pytrax Documentation

Release 0.1.0

PMEAL

Jul 06, 2019

Contents

CHAPTER 1

Getting Started

1.1 Requirements

Software: pytrax only relies on a few core python modules including Numpy and matplotlib, among others. These packages can be difficult to install from source, so it's highly recommended to download the Anaconda Python Distrubution install for your platform, which will install all of these packages for you (and many more!). Once this is done, you can then run the installation of pytrax as described in the next section.

Hardware: Although there are no technical requirements, it must be noted that working with large images (>500**3) requires a substantial computer, with perhaps 16 or 32 GB of RAM. You can work on small images using normal computers to develop work flows, then size up to a larger computer for application on larger images.

1.2 Installation

pytrax is available on the Python Package Index (PyPI) and can be installed with the usual `pip` command as follows:

```
pip install pytrax
```

When installing in this way, the source code is stored somewhere deep within the Python installation folder, so it's not convenient to play with or alter the code. If you wish to customize the code, then it might be better to download the source code from github into a personal directory (e.g C:\pytrax) then install as follows:

```
pip install -e C:\pytrax
```

The '-e' argument means that the package is 'editable' so any changes you make to the code will be available the next time that pytrax is imported.

1.3 Basic Usage

To use pytrax simply import it at the Python prompt:

```
>>> import pytrax as pt
```

At the moment the package is very lightweight and it is expected that users have their own images to analyze. For testing purposes we make use of another one of our packages called PoreSpy. As well as having lots of image analysis tools for porous media, PoreSpy also contains an image `generators` module to produce a sample image as follows:

```
>>> import porespy as ps
>>> image = ps.generators.blobs(shape=[100, 100])
```

Running the random walk simulation to estimate the tortuosity tensor is then completed with a few extra commands:

```
>>> rw = pt.RandomWalk(image)
>>> rw.run(nt=1000, nw=1000, same_start=False, stride=1, num_proc=None)
```

Here the RandomWalk class is instantiated with the image that we generated and run with some parameters: `nt` is the number of time steps, `nw` is the number of walkers, `same_start` sets the walkers to have the same starting position in the image and is `False` (by default), `stride` is the number of steps between successive saves for calculations and output and `num_proc` is the number of parallel processors to use (defaulting to half the number available).

1.4 Plotting Results

pytrax has some built in plotting functionality to plot the coordinates of the walkers and also the mean square displacement vs. time which can be viewed with the following commands:

```
>>> rw.plot_walk_2d(check_solid=True, data='t')
>>> rw.plot_msd()
```

The first plotting function plots the image and the walker steps and is colored by time step, changing the `data` argument to be `w` changes the color to walker index. The `check_solid` argument checks that the solid voxels in the image are not walked upon which is useful when changes to the code are made as a quick sense check. The walkers are free to leave the original image providing that there is a porous pathway at the edges. When this happens they are treated to be travelling in a reflected domain and the plotting function also displays this. The second plotting function shows the mean and axial square displacement and applies linear regression to fit a straight line with intercept through zero. The gradient of the slope is inversely proportional to the tortuosity of the image in that direction. This follows the definition of tortuosity being the ratio of diffusivity in open space to diffusivity in the porous media.

1.5 Exporting Results

For 3D images the `plot_walk_2d` function can be used to view a slice of the walk and image, however, for better visualization it is recommended to use the export function and view the results in Paraview. A tutorial on how to do this is provided but the following function will export the image and walker data:

```
>>> rw.export_walk(image=None, path=None, sub='data', prefix='rw_', sample=1)
```

The `image` argument optionally lets you export the original image or the larger reflected image which are both stored on the rw object as `rw.im` and `rw.im_big`, respectively. Leaving the argument as `None` will not export any image. `path` is the directory to save the data and when set to `None` will default to the current working directory, `sub` creates a subfolder under the path directory to save the data in and defaults to `data`, `prefix` gives all the data a prefix and defaults to `rw_` and finally `sample` is a down-sampling factor which in addition to the stride function in the run command will only output walker coordinates for time steps that are multiples of this number.

CHAPTER 2

User Guide

CHAPTER 3

Examples

The following 4 examples demostrate the use of pytrax for different types of image:

3.1 Example 1: A Random Walk in Open Space

This example is the simplest use of pytrax but also illustrates an underlying theory of diffusion which is that the mean square displacement of diffusing particles should grow linearly with time.

Topics Covered in this Tutorial

Learning Objectives

1. Introduce the main class in the pytrax package, RandomWalk
2. Run the RandomWalk for an image devoid of solid features to demonstrate the principles of the package
3. Produce some visualization

Hint: Python and Numpy Tutorials

- pytrax is written in Python. One of the best guides to learning Python is the set of Tutorials available on the official Python website). The web is literally overrun with excellent Python tutorials owing to the popularity and importance of the language. The official Python website also provides an long list of resources
- For information on using Numpy, Scipy and generally doing scientific computing in Python checkout the Scipy lecture notes. The Scipy website also offers as solid introduction to using Numpy arrays.

- The Stackoverflow website is an incredible resource for all computing related questions, including simple usage of Python, Scipy and Numpy functions.
- For users more familiar with Matlab, there is a Matlab-Numpy cheat sheet that explains how to translate familiar Matlab commands to Numpy.

3.1.1 Instantiating the RandomWalk class

The first thing to do is to import the packages that we are going to use. Start by importing pytrax and the Numpy package:

```
>>> import pytrax as pt
>>> import numpy as np
```

Next, in order to instaintiate the RandomWalk class from the pytrax package we first need to make an binary image where 1 denotes open space available to walk on and 0 denotes a solid obstacle. In this example we are going set our walkers to explore open space and so we can build an image only containing ones. The size of the image doesn't matter, which will be explained later but for demostration purposes we will make the image two-dimensional:

```
>>> image = np.ones(shape=[3, 3], dtype=int)
>>> rw = pt.RandomWalk(image=image, seed=False)
```

We now have a RandomWalk object instantiated with the handle `rw`.

- The `image` argument sets the domain of the random walk and is stored on the object for all future simulations.
- The `seed` argument controls whether the random number generators in the class are seeded which means that they will always behave the same when running the walk multiple times with the same parameters. This is useful for debugging but under all other circumstances as it results in only semi-random walks.

