BIAS2016 **MODULE 5**

AUTHOR:
Simon F. Nørrelykke
ScopeM, ETH Zürich

Introduction to Matlab

BIAS2016 BioImage Data Analysis Course
EMBL Heidelberg
5–11 June 2016

REVIEWERS/TEACHERS:
Perrine Paul-Gilloteaux
CNRS University of Nantes

Compiled on: June 7, 2016

Contents

5.1 Aim

You will learn to use MATLAB to analyse data extracted from the preceeding tracking module (as you have no doubt noticed, the strength of ImageJ does not lie in data-analysis). You will also be introduced to some of the powerful and flexible image-analysis methods native to MATLAB. If this is the first time you code, except from writing Macros in ImageJ, then this will also serve as a crash course in programming for you.

5.2 Tools

We shall be using the commercial software package MATLAB as well as some of its problem specific toolboxes, of which there are currently more than 30.

5.2.1 MATLAB

Don't panic! MATLAB is easy to learn and easy to use. But you do still have to learn it. MATLAB is short for *matrix laboratory*, hinting at why MATLAB is so popular in the imaging community—remember that an image is just a matrix of numbers. MATLAB is commercial software for numerical, as opposed to symbolic, computing.

5.2.2 Image Processing Toolbox

Absolutely required if you want to use MATLAB for image analysis.

5.2.3 Statistics and Machine Learning Toolbox

Somewhat necessary for data-analysis, though we can get quite far with the core functionalities alone.

5.3 Getting Started with MATLAB

That is what we are doing here! However, if you have to leave now and still want an interactive first experience: Head over here for a free, two hour, interactive tutorial that runs in your web-browser and does not require a MATLAB license (they also have paid in-depth courses).

5.3.1 Baby steps

Start MATLAB and lets get going! When first starting, you should see something similar to Fig. 5.1

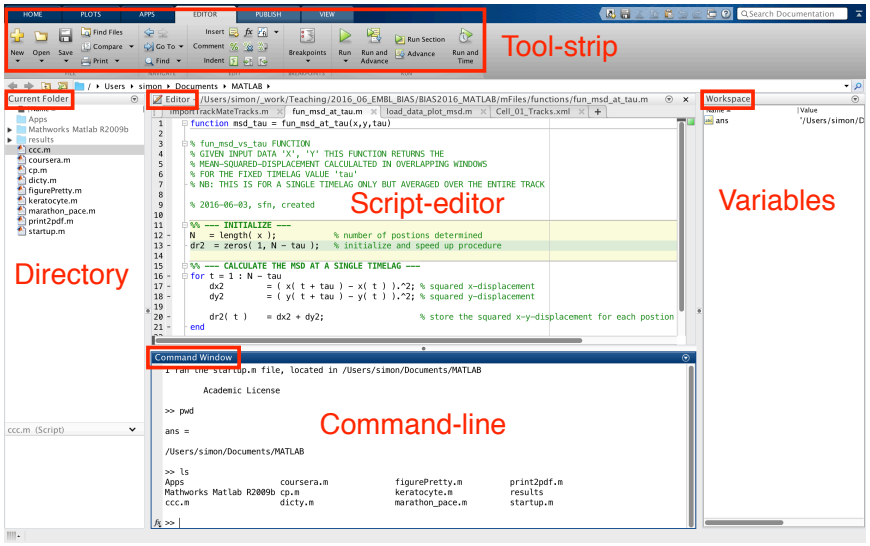

Figure 5.1: The full MATLAB window with default layout of the windows. Some preset layouts are accessible in the tool-strip, under the `HOME` tab, in the `Layout` pull-down menu. Double-click on the top-bar of any sub-window to maximize it, double-click again to revert.

First we are just going to get familiar with the command line interface. To reduce clutter, double-click on the bar (grey or blue) saying `Command Window`. This will, reversibly, maximize that window.

Now, let us add two numbers by typing `5+7`, followed by return. The result should look like in Fig. 5.2

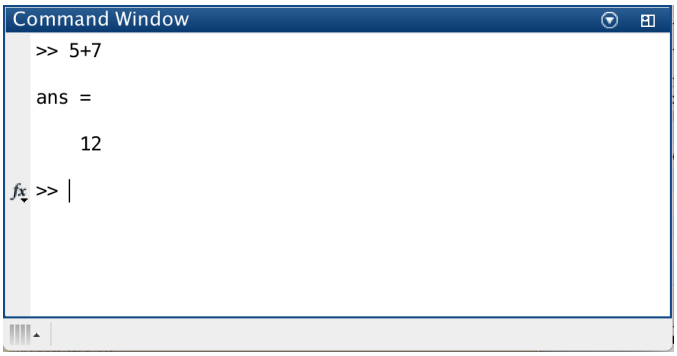

Figure 5.2: The command window in MATLAB after entering `5+7` and hitting the return key. The result, 12, is displayed and stored in the variable `ans`.

Next, let us define two variables `a` and `b` and add them to define a third variable `c`

```
>> a=5

a =

     5

>> b=7

b =

     7

>> c=a+b

c =

    12
```

This time, we notice that the result of our operation are no longer stored in the variable `ans` but in the variable with the name we gave it, i.e., `a`, `b`, and `c`.

Finally, let us change one of the variables and see how the other two change in response to this.

```
>> a=10

a =

    10

>> c

c =

    12

>> c=a+b

c =

    17
```

Here, you should notice that the value of `c` does not change until we have evaluated it again — computers are fast, but they cannot not read our minds (most of the time), so we have to tell them *exactly* what we want them to do.

Ok, that might have been somewhat underwhelming. Let us move on to something slightly more interesting and that you can probably not so easily do on your phone.

5.3.2 Plot something

Here are the steps we will take:

1. Create a vector x of numbers
2. Create a function y of those numbers, e.g. the cosine or similar
3. Plot y against x
4. Label the axes and give the plot a title

5. Save the figure as a pdf file

First we define the peak-amplitude (half of the peak-to-peak amplitude)

```
>> A = 10

A =

    10
```

Then we define a number of discrete time-points

```
>> x = 0 : 0.1 : 5*pi;
```

Notice how the input we gave first, the `A`, was again confirmed by printing (echoing) the variable name and its value to the screen. To suppress this, simply end the input with a semicolon, like we just did when defining x. The variable x is a vector of numbers, or time-points, between 0 and 5π in steps of 0.01. Next, we calculate a function $y(x)$ at each value of x

```
>> y = A * cos( x );
```

Finally, we plot y versus x

```
>> figure;plot(x,y)
```

To make the figure a bit more interesting we now add one more plot as wells as legend, labels, and a title. The result is shown in Fig. 5.3.

```
>> y2 = y .* x;
>> hold on
>> plot( x, y2, '--r')
>> legend('cos(x)', 'cos(x)/x')
>> xlabel('Time (AU)')
>> ylabel('Position (AU)')
>> title('Plots of various sinusoidal functions')
```

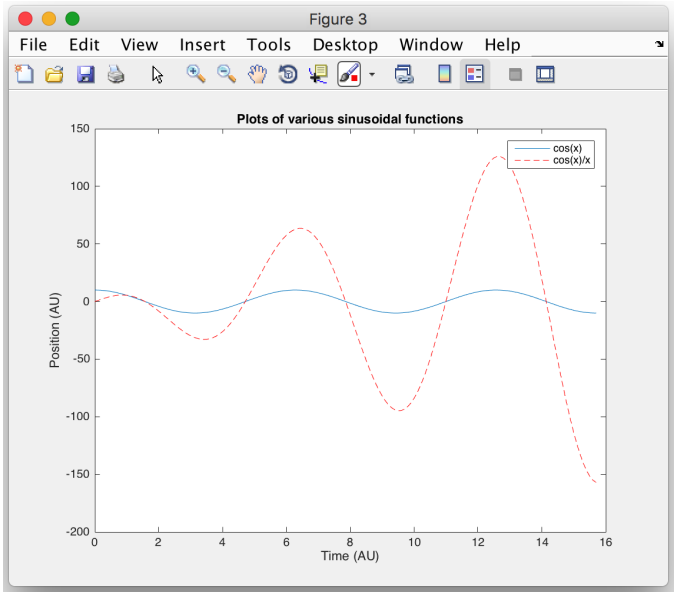

Figure 5.3: Two sinusoidal plots with legend, axes labels, and title

Here, `hold on` ensures that the plots already in the figure are "held", i.e., not erased, when the next function is plotted in the same figure window. We specify that the use of a dashed red line, for the new plot, by the `'--r'` argument in the `plot` function. You will also have noticed that we multiplied using .∗ and not just ∗ — this is known as element-wise multiplication, as opposed to matrix or vector multiplication (more on that in a little while).

