# Chapter 1. Introduction

The TSlib toolbox is a suite of open-source Matlab™ functions and Graphical User Interfaces (GUIs) for organizing and analyzing time-stamped event records. By ‘time-stamped event records’ (tsdata, for short) we understand a two column array in which the first column is a time stamp and the second is an integer event code. Each row in the array specifies the time at which the event specified in the second column occurred. The time stamp is a positive number in arbitrary units (that is, in ticks of the clock used to measure time elapsed in the experimental setup; this varies from lab to lab). It is always greater than or equal to the immediately preceding time stamp. The event code numbers must be positive integers and may range from 11-99999.

Variables names (readily understandable textual names like “PokeOn” or “Feed”) are associated with the event code numbers to make it easier to refer to events when writing data-analysis code using the commands in the TSsystem toolbox.

In a typical animal cognition experiment, subjects are placed in the test boxes each day for a session lasting as little as a few minutes to as much as a few hours. However, in the “fully-automated” system developed in the Gallistel lab, the animals live in the test environment throughout testing. They may be tested with several different protocols within single sessions lasting several weeks.

In either case, proprietary process-control code that controls the experimental apparatus and logs the events (for example, MedPC™) generates a separate data file for each subject for each session. (In the fully-automated system, this data file is read repeatedly while the experiment is in progress.) Groups of sessions are associated with a subject. Together, the subjects and their associated sessions form an experiment. A core feature of the TS toolbox is that the raw data for every session for every subject are stored in a single Matlab structure. The commands in the TS toolbox assume the existence of this structure as a global[[1]](#footnote-1) variable in the Matlab workspace. They operate on data contained in fields within this structure, and they output the results of their operations into other fields within this same structure. The results derived from the raw data cannot become separated from the raw data, because the results and the raw data are all together in a single complex, hierarchical data structure. This insures a secure data trail, a trail from the summary results and graphs that appear in publications back through all the intermediate calculations to the raw data.

The simple raw-data text files generated by the control software also contain additional information about a session, such as which subject was run, the time and date when the session started, in which box the subject was tested, using which experimental protocol, etc. This information is usually in a header at the top of the text file (see Figure 1), but some or all of it may reside in the file name. There are TSloadsession commands for several different raw data formats. If none of these matches your format, contact C.R. Gallistel for help in creating a program that will load your raw data into the system.

Included with this manual is a folder called Examples. It contains raw data from two simple experiments, one run with pigeons conducted in the laboratory of Timothy Shahan at Utah State University, and one run with rats in the Balsam laboratory. These data have been loaded into a experiment structures and analyzed using the functions in the TS toolbox. The Examples folder also contains data from an experiment run in the Gallistel laboratory in collaboration with Valter Tucci at the Italian Institute of Technology in Genoa and Pat Nolan at the Harwell MRC in Oxfordshire. {Not yet in folder.} This experiment was run under fully automated conditions, with the raw data being read into the experiment structure automatically at regular intervals and analyzed in quasi real time. We refer the new user to these examples at many places in this Manual. Thus, in working through this manual, the user should put the TSlib folder and its subfolders on Matlab’s search path, so that s/he may study the examples and try variations in the commands, thereby learning to use this toolbox.

**The structure of raw data files**

Raw data files that are to be loaded into the Experiment structure for analysis by TS Toolbox commands are typically organized in the following format. There is a header that contains identification information about the session. The header is followed by the two-column TSdata. Here is an annotated example

**Header**

6   \This is the start month of the session, 6/25/08

25 \This is the start day

08 \This the start year

6   \This & the following two are the end date of the session, 6/26/08

26

08

1    \This is the subject’s ID number

102    \Laboratory’s Experiment ID number

1    \Experimental ’Condition’ (or ‘Phase’ or ‘Group’) Identification Number

4    \Box Identification Number

13    \Start Time of session in hours minutes seconds military format 13:10:20

10

20

**TSData**

*Time Event*

1.96 113 In this example, time is specfied decimally in seconds.

1.98 115

82.78 1013

82.84 1003

82.92 1013

82.96 1003

82.96 1013

83 1003

83.1 1013

83.18 1003

84.02 1013

84.04 1003

84.04 1013

84.06 1003

**Figure 1.** *Annotated (and slightly schematized) portion of a raw data file. It is assumed that there is one such file for each session for each subject.*

## Installing TSlib

The Toolbox may be downloaded from GitHub

## Stylistic Notes

We adhere to the following conventions:

Underlined headings are TSlib functions.

The terms “group” and “phase” are be used interchangeably to refer to experimental conditions (distinct protocols). Each group/phase describes the particular experimental condition in force during a session. Fully automated sessions may include several phases; that is, the subject may be tested with several different experimental protocols in the course of single multi-day session.

## Guide to the Initial Chapters

**Chapter 2**: *The Experiment structure*. Explains the Matlab structure named ‘Experiment’ that is at the core of the system. The functions in the Toolbox operate on fields in this structure and put the results of their operations into other fields within the same structure. The path to this structure must be on Matlab’s search path.

**Chapter 3**: *The Experiment Browser*. The graphic user interface that enables the user to browse the structure. An indispensible aid to data analysis and visualization using the TS system.

**Chapter 4**: *Creating, saving and loading Experiment structures*

**Chapter 5**: *Loading data into an Experiment structure*. This is the essential first step; if the data are not in an Experiment structure, the Toolbox functions cannot be used to analyze them.

**Chapter 6**: *Creating a data-analysis script*. A script is a sequence of Toolbox commands (functions) that analyzes the data. Each command takes the statistics (or data) in one or more fields of the Experiment structure as input, computes other statistics from those data, and puts the results in one or more other fields in the Experiment structure.

**Chapter 7**: *Working with event codes*. Several basic Toolbox commands refer to events or sequences of events. The events are represented by integers in the raw data (numerical event codes). A dictionary maps textual event codes to the corresponding numerical codes enabling the user to refer to events by intelligible names. This enormously increases the ease of writing code and the intelligibility of already written code.

**Chapter 8:** *Searching**for event sequences*. Analyses generally begin by searching for single events or sequences of events. The events in a searched-for sequence need not be—and typically are not in fact—contiguous. This chapter explains the generally hidden workhorse function, **TSmatch**, that does the searching.

**Chapter 9**: *Computing**with found sequences*. First-order statistics result from computations performed on the sequences found by TSmatch, for example, the computation of the durations of the intervals between two events. This chapter explains the command, **TSparse** that, more often than not, does these computations.

**Chapter 10:** *Computing session-level statistics*. Computing statistics such as the times of occurrence of every feeding or every poke or peck or lever press in a session (and considerably more complex statistics) using **TSsessionstat**.

**Chapter 11:** *Defining trial types and computing trial statistics*. Often, one wants to analyze events or event sequences that occur repeatedly within other larger-scale sequences, for example, the interpeck intervals within successive trials of an autoshaping experiment. (There are typically many pecks, hence many interpeck intervals during each trial). This chapter explains how to define the larger-scale sequences (the sequences that constitute a type of trial), using the function **TSdefinetrialtype**, how to control which type of trial is the currently active type, using **TSsettrialtype,** and how to compute statistics based on recurring sequences within the trials of the currently active type, using **TStrialstat**.

**Chapter 12**: *Computing statistics from statistics*. Many statistics are computed not directly from the raw data but rather from other already computed statistics. For example, the pecking or poking or lever pressing rate in a session (a higher-order statistic) is computed from the number of pecks and the duration of the session (lower order statistics). This chapter explains the function **TSapplystat**, which computes one or more higher order statistics from one or more lower order statistics.

**Chapter 13:** *Compiling statistics*. To compile statistics from lower levels (e.g., the trial level or session level) into fields at higher levels (e.g., the subject or experiment level), which contain, for example, the poking rates from every trial of a given type in a given session, or across all sessions for a given subject, one uses **TScombineover.**

**Chapter 14:** *Limiting the scope of operations*. Many analyses only make sense when conducted on a subset of the data or the available statistics. The **TSlimit** command limits the scope of application of subsequently called functions, such as TStrialstat, TSsessionstat and TSapplystat. One can limit the scope to a range of trial index numbers, to a range of session index numbers, to a range of subject index numbers, or on the basis of the phase (experimental condition) during a given session.

The functions described in Chapters 8 through 14 are the key data-analyzing functions: TSmatch, TSparse, TSsessionstat, TSdefinetrial, TSsettrialtype, TStrialstat, TScombineover and TSlimit. When you understand the uses of those 8 functions, you can write scripts that analyze your data with surprisingly few commands, because these commands are very powerful. There are many other functions in the TS Toolbox, but they are of secondary importance. They are not the workhorses. In learning to use the Toolbox, concentrate on the 8 workhorses.

**TSfunctions Organized by Function**

**Creating, Loading and Saving Experiment Structures**

TSinitexperiment - Creates a new Experiment Structure

TSloadexperiment - Loads an Experiment Structure mat-file & writes dictionary i

TSsaveexperiment - Saves an Experiment Structure to mat-file

TSbegin - Leads the user through the process of creating an experimental structure for a fully automated experiment

TSstartsession - Leads user through process of starting a fully automated session

L**oading Data & Code Programs**

TSloadsessions - Loads in MED-PC data files

TSsetloadparameters - Sets several loading parameters

TSsetoverwritemode - Sets the overwritemode value

TSloadMEDPC - Loads Gallistelab formatted raw data files

TSload2ColAndHeader -Loads 2-col text data files

TSloadsstdxls - Loads data from Excel file, assuming they are in 1st 2 cols

TSloadcsv - Loads data from csv file, assuming they are in 1st 2 cols

TSloadprograms - Loads experiment-control (e.g., MedPC) code files into the Experiment.Programs structure

**Creating trials**

TSdefinetrialtype - Creates a trial definition

**Editing Data**

TSedit - Modifies or inserts TSdata codes found by TSmatch

**Finding Event Sequences**

TSmatch - Searches TSdata for matches to user-specified event sequences

**Computing statistics**

TSparse - Computes user-specified statistics from event sequences found by TSmatch; ; often called from TStrialstat & TSsessionstat

TSsessionstat - Computes session-level user-specified statistics taking TSData as input and putting results into user-specified field

TStrialstat - Creates trial-level statistics by finding user-specified sequences within trials and computing stats from those sequences using user- specified function; puts results in user-specified field

TSapplystat - Computes user-specified statistics from data or statistics in one or more fields in the structure and puts results into one or more new fields

TScombineover - Creates fields at a higher level by combining statistics from lower-level fields

**Controlling the data/statistics operated on**

TSlimit - Limits what subjects, sessions, phases, trial numbers are active (to- be-operated-on)

TSsettrialtype - Makes a specified trial type the active trial type

TSsetdata - Sets the active data field

**Handling Event Codes**

TSdeclareeventcodes - Call this to declare the event codes as global and use them

TSimporteventcodes - Imports a text readable list of codenames and values

TSexporteventcodes - Exports event codes from the Experiment to a text file

TSsetdefaulteventcodes - Sets the TSlib default event codes when no Experiment is loaded

TSaddeventcodes - Adds one or more event codes to the current Experiment

TSrmeventcodes - Removes one or more event codes from the current Experiment

**Graphing**

TSraster - Creates a raster plot of Time Stamped Data.

TSplot - Creates labeled x-y line and scatter plots, with multiple panels

TSplotcdfs - Creates labeled plots of cumulative distributions, with multiple panels

TSplotcumrecs - Creates labeled plots of cumulative records, with multiple panels

**GUI Tools**

TSexperimentbrowser - General browsing tool for the Experiment

TSrastergui - GUI interface to TSraster, makes raster plots

**Miscellaneous**

TSrmfield - Recursively removes fields from the structure

TScheckconsistency - Checks the consistency of the structure.

TSaddsubjects - Allows the user to add subjects to the structure by answering the prompts

TSorderfields - Changes the order of the fields at a given level

TSrenamefields - Renames fields

TSremovesession - Removes a session given Subject ID# & session #

TSFunctions -Displays a list of the TS functions in the workspace

TSaddprograms -Puts MEDPC (process control) programs into a field in the Experiment structure

TSchangeprotparameters -Changes the parameters in a running MedPC program

TSdeletesubject -Deletes a subject from the Experiment structure

TSeventname -Finds the name of an even given its numerical code

TSloadscript -Loads into a field in the Experiment structure the script that generated the structure

TSexportscript -Writes the script from the field in the Experiment structure to a .m file

TSorderevents -Reorders near-simulataneous events recorded in wrong order in raw data

TSorderfields -Changes the ordering of the fields at a given level in structure

TSstriptoraw -Strips Experiment structure down to only the TSData fields and other  
pre-determined field, removing all the fields created by a script

# Chapter 2. The Experiment Structure

The TSlib functions and graphical user interface (the Browser) always work with a Matlab structure named Experiment, which must be a global variable in the Matlab workspace. A *structure* in Matlab is a hierarchical data structure with named fields. A field may contain any kind of data: just a single number; a row or column of numbers; an array of numbers; a cell array; some text; or even other structures. The name of a field is a guide to its contents; it serves the same function as a column header on a spreadsheet. The Experiment structure contains all information about the subjects, the sessions of time-stamped data, a number of settings that control how toolbox commands operate, a number of fields that contain statistics that toolbox commands compute automatically (without being told to do so by the user) and many other fields that contain statistics generated by Toolbox commands that the user has invoked. The names for these fields are specified by the user. In short, the Experiment structure contains all the information pertaining to one experiment, in accord with the “keep it all together” principle that motivated this aspect of the design of the Toolbox.

There is a graphical user interface (GUI), which may be called by typing TSexperimentbrowser. It is your indispensible friend when using the TSsystem. You should always call it as soon as you begin to work with an Experiment structure. Hereafter, we refer to it simply as ‘the Browser.’ The Browser enables the user to browse the Experiment structure, output data to the Matlab workspace, make plots of data, and edit data (see Chapter 3).

In this chapter, we explain the Experiment structure. An understanding of this structure is essential for any use of the Toolbox.

## The Experiment level

The highest level of the structure is the Experiment level. Each field at this level is of the form 'Experiment.<fieldname>', for example ‘Experiment.MeanTrialsToAcq’ (a plausible name for a field containing the mean trials to acquisition for each group of subjects). The information contained in fields at this level is pertinent to the entire experiment. The Experiment level always contains the following fields (Figure 2):



**Figure 2.** *The top level of the Experiment hierarchy. Fields shown in black are terminal fields. A terminal field may contain numerical data or text or even a cell array, but it does not contain a Matlab structure. That is, a terminal field does not have its own subfields. Non-terminal fields are shown in red. They contain Matlab structures. A Matlab structure contains its own subfields. Thus, a non-terminal field is a structure within a structure. If a subfield is indexable, it ends with a number in parentheses (for example, Subject(1)). Terminal fields shown above the red fields in this figure are created automatically by the Toolbox function(s) that create the Experiment structure. They appear in every Experiment structure. Terminal fields below the red fields are created by Toolbox commands called by the user. They have user-assigned names. In most basic Toolbox commands, the first argument is the user-chosen name of the field into which the function will put the results that it computes.*

You can display the contents of a field at the Experiment level by typing, for example, Experiment.StartDate in the Matlab workspace and hitting return. The contents of a field may also be viewed using the Browser (type TSexperimentbrowser), which is a graphical user interface that allows the user to browse the complex Experiment hierarchy. (It is explained in detail in Chapter 3). What follows is an explanation of the red fields in Figure 2. These are the structure-containing fields at the Experiment level.

### **The Experiment.Info structure**

The fields of the Info structure contain information for internal use by the Toolbox functions. As with any field in the Experiment structure at any level, the user *can* set the contents directly, either by commands in Matlab’s workspace or using the Browser. However, we urge the user not to do this except under unusual circumstances and when the user is confident that she understands the consequences of altering the contents of these fields. Ordinarily, the user alters the contents of these fields only by using Toolbox commands intended for that purpose, such as, for example, TSlimit (see Chapter 14).

### **The Experiment.EventCodes structure**

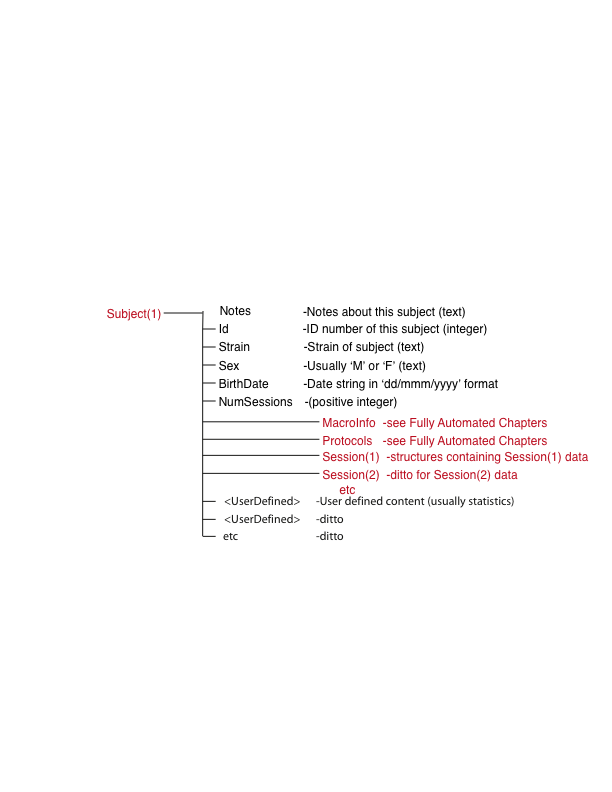
The fields under Experiment.EventCodes pair textual event identifiers (e.g., ‘PokeOn’ or ‘LightOn’ or ‘Feed’) with the corresponding numerical event codes that refer to those events in the raw data files (see Listing 2 in Chapter 7 for examples). These numerical event codes are specific to the experiment being conducted, so the user must specify them. Event Codes are discussed at length in Chapter 7. In this structure, the textual event codes are the field names and the content of each field is the corresponding numerical code for that event. When using Toolbox commands, the user refers to events by their names rather than by the corresponding numerical code. This makes the data-analysis code MUCH easier to comprehend. It relieves the user of the burden of remembering a long list of arbitrary numerical names for events.

### **The Experiment.Programs structure**

This Experiment-level structure is only relevant for users of the fully-automated system, which automatically reads into the Experiment structure the process-control code (e.g., MedPC™ code) for each protocol used within a given experiment. The Programs structure indexes each distinct process-control code file. Every Med PC program file that is active at any point for any animal throughout the experiment is stored here when it first becomes active (that is, at the first session controlled by that code). If the user is not using the fully automated process, this field may be left empty. However, we strongly recommend that the process-control code files be loaded into the structure. The principle behind the use of a single Matlab structure to store *everything* is “keep it all together.” One wants every bit of information relevant to an experiment to be stored in one place, so that the different kinds of relevant information do not become separated in the course of preparing the results for publication and archiving the results. The process-control code for the experimental protocols is highly relevant information for anyone who wants to replicate an experiment or who wants to know in minute detail exactly how an experimental protocol was implemented. The published descriptions rarely describe the minute details of the implementation.

## The Experiment.Subject(#) level (aka the ‘Subject level’)

Each field at the Subject(*#*) level is of the form Experiment.Subject(*#*).<fieldname>. The *#* indicates which subject (by index) is referred to. All fields subordinate to a numerically indexed subject contain data only from that subject. An Experiment.Subject(*#*) structure always contains the following fields:



**Figure 3.** *The structure of a Subject structure. A Subject(#) field at the Experiment level is itself a complex hierarchical structure, with several levels of subordination. These Subject fields are indexed, meaning that there are typically many instances of a Subject field (Subject(1), Subejct(2), etc), each one of which contains the same complex substructure here diagrammed. The terminal fields (black font) above the red fields are created when the Experiment structure is created. They appear in the Subject structures of every Experiment structure.* *Terminal fields below the red fields are created by Toolbox commands called by the user. They have user-assigned names. The red fields contain structures that have still further subfields.*

You can display the contents of a field at the Subject level by typing, for example, Experiment.Subject(1).BirthDate in the Matlab workspace and hitting return. The contents of fields at this level (or any level!) may also be viewed using the Browser (see Chapter 3).

