DynamicAtlas code tutorial for MATLAB-based atlas assembly, timeline creation and fixed-data timestamping

Below follows a tutorial for users who wish to use the atlas methods on a new ensemble of data – whether by using a different set of data in the atlas, or by adding their own data to the atlas. We developed this more intensive code in MATLAB. Users who wish only to query the atlas and perform analysis of data of choice can do so more easily on our Python interface, for which we also have created a separate tutorial.

Example screenshots will be shown from running the code on a 2023 MacBook Pro (Apple M3 Pro chip) with 18 GB RAM and 512 GB storage, running MacOS Sequoia 15.1. However, the code is platform-independent, and the same procedure also works on the Windows or Linux operating systems (tested on Windows 10 and Ubuntu 18.04.6 LTS). The code should take on the order of 45 minutes to run on the demo dataset, depending on machine capabilities.

—

Table of Contents:

—

Required Specs and Download Instructions

Below follows information about the required specifications for running the code, and information for the required downloads: MATLAB, demo data, and DynamicAtlas code.

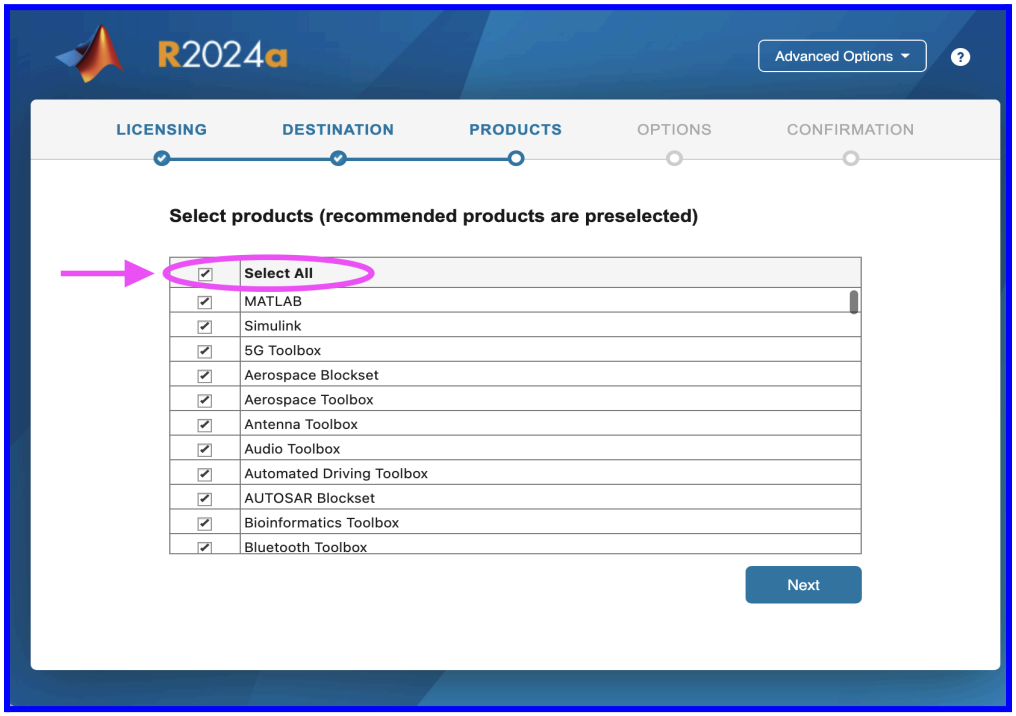
Required Specs:

Any major operating system: Windows, MacOS, Linux
30 GB hard drive space (~25 GB for MATLAB, <5 GB for data & code download)
18 GB RAM or more (may still run with less, but subject to stalling or slowing)
Recent MATLAB version, from R2020a onwards (shown below on R2024a)
Tool to unzip compressed files (like 7Zip on Windows, ArchiveUtility on MacOS, or ArchiveManager on Linux)

Downloading MATLAB:

Follow the instructions listed here to download the MATLAB application:
https://www.mathworks.com/help/install/ug/install-products-with-internet-connection.html

During the installation process, a screen will appear inquiring which toolboxes to download. Check “select all” on this screen (shown below), as some functions used by DynamicAtlas are defined in the toolboxes.

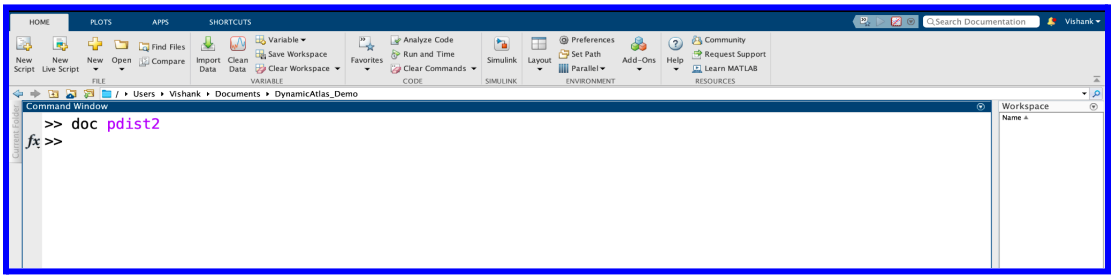


NOTE: A pre-installed version of Matlab may be used, but it is possible a function may not be recognized due to a toolbox not being previously downloaded. This may yield an error similar to the following (here, function 'pdist2' is missing):

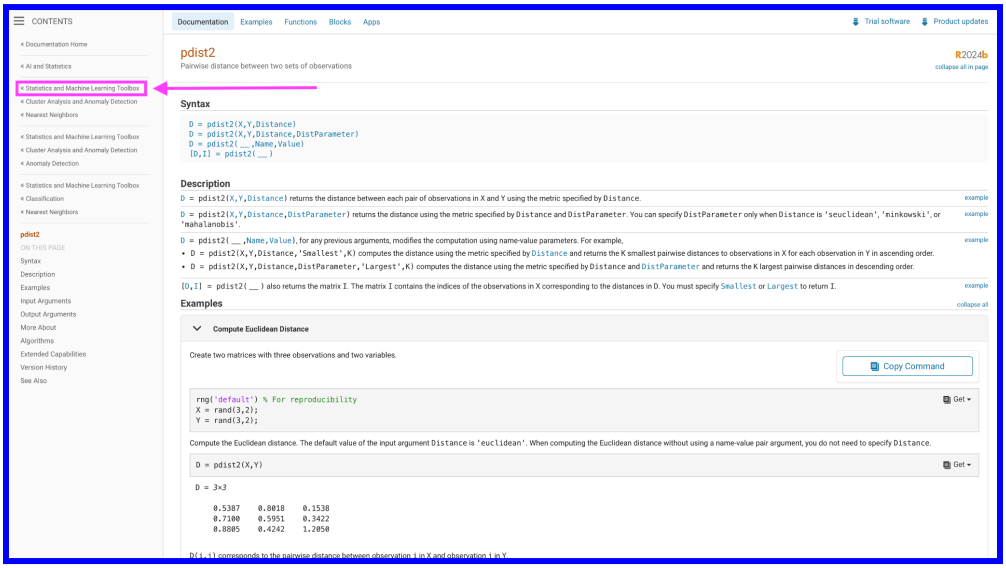
```
Undefined function 'pdist2' for input arguments of type 'double'.

Error in dynamicAtlas.dynamicAtlas/makeMasterTimeLineRealspace (line 1231)
                        d1 = pdist2(cI, cJ);

Error in dynamicAtlas.dynamicAtlas/makeMasterTimeline (line 200)
                da.makeMasterTimeLineRealspace(genotype, label, Options)

Error in demo_dynamicAtlas_functionality (line 127)
da.makeMasterTimeline('WT','Runt', Options)
```

This can be fixed by looking up the toolbox containing this function, and downloading it, see instructions for downloading and adding new toolboxes here: (https://www.mathworks.com/help/install/ug/add-to-existing-installation.html).

To look up the toolbox of a given function, one method is to use the 'doc' command to access the function documentation, as below for 'pdist2':



which will open the function's documentation webpage. The toolbox containing the function can be found at the upper left corner of the documentation page, as shown (here, the Statistics and Machine Learning Toolbox contains 'pdist2'):



Downloading Demo Data:

We have uploaded our Atlas data to a Dryad repository (https://datadryad.org/stash/dataset/doi:10.25349/D9WW43 — downloads work best on the Firefox or Google Chrome browsers).

For this tutorial, however, we have assembled a small Demo dataset with minimal inputs. This demo dataset, which contains five Runt-labeled samples from the Atlas, can be accessed at its own link (hosted as a Zenodo data repository):

https://doi.org/10.5281/zenodo.14792464

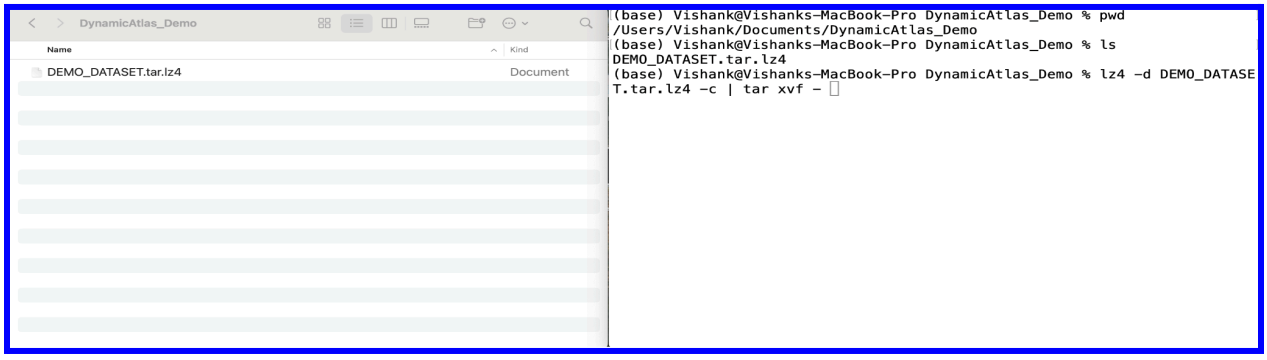
with the demo file (~1 GB) called

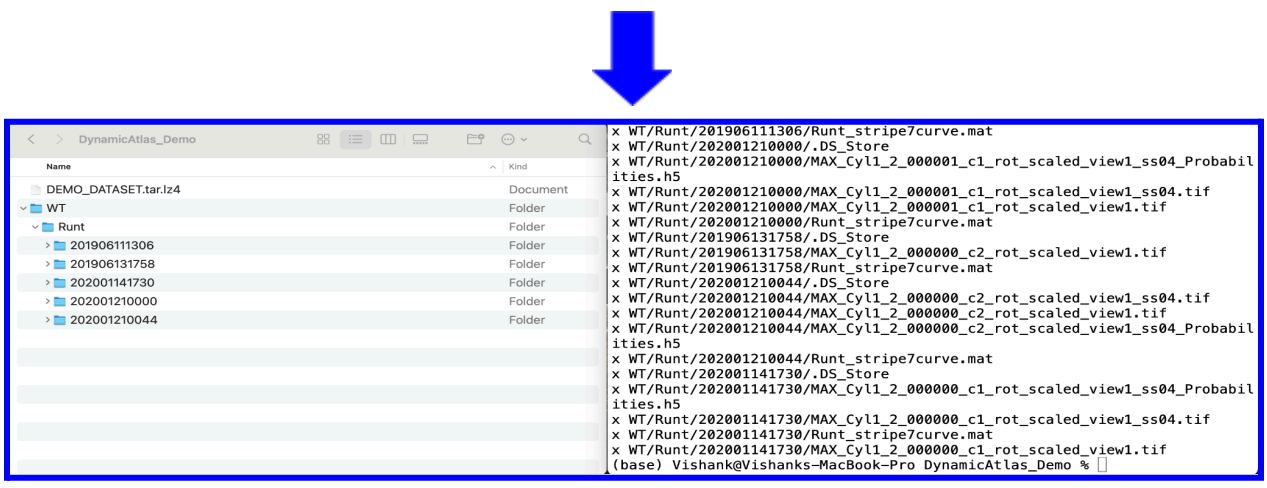
DEMO_DATASET.tar.lz4

Download this dataset to a directory of choice. This data is in a compressed file format, and can be extracted with an archive utility. The extraction can also be done through a command line (Terminal on MacOS/Linux, Command Prompt on Windows), for example with the command…

lz4 -d DEMO_DATASET.tar.lz4 -c | tar xvf -

…as demonstrated below, before and after executing the command (on MacOS):





(Note pwd lists the working directory, and ls lists the files.) The data will then be extracted to a folder structure. The above extraction works on Linux similarly. To extract a .tar.lz4 file on Windows, if 7Zip doesn't work (a possibility), one can follow the method described here: https://superuser.com/questions/1292802/how-to-decompress-lz4 This will convert the .tar.lz4 to a .tar, and 7Zip will then work as normal on this file, as will the Command Prompt command 'tar'. Extraction may also create extra .DS_STORE and _MACOSX files; delete them (these just signify the file was compressed on a Mac).

For our code, we use the following directory convention:

FOLDER CONVENTION: The data must be organized in folders as

Genotype > Marker > Dataset ID

For example, WT > Runt > 201906111306. If folders are not organized with this convention, errors may occur when the function buildLookupMap.m is called to construct the Atlas object. Depending on the file downloaded and the extraction method, folders may need to be manually created or arranged to ensure this convention.

Downloading DynamicAtlas code:

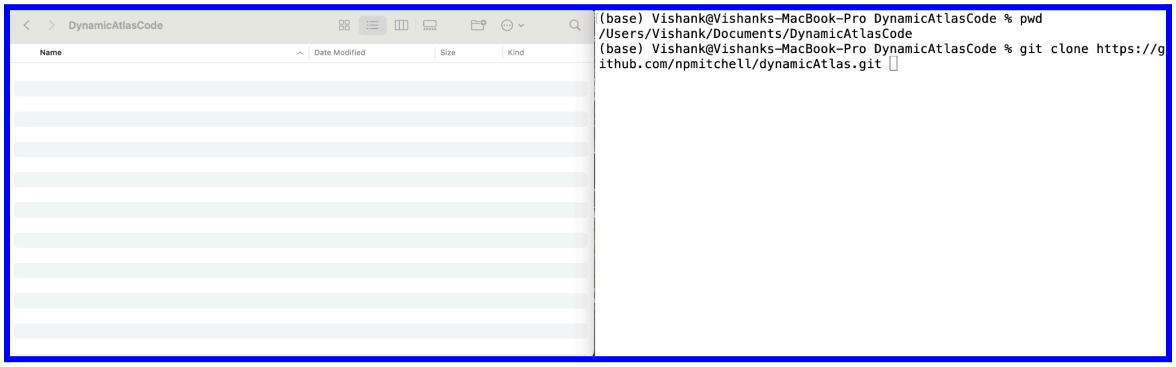
Code for the MATLAB-based atlas methods is located at the following Github repository:

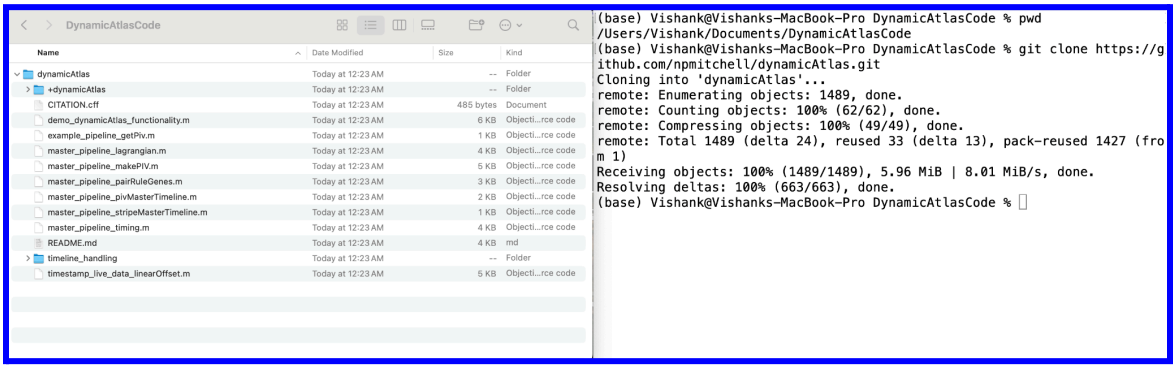
https://github.com/npmitchell/dynamicAtlas

The code can be downloaded to a folder of choice from the GitHub website, or, preferably, cloned using the command line, e.g. with the command…

git clone https://github.com/npmitchell/dynamicAtlas.git

…as depicted with the following screenshots before and after executing the command:





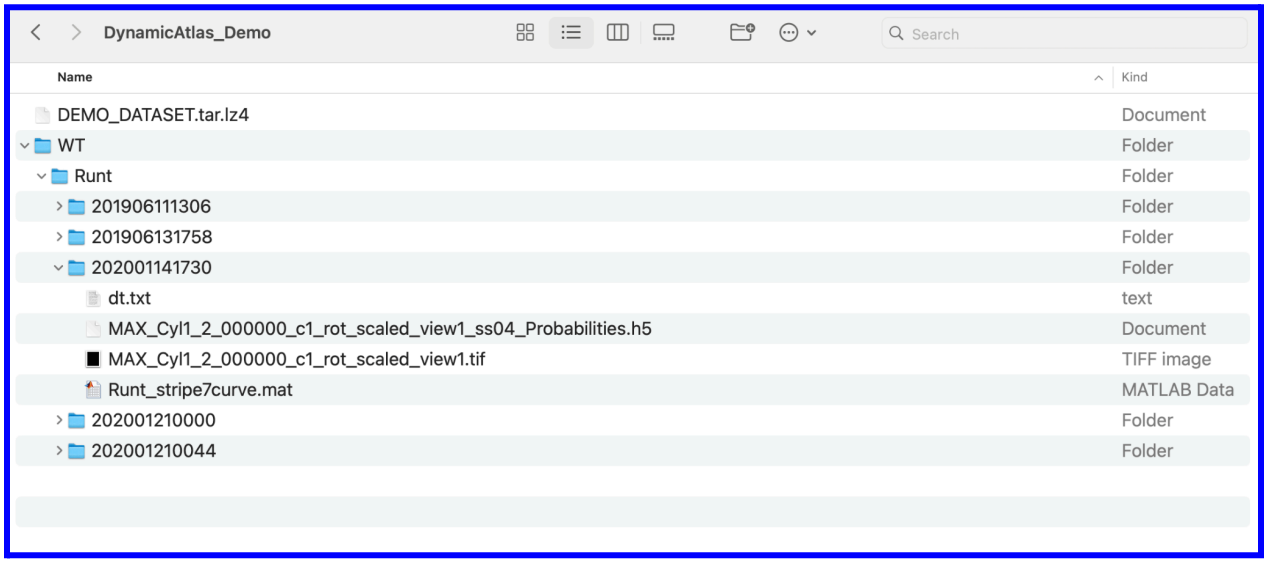
This folder, together with the MATLAB native library and toolboxes, comprises all the dependencies needed to run the MATLAB-based code. The code that has been written for this tutorial is contained within the…

demo_dynamicAtlas_functionality.m

…script, and what follows below will be a walkthrough of this code demonstrated on the above demo dataset.

