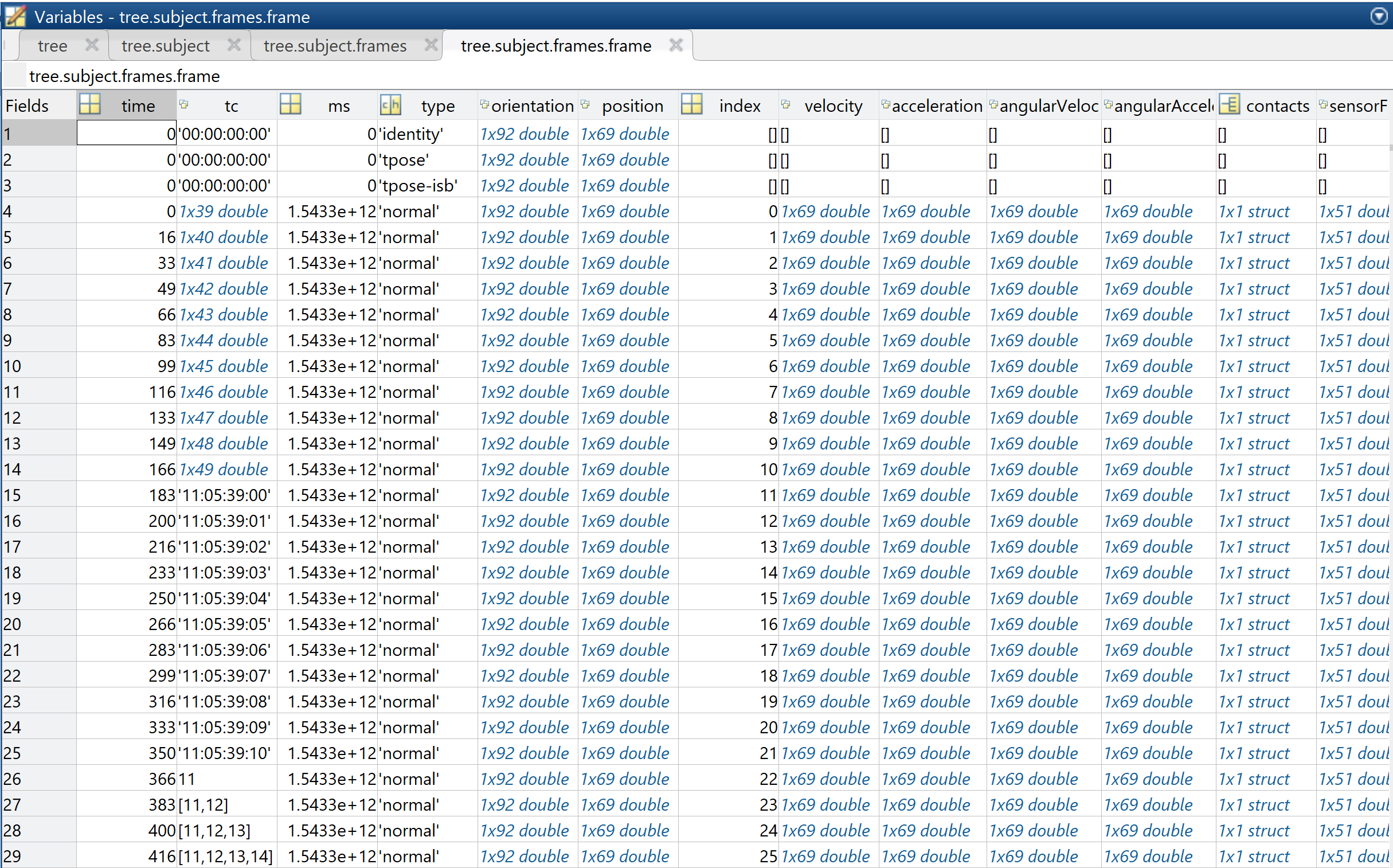
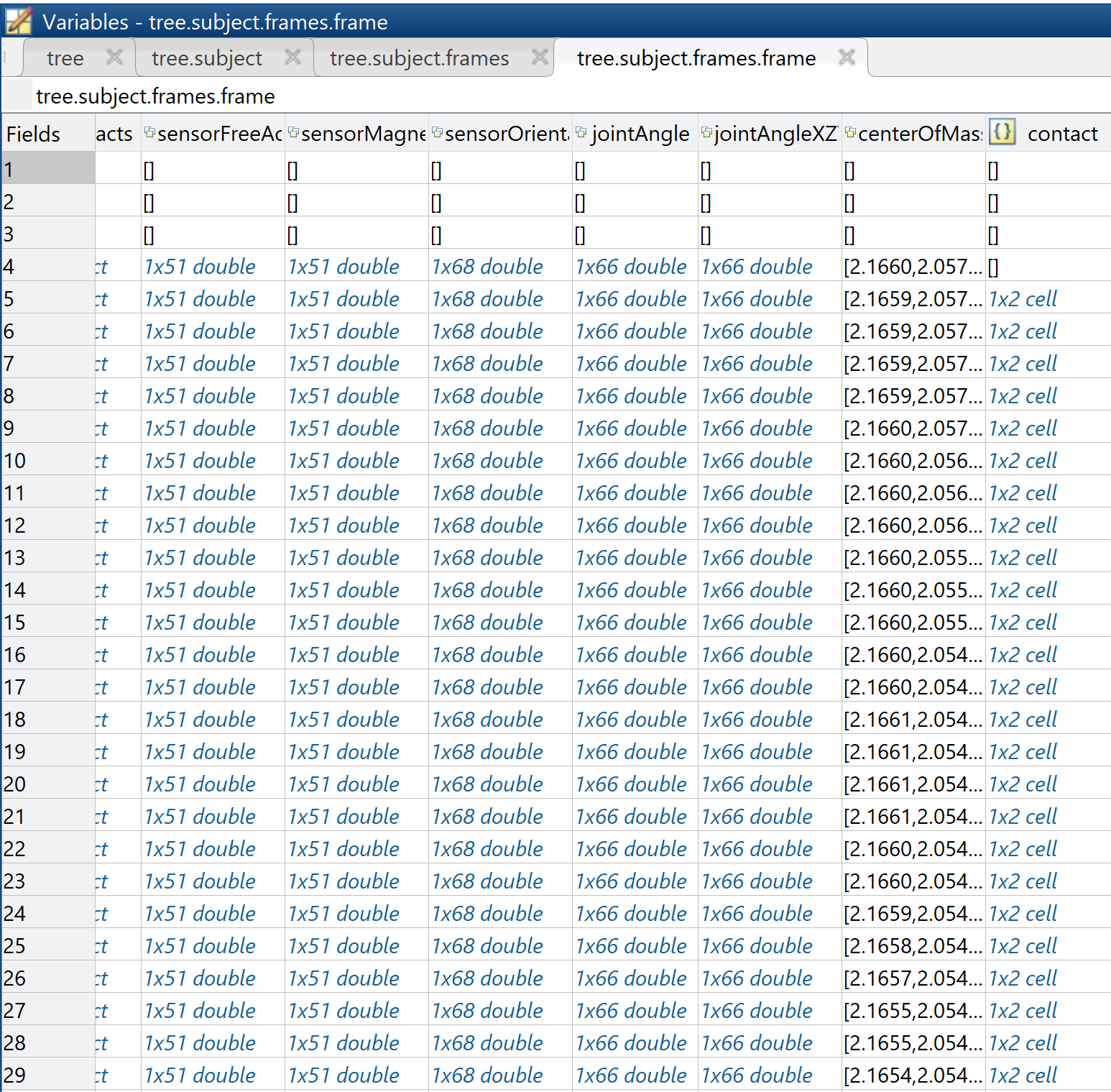
**Data Forms and Types**

**Background**

Before moving onto how the data is actually used to train and test an RNN model, it’s worth outlining the forms the data can take, what it actually represents, and how we treat these forms differently. This is necessary so that the RNN model itself is able to treat these data files in the same way (though giving noticeably different results depending on the nature of the data file); this avoids the problem of having to create a unique RNN script for every different type of measurement (‘input type’) and model type we are targeting (‘output type’).