### **The Experiment.Subject(#).MacroInfo structure**

The structure that goes in this field is only used when the process is fully automated. If you are not using the fully automated procedure, this field may not exist. See Chapter ? for an explanation of this structure and its function in the fully automated system.

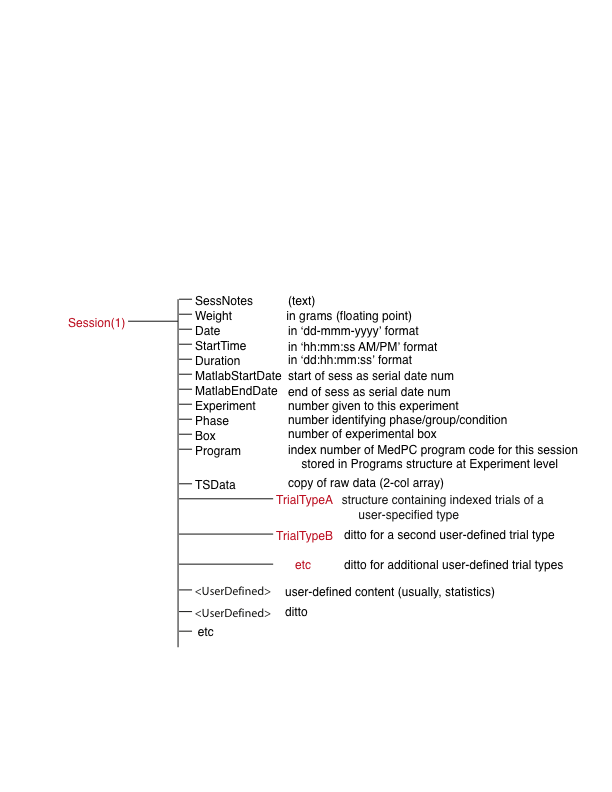
### **The Experiment.Subject(#).Protocols structure**

The Protocols level is only used for the fully automated process. The contents of this field are not needed if the program is used for non-real-time data analysis. See Chapter ? for an explanation of the role that the information in this field plays in the fully automated system.

FOR ME: Only relevant for fully automated and when matlab is deciding to go from one phase to the next- at least as many programs as there are conditions- once the mouse has eaten 100 pellets, done 100 trials, etc.- put all the different codes in 1 med pc file and have a flag that controls which code is being used. Flags are the last thing in the array of parameters. Matlab talking to Med PC while Med PC is running- write to the text file “background” command in med pc that makes it execute a Pascal program- guarantee that a certain output from an input won’t happen after a certain time (time constraint) (drag the person off the stage if they run over)- no open ended things- no loops- keeps going when the Pascal program is called, and then the Pascal program reports back- the array gets overwritten so it can switch the flag to being done. The list of numbers/parameters/flags as a whole counts as a protocol- together they define the experimental condition when fed to the code

## The Experiment.Subject(#).Session(#) level (aka the Session level)

Each field at the Session level is of the form Experiment.Subject(#).Session(#).<fieldname>. As the name suggests, the information at this level is specific to a single session for a single subject. The Experiment.Subject(#).Session(#) level contains the following fields*:*



**Figure 4.** *The structure of a Session(#) structure at the Subject(#) level. Session(#) fields at the Subject level are indexed, that is, there may be many Session(#) fields subordinate to each Subject, depending on how many sessions were run with that subject. Session(#) fields themselves contain complex structures, the fields of which are shown here. The terminal fields (black font) above the red fields appear in every Session structure.* *These fields* *are created when the data file for a session is loaded into the Experiment structure using the TSloadsessions command. Terminal fields below the red fields are created by Toolbox commands called by the user. They have user-assigned names, which names indicate something about their contents. The red fields are fields that themselves contain structures, that is, they are fields with subfields. These Trial-Type Structures are created only when the user runs commands that compute statistics for a type of trial that the user has defined. Trial types are explained in Chapter 11.*

You can display (or operate on) the contents of a field at the session level by typing, for example, Experiment.Subject(3).Session(2).Duration in the Matlab workspace and hitting return. The contents of fields at this level (or any level!) may also be viewed using the Browser (see Chapter 3).

## The Experiment.Subject(#).Session(#).Trial<trialname> level (aka the Trial-Type level)

This level distinguishes between the different user-defined trial types. A trial type is a sequence of events that uniquely define a recurring chunk of time-stamped data, within which the user wants to search for event sequences. Example 1: the sequence [Feed Feed] defines an inter-reinforcement interval (IRI) as a type of trial. All the time-stamped events falling between two successive occurrences of the Feed event constitute one instance of this trial. Example 2: the sequence [LightOn Feed LightOff] defines an autoshaping trial type in which each trial begins with the light coming on in the feeding hopper followed after some time by the dispensing of a pellet and the simultaneous turning off of the light in the hopper. Again, all the time stamped data falling between the LightOn event and the LightOff event constitute an instance of this trial type. We name this trial type ‘CS’ (short for conditioned stimulus). Example 3: the sequence [LightOff LightOn] defines as a type of trial the interval between the end of one trial and the beginning of the next trial, that is, an Inter-Trial-Interval (which we name ‘ITI’). As this last example makes clear, "trial" in the TS system is generic. It refers to any recurring pattern of events within which one wishes to look for other patterns (that is, for within-trial sequences).

Each trial type is assigned a user-chosen trial name, denoted <trialname>. Thus, the different *trial types* defined by the user are distinguished by their names- *not by index numbers*. On the other hand, recurring instances of a given type of trial are distinguished by index numbers. For example, the above referred to type of trial defined by the sequence [LightOn Feed LightOff], which we named ‘CS’ is distinguished from the type of trial defined by the sequence [LightOff LightOn], which we named ‘ITI’ by the difference in the two field names: ‘TrialCS’ and ‘TrialITI’, whereas subordinate to each of these two trial types, one will find Trial(1), Trial(2), etc, which refer to different instances of that trial type.

Each field at the Trial<trialname> level is of the form Experiment.Subject(#).Session(#).Trial<trialname>.<fieldname>. *Notice that, for technical reasons, the system automatically prepends ‘Trial’ to the user-specified trial names.*

The trial-type level contains the following fields:

**NumTrials**- The number of trials of this type in this session (an integer). [This is a terminal field]

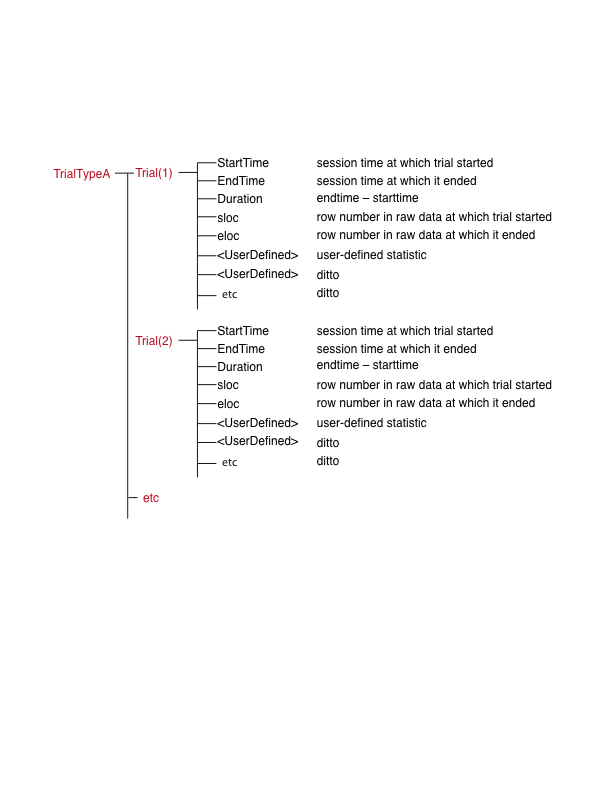
**Trial**(#)- Trial *structures* each containing statistics from one instance of a trial of this type (see below)

## The Experiment.Subject(#).Session(#).Trial<trialname>.Trial(#) level (aka the Trial level)

Beneath each user-defined trial type are some number of indexed trial structures (Trial(#))

Each field in a trial structure is of the form Experiment.Subject(#).Session(#).Trial<trialname>.Trial(#).<fieldname>

An Experiment.Subject(#).Session(#).Trial<trialname>.Trial(#) structure contains the following fields:



**Figure 4.** *Each Trial structure contains the subfields shown above. All of the fields in this level are terminal (black font), meaning that fields at the Trial level do not contain structures. Put another way, the fields subordinate to a Trial(#) field are the bottom of the hierarchy. All of these fields are created when the user defines a trial type and then uses TStrialstat to compute a statistic for trials of the defined type. The user can use the Browser to add fields containing text (e.g., notes about an unusual trial)*

You can display the contents of a field at the trial level by typing, for example, Experiment.Subject(3).Session(2).Trial<trialname>.Trial(4).Duration in the Matlab workspace and hitting return. The contents any field at this level (or any other level) may also be viewed using Browser (see Chapter 3)

## Matlab’s search path

As this chapter has made clear, when you begin to use the TSlib system, the first thing you do is either create an Experiment structure or find an already created structure. In either case, it is essential that you identify the complete path to the folder that contains the Experiment structure that you want to work on. You must also identify the complete path to the folder containing the TSlib software. And you must make sure that both of these are on Matlab’s search path. Whenever you call a function (aka command) in Matlab, whether it is native to Matlab or a function in TSlib, or a custom helper function that you have created (about which more later), or whenever you ask Matlab to access a file (with, say, a load file or read file command, Matlab searches through all of the folders on its search path. The command you have issued will only work if Matlab finds on its search path the function you have called or the file you are attempting to read or load. For the beginning user, this may pose two challenges: 1) Learning how to set Matlab’s search path so that it includes the folders you want it to include. 2) Finding the complete path to the folders you want to have on Matlab’s search path.

### Setting Matlab’s Search Path

Open Matlab and select the Home tab and locate the Set Path icon. Click on this icon to bring up the dialog box that enables you to browse your computer’s folder structure to find a folder that you want to add to the path. When you find the folder, select it and then ask yourself whether you want to add ONLY the contents of that folder NOT the contents of whatever subfolders it may contain, or you want to add the contents of that folder and all of its subfolders (and their subfolders ad infinitum) to the search path. If the former, click on Add Folder; if the latter, click on Add with Subfolders. (In older versions of Matlab, you access Search Path through the File pull-down menu at the top left of Matlab’s window.)

### Finding the Folder That Contains A File

If you have just created or are about to create an Experiment structure, you will usually want to save it inside a folder that you create specifically to hold all the files relevant to your experiment. You can create and name a new folder for the experiment either, outside Matlab by using the basic commands of your computer operating system or, within Matlab, by right clicking on the Current Folder heading for the part of the window that shows the contents of the current folder, which brings up a menu that includes the option of creating a new folder. Or you can use Matlab’s file browser to browse the folder structure of your computer until you find the folder that contains the already existing Experiment structure. When you select a folder that you have found using Matlab’s browser, it becomes Matlab’s “Current Folder.” The Current Folder is always on Matlab’s search path. However, if you do not add it to the search path, then if you leave it for some reason—that is, if you make some other folder the Current Folder—then Matlab will no longer find the files in the folder you were working within. If you type savepath in Matlab’s command window before leaving the folder you have found, you will add that folder to Matlab’s search path. To see the complete path to the Current Folder, type cd.

# Chapter 3. The Experiment Browser

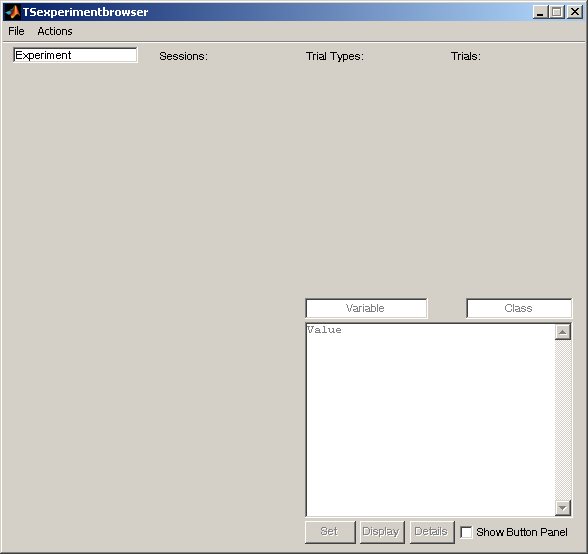
## Introduction

The Experiment Browser is a graphical user interface (GUI) that allows the reader to browse the large and complex hierarchy that is the Experiment structure. You use it whenever you are using the Toolbox to analyze and graph your time stamped data. Because you will use it so often, we suggest that you put a Shortcut button in the space above Matlab’s Command Window. To do that, type the following Browser-opening command into Matlab’s workspace and hit return:

TSexperimentbrowser

Like any just-executed command, the command will appear in Matlab’s Command History window. Select it and drag it to the Shortcut space immediately above Matlab’s Command Window. This creates a shortcut button. Clicking on this button will open the Browser.

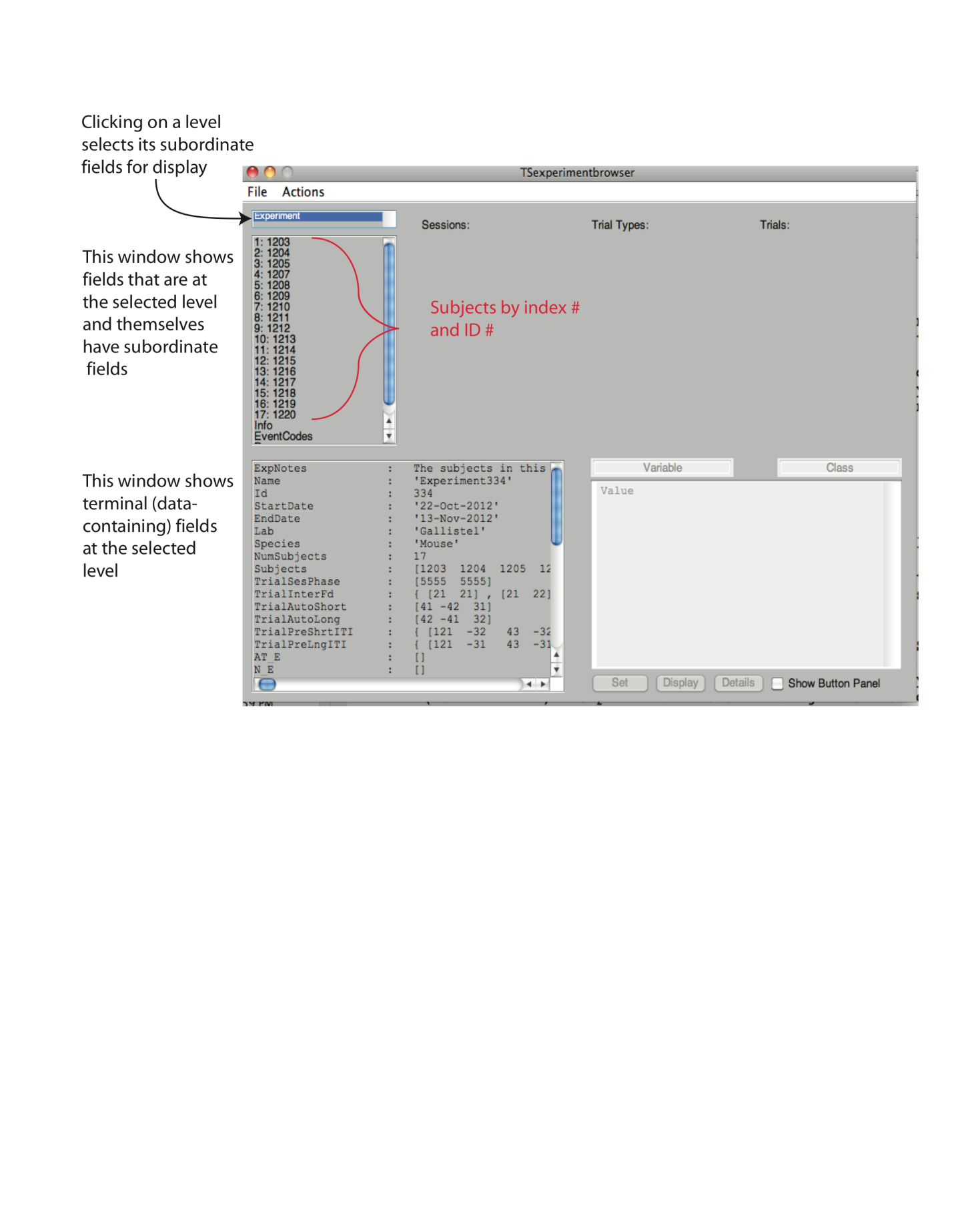
When the Browser opens, the first thing you see is in Figure 3-1. If you have not yet created your own Experiment structure or loaded one of the sample Experiment structures into the Matlab workspace—in other words, if there is no ‘Experiment’ variable in the workspace—then when you click on the Experiment text in the white rectangle at upper left, all you will see is a message in the GUI saying that “Experiment Structure does not exist.” You will also see a message in Matlab’s workspace, saying “There is no experiment structure in the workspace.” Before going any further, load your own Experiment structure or one of the examples into the workspace. The examples are in the Examples folder. You can load one of them by typing TSloadexperiment in the Matlab workspace, which causes a file-browsing GUI to appear, which allows you to select the example you want to load into the workspace.