After having created a figure and adjusted it to your liking, you may want to export it for use in a paper or presentation. This can be done either via the pull-down menus, if you only need to do it once, or via the command line if it is a recurrent job:

```
>> print( '-dpdf', '~/Desktop/cosineFigure.pdf')
```

The `print` function is not confined to the pdf format but can also export to png, tiff, jpeg, etc.

5.3.3 Make it pretty

We have a large degree of control over how things are rendered in MATLAB. It is possible to set the typeface, font, colors, line-thickness, plot symbols, etc. Don't overdo this! The main objective is to communicate your message, and that message is rarely "look how many colors I have" — if you only have two graphs in the same figure, gray-scale will likely suffice. Strive for clarity!

5.3.4 Getting help

At this point you might want to know how to get help for a specific command. That is easy, simply type `help` and then the name of the command you need help on. Example, for the `xlabel` command we just used:

```
>> help xlabel
 xlabel X-axis label.
    xlabel('text') adds text beside the X-axis on the
       current axis.

    xlabel('text','Property1',PropertyValue1,'Property2',
       PropertyValue2,...)
    sets the values of the specified properties of the
       xlabel.

    xlabel(AX,...) adds the xlabel to the specified axes.

    H = xlabel(...) returns the handle to the text object
       used as the label.

    See also ylabel, zlabel, title, text.

    Reference page for xlabel
```

If you click the link on the last line it will open a separate window with more information and graphical illustrations. Alternatively, simply go directly to that page this way

```
1 >> doc xlabel
```

Expect to spend substantial time reading once you start using more of the options available. MATLAB is a rich language and most functions have many properties that you can tune to your needs, when these differ from the default.

5.4 Working with images

Because MATLAB was designed to work with matrices of numbers it is particularly well-suited to operate on images. Recently, the Mathworks have also made efforts to become more user-friendly. Let's demonstrate:

1. Save and image to your desktop, e.g. "Blobs.tif" from ImageJ
2. Open the MATLAB app `Image Viewer` either from the tool-strip or by typing `imtool`
3. From the `Image Viewer` go to `File > Open ...` and select an image
4. Adjust the contrast, inspect the pixels, measure a distance, etc, using the tool-strip shortcuts

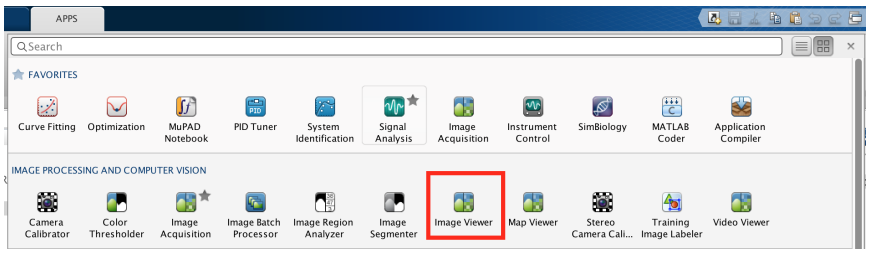

Figure 5.4: Access to various apps in the tool-strip of MATLAB. The apps accessible will depend on the tool-boxes you have installed.

5.4.1 Reading and displaying an image

This, however, is not much different from what we can do in ImageJ. The real difference comes when we start working from the command-line and

making scripts — while this is also possible in ImageJ, it is a lot easier in MATLAB. Assuming you have an image named "blobs.tif" on your desktop, try this

```
>> cd ~/Desktop
>> myBlobs = imread( 'blobs.tif' );
>> figure(1); clf
>> imshow( myBlobs )
>> figure(2); clf
>> imshow( myBlobs , 'displayrange', [10 200], ...
'initialmagnification', 'fit' )
```

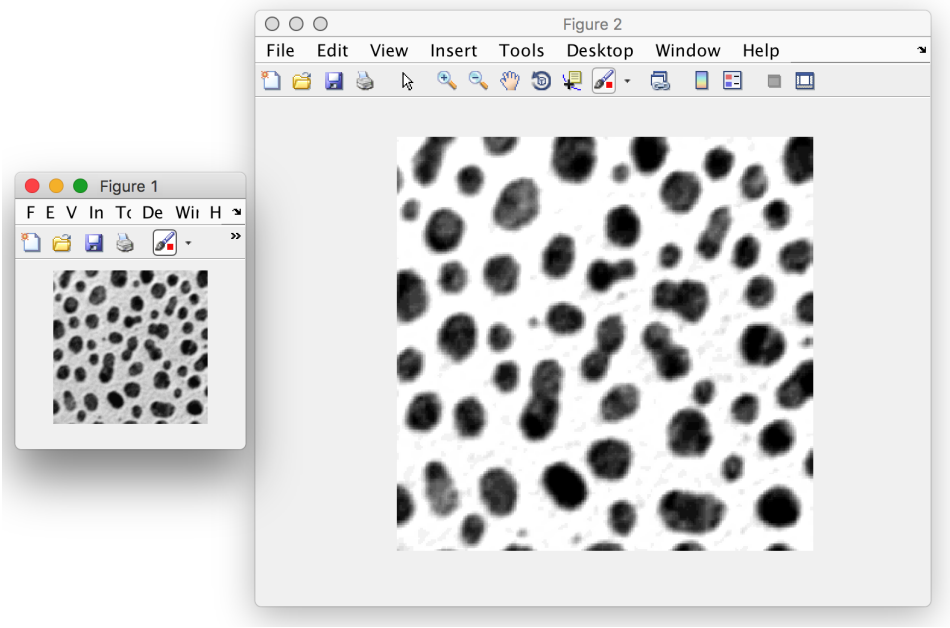

Figure 5.5: The "blobs" from ImageJ displayed without (left) and with (right) scaling of intensity and size.

Here is what we just did: 1) We navigated to the directory holding our image; 2) Read the image into the variable `myBlobs` using the `imread` command; 3) Selected figure number 1 (or created it if it didn't exist yet) and cleared it; 4) Displayed the content of our variable `myBlobs` in figure 1; 5) Selected, or created, figure number 2 and cleared it; 6) Again displayed the content of `myBlobs` but now with the displayed gray-scale confined (especially relevant for 16bit images that otherwise appear black), and the

displayed image fitted to the size of the window.

5.4.2 Extracting meta-data from an image

Because we are becoming serious image-analysts we also take a look at the meta-data that came with the image.

```
>> blobInfo = imfinfo('blobs.tif');
>> whos blobInfo
  Name          Size            Bytes  Class     Attributes

  blobInfo      1x1              5908  struct

>> blobInfo

blobInfo =

                     Filename: '/Users/simon/Desktop/blobs.
                        tif'
                  FileModDate: '05-Jun-2016 09:45:04'
                     FileSize: 65172
                       Format: 'tif'
                FormatVersion: []
                        Width: 256
                       Height: 254
                     BitDepth: 8
                    ColorType: 'grayscale'
              FormatSignature: [77 77 0 42]
                    ByteOrder: 'big-endian'
               NewSubFileType: 0
                BitsPerSample: 8
                  Compression: 'Uncompressed'
    PhotometricInterpretation: 'WhiteIsZero'
                 StripOffsets: 148
              SamplesPerPixel: 1
                 RowsPerStrip: 254
              StripByteCounts: 65024
                  XResolution: []
                  YResolution: []
               ResolutionUnit: 'Inch'
```

```
                    Colormap: []
          PlanarConfiguration: 'Chunky'
                   TileWidth: []
                  TileLength: []
                 TileOffsets: []
               TileByteCounts: []
                  Orientation: 1
                    FillOrder: 1
              GrayResponseUnit: 0.0100
                MaxSampleValue: 255
                MinSampleValue: 0
                  Thresholding: 1
                        Offset: 8
             ImageDescription: 'ImageJ=1.50b...'
```

After your experience with ImageJ you should have no problems understanding this information. What is new here, is that the variable `blobInfo` that we just created is of the type `struct`. Elements in such variables can be addressed by name, like this:

```
>> blobInfo.Offset

ans =

     8

>> blobInfo.Filename

ans =

/Users/simon/Desktop/blobs.tif
```

When addressing an element by name, you can reduce typing by hitting the TAB-key after entering `blobInfo.` — this will display all the field-names in the structure.