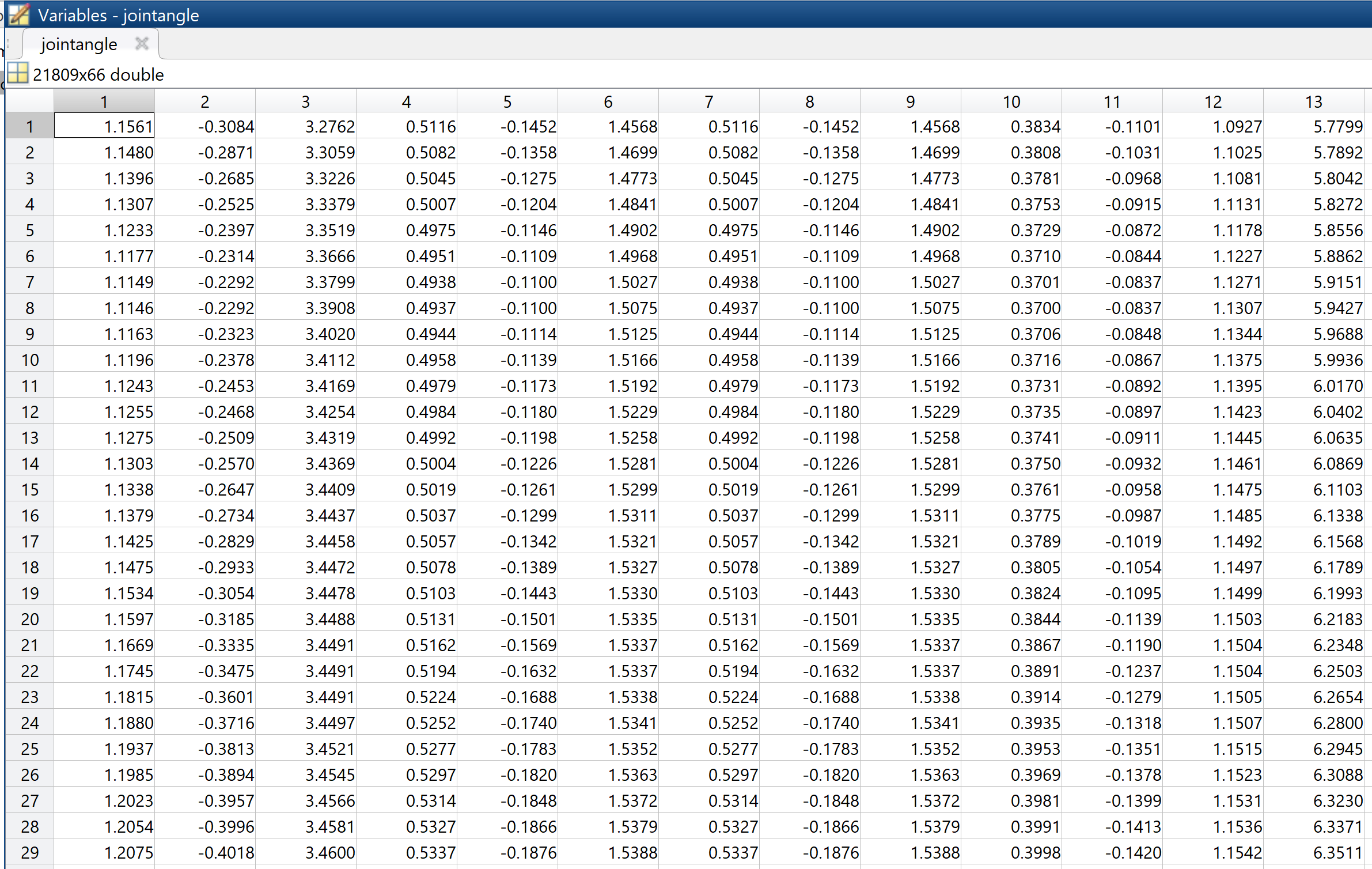
**The Source ‘.mat’ Files**

All the data that we are concerned with is captured by subjects wearing the Xsens MVN inertial motion capture system (i.e. the body suit). Further information on how this suit works can be found in the “MVN User Manual.pdf” file in the base ‘project directory’. Each single file have is captured by one instance of a single subject wearing the suit (i.e. one subject’s visit to the hospital). The files themselves are stored as ‘.mat’ files within a tree structure containing the data measurements themselves, among other metadata including segment, sensor, and joint names, the time and date the data was captured, and other relevant metadata. To access this information we are concerned with (i.e. non-metadata), we can first open the ‘.mat’ file in MATLAB before navigating to ‘tree.subject.frames.frame’. An example of what the data values within a ‘.mat’ file look like at this point can be seen below as the aforementioned ‘data values’ within the ‘.mat’ file:

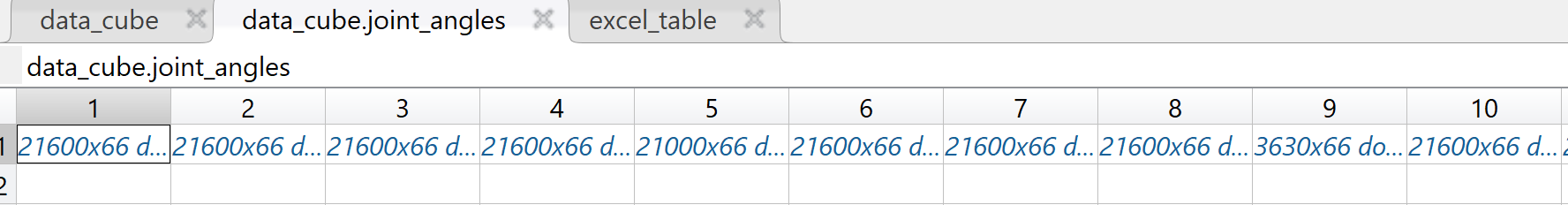
Note that each row is a single sample (what we call a ‘frame’) captured by the bodysuit and 60 of these rows correspond to the data captured over 1 second (as the sensors sample at 60 Hz). The columns, meanwhile, primarily consist of measurements that are captured by the suit’s 17 inbuild sensors. Note that there are different aspects that are measured by the suit depending on the measurement; for example, for ‘position’ the sensors measure the positions of 23 segments on the body suit in 3D space which, given that it’s measuring in 3 dimensions, correspond to 69 values for a given time instance (i.e. a single ‘row’ of data), while ‘jointAngle’ measures using the 22 joints of the suit, which gives 66 total values. The result is that, over a single time instance of 1/60th of a second, approximately 739 distinct values are captured by the suit. This is what we mean when we refer to an ‘**all data**’ file (or ‘**AD’ file** for short): it contains all the possible data captured by the suit for an instance of the subject wearing the suit.

In contrast to this, we are also provided with ‘.mat’ files that correspond to the same subject’s instance of wearing the suit but that only contain the joint angle measurements. The idea behind this is that, as it’s believed that joint angles will be an important measurement for model training in several scenarios (classification of file type, regression of overall NSAA score, etc.), the ‘**joint angle’** files (or ‘**JA’ files** for short) provide a simple jumping-off point to train some preliminary models (the results of which we shall see in the ‘Results’ section. The form of a JA file that corresponds to that of the AD file seen above is the following:

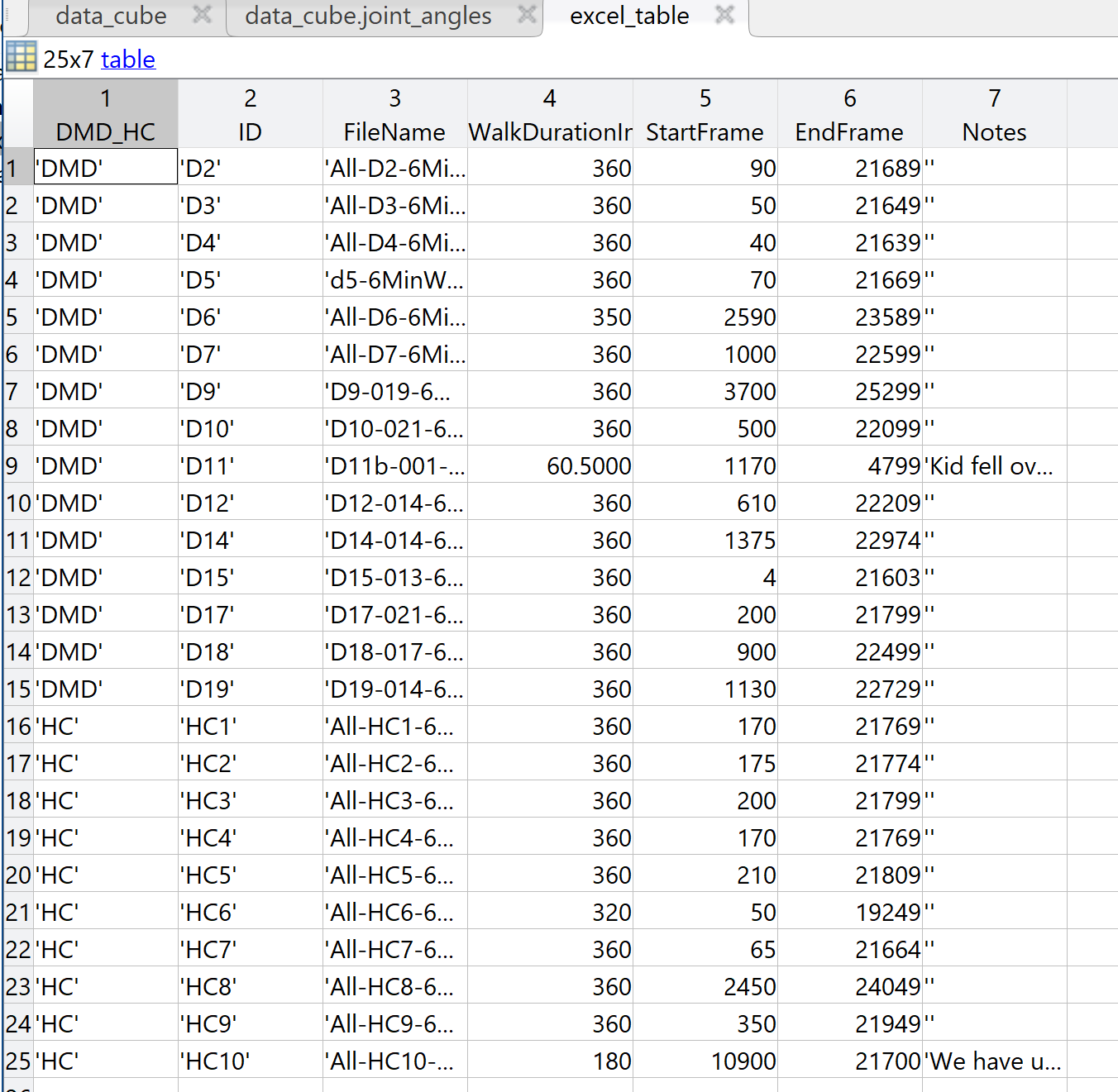


As can be seen above, the table now looks very similar to how a corresponding ‘.csv’ file would also look, which lends itself to simplicity in reading the data into Python scripts and using it to train a model. Note that the dimensions are (21809, 66), with ‘21809’ corresponding to ~363 seconds worth of data over 66 dimensions (i.e. 3 dimensions of 22 joint angles).

These joint angles files are also contained within what we call a ‘**data cube**’ (or ‘**DC**’ for short). This single ‘.mat’ file contains the joint angle data for 25 subjects and a table that contains information about them, as seen below:



The image above shows how the data cube contains cells which each contains a whole joint angle file’s worth of data within them (that look similar to that seen above), while the table below shows the table contained in the data cube that contains useful information (metadata) about the respective joint angle files. Hence, due to the useful structure and provided table, when we look to train an RNN model on raw joint angle data, we use the data cube as standard for the first several experiment sets (though we later extract the joint angle data from ‘AD’ files as is done for other raw measurements in later experiment sets and model predictions sets).



Finally, an important distinction to make is what sort of real-world activity each ‘.mat’ file actually shows. The first type is ‘**6minwalk**’, which as the name suggests is 6 minutes’ worth of data (so approximately 21600 frames) of the subject walking around a given area more-or-less continuously. This is provided to us as ‘AD’, ‘JA’ and ‘DC’ files, so for a given subject we can chose how we wish to interpret their data. The second type we are currently concerned with is ‘**NSAA**’, which are files usually between a length of 3 and 10 minutes that contain the subject carrying out the 17 activities that are set as part of the North Star Ambulatory Assessment. These files, however, are only provided to us as ‘AD’ files and do not come in ‘JA’ or ‘DC’ form (though we can still extract the raw joint angle measurements from an NSAA file via the ‘ext\_raw\_measurements.py’ script; more on that in the ‘Script Ecosystem Overview’ chapter). The final type is the natural movement behaviour data set provided to us in the ‘**allmatfiles**’ directory (which contains the joint angle data for the natural movement behaviour data of the subjects) and in the ‘**NMB**’ (which contains all the measurements from the natural movement behaviour files and which we received late in the project, hence why we used ‘allmatfiles’ for a long time). These directory contains numerous files from the same subjects as outlined previously performing various natural movement behaviours, such as sitting and talking, eating lunch, playing, and so on; while this isn’t used in most of the experiment sets to begin with, we do use this later in some of the model predictions sets.

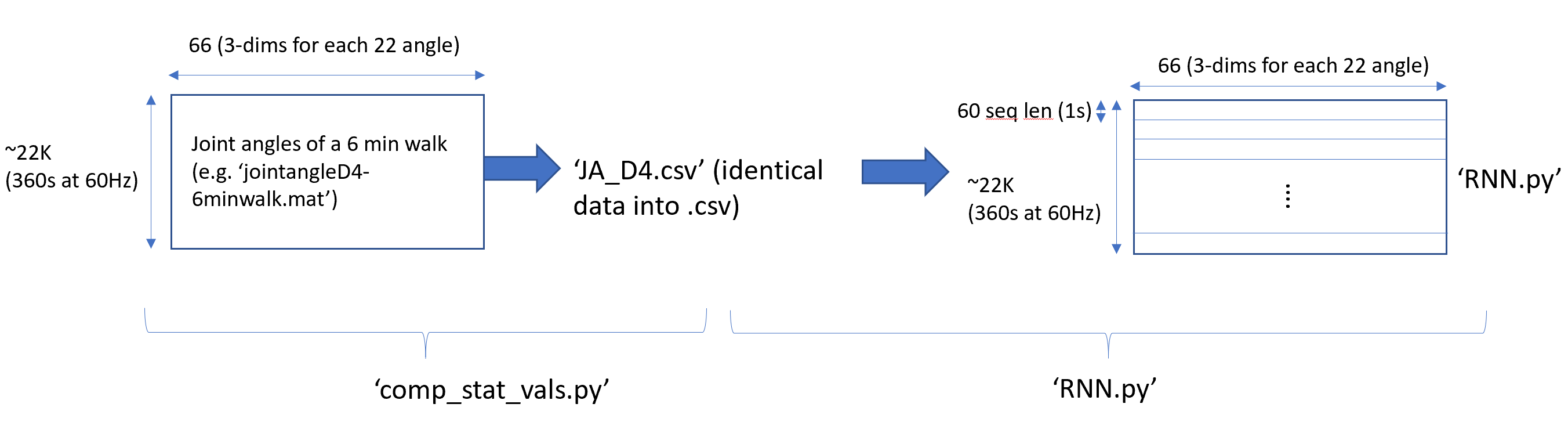
With the forms that the data can take now summarized, we can move onto what we actually do with this data prior to it being used to train an RNN.