**Figure 3-1.** *The Experiment Browser GUI when it is first pulled up (nothing has been selected yet).*

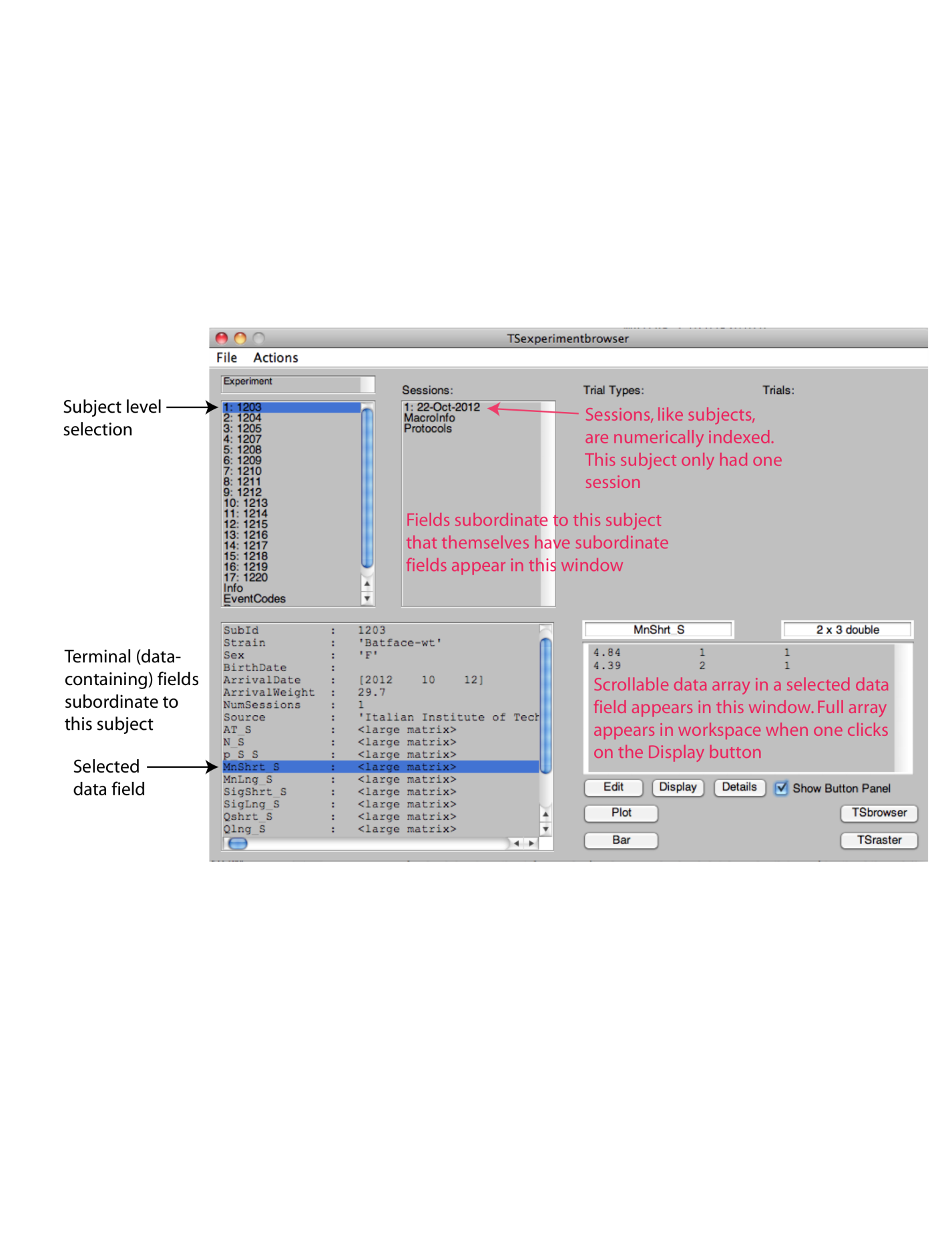
**Clicking Your Way Down the Hierarchy**

When there is an Experiment structure in the workspace, click on “Experiment” in the rectangle at upper left of the Browser. Two panes will now open within the GUI, one beneath the other (Figure 3-2). The top pane displays the non-terminal fields at the Experiment level. Recall that a non-terminal field is itself a structure. Put another way, a non-terminal field gives access to other fields still lower down in the hierarchy. The bottom pane of the Browser shows the terminal fields. Recall that terminal fields always contain data of some kind, either an array of numbers, or a cell array, or some text. Among the user-defined terminal fields at the Experiment level are the fields giving the event sequences that constitute the definitions of the user-defined trial types. The name that the user has given to a trial type (e.g., ‘AutoShort’) is always prefaced by ‘Trial.’ In the example in Figure 3.2, there are 6 of these user-defined trial types: “TrialSesPhase”, ”TrialInterFd”, ”TrialAutoShort”, ’TrialAutoLong, ”TrialPreShrtITI”, and “TrialPreLngITI”.



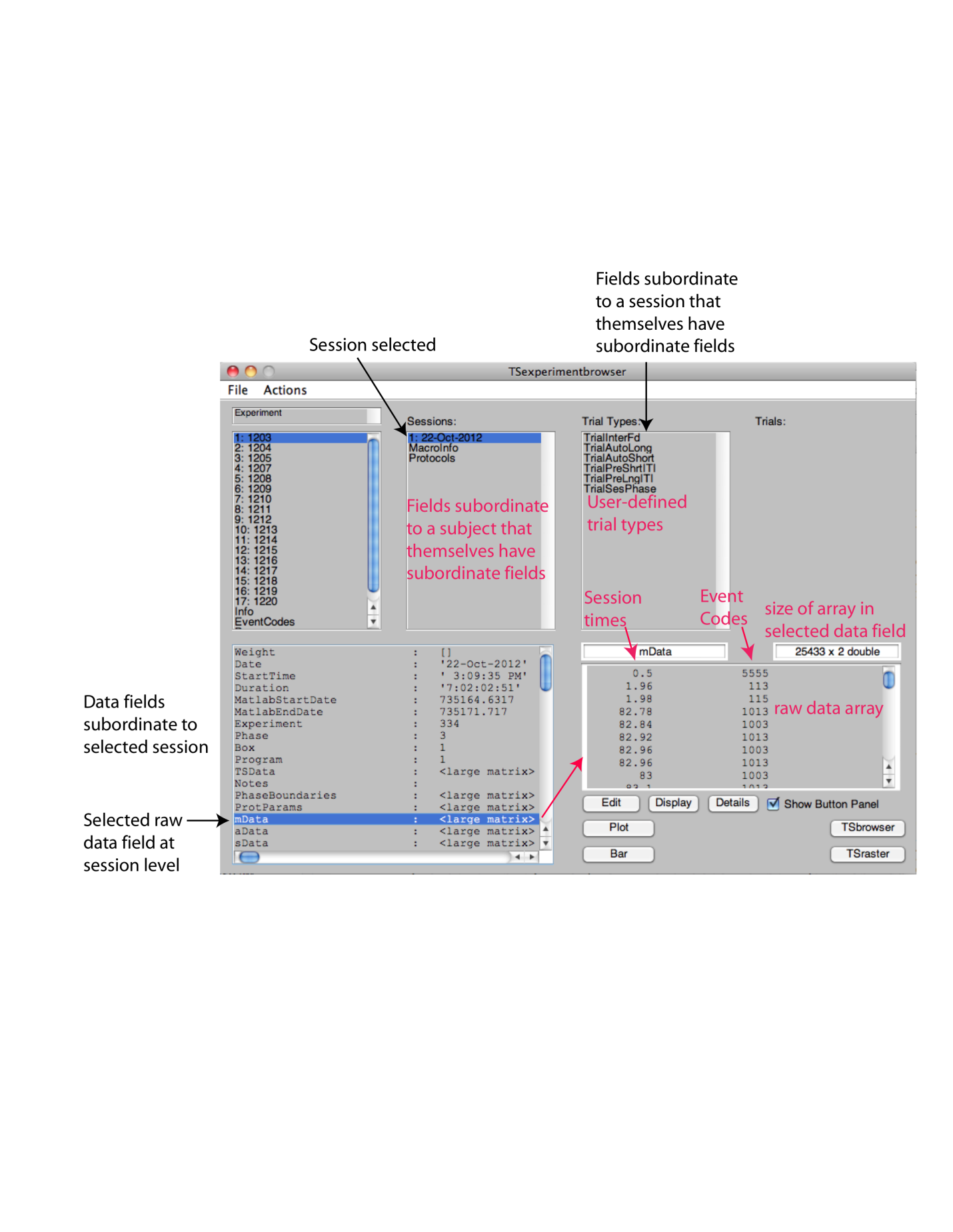
**Figure 3.2.** *The Browser once ‘Experiment’ is selected. The non-terminal fields (the fields that are themselves structures) appear in the top pane. Topping the list of non-terminal fields is a field for each subject. This field shows the subject’s index number (1, 2, 3, etc) followed by its ID number (e.g., 1203). Below these numerically indexed subject fields is the ‘Info’ field. Below that is the ‘EventCodes’ and field, and below that is the ‘Programs’ field. This last is not visible in the Figure because it is beneath the sill of the pane. To see that field, one would use the scroll bar to the right of the pane to scroll down to it. The terminal fields at the Experiment level appear in the bottom pane. Again, they are not all visible in this display, which is why there is a scroll bar on the right of this pane.*

To select a subject, click on one of the ID numbers in the top left pane. A pane will appear to the right under the title “Sessions:”. This newly appeared pane displays the non-terminal fields at the Subject level, the fields that are themselves structures. Just as when you clicked on Experiment, the terminal fields of the selected structure appear in the pane at bottom left of the Browser window (see Figure 3.3). In other words, the bottom left pane always shows the terminal fields of whatever structure one has selected in one of the top panes (“Experiment,” “Subjects” “Sessions,” “trial types” and “Trials”, as the case may be)



**Figure 3.3.** *The Browser with a subject selected. The pane at bottom left, which previously displayed the terminal fields at the Experiment level, now displays the terminal fields for the selected subject. Clicking on one of those terminal fields causes its contents to appear in the pane at bottom right. The pane at bottom right is for the display of data in terminal fields. The pane that has opened to the right of the Subjects pane under the title “Sessions:” displays the non-terminal fields at the Subject level. Topping the list of non-terminal fields at the Subject level are the indexed Session(#) fields. In this example, there is only one session, but often there are many. Below the indexed Session fields in this example are ‘MacroInfo’ and ’Protocols’. These two non-terminal fields are only found in Experiment structures from fully automated experiments, in which the experimental data are loaded into the structure every few hours while the experiment is running so that Matlab can analyze the data as the experiment progresses and decide when to advance a mouse from one protocol to the next.*

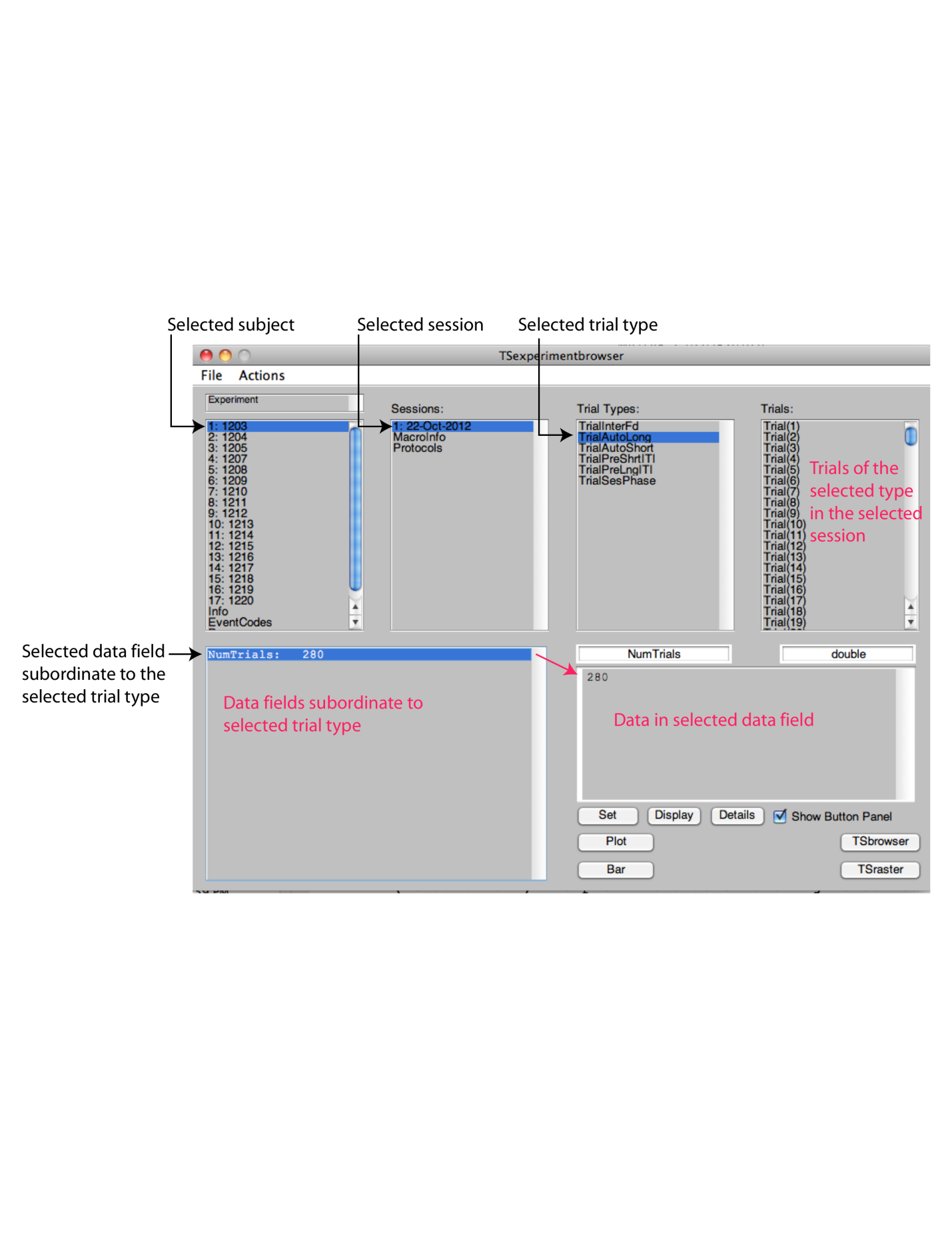
To look at one of the sessions that subject was in, click on it in the second pane (the pane under “Sessions.” If the user has defined one or more trial types, a new pane will open to the right under the title “trial types” (see Figure 3.4). The names the user assigned to the user-defined trial types will be displayed in this pane. The names by which trials of different types are distinguished are always prefaced by ‘Trial’. Thus, we wee TrialInterFd, TrialAutoLng, TrialAutoshort, etc.



**Figure 3.4.** *The Browser with a Session selected. The names of the 6 user-defined trial types appear in the pane under the title “trial types.” Now, the terminal fields at the Session level appear in the pane at bottom left. As always, clicking on a terminal field displays the data it contains in the panel at bottom right. In this example, we have clicked on the mData field, which contains the tsdata from the concurrent VI matching phase of an experiment in which each mouse was tested on several different protocols.*

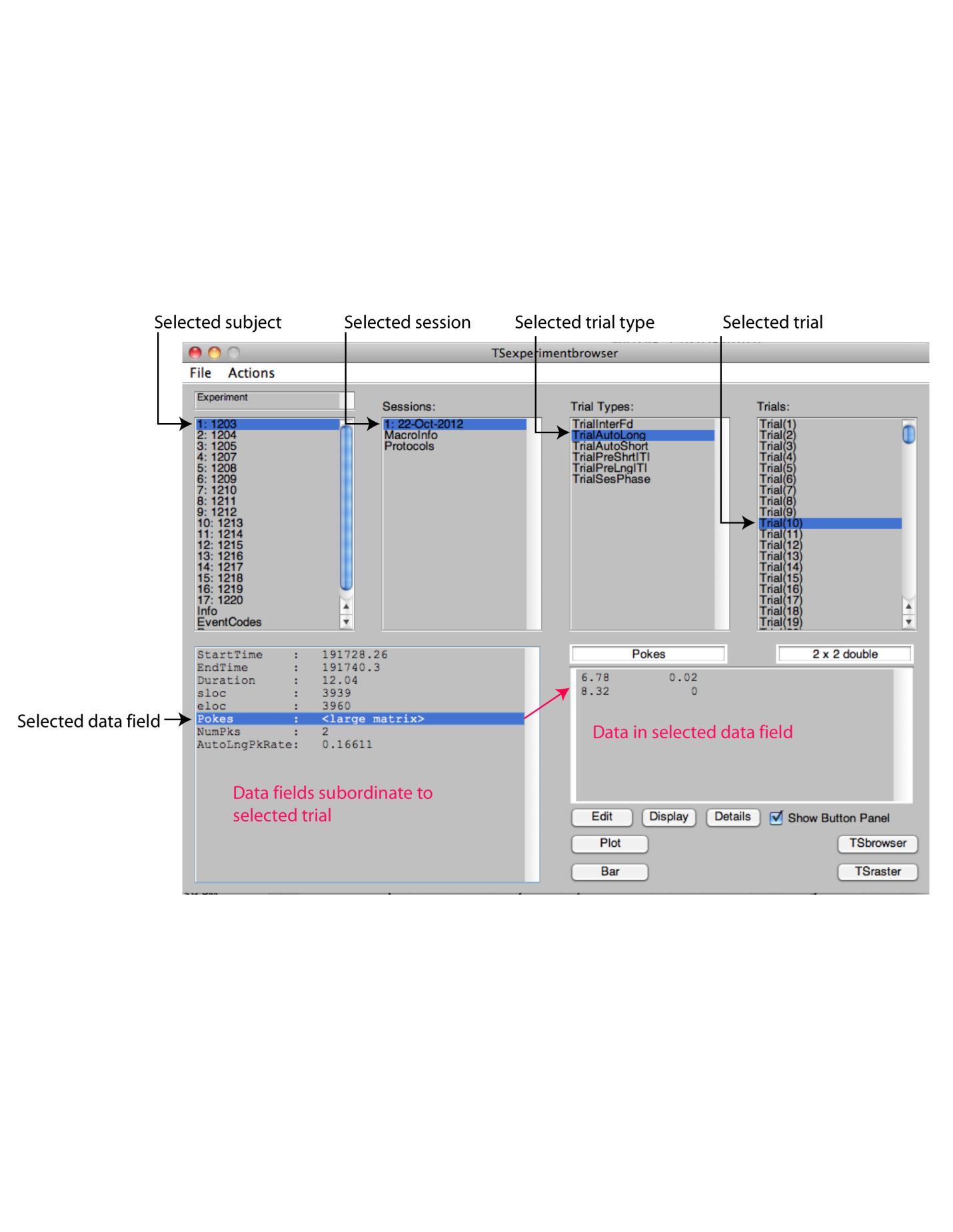
As you may guess, if you select one of the trial types, all of its substructures (all the trials of that type) appear in the rightmost pane on the top, while its terminal fields appear in the bottom left pane. At any time, you can go back to a previous pane and select different items for browsing.

Clicking on a trial type, causes the display of the non-terminal fields for that trial type to appear in a pane to the right under the title “Trials:” (Figure 3.5). Trials within a trial type are numerically indexed, just as are Subjects and Sessions. Any terminal field at the trial type level (NB, not the trial level) is displayed in the bottom left pane, where the terminal fields are always on display. In this example, the only terminal field at the trial type level is the field that specifies how many trials there are of that type. This is another one of the fields that are automatically computed by a call to TStrialstat, the Toolbox function that computes statistics at the Trial level.



**Figure 3.5.** *The Browser with a Session and a trial type selected. The indexed trials of the selected type [Trial(1), Trial(2), etc) appear in the pane to the right under the title “Trials:”. These are non-terminal fields, that is, they have fields underneath them. The sole terminal field (“NumTrials”) at the trial type level appears in the pane at bottom left. Clicking on it displays the data it contains in the panel at bottom right (a single number).*

When you click on an indexed Trial in the “Trials:” pane, you have reached the bottom of the hierarchy; all the fields at the Trial level are terminal fields, so, of course, they are all displayed in the bottom left pane (Figure 3.6)

**

**Figure 3.6** *The Browser with an indexed trial selected. The Trial level is the bottom of the hierarchy, which means that all of the fields at this level are terminal fields. They are displayed in the bottom left pane. As always, clicking on a terminal field in the bottom left pane, displays its contents in the bottom right pane.*

**Working With Data from Inside the Browser**

There are two small rectangular boxes above the bottom right pane in the Browser. The one on the left contains the name of the selected field, while the one on the right contains the type and size of the contents in that field. For example, if you select the field containing a subject’s ID number, the small box to the left would contain “SubId” while the box on the right would show “1x1 double.” If the selected field were any type of array, including a string (since a string is simply an array of characters), the right box would contain the size of the array, in addition to its type. For example, if the selected field were “Source,” and the contents of the field is ‘Jackson,’ the box on the right would contain “1 x 7 char,” meaning that the field contained a character array of size one by seven.

There are also three buttons underneath the bottom right pane: Set (or Edit, depending on the selected field), Display, and Details. We will discuss Display first, then Details, then Set/Edit.

The ‘Display’ button is used, as the name suggests, to display the entire contents of the currently selected field in Matlab’s Command Window. If the field contains an array that is too large to be displayed in the lower left pane of the Browser, it will simply display “<large matrix>” next to the field name. Once you select that field, the first 100 lines of the array it contains will be displayed in the pane on the bottom right pane. If it’s particularly large (over 100 rows), like TSData, for example, it will display the first 50 lines and the last 50 lines of the array with text showing how many intermediate lines were truncated. If you click ‘Display’ while TSData, or any field, is selected, the field’s contents *in its entirety* will be displayed in the Matlab Command Window. (Bear in mind that TSData may be many thousands of rows long!)

