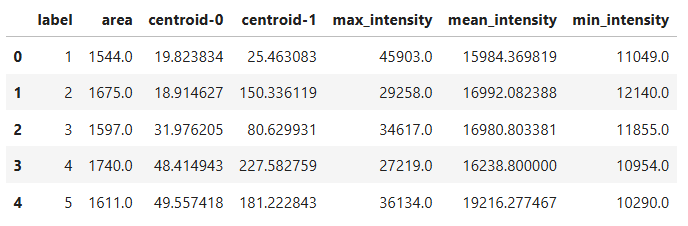
### Regionpropsの紹介

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。



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') |

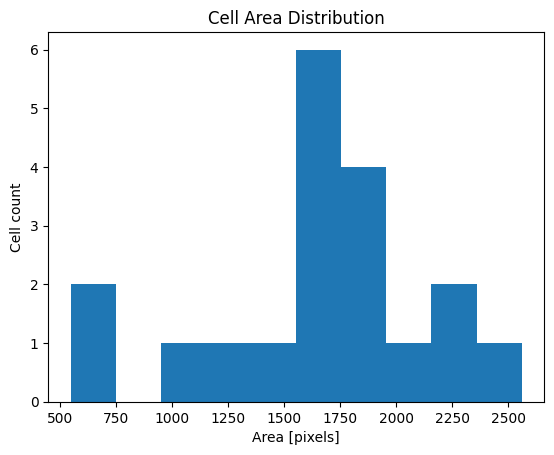


図1７:簡単な面積分布測定

## 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。

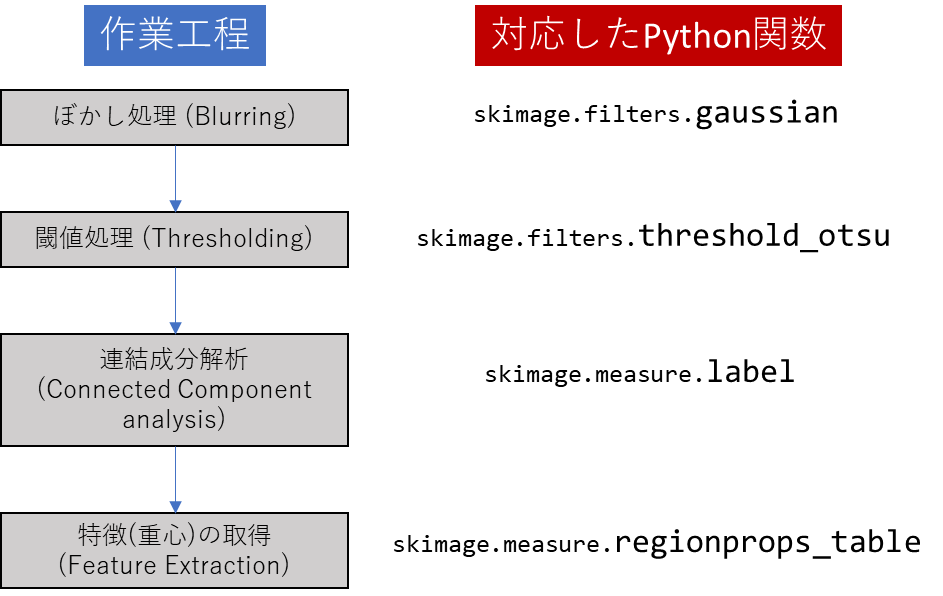


図18：粒子重心を検出するワークフロー

|  |
| --- |
| # 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。

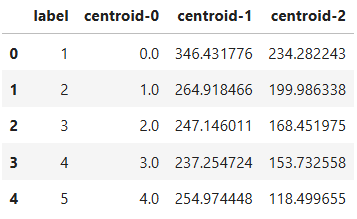