**The Data Pipeline**

What is referred to as the ‘data pipeline’ is shorthand for four Python scripts (‘comp\_stat\_vals.py’, ‘mat\_act\_div.py’, ‘ext\_raw\_measures.py’, and ‘ft\_sel\_red.py’) that read from data files, manipulate data, and write to new files in the form of ‘intermediate data’. The aim of the pipeline is to convert the data that is specified by the user of the script (via arguments passed to the Python scripts) into a format that is ultimately usable by the RNN model. The specifics of what each script does are not covered here for the sake of brevity (see ‘Script Ecosystem Overview’ for this), so we instead focus on the different shapes and forms the data goes through depending on whether it came in as an ‘AD’ or ‘JA’/’DC’ file (as the ‘DC’ simply contains multiple ‘JA’ files, we treat both ‘DC’ and ‘JA’ files the same way). Refer back to the subsection ‘The Local Directory: Data Sets’ within the ‘Project and Local Directories’ chapter for information about the source and destination folders used for the variations outlined below.

**Variation 1: Joint angles from ‘JA’ and ‘DC’**

Below, we can see the pipeline with respect to ‘JA’ or ‘DC’ file(s) as input to the RNN.

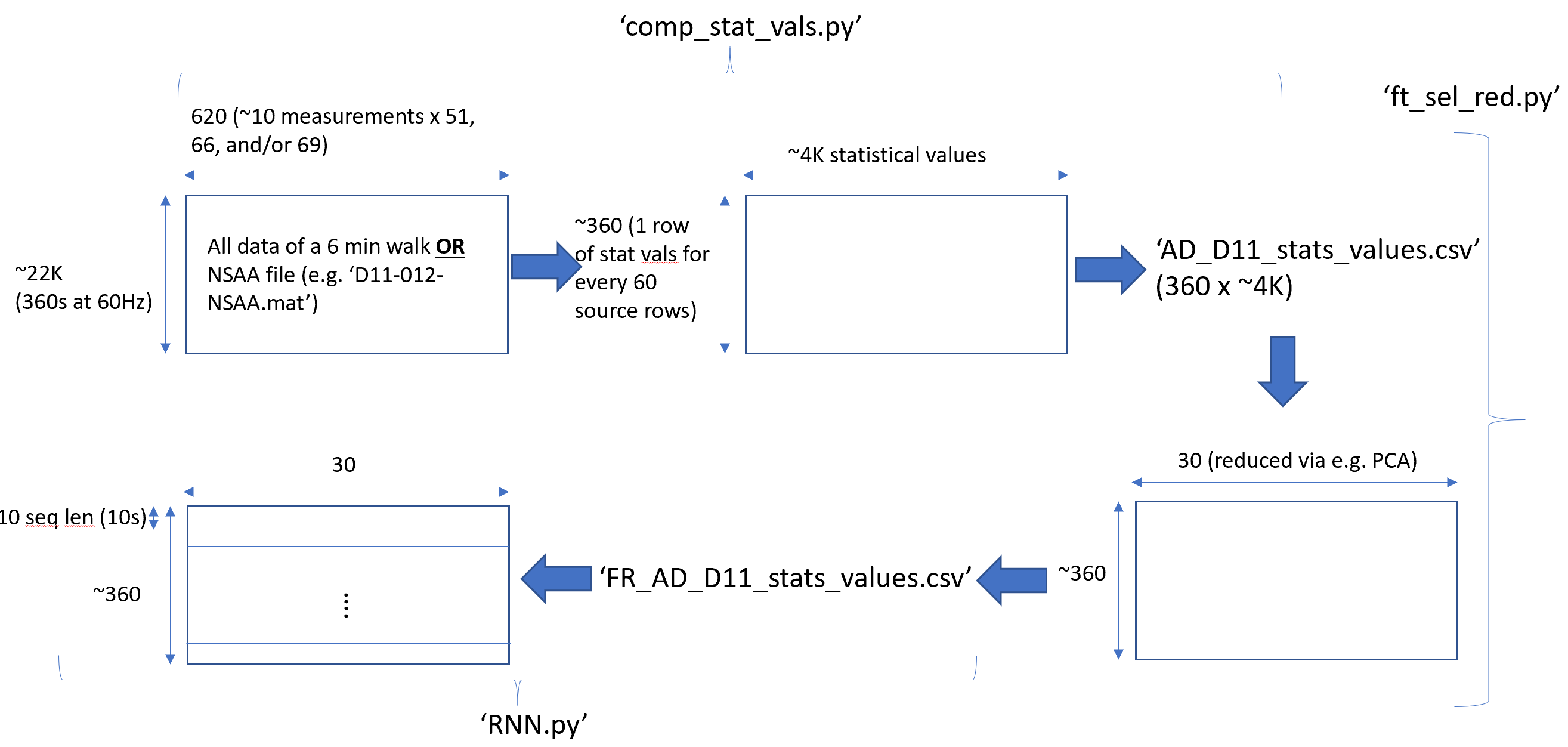


This is what is done when we are dealing with **raw joint angle** data. Let’s say we wish to feed in the ‘JA’ file for subject ‘D4’, as seen in the diagram above. The data comes in the form of a 22K x 66 matrix, as can be seen in a previous image, which is loaded by the ‘comp\_stat\_vals.py’ Python script and simply written back out in exactly the same way to a ‘.csv’ format. In essence, we are removing some of the metadata from the ‘.mat’ file and then transforming the data values into ‘.csv’ format. The reason we do this is because the ‘RNN’ is expecting to load data from a ‘.csv’ format when dealing with all files regardless of its setup, so this essentially standardises the process so the RNN works in the same way for each different original data file type. An important aspect to note, though, is that this is also what is done with the ‘JA’ or ‘DC’ files: in the case of ‘AD’ files, to get the raw measurements we instead use ‘ext\_raw\_measures.py’.

In the ‘JA’/’DC’ case, this data is then loaded by the RNN from ‘JA\_D4.csv’ and has sequences made out of it. In this context, we refer to a **sequence** as a 2-dimensional grouping of data that the RNN will treat as a single sample of data. In the above case, the sequence is a (60, 66) matrix of values: each row of 66 values are fed into the RNN one after another, repeating for a total of 60 times before the output of the RNN is observed; the RNN is essentially ‘reset’ before the next matrix of values is considered, hence the sequences are essentially independent of each other. In this case, the data is time dependent of each other only in 1 second intervals (given a 60 Hz sampling rate of the body suit), while outside of these sequences the data is treated independently of each other (in the same way as each image being classified by a convolutional neural network is independent of each other).

**Variation 2: Computed statistical values from ‘AD’ files**

Next, we can see below the pipeline with respect to ‘AD’ files(s) as input to the RNN:



The first point to notice is that the data starts with far more dimensions than ‘JA’ or ‘DC’ files. This is because the ‘AD’ files consider ALL the measurements and not just one measurement (the joint angles). With using them as input to ‘comp\_stat\_vals.py’, we process them differently; namely, we don’t just write them directly to a .csv, but rather calculate statistical values. These operations (which including finding the mean, variance, first/second eigenvalues of covariance matrices, fast Fourier transform values, mean sum absolute values, and so on) operate on only a limited number of rows of the data at a time. For example, in the above case, we select a given number of rows of the 22K frames at a time (in this case, 60 to correspond to 1 second’s worth of data), calculate statistical features over every single one of its 620 dimensions (along with between some of the 620 dimensions), and compute this as a row of approximately 4000 statistical values. This is then repeated for every other 60-frame part of the original 22K to produce a total of approximately 360 rows, each containing 4000 statistical values.