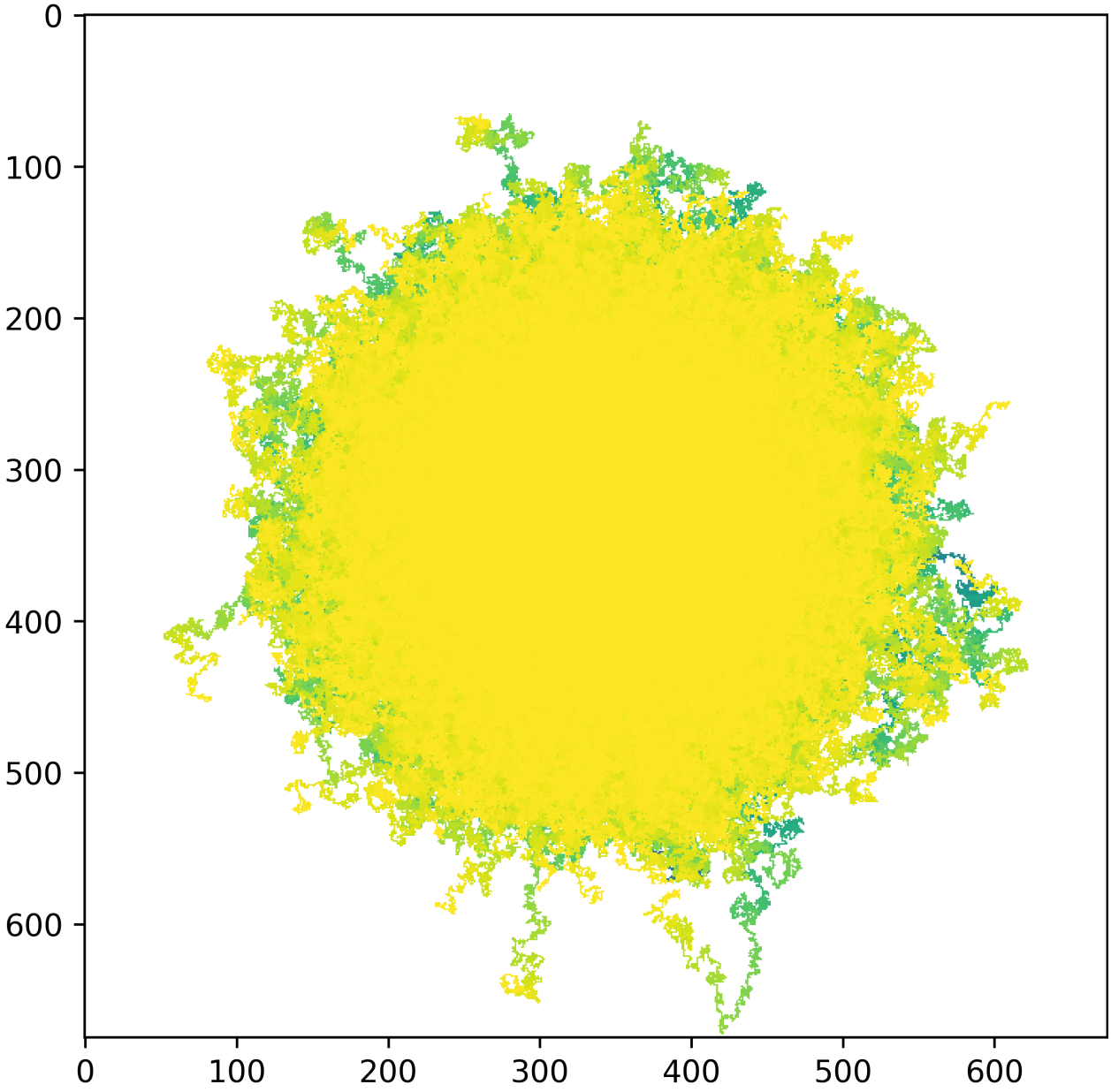
3.1.2 Running and ploting the walk

We're now ready to run the walk:

```
>>> rw.run(nt=1000, nw=1000, same_start=False, stride=1, num_proc=1)
>>> rw.plot_walk_2d()
```

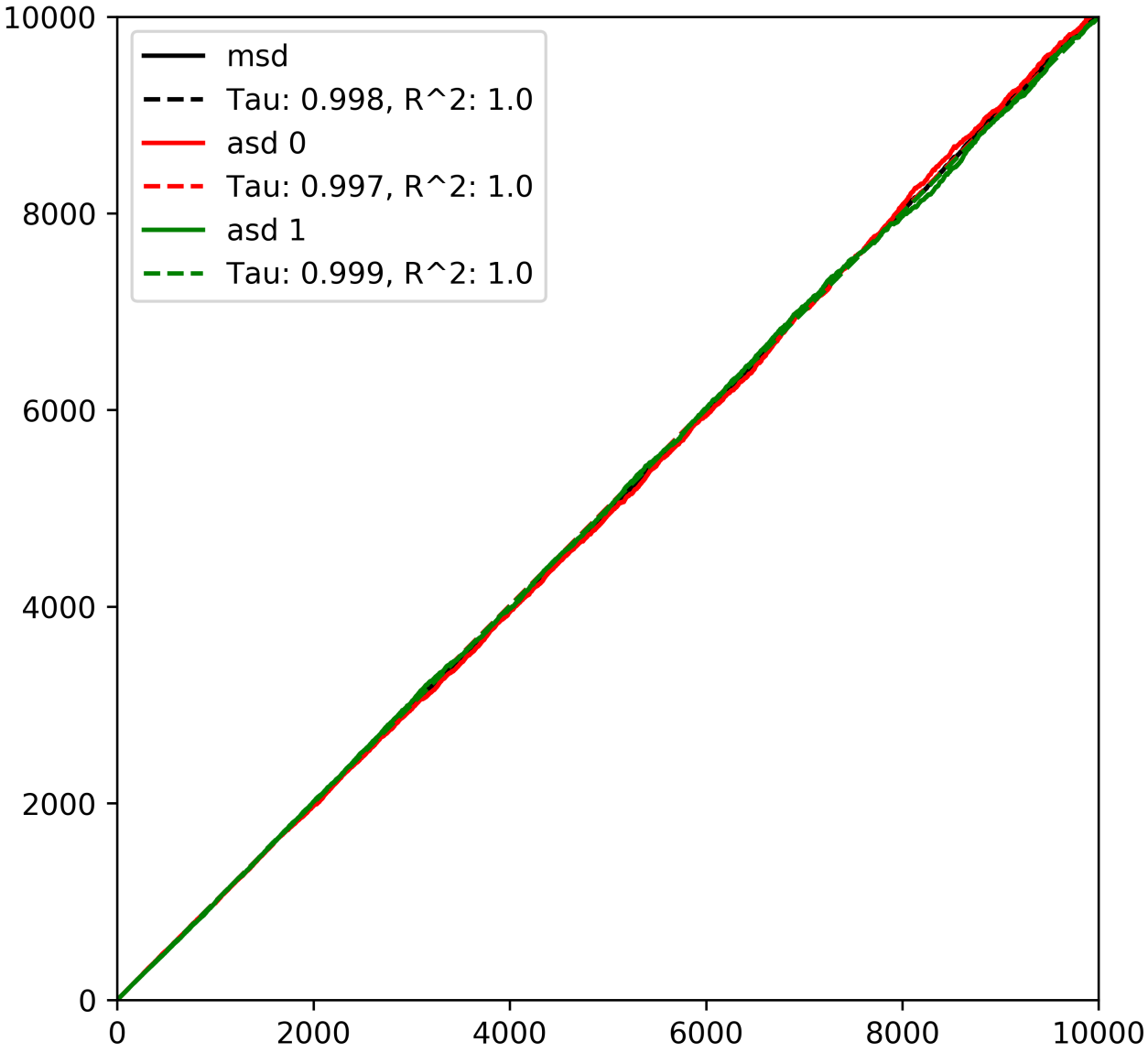
- `nt` is the number of steps that each walker will take
- `nw` is the number of walkers to run concurrently
- `same_start` is a boolean controlling whether the walkers all start at the same spot in the image. By default this is False and this will result in the walkers being started randomly at different locations.
- `stride` is a reporting variable and does not affect the length of the strides taken by each walker (which are always one voxel at a time), but controls how many steps are saved for plotting and export.
- `num_proc` sets the number of parallel processors to run. By default half the number available will be used and the walkers will be divided into batches and run in parallel.