When the ‘Details’ button is clicked, the Details window pops up. This window provides some of the same information that can be found in the Browser, such as the name of the field, its value, and its type, in text boxes. It also shows the path through the structure where the selected field can be found. For example, if the selected field was at the Subject level, ‘Path’ would be Experiment.Subject(i).<fieldname>. This path to the field can be copied and pasted into Matlab’s command window in order to access the contents of that field. Alternatively, if you press the ‘Display in Matlab’ button directly below the ‘Path’ field, the contents of the field will be displayed in the Matlab command window. Thus, there are three ways to get a field’s value to display in Matlab’s command window: 1) Click the ‘Display’ button in the Experiment Browser, 2) Press the ‘Display In Matlab’ button in the Details window, and 3) enter it the field name the Command line, either by manually typing it or copying the Path in the Details window and pasting it and pressing return. Below the ‘Display In Matlab’ button is the ‘Copy to Workspace as’ button, which performs a similar function, but there is a distinction that it is important to understand. You will notice a text box next to this button, which by default contains the field’s name. When you press the ‘Copy to Workspace as’ button, Matlab creates a *new* variable in the workspace with that name (the field name), whose value is the same as that of the contents of the field. If you intend to do a lot of work with this variable, you may want to change its name from the default to something short and sweet, like, say “D,’ before you click on the Copy to Workspace button. Copying data from the Experiment structure to another variable in the Workspace is useful if you are interested in working with that data without changing the original data in the Experiment structure. We stress that this creates a copy; no functions applied to this variable will be applied to the information in the Experiment structure. It is useful for testing out a function on some data, or any time you would like to work with a piece of information while still retaining the original copy of it unaltered in the Experiment structure. No changes made in the Details window will alter the Experiment data structure; all of the values in the Experiment will retain their values, regardless of edits made in the Details window. The function of having the changeable text boxes is simply to allow the user to copy contents of the box into the Matlab workspace as the contents (or “value”) of a variable.

Whether the Set/Edit button displays ‘Set’ or ‘Edit’ depends on the currently selected field. ‘Edit’ is shown when the selected field contains a numerical matrix; when it contains a character array or a single number (i.e., an integer, a double), ‘Set’ is shown.

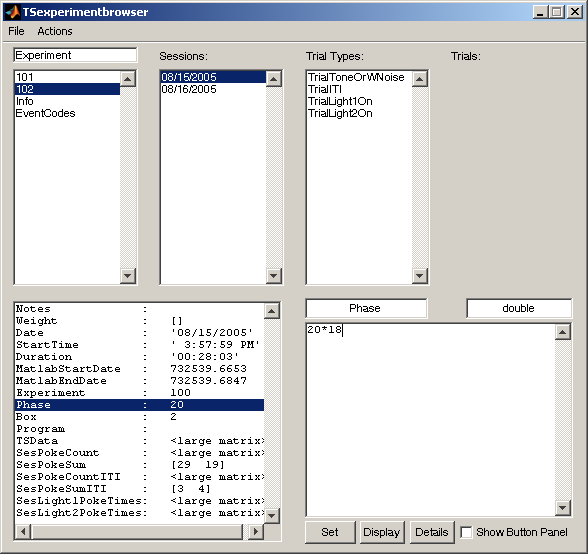
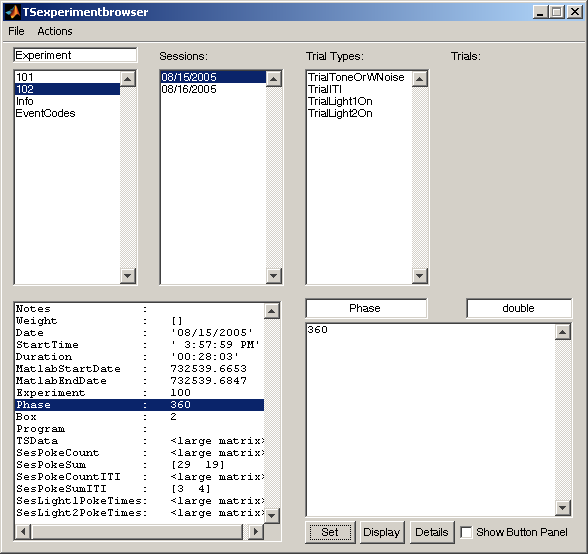
When ‘Edit’ is shown, a numerical matrix is selected and displayed in the bottom right pane in the Browser. You cannot edit the field directly from the Browser; instead, you must click ‘Edit’ and Matlab’s array editor will open. The array editor opens a spreadsheet-like window, where you can manually change one or more values in the array. However, we urge you not to use this method to apply any calculations or functions to the array. Because the ‘Edit’ function leaves no trail of its changes, it could become extremely confusing if functions are applied to entire arrays with no explanation as to why the values are what they are. The ‘Edit’ function is useful, though, if there is a mistake in the array that you would like to correct. For example, say that the date at the top of the array were wrong. You could click ‘Edit,’ change the cell that contains the date so that it is accurate, and make a note of it (in the ‘Notes’ section) so that anyone looking at the data knows that there is a discrepancy between the raw data and what the field currently contains. This note is also useful so the user is reminded to look for a bug when the computer is recording the date.

When ‘Set’ is shown, the selected field contains either a single numeric value or a string. Fields that display the ‘Set’ button can be updated directly in the Experiment Browser (unlike ‘Edit,’ where updating is by means of the array editor, as just explained). Simply click in the bottom right pane to make whatever changes you would like in the contents of the field, and then click ‘Set’ to update these changes in the Experiment structure. This is especially important for making Notes to document any changes or irregularities in any fields in the Experiment structure.

Edit/Set, Display, and Details are the only three buttons that are shown by default in the Browser. If you check the ‘Show Button Panel’ checkbox, up to 8 other buttons will appear that can be used to make various kinds of graphs from within the Browser, using data in a selected field. Those buttons, and their graphical functions, will be discussed in a later chapter on Toolbox graphics (Chapter 15).

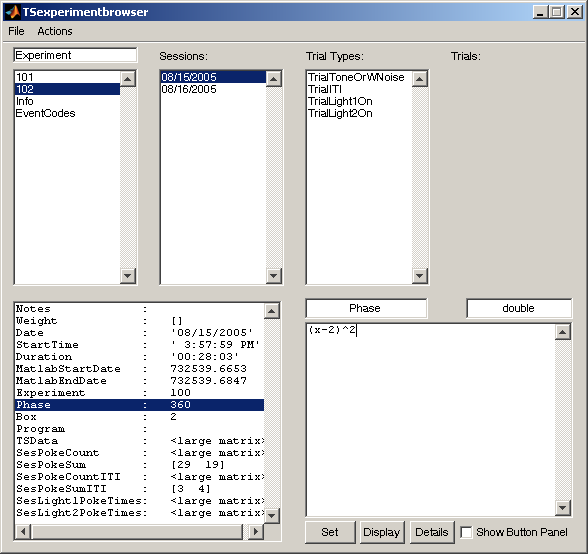
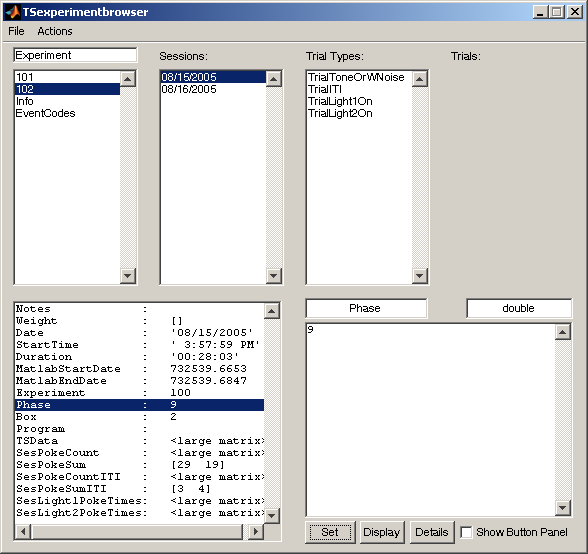
**Special Features**

When setting values in the bottom right pane of the Experiment Browser, in addition to typing the value, you can type mathematical expressions. They will be evaluated when you press ‘Set.’ When there is a numeric value, the Browser uses the ‘Eval’ function to evaluate the string as MATLAB expressions. When you click ‘Set,’ the expression is evaluated and reduced to a single value (see Figures 3.7 & 3.8).

**Figure 3.7** *To edit a field, select it from the bottom left window; its value will appear in the text box to the right. Click on the value, and replace it with the updated value. If it is an expression, like on the right, it will be evaluated into a single value once ‘Set’ is pressed.*

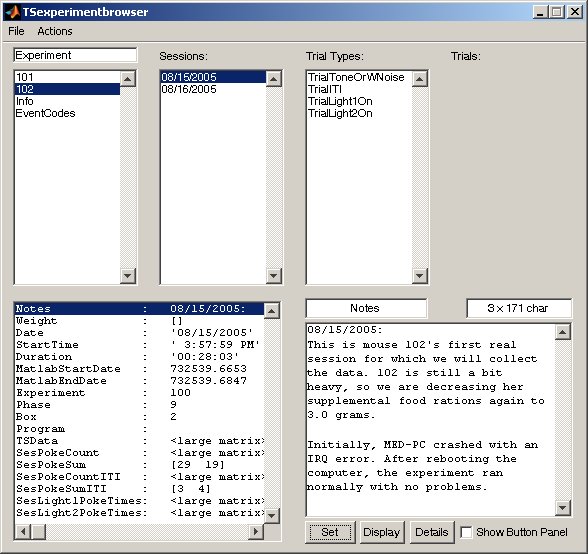
You can even refer to workspace variables in the expressions you type in the lower right pane. For example, go to the MATLAB command window and type ‘x = 5’ to set x as a workspace variable with value 5. You can now use x in an expression in the text box and MATLAB will evaluate it correctly (see Figure 3.8):

**Figure 3.8** *A field is being edited using a variable (x) that has been declared in the workspace. Once ‘Set’ is pressed, the expression is evaluated according to the value assigned to the variable in the workspace.*

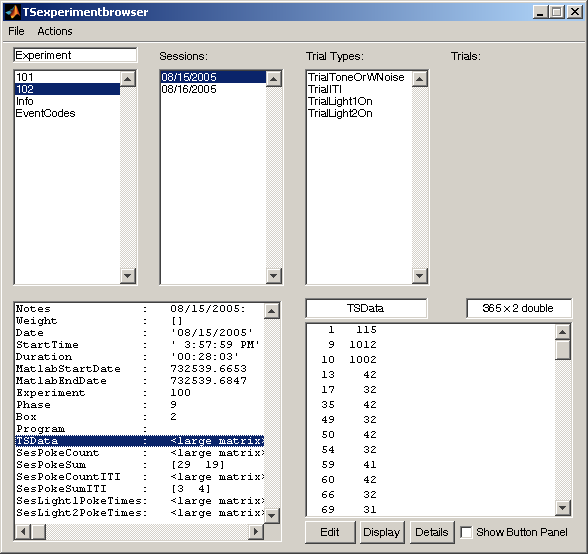
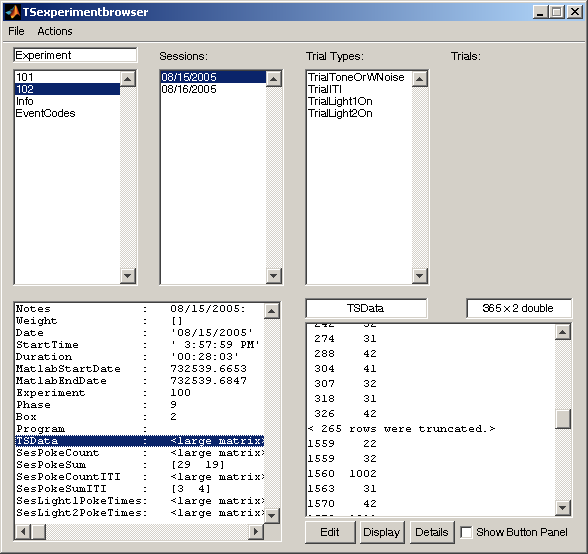
A non-numeric field, such as Notes (under the Session level), will not be evaluated when the ‘Set’ button is pressed. Because ‘Notes’ is of type character (which can be found in the box to the upper right of the lower right pane), if you were to type ‘1+1’ and press ‘Set,’ ‘Notes’ would contain “1+1”, a 1 x 3 char array, *not* the integer 2.

There is a field for notes at the Experiment level the Subject level and the Session level; however, notes can be added as a user-defined field in any level. We recommend entering notes into Experiment wherever possible. The purpose of the Experiment structure is a single repository that contains every piece of the puzzle. Anyone should be able to look at the Experiment structure and comprehend exactly what is going on. Explanatory notes greatly advance this goal. Both you and others will deeply appreciate the time taken to add notes. You can type as many lines of text as you wish in the ‘Notes’ field; it will become a large char array (see Figure 3.9):



**Figure 3.9** *The field ‘Notes’ is selected from the bottom left window, and its contents appear in the bottom right text box. It is a char array, and can grow to accommodate an arbitrarily lengthy note.*

For matrices with multiple rows of data (i.e., TSData), the Browser will display <large matrix> as the value in the field pane (the lower left pane). In the lower right pane, it will try to display as many lines as it can; however, we have put a limit of 100 lines of data displayed in the this pane because MATLAB often hangs or freezes if you attempt to display more. If your matrix is more than 100 lines, the first 50 lines and the last 50 lines will be displayed, and in the middle of the textbox a note will inform you of how many lines were truncated from the display (see the screenshot on the right in Figure 3.10).

**Figure 3**.**10** *The TSData field is selected, and since it is a large 2-column array, it cannot be displayed in the window (instead, it shows <large matrix>). 100 lines of its contents are displayed in the text box to the left, with a note explaining how many lines were truncated.*

## Loading and Saving with the Browser

Under the file menu at the extreme top left of the Browser window, several features make it easier to manipulate the Experiment structure. You can load, save, create, or clear the Experiment structure from this menu. When you pull down this File menu, you see the following options:

**New Experiment:** Creates a new Experiment structure in memory. This option provides a nice GUI-front for the *TSinitexperiment* function. You are prompted to give a name, number, the subjects, lab, and species of the Experiment (lab and species are optional). Thus, one way to get your own experiment structure is to open the Browser and use this option under the File menu in the Browser.

**Load Experiment**: Loads a new Experiment.mat file. A file selector will popup, prompting you to select your experiment. If an Experiment already exists in the Workspace, you will be prompted to okay or veto overwriting it.

**Save** or **Save As**: Saves the current Experiment. A window will popup if you select Save As, so you can choose in which folder to put the file. If you press Save, the current MATLAB path will be used and the name of the Experiment will be the name of the file.

**Clear Experiment**: Clears the current Experiment without saving it. This command is equivalent to the command clear Experiment entered in the command line.

To exit the Browser, simply press the X in the corner to close your Experiment Browser window. Since the Experiment structure is in the MATLAB workspace (the Browser is just a visually simple way to navigate the structure), you won’t lose any information unless you quit out of MATLAB as well.

## Browsing An Example

We suggest that at this point you load the Craig-Shahan Experiment structure into the Matlab workspace and explore it with the Browser. You will find it in the CraigShahan subfolder of the Examples subfolder in the TSlib folder. Be sure that the CraigShahan folder is on Matlab’s search path. The easiest way to do that is to click down to the CraigShahan folder within Matlab’s directory window, which is the window on the left in the default layout. Double clicking on the CraigShahan folder will make it Matab’s Current Folder. We gave this experiment the ID number 1000 to separate it strongly from the ID numbers of the experiments run in the Gallistel lab. Following are common practice, we save the Experiment structure in a file named ‘Experiment1000’. To load it type: TSloadexperiment('Experiment1000') at the Matlab command prompt. The variable Experiment appear in the Workspace pane at upper right of Matlab’s default Desktop layout. You will also see in this pane the variables Peck and Feed, which were the only two events in this experiment. (You will also see the variable ans.) Now type TSexperimentbrowser to open the Browser. Click ‘Experiment’ in the upper left pane of the Browser and you will see that there were 8 subjects in this experiment. If you click on one of them, you will see that each subject was run for 30 sessions. If you click on one of the sessions, you will see in the trial types pane that we defined only on type of trial ‘TrialIRI’, where ‘IRI’ is short for the interval between two reinforcements. If you click on TrialIRI, you will see in the Trials pane the 30 the 39 trials that comprised a session.

# Chapter 4. Creating, Saving, and Loading Experiment Structures

Users of the TSsystem fall into two categories: 1) Those who are running or have run an experiment that generates time-stamped events and who want to use the TS Toolbox to analyze their data. 2) Those who are using the fully automated or almost fully automated TSsystem in which the data being written to a text file by a MedPC experiment-control program as the experiment runs are harvested and analyzed at regular intervals by a data-analysis and graphing program written using the commands in the Toolbox. In either case, the user must first create an Experiment structure to contain the raw data. The commands in the Toolbox presuppose the existence of an Experiment structure in the workspace; they cannot operate without it (except for those under the File menu in the Browser). The best way to create the Experiment structure depends on how you will use the Toolbox.

There are three ways to create an Experiment:  
 1) Using the Experiment Browser  
 2) Using TSinitexperiment  
 3) Using TSbegin

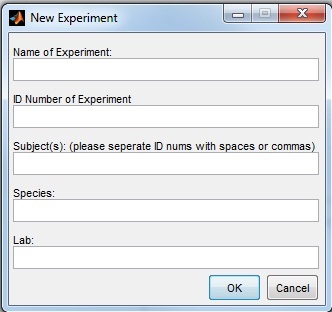
The third option, using the TSbegin function, is for users of the fully automated system. We explain it in the Chapters on the fully automated system. The first two ways of creating a structure are explained in this chapter.

## Creating an Experiment Structure from the Browser

For a first time user who has data ready to be analyzed, the simplest way to create the Experiment structure is to open the Browser (either by typing *TSexperimentbrowser* in the Matlab command line, or pressing the 'Browser' button in the Matlab shortcut bar if it exists. (If it does not exist, we suggest you put it there—see Introduction for how to add a frequently used command to Matlab’s shortcut bar).

*Before you go any further*, however, *you will need to have in front of you ID numbers of the subjects in your experiment.* This is essential, because there must be some way of telling which subject is which. If the data files you will be loading were generated by, for example, MedPC™, then each raw data file output by that software will contain somewhere a number that identifies a subject, the subject whose data are in that file. You will not be able to proceed any further unless you know enough about the raw data files to be able to find the ID number of the subject within a given file. Don’t go on from here until you have in front of you (or know how to generate) a list of the ID numbers for your subjects. If your subjects are identified by names (character strings) rather than by numbers, make up numbers for them, because the TSlib system insists that subjects be identified by numbers. You can think of these numbers at the mouse or rat or pigeon or monkey equivalent of a social security number. Although not absolutely essential, it is also desirable that you have a name for your experiment (say, the name of the file in which you intend to save the Experiment structure). Also, you may have an ID number for it. And, of course you know the species of subject (human, rat, mouse,monkey,pigeon, etc) and the name of the lab in which it was run.

Once the Browser appears, select 'New Experiment' from the File menu at the extreme top left of the Browser window, or use the keyboard shortcut Ctrl+N (Command+N for Mac users). The 'New Experiment' window will pop up (see Figure 4.1), with five text fields.



