Working with EMA data files in MATLAB

Jens Roeser & Adamantios Gafos*

Department of Linguistics, University of Potsdam

*email: gafos@uni-potsdam.de

October 19, 2014

1 Introduction

This guide describes how to handle data structures containing data from electromagnetic articulography (EMA) in MATLAB. We are presupposing basic MATLAB working experience. However, all necessary information to work with EMA data in MATLAB are included. We will make reference to programming concepts like data types, which we consider important for working with data. However, since we do not elaborate on data type concepts (we merely tell the reader what these types are and carry on with our narrow interests in this guide), we remind the reader of the MathWorks homepage for further information. There is of course a wealth of introductions to MATLAB, such as Attaway (2012), Gilant (2011), and Rosenbaum (2012), if the reader feels unable to follow the MATLAB coding herein. Quarteroni et al. (2010) is recommended for readers with interests beyond the scope of the present guide.

In this guide, we focus on data sets that are provided by experiments carried out with the EMA method. While reading this guide, apply the provided commands in your MATLAB console or from a script to an available binary `.mat` EMA data file. Type the commands rather than copy-paste them. Solve the included tasks to understand the structure of EMA data files and how to extract relevant information.

EMA data files have a `.mat` extension and contain acoustic and articulatory information. These files are data structures which are called structure arrays – a particular data type in MATLAB. A structure array is a collection of information of different types (e.g., strings, also known as character arrays, or numbers like integers or floating-point numeric data) and different lengths all yoked together in one object (here, usually a stimulus elicited from a participant in our experiments). Inside such data structures there are fields each containing a subset of the data (the movement in time of one sensor or another or the acoustic waveform). We will make this more concrete in what follows. In contrast to cell arrays, a structure array has one name for many data subsets (here, different trajectories and the audio waveform).

2   Data structure

If the relevant EMA data file that you are going to work with is contained in your current working directory, the `load` function can be used to get the file into the MATLAB working environment. The name of the data file with `.mat` extension must be given to the function as argument. Note that the name must be a string in single quotes. If the file is located somewhere else, the entire path or a relative path (relative to the current directory) to that file must be given as an argument. The structure array is assigned to the variable `file` using the equal sign. `file` now contains a 1×1 structure array with only one element. We will see in a bit that an 1×7 structure array is embedded in this element and can be accessed for processing.[1] The double arrows `>>` indicate a command to be given to MATLAB by you, the user. The lines without these arrows indicated a MATLAB-generated echo. Do not copy `>>` when copying the code in the MATLAB command window.

```
>> file = load('cascade_09_0823.mat')

file =

    cascade_09_0823: [1x7 struct]
```

If you think it is annoying to type long file names to load data, the names of the files in your directory can be accessed using `dir` as in the code below. This command returns the names of and information about the files in a directory. The asterisks $*$ is used as a wild card assigning only the files that end in `.mat` as specified in the string below to the variable `list`. The `.mat` files in your current working directory will be listed in an $n \times 1$ structure array with n being the number of files in the current directory and the first field of the returned structure array being the file name. The other fields are not relevant here. Fields in a structure array can be accessed using a dot `.` after the name of the structure array and before the field name that must be accessed shown in the line after. Curly brackets $\{\}$ return all information listed in the `name` field being the names of the `.mat` files in your current working directory.

---

[1]Unfortunately, the single quotes used in the commands throughout this guide are not the same characters as those corresponding to what MATLAB considers to be a quote, due to the LaTeX compilation. Thus, if you were to copy and paste the commands used in this report into the MATLAB console, you will see `Error:  The input character is not valid in MATLAB statements or expressions`. Change the quotes in the copied command by entering single quotes from your keyboard.

```
>> list = dir('*.mat')

list =

4x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum

>> {list.name}

ans =

  Columns 1 through 2

    'cascade_09_0823.mat'    'cascade_10_0865.mat'

  Columns 3 through 4

    'clean_01_0048.mat'    'clean_02_0152.mat'
```

As we did above, the first file listed in `list.name` can be loaded with the `load` function. To load this file, the structure array `list`'s field name is indexed by 1 for the first file like shown below. Again, the data are encapsulated in a 1×1 structure array with one field named after the file (without extension).

```
>> file = load(list(1).name)

file =

    cascade_09_0823: [1x7 struct]
```

This was an alternative way to load a file into MATLAB. Now we are back from where we started before – an EMA file has been loaded. This alternative way to load data is more flexible for loading multiple files and avoids annoying typing of long file names.