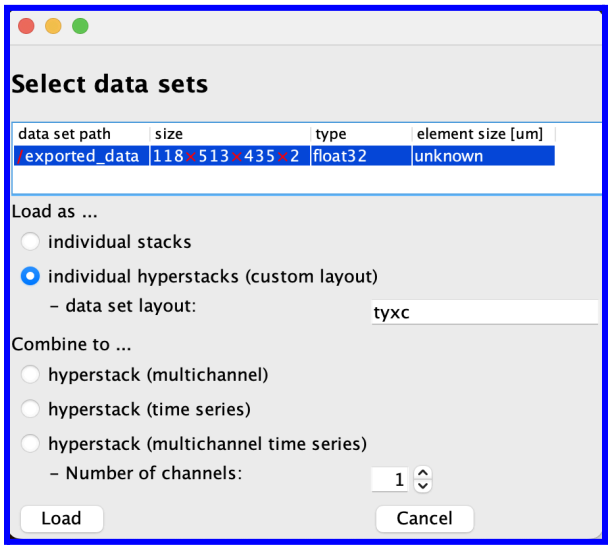
Demo code walkthrough: Data Conventions

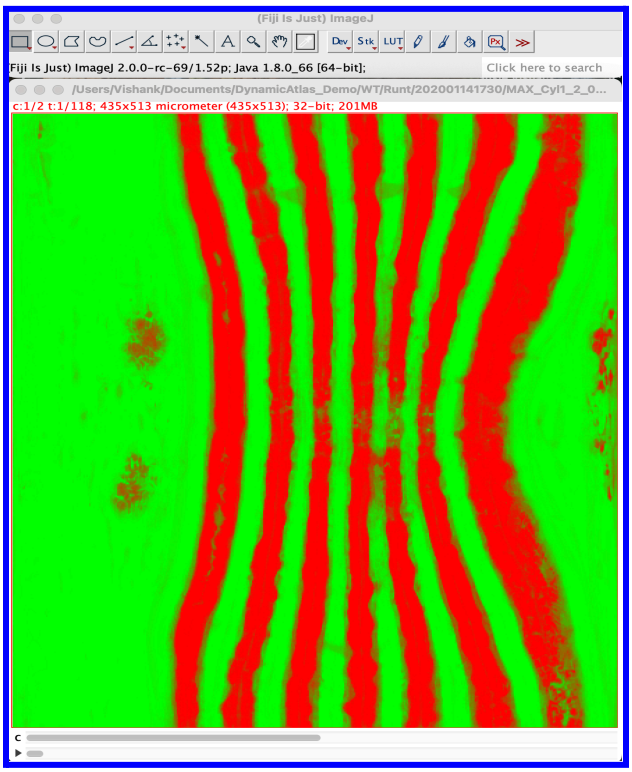
The Atlas code requires data to follow standard conventions across experiments, in terms of both naming conventions and content. The demo code demonstrates the creation of a master timeline using gene expression geometry to determine morphological time (see Figure 2 of the manuscript). Shown below are the minimal inputs required for running this code, for an example live dataset (ID: 202001141730):



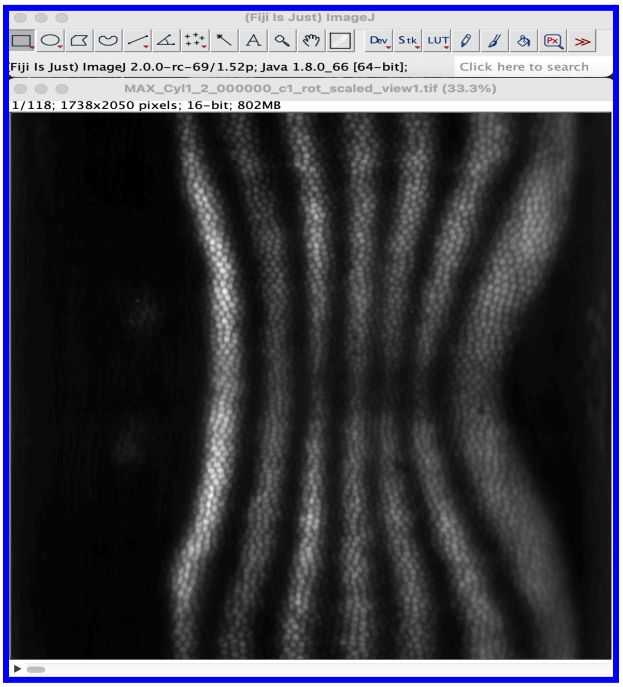
There are four files required. First, is a text file, ‘dt.txt’, which contains the time resolution of the data as a single number, in minutes per frame. The demo data was taken at 1 minute time resolution, so the file just contains the number 1, as shown:

The second file is an h5 ‘probabilities’ file where each pixel is assigned a probability of being part of the expression pattern, based on a user’s training. This can be generated with the program Ilastik (https://www.ilastik.org/), or with a similar pixel classification algorithm. Shown below is the example h5 file loaded into Fiji (via File > Import > HDF5, note that the dataset must be selected and the dimensions listed must match those of the stack):



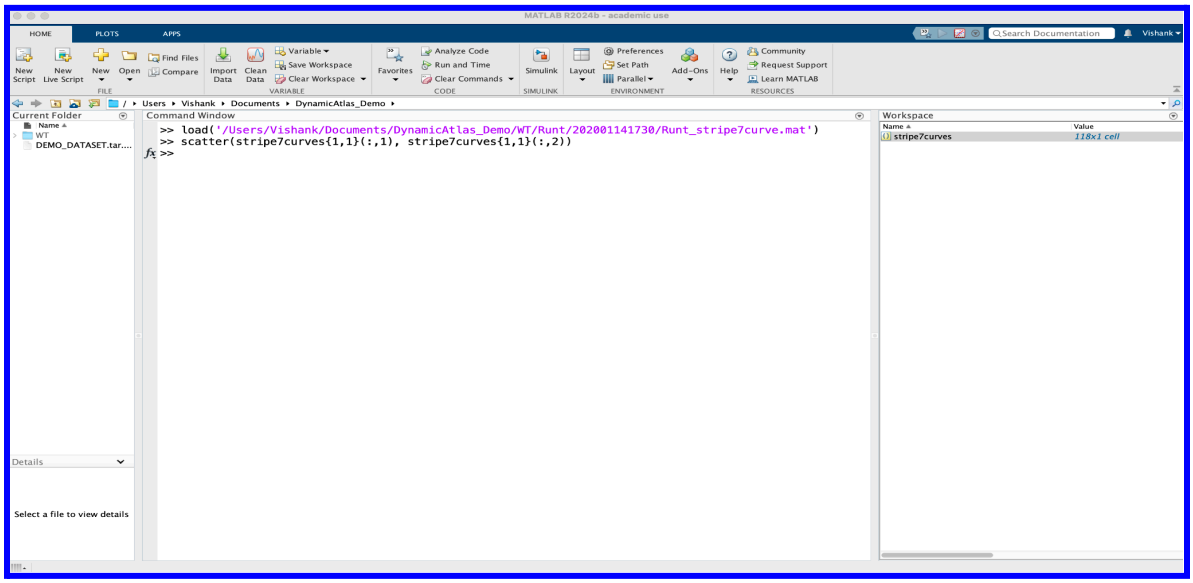


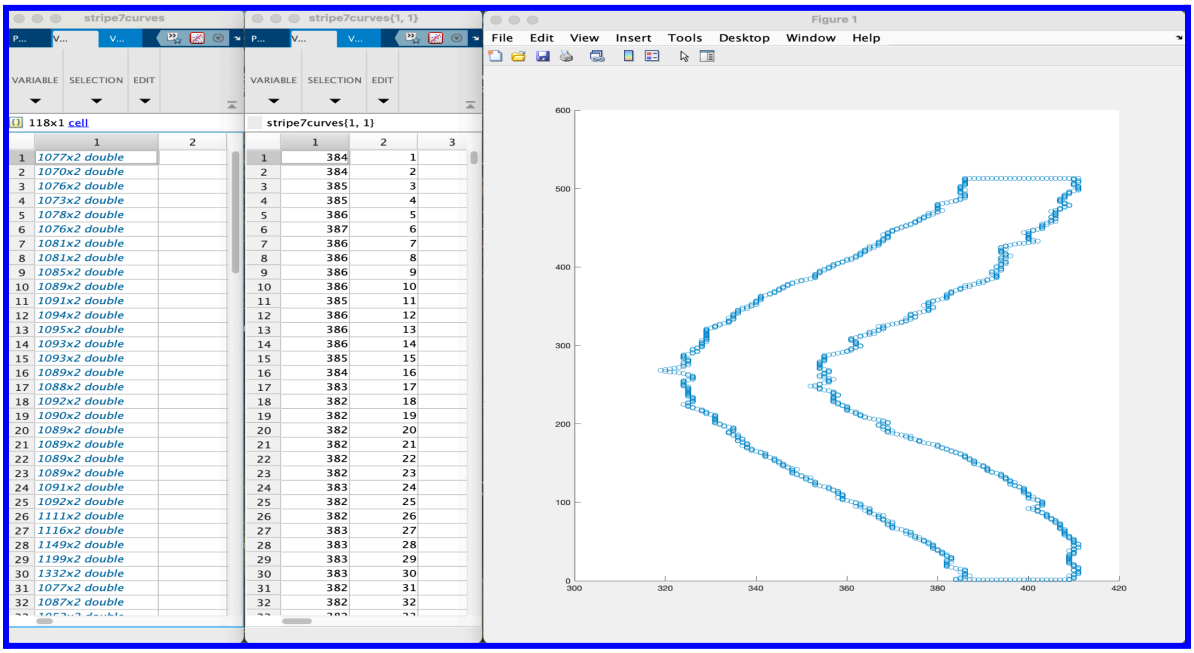
The third file is the data itself, a multipage tiff stack showing a movie of the projection of the surface over time. Opening this example file in Fiji shows the following:



By convention, all of the data take the form of map projections with spatial size of 1738x2050 pixels (this resizing yields an error of order ~4%, see SI). This enables all datasets to be quantitatively compared to each other in space. The datasets are also all named with the convention: "MAX_Cyl1_2_******_c*_rot_scaled_view1.tif" where the asterisks are replaced with time and channel respectively, though a different convention can be chosen as long as suitably specified in the options fed into the function.

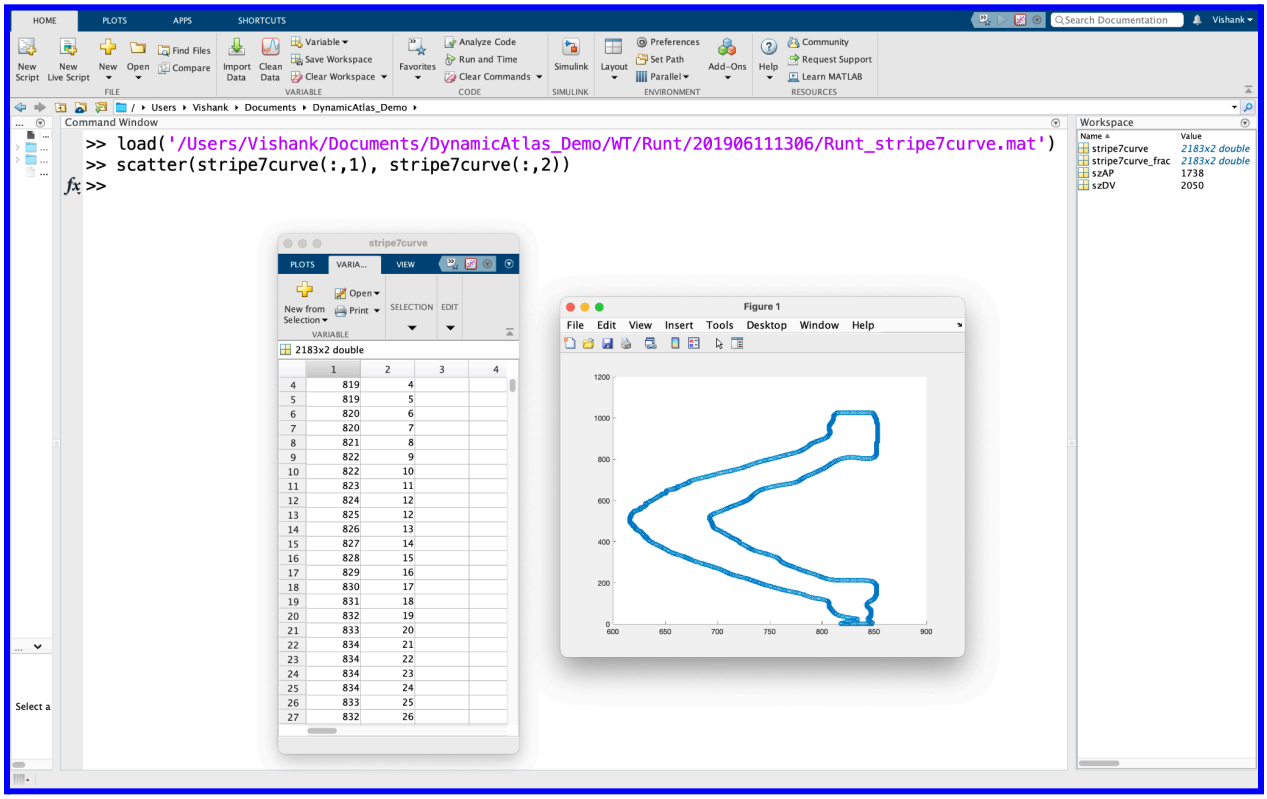
The fourth file is a Matlab variable in the .mat format containing the xy-coordinates of the detected expression pattern boundary, in this case the boundary of Runt stripe 7 ('Runt_stripe7curve.mat'). The atlas code has functions for performing the boundary detection (see dynamicAtlas/+dynamicAtlas/stripeExtraction) but this can also be done using another method of choice, as long as the result is saved in the same format. Shown below are the Matlab variable, and the (x,y) points plotted in space, for the first frame of the example movie, with data for the curves contained in 'stripe7curves':





Each live dataset requires these four files. Fixed datasets do not require the .h5 file or the .txt file, but each fixed dataset does require the .tif file and the .mat file.

For the fixed datasets, the .mat file denotes the curve by a similar variable (stripe7curve). Additionally, the file will have three extra variables, denoting the coordinates as fractional positions along the projection (stripe7curve_frac), and also variables indicating the AP and DV dimensions (szAP, szDV), as measured by the pixel dimensions of the image projections they came from. In the convention followed by the existing .tif data (described above), szAP will be 1738, and szDV will be 2050. An example .mat variable for a fixed dataset is depicted below:

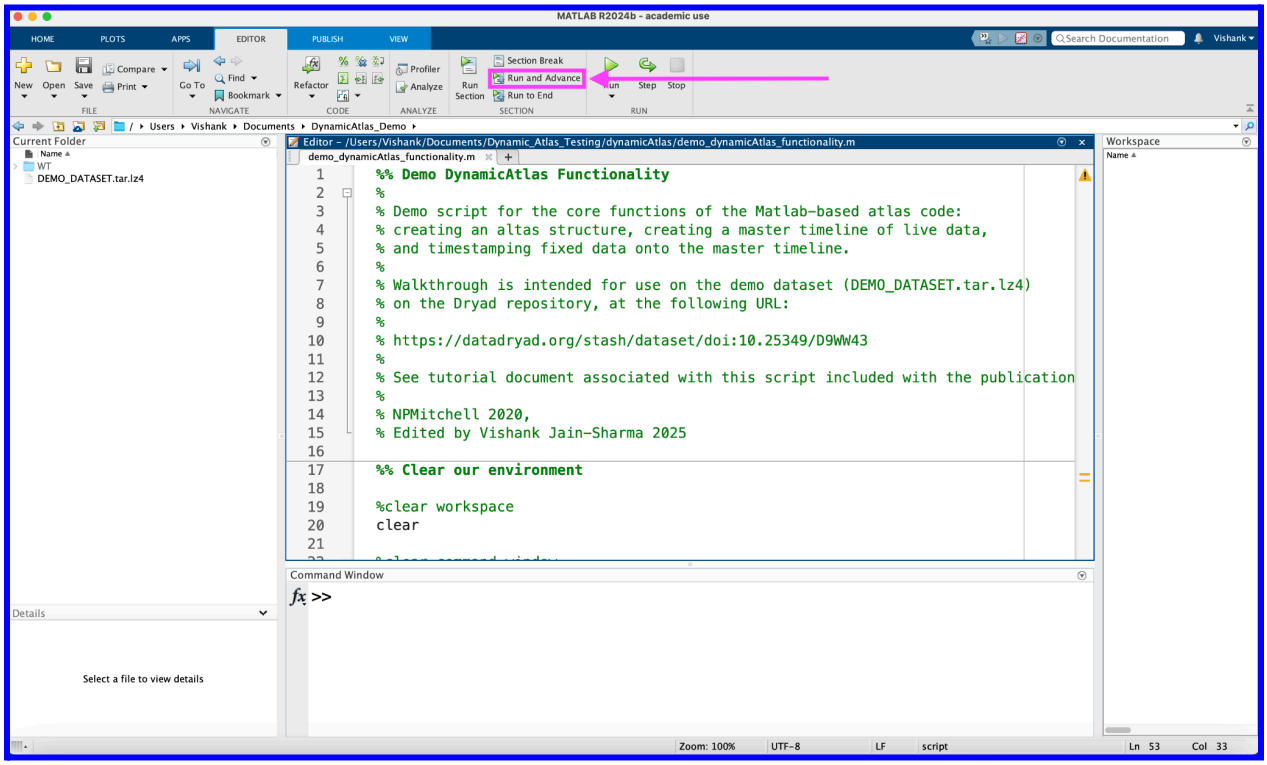


Given the aforementioned files, with the appropriate consistent naming conventions, the core atlas code shown in the demo script can be successfully run.