**Figure 4.1** *The 'New Experiment' window that appears when New Experiment is chosen from the File menu in the Browser window. There are fields for the Name of the Experiment, the experiment's ID number, the subjects, the species of the subjects, and the name of the lab. Once you enter the information and click OK, the Experiment structure to contain the data and statistics from your experiment will be created.*

As you can see, this only includes the minimal information needed to set up the structure. At this state, the only goal is to create the structure itself. In Chapter 5, we explain how to load data into it. Actually, it includes a bit more than the absolute bare minimum. The only absolutely essential input at this stage is the list of subject ID numbers. If, as is often the case, the Subjects have consecutive ID numbers, for example, 401, 402,…,440, then the only absolutely essential thing you need to do is enter in the “Subjects” field the following: 401:440. Matlab will convert this into the row vector [401 402 403 … 440], saving you the trouble of typing in all 40 subject ID numbers. If the subjects are not numbered consecutively, then you will need to type in each number, making sure to put a space between each one. If some stretches of subjects are consecutively numbered but others are not, you can do 401:409 420 431 435:440. We suggest that you enter them in ascending numerical order; if you do not, the TSlib function will sort them into ascending order before entering them into its list of subjects. If you enter some duplicate ID numbers, the function will delete the duplicates and print into the work space the sequence of unique ID numbers that it accepted. Because ID numbers function like social security numbers, they must be unique.

## Creating an Experiment Structure Using TSinitexperiment

When you create the structure from the Browser (see above), the Browser calls TSinitexperiment and passes to it the information you provided. TSinitexperiment is the Toolbox function that actually creates an Experiment structure; the window called up from the File menu in the Browser is just a little more user-friendly.

You can call TSinitexperiment directly by typing it into Matlab's Command Line. TSinitexperiment has a minimum of three arguments: the name of the experiment (a string), the ID number of the experiment (an integer), and the subjects (an array of the ID numbers of the subjects). There are two more optional arguments that can be used if you choose to provide more information: the species of the subjects (a string), and the name of your lab (a string).

>> TSinitexperiment('Experiment1', 1, [401 402 403 404])% When entered into the Command line, this creates a new Experiment structure titled 'Experiment1', with the Experiment ID number 1, and four subjects with IDs 401, 402, 403, and 404*.*

*>>* TSinitexperiment('Experiment2',2,[5 6 7 8],'Mouse','Gallistel')% This creates a new Experiment structure called 'Experiment2', with the ID number 2, four subjects numbered 5 through 8, with their species identified as 'Mouse' and the experiment taking place in the 'Gallistel' lab

## Saving an Experiment Structure

TSsaveexperiment; % Save the experiment structure under its current name, that is, the name specified in the Experiment.name field, the name you may have specified when you filled in the Name blank on the New Experiment GUI called up from the browser

TSsaveexperiment('New\_Example\_100');% Saves the experiment under the name specified inside the single quotes, namely New\_Example\_100

TSsaveexperiment saves the Experiment structure to a .mat file. If no filename is specified, it will be saves under its current name (the name you specified in either TSinitexperiment or TSbegin). If a file name is specified as an argument of TSsaveexperiemnt, the structure will be saved under this new name, regardless of the name is specified for the experiment inside the Experiment structure itself.

GUI Equivalent: To save an Experiment structure from the Experiment Browser, select **Save Experiment** (ctrl-S) from the file menu. To save the structure under a new name, select **Save As** (ctrl-A) and you will be prompted for the new name (see "Loading and Saving with the Browser" in Chapter 3).

## Loading an Existing Experiment Structure

TSloadexperiment;

TSloadexperiment('Manual\_Example\_100’);

TSloadexperiment loads a saved Experiment .mat file. If given no arguments, it brings up a file-browser for you to use in locating the to-be-loaded file. Given as an argument the name of a file containing an Experiment structure (and *only* an Experiment structure), it loads the structure in that file, *provided the file is on Matlab’s search path*. It is not necessary to include the extension .mat when giving the name of the file.

Just as with any .mat file, you can also load it into the Matlab Workspace by double-clicking the file in the Directory on the left of the screen. However, when an Experiment structure is loaded in this way, the dictionary of event codes is not read out from the structure into Matlab’s workspace. When this dictionary is in the Workspace, it enables you to write code that refers to events by intelligible names, rather than by their numerical codes. If this dictionary is not read into Matlab’s workspace, code that refers to events by their names—which is to say almost all code—will not run. When you use TSloadexperiment to load a structure into Matlab’s workspace, the Event Codes Dictionary contained in that structure is automatically read into Matlab’s workspace. If you have loaded the Experiment by double clicking on it (or by Matlab’s load command), then you can read its Event Codes Dictionary into Matlab’s workspace by typing TSdeclareeventcodes at the command line prompt. (As an aside to sophisticated programmers: you use TSdeclareeventcodes when creating a custom helper function that needs access to the global Experiment variable to make the dictionary accessible in the workspace of your custom helper function.)

GUI Equivalent: To load an Experiment structure directly from t he Experiment Browser, select **Load Experiment** (ctrl-L) from the file menu (see "Loading and Saving with the Browser" in Chapter 3). Loading it this way calls Tsloadexperiment, which insures that the event codes will also be written into workspace variables, making them accessible.

# Chapter 5. Loading Data Into the Experiment Structure

## Choosing a Helper Load Function

If you are using the TS toolbox to analyze data from an experiment that has already been run, you will have files contaning the raw data. These files may be text files or Excel files or csv files. We assume there is one file for each session for each subject. Thus, if there are 10 subjects and 30 sessions per subject, there will be 300 raw data files. When they are loaded into the Experiment structure, there will be 30 numbered sessions for each of the 10 subjects.

The TSloadsessions function loads these files into the structure. To do so, it calls another TS function called a load function. The job of this other function is to understand the format of your raw data files and to use its understanding of that format to extract from them the information that TSloadsessions needs. To summarize: TSloadsessions loads raw data from individual sessions into the Experiment structure. The code in TSloadsessions is not, however, tailored to the lab-specific format of raw data files. TSloadsessions copes with this by calling a helper function. The code for the helper function must be tailored to the format that is peculiar to the files you need to load.

It is essential that every raw data file has *somewhere* *within the file*  or *in the file name* the following numbers:

1. A number that identifies the subject
2. A number that specifies the year in which the session began, e.g., 2013
3. A number that specifies the month in which the session began, e.g., 06
4. A number that specifies the day of the month on which the session began, e.g, 14

If you have raw data files that contain only rows with time stamps and corresponding event codes but without this essential "header" information, then the question is, How do you (or anyone) know which file is which? One likely answer is that the critical information is in the name of the file. MedPC is often set to name the data files with the Subject ID number and date on which they were created (commonly prefaced with '!', which serves to mark the file as a data file). If the critical information is only in the file’s name, the TSloadWB.m in the Examples folder shows a basic load function that extracts necessary information from the name of the file. You can use this as your load function or use it as a template to write your own. There are other possibilities, too. You may, for example, have created separate folders for each subject, in which case the subject ID information is in the folder name. This critical information must reside somewhere and it must be made available to the load function in order for TSloadsessions (see below) to know where to put the data within the Experiment structure.

If the information is in the file’s name or somewhere else, you have two choices: 1) You can use a spreadsheet program or a text editor to add this information as header rows to every raw data file. If you have only a modest number of data files, this will probably be the easier option. 2) If this essential information is contained only in the names of the files and if there are a great many files, extracting the information from the titles 'by hand' and entering it as header information inside the file is an unappealing option. In that case, you should consider writing a custom load function. Whether this second option is appealing or not will depend on how much experience you have writing Matlab code that extracts information from text strings. The TSloadWB.m file in the WardBalsam subfolder of the Examples Folder shows one such function. It may work for your files with little or no modification. On the other hand, it may not, but it will serve as an example of the kind of code that you need to have in a load function. The TsloadShahan.m function in the CraigShahan subfolder gives another example. The format of the raw data files on which this code operates is given in the ‘SampleRawDataWnotes.txt‘ file.

Ideally, your raw data files should contain somewhere in them considerable additonal information (beyond the time-stamped events and the critical information identifying subject and date). The additional information that TSloadsessions can accept from a load function is:

Success: true or false (1 or 0) indicating that the load function sucessfully extracted the required information

exp: the ID number of the experiment

phase: the phase or group or condition

box: the number of the experimental setup in which this subject was run

Duration: the duration of the session

Notes: notes in the header of the raw data

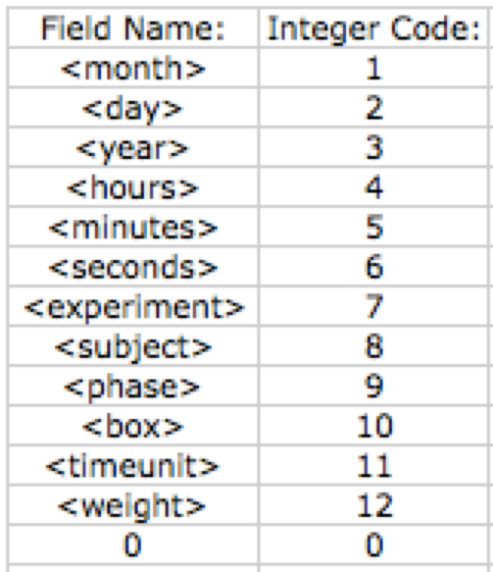
Weight: the subject’s weight at beginning of session

Program: the ID number of the MedPC code file controlling the experiment [not yet implemented]

timeunit: the unit of time in the raw data being loaded in seconds, e.g., 0.02, if each tick of the session clock is 1/50th of a second

The Toolbox comes with three load functions that provide TSloadsessions the necessary information extracted from *some*(!) common raw-data formats: TsloadMedPC, TSload2ColAndHeader and TSloadxls. If your raw data do not match one of these formats, you may either write a custom load function that can interpret your raw data files, or you may edit your raw data file so it is compatible with one of these load functions. Each of these functions reads in a raw data file of a specific format and provides to TSloadsessions the information that it needs. If you are using the fully automated system, then the load function for that system will be put into the Experiment structure by Tsbegin.

The provided load functions assume that the first few rows of a raw data file give the critical information (Subject ID and start date) in a sequence of 2-column rows. In the first column is a critical number; the 2nd column contains an integer code that tells the load function what the critical number stands for (see Figure 3.1). For example, the row that has the number of the month in which the session began in its first column must have the integer '1' in its second column. The rows do *not* have to appear in numerical order. It is the integer in the second column, not the line number, that denotes the referent of the number in the first column. The difference between the three default load functions is the way in which the columns are defined; the integer codes in Figure 3.1 apply to all three functions.



**Figure 3.1** *When using any of the provided load functions to load your data into the Experiment structure, the header of the raw data file must contain some identifying information so the program knows the correct place to store the time-stamped data. The header must use the integer codes above; however, only the rows containing the subject and the year, month, and day the experiment started (rows with integer codes 8, 1, 2, and 3) are technically required. You will get a warning message, though, and that field will remain blank if you leave out any non-necessary information.*

## The Standard Load Functions

TSload2ColWithHeader

*TSload2ColWithHeader* is used to load text files consisting of two columns of time stamped integer event codes, when the data are preceded by a 2-col header section and the end of this header section is indicated by a row with [0 0]. The column-separating character in each row may be any of the three common separators: white space, comma, or tab. If some other character is used as the separator, the user may specify a separator character as a 2nd argument in the call to TSloadsessions. TSloadsessions will pass that separator-specifying argument on to this load function.

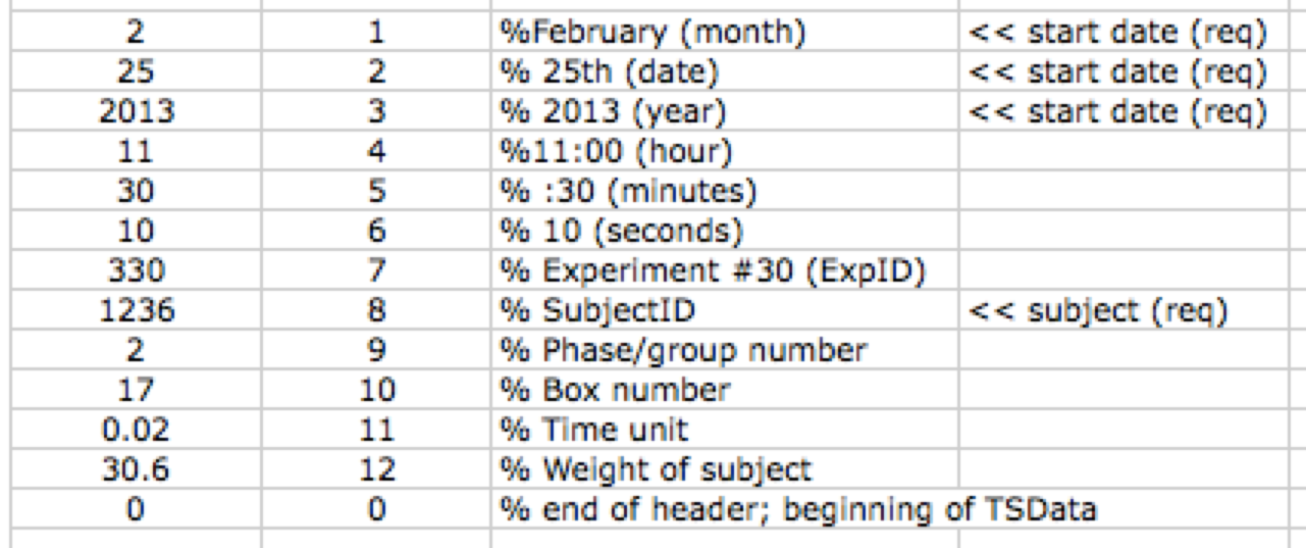
The header rows may specify or all of the information indicated in Figure 3.1. Which row specifies which information is indicated by the number in the 2nd column of that row. For example, if an experiment began in February, the row would read [2, 1]. If an integer that does not make sense as a month specifier (e.g, 0 or 15) is given in the first column of a row that has 1 in the 2nd column, the TSloadsessions function will reject it when the load function passes that value to it as a month-specifier. Similarly, when the integer 3 appears in the 2nd column, the number in the first column is the year in which the experiment started. (Thus, in order to make sense, the number in the first column must be something like 2013, not, for example, 36.) The integer 2 in the 2nd column indicates that the first column is the number of the day of the month on which the session began. (Thus, the row [37,2] would be unintelligible.) The following table gives the integer code for each different bit of information that may be contained in a readable header. The ordering of the rows in which this header information is provided is irrelevant; it is the integers in the 2nd column that tell the load function what the numbers in the first column mean, no the order of the rows. A zero in both columns denotes the end of the header and the beginning of the time stamped events. An example set of data:

2, 1 % February (month) << start date (req)  
25, 2 % 25th (date) << start date (req)  
2013, 3 % 2013 (year) << start date (req)  
11, 4 % 11:00 (hour)  
30, 5 % :30 (minutes)  
10, 6 % 10 (seconds)  
330, 7 % Experiment #330 (ExpID)  
1236, 8 % Subject ID number << subject (req)  
2, 9 % Phase/group number  
17, 10 % Box number  
.02, 11 % Time unit  
30.6, 12 % Weight of subject  
0, 0 % End of header; beginning of TSData

To repeat, only the rows identifying the subject and the year, month and day on which the session began are essential; you can read in the raw data file without having any of the other information, and the corresponding fields in the record for that session in the Experiment structure will simply remian empty. *TSloadessions* will, though, give you a message warning you that a given bit of information is missing. We recommend that your headers include as much information as possible.

TSloadstdxls

The *TSloadstdxls* is used to load raw data files in the form of a Microsoft Excel Spreadsheet. As in the preceding load function, the raw data are assumed to be in two columns within the spreadsheet. The first few (up to 12) rows are assumed to contain the header information. The values on the right side (Column B) denote into which field its corresponding value in Column A will be put; the same integer codes are used as in *TSload2ColWithHeader*. Here is the same example data :



As before, these fields do not necessarily have to occur in ascending numeric order; it is the number on the right that determines what the value on the left will be considered, *not* on which of the first twelve lines it is. The 13th line of zeroes must remain at the thirteenth line, because all information in rows after that will be put into the Experiment structure as TSdata.

The format of the files loaded by TSloadMedPC is described in the Chapters on the fully automated system.

## Setting Load Parameters

We have yet to discuss the 'Info' field located at the Experiment level (Experiment.Info). This field contains subfields that contain, among other things, basic information about how information should be loaded into the Experiment. Once any information is in the Experiment structure, it can be accessed by any function. For this reason, we enter specific load parameters into the Experiment.Info field so that when load functions are called, it can 'see' the parameters it should use. The *TSloadparameters* function is used to set these parameters; it doesn't actually do the loading of information. *TSloadsessions* does, but when *TSloadsessions* is called, it uses the parameters in the relevant Experiment.Info subfields.

Seven fields in the Experiment.Info structure are referred to during the loading of data: InputTimeUnit, OutputTimeUnit, LoadFunction, FilePrefix, FileExtension, FilesLoaded and OverWriteMode. The data in these fields are case-insensitive.

**InputTimeUnit**: This field specifies the unit of time in seconds used in the raw data file. For example, if the loaded files are time-stamped in 50ths of a second, that is, for example, if the time stamp for an event that occurs .5 s after the start of the session is 25, then the value specified here should be .02. If you choose to work with time in seconds (see OutputTimeUnit), then every time stamp in the raw data file is multiplied by .02 when the data are read in, thereby converting the raw time stamps to time in seconds. When an Experiment structure is created with *TSinitexperiment*, .02 is the value that is automatically inputted into this field. If your data are measured in something other than that and you wish to overwrite this value, you can do so directly in the Browser, or you can call the function *TSsetloadparameters* (see below). This value is only used when converting time-stamps to other units with the OutputTimeUnit parameter. If your raw time data are in hundreths of a second, that is if 50 is the time stamp for an event that happens 0.5 s into the session, then the input unit would be .01. If your raw time data are in minutes, then the input unit is 60; if in hours, then 3600.

**OutputTimeUnit**: This specifies the unit of time (in seconds) that you to be used when storing tsdata in the Experiment structure. If, for example, the InputTimeUnit for the raw data is in terms of 50ths of a second, it can be difficult for humans to think of the data in this way and convert the time stamps into a more manageable unit in their heads (e.g., seconds). When the Experiment structure is created, the default OutputTimeUnit is seconds. If you wish to work with the data in a different unit, say minutes, then you can user the *TSsetloadparameters* function, or change the value directly from either the Browser or the command line (Experiment.Info.OutputTimeUnit = <desiredunit>). Suppose that your raw data are time stamped in units of .01 s (so that 100 stamps the first 1s) and you want to work with time in minutes. You would enter .01 as the input time unit and 60 as the output unit. In that case, an event occurring 30 seconds into the session, which would have the time stamp 3000 in the raw data file would have the stamp 0.5 in the TSData field of the Experiment structure.

**LoadFunction**: When *TSloadsessions* is called to load in a session file, it uses a helper function to extract the necessary information from the raw data files (see above, 'Loading Data into the Experiment'). This field (which contains a string) specifies which helper load function is used. The default for this is *TSloadMEDPC*. Depending on the format of your data files (see above), you may want to use a different one. If you would like to use one of the other load functions provided, or one you've written yourself, you can edit this field in the structure directly (either in the Browser or from the command line), or use the *TSsetloadparameters* function.

**FilePrefix**: *TSloadsessions* loads in the raw data files from a specified folder. It may be the case that other files are also stored in the same folder as the raw data files. Therefore it can be useful to specify that only files with a given prefix string be loaded. *TSloadsessions* will read in any and all files beginning with this string. By default, this is set to the exclamation point (!), so when *TSloadsessions* is called, it will attempt to load all files in the given folder that begin with '!'.