`file` is a variable which we set equal to whatever MATLAB data structure corresponds to the file `cascade_09_0823.mat`. Each variable in MATLAB (and any programming environment) is of a specific Class (or Type). The type of this variable is what MATLAB refers to as a `struct array`. You can see the type above in the `[1x7 struct]` part or you can ask MATLAB to give you the type using the `whos` command below. Note, that it is often important to keep track of the data type.

```
>> whos file
  Name      Size              Bytes  Class      Attributes

  file      1x1              591104  struct
```

We do not know what and how information is stored in `file`. To find that out, we need to know the *fields* of the structure (because in MATLAB structure arrays are data structures that store information in separate fields). To get to know what these fields are, we apply the function `fieldnames` to `file` which returns a cell array of strings containing the structure field names associated with some structure. The `help` function gives the basic information about the function of interest.

```
>> help fieldnames

 FIELDNAMES Get structure field names.
    NAMES = FIELDNAMES(S) returns a cell array of strings containing
    the structure field names associated with the structure s.

>> names = fieldnames(file)

names =

    'cascade_09_0823'
```

It just happens that our structure array assigned to `file` actually has just one field and the name of the field is the name of the file itself. That is not always the case. It could be that the structure array has more than one field (e.g., `date`, 'date data were collected', `rate`, 'sampling rate' and so on for `list` above. Since now we know the field name of our structure array, we can ask to see its content by using the function `getfield` below with the structure array `file` as its first argument, and the name of the file as its second argument.

```
>> help getfield

 GETFIELD Get structure field contents.
    F = GETFIELD(S,'field') returns the contents of the specified
    field.

>> data = getfield(file, names{1})

data =

1x7 struct array with fields:
    SIGNAL
    SRATE
    NAME
```

We have learned above that the function `fieldnames` gives us a cell array of strings, and for cell arrays one needs to use curly brackets {} to access any specific entry in the cell array and since we want to access the first entry here we need to write `names{1}` to retrieve the first file.

`data` contains a 1×7 structure array with the fields `SIGNAL`, `SRATE`, and `NAME`. This may differ from your output, since EMA data files contain different kinds of information. Each of the seven entries of the structure array contains one field for the audio signal and six fields for movement trajectories. To see which signals are available, the field `NAME` can be accessed as shown above for the files in `list`. Here, the field `NAME` is accessed in the structure array `data`. The values of the field `NAME` are returned as strings and assigned to the variable `trajectories` (which is a cell array of strings).

```
>> trajectories = {data.NAME}

ans =

  Columns 1 through 6

    'audio'    'TBPOS'    'TMPOS'    'TTPOS'    'LLPOS'    'ULPOS'

  Column 7
```

```
'JAWPOS'
```

Exercise 1: The field `SRATE` contains information about the sampling rate used during the data recording. As we have done with the file names and signal names before, check which sampling rate has been used for audio and for the individual trajectories by changing the previous command (i.e., instead of looking for the `NAME` field, ask for the `SRATE` field).[2]

---

[2]Note that field names are in capital letters.

3   Elementary signal processing

3.1   Spatial signals

The field `SIGNAL` contains the signal that is discretized as sequence of values called samples, each spanning a small time slice as determined by the sampling rate. `SIGNAL` contains positional values for the x- and y-dimension of the tongue kinematics for 2D data files and additionally the z-dimension for 3D data files. Structure array information of single signals can be accessed using the column number from above, which is the corresponding field name. The names of each signal have been assigned to the variable `trajectories`. The following code shows how to find the index of the desired signal using `strmatch`. This function takes the name of the signal (here, `TBPOS` which stores the position of the tongue back sensor) as first argument and the vector with the signal names `trajectories` as second argument. This will provide an index saved in `idx` (here, 2) that can be used as an index for `data` corresponding to the signal name `TBPOS`.

```
>> idx = strmatch('TBPOS', trajectories, 'exact')

idx =

    2

>> data(idx)

ans =

  SIGNAL: [928x2 double]
   SRATE: 400
    NAME: 'TBPOS'
```

`NAME` is the trajectory's name, `SRATE` is the sampling rate (here, 400 Hz) used to record `TBPOS`. `SIGNAL` is a 928×2 matrix with 928 rows and 2 columns. The column number represents the x and y positional signals of the sensor glued on the tongue body and the row number corresponds to one row for each recorded sample (i.e., 928 samples for this file). The command below returns the signal of the x- and y-dimension for the first 5 values of `TBPOS` using indexing `1:5`, i.e. row 1 to 5 and all (i.e., both) columns using colon `:`. Indexing in MATLAB and many other environments takes first the row index

and then the column index. The left column shows the x-values and the right column the y-values. Again, every row or pair of values below corresponds to one sample of the position of the sensor in 2D. The `SIGNAL` values of `TBPOS` are saved in a vector matrix `signal_tbp`.

```
>> data(2).SIGNAL(1:5,:)

ans =

   42.4871    -5.1379
   42.4776    -5.1641
   42.4686    -5.1885
   42.4603    -5.2097
   42.4531    -5.2264


>> signal_tbp = data(2).SIGNAL;
```

Exercise 2: Repeat the same for the tongue tip trajectory (i.e., `TTPOS`). Use `strmatch` for `TTPOS` instead of `TBPOS` and index `data`. Then, find the positional values x and y of the 266$^{\text{th}}$ sample in the `TTPOS` trajectory. You can access positional values using `data(n).SIGNAL(r,c)` by replacing `n` with the field number of the trajectory name, `r` with the row number which is the sample index, and `c` with the column (i.e., 1 for x-values, 2 for y-values and for 3D data, 3 for z-values).