Each walk is completley independent of any other which sounds strange as Brownian motion is intended to simulate particle-particle interactions. However, we are not simulating this directly but encompassing the behaviour by randomly changing the direction of the steps taken on an individual walker basis. The second line should produce a plot showing all the walkers colored by timestep, like the one below:



The appearance of the plot tells us a few things about the process. The circular shape and uniform color shows that the walkers are evenly distributed and have walked in each direction in approximately equal proportions. To display this information more clearly we can plot the mean square displacement (MSD) over time:

```
>>> rw.plot_msd()
```



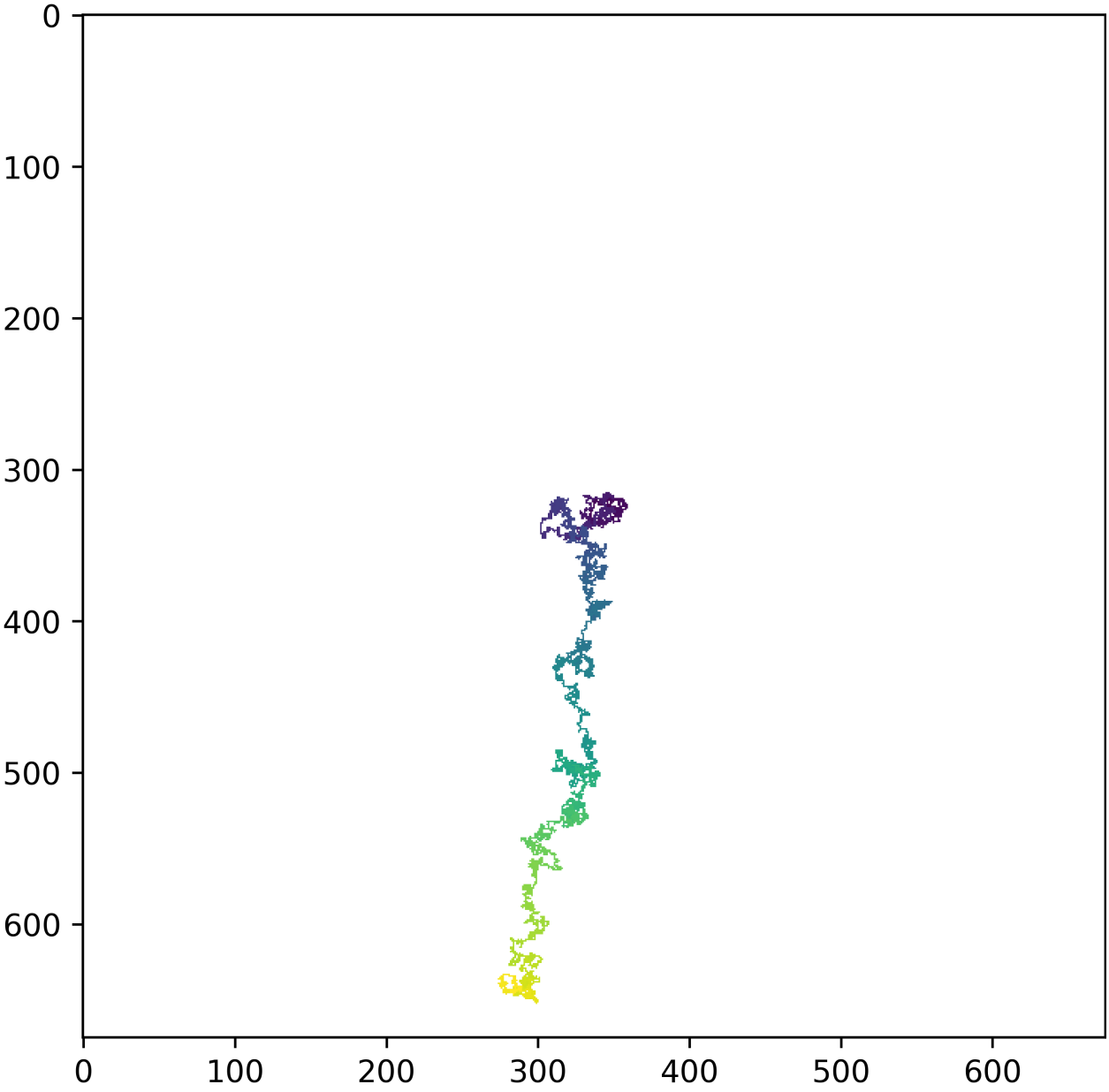
The `plot_msd` function shows that mean square displacement and axial displacement are all the same and increase linearly with time. A neat explanation of why this is can be found in this paper http://rsif.royalsocietypublishing.org/cgi/doi/10.1098/rsif.2008.0014 which derives the probability debnsity function for the location of a walker after time `t` as:

..math:

```
p(x,t) = \frac{1}{\sqrt{4\piDt}exp\left(\frac{-x^2}{4Dt}\right)
```

Which is the fundamental solution to the diffusion equation and so walker positions follow a Gaussian distribution which spreads out and has the property that MSD increases linearly with time. pytrax makes use of this property to calculate the toruosity of the image domain by using the definition that tortuosity is the ratio of diffusion in a porous space compared with that in open space. This simply translates to the reciprocal of the slope of the MSD which is unity for open space, as shown by this example. As a result of plotting the MSD we have some extra data on the RandomWalk object and we can use it to find the walker that travelled the furthest:

```
>>> rw.plot_walk_2d(w_id=np.argmax(rw.sq_disp[-1, :]), data='t')
```



The attribute `rw.sq_disp` is the square displacement for all walkers at all stride steps which is all steps for this example. Indexing `-1` takes the last row and indexing `:` takes the whole row, the numpy function `argmax` returns the index of the largest value and this integer value is used for the `w_id` argument of the plotting function which stands for walker index.