**FileExtension** : Just as the FilePrefix field specifies a character so that any file beginning with that character is loaded into the Experiment structure, the FileExtension field specifies a file extension that serves the same purpose. In the given folder, any and all files that end in the designated FileExtension will be loaded into the Experiment structure when *TSloadsessions* is called. When the structure is first created, this field defaults to 'empty'; if you wish to add a FileExtension to the structure (e.g., ‘.dat’), you can edit this field by the Browser window, the command line, or the *TSsetloadparameters* function. Then, TSloadsessions will attempt to load only files with that extension.

**FilesLoaded:** TSloadsessions writes into this field the name of every file that it loads into the Experiment structure.

**OverWriteMode:** If this is set to false, then when TSloadsessions encounters a file with a name identical to the name of a file it has already loaded, TSloadssions skips that file; it does not reload it, and therefore it does not overwrite the data already loaded from that file. If, on the other hand, OverWriteMode is set to true, then TSloadsessions reloads the file, overwriting the TSData fields that contained the data previously loaded from a file with that name. In the fully-automated mode of operation, the raw data files are continually written to by the experiment-control program and loaded intermittently into the Experiment structure by automated calls to TSloadsessions. Under those circumstances, OverWriteMode is set to true, and the data in the TSData fields are overwritten by the updated data every time TSloadsessions is called.

TSsetloadparameters

Tssetloadparameters('InputTimeUnit',.02);  
% Indicates that the time stamps in the loaded data are in 50ths of a second.

Tssetloadparameters('OutputTimeUnit',60);  
% Indicates that the time stamps should be converted to minutes in the Experiment structure.

Tssetloadparameters('LoadFunction','loadACME');  
% Indicates that when data is loaded, the function called loadACME should be used.

Tssetloadparameters('FilePrefix','!');  
% Indicates that when data is loaded, only files beginning with should be used.

Tssetloadparameters('FileExtension','mpc');  
% Indicates that when data is loaded, only files with the .mpc extension should be used.

Tssetloadparameters('InputTimeUnit',.02, 'OutputTimeUnit',60);  
% Indicates that the time stamps in the loaded data are in 50ths of a second and should be converted to minutes.

The *TSsetloadparameters* function can be called with any number of argument, but always in sets of two, that is, as what Matlab calls variable-value pairs. The first argument is a string of the field name that you'd like to edit. The second argument is the value you want to put into that field (for the input and output time units, this is a double; for the rest, it is a string). You may update more than one field at once (e.g., TSsetloadparameters('InputTimeUnit', .02, 'OutputTimeUnit', 60) ).When *TSloadsessions* is called, it will use the helper load function specified in this level, as well as the other parameters stored here.

## Loading the Data with TSloadsessions

TSloadsessions

*TSloadsessions* loads raw data files from one or more session(s) into the Experiment structure, with the help of a load function that interprets the raw data files (see preceding section). If called with no arguments, a file-browsing window appears, within which you can search for the folder where your raw data files are stored. You may call *TSloadsessions* with an argument specifying the path to the folder where your files are stored. This function loads into the structure the file(s) from this folder that start with the FilePrefix parameter and/or end with the FileExtension parameter using the load function specified by the LoadFunction parameter in Experiment.Info as its helper. *TSloadsessions* will assume that the time stamps in the data file are in the unit specified in InputTimeUnit, and will convert them to the unit entered in OutputTimeUnit before storing them into the Experiment structure. The time-stamped data will be stored in the TSData field on the Session level (Experiment.Subject(#).Session(#).TSData).

TSloadsessions;  
% Brings up a dialog to find the folder containing sessions

TSloadsessions(‘c:\experiment\data’);  
% Loads files from specified data folder

The *TSloadsessions* command returns 2 arguments: 'result' and 'SubSes'. Result is simply a 1 or 0, to indicate whether or not the sessions were loaded successfully. SubSes is a 2-column array; it has a row for each session that was loaded, with the subject's ID number in the first column and the session number in the 2nd column.

*TSloadsessions* will not reload previously loaded sessions unless the OverWriteMode mode is turned on (see *TSoverwritemode*).

## Deleting Unwanted Sessions

A situation may arise where you have loaded a session into the Experiment structure that you do not want to be there. This could be a simple mistake, if the session simply belongs to another experiment; it may be a session that was started but aborted before it finished. Whatever your reason may be, the TS toolbox comes equipped with the *TSremovesession* function that allows you to delete one or multiple sessions from the Experiment structure.

TSremovesession

*TSremovesession(1482, 13);  
 % remove Subject #1482's thirteenth session from the Experiment structure*

The *TSremovesession* function is called with two arguments: the first is the ID number of the subject (NB *not the index number*), and the second is the index number of the session you want to remove. In this way, you may delete one session at a time using the subject's ID number and the session's index number. This function also decreases the 'NumSessions' field at the subject level (Experiment.Subject(#).NumSessions) to accurately reflect that subject's new number of sessions in the structure. The value entered in the 'NumSessions' data field is used by other functions, and thus problems may arise if it is not the same as the actual number of sessions. The TSremovesession function also recalculates the start date and end date of the experiment (Experiment.StartDate and Experiment.EndDate). If you are removing the first and/or last session of the experiment, the start and/or end date may be altered. Again, other functions use the data stored in these fields, so it is important that the dates accurately reflect when the first session started and the last session ended. *TSremovesession* will NOT remove the session file from the Experiment.Info.FilesLoaded field. You CAN remove those files “by hand.”

In essence, *TSremovesession* does three tasks: 1) it takes the indicated session out of the Experiment structure, 2) it updates that subject's NumSessions, and 3) it updates the experiment's start and end dates. If you wish to do these three tasks manually, instead of using the *TSremovesession* function, you may do so. However, it is crucial that all three tasks are completed when deleting a session, to maintain consistency across the structure, so we recommend using this function instead. To take a session out of the structure, all you have to do is set the field to 'empty' in Matlab's Command window (Experiment.Subject(#).Session(#) = []). This will not only clear all of the information, but also remove the session completely. To decrement 'NumSessions' by one, simply set it to one less than its current value in Matlab's Command Window (Experiment.Subject(#).NumSessions = Experiment.Subject(#).NumSessions – 1), or enter the correct value directly into the Browser. The simplest way to update the Experiment's start and end dates is to look at the first session's start date and enter it manually into the Experiment.StartDate field, and enter the last session's end date into the Experiment.EndDate field. Again, using the *TSremovesession* function does all of this quickly and automatically. We strongly suggest using that instead of changing the structure “by hand.”

# Chapter 6. Creating a Data-Analysis Script

The TSlib functions allow the user to load data in a collection of raw tsdata files into the Experiment structure, analyze the data, and store the results of that analysis in the Experiment structure. The results or "statistics" generated by one analysis can be further combined and analyzed by the same or different TS functions.

The strongly recommended way to perform these operations is by building a single m-file script. The script will contain a sequence of TSlib functions. Every time the script is run, all of the functions in it will be applied to the Experiment in the order in which they appear in the script. This way, it is possible to rebuild your entire Experiment and analysis with a single command (simply calling the script). This makes what would be an otherwise tedious process simple and easily reproducible.

The construction of a data-analyzing script is greatly facilitated by Matlab's cell feature. This feature allows the programmer to break up the script into little chunks, called cells. Each cell contains only a few commands, sometimes as few as one. The user can execute one cell at a time, running only what is in that cell. This allows you to build the script one or a few commands at a time, cell by cell, while preserving the ability to run the entire script (all the cells) together, thereby redoing the entire analysis from the beginning. It gives you the best of both worlds: the ability to proceed with your analysis one small step at a time and the ability to readily repeat the entire sequence of steps.

In the Example folder there is a file called "Manual\_Example\_100.m". This file demonstrates the use of an m-file to store sequences of Toolbox commands. See the section called "An Example Script" for a full listing and explanation of this script. The results of running this file have been stored as Manual.Example\_100.mat. To load this example Experiment Structure, use the TSlib function TSloadexperiment('Manual\_Example\_100');, with the Example folder being the current MATLAB directory.

Once you have loaded this experiment, you can look at the fields directly from MATLAB. For example, typing in "Experiment.Species" will return "Mouse". However, for most purposes, it is much more convenient to use the Browser to look at what is in the Experiment structure (see Chapter 3).

# Chapter 7. Working with Event Codes

## Event Code Files

Event codes in the TSData field must be integers between 11-99999 (the codes 0-10 are privileged and may not be used). When dealing with tsdata, remembering the numbers that code for the different events is difficult, especially if there are a lot of them. It is therefore useful to have intelligible variable names that can be used to refer to these events.

It is good practice to maintain some consistency within a lab regarding the event codes. If this is done, then there is a good chance that one could interpret raw experiment files years later. Each Experiment structure maintains its own set of event codes and their titles, so that when an Experiment structure is shared, all of its named event codes travel with it. Again, this reinforces the advantage of using a single structure in order to keep all of the relevant information together. TSlib supports the importing and exporting of event codes from text files. Event code files are simply plain-text files of the form:

<event-code-name1> = <event-code1>;

<event-code-name2> = <event-code2>;

The event code file is just a series of these variable definition for use within MATLAB. For example, the list shown below is the event code file used in ???.mat in the Examples folder:

Feed1 = 0021

Feed2 = 0022

PokeOff1 = 1001

PokeOff2 = 1002

PokeOff3 = 1003

PokeOn1 = 1011

PokeOn2 = 1012

PokeOn3 = 1013

StartWait1 = 1111

StopWait1 = 1101

StartWait2 = 1112

StopWait2 = 1102

HouseLightOn = 0116

HouseLightOff = 0115

StartEarly = 0121

StartMiddle = 0123

StartLate = 0125

StartTraining = 0127

StopEarly = 0122

StopMiddle = 0124

StopLate = 0126

StopTraining = 0128

PelletError = 1257

StartSession = 113

EndSession = 114

LightOff1 = 0031

LightOff2 = 0032

LightOff3 = 0033

LightOn1 = 0041

LightOn2 = 0042

LightOn3 = 0043

StartTrial = 0111

EndTrial = 0112

OperTrial = 0223

AutoTrial = 0224

ShortTrial = 0225

LongTrial = 0226

PDeliver1 = 9999

PDeliver2 = 9998

PRetrieve1 = 9997

PRetrieve2 = 9996

InNest = 0101

InTest = 0102

ToneOn = 0081

ToneOff = 0071

AnalyzeNow = 1405

PelletOn1 = 9991

PelletOff1 = 9992

PelletOn2 = 9993

PelletOff2 = 9994

**Listing 2.** Manual example event codes (ManualCodes.txt)

## TSData with Privileged Event Codes

The numbers between 0 and 10 cannot be used as event codes; however, if you have already run your experiment before knowing this, you may have long tsdata files with disallowed event codes. You will have to replace these codes with numbers that the TSlib commands can understand. One of the simplest way to do that is to multiply each one by ten to create a valid event code (i.e., 3 becomes 30, 7 becomes 70, etc.). Obviously you must make sure that these new values are not already assigned to a different meaning. You must write a small piece of code to do this conversion, and a simple function could be as follows:

for S = 1:Experiment.NumSubjects *% Go through each subject; 'S' is the  
 % current subject.*  
 for s = 1:Experiment.Subject(S).NumSessions *% Go through each session; 's' is the   
 % current session.*  
 LV = Experiment.Subject(S).Session(s).TSData(:,2)<11 *% flags event codes < 11* Experiment.Subject(S).Session(s).TSData(LV,2) =…  
 10\* Experiment.Subject(S).Session(s).TSData(LV,2); % multiples flagged values by 10  
 end  
end

You could set each event code individually (e.g., 1 becomes 43; 2 becomes 1002, etc…). All that is important is that every event code is an integer between 11 and 99999, and that you have a record of what each event code stands for. It is important to remember that once you write a function to update your event codes, you have to update the definitions of any of event codes that changed in the Event Code Dictionary in the Experiment Structure (Experiment.EventCodes). It is important that the event code definitions match what the codes in TSData mean so that the data will be analyzed correctly. Each field in the Experiment structure must be consistent with each other.

## Manipulating Event Codes

TSimporteventcodes

TSimporteventcodes; % Import event codes from a located file

TSimporteventcodes ('ManualCodes.txt’); % Import event codes from the file 'ManualCodes.txt'

The *TSimportevencodes* function reads in a text file that contains the user-defined event code assignments (see above) and stores it in the Experiment structure (in the Experiment.EventCodes field). If the function is called with no arguments, a file-browsing window appears, within which you can search for the file. Or, you may enter the name of the file, enclosed by single quotes, as and argument of *TSimporteventcodes*.

TSexporteventcodes

TSexporteventcodes ('OurCodes.txt’);   
% Save the Experiment structure event codes to a file called "OurCodes.txt".

TSexporteventcodes; % Saves the event codes to a file to be specified

*TSexporteventcodes* takes the event codes defined in the current Experiment structure and saves them into a text file. If this function is called with one argument, that argument is the name of the text file to which the event codes are exported. itt will be saved as file with that name in the current directory. When called with no arguments, a window appears, in which you can choose the folder in which to save the event codes and a file name.

TSsetdefaulteventcodes

TSsetdefaulteventcodes(‘OurCodes.txt’);   
% Load OurCodes.txt as the default code set.

TSsetdefaulteventcodes;   
% Prompt user to supply a filename or use Current Codeset as default

This function is used to set a default code set, which will be used in the event that there is no Experiment Structure defined, or if the current Experiment does not have any codes associated with it. This is stored into MATLAB’s preferences; not the actual Experiment structure. This may be useful if you need to perform testing on data cleanup or analysis routines but do not have an Experiment currently loaded. If you know the name of the file where these event codes are stored (and the file is stored in the current directory), you may enter the filename as a singular argument when calling *TSsetdefaulteventcodes*; otherwise, you may call the function with no arguments, and a window will appear where you can browse to find the file you want.

TSaddeventcodes

TSaddeventcodes('Feed3',23);   
% Add an event code called Feed3 with the value 23.

TSaddeventcodes('Feed3',23,'Feed4',24);   
% Add an event code called Feed3 with the value 23 and an event code called Feed4 with value 24.

This function allows one to add one or more event codes to codes that already exist in the Experiment structure. The first argument is the variable name, while the second argument is the numerical event code. Multiple event code definitions may be added in this fashion, in sets of two arguments (variable-value pairs) with the name first and the numbered code second.

TSrmeventcodes

TSrmeventcodes('Feed3'); % Remove the event code called Feed3.

TSrmeventcodes('Feed3','Feed4',); % Remove the event codes Feed3 and Feed4.

This function allows one to remove one or more event codes from the list of codes currently stored in the Experiment structure. Each argument is simply the name of the event code (not the actual numerical code) that you wish to remove; multiple event codes may be removed at once in this fashion.

TSdeclareeventcodes

TSdeclareeventcodes; % Allow the use of event code names in a function

This function takes all the event code definitions and declares each one as a global variable. Each variable is put into the Matlab workspace of both the calling function as well as the base workspace. This allows you to use the event codes by their variable name while writing scripts rather than referring to them by number. To allow for this within your own function, simply include this function call at the top of your function. Once this is done, you may refer to all event codes by name in that function.

S = TSdeclareeventcodes; % Get the event code structure as an output.

When an output is used with TSdeclareeventcodes, it returns the current code structure, which can be either the list stored in Experiment.EventCodes, or the default code set stored in your preferences. This is useful if your function needs a list of all the event code names, which you can get simply by using the fieldnames function on the returned structure. For example, *TSrastergui* uses this method to populate the event code lists. In the above example, the variable 'S' now contains the list of all the event codes.

# Chapter 8: Searching for Event Sequences

TSlib provides a number of functions to analyze tsdata and generate what we call “statistics,” using that term in its most generic sense, which is to say to refer to anything computed from raw data. All of the statistics depend either directly or indirectly on searches through the tsdata. These searches look for ***sequences of not-necessarily-contiguous events***. A searched-for “sequence” may consist of as little as a single event (e.g., a search through tsdata for pecks), or it may be complex sequences with several positive and several *negative events*. Negative events are events that must not occur in between two positive events in the sequence. The function that does the searching is TSmatch. It is unusual among TS functions in that it does not have as its first input argument the name of a field into which it is to put its outputs. That’s because it passes its outputs to other functions (most often to TSparse) rather than putting them into a field in the Experiment structure.

A searched-for sequence is called a *match code*. TSmatch can search simultaneously for more than one match code, that is, more than one sequence.

Syntax: *[match, bindings] = TSmatch(tsdata,[Event1 Event2 etc])*

or

*[match, bindings] = TSmatch(tsdata,{[Event1 Event2 etc] [Event1 Event3] etc})*

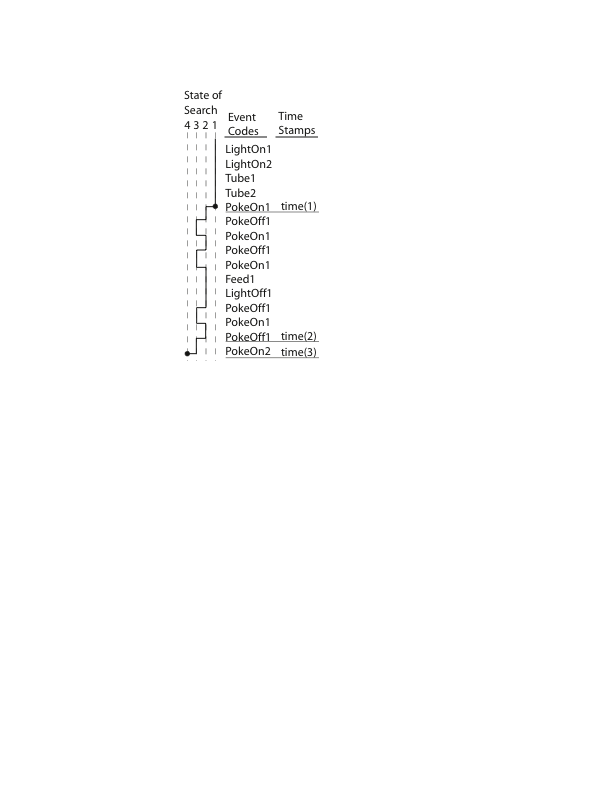
The tsdata argument must be a 2 column array of time-stamped events, with the time stamps in the first column and the numerical event codes in the 2nd column. The *match* output argument is a vector that reports which match code generated each successive successful match. Because TSmatch can search for multiple match code sequences simultaneously, when it finds one, it must specify which one it found. The match output vector contains those specifications. The *bindings* output argument is a cell array. Each cell contains a vector of row numbers for each successive match. Suppose that there are 3 events in the first of two match codes that TSmatch is searching for and 5 events in the second match code. Suppose it first finds a match to the first of the two match-code sequences. In this match, the first event occurs at Row 113 of the tsdata, the second at Row 118 and the third at Row 123. At this point in its search, TSmatch puts a 1 in the *match* output vector, to indicate that the match was to the first of the two match code sequences. It puts the vector [113 118 123] in the first cell of the *bindings* output. In what follows, these vectors of row numbers are referred to as the *bindings*.

TSmatch is the hidden workhorse of the TSsystem. It is essential to understand it in order to use the system effectively. When you write data-analysis code using TS functions, you rarely or never call TSmatch directly, but the functions that you do call often do call it. These include *TSParse*, *TStrialstat*, *TSsessionstat*, and *TSedit*.

Generally speaking, the only time one calls TSmatch directly is when one is trouble-shooting. You can use it to find where in a huge tsdata array the unexpected event sequences occur that are cause your analyses to run off the rails. Also when searching for a complex sequence, you can use TSmatch in a trial-and-error mode to determine a match code such that your command finds all and only what you want it to find. When using complex match codes containing negative events, one’s first attempt at constructing an appropriate sequence often fails to detect some sequences or (more often) detects unintended sequences.