Speech signals (audio or kinematics) are stored as sequences of samples. These samples are taken at regular time intervals, whose regularity is expressed by a certain sampling rate or frequency. Frequency is the number of cycles of a periodic process (here, the sampling process) in one second, i.e. how many times the process repeats itself in one second. Frequency is expressed in units of Hz, with 1 Hz corresponding to one cycle per second. One property of a speech signal is its duration which is often expressed in real-time units such as seconds or milliseconds (msec). To convert from samples to duration, we need to know the frequency at which our signal was sampled. Equation (1) is the well-known expression for the relation between frequency f (in Hz) and period T (in seconds) of a repeating process. Hence, to find the duration t of a sampled signal in seconds we divide the number of samples n by the sampling rate f as in equation (1'). Conversely, if we know the duration or time stamp t in our signal and we would like to compute the number of samples n corresponding to that duration or time stamp, we do so by taking

the product of duration t and the sampling frequency f as shown in equation (1″). We will give MATLAB code for equations (1′) and (1″) below, when we need to compute the time stamp corresponding to some sample number or the sample number corresponding to some time stamp.

$$f = 1/T \tag{1}$$

$$t = n/f \tag{1′}$$

$$n = t^{*}f \tag{1″}$$

Let us apply these notions to finding the duration in msecs of one of our signals TBPOS with the following code. The total number of rows (samples) stored in the field SIGNAL can be found using the length function. We subtract 1 from that and divide the term by the signal's sampling rate SRATE (see equation (1′)). Finally, we multiply this expression by 1000 to convert from seconds to msecs. The reason for the subtraction by 1 is conventional: The first sample should correspond to 0 msecs. More explicitly, the sampling rate for TBPOS is 400 Hz (try data(2).SRATE). This means that a sample is taken every 0.0025 seconds (try 1/data(2).SRATE) or every 2.5 msecs. If length returns 1 as the number of samples in our data file, the formula below would give 0 msec; for 2 samples, the formula would return 2.5 msec; for 3, 5 msec; for 4, 7.5 msec and so on.[3]

```
>> dur = 1000*(length(data(2).SIGNAL)-1)/data(2).SRATE

dur =

   2.3175e+03
```

We now visualize individual sensor positional signals. The TBPOS signal can be plotted by entering the command below. (:,2) selects all rows of the second column which is the position of the signal in y-dimension. Again, data(2) refers to TBPOS. Figure 1 shows the y-signal of TBPOS plotted against sample number. The plot shows the excursion of the tongue back sensor in the y-dimension over the course of time, which is here represented in terms of samples.

```
>> plot(data(2).SIGNAL(:,2))
```

---

[3]Note: In the formula output, MATLAB gives us 2.3175e+03 in scientific notation, that is, 2.3175 multiplied by a power-of-10 scale factor, here multiply by 1000, i.e., 2317.5 msec.

Exercise 3: Create the plot in Figure 1 using the vector matrix `signal_tbp` from above. Also, plot the signal for the x-dimension of `TTPOS` by changing the relevant indices in the plot command for Figure 1. Remember, `data.NAME` gives the trajectory names for the index of `TTPOS`.

As we saw above, the audio information is contained in the first field of the structure array. Applying the `soundsc` function below to the signal infor-

Figure 1: Signal of `TBPOS` in y-diminsion.

mation of audio as the first argument and the corresponding sampling rate as the second argument the sound of the file can be played. `soundsc` scales the values of the audio signal to the range from -1.0 to 1.0, and sends the data to the speaker at the given sample rate 25600 Hz (try, `data(1).SRATE`). The positional values of the mono waveform are the signal of the field audio with only one dimension which is shown in Figure 2 using the `plot` command below. Note here that `SIGNAL`, `SRATE`, and `NAME` are fields contained in every sensor, thus for audio and all trajectories, but they differ in e.g. number of dimensions.

```
>> soundsc(data(1).SIGNAL, data(1).SRATE)
>> plot(data(1).SIGNAL)
```

The x-axis in Figure 2 displays samples. This is less intuitive, therefore the x-axis should be transformed to msec. The `linspace` command like shown below can be used taking three arguments. This function produces a linearly spaced vector with respect to every single row value of the signal involved between 0 as defined below and the length of the signal stored in `signal_len` as the last argument. The signal length `signal_len` divided over the sampling rate for duration shown in equation ($1'$) as second

Figure 2: Mono waveform of the audio array over samples.

argument defines how to linearly space the values. The output is saved in `signal_sec` which is multiplied by 1000 for msec in the plot. Defining the duration of audio signal as the first argument plots the duration on the x-axis and the spatial signal as second argument on the y-axis. The `xlabel` function adds a label *Time (msec)* to the x-axis. The returned plot is shown in Figure 3.

```
>> signal_len = length(data(1).SIGNAL);
```
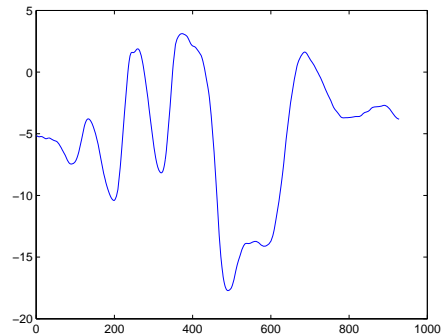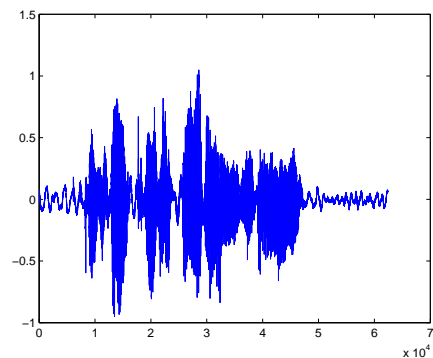
```
>> signal_sec = linspace(0, signal_len / data(1).SRATE, signal_len);
>> plot(signal_sec*1000, data(1).SIGNAL)
>> xlabel('Time (msec)')
```