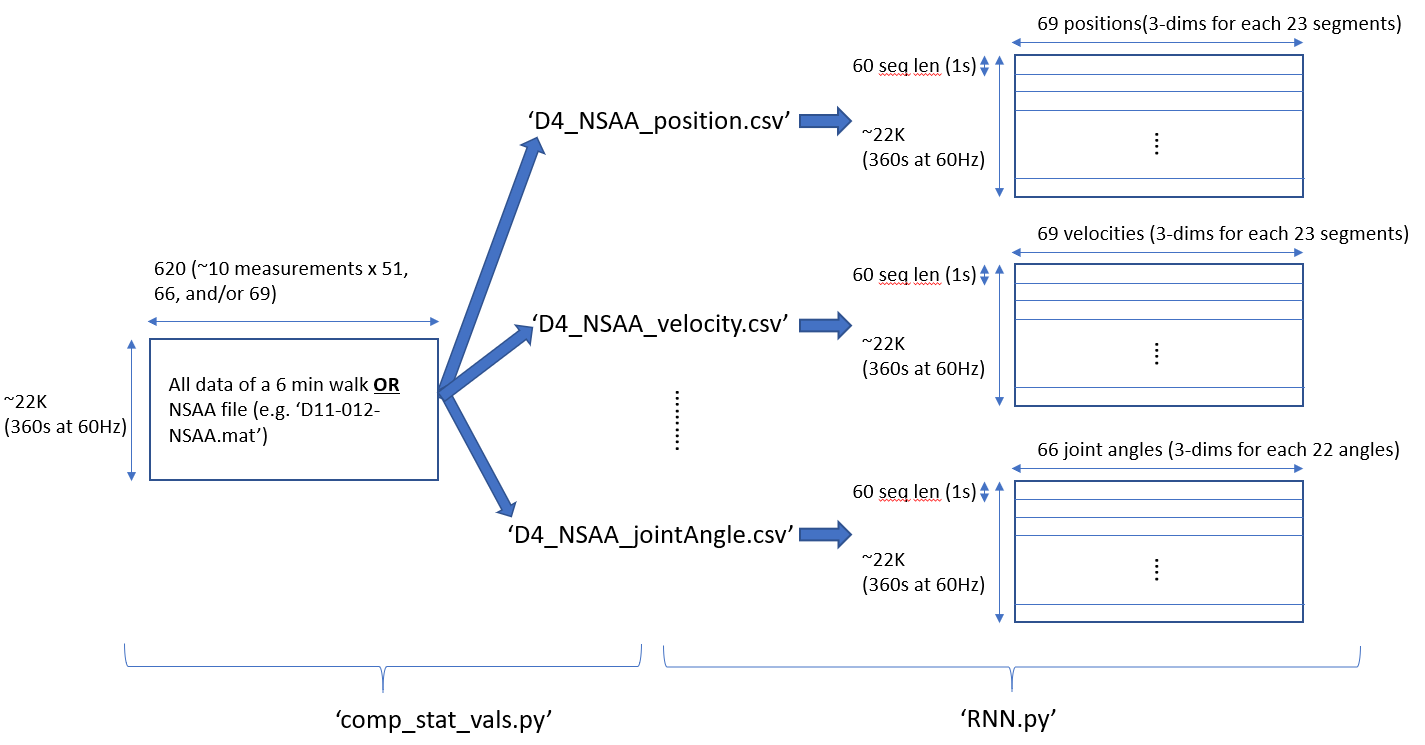
Much like the sequence length that the RNN processes, the length of time to compute the statistical features over is a parameter that is set as an argument to the script (i.e. a hyperparameter that we set ourselves); hence, the size of ‘60’ is a value that has been determined to produce enough data while calculating over enough rows. A thing to bear in mind, though, is that if we increase this parameter so the statistical values are calculated over a longer time period, we will reduce the number of rows that are outputted from this script, hence reducing the amount of data available at the next stage of the pipeline (i.e. the data going into the feature reduction script outlined below).

With this data having its statistical values computed and outputted to a new file (in this case of the above image, ‘AD\_D11\_stats\_values.csv’), we then load this .csv into a new Python script called ‘ft\_sel\_red()’. The purpose of this is to simply reduce the dimensionality of the data while preserving as much useful information as possible and to then rewrite this to a new file. While there are several options in feature reduction and feature selection (including principal component analysis, random forest feature selection, feature agglomeration, among others), we use PCA as standard based on its prevalence in other studies. Hence, when run on the input data and with a target reduction size set to 30 dimensions (again, this is a hyperparameter set as a script argument and is subject to experimentation in experiment set 7), the data is transformed from a dimensionality of (360, 4000) to (360, 30), which is far more conducive to being used as training data for the RNN in the next stage, as this reshaping helps minimize the effects of the ‘curse of dimensionality’.

Finally, this is fed into the RNN and sliced up into sequences of length that we defined in the same way as in ‘JA’/’DC’ as previously described. A choice of ‘10’ is used here as the sequence length due to the limited size of the data (360, 30) in comparison to the case of reading from raw joint angle files (22000, 60) Also, it’s worth noting that even though the sequence length is only ‘10’ here, as each of the rows here have a full 60 rows worth of data from the original ‘AD’ file (which corresponds to 1 second’s worth of data embedded within them), each sequence from an ‘AD’ file to the RNN has 10 seconds worth of data encoded into it as statistical values. In comparison, the raw joint angle data has a sequence length of 60 and encodes only 1 second’s worth of data. This evidently plays a role in the difference in their respective results, which we shall discuss shortly.

**Variation 3: Extracted raw measurements from ‘AD’ files**

Below we can see the pipeline with respect to using ‘AD’ files as a source of raw measurements:



Unlike the previous two variations of the data pipeline, here we don’t make use of either ‘comp\_stat\_vals.py’ or ‘ft\_sel\_red.py’. This is because in this setup we neither compute statistical values, nor extract exclusively joint angles, nor do we need to do any dimensionality reduction. Rather, we wish to extract any raw measurements that we need from a given file. This is done by the ‘comp\_stat\_vals.py’ script: it takes the name of a file to extract the raw measurements from (or multiple names to carry out this process on) and the names of the raw measurements we wish to extract. From here, the script will seek out the columns of relevance within the tree structure of the corresponding source ‘.mat’ file for the given subject(s), as each vector of a certain raw measurement at each time instance is stored in its own column of the ‘tree.subject.frames.frame’ table which, when expanded, becomes a matrix of values for the raw measurement. This matrix of values (in the case of the ‘position’ measurement having a rough shape of (22000, 69)) is then written to a new ‘.csv’ file with a name reflecting the subject name the measurements are from along with the measurement name (e.g. ‘D4\_NSAA\_position.csv’).

From here, the ‘rnn.py’ script is able to build models from these other measurement types in the same way it has done so for the joint angle data and the computed statistical values from the first two variations of the data pipeline by providing a different raw measurement name (e.g. ‘sensorMagneticField’ or ‘acceleration’) as opposed to just either ‘AD’ for computed statistical values (note here that ‘AD’ references to the computed statistical values; see the Glossary for further clarification about the two ways in which ‘AD’ is referenced) or ‘jointAngle’. After this, the following steps are identical, from the creating of sequences to the training and testing of models. It should also be noted that, like the ‘JA’ and ‘DC’ files, the extracted raw measurement values from ‘AD’ each represent 1/60th of a second’s worth of data; hence, when they are used to create sequences of 60 rows of raw measurement data, this represents 1 second’s worth of data in the same way it does for the ‘JA’/’DC’ files even though it comes from the ‘AD’ files: it’s only when computed statistical values are outputted that the data originating from ‘AD’ files that it comes to represent 1 second for each row rather than only 1/60th of a second.

**Output Types**

Now that the data has been prepared as a series of sequences of data either from a single ‘AD’, ‘JA’, or ‘DC’ file, or multiple of each, we now look at what the recurrent neural network models we have built actually do. It’s important to note that each of the variations works with every type of input file, be they from raw joint angle files or ‘AD’ files capturing either 6-minute walk or NSAA data; that is, after all, a primary purpose of using the pipeline. It’s also necessary to note that the trained model in every variation operates on a sequence-by-sequence basis even for testing; that is to say, it doesn’t classify or provide a regression score for a complete file but rather for a sequence of a pre-specified length within said file. The classifications or regression values of each of the sequences of the file being assessed can then be aggregated together to provide a classification or regression score for the whole file (how this is exactly carried out is subject to further research). Finally, it’s worth noting that each variant is implemented within one Python script called ‘rnn.py’, and the necessary architectural differences are setup based on arguments passed into the script.