For those with a computer programming background, TSmatch is a regular-expression search function; it finds any sequence that can be found by a finite state automaton. For those without programming background, this basically means that this function can find any sequence that can be defined by specifying a sequence of states into which the search enters as it progresses (see Figure 8.1). In the initial state, the search function looks through the event codes in the 2nd column of tsdata only for the first event in the specified sequence. If the specified sequence is only one event long, then, of course, the search terminates as soon as it finds the first occurrence of that event. If, the specified sequence has two events, then when the search finds the first of those two, it enters into its second state. In its second state, it searches only for the second event in the specified sequence. That second event may be the same as the first or it may be a different event. Suppose for example, the specified sequence is [Poke Poke]. The search begins in State 1, searching for Poke. When it finds the first Poke, it goes into State 2. In State 2, it is again searching for Poke. When it finds a 2nd poke, the search ends. Suppose, on the other hand, that the specified sequence is [Poke Feed]. The search begins in State 1, searching for Poke. When it finds a poke, it goes into State 2. In State 2, it is searching for a Feed. There may be many other pokes after the first poke and before the first feed, and there may be many, many other events as well. It ignores them all, because in State 2, it is looking only for Feed.

A negative event is an event that, when encountered, sends the search back to the immediately preceding state (unless it is in State 1). Suppose, for example, that the specified sequence is [Poke -LightOn Feed]. The search begins in State 1, searching for a Poke. When it finds the first poke, it goes into State 2. In State 2, it is searching for a feed, but it is also on the lookout for a LightOn event. If it detects the latter, it goes from State 2 back into State 1, where it is looking for a poke. Thus, the search terminates only when it finds a poke and somewhere after that a feed *without a LightOn having occurred between the poke and the feed*. The ability to specify these negative events greatly increases the power of this search function

**Figure 8.1.** *The search for the sequence [PokeOn1 PokeOff1 -PokeOn1 PokeOn2] has four states: It begins at Stage 1, where it can only move up to State 2 if it encounters a PokeOn1 event. Once it moves to State 2, it can only move up to State 3 if it finds a PokeOff1 event. In State 3, it can only move to State 4 and finish the search once it finds a PokeOn2 event; however, if it finds another PokeOn1 event before then, it will be knocked back down into State 2, where it will resume looking for a PokeOff1 event. As you can see, the search begins in State 1 searching for a PokeOn1 event, which it finds at the fifth row. It immediately jumps up to State 2 and begins looking for PokeOff1, which it finds in the very next row. Now at State 3, it is now searching for PokeOn2, but unfortunately it runs into a PokeOn1 event in the next row, bumping the search back down to State 2. Since it is back at State 2, it is now looking for PokeOff1, which it finds; again, once it is in State 3, it runs into PokeOn1 before finding PokeOn2 and once again goes back down to State 2. It continues to cycle between States 2 and 3 until it finally reaches State 3 (finds a PokeOff1 event) at line 14, followed immediately by PokeOn2, which bumps the search to its fourth and finished state. In State 4, it have found a full “visit” to Hopper 1 (PokeOn1, PokeOff1) followed by the onset of a visit to Hopper 2 (PokeOn2). TSmatch reports the row numbers of these events (here, rows 5, 14, and 15) as a vector in ‘bindings’ and the number of the sequence this matches (since we only searched for one sequence, it would just be 1)*

## Match Codes

Match codes are the input arguments to TSmatch. They are user-specified sequences of events for which TSmatch searches. For example, say that you want to find how long it took a mouse to leave the feeder once there was no more food left. You would want to search for the event code that signals that a feeding is over, followed by an event code that signals the mouse left the feeder. Together, these two event codes constitute a match code. A match code is simply a vector of event codes that you're looking for in a set of tsdata. It identifies a pattern of behavior or behavior-relevant events (like noise-on or feed). One may want to make a computation using that pattern (for example, compute the latency from feeding termination to departure. Or one may want to pull a possibly lengthy chunk of data out of the data stream in order to search for patterns within it, as for example, when wants to find the number of pokes within each successive inter-trial interval). Although tsdata contains both time stamps (in Column 1) and event codes (in Column 2), TSmatch only searches through the column with event codes (the 2nd column).

A match occurs when the sequence of event codes in the Match Code vector is found in the tsdata. The events in the sequence need not happen directly after one another; other events can occur between them, as long as these intervening events are not “forbidden” at that point in the sequence; the forbidden events are those with a minus sign in the match-code sequence.

A statistic that researchers are often interested in is the inter-repsonse interval (IRI). To find the IRI for say, pokes into Hopper 1, you need to find the interval between one poke and the next. In order to do that, you have to find every pair of pokes- this is what TSmatch does, and its results are passed to another function, like TSparse (see Chapter 9), which computes the actual statistic. Let’s say the event code for Poke1 (a poke into Hopper 1) is 20, and we want to find the pattern [Poke1 Poke1] to find the IRI for pokes. Take the following short example of Tsdata (time stamps on the left (2nd col of table), event codes on the right(3rd col of table):

|  |  |  |
| --- | --- | --- |
| Row # (for ref) | Time Stamp | Event Code |
| 1 | 1023.5 | 20 |
| 2 | 1024.2 | 30 |
| 3 | 1030.1 | 70 |
| 4 | 1031.7 | 20 |
| 5 | 1031.9 | 90 |
| 6 | 1056 | 70 |
| 7 | 1056.6 | 80 |
| 8 | 1056.9 | 15 |
| 9 | 1060.2 | 20 |
| 10 | 1061.7 | 25 |
| 11 | 1075.2 | 20 |

You would find a match for the [Poke1 Poke1] match code vector at Rows 1 and 4. They do not have to occur one right after the other; as you can see, Event 30 and Event 70 happened between the two Poke1's. Generally speaking sequences that match a match code will occur many times in the long list of tsdata, especially for a common event like Poke1. Once a match is found, TSmatch continues looking for another instance of one of the match code sequences, beginning right where the previous match finished (in this case, Row 4). In the above example, [Poke1 Poke1] first occurs at Rows 1 and 4; it appears again at Rows 4 and 9 and again at Rows 9 and 11. So for the match code [20 20], measuring the time between two pokes into Hopper 1, we have matches at [1 4] and [4 9] and [9 11]. The cell array ‘bindings’ would have the vector [1 4] in its first cell, while bindings{2} would have the vector [4 9] and bindings{3} the vector [9 11].

Match codes can be very complicated; they can contain as many event codes as you like. In the above example, searching for the match code [30 70 70 20] would be matched at Rows 2, 3, 6, and 9. If a match code consists of a single event code, a match will occur at each position that code matches. For example, if your match code were simply [Poke1], then there would be matches at Row 1, Row 4, and Row 9.

## Using Negative Event Codes

Match code sequences can also contain events that *cannot* appear at a specified location in the sequence in Tsdata; this greatly increases the flexibility of the search. Event codes that must not appear in a certain position are preceded by a minus sign (-). For example, take the match code vector [20 20] that we used before. It returned two matches: at Rows 1 and 4, and 4 and 9. Now take the match code vector [20 -15 20], where 15 codes for some other event, say, LightOn. Now, we are looking for a sequence of two instances of Event 20 such that Event 15 does not occur between them. The match at Rows 1 through 4 contained the following event codes:20, 30, 70, and 20. There is no Event 15 here, so this match would be returned when looking for the [20 -15 20] sequence. Note, though, that the vector in ‘bindings’ remains [1 4]; only positive numbers are included in the bindings, not negative numbers. (If you think about it, there is no way that one could include the row numbers at which the negative events “occurred” because an event that does not occur cannot have a row number. This point has escaped the notice of associative learning theorists for generations; they treat the non-occurrence of events as if they were events in their own right.) Look at the match between Rows 4 and 9, though. This match contains event codes 20, 90, 70, 80, 15, and 20. Here, Event 15 occurs between the two instances of Event 20, so this sequence is not a match for the match code sequence [20 -15 20]. When *TSmatch* is in State 2, having already found the first Event 20 and looking for the second, it runs into the forbidden Event 15 at Row 8 and is bumped back down to State 1, where it resumes looking for the first Event 20. It resumes this search, finds it, and is bumped up to State 2 at Row 9. However, the tsdata finishes without finding a second Event 20, so no match is found. If multiple events have been already been matched, a negative event only brings the search down to the immediately preceding state; it does not necessarily start the search over from the beginning. For example, take the match code sequence [20 30 40 -50 60]. If 20, 30, and 40 are all matched, and it finds Event 50 before Event 60, it will ‘undo’ the match to 40, go back to the preceding state (State 3), and simply re-start its search for Event 40 at the row where it found Event 50.

Match code vectors can contain any number of positive and negative event codes, and they often become quite complicated. Take the hypothetical match code vector [50 -20 -30 70 80 -90 50]. If looking through a large set of tsdata for this match code, you would be looking for Event 50 followed by Event 70, with neither Event 20 nor 30 occurring between the two. After Event 70, you would look for Event 80, and then Event 50, so long as Event 90 did not occur between 80 and 50.

## Searching for More Than One Match Code Simultaneously

*TSmatch* accepts more than one match-code sequence as input arguments. To make it search for more than one sequence, you give it for its second argument a cell array, with the different sequences in different cells. When thus given more than one sequence, it searches for all them simultaneously. However, whenever its search terminates, it reports only one match even if in fact two or more of the searched-for sequences terminated simultaneously at the same event.

Take the following tsdata:

1 20  
2 30  
3 90  
4 20  
5 20  
6 70  
7 80  
8 15  
9 70  
10 25

Say there is another hopper, Hopper 2. You are interested in the IRIs (inter-response intervals) at both hoppers. So, you need first to find for each hopper each poke at that hopper followed by the next poke at that hopper. The textual event code for a poke at Hopper 1 is Poke1 (event code #20) and the textual event code for a poke at Hopper 2 is Poke2 (event code #70). Now, if we want to search the tsdata to find the poke pairs in both hoppers, we would call the following:

[match, bindings] = TSmatch(tsdata, {[Poke1 Poke1] [Poke2 Poke2]});

As you see, the second argument, containing the match sequence, is no longer a single match code vector; rather, it is a cell array with a match code vector in each cell. Assuming that Poke1 = 20 and Poke2 = 70, and using the above tsdata, this function would return 'bindings' = { [1 4] [4 5] [6 9] } and 'match' = [1 1 2]. Let's take a closer look at these results. Bindings{1} contains row numbers 1 and 4; this is a match for two Event 20’s, [Poke1 Poke1], which was the match code in the first cell of the cell array argument. Thus the corresponding element in 'match', match[1], contains the integer 1 to denote the position in the passed-in cell array of this match codes. Now take the bindings{2}; this contains the row numbers 4 and 5, which are again matches for the first match code sequence. Therefore, match(2), the second element in the match vector, is again the integer 1. You may notice that Event 20 appears at Rows 1, 4, and 5. In theory, this is three-fold match: 1-4, 4-5, and 1-5. However, TSmatch does not recognize the third match (1-5). Once it finds a complete match, such as the one at Rows 1 through 4, it terminates and completely restarts. For all intents and purposes, it has ‘forgotten’ about the Event 20 that happened at Row 1. It can only see the current event 20 (at Row 4) and event 20’s further along in the TSdata (e.g., at Row 5). The third cell in ‘bindings’, bindings{3}, contains a vector containing Row numbers 6 and 9, which is the location of the match for [Poke2 Poke2], the second sequence in the passed-in cell array. Because bindings{3} represents a match for the second sequence, match(3), the third element in the match vector, is the integer 2.

## Overlapping Match Codes

### Shingled Matches

When searching for more than one sequence at a time, sequences may overlap. When a new match code begins in the middle of a different match code, we call it shingling. TSmatch works in such a way that only one of the matches will be found and reported. Say we make the call TSmatch(tsdata, { [30 70] [10 50] }), using the following data:

1 10  
2 20  
3 30  
4 40  
5 50  
6 60  
7 70

This would return only one match: Rows 1 and 5, which matches the second sequence. The first sequence appears at Rows 3 and 7. It is not found, because once TSmatch finds a complete match sequence in the tsdata (here, this happens at Row 5), it starts its next search at the row where that match terminated. Thus, when it finds the match for [10 50] at Row 5, it re-starts at that row. Starting from Row 5, it only sees what is ahead of it (events 50 60 and 70); so it finds no match with the first of the two match code sequences. In essence, when it terminated the [10 50] match, it ‘forgot’ that it had seen Event 30 already, and thus does not recognize that a match for the sequence [30 70] exists in this set of data. When it restarts at Row 5, it restarts its search for *all* match sequences at State 1. The easiest way to remember this is that when matches overlap at all, the match that *terminates first*, regardless of where it started or how long it is, will be the one that TSmatch records. In order to circumvent this, if you wish to recognize all codes that overlap each other, you must call TSmatch separately for each match code (in this case, you would call TSmatch(tsdata, [10 50]) separately from TSmatch(tsdata, [30 70])).

### Matches that start at the same time

Two different match code sequences may start at the same row. For example, using the same tsdata as above, say we made the call TSmatch(tsdata, { [10 30] [10 60] }). Here, the two match code sequences have different ending points (30 and 60) but both begin at event 10. Only the match that ends first will be found. Again, in order for TSmatch to recognize both of these matches, you must make separate function calls for each match code (TSmatch(tsdata, [10 30]) and TSmatch(tsdata, [10 60])).

### Match surrounded by another match

Match codes often occur so that one is completely surrounded by another. Again, let’s use the simple tsdata used above, and make the call TSmatch(tsdata, { [10 50] [30 40]} ). Here, the sequence [10 50] makes its first match right on Row 1; however, the sequence [30 40] begins and ends its entire match before it can find its second match (which would be event 50). As we have said, once a match is completed and terminated, TSmatch starts over at State 1 for all sequence. Thus, once it encounters event 30 at Row 3, and enters State 2, and then finds event 40 at Row 4, it enters the final state for that sequence and terminates. It records this match, then starts with a clean slate at Row 4, ‘forgetting’ that it saw event 10 at Row 1 and thus missing the match for sequence [10 50]. When tone matches surround each other, it is the inner one that TSmatch records; the outer surrounding match is ignored.

### Matches that end at the same time

Using the same tsdata as above, say we make the function call TSmatch(tsdata, { [20 50] [40 50]}). These sequences occur at Rows 2 through 5 and 4 through 5, respectively. Thus, both searches conclude when Event 50 is found in Row 5. However, only one will be “found,” that is, reported by TSmatch, namely whichever sequence is in the earlier (or earliest) cell in the cell array of sequences. The sequence [20 50] is in the first cell, followed by [40 50] in the second cell, so it takes precedence: the match at Rows 2 through 5 will be recorded while the match at Rows 40 through 50 will be ignored. No matter how many cells the passed-in array contains, the match code that appears earliest in the cell array will take precedence. Once again, in order for TSmatch to recognize each of these matches, you must call it separately for each sequence that you want to search for (TSmatch(tsdata, [20 50]) and TSmatch(tsdata, [40 50])).

### Using the 's' flag

You may want TSmatch to use a different system for breaking ties when sequences end at the same time. By default, when two matches end at the same time, TSmatch uses the order of the sequences in the passed-in cell array (see above). However, you can use the matches’ starting points to determine which match should be ignored, instead. When using the ‘s’ flag, precedence is given to the match whose first event code appears first, *not* the match code that appears first in the passed-in array. To access this mode, simply add ‘s’ as a final cell when calling TSmatch with a cell array of matchcode sequences:

[match, bindings] = TSmatch(TSData,{[Seq1] [Seq2] …‘s’ });  
% Notice that ‘s’ is the content of the final cell in the  
% cell array of matchcode sequences; it is not a separate  
% argument in its own right

Let’s assume the same tsdata used above (reprinted here):

1 10

2 20

3 30

4 40

5 50

6 60

And perform the following call to TSmatch without the ‘s’ flag, using the default mode described above (LightOn1 = 30; LightOn2 = 20; LightOn3 = 50):  
  
[match, bindings] = TSmatch(tsdata,{[LightOn1 LightOn3] [LightOn2 LightOn3]});

This function call would return a singular match for the first sequence, [30 50], at Rows 3 through 5. Although there is match for the second sequence, [20 50], at Rows 2 through 5, since they end at the same place preference is given to the sequence that appears first in the passed-in array. Now, try using the same function call except add the ‘s’ flag as a final cell in the 2nd argument

[match, bindings] = TSmatch(tsdata,{[LightOn1 LightOn3] … [LightOn2 LightOn3] 's'});

Obviously, both matches still exist in the tsdata: the first sequence matches at Rows 3 through 5, while the second sequence matches at Rows 2 through 5. This time, though, the match at Rows 2 through 5 will be recognized and recorded because of its earlier start position, while the match at Rows 3 through 5 is ignored.

### Using TSstart and TSend

It is often useful to be able to match to the very first event code in the tsdata or the very last code, regardless of what code appears there. The special codes ‘TSstart’ and ‘TSend’ can be used for this purpose. For example, suppose we want to find all ranges in the tsdata where the mouse had its head in the first hopper. A first approximation to this would be to simply use the match code [PokeOn1 PokeOff1]. This, however, could miss the case where the mouse had its head in the hopper before the session started and/or it had its head in the hopper when the session ended. The cure for this is to use the match codes {[PokeOn1 PokeOff1], [TSstart PokeOff1] [PokeOn1 TSend]}. Another way you may use these features is if you want to find where the first PokeOn1 event is. You would want to start your search at the beginning of the tsdata, and complete your match as soon as you see the PokeOn1 event. To do this, you would use the match code [TSstart PokeOn 1].

# Chapter 9. Computing with Found Sequences (TSparse)

When a sequence of interest has been found, you will usually want to make some simple computations on the event times that have been found. The “computations” may involve only reporting what those event times are. However, they often involve computing differences between the times of the found events in order to get interevent intervals. Occasionally, one may wish to perform even more complex computations on those event times. The command that enables these computations is TSparse. Like TSmatch, TSparse is seldom called directly. And, again like TSmatch, it is unusual in that it does not include among its input arguments the user-specified name of a field into which it is to put its outputs. That is because, like TSmatch, it passes its outputs to the TSfunction that called it, usually TSsessionstat or TStrialstat.

Syntax: result = TSparse(tsdata, mcode, matchcodes);

or

result = TSparse(tsdata, funhand, matchcodes);

TSparse's first argument is *tsdata* which is usually passed to it automatically from the function that calls it (e.g., TSsessionstat or TStrialstat –see Chapters 10 & 11). Its second argument (*mcode* or *funhand*) is either a short piece of MATLAB code enclosed in single quotes or the handle to a MATLAB function. The third argument is a cell array of match codes that TSmatch will be looking for. TSparse searches through the tsdata (through the data given to it as its first argument), looking for the event sequences (matchcodes) given to it as its third argument, and applying to each such sequence the computation specified by its second argument. (As you might guess, it calls TSmatch to do the search for the matchcode sequences.) TSparse passes the result of each such computation back to the function that called it (that is, it outputs the result of its computations on the found sequences).

To repeat: whenever it is called, TSparse first takes the tsdata that is passed in as its first argument, calls TSmatch to search that data for matchcodes, and does a user-specified computation on each sequence in the tsdata that matches a matchcode. The result of each such computation must be a row vector (!) and every such computation must yield a row vector of the same length as every other one, so that the results can be vertically concatenated to form an output array.

### mcode Privileged Words

*mcode* is simply a short bit of Matlab code; when using mcode with TSParse, there are four privileged words available for your use to simplify your code. The four privileged variables are time, match, starttime, and endtime.