It is important to realize that `imread` will behave different for different image formats. For example, the tiff format used here supports the reading

of specific images from a stack via the `'index'` input argument (illustrated below) and extraction of pixel regions via the `'pixelregion'` input argument. The latter is very useful when images are large or many as it can speed up processing not having to read the entire image image into memory. On the other hand, jpeg2000 supports `'pixelregion'` and `'reductionlevel'`, but not `'index'`.

5.4.3 Reading and displaying an image-stack

Taking one step up in complexity we will now work with a stack of tiff-files instead. These are the steps we will go through

1. Open "MRI Stack (528K)" in ImageJ (File > Open Samples)
2. Save the stack to your desktop, or some other place where you can find it (File > Save)
3. Load a single image from the stack into a two-dimensional variable
4. Load a multiple images from the stack into a three-dimensional variable
5. Browse through the stack using the `implay` command
6. Create a montage of all the images using the `montage` command

After performing the first two steps in ImageJ, we switch to MATLAB:

```
1 >> mriImage = imread( 'mri-stack.tif', 'index', 7 );
2 >> imshow(mriImage)
3 >> mriStack( : , : , 7 ) = imread( 'mri-stack.tif', 'index'
    , 7 );
4 >> for imageNumber = 1 : 27
5 mriStack( : , : , imageNumber ) = imread( 'mri-stack.tif',
    'index', imageNumber );
6 end
7 >> whos
8   Name               Size                    Bytes  Class
          Attributes
9
```

```
  imageNumber        1x1                          8  double
  mriImage         226x186                    42036  uint8
  mriStack         226x186x27               1134972  uint8
>> implay(mriStack)
```

Here, we again used the `imread` command, but now with the extra argument `'index'` to specify which single image to read — here we chose image number 7 and displayed the result, again with `imshow`. Next, we load the entire mri-stack one image at a time. This is done by writing into the three-dimensional array (data-cube) `mriStack` using a `for`-loop (this concept should already be familiar to you from the ImageJ macro sections). We use the colon-notation to let MATLAB know that it should assign as many rows and columns as necessary to fit the images. The `implay` command is very much like in ImageJ.

Finally, we want to create a montage. This requires one additional step because we are working on single-channel data as opposed to RGB images:

```
>> mriStack2 = reshape( mriStack, [226 186 1 27]);
>> whos mriStack*
  Name             Size                    Bytes  Class
     Attributes

  mriStack       226x186x27              1134972  uint8
  mriStack2        4-D                   1134972  uint8
> size(mriStack2)

ans =

   226   186     1    27
>> map = colormap('bone');
>> montage(mriStack2, map, 'size', [3 9])
```

The `reshape` command is used to, well, reshape data arrays and here we used it to simply add one more (empty) dimension so that `montage` will read the data. The result is shown in Fig. 5.6.

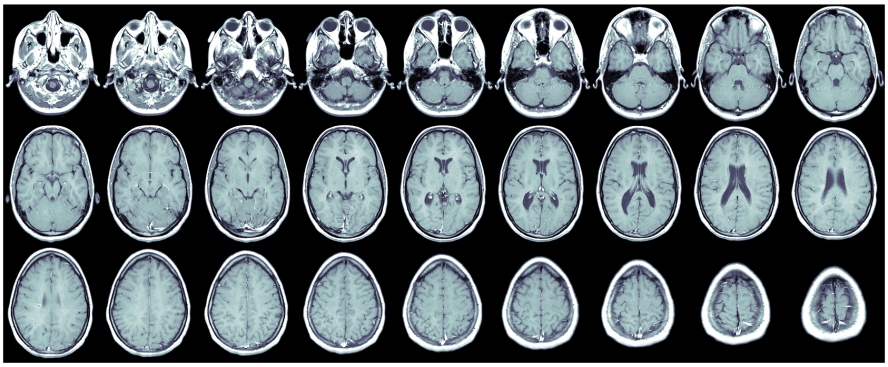

Figure 5.6: A montage of the 27 images in the MRI stack, arranged as 3×9 and displayed with the colormap "bone".

5.4.4 Smoothing, thresholding and all that

Yes, of course we can perform all these operations and here is a small taste of how it is done. We are going to

1. Load an image and invert it
2. Create a copy of it that has been smoothed with a Gaussian kernel
3. Determine the Otsu threshold for this copy
4. Create a binary image based on the smoothed copy
5. Display the mask on the original
6. Apply this mask (binary image) to the original and make measurements

```
>> blobs = imread('blobs.tif');
>> blobs_inv = 255 - blobs;
>> figure(1); imshow( blobs_inv, 'initialmagnification', '
   fit')
>> blobs_inv_gauss = imgaussfilt( blobs_inv, 2 );
>> OtsuLevel = graythresh( blobs_inv_gauss )

OtsuLevel =

    0.4627
```

```
10
11 >> blobs_bw = im2bw( blobs_inv_gauss, OtsuLevel );
12 >> figure(2); imshow( blobs_bw, 'initialmagnification', '
   fit')
13 >> blobs_bw_uint8 = uint8( blobs_bw );
14 >> blobs_masked = blobs_inv .* blobs_bw_uint8;
15 >> figure(3); imshow( blobs_masked, 'initialmagnification',
    'fit')
```

As an alternative to showing the masked image we can choose to show the outlines of the connected components (the detected blobs):

```
1 >> blobs_perimeter = bwperim( blobs_bw );
2 >> blobs_summed = blobs_inv  + uint8(blobs_perimeter) *
   255;
3 >> figure(4); imshow( blobs_summed , 'initialmagnification'
   , 'fit' )
```

In step two we convert the `logical` variable `blobs_perimeter` to an 8-bit unsigned integer on the fly, before adding it to the image. If you wonder why we do this conversion, just try to omit it and read the error-message from MATLAB.

Now, let's make some measurements on the b/w image and display them on the `blobs_summed` image from above:

```
1 >> stats = regionprops( blobs_bw, {'area' 'perimeter' '
   centroid'} );
2 >> centroids = cat(1, stats.Centroid );
3 >> figure(4); hold on; plot(centroids(:,1) , centroids(:,2)
   , '*r')
```

The result of this step is shown in Fig. 5.7.

Finally, we measure the gray-scale image using the masks — this should remind you of the "Redirect to:" option in ImageJ (Analyze > Set Measurements ...):

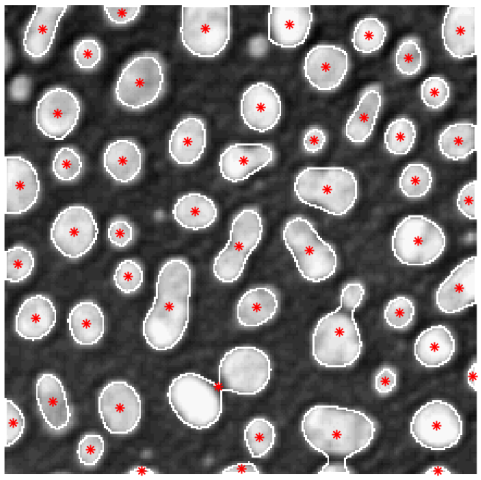

Figure 5.7: "Blobs" shown with outlines of threshold-based segmentation overlaid. The centroids of each connected component is marked with a red asterisk. Spot the problem. What would you do to fix it?

```
1 >> labels = bwlabel( blobs_bw );
2 >> statsGrayscale = regionprops( labels, blobs_inv, '
     meanintensity');
3 >> meanIntensity = cat( 1, statsGrayscale.MeanIntensity );
4 >> figure(3); text( centroids(:,1) - 10, centroids(:,2),
     num2str( meanIntensity, 4 ), 'color', 'blue', 'fontsize'
     , 14)
```

Exercise: Do this and understand each step! The result is shown in Fig. 5.8.