If the audio signal must be saved as a `.wav` file outside of the `.mat` file, the function `audiowrite` can be applied to the signal shown in the following command.[4] A string must be specified as output name of the `.wav` file. Further arguments are the signal of the audio data array, and the corresponding sampling rate. In this way, the binary `.mat` structure can be converted to an audio file.
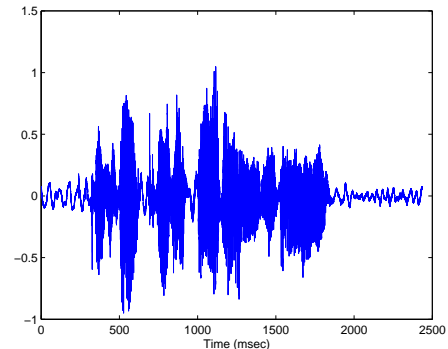

Figure 3: Mono waveform of the audio array plotted over time (msec).

```
>> audiowrite('test.wav', data(1).SIGNAL, data(1).SRATE);
```

Exercise 4: MATLAB might give you a warning here, that can be circumvented by normalizing the amplitude of the signal to 1 like shown in the code below. The normalized audio signal is assigned to `audio_signal`. Normalize the audio signal before saving it to a `.wav` file circumventing the warning from before. You will basically have to apply the function `audiowrite` to `audio_signal`. If you were successful, no warning should appear in the console if previously seen.

```
>> audio_signal =  data(1).SIGNAL/max(abs(data(1).SIGNAL));
```

The signals embedded in the structure array are synchronous. The samples (i.e., each row) have instances in all trajectories, thus, the positional values of the nth sample have representations in `TBPOS`, `TTPOS` and all other trajectories, and in `audio`. The trajectories contain information about the position of the tongue and lip at a particular time (i.e., sample). Therefore, given a particular sample, say 552, the positional values (x, y) for different trajectories can be extracted. For instance, the positional values for the 552 sample in the `TBPOS`, `TMPOS` (i.e., tongue mid posture), and `ULPOS` (i.e., upper lip posture) can be extracted using the respective field numbers 2, 3, and 6 to index the structure array `data`. Row 552 has been indexed for the 552nd sample and all columns : to return both the x- and y-value.

---

[4]Your current MATLAB version might not yet use `audiowrite`. Use `wavwrite` instead which works equivalently.

```
>> data(2).SIGNAL(552,:)

ans =

   39.0359  -13.8055

>> data(3).SIGNAL(552,:)

ans =

   24.5798  -10.5309

>> data(6).SIGNAL(552,:)

ans =

   -8.7424    3.4116
```

Exercise 5: Extract the positional values (x, y) for the 432$^{\text{nd}}$ sample of JAWPOS (i.e., jaw posture), LLPOS (i.e., lower lip posture), and audio. You saw above how to get to the index of the signal fields.

3.2   Local extrema

Imagine you are given a time stamp in milliseconds (e.g., 650 msec) in the vowel /a/ and you are supposed to find the time stamp corresponding to the largest deflection in y-dimension in the region around that time stamp (a local maximum). For the vowel /a/ it is reasonable to concentrate on the TBPOS trajectory. First, we must convert the given time stamp into the corresponding sample index shown in the code below. We start by assigning the relevant time stamp to a variable `msecs`, extract the sampling rate of TBPOS by accessing the field SRATE and assign it to `sRate`, and calculate the corresponding sample index `samps` of the given time stamp which is the product of the duration and the sampling rate as in equation ($1''$) in Section 3.1, divided by 1000 for seconds. The function `floor` truncates any decimal digits of the resulting value to the closest integer (sample numbers can only be integers) and then we add 1 analogue to the explanation in

13

Section 3.1. The output sample index corresponds to a value on the x-axis in Figure 1.[5]

```
>> msecs = 650; sRate = data(2).SRATE; samps = floor(msecs*sRate/1000)+1

samps =

   261
```

Exercise 6: Calculate the sample index of `TBPOS` that corresponds to a time stamp at 1200 msec. Also, calculate the sample index for `audio`. Note that we saw that the sample index of a given value in msecs depends on the sampling rate (see equation ($1''$) in Section 3.1). The sampling rate may differ for `audio` and for other signals.