3.1.3 A note on the image boundaries

As mentioned previously, the size of the image we used to instantiate the RandomWalk class for this example did not matter. This is because the walkers are allowed to leave the domain if there is a path in open space allowing them to do so. The image is treated as a representative sample of some larger medium and if the walkers were not allowed to leave the original domain their MSD's would eventually plateau and this would not be represtative of the general diffusive behaviour. The plotting function is actually showing an array of real and reflected domains with the original at the center, although this is hard to see with this example as there are no solid features and so the reflected images are identical to the original. We will discuss more on this later.

3.2 Example 2: The Tortuosity of Tau

This example shows the code working on a pseudo-porous media and explains some of the things to be careful of.

Topics Covered in this Tutorial

Learning Objectives

1. Practice the code on a real image
2. Explain the significance of the number of walkers and steps.
3. Visualize the domain reflection

3.2.1 Obtaining the image

As with the previous example, the first thing to do is to import the packages that we are going to use including some packages for importing the image we will use:

```
>>> import pytrax as ps
>>> import numpy as np
>>> import urllib.request as ur
>>> from io import BytesIO
>>> import matplotlib.pyplot as plt
>>> from PIL import Image
```

Next we are going to grab an image using its URL and make some modifications to prepare it for pytrax:

```
>>> url = 'https://i.imgur.com/nrEJRDf.png'
>>> file = BytesIO(ur.urlopen(url).read())
>>> im = np.asarray(Image.open(file))[:, :, 3] == 0
>>> im = im.astype(int)
>>> im = np.pad(im, pad_width=50, mode='constant', constant_values=1)
```

The image is a .png and has 4 layers: r, g, b and alpha. We use the alpha layer which sets the contrast and make the image binary by setting the zero-valued pixels to `True` then converting to a type `int`. Finally we pad the image with additional pore space which is designated as 1.

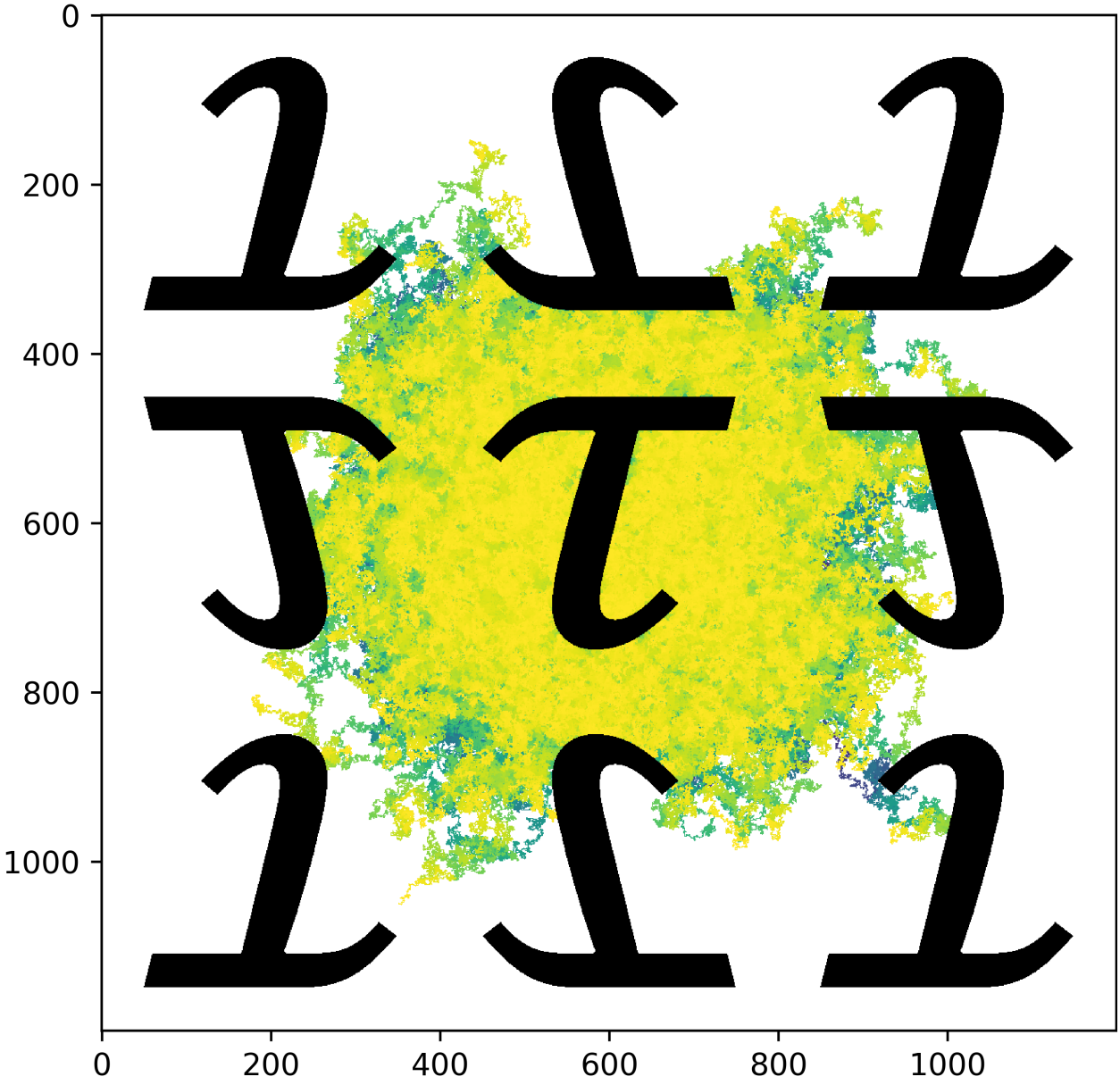
```
>>> rw = pt.RandomWalk(image=ima, seed=False
```