5.5 Automating it

The command-line is a wonderful place to quickly try out new ideas — just type it in and hit return. Once these ideas become longer we need to somehow record them in one place so that we can repeat them later on without having to type everything again. You know what we are getting

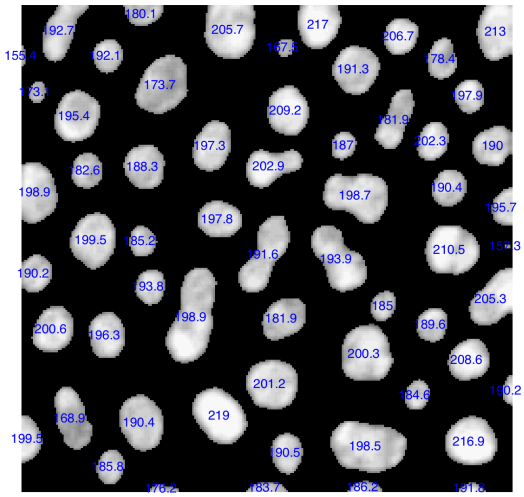

Figure 5.8: Masked version of "blobs" with the measured mean intensity for each connected component shown.

to: The creation of computer programs.

In the simplest of cases we can take a series of commands, that were executed in the command line, and save them to a file. We could then, at a later stage, open that file and copy these lines into the command line, one after the other, and press return. This is actually a pretty accurate description of what takes place when MATLAB runs a script: It goes through each line of the script and tries to execute it, one after the other, starting at the top of the file.

5.5.1 Create, save, and run scripts

You can use any editor you want for writing down your collection of MATLAB statements. For ease of use, proximity, uniformity, and because it comes with many powerful extra features, we shall use the editor that comes with MATLAB. It will look something like in Fig. 5.9 for a properly typeset and documented program. You will recognize most of the commands from when we plotted the sinusoidsal functions earlier. But now we have also added some text to explain what we are doing.

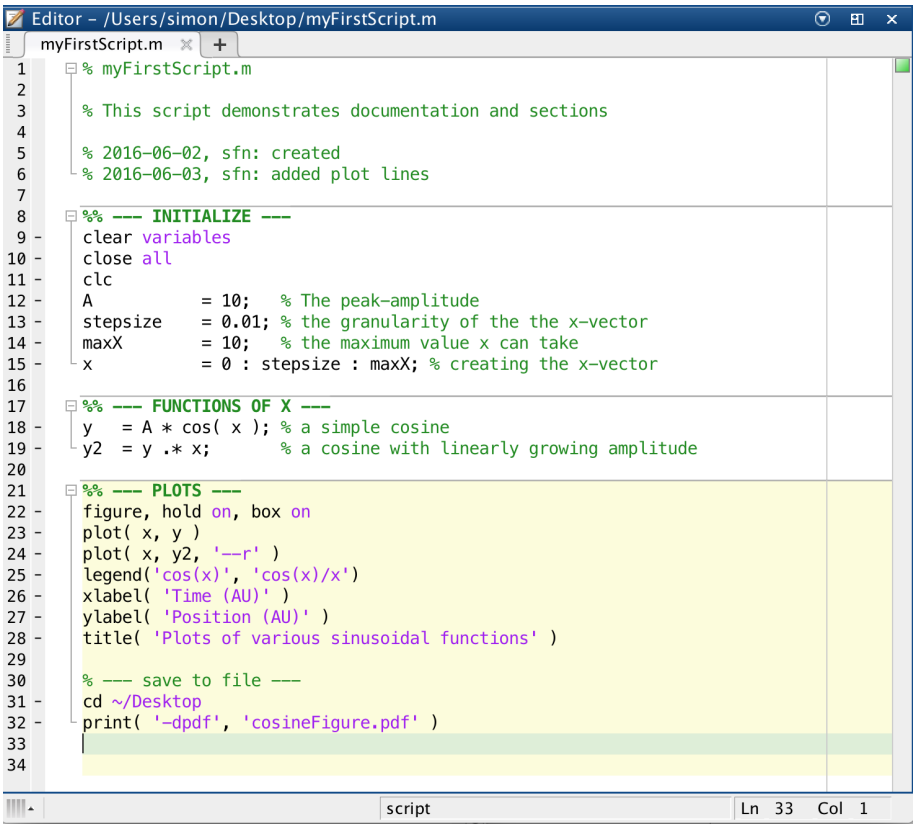

Figure 5.9: The editor window. The script is structured for easy human interpretation with clear blocks of code and sufficient documentation. Starting from the percent sign all text to the right of it is "outcommented" and appears green, i.e., MATLAB does not try to execute it. A double percent-sign followed by space indicates the beginning of a code-block that can be folded (command-.), un-folded (shift-command-.) and executed (command-return) independently. The currently active code-block is yellow. The line with the cursor in it is pale-green. Notice the little green square in the upper right corner, indicating that MATLAB is happy with the script and has not warnings or suggestions.

A script like the one in Fig. 5.9 can be run in several ways: 1) You can click on the big green triangle called "run" in Editor tab; 2) Your can hit *F*5 when your cursor is in the editor window; or 3) You can call the script by name from the command line, in this case simply type `myFirstScript` and hit return. The first two options will first save any changes to your script, then execute it. The third option will execute the version that is saved to disk when you call it. If a script has unsaved changes an asterisk appears next to its name in the tab.

When you save a script, please give it a meaningful name — "script5.m" is not a good name even if you intend to never use it again (if it is tem-

porary call it "scratch5.m" or "deleteMe5.m"). Make it descriptive and use underscores or camel-back notation as in "my_first_script.m" or "myFirstScript.m". The same goes for variable names.

5.5.2 Code folding and block-wise execution

As you will have noticed, in the screenshot of the editor, the lines of codes are split into paragraphs separated by lines that start with two percent signs and a blank space. All the code between two such lines is called a code-block. These code-blocks can be folded by clicking on the little square with a minus in it on the left (or use the keyboard shortcut `command-.`, to unfold do `shift-command-.`). This is very useful when your code grows.

You can quickly navigate between code-blocks with `command-arrow-up/down` and once your cursor is in a code-block you are interested in you can execute that entire block with `command-return`. Alternatively, you can select (double-click or click-drag) code and execute it with `F7`. For all of these actions you will see the code appearing and attempting to execute in the command window.

A list of keyboard shortcuts as well as settings for code-folding can be found in the preference settings, via the `command-,` shortcut, as always, on a mac. What is it on a PC?

5.5.3 Scipts, programs, functions — nomenclature

Is it a script or a program? It depends! Traditionally only compiled languages like C, C++, Fortran, and Java are referred to as programming languages. Whereas languages such as JavaScript and Perl, that are not compiled, were called scripting languages. Then there is Python, sitting somewhere in between. MATLAB also is in between, here it what the MathWorks have to say about it

> Program files can be scripts that simply execute a series of MATLAB statements, or they can be functions that also accept input arguments and produce output. Both scripts and functions contain MATLAB code, and both are stored in text files with a .m

extension. However, functions are more flexible and more easily extensible.

Ok, so when we save our creations to an `m`-file (a file with extension .m) we call it a program file. But the thing we saved was either a script or a function, or perhaps a new class definition. We shall use the word "program" to refer to both scripts and functions, basically whatever we have in the editor, but may occasionally specify which of the two we have in mind if it makes things clearer.

5.5.4 Read from a folder

To get a list of files in a folder you have several options: 1) Navigate MATLAB to the folder (by clicking or using the `cd` command) and type `ls` or `dir`; 2) Give the `ls` or `dir` command followed by the path to the folder, like this

```
>> ls -lha ~/Desktop/
total 2376
drwx------@   6 simon  staff   204B Jun  6 18:30 .
drwxr-xr-x@ 109 simon  staff   3.6K Jun  4 21:18 ..
-rw-r--r--@   1 simon  staff   6.0K May 30 10:55 .DS_Store
-rw-r--r--    1 simon  staff   1.9K Apr 12 21:54 .Rhistory
-rw-r--r--    1 simon  staff    64K Jun  5 11:45 blobs.tif
-rw-r--r--    1 simon  staff   1.1M Jun  5 13:14 mri-stack.
   tif

>> dir ~/Desktop/

.              .DS_Store      blobs.tif
..             .Rhistory      mri-stack.tif
```

We can also assign the output to variables:

```
>> lsList = ls('~/Desktop/');
>> dirList = dir('~/Desktop/');
```

What is the difference between the two variables `dirList` and `lsList`?