図19

### 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。

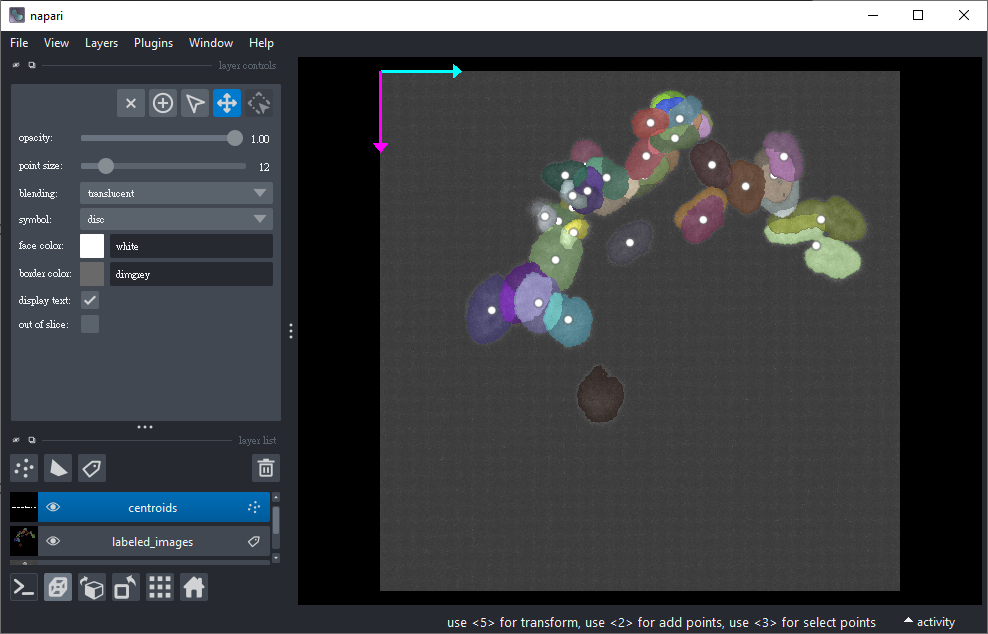


図20:ポイントレイヤーの表示

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。

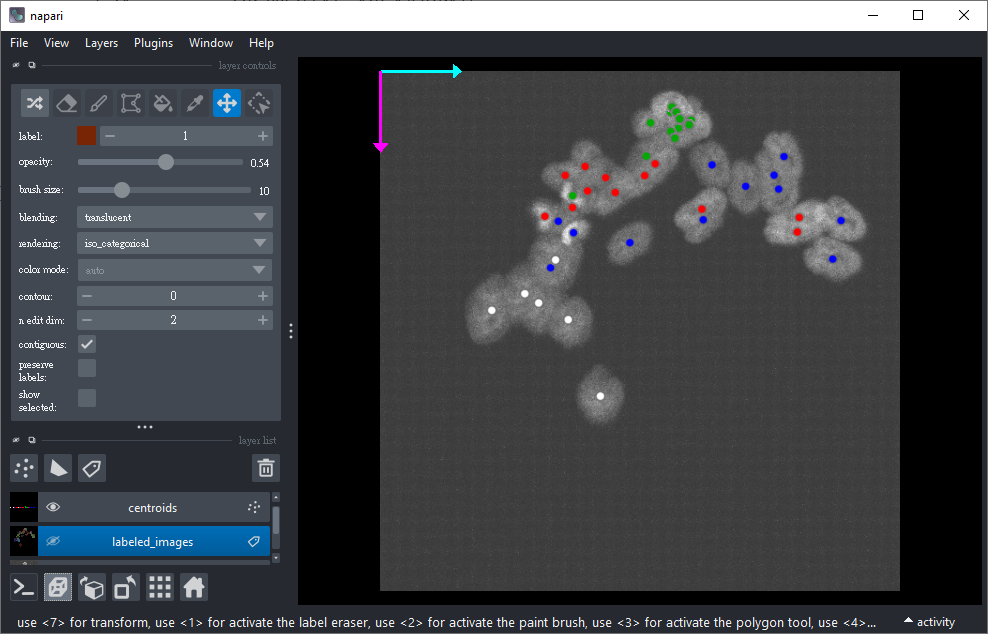


図2１: ポイントレイヤー修正後の結果

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。

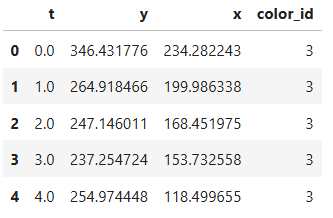


図２２

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() |