Let us get back to the problem from above. We are given a time stamp at 650 msec that we converted

Figure 4: Region of `TBPOS` around 650 msec.

to 261 samples. In order to get an impression of the region around this value, we can plot an arbitrary region around 261 for the y-dimension in `TBPOS` as shown in the following code. The rows from 250 to 270 for the second column (y-values) have been indexed. The resulting plot in Figure 4 shows the tongue movement in y-dimension for the `TBPOS` around 650 msec.

```
>> plot(data(2).SIGNAL(250:270,2))
```

Figure 4 shows that the chosen range around the sample number corresponding to the time stamp includes a local maximum. Note that there is no guarantee that this will always be the case. For example, in Figure 5 we plotted the trajectory of samples above the given time stamp with index from row 261 until 270. If a maximum is calculated for the latter region, a value at the left boundary will be returned. This may not correspond to what you would
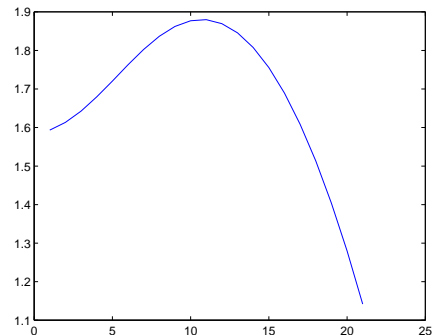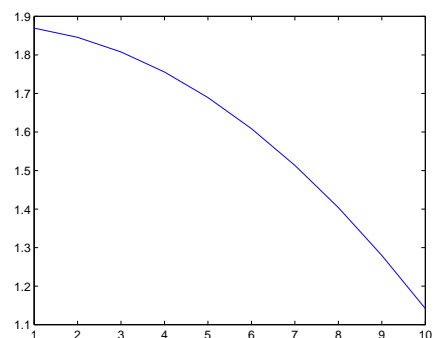
Figure 5: Region of `TBPOS` after 650 msec.

---

[5]Note, MATLAB is case sensitive, which means it will not confuse the field name `SRATE` of the structure array `data` and the vector `sRate`, since the first is capitalized and the latter is not.

intuitively call a maximum. The range of samples chosen for Figure 5 is monotonically decreasing in their y-values, and so the "real" maximum probably occurred at some sample before this range.

```
>> plot(data(2).SIGNAL(261:270,2))
```

Given the sample range that includes the actual local maximum (Figure 4), the maximum position in y-dimension can be calculated in the way shown in the following code. First, a vector `rng` is created containing all sample indices in the range from 250 to 270. All corresponding y-values (second column) in the range `rng` have been index and assigned to the vector `y`. The `max` function in the next line searches for the largest y-value in the vector `y`. Two output arguments have been returned by the `max` function

Figure 6: Region of `TBPOS` around 650 msec.

– the maximum positional value `val` and its index `idx` of the maximum value in the argument values, i.e., not in the entire signal. We are interested in the second output argument `idx` that can only be extracted by also calling the first output argument. Multiple output arguments are assigned to variables by combining these with squared brackets `[]`. The local index of the maximum value `idx` has been given as index to the vector `rng` to get the relevant sample value of the signal that corresponds to the maximum y-value, which is assigned to `idx_new`. Given the sample index of the file, the time stamp of the local maximum can be calculated by subtracting 1 from the sample value (see Section 3.1), divided over the sampling rate to get the duration as in equation $(1')$ in Section 3.1, multiplying it by 1000 for msec. We thus find that the local maximum of the y-value in `TBPOS` occurs at 647.50 msec.

```
>> rng = 250:270

rng =

  Columns 1 through 16

   250   251   252   253   254   255   256   257   258   259   [...]

  Columns 17 through 21
```

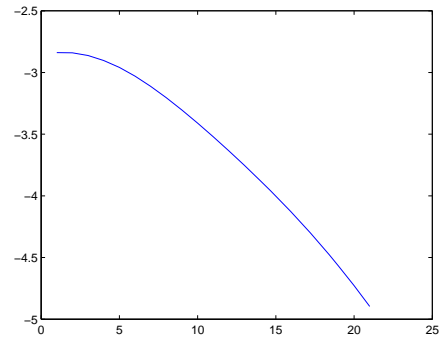15

```
    266    267    268    269    270

>> y = data(2).SIGNAL(rng,2)

y =

    1.5934
    1.6134
    1.6427
    1.6794
...

>> [val idx] = max(y)

val =

    1.8798

idx =

    11

>> idx_new = rng(idx)

idx_new =

    260

>> maxms = 1000*(idx_new-1)/data(2).SRATE

maxms =

  647.5000
```

The region of interest defined above including the relevant time stamp in samples can also be applied to other trajectories. All trajectory fields and the audio field are synchronous in terms of samples. The trajectory `TMPOS` can be accessed by changing the field to 3 as in the code below. The same sample region as for the trajectory `TBPOS` in Figure 4 is shown in Figure 6 for the `TMPOS` trajectory, which shows different spatial configurations for different sensors at the same point in time.

```
>> plot(data(3).SIGNAL(250:270,2))
```