5.5.5 Path and file names

To illustrate how to work with and combine file-names and path-name we will introduce the dialogue window (again assuming we are in the `~/Desktop/` directory and have a file called "blobs.tif" there):

```
>> fileName = uigetfile('.tif')
```

In response to which we should see a dialogue window similar to Fig. 5.10.

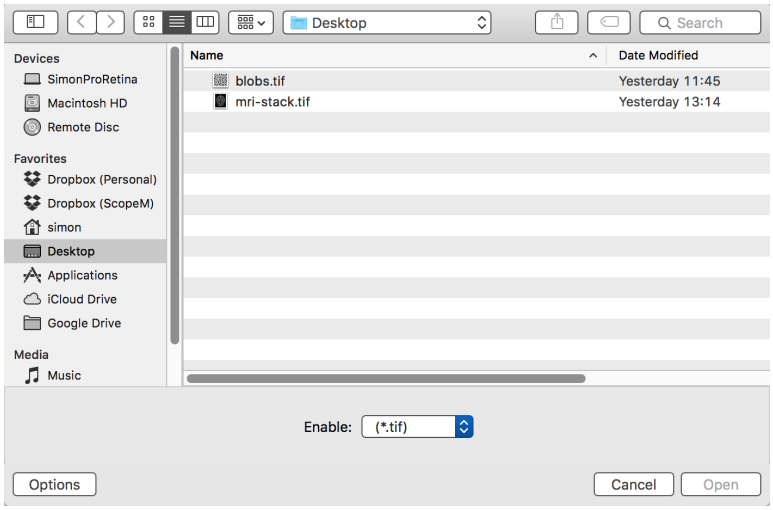

Figure 5.10: The dialogue window, in OS X, that appears in response to the `uigetfile` command.

We should also be told the name of the file we selected:

```
fileName =

blobs.tif
```

If we want more information, such as the location of the file we do:

```
>> [fileName, pathName] = uigetfile('.tif')

fileName =
```

```
4
5 blobs.tif
6
7
8 pathName =
9
10 /Users/simon/Desktop/
```

From the file- and path-name we can now create the full file-name, incl. the path, using the command `fullfile`:

```
1 >> fullFileName = fullfile(pathName, fileName)
2
3 fullFileName =
4
5 /Users/simon/Desktop/blobs.tif
```

Obviously, if you are working on a different system you file-separator might look different. However, that is because `fullfile` inserts platform-dependent file separators. If you want more control over this aspect you should look into the `filesep` command.

Reversely, if you had the full name of a file and wanted to extract the file-name or the path-name, you could do this:

```
1 >> [pathstr,name,ext] = fileparts(fullFileName)
2
3 pathstr =
4
5 /Users/simon/Desktop
6
7
8 name =
9
10 blobs
11
12
13 ext =
14
```

```
15 .tif
```

Alternatively, if all we wanted was the name of a directory we would use the command `uigetdir` — you can guess what it does.

Why did we just do all this? We did it because we often have to spend a lot of time on data-wrangling before we can even get to the actual data-analysis. Knowing how to easily extract file and path names for your data allows you automate many later steps. Example: You might want to open each image in a directory, crop it, scale it, smooth it, then save the results to another directory with each modified image given the same name as the original but with "_modified" appended to the name.

5.6 Time-series analysis

MATLAB has a dedicated data type called simply `timeseries`. We shall not be using this class here as it is too specialized for what we want to do. At a later stage in your research you might find it useful, but be warned that is was developed probably more with the financial sector in mind and may not support quite the kind of analysis you need to perform.

Whether or not you actually have a time-series or simply an ordered list of data often does not matter. Many of the tools are the same but were indeed developed by people doing signal-processing for, e.g., telephone companies, i.e., they worked on actual time-series data.

NB: As the operations are getting more complex, We will be working almost exclusively in the editor from now on.

5.6.1 Simulating a time-series of Brownian motion (random walk)

Physical example: Diffusing molecule or bead. A particle undergoing Brownian motion is essentially performing a random walk: In one dimension, each step is equally likely to be to the right or left. If, in addition, we make the size of the step follow a Gaussian distribution, we essentially

have Brownian motion in 1D, also known as diffusion.

The code for generating the random numbers goes something like this:

```
%% --- INITIALIZE ---
dt          = 1; % time between recordings
t           = 0 : dt : 1000 * dt; % time

%% --- GENERATE RANDOM STEPS ---
stepNumber  = numel( t ); % number of steps to take
xSteps      = randn( 1, stepNumber ) * sqrt(dt); % Gaussian
    distributed steps of zero mean
```

After that we calculate the positions of the particle and the experimentally determined speeds (we will return to these in detail below):

```
%% --- CALCULATE POSITIONS AND SPEEDS ---
xPos        = cumsum( xSteps ); % positions of particle
varSteps    = var( xSteps );    % variance of step-
   distribution

xSpeed      = abs( diff( xPos ) ) / dt; % "speed"
meanSpeed   = mean( xSpeed );
stdSpeed    = std( xSpeed );


%% --- DISPLAY NUMBERS ---
disp( ['Average speed = ' num2str( meanSpeed ) ] )
disp( ['STD speed = ' num2str( stdSpeed ) ] )
disp( ['VAR steps = ' num2str( varSteps ) ] )
```

In the last two lines we used the command `disp` that displays its argument in the command window. It takes as argument variables of many different formats, incl. numerical and strings. Here, we gave it a string variable that was concatenated from two parts, using the { and } operators. The first part is an ordinary string of text in single quotes, the second part is also a string but created from a numeric variable using the command `num2str`.

The other MATLAB commands `cumsum`, `diff`, `mean`, and `std` do what they

say and calculate the cumulative sum, the difference, the mean, and the standard deviation, respectively. Look up their documentation for details and additional input arguments.

5.6.2 Plotting a time-series

Ok, now let us plot some of these results:

```
%% --- PLOT STEPS VERSUS TIME ---
figure; hold on; clf
fs  = 16; % font-size for labels
plot( t, xSteps, '-k' )
xlabel( 'Time [AU]', 'fontsize', fs )
ylabel( 'Step [AU]', 'fontsize', fs  )
title( 'Steps versus time', 'fontsize', fs  )
set( gca, 'fontsize', fs ) % size of numbers on the axes
```

The output of these lines, and a similar pair for the positions, is shown in Fig. 5.11.

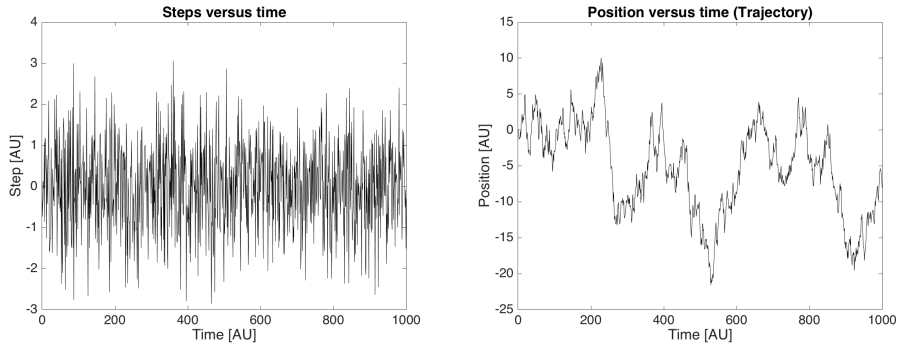

Figure 5.11: Steps (left) an positions (right) as a function of time for a one-dimensional random walk.

5.6.3 Histograms

Let us now examine the distribution of step-sizes. We do that by plotting a histogram:

```
%% --- PLOT HISTOGRAM OF STEPS ---
figure; hold on; clf
binNumber = floor( sqrt( stepNumber ) );
histogram( xSteps, binNumber )
xlabel( 'Steps [AU]', 'fontsize', fs  )
ylabel( 'Count', 'fontsize', fs  )
title( 'Histogram of step-sizes', 'fontsize', fs  )
set( gca, 'xlim', [-4 4] )
set( gca, 'fontsize', fs ) % size of numbers on the axes
```

Figure 5.12 show the resulting plot. The command `histogram` was introduced in MATLAB R2014b and replaces the previous command `hist` — they are largely similar but the new command makes it easier to create pretty figures and uses the color-scheme introduced in MATLAB R2014b: Since version R2014b, MATLAB's new default colormap is called "parula" and replaces the previous default of "jet".