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。

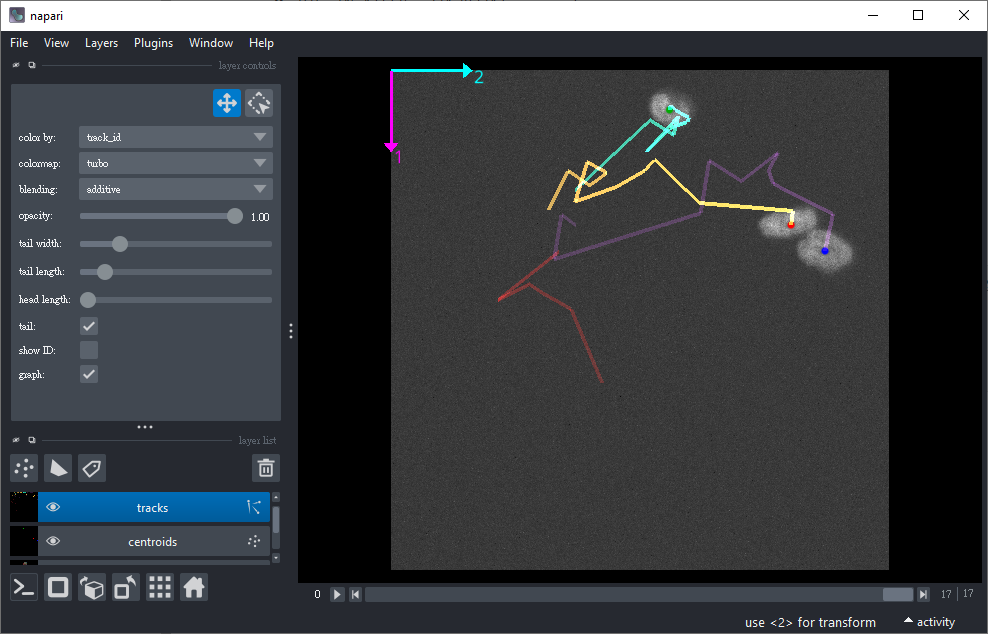


図２4:Tracks Layer の表示

### 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)。

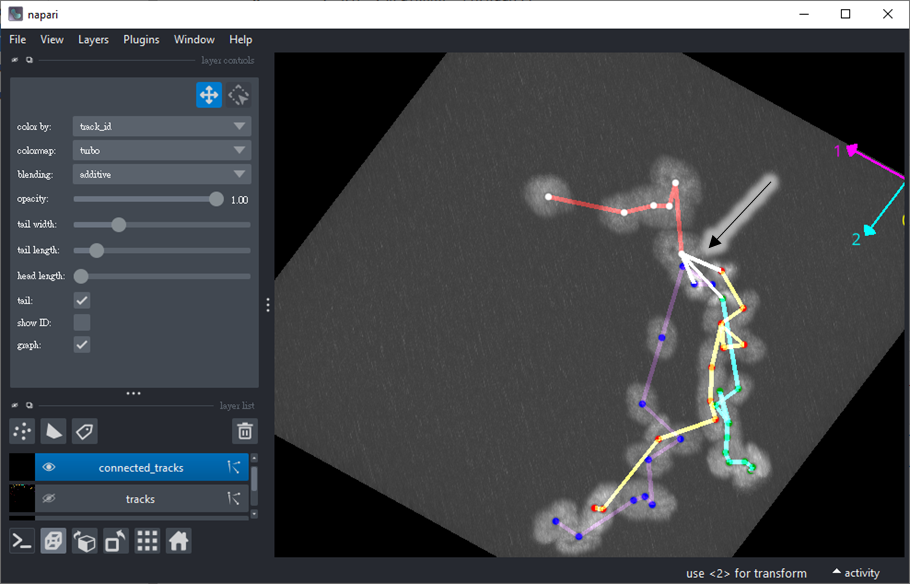


図２5:Tracksを3Dでの表示。分岐している所が連結されていることをハイライトしている。

|  |
| --- |
| # 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') |

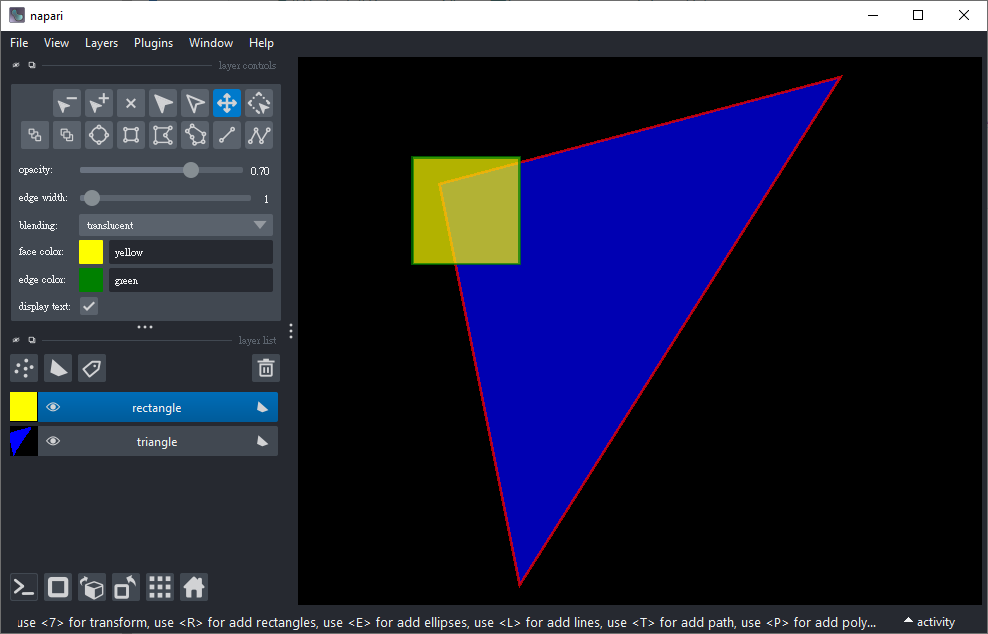


図２6: Napari ViewerのShapes Layer

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:

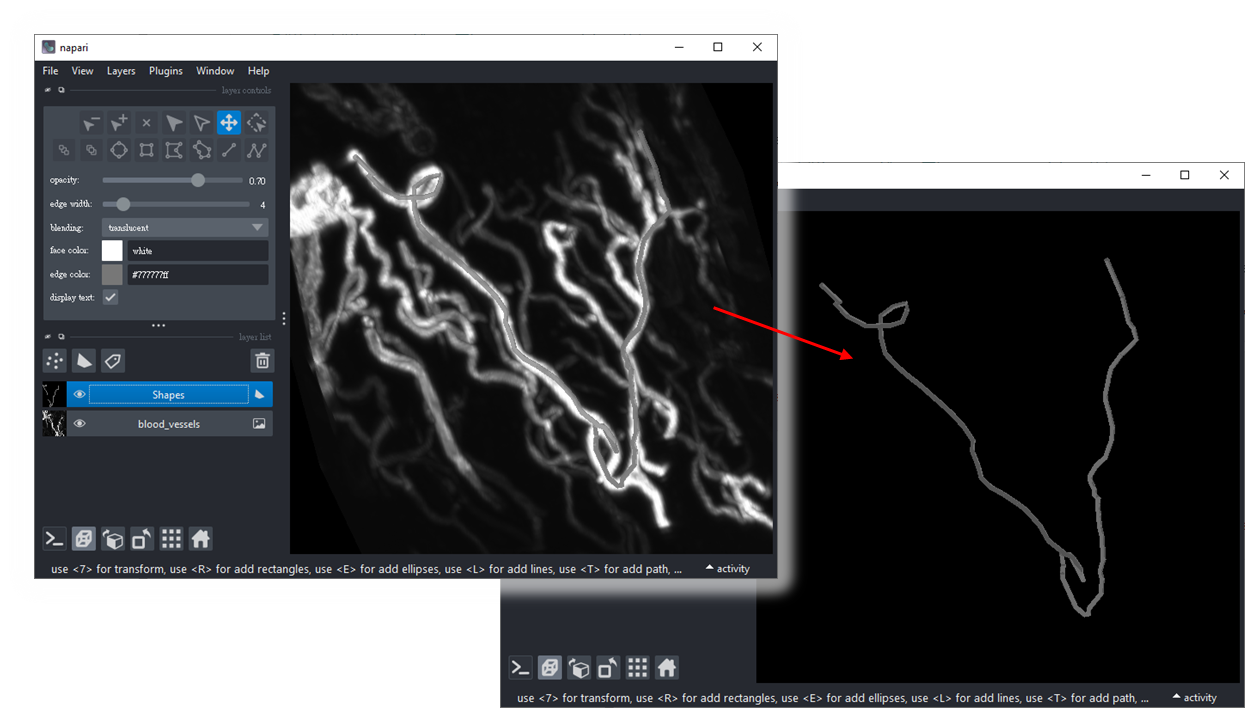


図２７:血管のトレーシングをNapari Shapes Layerで表示

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。

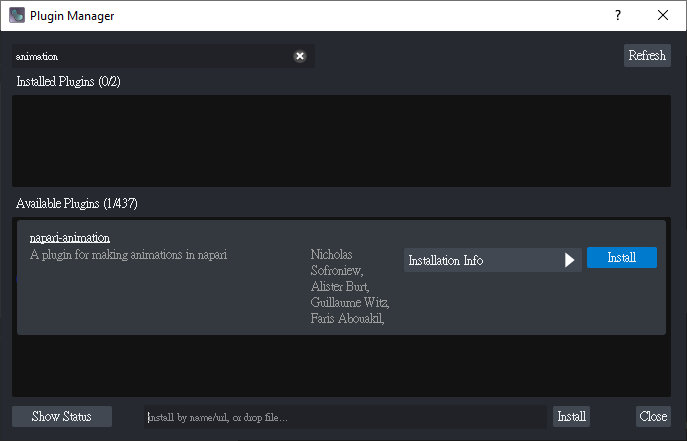


図２8:NapariのPlugin Installer