3.3 Velocity signals

Now we come to the calculation of the vertical, horizontal, and tangential velocity as exemplified in the code below. Velocity values are sometimes predefined for some fields. If so this is usually seen in the field name, which should have an `_vel` extension. However, we will show for `TBPOS` how to calculate velocity values. Feel free to calculate the velocity for other trajectories. First, the signal values of `TBPOS` are assigned to `s`. The following formula computes the velocity and stores it in the vector `vel`. The formula differentiates the positional signals using the so-called *central difference approximation* for the calculation of the derivative. This calculation is done on the basis of samples. To go from `vel` to the individual dimension velocities (x, y) expressed in centimeters per second (cm/sec) we proceed as follows. We need to convert from samples to secs and from millimeters (mm) to cm. For the first conversion, the signal needs to be multiplied with the sampling rate of the trajectory `SRATE`, according to equation $(1'')$ in Section 3.1. For the second conversion, in all formulas, we must divide by 10 (`./` below means that every cell entry in the array will be divided by 10) for the unit conversion from mm to cm (1 cm = 10 mm), so the resulting velocity values are in the usual cm per second units. The vertical velocity can be appended to the array of `TBPOS` (i.e., `data(2)`) by defining a new field in it (here, `VEL_y`). Note that only the second column of `vel` is used, since the vertical velocity corresponds to the y-value. The same can be done for the horizontal velocity which is shown in the line after using only using the x-values in the first column.

```
>> s = data(2).SIGNAL;
>> vel = [diff(s(1:2,:)) ; s(3:end,:) - s(1:end-2,:) ; diff(s(end-1:end,:))] ./ 2;
>> data(2).VEL_y =  data(2).SRATE*vel(:,2)./10;
>> data(2).VEL_x =  data(2).SRATE*vel(:,1)./10;
```

The tangential velocity (also known as resultant velocity) is defined as the square root of the sum of the squared velocities in each dimension (see equation (2)). Following this

equation, we can compute the tangential velocity using the command below and appended it to `data(2)` in the field `VEL_tang`, again after multiplying by sampling rate according to equation ($1''$) in Section 3.1 and dividing by 10 for cm per second velocity units. `data(2)` shows the three new fields we have just appended to the `TBPOS` corresponding to the different velocities.

$$v_t = \sqrt{v_x^2 + v_y^2} \tag{2}$$

```
>> v_t = sqrt(sum(vel.^2,2));

>> data(2).VEL_tang = data(2).SRATE*v_t./10;

data(2)

ans =

      SIGNAL: [928x2 double]
       SRATE: 400
        NAME: 'TBPOS'
       VEL_y: [928x1 double]
       VEL_x: [928x1 double]
    VEL_tang: [928x1 double]
```

Exercise 7: Plot the tangential velocity of `TBPOS` in the region that has been used for the local maximum (250 to 270). Also, determine the corresponding local minimum of the tangential velocity. This minimum corresponds to what labeling software of articulatory kinematics considers to be a maximum constriction. You basically will have to change the field `SIGNAL` to `VEL_tang`. Then, calculate the minimum by applying the function `min` instead of `max` (use the `help` function for `min`). You might see that the given range does not include a local minimum (cf. Figure 4 and 5 in Section 3.2). In this case, increase the range and re-plot. If you follow the detection of the local maximum, this exercise can be solved easily.

3.4   Relating spatial and velocity signals

We have seen that in each file one time stamp or sample index has corresponding spatial values in different signals. Signals can be expressed as spatial in the x- and y-dimension

or as velocity in a particular dimension and as tangential velocity. Like introduced in the formula in the previous Section 3.3, we first assign the signal `TBPOS` to a variable `s` for an easier handling and its velocity in terms of samples to `vel`. For now, we keep both signal and velocity in the samples unit.

```
>> s = data(2).SIGNAL;
>> vel = [diff(s(1:2,:)) ; s(3:end,:) - s(1:end-2,:) ; diff(s(end-1:end,:))] ./ 2;
```

Since we want to plot different signals independently in a comparable fashion the `subplot` command is introduced below. To illustrate the correspondence between particular time slices and different signals, the following command plots different signal types of `TBPOS` in the sample range from 150 to 600. Using `figure(1)` a figure object is created. The command `subplot` takes here three arguments. The first determines the number of rows of the output figure and the second the number of columns,

Figure 7: Vertical movement (upper panel) and vertical velocity (lower panel) of `TBPOS`.

such that a 2×1 plot is produced. The last number in `subplot` specifies the position of the plot. The plots are specified after the comma. For both the signal `s` and the velocity `vel`, the y-signal, i.e. the vertical spatial signal and the vertical velocity, is index by `2` for the range between 150 and 600. Thus both vertical signals correspond to the same range in the same trajectory (i.e., `TBPOS`). In the first argument of both plots, the sample range is specified. The output is shown in Figure 7. By default, MATLAB aligns the x-axis starting with 0 (try out, and remove the first argument of the `plot` functions). The upper panel of Figure 7 shows the vertical movement of `TBPOS`. The vertical velocity shown in the lower panel corresponds to the vertical movement. At approximately 380 samples a positive peak in the positional signal (upper panel) is surrounded by a two extrema in the velocity signal (lower panel) with one positive extremum and the other negative and in between the two extrema the velocity cross zero. You can close the open figure by typing `close all` in you MATLAB console.

```
>> figure(1)
>> subplot(2,1,1), plot(150:600, s(150:600,2))
>> subplot(2,1,2), plot(150:600, vel(150:600,2))
```