Demo code walkthrough: Atlas Assembly

[Blocks I - IX]

With the data following the above conventions, the demo code can now be run. The first part of the code involves creating a dynamicAtlas object, which allows one to load data of choice and explore their properties (this is similar to the functionality of the Python-based interface that is also included in the Atlas, see SI). The code is separated into distinct blocks (via the %% symbols), which can be run sequentially in Matlab using the 'Run and advance' button in the Matlab editor, as shown:



The walkthrough will now show the individual blocks of this code one by one, along with explanations and outputs. Note that comments in the code are included with the % symbol. To follow along interactively, open Matlab, and open the script

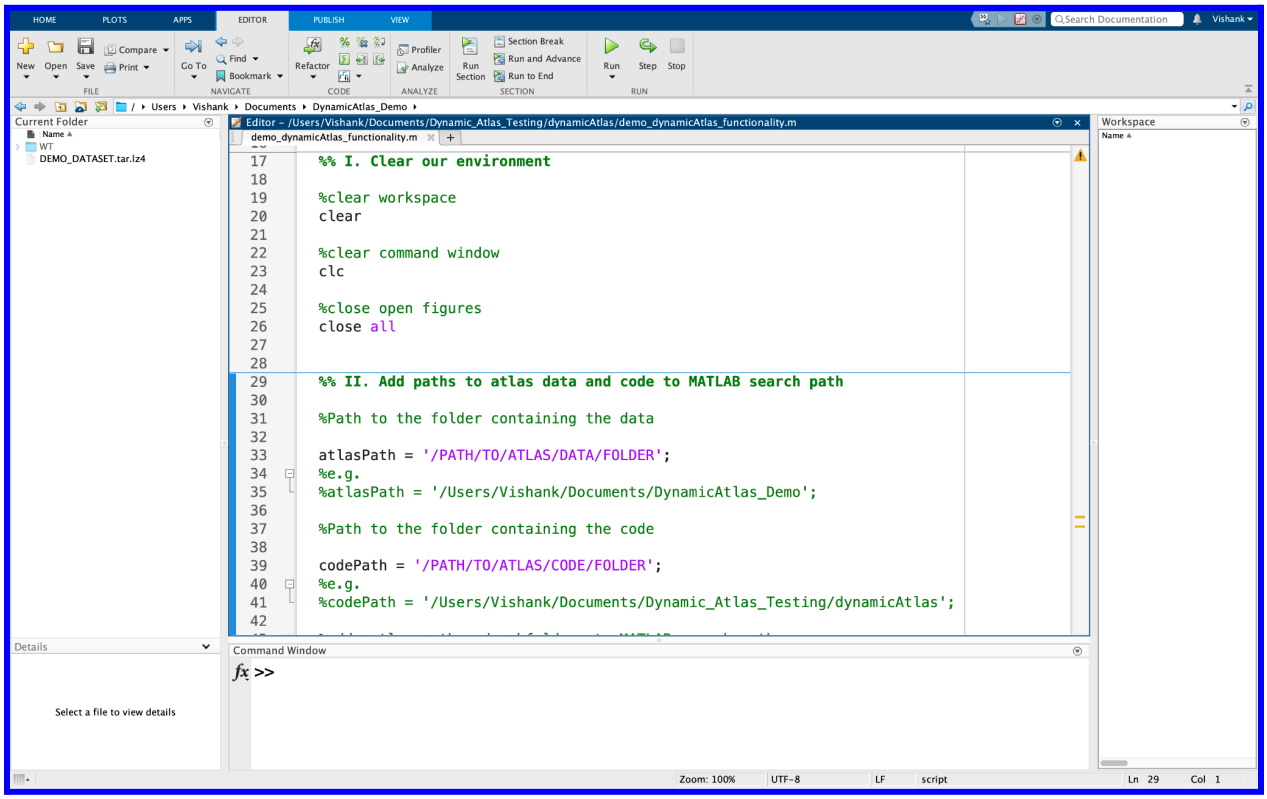
demo_dynamicAtlas_functionality.m

in the Matlab editor.

The first block of code (after the introductory comments), **Code block I**, merely serves to clear our Matlab environment:

```
%% I. Clear our environment

%clear workspace
clear

%clear command window
clc

%close open figures
close all


```

In particular, this clears our workspace, our command window, and closes all open figures. This helps to avoid duplicate variables carrying over from previous work. The result will look like this:

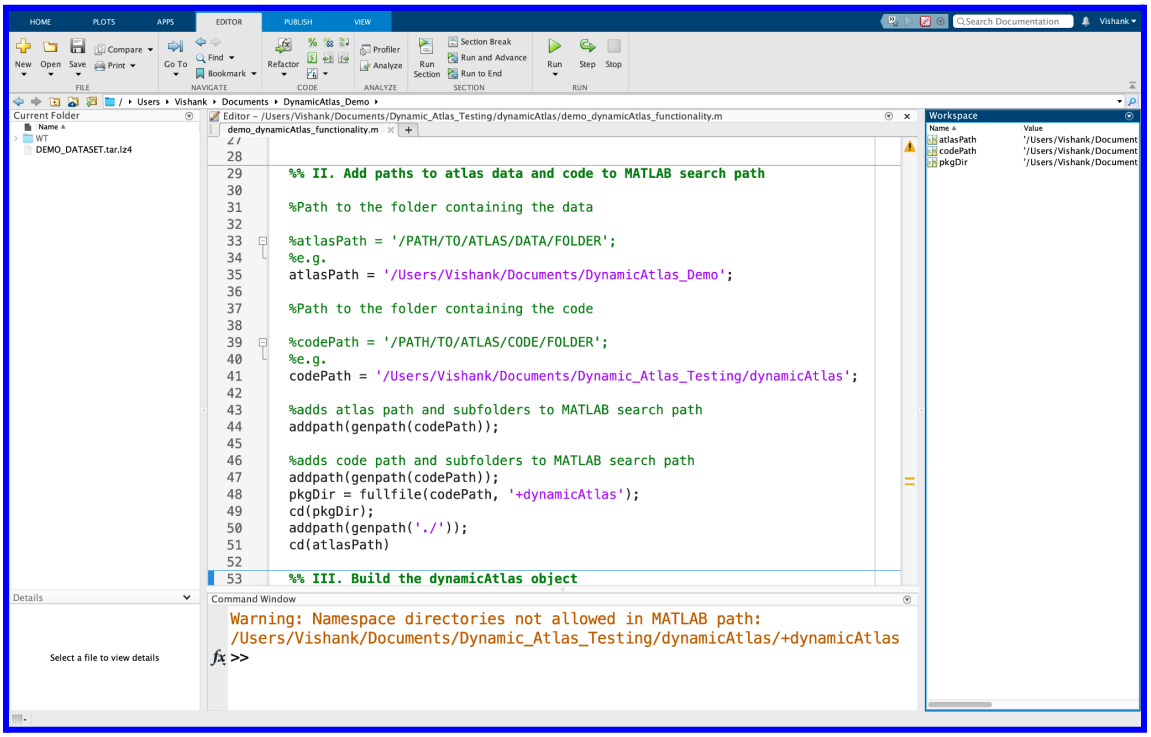


Code block II adds the necessary paths to the Matlab search path, so that Matlab is able to read information from them. In particular: the path to the atlas data, and the path to the atlas code. (If the paths are not explicitly defined, Matlab can throw errors simply based on not finding the appropriate files.) The user should input the path to the

directory where the atlas data is located (here, the directory containing the folder ‘WT’) into the variable atlasPath. The user should input the path to the directory ‘dynamicAtlas’ (i.e. the code folder that the user downloaded to a location of choice) into the variable codePath. The block adds these paths, and all their subfolders, to the Matlab search path (a warning might appear – ignore it).

```
%% II. Add paths to atlas data and code to MATLAB search path

%Path to the folder containing the data

%atlasPath = '/PATH/TO/ATLAS/DATA/FOLDER';
%e.g.
atlasPath = '/Users/Vishank/Documents/DynamicAtlas_Demo';

%Path to the folder containing the code

%codePath = '/PATH/TO/ATLAS/CODE/FOLDER';
%e.g.
codePath = '/Users/Vishank/Documents/Dynamic_Atlas_Testing/dynamicAtlas';

%adds atlas path and subfolders to MATLAB search path
addpath(genpath(codePath));

%adds code path and subfolders to MATLAB search path
addpath(genpath(codePath));
pkgDir = fullfile(codePath, '+dynamicAtlas');
cd(pkgDir);
addpath(genpath('./'));
cd(atlasPath)

```

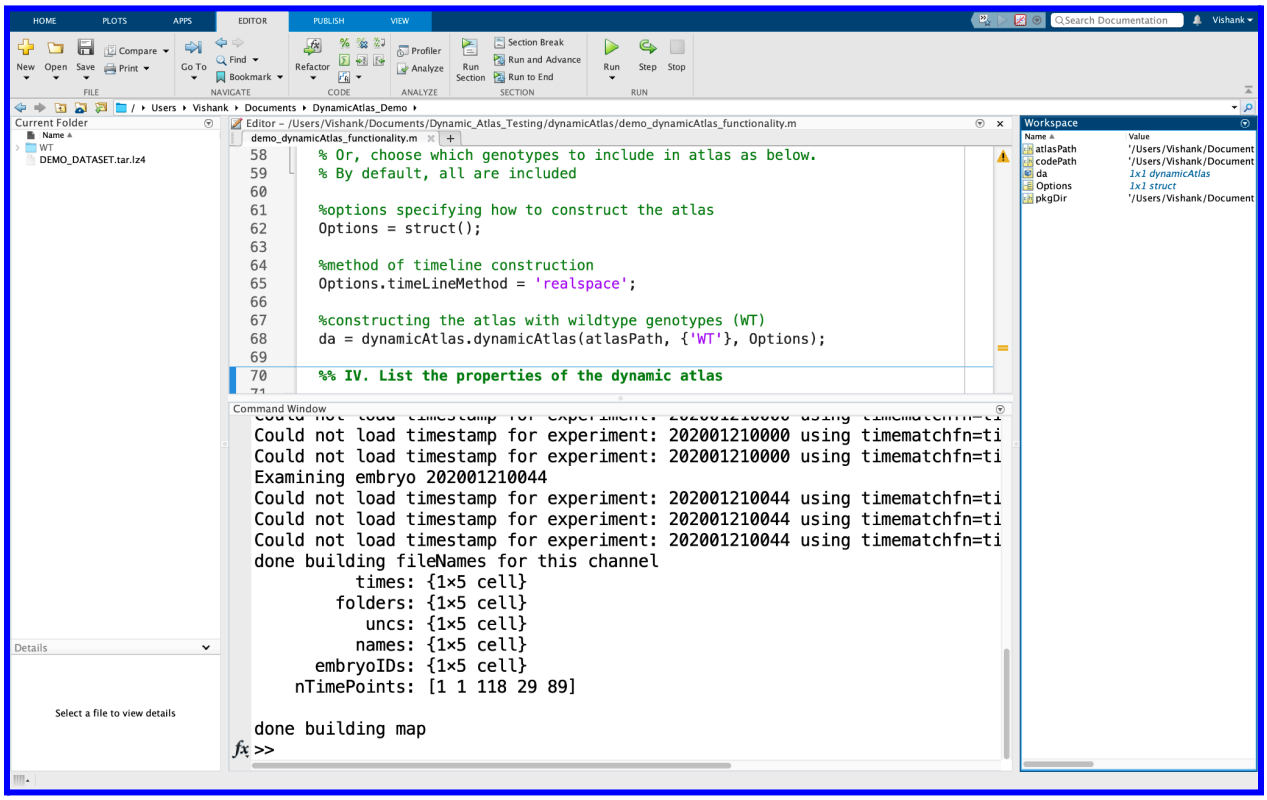
The result will look similar to the following:



Code block III builds a dynamicAtlas object, which is a structure that represents the atlas as a whole. The user can choose which genotypes to include in the atlas by specifying them as an argument in the atlas definition (line 68 below).

```
53  %% III. Build the dynamicAtlas object
54  %
55  % Build dynamic atlas with all genotypes in the atlasPath using:
56  % da = dynamicAtlas.dynamicAtlas(atlasPath);
57  %
58  % Or, choose which genotypes to include in atlas as below.
59  % By default, all are included
60
61  %options specifying how to construct the atlas
62  Options = struct();
63
64  %method of timeline construction
65  Options.timeLineMethod = 'realspace';
66
67  %constructing the atlas with wildtype genotypes (WT)
68  da = dynamicAtlas.dynamicAtlas(atlasPath, {'WT'}, Options);
69
```

Here, the demo atlas only has the wildtype genotype, so the argument used is just {‘WT’}. In general, this should be specified as { ‘Genotype1’, ‘Genotype2’, … }. If no argument is specified, all genotypes are included by default. The code attempts to find information about the data contained within the folders corresponding to these genotypes, and loads in all the metadata it can find. The output will look like this:

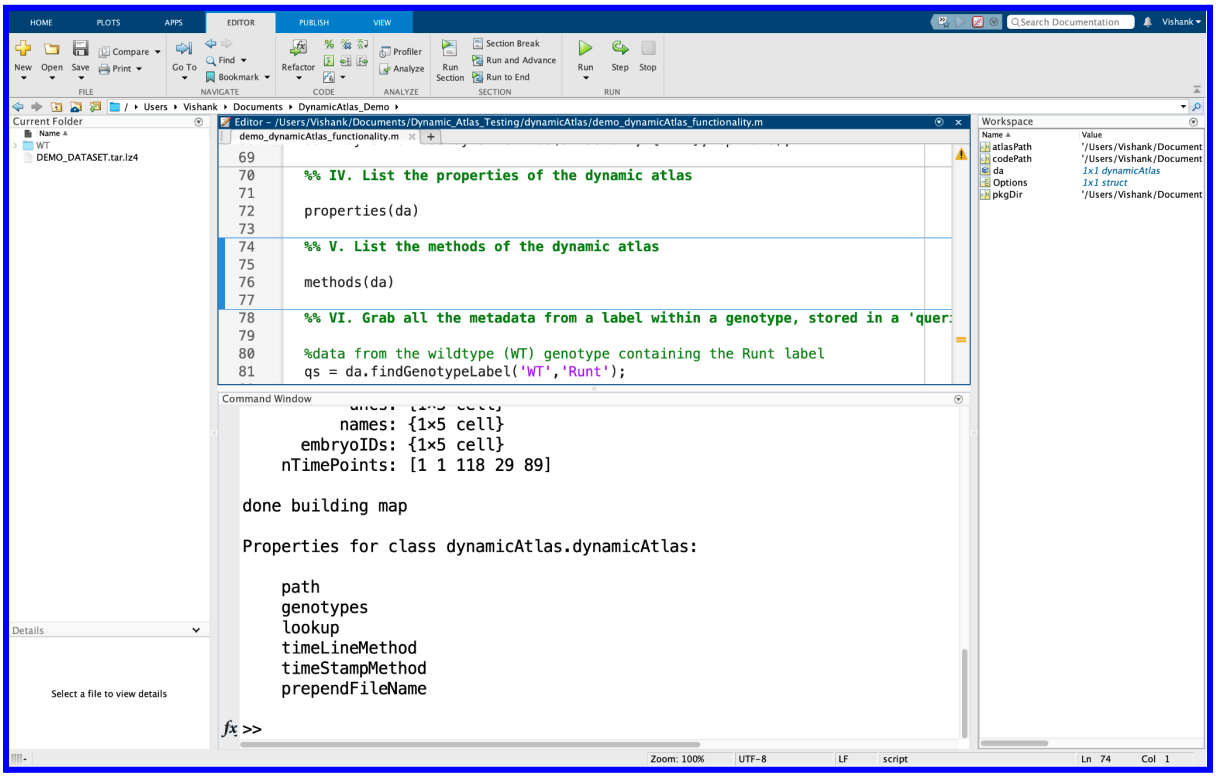


Shown are examples of the information found for the demo dataset. For example, there are two fixed datasets here (201906111306 and 201906131758), and so there are two corresponding entries of 1 in nTimePoints. The variable ‘da’ here represents the atlas object, and other methods in the code interact with this object to perform computations.

Code block IV lists the properties of the dynamicAtlas object:

```
%% IV. List the properties of the dynamic atlas

properties(da)

```

and will show an output like:

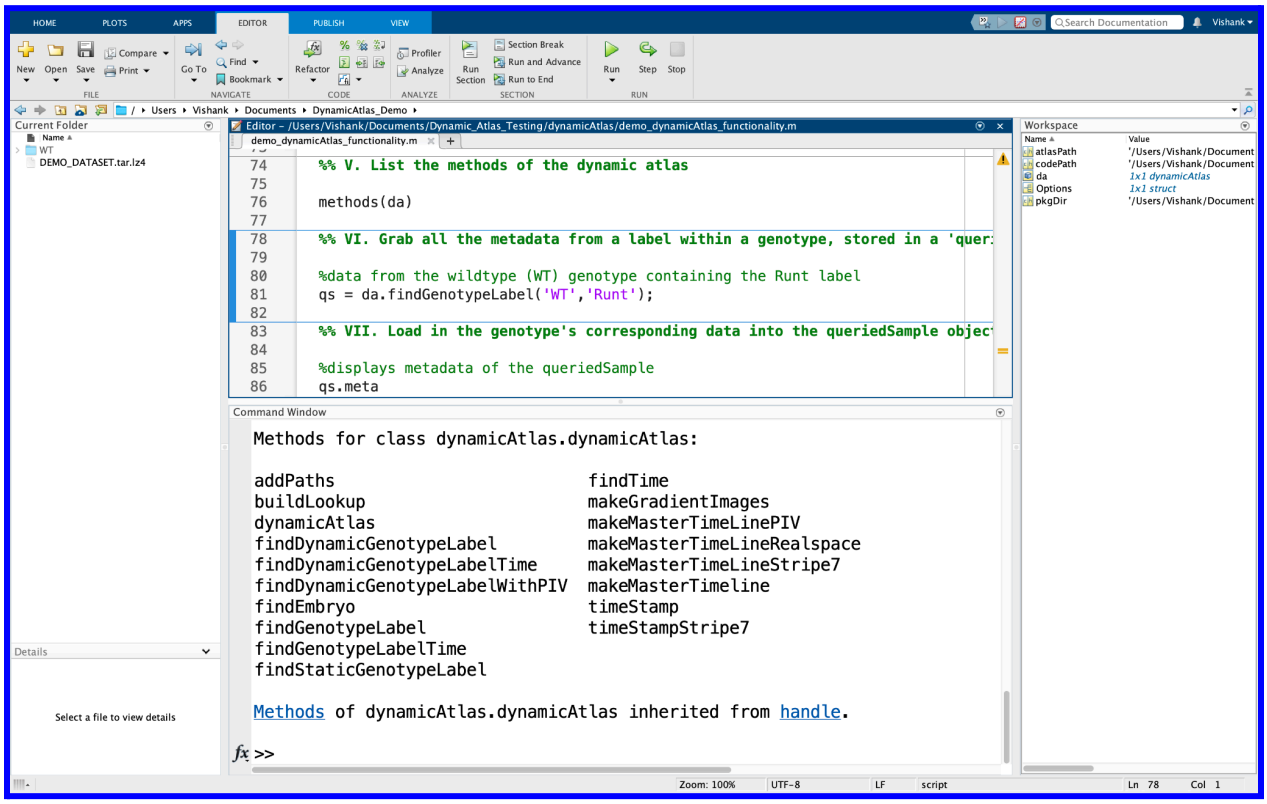


These are the fields of information contained in the ‘da’ variable, which can also be examined by double clicking on the variable in the Matlab workspace.