3.2.2 Running and ploting the walk

We're now ready to run the walk setting the number of walkers to be 1,000 and the number of steps to be 20,000:

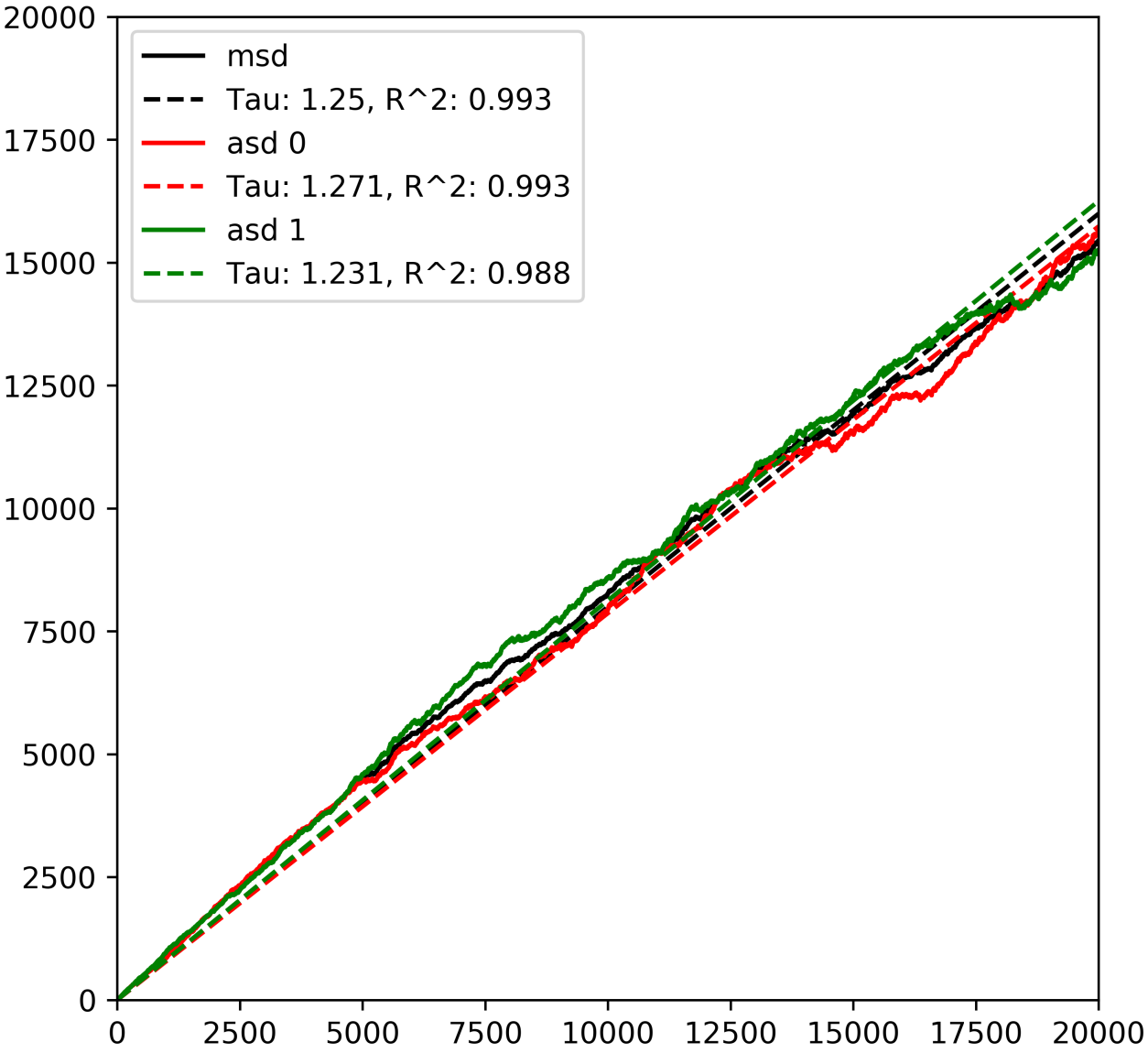
```
>>> rw.run(nt=20000, nw=1000)
>>> rw.plot_walk_2d()
```

The following 2D plot is produced:



This time it is clear how the domain reflections happen with the original in the center surrounded by reflections of the Tau symbol along the two principal axes. Even though the number of steps taken by the walkers seems quite large, the number of pixels in the image is also large and the length of the average walk is no larger than the original domain. Let's plot the MSD to get a little more information:

```
>>> rw.plot_msd()
```



The `plot_msd` function shows that mean square displacement and axial displacement are increasing over time but not in such a linear fashion as the previous example. There also appears to be some anisotropy as the 0th axis has a different tortuosity to the 1st axis. The MSD plot shows that we should run the simulation again for longer as the length of the walk is about the same length as the average pore size. It really needs to be at least 4 or 5 times longer so that the effect of hitting pore walls is evenly distributed among the walkers over time. The number of walkers should also be increased as we can see that the MSD is not particularly smooth and creating a larger ensemble average will give better results. Try running the same simulation again but increasing both `nt` and `nw` by a factor of 4.

3.3 Example 3: The Sierpinksi Carpet

This example will demonstrate the principle of calculating the tortuosity from a porous image with lower porosity.

Topics Covered in this Tutorial

- *Example 3: The Sierpinksi Carpet*

Learning Objectives

1. Generate a fractal image with self-similarity at different length scales
2. Run the RandomWalk for a fractal image
3. Produce some visualization

3.3.1 Instantiating the RandomWalk class

Assuming that you are now familiar with how to import and instantiate the simulation objects we now define a function to produce the Sierpinski carpet:

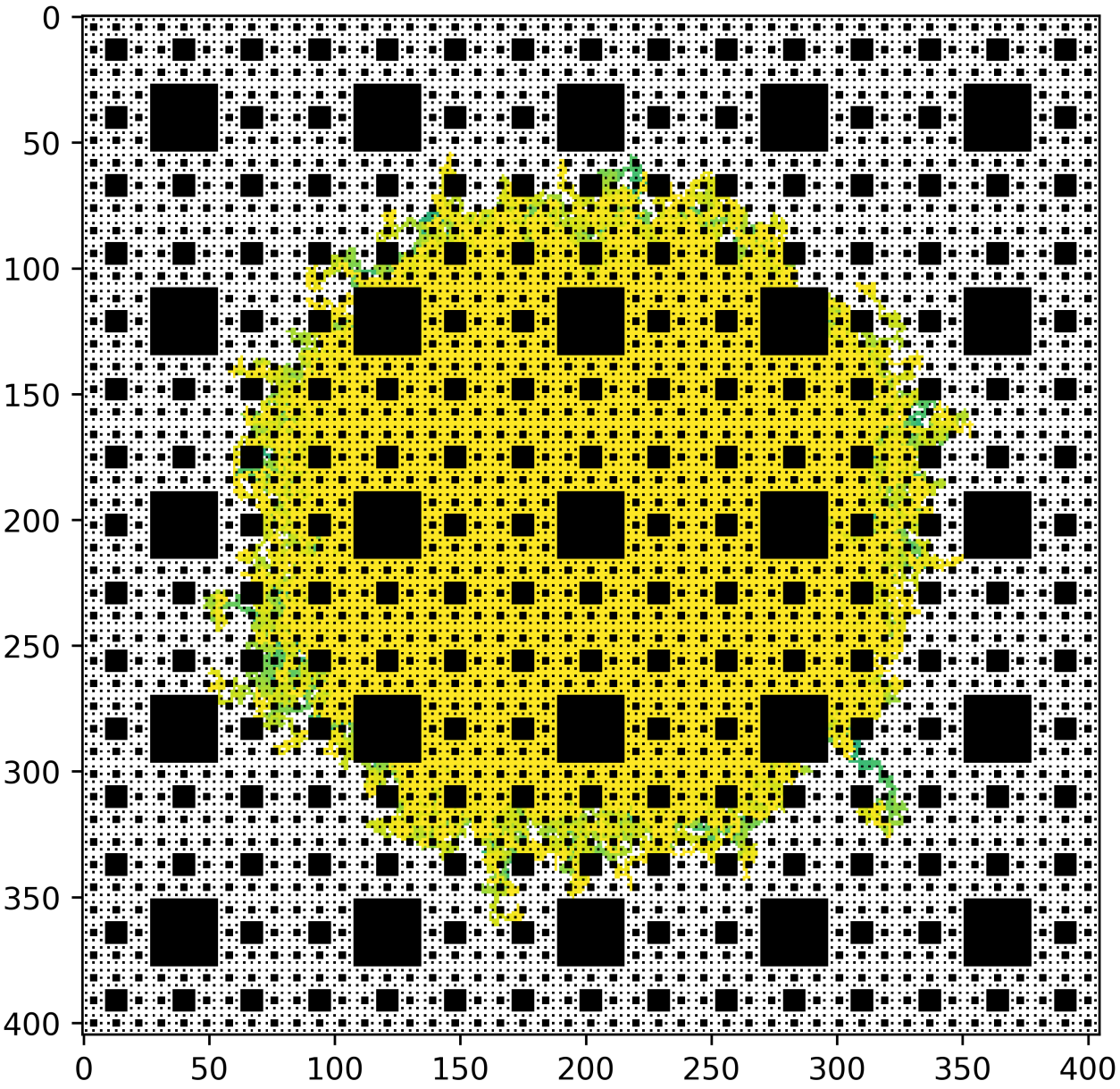
```
>>> def tileandblank(image, n):
>>>     if n > 0:
>>>         n -= 1
>>>         shape = np.asarray(np.shape(image))
>>>         image = np.tile(image, (3, 3))
>>>         image[shape[0]:2*shape[0], shape[1]:2*shape[1]] = 0
>>>         image = tileandblank(image, n)
>>>     return image
>>> im = np.ones([1, 1], dtype=int)
>>> im = tileandblank(im, 4)
```

3.3.2 Running and ploting the walk

We're now ready to instantiate and run the walk, this time lets test the power of the program and run it for 1 million walkers using 10 parallel processes (please make sure you have that many avaialble):

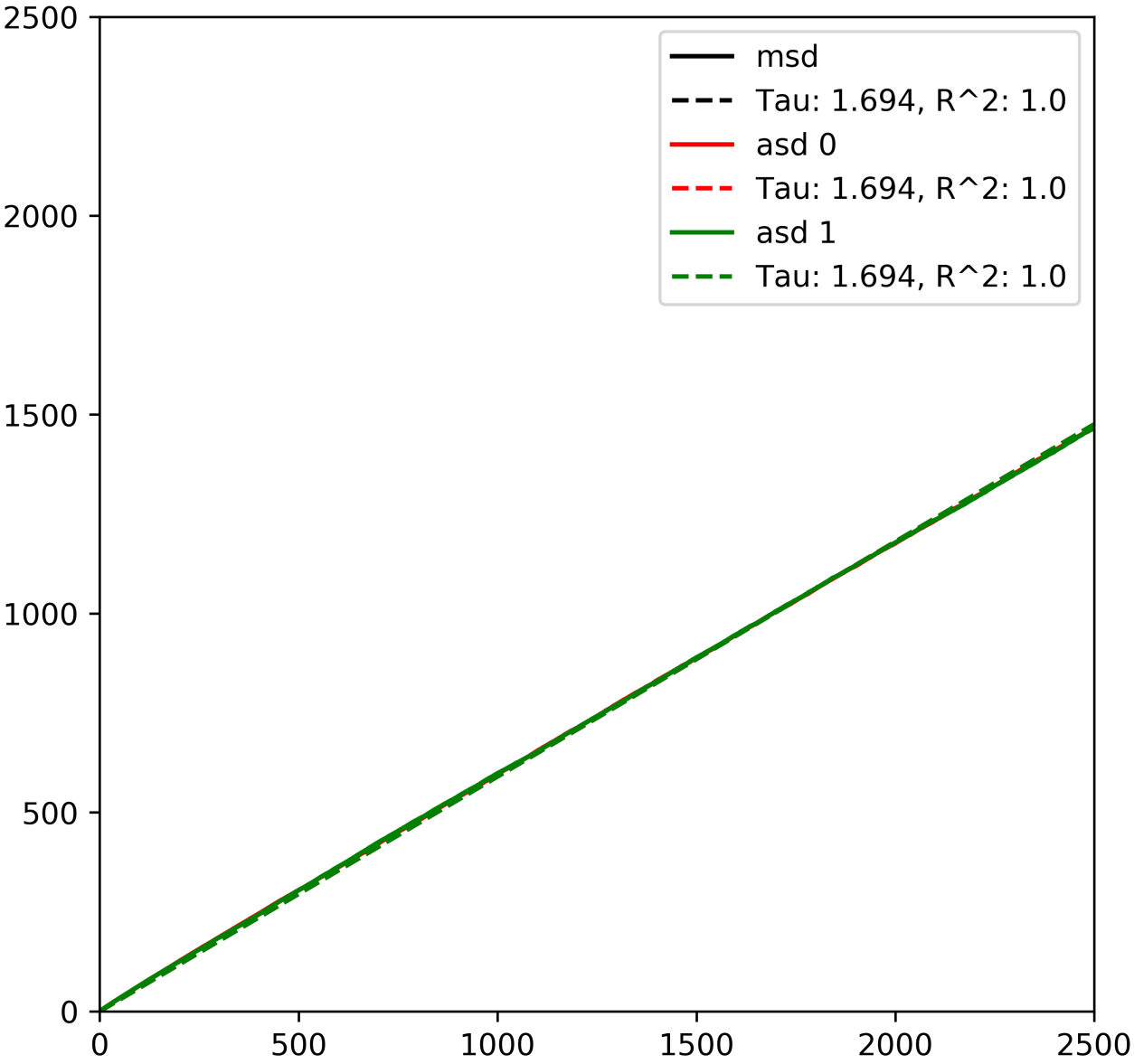
```
>>> rw = pt.RandomWalk(im)
>>> rw.run(nt=2500, nw=1e6, same_start=False, stride=5, num_proc=10)
>>> rw.plot_walk_2d()
```