The same code as for the vertical signals in Figure 7 can be applied to the horizontal signal by changing the column index to `1` which corresponds to the x-values as shown
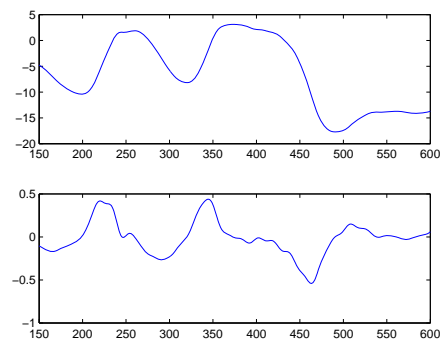
below. Note, the range of the signal is still the same, such as the trajectory. Now, the signal corresponds to the movement in the horizontal dimension. Figure 8 shows the output plot. Again, the upper panel shows the spatial movement in the horizontal dimension, and the lower panel shows the corresponding velocity.

```
>> figure(2)
>> subplot(2,1,1), plot(150:600, s(150:600,1))
>> subplot(2,1,2), plot(150:600, vel(150:600,1))
```

Exercise 8: Create the similar plots like in Figure 7 and in Figure 8 for `TMPOS` in the sample range from 550 to 650. Observe the sensitivity to local extrema under the variation of the sample range. Vary the sample between (e.g., 200 to 700, 300 to 500, 400 to 450). You will have to change to index of the signal `s` assignment and the range values. Hint: You can save yourself some work, by creating a vector, say `range`, that contains the sample slice of interest.
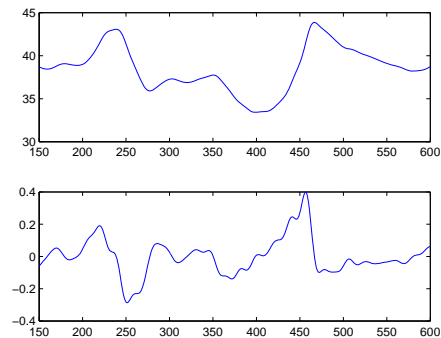

Figure 8: Horizontal movement (upper panel) and horizontal velocity (lower panel) of `TBPOS`.

In Figure 7 and Figure 8, we are still in the realm of spatial movement and velocities in sample coordinates for the x-axis. In the code below, the velocity values will be converted to cm/sec similar to the conversion shown in Section 3.3. The tangential velocity is computed as shown in equation (2) in Section 3.3 and assigned to `t`. All three velocities are multiplied by its sampling rate as in equation $(1'')$ in Section 3.1 and divided by 10 for cm/sec. The sampling rate of the respective trajectory `TBPOS` was assigned to `sRate`.

```
>> sRate = data(2).SRATE;
>> yvel =  sRate*vel(:,2)./10;
>> xvel =  sRate*vel(:,1)./10;
>> t = sqrt(sum(vel.^2,2));
>> tvel = sRate*t./10;
```

Additionally, the x-axis in the plots should be in a time unit (msec). In the code below the sample sequence that we are interested in is assigned to `range` for convenience. A time vector `ms_unit` is created with `linspace` similar to the example shown in Section 3.1. Given the vector `range` it is easier to change the range later on and re-plot the figures without changing the sample slice in each line. Some changes in the `linspace` command

20

must be pointed out: The first argument, which is the beginning of the spaced vector is the first element in range index by `1`. This value is divided by the sampling rate `sRate` for seconds as in equation ($1'$) in Section 3.1. The second argument is the end of the spaced vector specified by the final index in `range` using `end` which returns the last value. The length of the vector `range` is provided as last argument. The results are multiplied with 1000 for msec and assigned to `ms_unit`.

```
>> range = 150:600;
>> ms_unit = 1000*(linspace(range(1)/sRate, range(end)/sRate, length(range)));
```

Figures with the different velocity types can now be created in the code below, introducing a new figure with `figure(3)` and creating a 3×1 subplot like described above. The time vector `ms_unit` created above is the first argument in the `plot` function and determines the values of the x-axis. Since we saved the sequence of interest in the vector `range`, it can be applied to each of the velocity vectors (i.e., `yvel`, `xvel`, and `tvel`). This manner makes it easier to change the range for new plots. The limits of the

Figure 9: Vertical velocity (upper panel), horizontal velocity (middle panel), and tangential velocity (lower panel) of `TBPOS`.