Code block V lists the methods in the dynamicAtlas software that can be invoked by the dynamic atlas object:

```
%% V. List the methods of the dynamic atlas

methods(da)

```

and will show an output like:

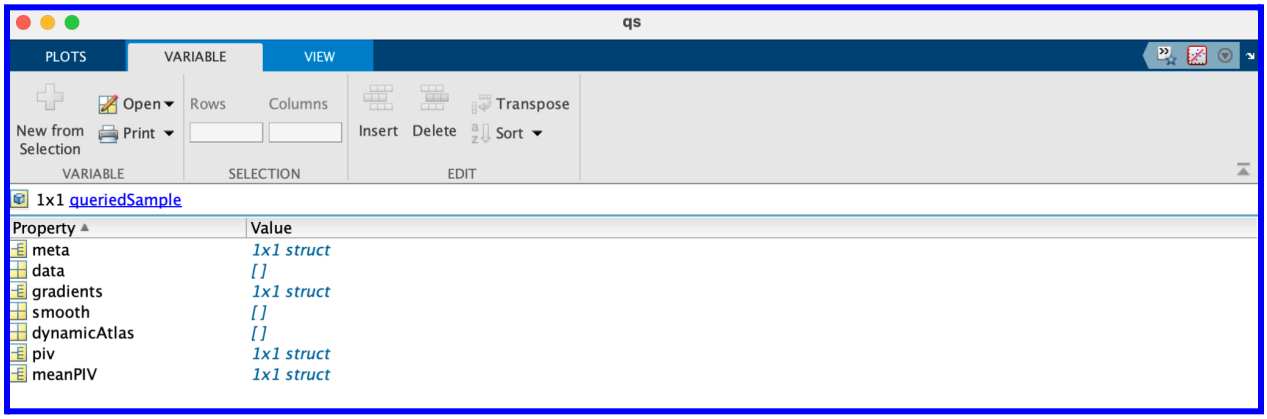


These can be invoked by any dynamicAtlas object, such as ‘da’ above.

Code block VI grabs all the metadata from a label (e.g. Runt) within a given genotype, and stores this within an object called a queriedSample.

```
78    %% VI. Grab all the metadata from a label within a genotype, stored in a 'queriedSample'
79
80    %data from the wildtype (WT) genotype containing the Runt label
81    qs = da.findGenotypeLabel('WT','Runt');
82
```

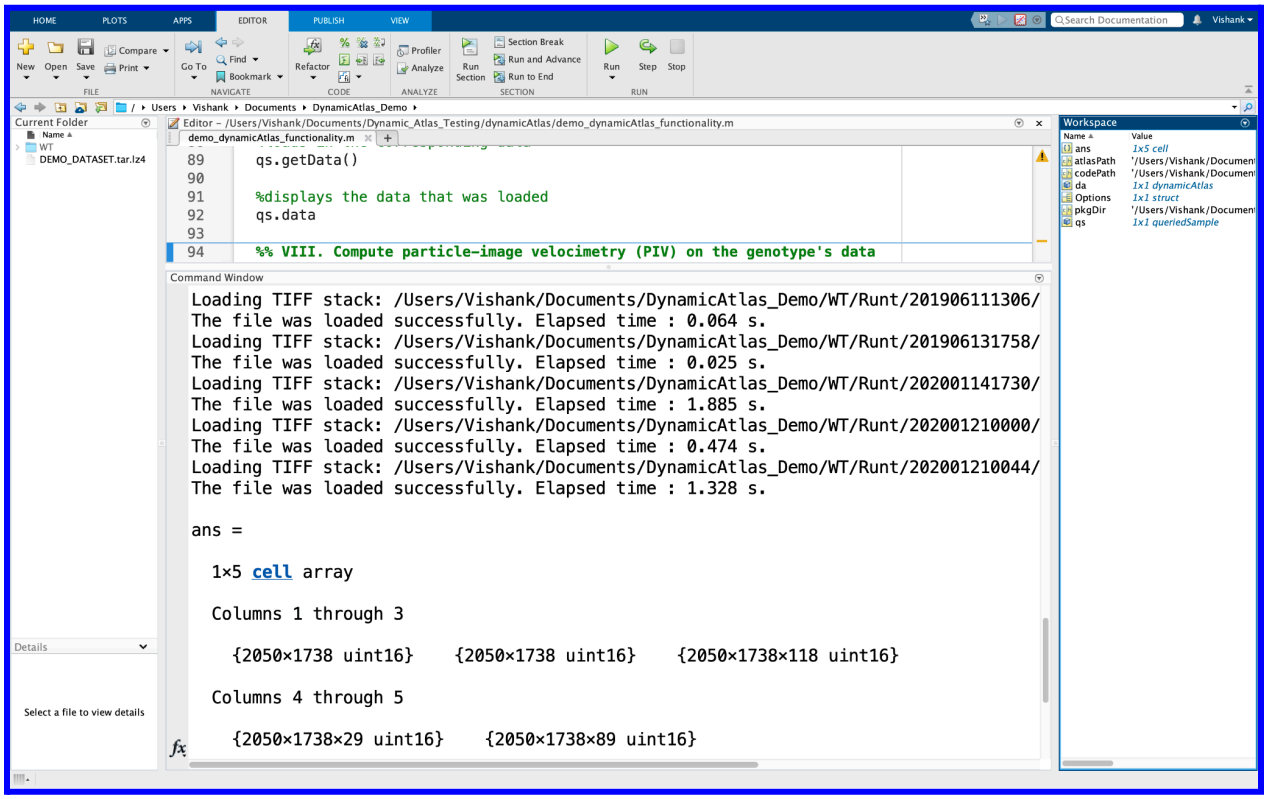
There is no additional output to the command window, but the queriedSample variable ‘qs’ is generated, as below. The fields will contain the metadata, and places where data can be stored.



Code block VII uses the metadata pointed to by qs to load in the data, and displays the contents of qs to the command window.

```
%% VII. Load in the genotype's corresponding data into the queriedSample object

%displays metadata of the queriedSample
qs.meta

%loads in the corresponding data
qs.getData()

%displays the data that was loaded
qs.data

```

The output will look like below:

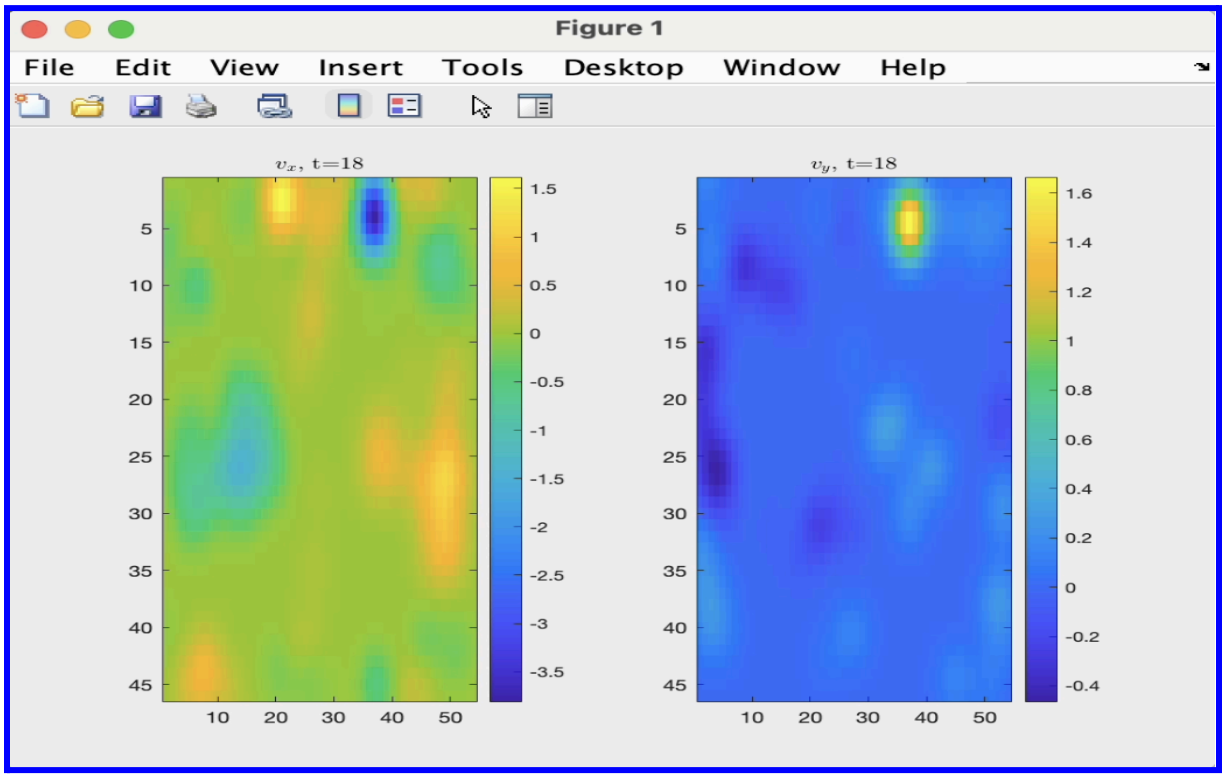


The data contained within the tiff files of the folders assigned to a label (here, embryos labeled for Runt within the WT genotype) are loaded into qs one by one, and the tiff dimensions are displayed in the command window. Contents are stored in qs.data, and can be accessed there.

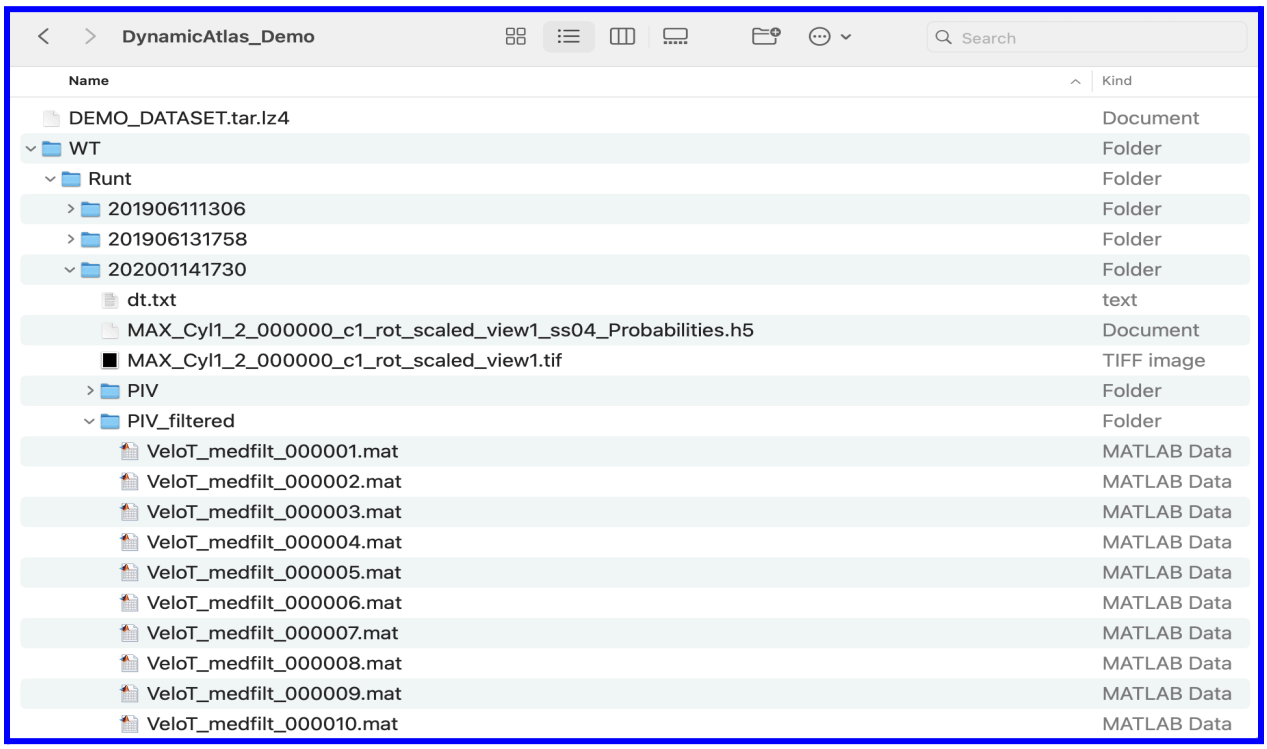
Code block VIII computes the flow on the data within the queriedSample, via particle image velocimetry (PIV). (Note that if PIV has already been computed and one does not wish to re-compute it again, Options.overwrite can be changed to 'false', and this will be skipped.)

```
94    %% VIII. Compute particle-image velocimetry (PIV) on the queriedSample data
95
96    %options specifying how to compute the PIV
97    Options = struct();
98
99    %overwrite existing PIV computations
100   Options.overwrite = true;
101
102   %computes PIV on the queriedSample data
103   qs.ensurePIV(Options);
104
```

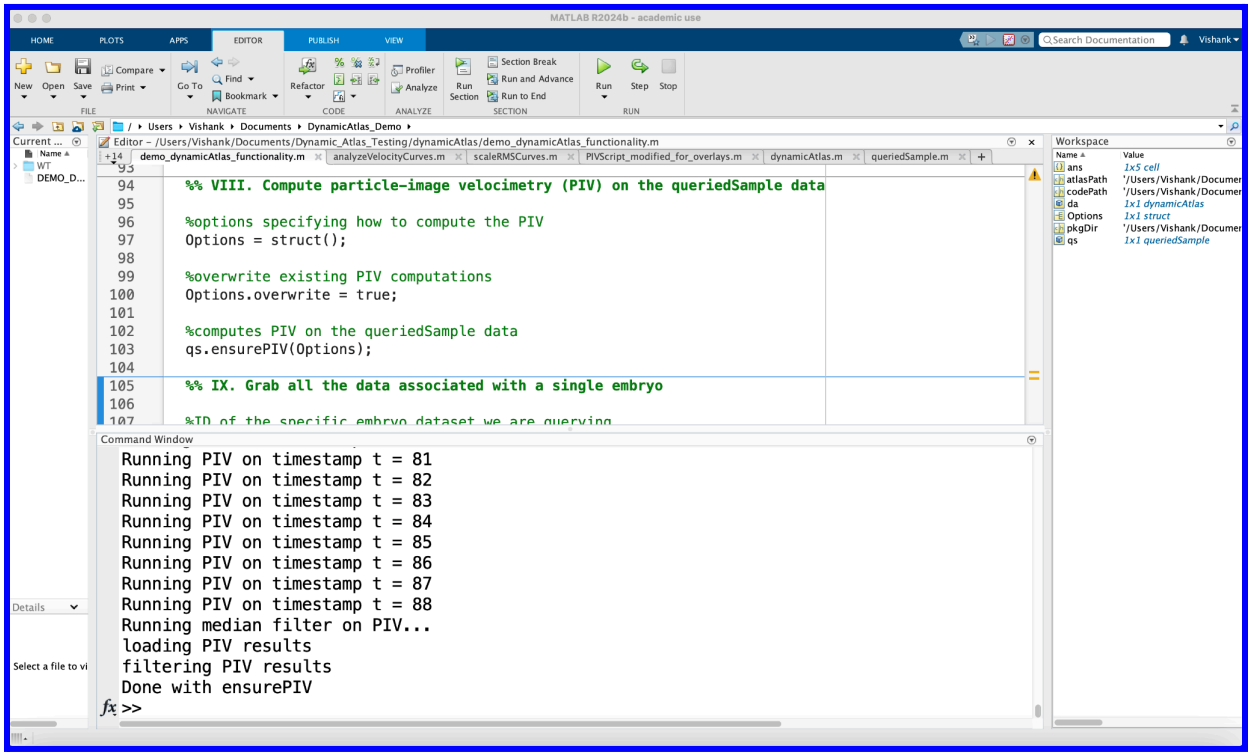
When this code block is run, the data from the live datasets will be loaded in. Windows similar to the following will pop up as the code is running:



These windows show the computations as they are being performed, displaying the x and y components of the velocity respectively in space, here at time frame 18. When the computations are done, two new folders will be generated in each live dataset's folder — PIV and PIV_filtered (displayed opened up below) — as shown:



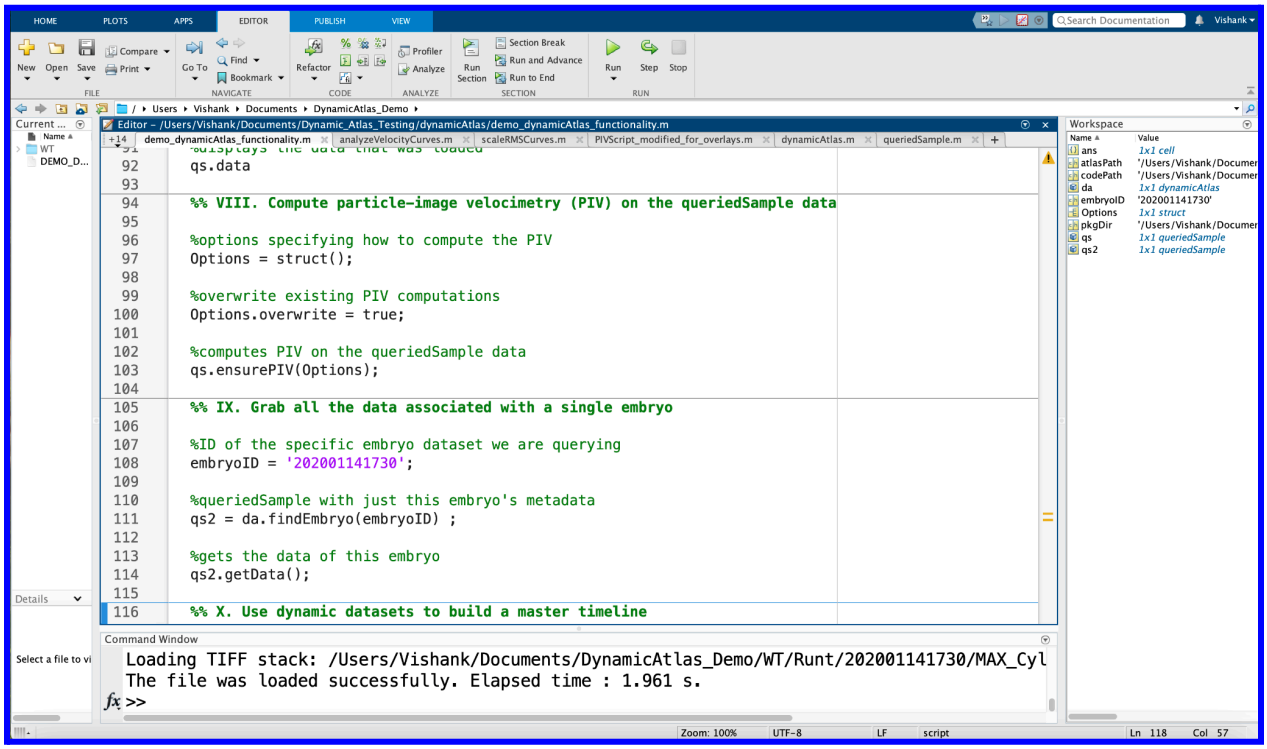
Both contain results of the flow field computation. ‘PIV’ contains the computed flow field, and ‘PIV_filtered’ contains the result of performing a median filter of order 3 on the contents of ‘PIV’, to smooth the flow field in time. Each .mat file contains the velocity components VX and VY at the indicated time frame. The output of the code block in Matlab will look like below:



Code block IX loads in the data associated with a given embryo, by defining a new queriedSample (qs2) with just one embryo (here, ID: 202001141730).

```
%% IX. Grab all the data associated with a single embryo

%ID of the specific embryo dataset we are querying
embryoID = '202001141730';

%queriedSample with just this embryo's metadata
qs2 = da.findEmbryo(embryoID) ;

%gets the data of this embryo
qs2.getData();

```

The data for this embryo will be stored in qs2. The output will look like below:



This method can be used to examine properties of any embryo of interest within the overall dataset.