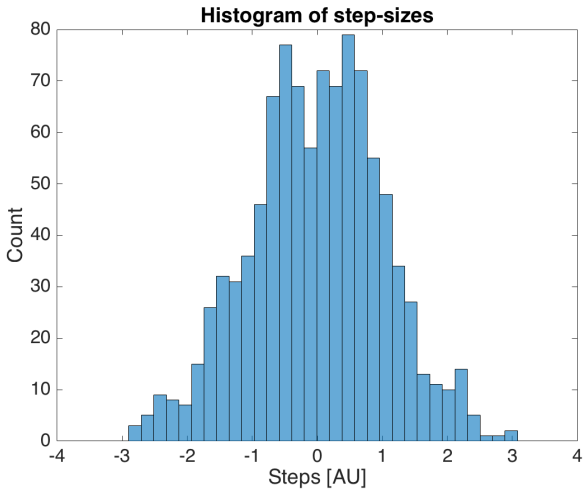

Figure 5.12: Histogram of step sizes for a random walk. The steps were generated with the command `randn` that creates pseudo-random numbers from a Gaussian distribution.

5.6.4 Codehygiene

It is important for your future self, not to mention collaborators, that you keep good practices when coding.

- The actual code should be easy to read, not necessarily as compact as possible
- Use descriptive names
- Document the code
- Insert plenty of blank spaces: Let your code breathe!

Figure 5.13 is and example of how your code could look, when folded, if you take care to structure it nicely — notice how easy it to figure out what goes on where, without having to read a single line of actual code. Also, the currently active code-block is highlighted in yellow.

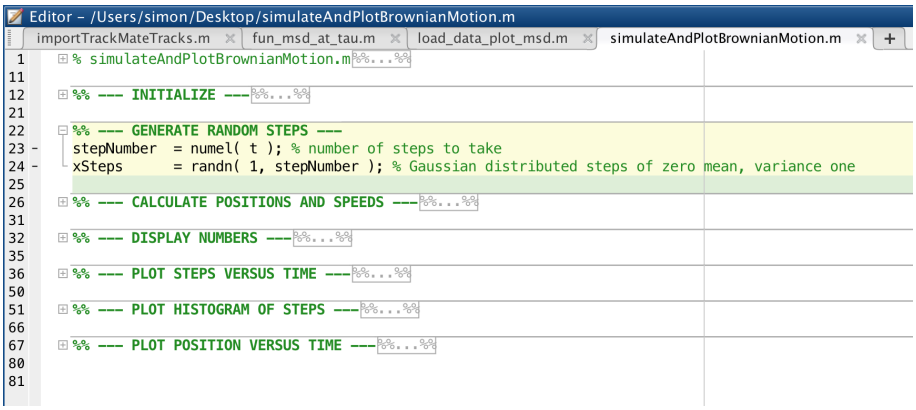

Figure 5.13: Screenshot of code that is clearly structured and folded

5.6.5 Smoothing (filtering) a time-series

If you suspect that some of the jitter in your signal is simply noise, you can smooth the signal. This is very much the same procedure as when smoothing an image. The relevant command is `smooth` and it has several options for adaptation to your needs:

```
% --- simple smoothing ---
xPosSmoothed = smooth( xPos ); % defaults to moving average
    over 5 data points

% --- sophisticated smoothing ---
span   = 7;          % number of data points to average over
method = 'sgolay' ; % Savitsky-Golay filter
degree = 3;          % the order of the s-g filter

xPosSmoothed = smooth( xPos, span, method, degree );
```

5.6.6 Sub-sampling a time-series (slicing and accessing data)

Sometimes we can get useful information about our time-series by sub-sampling it. An example could be a signal x, that is corrupted by nearest-neighbor correlations: To remove this, simply remove every second data-point, like this:

```
xSubsampled = x( 1 : 2 : end );
```

Or, if you wanted only every third data-point from the first 200 entries:

```
xSubsampled = x( 1 : 3 : 200 );
```

What we just illustrated, was how to read only selected entries from a vector; in the first example we read every second entry from the beginning (the first element in a vector in MATLAB has index 1, not 0), in steps of 2, until the end. The same idea holds for arrays of arbitrary dimension in MATLAB; each dimension is treated independently.

If we wanted, we could also have given a list of indices to read, like this:

```
readThese   = [2 5 7 88 212]; % data-points to read
xSubsampled = x( readThese );
```

Alternatively, if we only wanted to replace an element, say in entry 7, with the number 3. Or find all entries larger than 0.94, then set them to 1.

```
% --- replace single element ---
x( 7 )  = 3;

% --- replace several elements ---
xIndex      = find( x > 0.94 );
x( xIndex ) = 1;
```

The `find` command is very useful for data-wrangling and thresholding. Combined with the query command `isNaN` (asking if something "is not-a-number") you will certainly find yourself applying it once working with real-world data.

5.6.7 Investigating how "speed" depends on subsampling or *dt*

After having carefully examined the steps and trajectories we may get the idea of also looking into the velocities and their sizes (speeds). Velocities can be calculated from positions by differentiation wrt. time. Since we have a discrete time-series, we do that by forming the difference and dividing by the time-interval dt — this is what we did above with the help of the `diff` command.

And this is where it gets interesting: When we vary dt, our estimate of the speed also changes! Does this make sense? Take a minute to think about it: What we are finding is that, depending on how often we determine the position of a diffusive particle, the estimated speed varies. Would you expect the same behavior for a car or a plane? Ok, if this has you a little confused you actually used to be in good company, that is, until Einstein explained what is really going on, back in 1905 — you might know the story.

The take-home message is that speed is ill-defined as a measure for Brownian motion. If you are wondering what we can use instead, read on, the next section, on the mean-squared-displacement, has you covered.

5.6.8 Simulating confined Brownian motion

Brownian motion doesn't have to be free. The observed particle could be trapped in a small volume or elastically tethered to a fixed point. To be specific, let us choose as physical example a sub-micron sized bead in an optical trap, in water. This turns out to be just as easy to simulate as pure Brownian motion. Writing down the equations of motion and solving them (or using intuition) we see that the observed positions are simply given by random numbers from a Gaussian distribution. The width of the distribution is determined by the strength of the trap (size of the confinement, stiffness of tether). Importantly, we are not sampling the position of this bead very often, only every millisecond or so, rarely enough that it has time to "relax" in the trap between each determination.

```
1 sampleNumber = 1000; % number of postion determinations
2 xTrapped     = randn( 1, sampleNumber ); % position of bead
      in trap
```

What do we get if we repeat the above analysis? Try it.

5.6.9 Simulating directed motion with random tracking error

We may also want to create a time-series that is a hybrid: We have a particle that moves with constant speed in one direction, but the position determination is contaminated with random tracking errors. The simulation, again, is simple:

```
1 %% --- INITIALIZE ---
2 dt = 1;  % time between recordings
3 t  = 0 : dt : 1000 * dt; % time
4 v  = 7;  % constant translation speed
5
6 %% --- GENERATE POSITIONS ---
7 xPos = v*t + randn( 1, sampleNumber ); % position of bead
     in trap
```

Repeat above analysis for this new time-series. How does the speed determination depend on the degree of smoothing, sub-sampling, or *dt*? Here, the concept of speed does make sense, and averaging over time (smoothing) should give a better determination.

5.6.10 Loading tracking data from a file

Instead of analyzing simulated data we often want to work on actual experimental data. Our example today is the data generated in ImageJ when tracking spots in cells. That output was an XML file and we need a parser (reader) for it called `importtrackmatetracks.m`. This function will return a cell-array of tracks consisting of time, x, y, and z positions:

```
function [tracks, metadata] = importTrackMateTracks(file, clipz, scalet)
%%IMPORTTRACKMATETRACKS Import linear tracks from TrackMate
%
% This function reads a XML file that contains linear tracks generated by
% TrackMate (http://fiji.sc/TrackMate). Careful: it does not open the XML
% TrackMate session file, but the track file exported in TrackMate using
% the action 'Export tracks to XML file'. This file format contains less
% information than the whole session file, but is enough for linear tracks
% (tracks that do not branch nor fuse).
%
% SYNTAX
%
% tracks = IMPORTTRACKMATETRACKS(file) opens the track file 'file' and
% returns the tracks in the variable 'tracks'. 'tracks' is a cell array,
% one cell per track. Each cell is made of 4xN double array, where N is the
% number of spots in the track. The double array is organized as follow:
```

```
17 % [ Ti, Xi, Yi, Zi ; ...] where T is the index of the frame
       the spot has been
18 % detected in. T is always an integer. X, Y, Z are the spot
       spatial
19 % coordinates in physical units.
20 .
21 .
22 .
```

To get a feeling for the data: Pick a few individual tracks and submit them to the same analysis as above. Try a few from the different experimental conditions (try both long and short tracks). Do you notice any difference?