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。

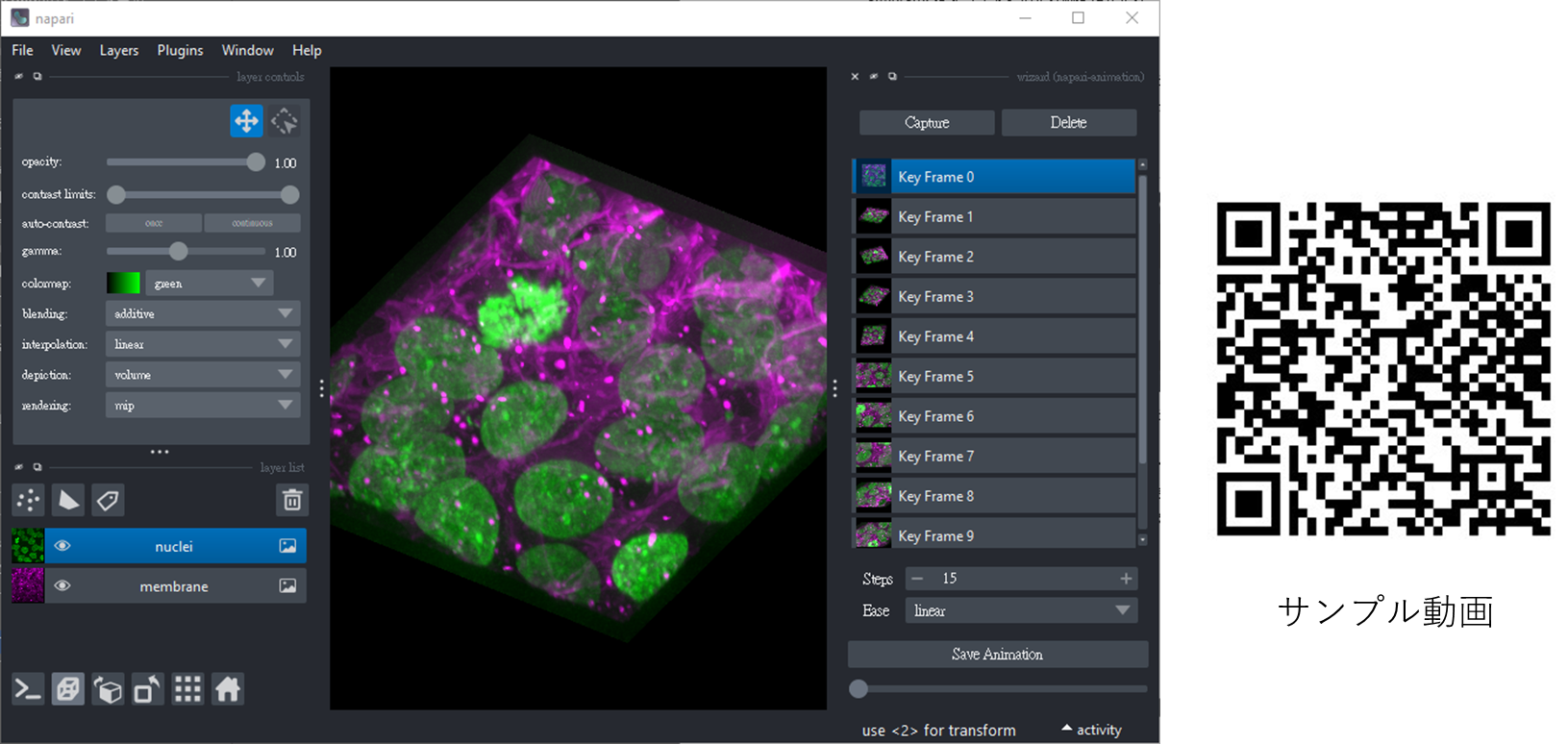


図２9:Napari-animation pluginのインターフェースとサンプル動画(QRコード)

## 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。

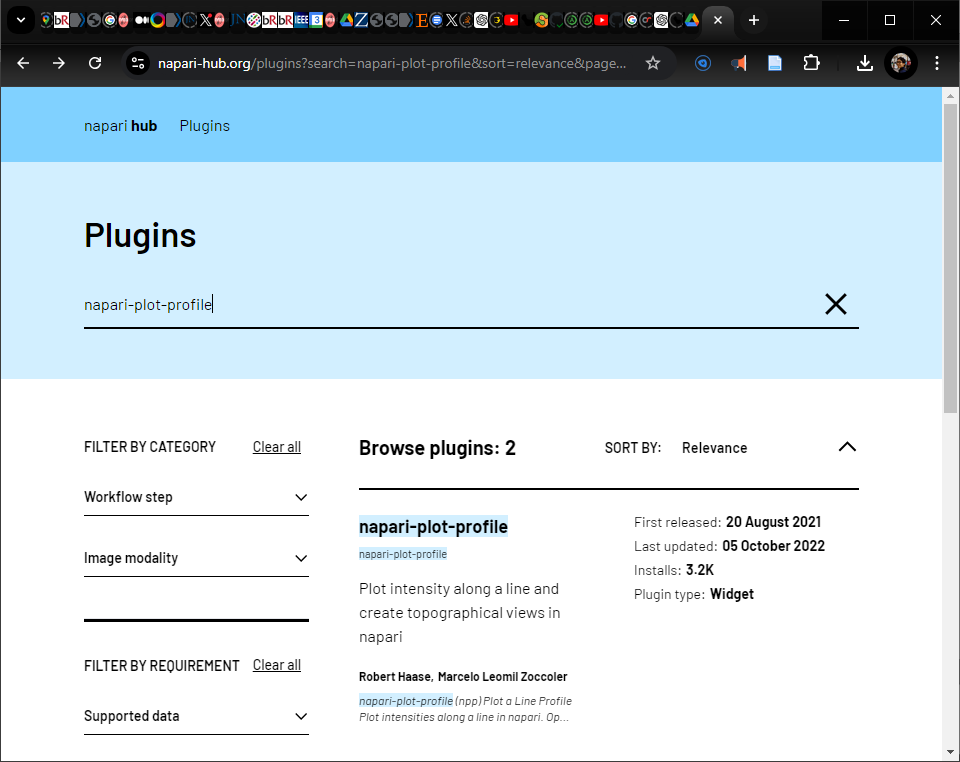


図30:Napari-hubでnapari-plot-profileを捜索

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](https://www.napari-hub.org/plugins/stardist-napari)、%60napari-n2v%60（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