Demo code walkthrough: Timeline Creation

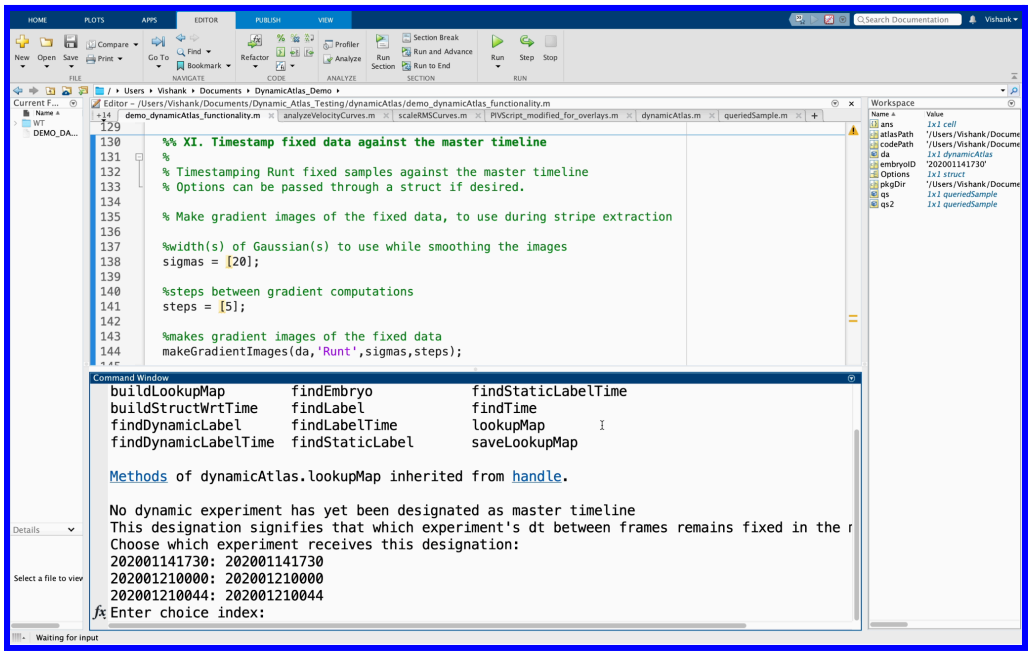
[Block X]

With the data loaded, and defined as above, the morphological timeline creation features of the atlas can now be used. Live datasets are systematically cross-correlated with each other, correspondence curves are found between their timelines, and these timelines are then annealed together to create one consensus timeline. This timeline creation procedure is mostly automated, but a few steps require manual input, and these are explained below. (Note: if the code gets interrupted, this section can just be run again, as the code is written to pick up where it left off when it was last attempted.)

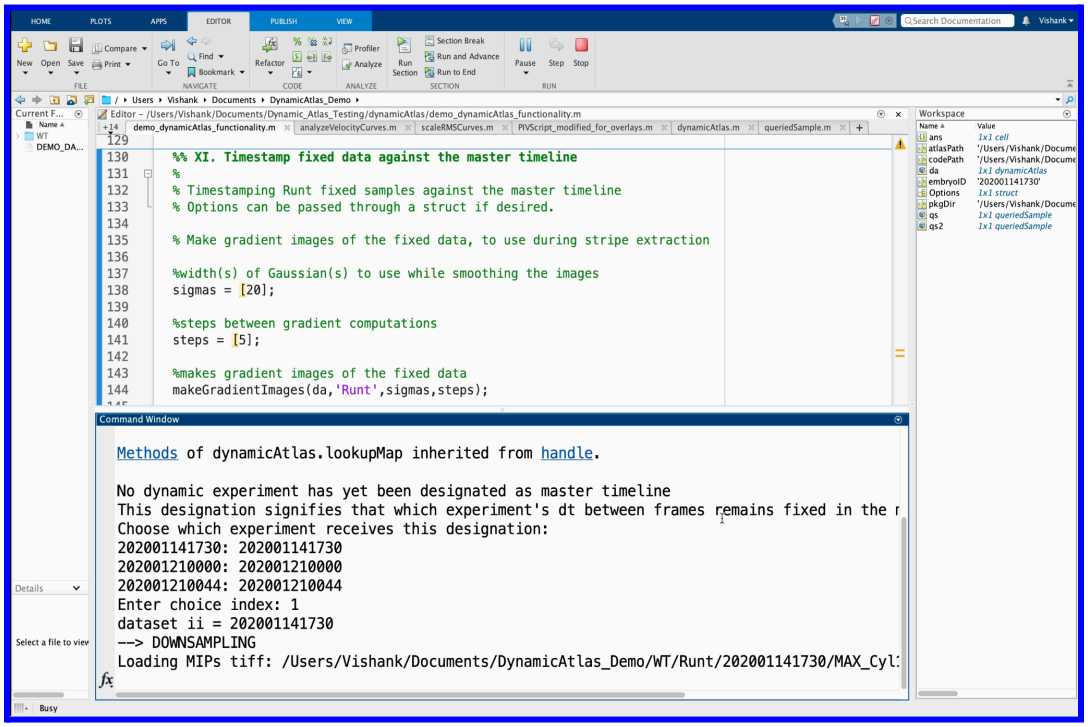
Code block X invokes master timeline creation with options of the user's choosing.

```
%% X. Use dynamic datasets to build a master timeline
%
% Aligning dynamic runt nanobody data against each other

%options specifying how to compute the timeline
Options = struct();

%saving the plots generated while computing the timeline
Options.save_images = 1;

%makes the master timeline from the live WT datasets with the Runt label
da.makeMasterTimeline('WT','Runt', Options)
```

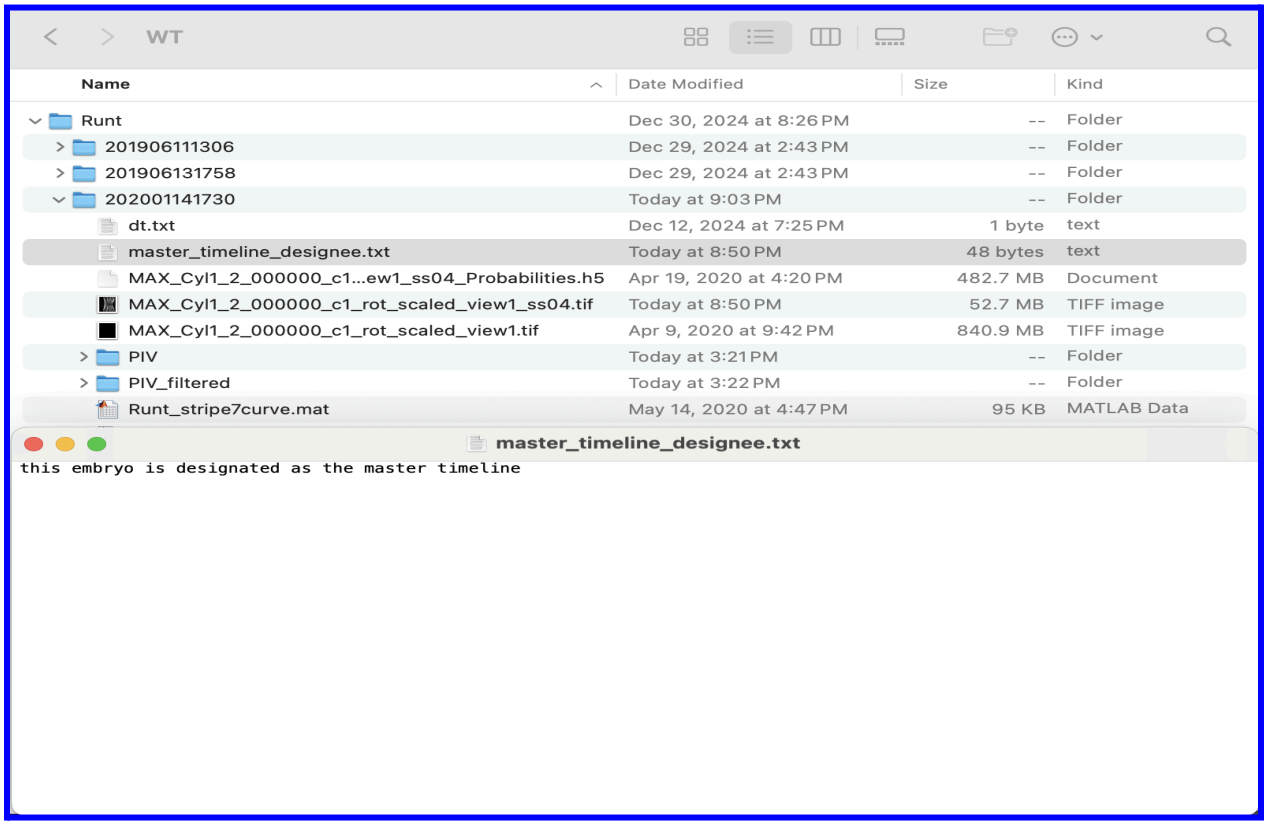
The first step is to choose a live dataset to be the 'master timeline designee'. This will be the dataset which all others are initially compared to for the alignment, but later just becomes an indicator of the equivalence classes of the timeline. The following prompt is displayed in the Matlab command window to do so, by entering a dataset index:



(Note that any choice can be made, but choosing a longer dataset tends to yield better time matching.) If, for example, the first live dataset is chosen (dataset 202001141730 here), the number 1 should be entered, and the following output will result:

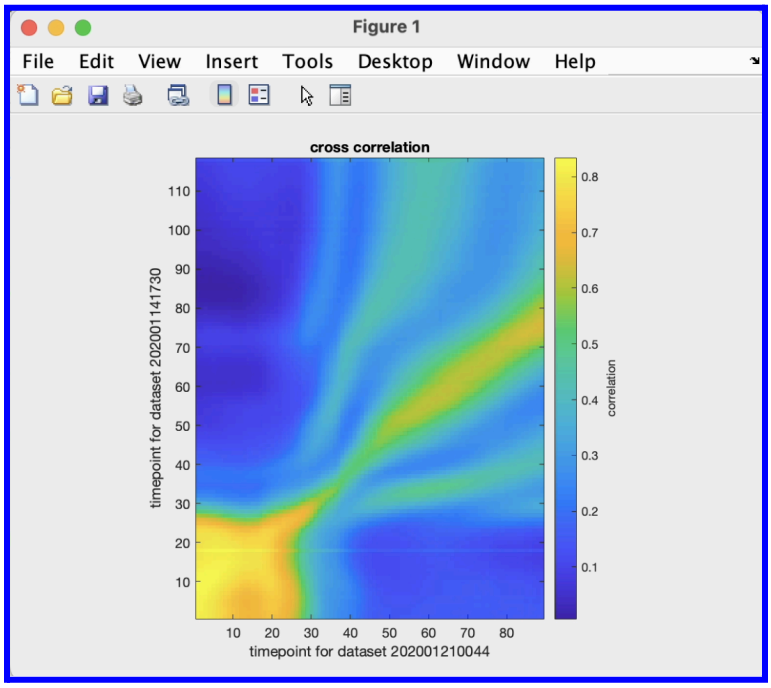


The script will also create a text file, ‘master_timeline_designee.txt’, in the folder of the selected dataset, as shown:



The live datasets will then all be loaded in.

Next, all the live datasets are systematically correlated with each other (cross-correlation), including to themselves (auto-correlation), yielding a correlation matrix for every combination. These correlations are computed using the Pearson correlation coefficient between the images (see Methods). This step may take some time, because the number of comparisons made scales quadratically with the number of live datasets present. After a correlation matrix is computed, it is briefly displayed as a heatmap. Shown is an example of such a heatmap:

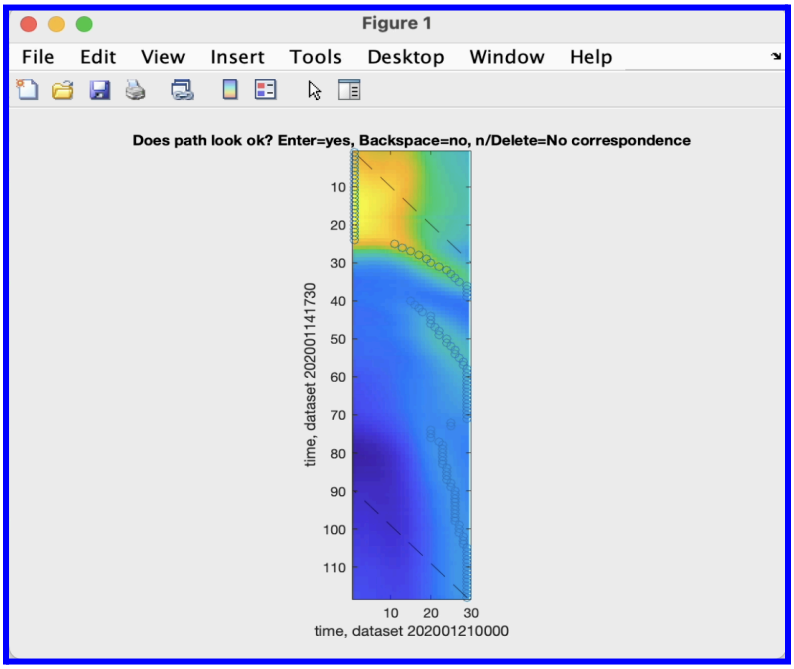


This step is fully automated, and the user does not need to perform any action while these correlation matrices are generated.

The next step is to find optimal correspondence curves through all of these heatmaps (see main text Figure 2). This step requires user input. The program attempts to guess endpoints of the curve based on maximum values of the heatmap at the edges. Users can change these endpoints if they are not correct. Given endpoints, the program attempts first to guess the points which will lie on the correspondence curve by finding high correlation values. If the user indicates these are not accurate, the program then performs the fast-marching algorithm between the endpoints to obtain the curve (see main text Methods). The user can then indicate whether this new result is acceptable, and the path will be stored.

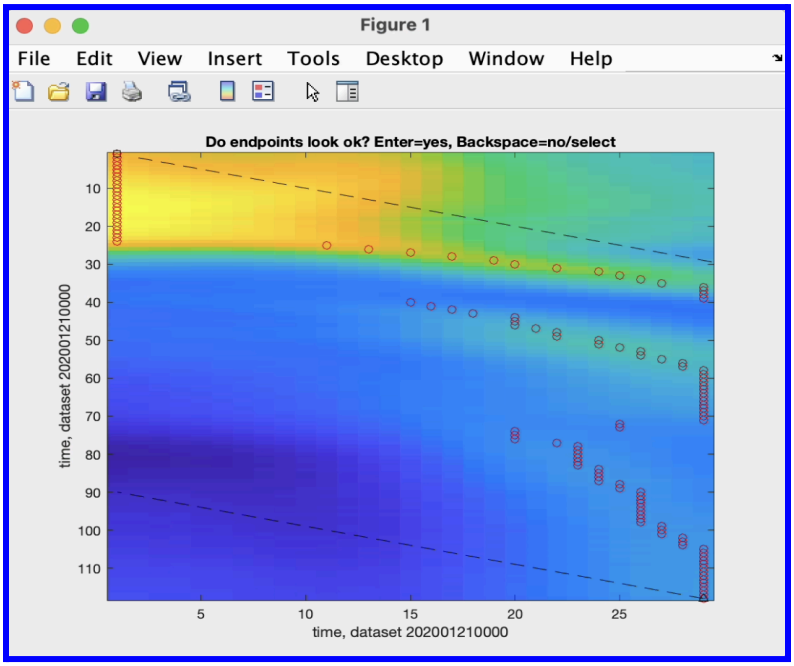
Note that for any prompts indicated on the figures, the user must click on the figure window to interact with them. Else, the user's input will be entered into the Matlab command window, and won't be read in by the interactive GUI.

Below shows the steps described above for an example heatmap. First, the user is prompted (instructions at the top of the heatmap) to check if the path looks ok, based on the guessed points that are displayed:

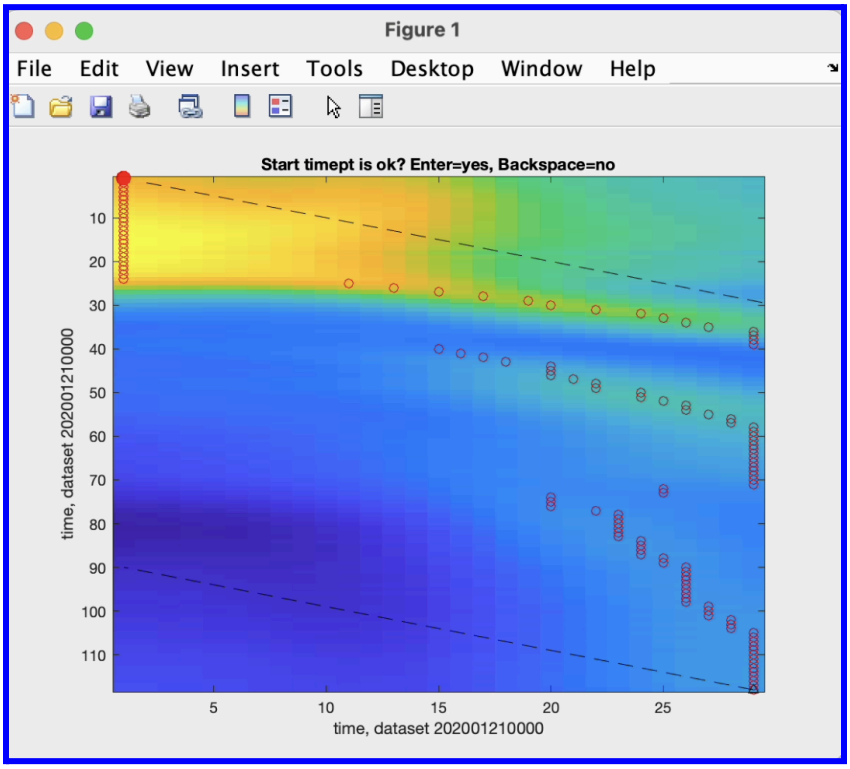


The path should follow a ‘ridge’ of high correlation through the heatmap, and only contain one point per row and column. If yes is selected, the path is accepted, and a new heatmap is shown. If no is selected, the user can then adjust the endpoints of the path and instruct the path to be recomputed. No is selected in this case because there are duplicate points on some columns. (Better paths tend to emerge when No is selected, and the user steps below are performed.)