5.7 MSD — Mean Square Displacement

Motivated by the shortcomings of the speed as a measure for motion, we try our hands at another measure. This measure, while a bit more involved, does not suffer the same problems as the speed but takes a little getting used to. Without further ado:

The mean square displacement for a one-dimensional time-series $x(t)$, sampled continuously, is defined as

$$\mathrm{msd}(\tau) \equiv \langle [x(t+\tau) - x(t)]^2 \rangle_t \ , \tag{5.1}$$

where $\langle \cdot \rangle_t$ is the expectation value of the content, with respect to t — think of it as the average over all time-points. What it measures is how far a particle has moved, in an average sense, in a time-interval of size τ.

Figure 5.14 shows theoretical and simulated results for the MSD for three different types of motion: 1) Brownian motion (free diffusion); 2) Brownian motion in an optical trap (confined diffusion); and 3) Random motion with finite persistence (Ornstein-Uhlenbeck process)

One of the beauties of the MSD is that there are no approximations when moving from continuous to discrete time: There are no sampling artifacts.

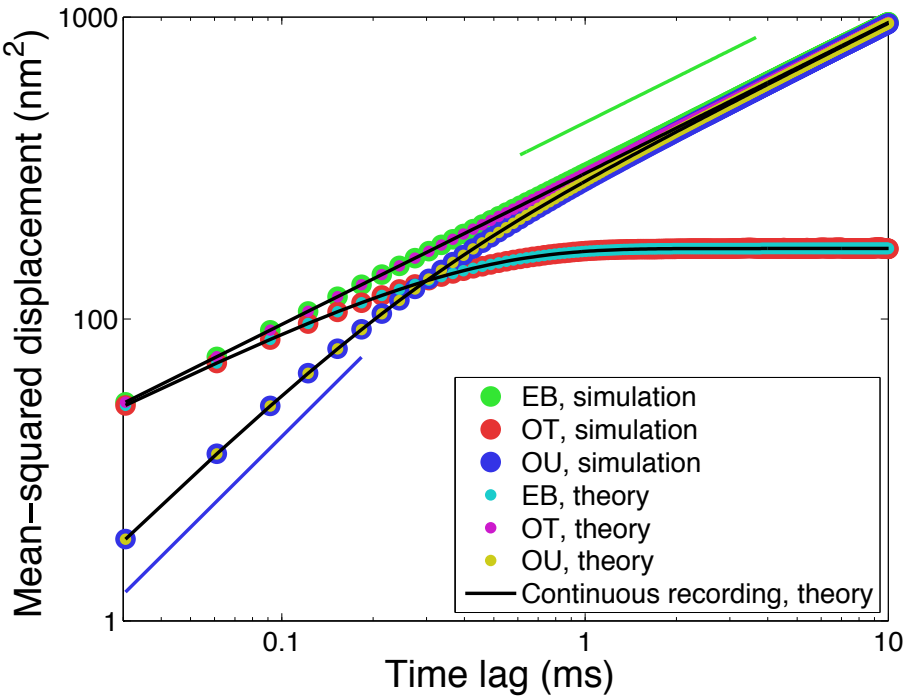

Figure 5.14: Mean-squared displacement for the Ornstein-Uhlenbeck process (persistent random motion), Brownian motion in an optical trap (confined diffusion), and Brownian-motion proper (free diffusion). Straight lines show slopes of one (green) and two (blue), for comparison to the cases of Brownian motion and linear motion. Green points: Freely diffusing massless particle (Einstein's Brownian motion); red points: trapped massless particle (OT limit, or OU velocity process); and blue points: freely diffusing massive particles (time integral of OU process). This is Fig. 8 in Nørrelykke and Flyvbjerg (2011).

For a fixed time-lag, the MSD can be calculate in MATLAB like this:

```
1  function msd_tau = fun_msd_at_tau(x,y,tau)
2
3  % fun_msd_vs_tau FUNCTION
4  % GIVEN INPUT DATA 'X', 'Y' THIS FUNCTION RETURNS THE
5  % MEAN-SQUARED-DISPLACEMENT CALCULALTED IN OVERLAPPING
       WINDOWS
6  % FOR THE FIXED TIMELAG VALUE 'tau'
7  % NB: THIS IS FOR A SINGLE TIMELAG ONLY BUT AVERAGED OVER
       THE ENTIRE TRACK
8
9  % 2016-06-03, sfn, created
10
11 %% --- INITIALIZE ---
12 N    = length( x );            % number of postions
```

```
    determined
13 dr2  = zeros( 1, N - tau );   % initialize and speed up
    procedure
14
15 %% --- CALCULATE THE MSD AT A SINGLE TIMELAG ---
16 for t = 1 : N - tau
17     dx2         = ( x( t + tau ) - x( t ) ).^2; % squared x
           -displacement
18     dy2         = ( y( t + tau ) - y( t ) ).^2; % squared y
           -displacement
19
20     dr2( t )    = dx2 + dy2;                    % store the
           squared x-y-displacement for each postion of the
           sliding window
21 end
22
23 msd_tau     = mean( dr2 ); % The mean of the squared
    displacements calculated in sliding windows
```

In this code-example you should notice that we declared a `function`, used the command `zeros` to pre-allocate memory hence speed up the procedure, and squared each element in a vector with the `.^` operator which should not be confused with the `^` operator that would have attempted to form the inner product of the vector with itself (and fail). A function is much like a normal script except that it is blind and mute: I doesn't see the variables in your workspace and whatever variables are defined inside of the function are not visible from the workspace either. The only way to get data into the function is to feed it explicitly as input, here as `x`, `y`, and `tau`. The only data that gets out is that explicitly stated as output, here `msd_tau`.

Using this function we can now calculate the MSD for a range of time-lags using a for loop

```
1 for tau = 1 : 10
2   msd( tau ) = msd_tau( x, y, tau );
3 end
```

After which we will have a vector of length ten holding the MSD for time-

lags one through ten. If the physical time-units are non-integers you simply plot MSD against these, do not try to address non-integer positions in a vector or matrix, they do not exist. This will become clear the first time you try it.

To build some further intuition for how the MSD behaves, let us calculate it for a couple of typical motion patterns.

5.7.1 MSD — linear motion

By linear motion we mean

$$x(t) = vt \ , \tag{5.2}$$

where v is a constant velocity and t is time. That is, the particle was at position zero at time zero, $x(t = 0) = 0$, and moves to the right with constant speed. The MSD then becomes

$$\mathrm{msd}(\tau) = \langle [vt + v\tau - vt]^2 \rangle = v^2\tau^2 \ , \tag{5.3}$$

i.e., the MSD grows with the square of the time-lag τ. In a double-logarithmic (log-log) plot, the MSD would show as a straight line of slope 2 when plotted against the time-lag τ:

$$\log \mathrm{msd}(\tau) = \log v^2 + 2 \log \tau \tag{5.4}$$

5.7.2 MSD — Brownian motion

By Brownian motion we mean

$$\dot{x}(t) = a\eta(t) \ , \tag{5.5}$$

where $\dot{}$ means differentiation wrt. time, $a = \sqrt{2D}$, D is the diffusion coefficient and η is a normalised, Gaussian distributed, white noise

$$\langle \eta(t) \rangle = 0, \quad \langle \eta(t)\eta(t') \rangle = \delta(t - t') \ . \tag{5.6}$$

See Wikipedia for illustration:
https://en.wikipedia.org/wiki/Brownian_motion

With this equation of motion we can again directly calculate the MSD:

$$\begin{aligned} \mathrm{msd}(\tau) &= \left\langle \left[\int_{-\infty}^{t+\tau} \mathrm{d}t' \dot{x}(t') - \int_{-\infty}^{t} \mathrm{d}t' \dot{x}(t') \right]^2 \right\rangle \qquad (5.7) \\ &= a^2 \tau = 2D\tau \ , \qquad (5.8) \end{aligned}$$

a result that should be familiar to some of you. Apart from prefactors, that we do not care about here, the crucial difference is that the MSD now grows linearly with the time-lag τ, and in a log-log plot it would hence be a straight line with slope one when plotted against τ.