The three main variants (or ‘**output types**’) that we have developed are described as follows:

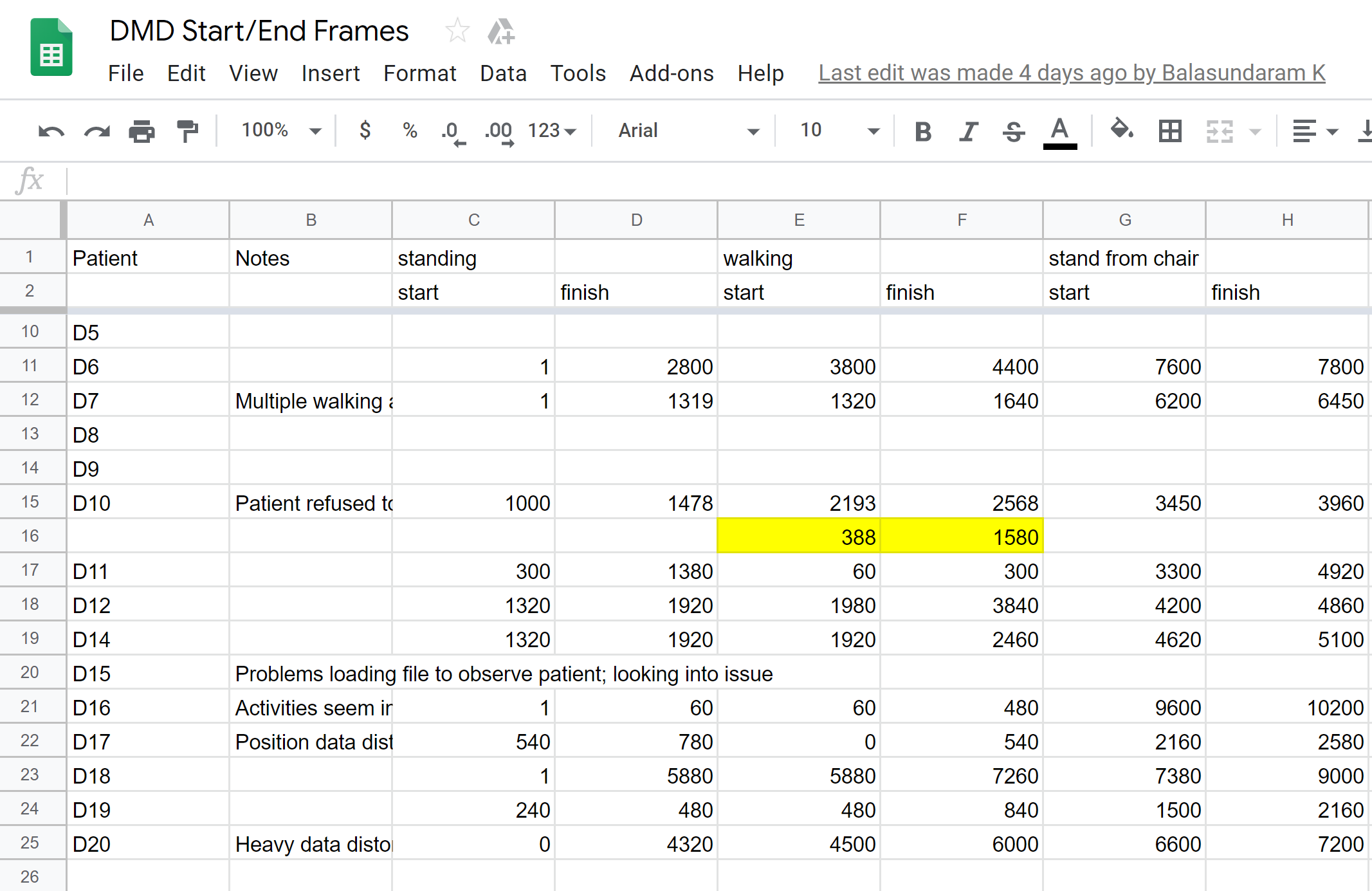
* **‘dhc’: Classification of sequences of being from ‘DMD’ or ‘HC’ subjects**: The purpose of this variation is to classify sequences as being from a file that is from either a ‘DMD’ or ‘HC’ subject. Consider the previously outlined case where ‘FR\_AD\_D11\_stats\_values.csv’ is fed in from the pipeline to ‘rnn.py’ at the end. When it is loaded into the ‘RNN’ script, it’s divided into ‘36’ sequences of length ‘10’ to give a data shape of (36, 10, 30): this is the ‘x’ data in the sense of a neural network. For the ‘y’ data, we simply look at the title of the file this data originated from to provide a label of either ‘1’ or ‘0’ depending on the nature of the file name: since it’s a ‘D’ file due to it being about patient ‘D11’ it gets a label of 1. This is then repeated for each sequence to obtain a list of ‘1’s of length 36. We then repeat this process for all other files pushed through the data pipeline, some of which will be from ‘D’ subjects and others from ‘HC’ subjects. The result, in the case of doing this over all ‘AD’ files for files corresponding to NSAA subject assessments, is an input shape of (742, 10, 30) for ‘x’ and (742, ) for ‘y’, which contains a mixture of 1’s and 0’s for each sequence. There is also only a single neuron output for the network that contains categorical value of either 0 or 1 for a given sequence: this output will go to 0 if the sequence inputted is predicted to be from a ‘HC’ subject or 1 if predicted to be from a ‘D’ subject.
* **‘overall’: Overall NSAA regression score**: Here, the RNN is tasked with taking a sequence and trying to predict the overall NSAA score of the subject that it comes from. This score corresponds to the accumulation of the scores of the 17 individual activities done in the subject’s assessment. As each activity is scored either a 0, 1 or 2 (with 2 being a perfect score), the overall NSAA score will range from a 0 for a subject with severe DMD and a 34 for a subject that shows no symptoms (i.e. a ‘HC’ subject). Using the above example, we start with a shape of (742, 10, 30) over all the NSAA AD input files. For each of these 742 sequences, we then check a table that contains the subject information (‘nsaa\_6mw\_info.xlsx’, as described in the previous chapter): this table provides a list of the subjects, their testing details, and crucially their individual NSAA scores (17 of these between 0 and 2) and overall NSAA score (1 of these between 0 and 34). This overall score is then used for every sequence from a given file that is inputted to the ‘RNN’. Using this, we then obtain 742 values between 0 and 34, with each value being the overall NSAA score of the source file of its corresponding ‘x’ component of shape (10, 30). From here, we again have in the RNN architecture a single output neuron, but this time it outputs a regression value (rather than a classification value as in the previous case) between 0 and 34; hence, each sequence that passes through the RNN will result in it making an estimate of the overall NSAA score of the subject that the sequence originates from.
* **‘acts’: Classification of NSAA single activity scores for all 17 actions**: In a similar vein to predicting the overall NSAA scores, the RNN is also able to train towards predicting individual activity scores; that is to say, given a single sequence, the RNN will output an array of 17 values, each being either a 0, 1, or 2, that corresponds to its prediction of the individual activity scores of the subject that the sequence originates from. Again, given the same ‘x’ data fed through the pipeline, the corresponding ‘y’ values to train and test on are obtained from the ‘nsaa\_6mw\_info.xlsx’ table to obtain the necessary array of 17 values for each sequence. Hence the data fed into the RNN now has a shape (in the case of all NSAA AD files being used) of (742, 10, 30) for ‘x’ and (742, 17) for ‘y’. To account for this, the RNN is modified to predict 17 individual classification labels of either 0, 1, or 2 for 17 output neurons.

Currently, all 3 variants have many of the same hyperparameters, including the train-test ratio of data (0.2; excluding the ‘--no\_testset’ optional argument being set which sets it to 0), the number of units in each LSTM cell (128), the number of hidden layers (2), and the learning ratio of the ‘adam’ optimisation algorithm (0.001). The reasoning behind this was that differences in experiment results will hopefully be down to differences in source data file types (e.g. raw joint angle files of 6-minute walks vs ‘AD’ files of 6-minute walks) rather than potentially different architectures (e.g. if one had more hidden layers, increased performance may be resulting from this rather than the source of the data going into the RNN, which is what we want to be able to compare). There are some differences though: a policy decided upon early was that the RNN should train until its loss more-or-less converges. This required different numbers of epochs dependent on the source data type. For example, if the data going into the RNN came from raw joint angles, it only needed approximately 20 training epochs to converge, whereas data from ‘AD’ computed statistical value files needed between 200-300 to converge. Finally, it’s also worth pointing out the main difference in training for classification (either for single or multiple output nodes) and regression was the different loss functions used, in that classification used binary cross entropy and regression used mean-squared error.