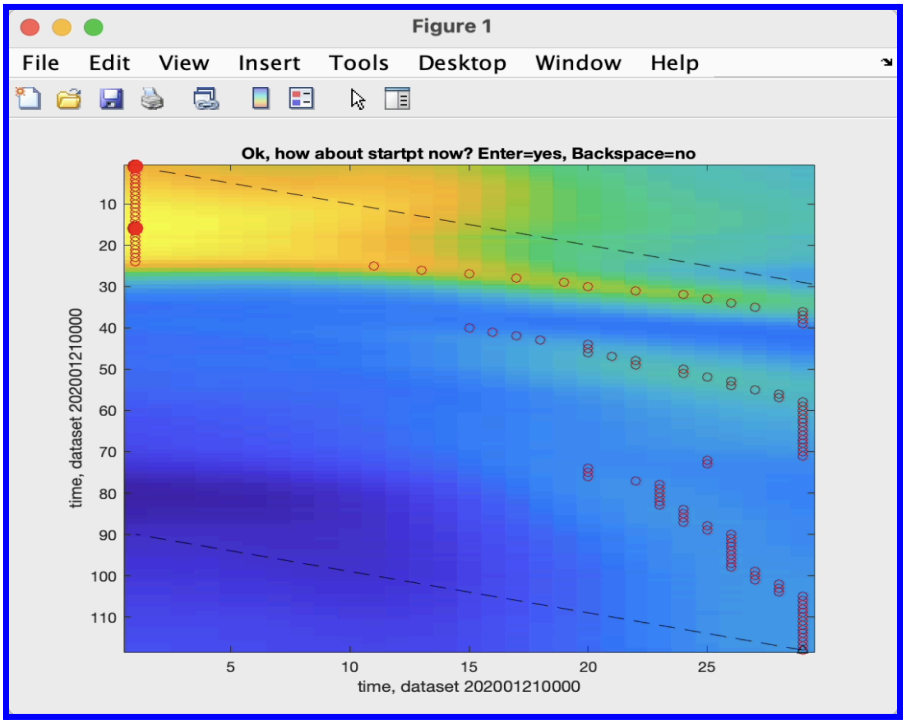
Next, the user is prompted to evaluate the guessed endpoints (slightly darker red):



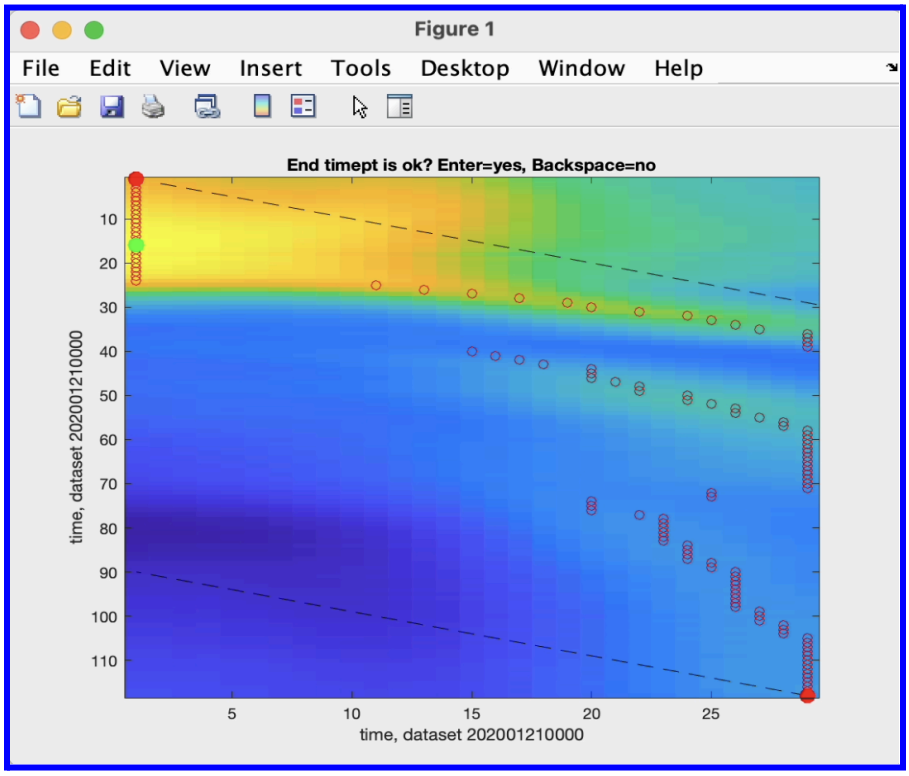
If no is selected, the start point is queried (solid red point):



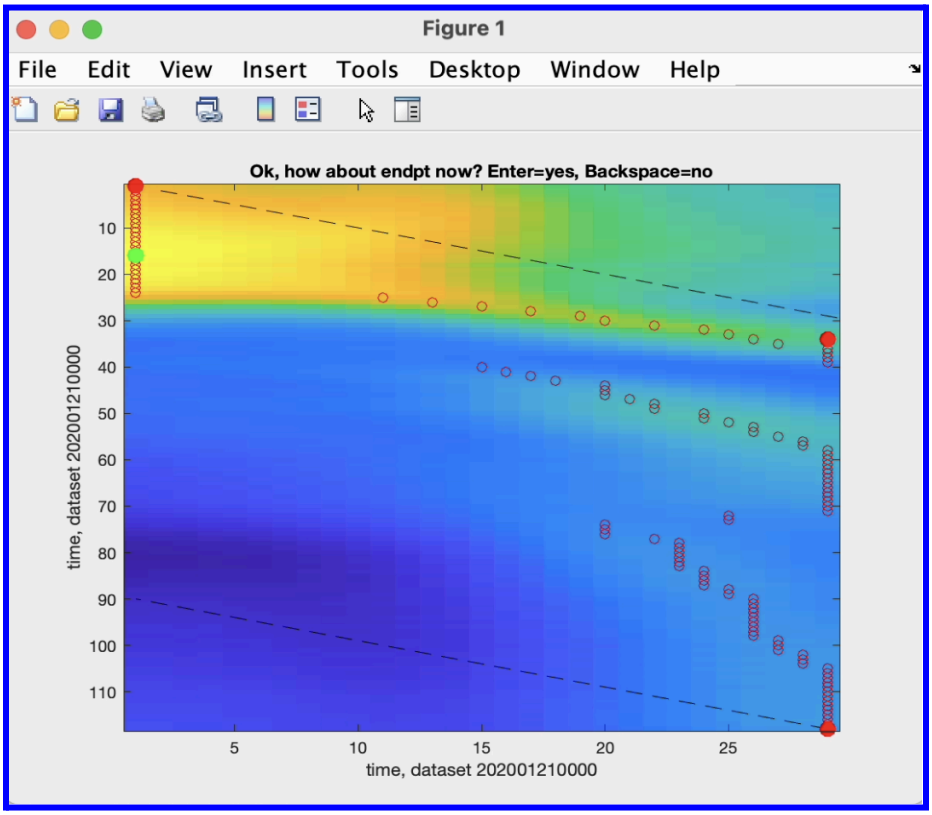
If no is selected, another start point is guessed:



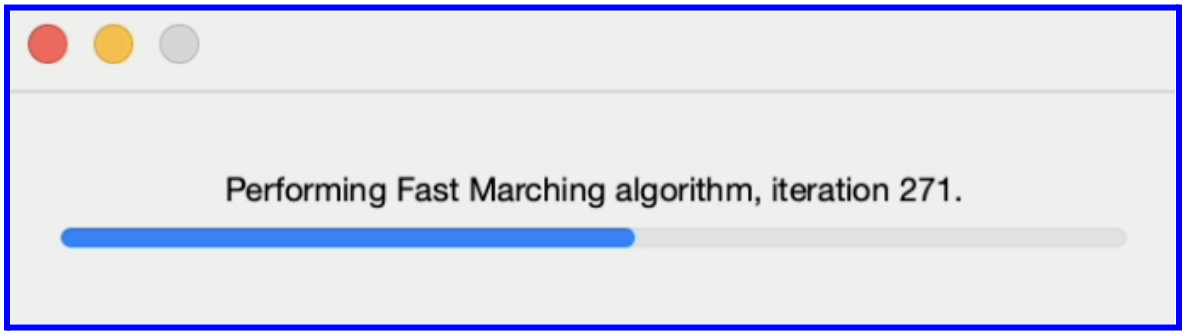
If no is selected, the user will be prompted to click on a point of choosing (not shown here), and the program will find the closest point to the user's input on the boundary of the map. If yes is selected, the start becomes green, and the end timepoint is queried (solid red point):



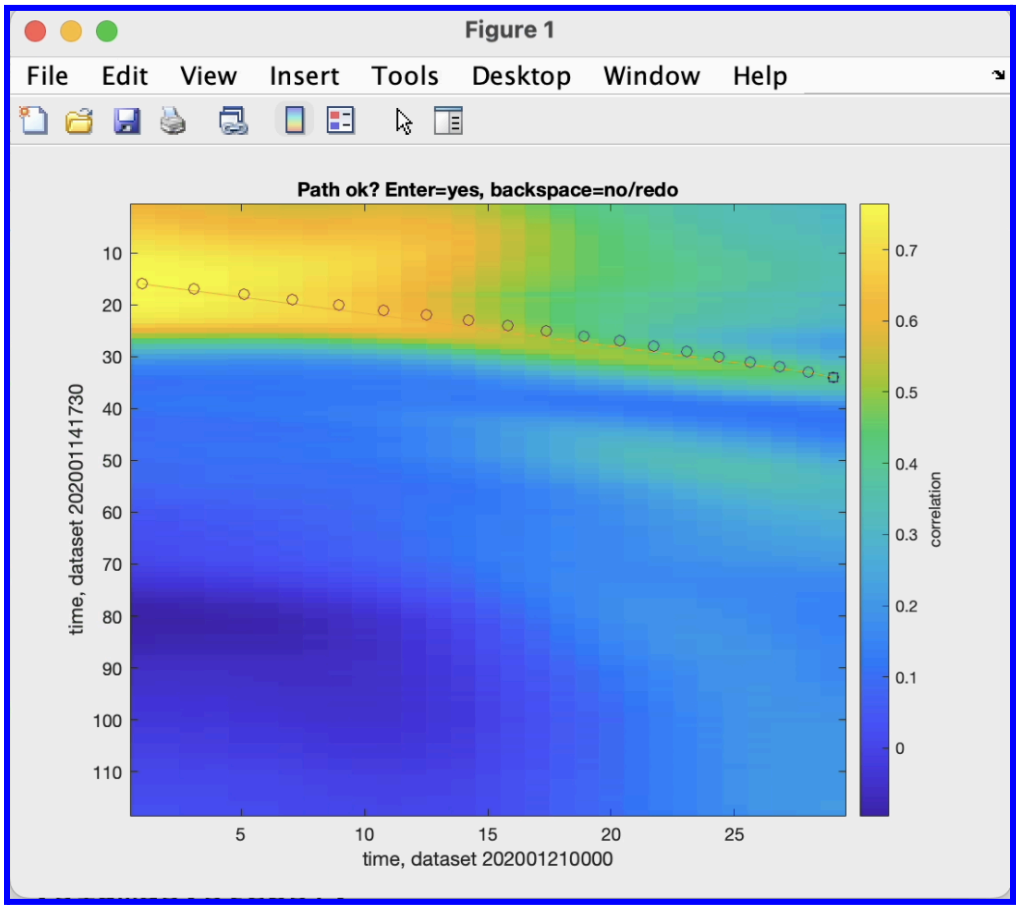
If no is selected, another end point is guessed:



If no is selected, the user will be prompted to click on a point of choosing (not shown here). If yes is selected, both endpoints are now chosen, and the fast-marching algorithm is performed between them, with a window similar to the following popping up during its iterations:

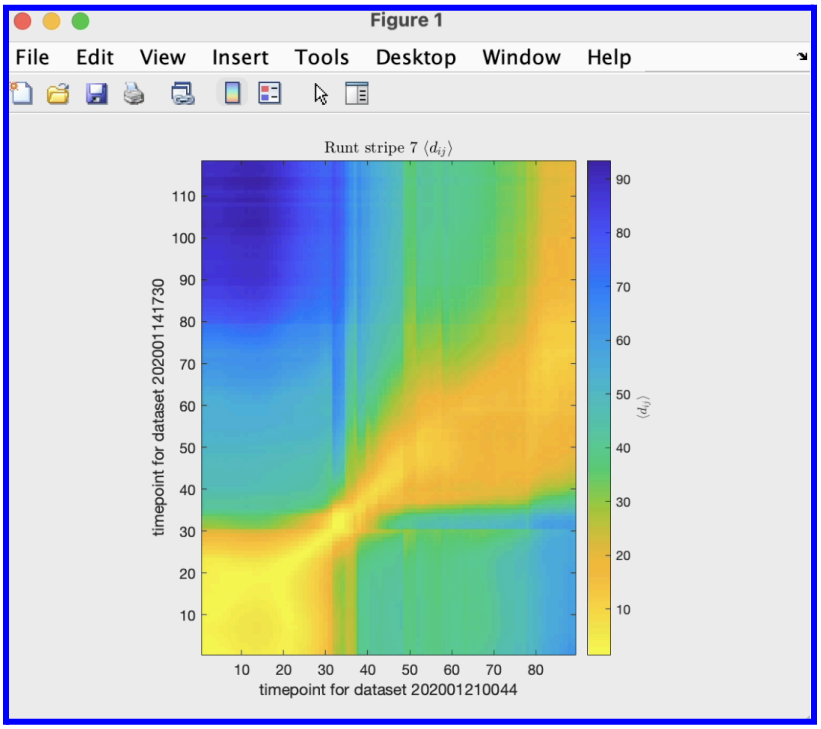


When the algorithm is done, the new guess for the path is displayed by showing sample points on the path (visually, these points should follow a high correlation 'ridge' from end to end):

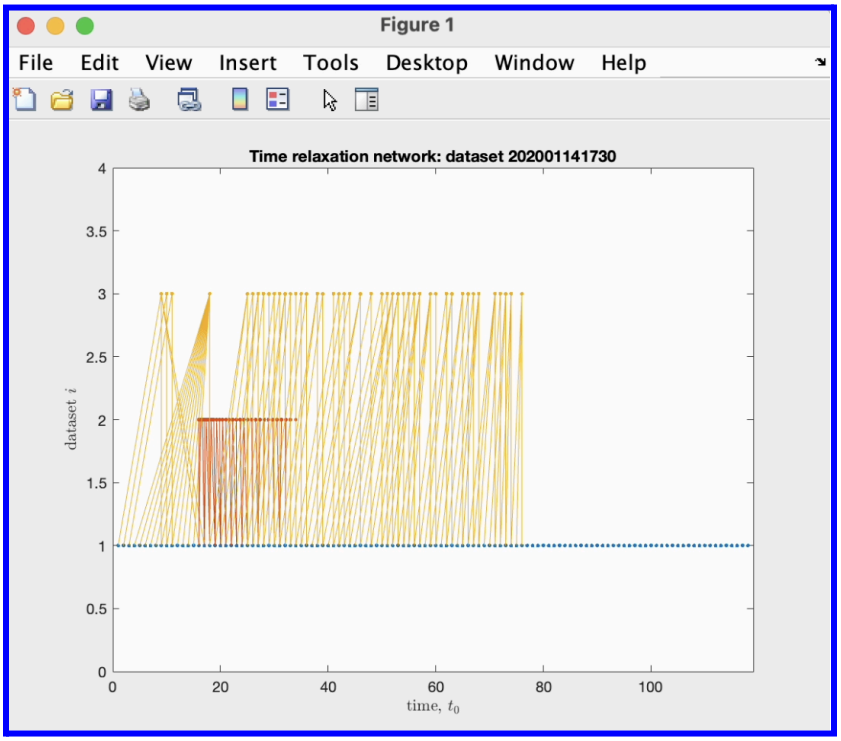


If no is selected, the process can be redone with a new choice of endpoints. If yes is selected, the path is then accepted, a new heatmap is shown, and the above procedures are repeated.

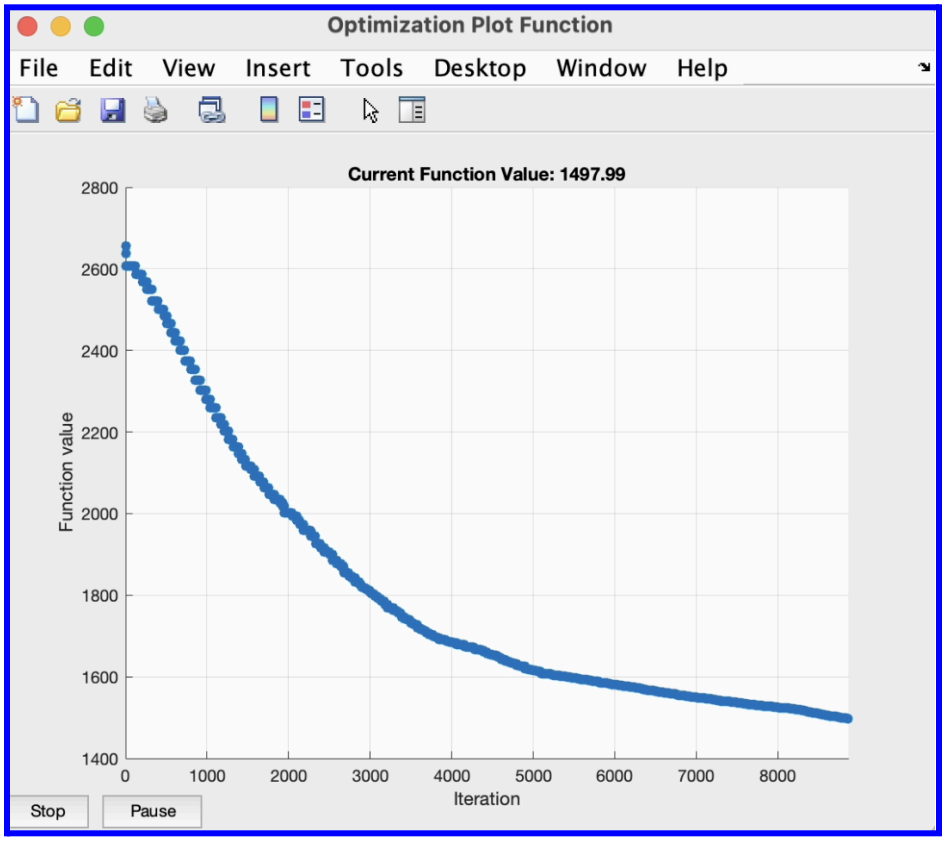
Once the correspondence curves are found for all the heatmaps, the rest of the program follows automatically. Correlations are computed between stripe 7 curves:



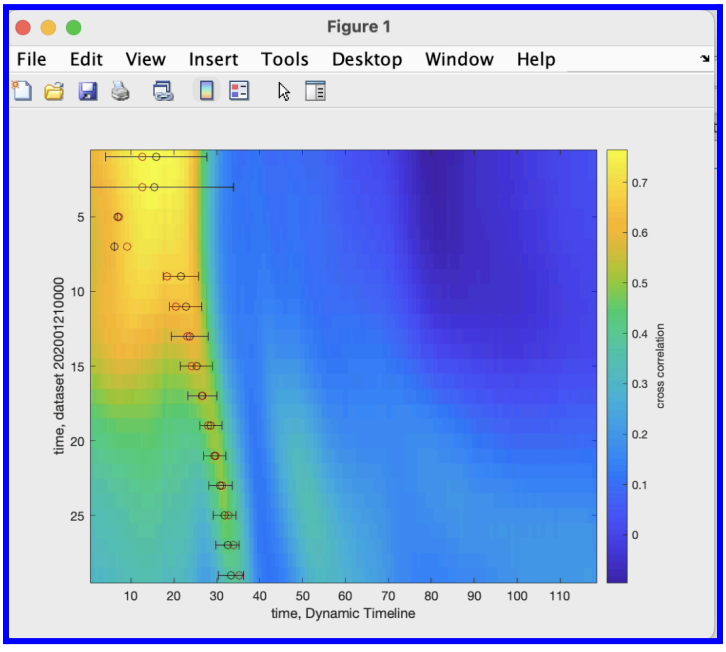
Correspondences are depicted by connecting bonds of a ‘network’ of time nodes between datasets according to the curves:



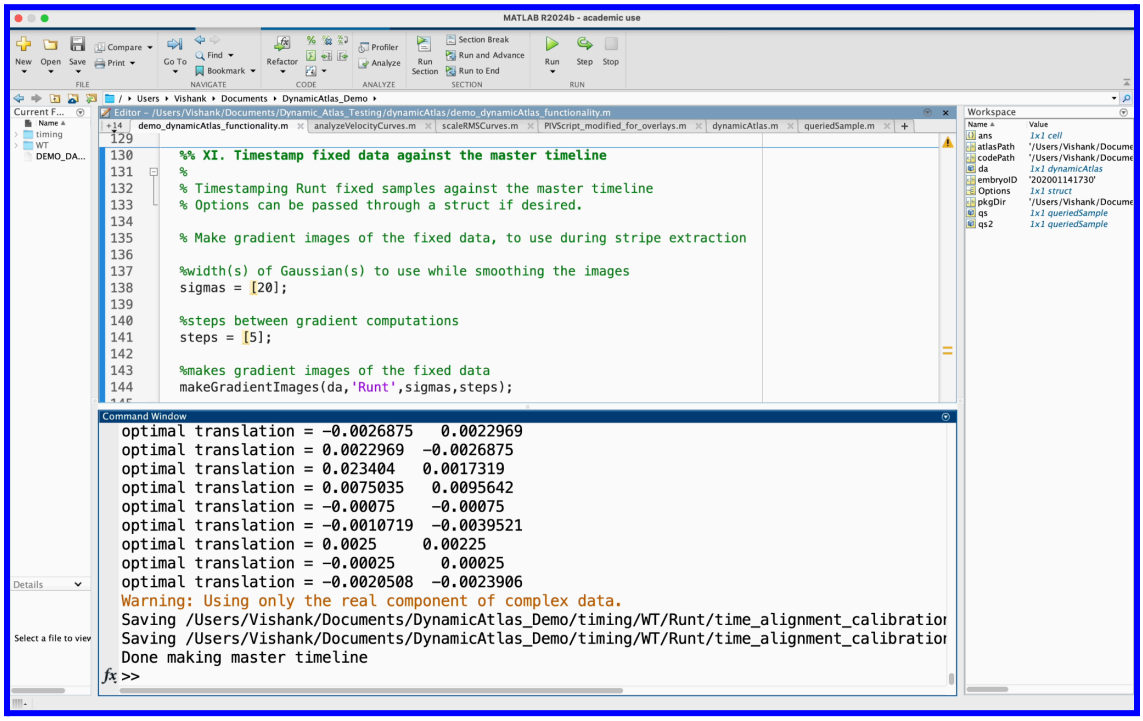
The network as a whole is then annealed together to form an optimal consensus timeline by treating the bonds as springs containing energy, and relaxing the network until an energy minimum is reached (this can take time):



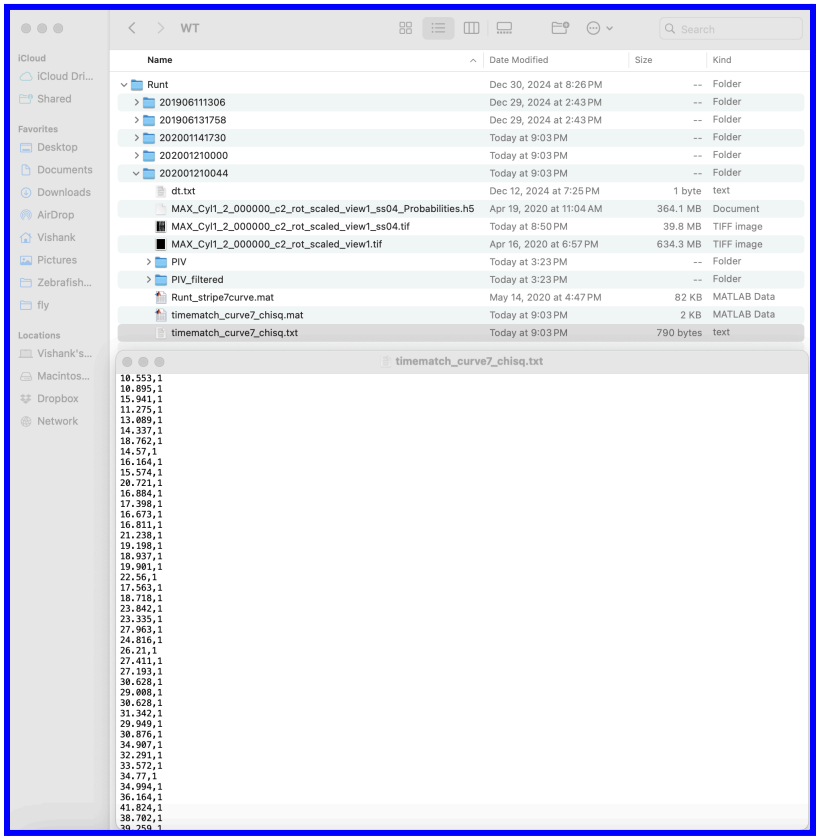
All dataset timestamps can then be represented on the master timeline with an uncertainty given by the chi-squared goodness of fit curve at each point (see main text Methods):



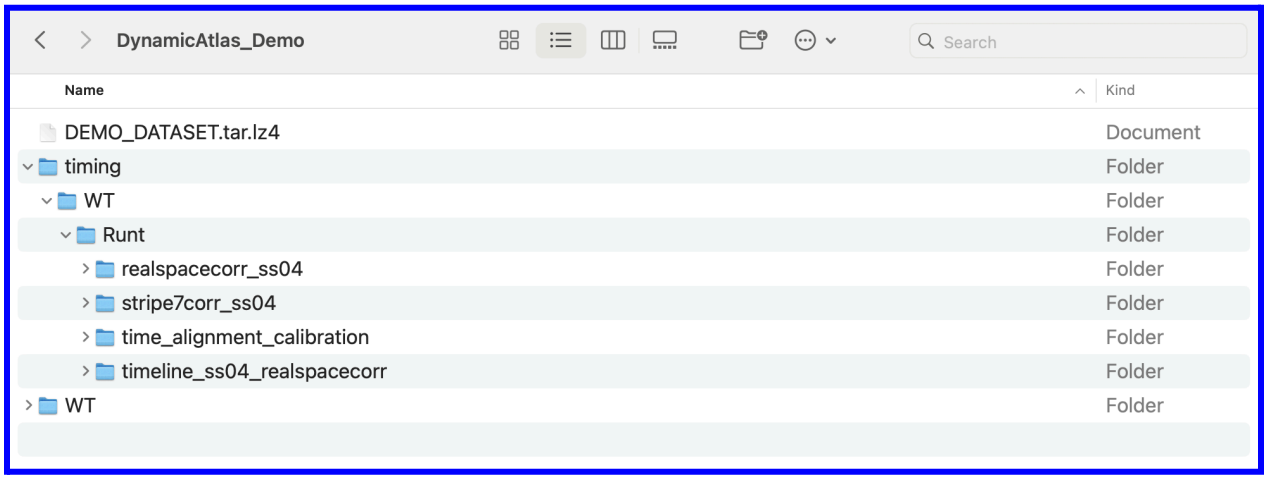
When the timeline creation is done, the code block will show the following output:



The program will save the time matching data to each of the live folders, in the form of both a .txt file and .mat file (same data in both). The data consist of the sequence of timestamps from the live dataset onto the master timeline. The second column indicates the resolution of time stamping (for this data, 1 minute, as this was the interval between frames for all movies):



Additionally, the program will generate a ‘timing’ folder for all timeline-related data and figures, on the same folder level as the genotype folder (WT in this case):



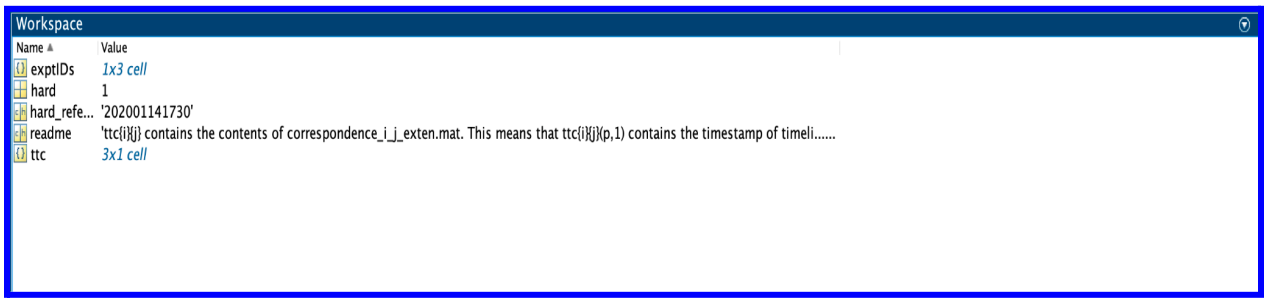
The folder ‘realspacecorr_ss04’ contains data of stripe detections, image correlations, and correspondence curves computed during intermediate steps of the timeline generation.

The folder ‘stripe7corr_ss04’ contains data of the correlations between the stripes.

The folder ‘time_alignment_calibration’ contain snapshots of the timestamping of the individual live datasets to the master timeline.

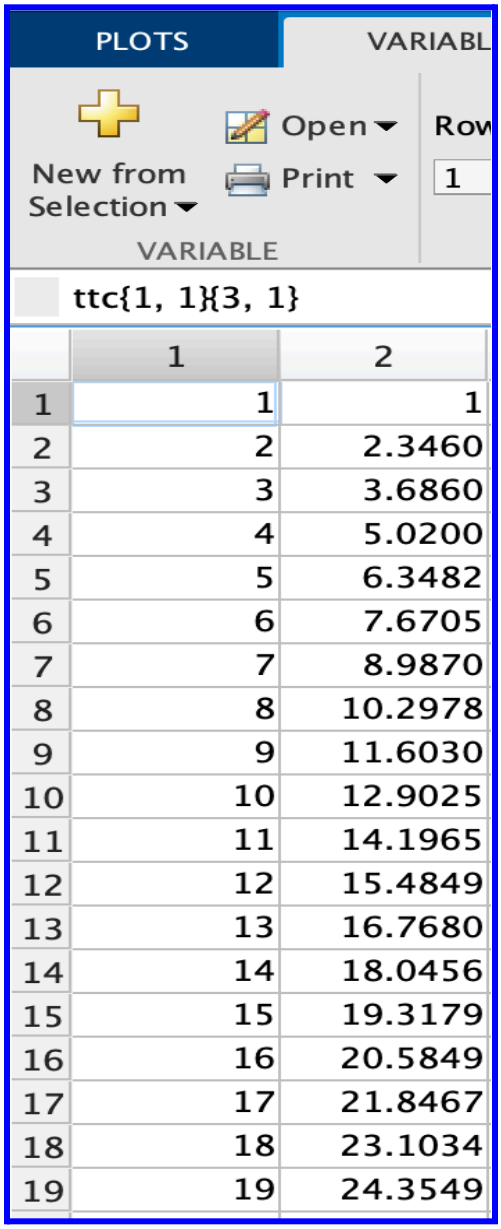
The folder ‘timeline_ss04_realspacecorr’ contains data of the master timeline as a whole, including the network construction, network relaxation, and the final timestamps of the master timeline. The timeline correspondences from live datasets to the master timeline can be visually observed in ‘time_correspondences.png’’.

Importantly, within this folder, the full master timeline data is stored within the variable ‘ttc.mat’, which contains all the explicit time correspondences between all datasets to each other. Opening this variable yields:



The variable exptIDs contains the experiment IDs, and their indices are given by their position in this array. The readme explains the convention used for comparisons, and ttc contains the

correspondence data. These consist of two-column arrays, with the left column representing the time frames of a selected embryo, and the right column representing the corresponding times in another embryo. An example of an array in ttc is shown below, which represents the correspondence between time frames in embryo 1 and equivalent times in embryo 3:



	1	2
1	1	1
2	2	2.3460
3	3	3.6860
4	4	5.0200
5	5	6.3482
6	6	7.6705
7	7	8.9870
8	8	10.2978
9	9	11.6030
10	10	12.9025
11	11	14.1965
12	12	15.4849
13	13	16.7680
14	14	18.0456
15	15	19.3179
16	16	20.5849
17	17	21.8467
18	18	23.1034
19	19	24.3549

From the saved information mentioned above, the master timeline for this series of live datasets can now be directly accessed and queried for use in other computations of choice. For example, a timeline was previously generated using this method from live data contained within the full atlas, and this is the source of the time matching data that one can query using the Python interface referenced in the main text and the Supplementary Information.

Demo code walkthrough: Fixed-data Timestamping

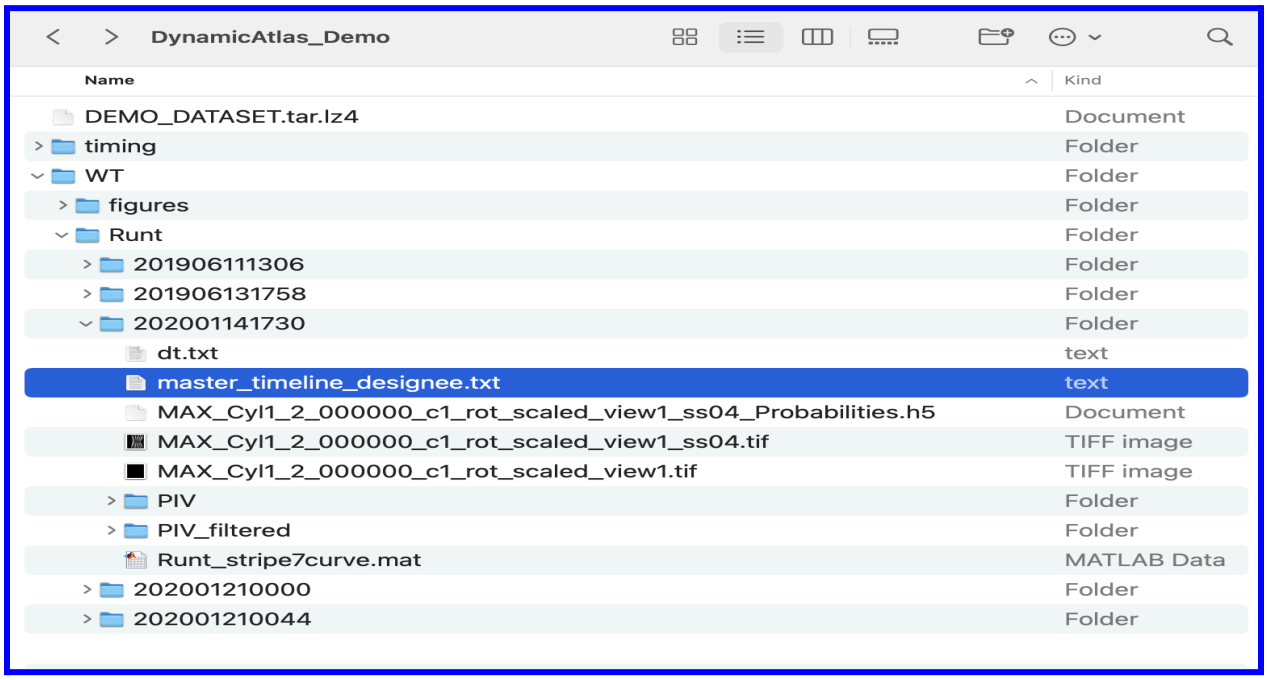
[Block XI]

The final step of atlas timeline creation is to timestamp a fixed dataset onto the live master timeline (see main text Figure 2). This code block performs the fixed timestamping for the fixed datasets within the ensemble.

Code block XI timestamps all the fixed Runt samples to the live-generated master timeline. (Note that in general, the fixed samples will include data from embryos that have been co-stained for another gene in addition to Runt, so this enables timestamping of other gene expression patterns by proxy – see main text Figure 2).

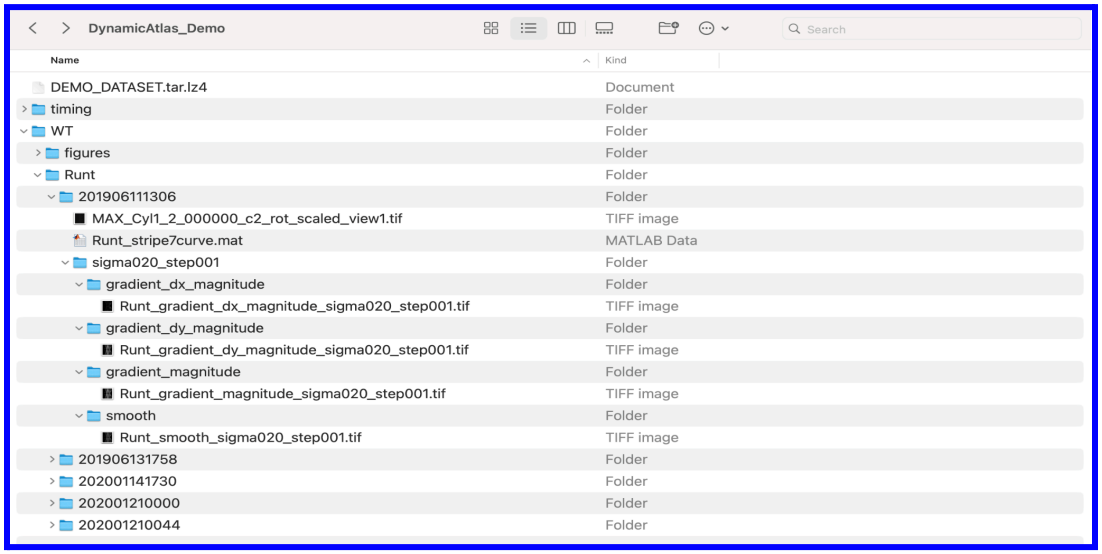
```
%% XI. Timestamp fixed data against the master timeline
%
% Timestamping Runt fixed samples against the master timeline
% Options can be passed through a struct if desired.

%path to the folder of the embryo chosen as the master timeline designee
%(done earlier in the timeline creation block X, the folder will have
% 'master_timeline_designee.txt' within it)
masterDesigneeDir = '/Users/Vishank/Documents/DynamicAtlas_Demo/WT/Runt/202001141730';

%width(s) of Gaussian(s) to use while smoothing the images
sigmas = [20];
%steps used by the gradient during the computation
steps = [1];
%specifying to compute gradients only on the fixed datasets
%(only fixed are timestamped, so only necessary to compute these here)
fixedOnly = 1;

%makes gradient images of the fixed data to use in the alignment
makeGradientImages(da,'Runt',sigmas,steps,fixedOnly);

%options specifying how to timestamp the fixed samples
Options = struct();
%passes in the directory of the master timeline designee
Options.masterDesigneeDir = masterDesigneeDir;
%indicating that stripe information should be loaded in from the .mat
%variable stored in the folder
Options.loadStripeMat = 1;

%timestamps fixed samples with the Runt label to the master timeline
da.timeStamp('WT', 'Runt', Options)

disp('Demo done.')
```

Note that the user must provide an input here: the path to the directory of the dataset previously chosen by the user as the master timeline designee in code block X (in this case, dataset ID: 202001141730). This folder is easy to verify by checking if it contains the ‘master_timeline_designee.txt’ file within it, as shown below:

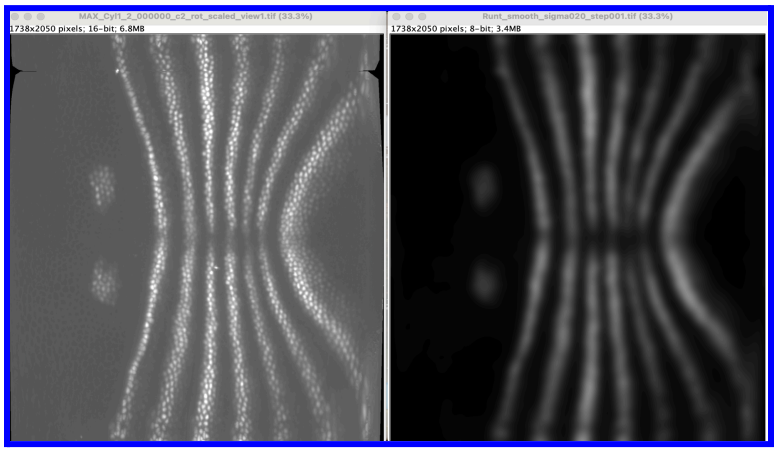


The rest of the variables in this code block do not need to be changed.