We are much more interested in the mathematical properties of this motion than in the actual thermal self-diffusion coefficient D: The temporal dynamics of this equation can be used to model systems that move randomly, even if not driven by thermal agitation. So, when we say Brownian motion, from now on, we mean the mathematical definition, not the physical phenomenon.

For those interested in some mathematical details, Brownian motion can be described via the Wiener process W, with the white noise being the time-derivative of the Wiener process $\eta = \dot{W}$. The Wiener process is a continuous-time stochastic process and is one of the best known examples of the broader class of Levý processes that can have some very interesting characteristics such as infinite variance and power-law distributed step-sizes. These processes come up naturally in the study of the field of distributions, something you can think of as being a generalization of ordinary mathematical functions, and also requires an extension of normal calculus to what is known as Itô calculus. If you are into mathematical finance or stochastic differential equations you will know all of this already.

5.7.3 MSD — averaged over many tracks

To start quantifying the motion of multiple tracks, we first calculate the mean-squared-displacement for individual tracks

$$\mathrm{msd}_{j,n} = \frac{1}{M_n - j} \sum_{i=1+j}^{M_n} \left((x_i - x_{i-j})^2 + (y_i - y_{i-j})^2 \right) \ , \qquad (5.9)$$

$j = 1, 2, \ldots, M_n - 1$ is the time-lag and M_n is the number of positions determined for track n. Notice, that we use a sliding window so that the $M_n - j$ determinations of the MSDs at time-lag $j\Delta t$ are not independent; this is known as over-sampling and trades independence for smaller error-bars Wang et al. (2007).

The weighted population average is

$$\mathrm{MSD}_j = \frac{1}{\sum_n (M_n - j)} \sum_n (M_n - j)\, \mathrm{msd}_{j,n} \ , \tag{5.10}$$

where the sums extend over all time-series with $M_n > j$. Here, the weights are chosen as the number of intervals that was used to calculate the MSD for a given time-lag and track.

5.7.4 Further reading about diffusion, the MSD, and power-laws

Papers dealing with calculation of the MSD: Qian et al. (1991), under conditions with noise Michalet (2010), theoretical expressions for several generic dynamics cases (free diffusion, confined diffusion, persistent motion both free and confined) Nørrelykke and Flyvbjerg (2011). Determining diffusion coefficients when this or that moves or not, this is an entire PhD thesis compressed to one long paper Vestergaard et al. (2014). How to fit a power-law correctly and what can happen if you do it wrong like everybody else, an absolute must-read Clauset et al. (2009).

5.8 Appendix: MATLAB Fundamental Data Classes

All data stored in MATLAB has an associated class. Some of these classes have obvious names and meanings while others are more involved, e.g. the number `12` is an integer, whereas the number `12.345` is not (it is a `double`), and the data-set `{12, 'Einstein', 7+6i, [1 2 ; 3 4] }` is of the class `cell`. A short video (5min) about MATLAB fundamental classes and data types.

Here are some of the classes that we will be using, sometimes without needing to know it, and some that we won't:

single, double 32 and 64 bit floating number, e.g. `1'234.567` or `-0.000001234`. `double` is the default.

int8/16/32/64, uint8/16/32/64 (unsigned-)integers of 8/16/32/64 bit size, e.g. `-2` or `127`

logical Boolean/binary values. Possible values are `TRUE, FALSE` shown as `1,0`

char characters and strings (largely the same thing), e.g. `'hello world!'`. Character arrays are possible (all rows must be of equal length) and are different from cell arrays of characters.

cell cell arrays. For storing heterogeneous data of varying types and sizes. Very flexible. Great potential for confusion. You can have cells nested within cells, nested within cells ...

struct structure arrays. Like cell arrays but with names.

table tables of heterogeneous but tabular data: Columns must have the same number of rows. Think "spreadsheet". *New data format from 2013b.*

categorical categorical data such as `'Good', 'Bad', 'Horrible'`, i.e., data that take on a discrete set of possible values. Plays well with `table`. *New data format from 2013b.*

5.8.1 MATLAB documentation keywords for data classes

The following is a list of search terms related to the `cell`, `struct`, and `table` data classes. They are titles of individual help-documents and are provided here because the documentation of MATLAB is vast and it can take some time to find the relevant pages. Simply copy and paste the lines into MATLAB's help browser in the program or on the web

```
Access Data in a Cell Array
Cell Arrays of Character Vectors
Multilevel Indexing to Access Parts of Cells
Access Data in a Structure Array
```

```
Cell vs. Struct Arrays
Create and Work with Tables
Access Data in a Table
```

And a video about tables and categorical arrays.

5.9 Appendix: Miscellaneous MATLAB tricks

To find out which toolbox a particular command requires simply search for it in the documentation and notice the path. Alternatively, use the `which` command:

```
>>  which('graythresh')
/Applications/MATLAB_R2015b.app/toolbox/images/images/
   graythresh.m
```

or the `matlab.codetools.requiredFilesAndProducts` command:

```
>> [fileList,productList] = matlab.codetools.
   requiredFilesAndProducts('graythresh');
>> productList.Name

ans =
MATLAB

ans =
Image Processing Toolbox
```

To find out which toolboxes you have installed, simply navigate to the folder where MATLAB is installed via command line or MATLAB or Finder. Example for an installation on a Mac, getting the list in iTerm (bash):

```
[simon@SimonProRetina ~]$ ls -lho /Applications/
   MATLAB_R2015b.app/toolbox/ | head
total 0
drwxr-xr-x    5 simon   170B Oct 13  2015 aero
drwxr-xr-x    7 simon   238B Oct 13  2015 aeroblks
```

```
5 drwxr-xr-x   12 simon   408B Oct 13  2015 bioinfo
6 drwxr-xr-x   23 simon   782B Oct 13  2015 coder
7 drwxr-xr-x   11 simon   374B Oct 13  2015 comm
8 drwxr-xr-x   40 simon   1.3K Oct 13  2015 compiler
9 drwxr-xr-x    8 simon   272B Oct 13  2015 compiler_sdk
10 drwxr-xr-x    8 simon   272B Oct 13  2015 control
11 drwxr-xr-x    8 simon   272B Oct 13  2015 curvefit
12 [simon@SimonProRetina ~]$
```

Here, you need to be able to recognize that the toolbox names are abbreviated, so that, e.g., `Image Processing Toolbox` is referred to simply as `images`. Any path to a function, as found with the `which` command, that inludes `.../toolbox/matlab/...` does not require a specific toolbox as it is part of the core MATLAB distribution.

Bibliography

Aaron Clauset, Cosma Rohilla Shalizi, and M E J Newman. Power-Law Distributions in Empirical Data. *Siam Review*, 51(4):661–703, November 2009.

X Michalet. Mean square displacement analysis of single-particle trajectories with localization error: Brownian motion in an isotropic medium. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 2010.

Simon F Nørrelykke and Henrik Flyvbjerg. Harmonic oscillator in heat bath: Exact simulation of time-lapse-recorded data and exact analytical benchmark statistics. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 83(4):041103, April 2011.

H Qian, M P Sheetz, and E L Elson. Single particle tracking. Analysis of diffusion and flow in two-dimensional systems. *Biophysical Journal*, 60 (4):910–921, October 1991.

C Vestergaard, P Blainey, and H Flyvbjerg. Optimal estimation of diffusion coefficients from single-particle trajectories. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 2014.

Y.M. Wang, H. Flyvbjerg, E.C. Cox, and R.H. Austin. *Controlled Nanoscale Motion: Nobel Symposium 131*, chapter When is a Distribution Not a Distribution, and Why Would You Care: Single-Molecule Measurements of Repressor Protein 1-D Diffusion on DNA, pages 217–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-49522-2. doi: 10.1007/3-540-49522-3_11. URL http://dx.doi.org/10.1007/3-540-49522-3_11.