The simulation should take no longer than a minute when running on a single process and should produce a plot like this:



Like the open space example the pattern is radial because the image is isotropic. We can see the reflected domains with the large black squares at the center. This time the walkers have escaped the original domain and some have travelled entirely through the next set of neighboring reflected domains and reached a third relflection. This signifies that the domain has been probed effectively over time. The MSD plot is as follows:

```
>>> rw.plot_msd()
```



The `plot_msd` function shows that mean square displacement and axial displacement are all the same and increase linearly with time. However, unlike the open space example the slope of the curve is less than one. This is because the walkers are impeded by the solid objects and it takes a longer time to go around them than in open space. The toruosity is calculated as the reciprocal of the MSD slope and is equal in both directions and very straight signifying that we have chosen an adequate number of walkers and steps.

3.4 Example 4: Anisotropic 3D Blobs

This example will demonstrate the principle of calculating the tortuosity from a 3D porous image with anisotropy.

Topics Covered in this Tutorial

- *Example 4: Anisotropic 3D Blobs*

Learning Objectives

1. Generate an anisotropic 3D imgage with the porespy package
2. Run the RandomWalk for the image showing the anisotropic tortuosity
3. Export the results and visualize with Paraview

3.4.1 Generating the Image with porespy

In this example we generate a 3D anisotropic image with another PMEAL package called porespy which can be installed with `pip`. The image will be 300 voxels cubed, have a porosity of 0.5 and the blobs will be stretched in each principle direction by a different factor or [1, 2, 5]:

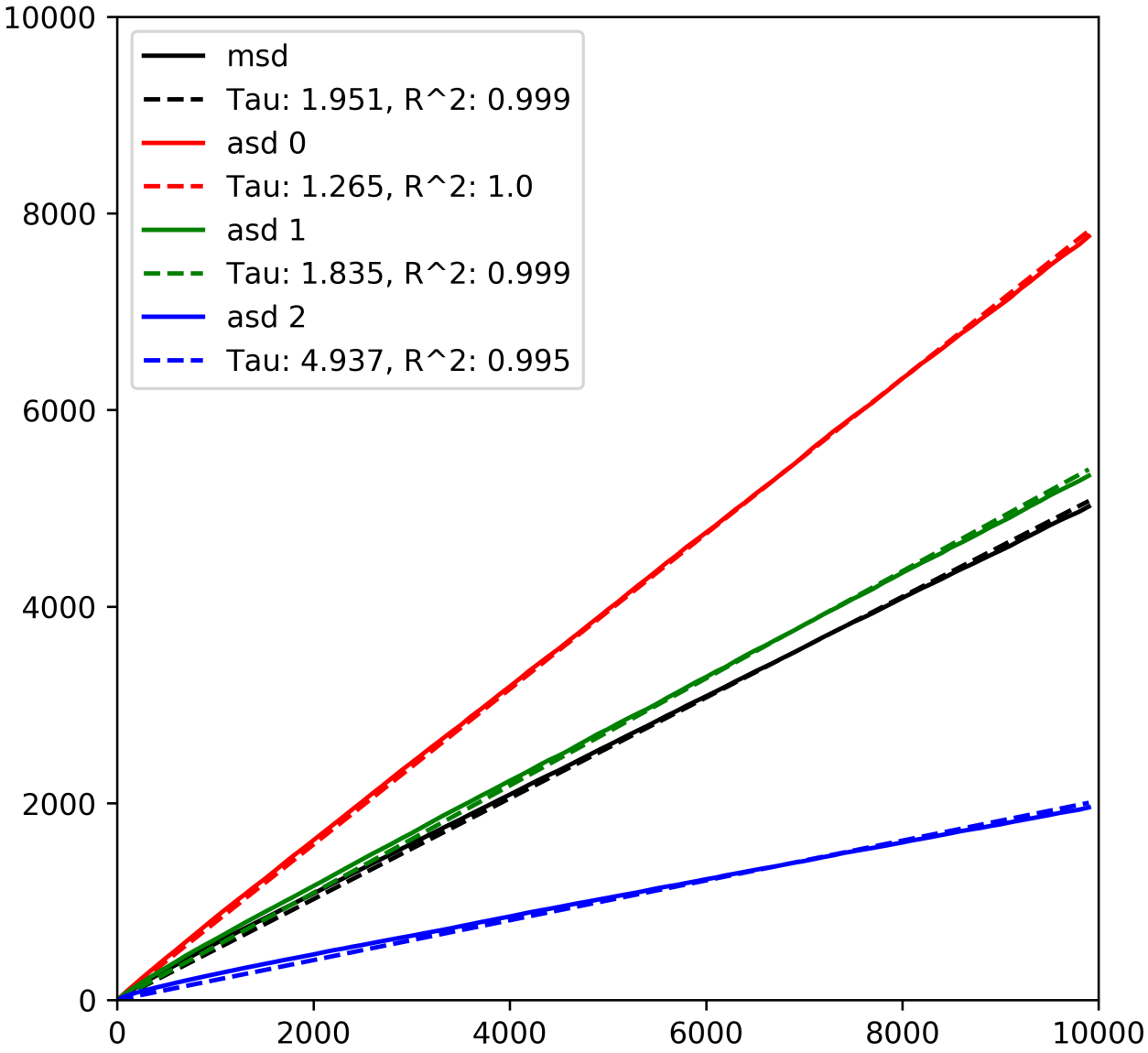
```
>>> import porespy as ps
>>> im = ps.generators.blobs(shape=[300], porosity=0.5, blobiness=[1, 2, 5]).
↪astype(int)
```