The first step of the code block is to create smoothed gradient images of the fixed datasets to be used to perform the time stamping. This is done with the function makeGradientImages, which will process the fixed datasets (here, ID: 201906111306 and 201906131758) using the sigma contained in 'sigmas' for the Gaussian and the gradient step size contained in 'steps'. When the code through makeGradientImages is run, gradient images will be saved in the folder of each fixed dataset, as shown for the fixed dataset 201906111306 and its folder 'sigma020_step001' below:

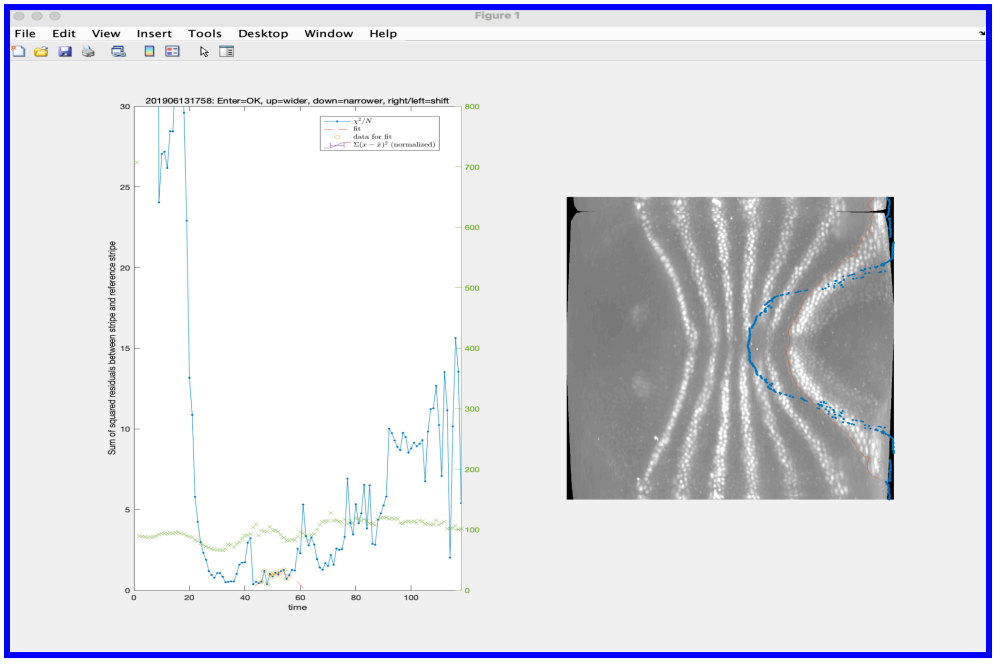


Below is a side by side comparison of the original data (MAX_Cyl1_2_000000_c2_rot_scaled_view1.tif) and the smoothed gradient image (Runt_smooth_sigma020_step001.tif) when opened in Fiji:

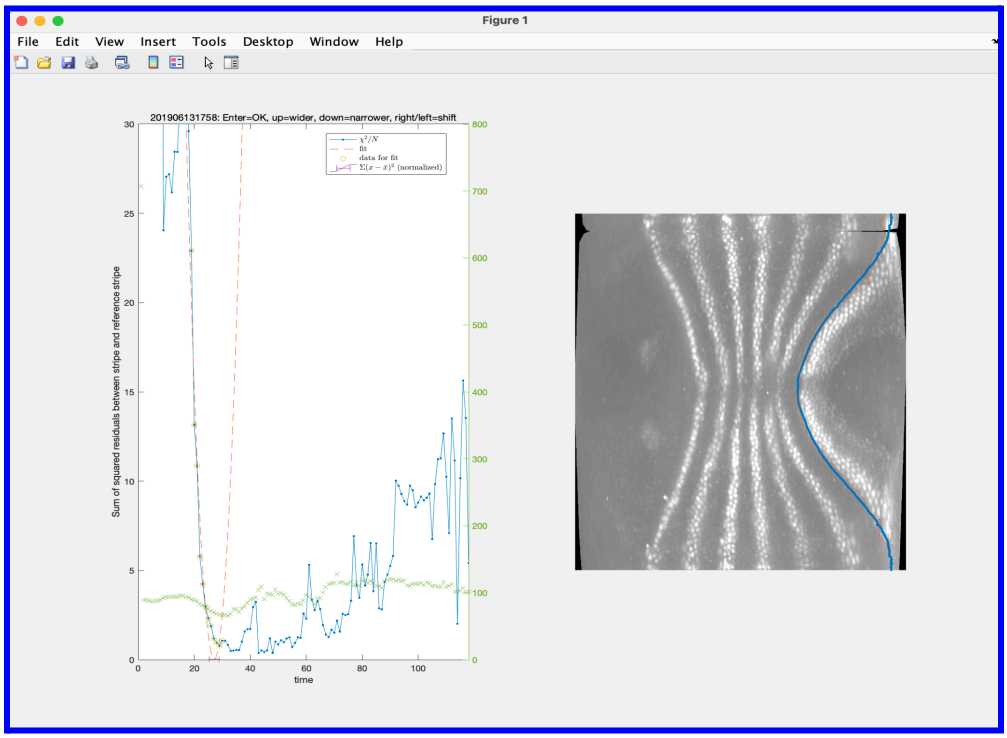


The smoothed images generated from this code are more amenable for comparison to the continuous reference timeline.

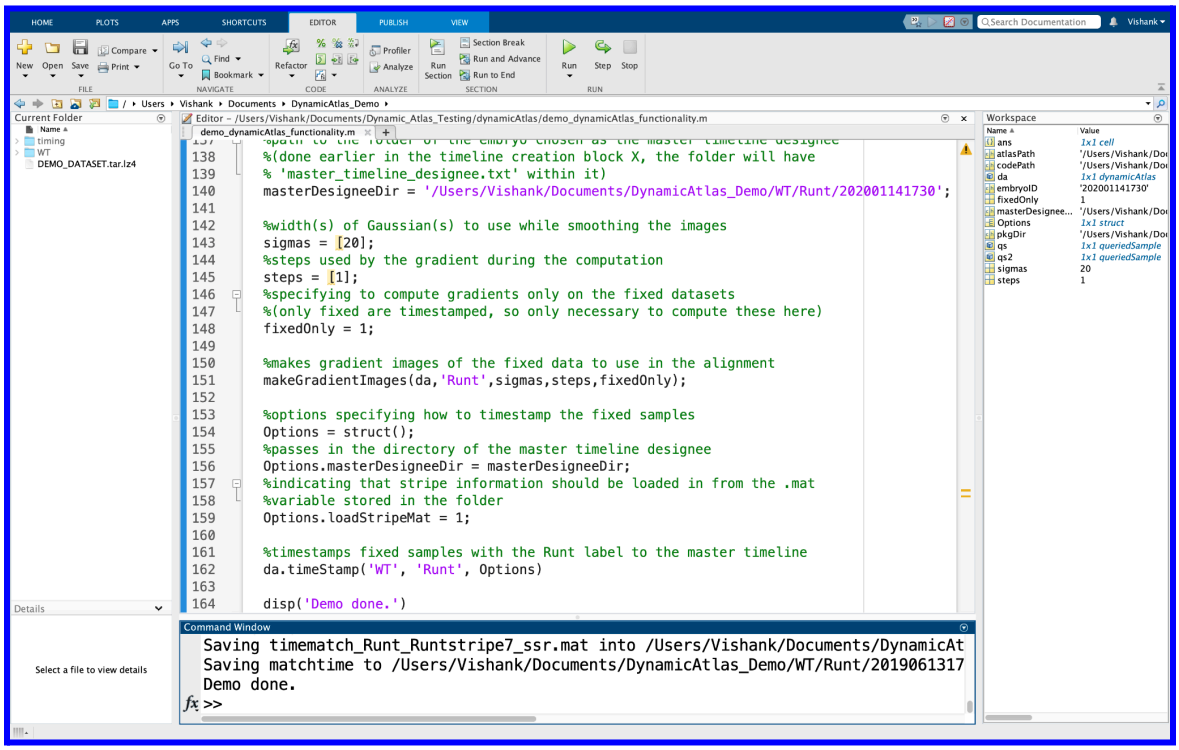
The remaining code performs the timestamping. Once run, a series of automatic computations will take place, then for each fixed dataset, a GUI window will pop up for the user to manually adjust the reference timeline coordinate to best fit the current dataset, by minimizing stripe residuals (left plot). The program provides a visual aid for this process (right plot), by displaying the dataset stripe (in orange) and the currently-selected reference timeline stripe (in blue). The instructions at the top of the left plot describe how to adjust the fit, in both position and width. The program will provide an initial guess for a timestamp, but typically the user will have to adjust this further. Below shows the initial window for dataset 201906131758:



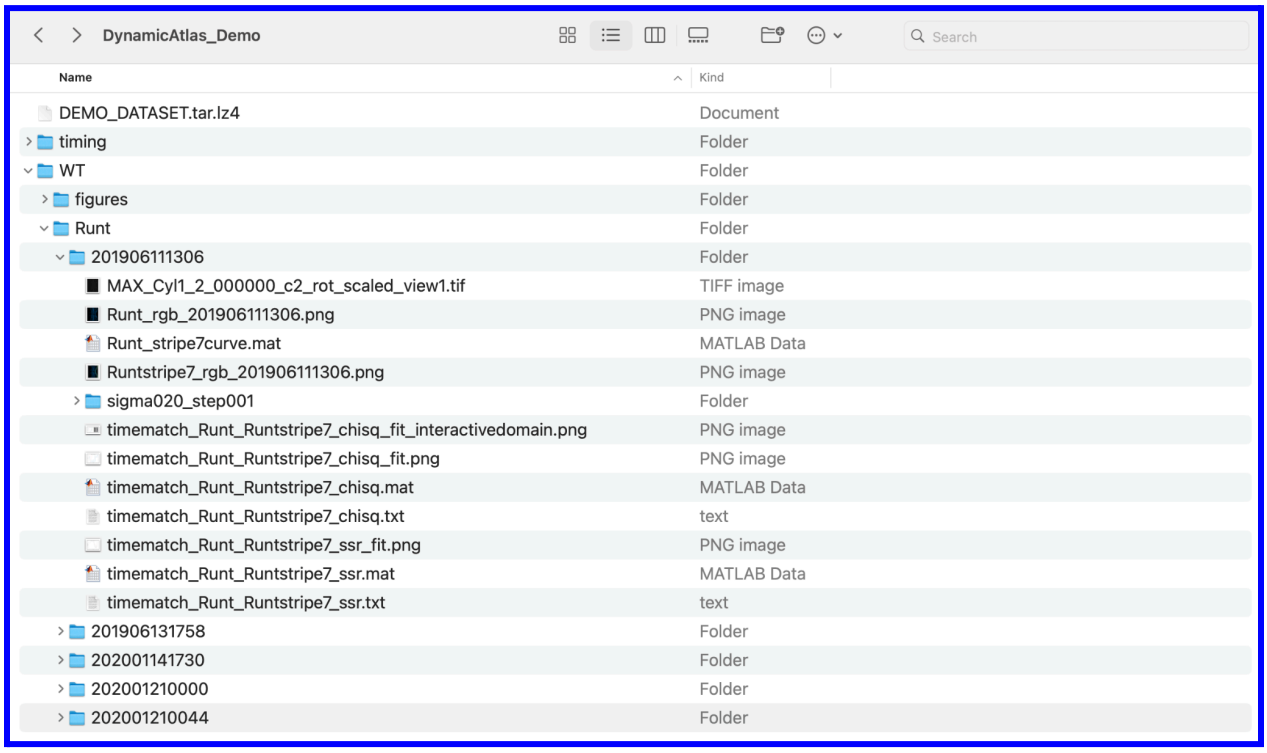
And an improved fit after user-inputted adjustments:



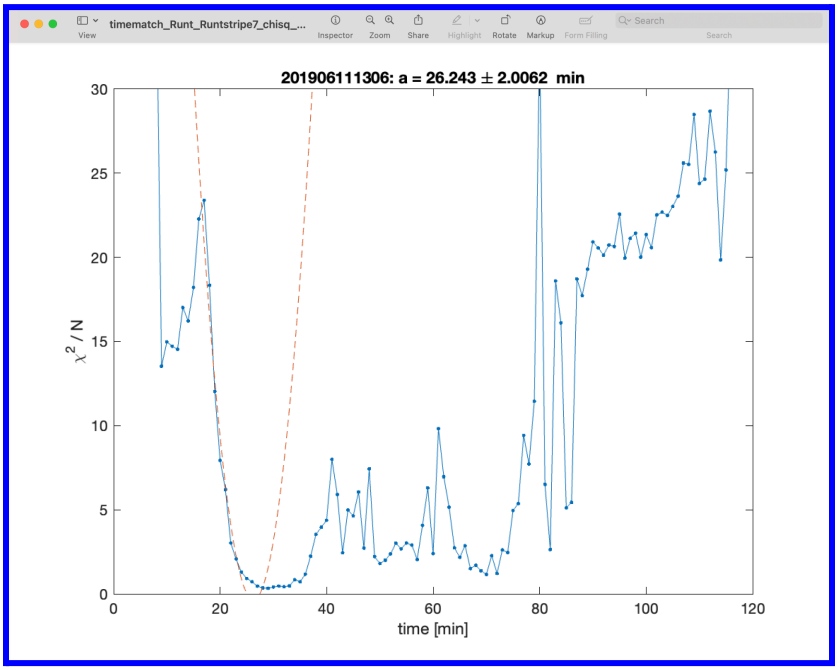
Pressing ‘Enter’ accepts the fit once it is completed. This process will be carried out for each fixed dataset. Once completed, further automatic computations will occur, and the resulting timestamps will be saved to disk. The code block, and the demo script as a whole, will then be complete, showing the following output:



After this step is complete, the time matching results will be saved in each of the individual dataset folders. For example, for 201906111306, the following files will now be contained within:



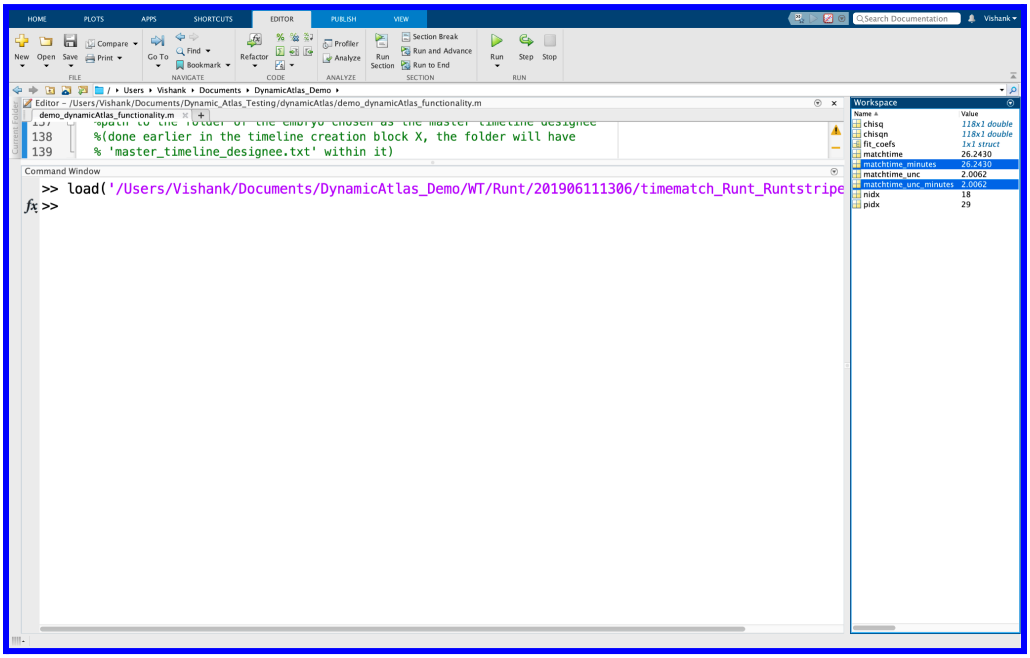
The time matching results, using the respective methods of minimizing chi-squared and sum-squared-residuals between stripes, are saved in the files prefaced by ‘timematch’. Specifically, the values stored are the timestamp, and the 1 sigma (68% CI) uncertainty. Shown below is the image ‘timematch_Runt_Runtstripe7_chisq_fit.png’:



The timestamp and the fit uncertainty values are shown at the top. These are also stored in the corresponding .txt file, and .mat file (with more detail), as shown below:

.txt file:

.mat file:



—

The demo code is now finished. All quantitative computations, and their corresponding visualizations, are stored as additional files within the existing dataset folders, or as files in the new ‘timing’ folder created during master timeline generation. The methods demonstrated in this demo are general, and other Atlas data on the Dryad repository, as well as new datasets (as long as arranged using the same conventions) can be processed similarly: to create an atlas, to create an ensemble timeline, and to timestamp fixed datasets onto the timeline.

—