x-axis have been defined for each plot using `xlim` which takes a matrix indicated by the squared brackets that contains the limits, though the beginning (i.e., `ms_unit(1)`) and the end of the x-axis (i.e., `ms_unit(end)`). Titles for each plot are added using the `title` function which takes a string as argument being the title. The x-label can be added using the `xlabel` function which takes a string that specifies the name of the x-axis. Figure 9 shows the returned plot with the vertical velocity in the upper panel, the horizontal velocity in the middle panel, and the tangential velocity in the lower panel. We have looked at the vertical and horizontal velocity before. When it comes to the tangential velocity, both extrema are positive and in between the extrema the velocity dips to a value close to zero (but not necessarily zero; it is never negative though since the tangential velocity entails the squared vertical and horizontal velocities). For instance, we find a positive extremum for the vertical velocity (upper panel) at approximately 800 msec that corresponds to a positive peak in the tangential velocity (lower panel). The zero crossing in the vertical velocity at approximately 1000 msec shows a peak in local minimum in the tangential velocity and the negative extremum in vertical velocity peaks at 1200 msec, which corresponds to a maximum peak in tangential velocity.
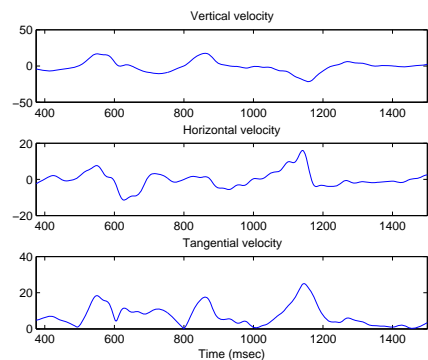
21

```
>> figure(3)
>> subplot(3,1,1), plot(ms_unit, yvel(range))
>> xlim([ms_unit(1), ms_unit(end)])
>> title('Vertical velocity')
>> subplot(3,1,2), plot(ms_unit, xvel(range))
>> xlim([ms_unit(1), ms_unit(end)])
>> title('Horizontal velocity')
>> subplot(3,1,3), plot(ms_unit, tvel(range))
>> xlim([ms_unit(1), ms_unit(end)])
>> title('Tangential velocity')
>> xlim([ms_unit(1), ms_unit(end)])
>> xlabel('Time (msec)')
```

Exercise 9: Repeat Exercise 8 for the different velocity signals of the code for Figure 9. Use `TMPOS` and the time slices in Exercise 8, compute the velocity signals in cm/sec and change the x-axis to msec. Re-plot `figure(3)` for different time slices determined in a vector `range`.

3.5   Storing labels

Having extracted a time stamp from a trajectory, the relevant information can be stored in a new structure array that will be called `labels`. The code below shows how to store information in a new structure array. For the first entry in the new structure array, `labels` must be indexed with 1. This new structure array has its own fields that you as the researcher will find useful. These field names are specified subsequently of `labels(1)` using a dot. You can have as many fields as you find useful. As fields for our structure here, we will use names that are mnemonic of the information we extracted in the example task in Section 3.2. For instance, the field `file` stores the name of the file that has been stored in the variable `names` (Section 2) or can be assigned directly as a string '`cascade_09_0823`'.

```
>> labels(1).file = names{1};
>> labels(1).trajectory = trajectories{2};
>> labels(1).name = 'MAX_C';
>> labels(1).timestamp = maxms;
>> labels(1).signal = 'y';
```

```
>> labels(1).phone = 'a'

labels =

          file: 'cascade_09_0823'
    trajectory: 'TBPOS'
          name: 'MAX_C'
     timestamp: 647.5000
        signal: 'y'
         phone: 'a'
```

The output shows the information that we have assigned to the structure array, e.g., the file name, the trajectory, the name of the landmark, the time stamp etcetera.

If a new hypothetical value is to be appended to the same structure array, increase the index to the next value. In the case below, we use 2, but if you are not aware of how many values are contained in a structure array you can name the index **end**. This appends the new information to the end of the structure array. As you will see in the output (type `labels`), the structure array has now the size 1×2, since we have added one information point.

```
>> labels(2).file = names{1};
>> labels(2).trajectory = trajectories{2};
>> labels(2).name = 'MAX_C';
>> labels(2).timestamp = 652.5;
>> labels(2).signal = 'x';
>> labels(2).phone = 'a'

labels =

1x2 struct array with fields:
    file
    trajectory
    name
    timestamp
    signal
    phone
```

23

If you are interested in the information that you have just stored in the structure array, it can be accessed with the index 2 like shown below. Also, you can access only particular information like the label's name by adding the field `name` to the index structure array (e.g., `labels(2)`).

```
>> labels(2)

ans =

          file: 'cascade_09_0823'
    trajectory: 'TBPOS'
          name: 'MAX_C'
     timestamp: 652.5000
        signal: 'tang vel'
         phone: 'a'

>> labels(2).name

ans =

MAX_C
```

Exercise 10: Given the results that you gained from Exercise 7, append your information to the structure array `labels`. Use `end` instead of increasing the index of `labels` as described in text above. The value for the field name `signal` must be something like `vel_tang` (i.e., tangential velocity) and obviously the value for the field `timestamp` is the time stamp that you determined in Exercise 7. If you are done with this, type `labels` in the MATLAB console, and a 1×3 structure array should be returned.

References

Attaway, S. (2012). *MATLAB: A practical introduction to programming and problem solving.*

Gilant, A. (2011). *MATLAB: An introduction with applications*, volume 4. Hoboken, NJ.

Quarteroni, A., Saleri, F., and Gervasio, P. (2010). *Scientific computing with MATLAB and Octave*, volume 3. Springer, Heidelberg.

Rosenbaum, D. A. (2012). *MATLAB for behavioral scientists.* Routledge, Mahwah, NJ.