3.4.2 Running and exporting the walk with Paraview

We're now ready to instantiate and run the walk:

```
>>> rw = pt.RandomWalk(im)
>>> rw.run(nt=1e4, nw=1e4, same_start=False, stride=100, num_proc=10)
>>> rw.plot_msd()
```

The simulation should take no longer than 45 seconds when running on a single process and should produce an MSD plot like this:



Unlike the previous examples, the MSD plot clearly shows that the axial square displacement is different along the different axes and this produces a tortuosity that approximately scales with the blobiness of the image. The image is three-dimensional and so we cannot use the 2D plotting function to visualize the walks, instead we make use of the export function to produce a set of files that can be read with Paraview:

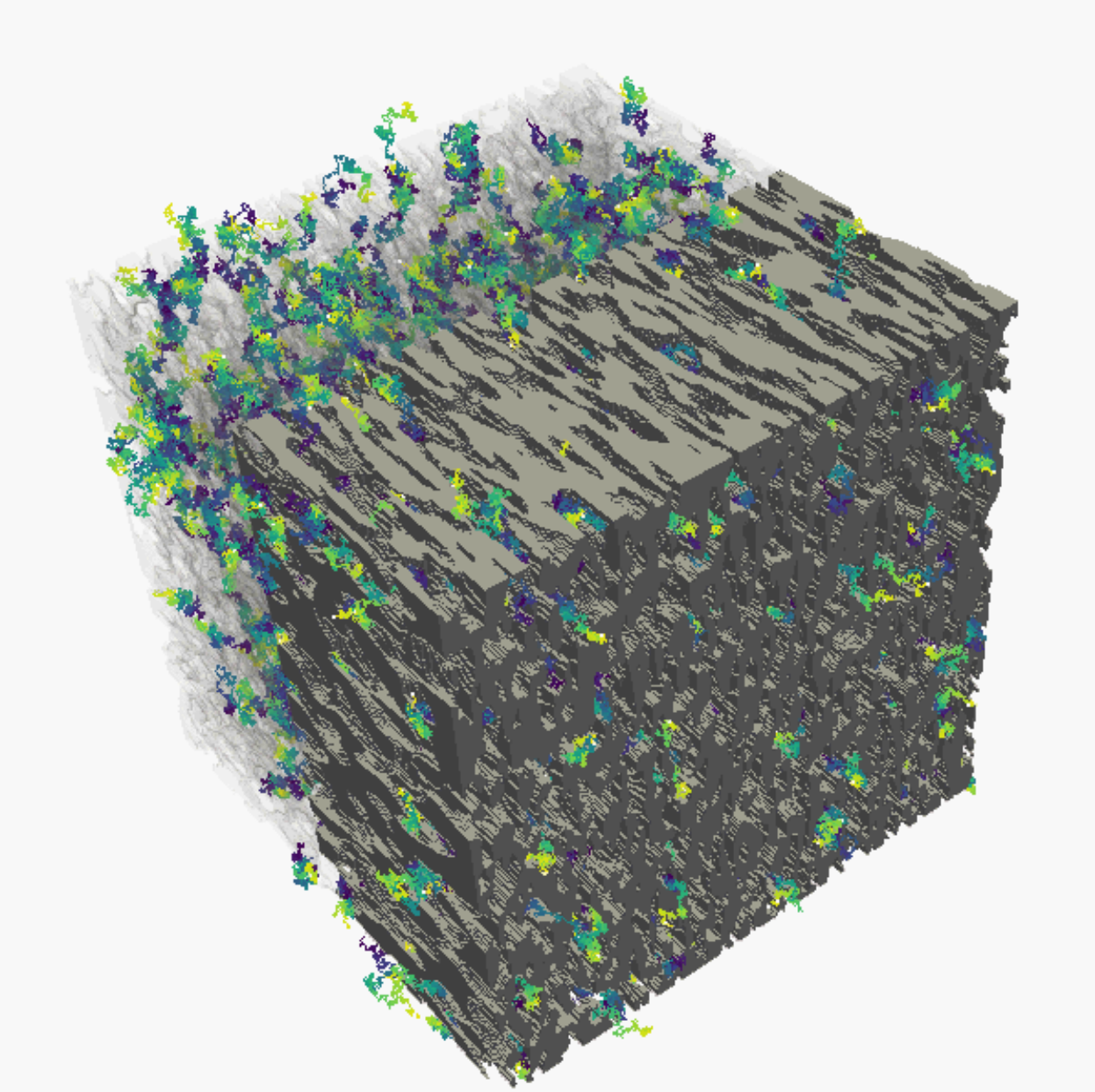
```
>>> rw.export_walk(image=rw.im, sample=1)
```

This arguments `image` sets the image to be exported to be the original domain, optionally we could leave the argument as `None` in which case only the walker coordinated would be exported or we could set it to `rw.im_big` to export the domain encompassing all the walks. Caution should be exercised when using this function as larger domains produce very large files. The second argument `sample` tells the function to down-sample the coordinate data by this factor. We have already set a stride to only record every 100 steps which is useful for speeding up calculating the MSD and so the sample is left as the default of 1. The export function also accepts a `path`, `sub` and `prefix` argument which lets you specify where to save the data and what to name the subfolder at this path location and also a prefix for the filenames to be saved. By default the current working directory is used as the path, `data` is used for the subdirectory and `rw_` is used as a prefix. After running the function, which takes a few seconds to complete, inspect your current working directory which should contain the exported data. There should be 101 files in the data folder: A small file containing the coordinates of each walker at each recorded time step with extention `.vtu` and larger file named

`rw_image.vti`. To load and view these files in Paraview take the following steps:

1. Open the `rw_image.vti` file and press Apply
2. Change Representation to Surface and under Coloring change the variable from `Solid Color` to `image_data`
3. Apply a Threshold filter to this object (this may take a little while to process) and set the Maximum of the threshold to be 0 (again processing of this step takes some time)
4. Now you can rotate the image and inspect only the solid portions in 3D
5. Open the `.vtu` as a group and click Apply and a bunch of white dots should appear on the screen displaying the walker starting locations.
6. Pressing the green play button will now animate the walkers.
7. If you desire to see the paths taken by each walker saved on the screen then select the coords object and open the filters menu then select `TemporalParticlesToPathlines`. Change the `Mask Points` property to 1, `Max Track Length` to exceed the total number of steps and `Max Step Distance` to exceed the stride.
8. To produce animations adjust settings in the Animations view.

The following images can be produced:



CHAPTER 4

Function Reference

- genindex
- modindex
- search