It should also be noted that there is one more output type that is slightly unlike the others and that is the ‘**indiv**’ output type. This setups the models to only have one output neuron that computes a score of between 0 and 2, much like the ‘**overall**’ output type (though models trained for the ‘overall’ output type can range from between 0 and 34 for an output value). However, for the ‘indiv’ output type, we are only concerned with ‘single-act’ input file data, i.e. files that contain data of a certain raw measurement (or computed statistical values) for a single activity for a certain subject that are produced by the ‘mat\_act\_div.py’ script. Hence, the aim of this output type is to compute, for a given subject’s single-act file that has a corresponding true value associated with it (that is contained within the ‘nsaa\_6mw\_info.xlsx’ table), the predicted value between 0 and 2. Note that this is a regression task, and so the model can predict anywhere between 0 and 2, with the final prediction made by ‘model\_predictor.py’ for a given subject based on the average of these predictions for a sequence and then rounding this average to the closest integer to get the predicted value for that complete subject. We should also note that the reason we don’t consider this output type with the others is that it operates on a different file type: while the other three output types can be obtained for any input file, the 'indiv’ output type can only be done for single-act files. This makes intuitive sense, as it’s somewhat pointless to ask a model to predict a single act score corresponding to a specific activity for a sequence from a subject which could have come from anywhere within the source file (i.e. any one of the 17 activities or the ‘in-between’ activities data).

**Additional Data Preprocessing Work and Tools Used**

One of the disadvantages of the NSAA files provided is that all the activities for each subject are provided in the same ‘.mat’ file and the table that provides information of these subjects’ trials with the body suit (‘nsaa\_6mw\_info.xlsx’) contain information about the overall cumulative NSAA score as well as the individual activity scores (among other meta data), but not the specific times of occurrence of each activity. These activity times are needed in order to ‘divide up’ the original ‘AD’ files into ‘AD’ files that contain only the timeframe of a specific activity via the ‘mat\_act\_div.py’ script, which would be useful to have for this project as well as other research initiative projects. Hence, we were tasked as a group to work together to create a table of each subject’s predicted times for each activity. A portion of the results, contained in a shared Google sheet called ‘DMD Start/End Frames’, can be seen below:



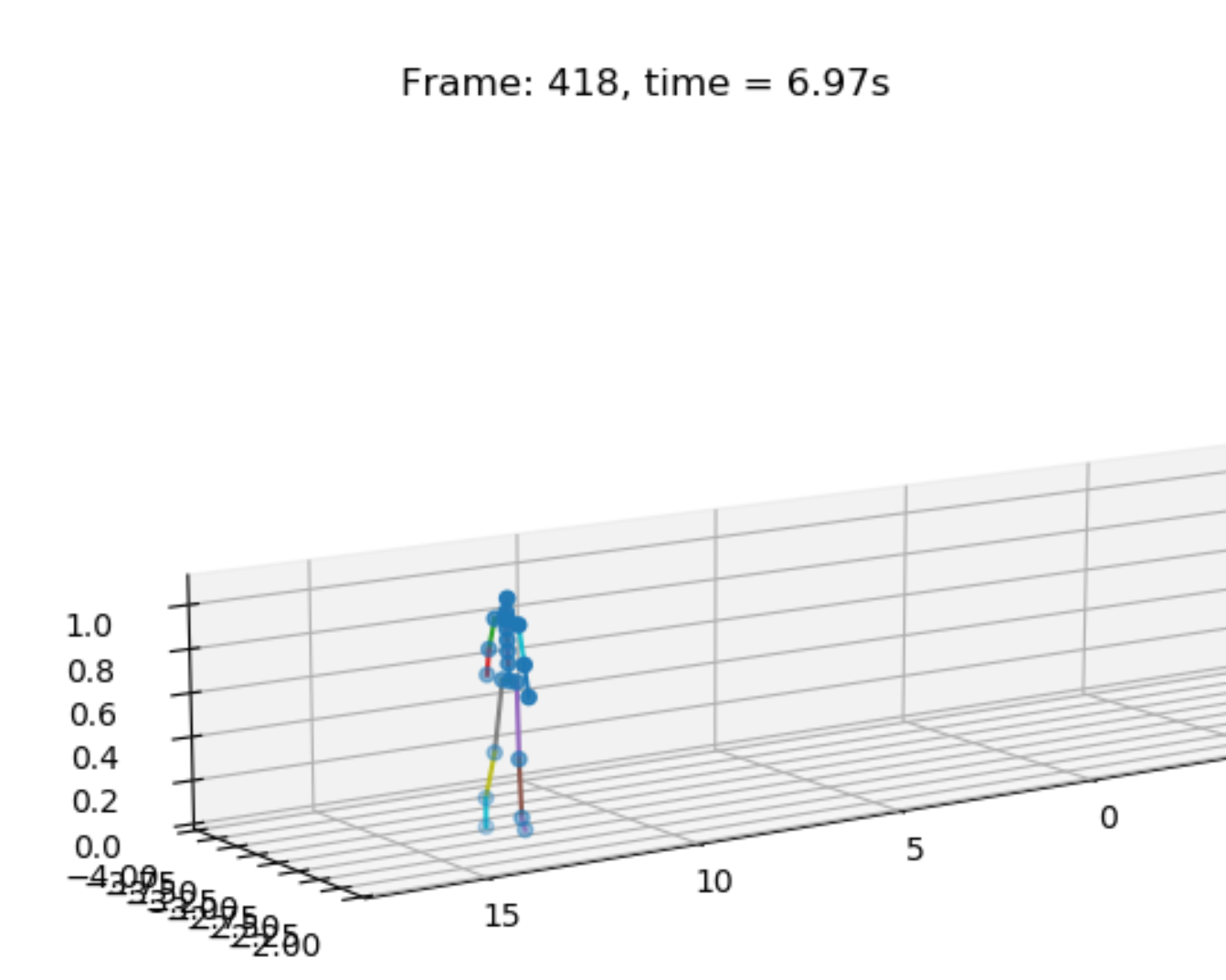
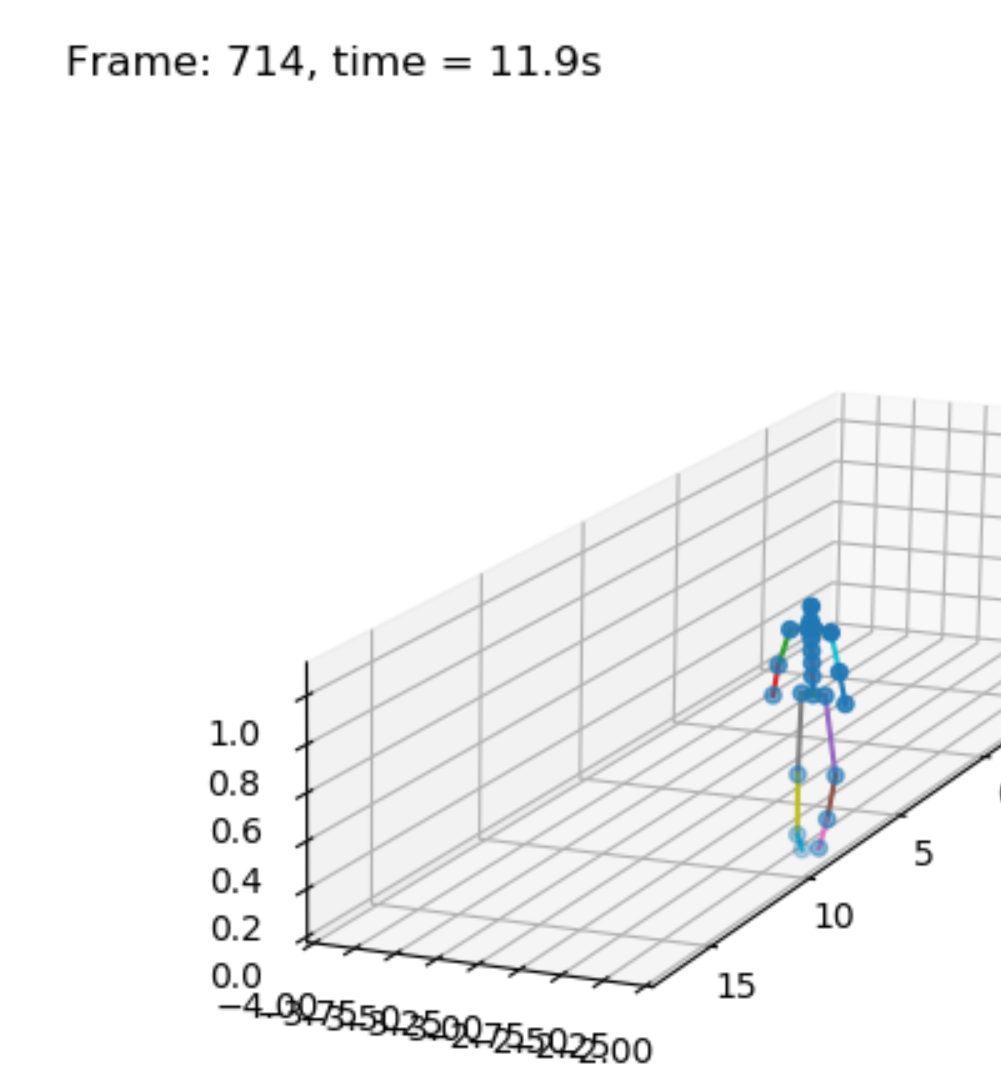
While we have touched on the above file within the ‘Google Annotations Sheet’ section of the ‘Reference Documents’ Chapter, it’s worth at this point clarifying a few things about how it was put together. The above image is only a portion of the table, and there are many more columns that aren’t displayed, but the above can be interpreted fairly simply. For example, for the ‘D11’ subject’s NSAA ‘AD’ file, the file is observed via the ‘dis\_3d\_pos.py’ script (elaborated upon in the ‘Script Ecosystem Overview’ chapter) to show the subject do the ‘standing’ activity between frames 300 and 1380 (corresponding to between 5s to 23s) and the ‘stand from chair’ activity between frames 3300 and 4920. There were several rules that were followed in order to extract these frame times by human observation:

* As activities were often repeated by the subject, we consider only the first completed activity as the start and end points in the sheet; for example, if the subject tried to do the activity ‘hop on right food’ several times without success before being able to do so, we only count the last successful attempt in the table.
* Ideally, we try to give a small amount of ‘leniency’ on the start/end times of the activity; the idea behind this is that no part of the activity is therefore ‘missed’ when it’s divided up into smaller ‘AD’ files.
* Some of the activities were either seemingly not performed or performed alongside another activity, or performed very subtly; for example, the ‘nod head activity’ was particularly tricky to detect in many subjects. In these cases, a best guess on the time of occurrence of the activity was made; the checking of how we divided up these files is further continuation work that can be done for the project and is covered later.

So given that we have the ‘rules of thumb’ for our annotation work, it was next necessary to actually ‘visualize’ these ‘AD’ files. As there was no easy way of ‘running’ one of these ‘AD’ files in ‘.mat’ format and as we (at that point) had no access to the ‘.mov’ video files that corresponded to each source ‘.mat’ file, the project necessitated a script that could use the thousands of ‘position’ values of the AD file (‘position’ being one of the measurements in an ‘AD’ file along with ‘jointAngle’, among others) to feed into a function that animated a stick figure as it moved around three-dimensional space. This was the work of the ‘dis\_3d\_pos.py’ Python script and, when the script is run with the required arguments, e.g. for the ‘D11’ subject , the script does the following:

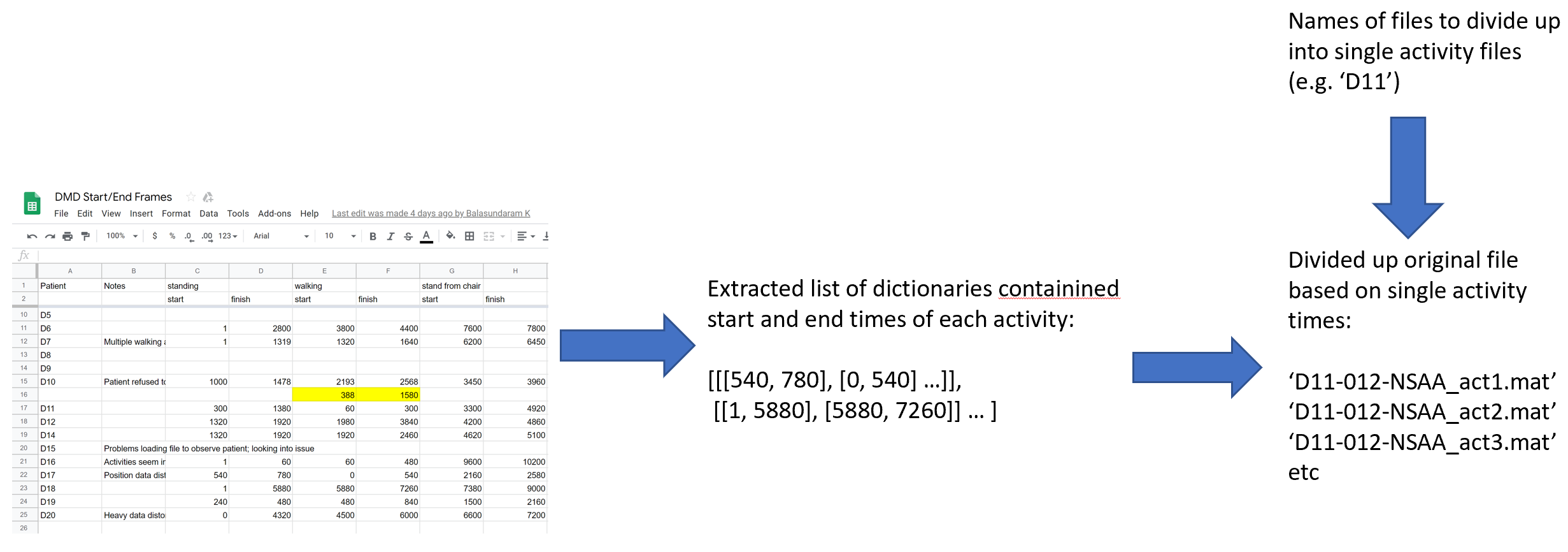
1. Loads the position values from the ‘AD’ file for subject ‘D11’ into a massive array.
2. Group each of the columns corresponding to ‘x’, ‘y’ and ‘z’ dimensions of all segments together.
3. Send these to the ‘animate’ function that animated it at a rate of 60Hz (so the animation would appear as real-time) and closed upon completing the plotting of all values for that file.

The result of this was 3D data in a new window that looked like the following:

Additionally, the current running time of the animation function was also printing to the console; this enabled us, by observing the movements of the stick figure and what activities it seemed to perform, to get a reasonably accurate idea of the times of occurrence of each activity.

With the Google annotations sheet now being complete with the help of the above plotting script, we could now use the sheet as a reference tool for a script that could ‘divide up’ a given source ‘AD’ file; namely, the ‘mat\_act\_div.py’ script. The rough functionality of the script can be summarised in the diagram below:



When the ‘mat\_act\_div.py’ script is then run with an argument given to be the name(s) of the file(s) to ‘divide up’ based on the annotated Google sheet, it does two things:

1. Reads in the Google sheet that we had previously annotated, locates the relevant row given the provided arguments, and extracts a list of the activity times (both start and end times) as pairs of integers and the names of the files that contains the activities.
2. Loads in the relevant ‘AD’ file for the given argument(s) and, using the pairs of integers in the above list, slices up the file into 17 non-overlapping parts and writes these to a subdirectory of the file(s)’s source data directory with names specific to the activity and subject source file (see the names in the bottom-right image above).

While these divided-up ‘AD’ files are not used to tune the models for the general cases of using the system, these are used instead in later model prediction sets to train RNN models to predict single-activity NSAA scores and evaluate the significance of particular activities with respect to predicting overall subject assessment.