Time is a vector that contains the time stamp for each event code that is matched. Just like any vector, you can access its contents by using parentheses; to access the time that the x’th event occurred, you would type time(x). As you can see above, TSmatch takes in a vector of matchcodes that makes up the searched-for sequence. For example, say that the matchcode argument is [Feed1 Feed2], meaning it would find the intervals between a Feed1 event and Feed2 event. Looking at the order the events occur in the sequence, Feed1 is in position 1 and Feed2 is in position 2. Within a match to this sequence, to find the time of the first event (Feed1), you would use time(1). To find the time of the second event (Feed2), you would use time(2). Suppose you are doing an experiment with pigeons and you want to know the session time at which each peck occurred. In that case the “sequence” you are searching for consists of a single event ( [Peck] ). Use the following function call:

result = TSparse(tsdata,’result = time(1)’,Peck)

In this call to TSparse, we first pass in the tsdata, then the mcode that will act on the tsdata (result = time(1)), and then the sequence for which we are searching. As we said before, this sequence only contains one event. As you can see, time(1) represents the time at which the first (and only) event in the sequence occurred. TSparse will return as the final output (‘result’) a column vector that gives the session time of each peck.

Another privileged variable is *match*. Match gives the index number of the sequence in the passed-in cell array. Say you want to find the duration of pokes at two different locations. You would search for the sequence [PokeOn1 PokeOff1] as well as [PokeOn2 PokeOff2]. If a match was found for the first sequence ([PokeOn1 PokeOff1], *match* would equal 1; similarly, a match for the second sequence ([PokeOn2 PokeOff2] would result in *match* equaling 2. To find the duration of these pokes, as well as which location they occur at, use the following function call:

result = TSparse(tsdata,‘result = [match time(2)-time(1)]’, {[PokeOn1 PokeOff1] [PokeOn2 PokeOff2]})

The final output, ‘result’, is a two-column vector. As you can see in the mcode, the first column is the *match* for that matchcode; it will be either 1 or 2 for this example, depending on whether the code matched with the first or second sequence. The second column in the ‘result’ vector will be time(2)-time(1), the difference between the second event in the sequence and the first event in the sequence (which is the duration of each poke).

The last two privileged words are *starttime* and *endtime*; as their names suggest, you use these variables to denote the very first and very last time-stamp in the tsdata, respectively. For example, say that you want to calculate how long before the end of a session each searched-for event occurred. We will search for the event, say, ‘Poke’, and create a two-column vector that records the session time at which the event occurred AND how long before the end of the session the poke occurred. Use the following function call:

result = TSparse(tsdata,’result = [time(1) endtime-time(1)];’,Poke)

We call TSparse and pass in 1) the tsdata, 2) the mcode that acts on the tsdata, and 3) the sequence of events we are looking for. The sequence here is only one event long; therefore, when using the *time* vector (explained above), time(1) denotes the time at which this event occurred. ‘Result’ is a two-column vector; in the first column we have time(1), which is how far from the beginning of the session it occurred. In the second column we have the difference between the end time of the session and the time of the event, which is the how long from the end of the session the event occurred. The privileged word *starttime* is used whenever the chunk of tsdata passed to TSparse does not begin at the beginning of the session, but rather at some unknown later time. Uusing *startime* allows you to compute how long after the first event in that sequence the poke event occurred. The use of this will become much more obvious when we come to explain TStrialstat (Chapter 12)

### Constraints on mcode- width of result vector

When passing TSparse a string of mcode, the code is applied to each found match, and the result is stored in a vector called "result". Each time the mcode is evaluated, the result gets appended to the ever-growing array to form the final result. *For this reason, each result from each application of the mcode must be of the same width*! In more complex instances, where you are searching for several different matchcodes (potentially of different lengths) and performing a different computation on each, it may happen that some computations yield vectors with x amount of columns, while other computations yield vectors with more or less columns. In that case, the result array will have the width of the widest result. The other computations *must* fill out their row vector with a ‘NaN’ in each column that has no number.

### Using Custom Function in Place of mcode

In addition to supporting embedded mcode, TSparse can accept a function handle for a function that it will call for each match. As you search for more than one or two sequences at once, and wish to apply different calculations to each one, the code may become rather lengthy. If the code you would like to apply to your data is more than a few lines, it becomes confusing and cumbersome to type directly into the function call. For this reason, you can create your own custom function, which can be as long as you like, and simply use the function handle as the argument instead of the mcode. When using this syntax, the function you call must be of the following form:

result = MyFunction(match, time, starttime, stoptime, matchcodes)

TSparse will pass the metavariables *match, time, starttime,* and *endtime* to your function (in that order). Your function should return a result back, just as was the case when using mcode. It may make use of any of the parameter variables passed to it by TSparse.

Here is an example that uses a function handle. First you must find or create a Matlab function that will produce your desired result; let’s use the following simple function that calculates a different statistic for each match sequence. If the first sequence is matched, it calculates the duration; if the second is matched, it calculates the time from the beginning of the session; if it matches the third sequence, it calculates the time from the end of the session. Each of these computations are in the second column of a two-column vector, while the first column denotes, let’s say, the number of the hopper where the feeding event took place.

*function result = example(match, time, starttime, endtime)*

*if match==1*

*result = [3 time(2)-time(1)];*

*elseif match==2*

*result = [4 time(1)-starttime];*

*elseif match==3*

*result = [5 endtime-time(2)];*

*end*

Using this function, our final result vector from TSparse will show in each row the number of the feeding hopper the event took place at (three, four, or five), as well as the respective calcuation.

result = TSparse(tsdata,@example,{[FeedOn3 FeedOff3]  
 [FeedOn4 FeedOff4] [FeedOn5 FeedOff5]});

This works exactly the same as the examples described above that use mcode, except it calls the function ‘example instead of executing a line of code. Keep in mind that TSparse calls TSmatch, which returns matches and bindings. Each match is then passed to the helper function; the helper function (‘example’, which we just wrote) acts on a single match, and is thus called for each match. Every time the helper function is called, the result will be appended onto the result vector in TSparse workspace. Once every match has been analyzed, the final result vector (containing all of the results from the helper function) is returned to either to the workspace if TSparse has been called from the command line, or, much more commonly, to the function that called TSparse, which is usually either TStrialstat or TSsessionstat.

Remember that TSparse is rarely, if ever, called directly (on its own). This all happens internally inside of some other function(s). However, in order to understand how those functions work, it is important to understand how TSparse works as well.

# Chapter 10. Computing Session-Level Statistics (TSsessionstat)

## Statistics From TSData at the Session Level- TSsessionstat

TSsessionstat(statname, funhand, funhandArg1, funahndArg2...)

As we see have seen in the previous chapters, we use the matchcodes to search for sequences in tsdata. TSparse is then used to compute simple statistics (e.g., interevent intervals) from the found sequences. But, as we noted in the previous chapter, where we explained TSparse, it does not operate on its own. It is called by other functions—usually by TSsessionstat or TStrialstat. The function TSsessionstat, as its name suggests, extracts summary statistics from the time-stamped event data in the TSData field of each active session for each active subject and puts it in a field at the session level whose name is specified by the first argument of TSsessionstat (*statname*). It uses a helper function to compute the statistic. Most often, the helper function is TSparse, but it can be any function that the user has written, provided, of course, the user’s helper function conforms to the requirements. The helper function is specified by the second argument (*funhand*) in the call to TSsessionstat. The remaining arguments in the call to TSsessionstat are actually arguments for the helper function, that is, TSsessionstat passes them into the helper function when it calls the helper to make a calculation. The helper function passes the results of its calculation back to TSsessionstat, which, as already noted, stores those results in the Experiment structure at the session level under the title ‘*statname*’ (Experiment.Subject(#).Session(#).<statname>).

This calling of a helper function and the passing of arguments to it is a complex business. Understanding why things are done this way may help to grasp the logic, that is, the way in which the necessary tasks are divided up between TSsessionstat (the “boss”, so to speak) and the helper (and the unsung helper of the helper, which is usually TSmatch). The boss’s job is to consult the fields that specify the currently active subjects and sessions—we have not yet described these fields and how they are manipulated—and then to go through each active session for each active subject, copying the data in the TSData field, passing that data to the helper function, receiving back from the helper the results of its calculations, and putting those results in the user-specified field. The helper’s job is to search through each array of tsdata that it is given (one array for each active session for each active subject) for matchcode sequences, perform computations on the found sequences, and return the results of those computations to TSsessionstat. The helper function can in principle have any structure that the user cares to give it, *provided that it accepts 2-col arrays of time stamped records and returns an array of results*! In practice, however, the helper function almost always calls on TSmatch to do a search for one or more matchcode sequences. Searching for sequences is TSmatch’s job! This means, however, that the helper function must give TSmatch the matchcode sequences for which it is to search. The helper function gets these sequences passed into it from TSsessionstat. Thus, the call to TSsessionstat almost always has at least 3 arguments: the first one specifies the name of the field into which the computed results array is to be placed; the second one specifies the helper function that will compute the results; and the third one specifies the matchcode sequences that are to be searched for and that will serve as the input to the helper function that computes the results. If the helper function that computes the results is TSparse, as is most often the case, then TSsessionstat must also tell it what computation it wants made. It does that either by passing to it an mcode argument (a bit of text in single quotes that is treated internally as a matlab command or sequence of matlab commands—see TSparse, Chapter 9) or by passing to it the helper of the helper, that is a handle on the function that is to carry out the actual calculations.

TSsessionstat requires as arguments the name of this new statistic field (statname), the function handle to the Matlab code to be executed (funhand), as well as any other additional arguments that your helper function might need. Much like TSparse (see Chapter 9), you must find or create a Matlab function that will perform the desired actions. Here, we will use a simple function that passes in a vector of three events, and then counts each time any one of them occurs, without distinguishing one from the other, so that the count is the total number of times that any of those 3 events occurred. We will use Feed1, Feed2, and Feed3, so in effect we are computing the statistic for the number of feedings.

*function result = countevents(tsdata,eventcodes)  
result = sum(ismember(tsdata(:,2),eventcodes)); % total #of feeds.  
% ismember(tsdata(:,2),eventcodes) flags with 1’s all the instances in which a value in column 2  
% of tsdata is a member of the set of events specified in the eventcodes vector [Feed1 Feed2  
% Feed3]*

This helper function returns the total number of feedings, regardless of where they occurred. In order to store this statistic in the Experiment structure, we will call TSsessionstat in the following way:

TSsessionstat(‘TotalSesFeedings’,@countevents,[Feed1 Feed2 Feed3])

The first argument, ‘TotalSesFeedings’ is the *statname* argument, the name of the field that will be created at the session level to contain the statistic. Next is the function handle, which here is the handle to the *countevents* function. Third and last comes the vector of events over which the count is to range. When this line of code is executed, a ‘TotalSesFeedings’ field will appear in every (active) session, and will contain the total number of feedings in that session.

As we have noted several times, TSsessionstat most often calls TSparse as its helper. TSparse is explained in Chapter 9. Say that you want to determine the intervals between a Feed1 event and a Feed2 event, but only when there are no intervening feeding events, that is, only in those cases where a the next feeding event after a Feed1 is a Feed2. As we saw in Chapter 9, TSparse parses the data according to the specified sequence(s) (here, [Feed1 –Feed1 –Feed3 Feed2]). TSparse will then apply either a function handle or a line of mcode to the parsed sequences; because we want to find the length of time, we will use ‘result=time(2)-time(1)’. TSparse will compute this duration for each matched sequence, and so the final result vector (stored in the newstat field at the session level) will contain the duration of each match. For example, if there are 10 instances of the match, the new statistic field will contain a vector of length 10, with each item reflecting the duration of the corresponding match (first, second, third…). To do all of this, make the following function call:

TSsessionstat(‘Feed1andFeed2Intevals’, @TSparse,…  
 ‘result =time(2)-time(1);’,{[Feed1 -Feed1 -Feed3 Feed2]})

Recall that the first argument (‘Feed1andFeed2Intevals’) is the name of the field where the resulting statistic will be stored; a field with this name will appear at the session level of every active session of every active subject. The second argument is the handle for the function to be used to make the computation. Everything after the function handle are additional arguments that are passed to the helper function (we know that TSParse requires mcode and a cell array of event sequences, so those are the additional arguments). Each time the helper function, TSparse, computes the duration of the sequence, it appends the result to the ‘results’ vector, which TSsessionstat then stores in the field named ‘Feed1andFeed2Intevals’.

## Chapter 11: Defining trial types and Computing Trial Statistics (TSdefinetrialtype & TStrialstat)

## Defining a trial type

TSdefinetriatypel(trialname, matchcodes);

In the TS system, a trial type is any pattern of events that recurs within a session and that can be used to delimit chunks of tsdata, within which the user can then search for other patterns (other match code sequences). Trial types are user-defined; you can create any subdivision of the raw data that you would like and deem it a trial. You could, for example, have inter-feeding trials, where the trial would run from one feeding to the next. A trial could be defined as every time the subject visits all of the different feeders, or as the span between when a light turns on and when it turns back off. A trial definition is simply some way of segmenting the tsdata into chunks. Using these definitions, statistics can then be computed for each trial, using the TStrialstat command. In other words, by defining trial types, you can search for sequences that recur within a larger sequences that also recur, event structures within event structures. The word ‘trial’ in ‘trial type’ is used in an extremely generic sense. It does not necessarily refer to what researchers familiar with animal learning experiments might understand by a trial. Although it can and often does so refer, it *can* refer to any recurring patterns of events that are to be isolated so that the user may search for sub-sequences of events that may occur one or many times within each instance of the larger sequence.

Often trial types are identifiable by specific start and end events so that the trials are also apparent to the subjects. For example, a trial type might start with a ToneOn event and end with a Feed1 or a Feed2 event. In this case, a suitable trial type definition would consist of the match codes {[ToneOn Feed1] [ToneOn Feed2]}. Notice how a single trial type can be defined by more than one sequence of events; here, a cell array contains two different sequences defining the same trial type. To create a trial type definition, use the *TSdefinetrialtype* function. This function takes in two arguments: first, the name you lassign to this trial type; second, a matchcode vector or a cell array of match code vectors that define that type of trial. TSdefinetrialtype always adds ‘Trial’ to the beginning of the name passed in as the first argument; in the following example, the name of the trial type created will be referred to as TrialFeeding. Multiple match codes can constitute the same trial type; in this example, we want to see when a subject is fed. Because there are two feeders, you have to account for both of them. We would make the following function call in the Matlab command window:

TSdefinetrialtype('Feeding',{[ToneOn Feed1] [ToneOn Feed2]});

Executing the above line of code adds a field containing the trial type definition to the Experiment structure at the Experiment level (Experiment.TrialFeeding) If ToneOn = 61, Feed1 = 21, and Feed2 = 22, then Experiment.TrialFeeding = {[61 21] [61 22]}, that is, the field ‘TrialFeeding’ has been added to the Experiment level and it contains the cell array whose two cells contain the eventcode vectors specifying the two different sequences that are to be regarded as defining that type of trial.

Trial type definitions may use any legitimate matchcode sequence, including sequences with negative events (see Chapter 8 for lengthy explanation of matchcode sequences). Each instance of a trial type compasses the range specified between the first event code in the sequence and the last. For example:

TSdefinetrialtype('NoFeeding',{[LightOn1 –Feed1 LightOff1]});  
% Create a trial type called "NoFeeding"

This defines a trial type that begins with a LightOn1 event and ends with a LightOff1 event but does not have any Feed1 events intervening; hence, a ‘No Feeding’ trial.

You want your matchcode sequences to be as descriptive as possible so they find all and only the trials you want to find. Thus, when you write the experiment-control code, you need to think about the trial types you will want to analyze. It’s important to have a different code in the raw data for each light bulb/tone/feeder/etc. For example, say every time that the subject is fed, the event code “Feed” is activated. For some purposes, this may be fine. However, if there is more than one feeder and you want to define a trial type specific to one of them, or a trial type where the subject moves from one feeder to the other, there is no way to differentiate between the two. It will be impossible to determine this after the experiment as been run. Defining trial types in the data analysis can be greatly simplified if these more specific codes appear in the raw data. It is a good idea to use as many codes that you think may prove useful later; in this case, less is not more.

In addition to defining a trial type, TSdefinetrialtype also automatically makes the newly defined trial the active trial by calling TSsettrialtype (see below).

## TSsettrialtype

At the Experiment.Info level, the ‘ActiveTrialType’ field displays the name of the trial type that is currently active. Commands that compute statistics at the trial level (or aggregate statistics from the trial level) always refer to this field and confine their operation only to the trials of the active trial type. You can change which previously-defined trial type you would like to be active by using the TSsettrialtype function. TSdefinetrialtype (explained above) creates the trial type, and in doing so adds a field at the Experiment level. The name of that field is the trial’s title; the content of the field is the vector(s) of numerical event codes that constitutes the definition. The user may define arbitrarily many types of trials. However, at any one time, only *one* type of trial is the active trial type. When you define a new trial type, it automatically becomes the active trial type. Use TSsettrialtype to change the active trial type when the currently active trial type is no longer the trial type on which you want a command or sequence of commands to operate.

For example, if we use TStrialstat to calculate the number of feeding events, it wouldn’t be relevant to trials that measure each meter the subject has traveled; for this reason, when calculating statistics specific to certain trials, it is only applied to the specific active trial type.

TSsettrialtype('Probe') or TSsettrialtype(‘CS’);

This call to TSsettrialtype sets the active trial type to ‘Probe’, which must be a previously defined trial type, whose definition has been stored in the Experiment structure (using TSdefinetrialtype; see above). Following this type-setting command, when any functions are used to analyze the data in the structure, they will act only on ‘Probe’ trials. When you select the ActiveTrialType field at the Experiment.Info level, you will see that it indicates ‘TrialProbe’ as the active trial type.

If there are no trials defined yet, the ‘ActiveTrialType’ field at the Experiment.Info level is automatically set to the string 'none'. If you are not operating on statistics at the trial level (e.g., if you are using functions such as TSsessionstat, which only creates statistics at the session level), setting the active trial type to the “none” will make the helper functions run faster. It will run significantly faster, especially if the Experiment structure contains a large amount of data, since the function will not need to loop through and check each trial. Remember, TSdefinetrialtype automatically calls TSsettrialtype, making a newly defined trial the currently active trial (see above).

## Statistics From TSData at the Trial Level- TStrialstat

TStrialstat(newstat, funhand, FunHandArg1, FunHandArg2…)

TStrialstat is used much in the same way as TSsessionstat (see Chapter 10), but instead of applying to an entire session’s tsdata, the statistics created with TStrialstat apply only to individual trials (and thus the statistic appears at the trial level of the Experiment structure). It applies a specified function (the *funhand* argument) to each instance of the active trial in the structure. It then stores the result of its calculations under a new field (the *newstat* argument) at the trial level in the Experiment hierarchy (Experiment.Subject(#).Session(#).<trialname>.Trial(#).<newstat>) . TStrialstat requires as arguments the name for the new statistic (newstat) and a function handle to the helper function you want to use to produce the new statistic (funhand), as well as any other additional arguments that may be required by the helper function. In addition, TStrialstat stores information about the number of trials found, as well as the start time, end time, and duration of each trial.

We will use the same ‘countevents’ function used in Chapter 10 (reproduced below).  
  
*function result = countevents(tsdata,eventcodes)  
result = sum(ismember(tsdata(:,2),eventcodes)); % total #of feeds.  
% ismember(tsdata(:,2),eventcodes) flags with 1’s all the instances in which a value in column 2  
% of tsdata is a member of the set of events specified in the eventcodes vector [Feed1 Feed2  
% Feed3]*Each time TStrialstat calls the countevents function, it passes to it only the tsdata from a single trial of the currently active trial type (see above for information on defining trials and setting active trials, and TSlimit for setting limits on indexed trials). *Countevents* returns the total count over the specified the events within the one instance of that trial type. TStrialstat uses TSmatch to find every instance of the currently active trial type, calls *Countevents* to count the events in each instance, and stores the resulting counts (one for each instance of the trial) in an indexed sequence: Trial(1), Trial(2), etc.

Now we use the same helper function we used in Chapter 10, but we call it from TStrialstat, not TSsessionstat. Using TStrialstat and this helper function and the same three events (Feed1, Feed2, and Feed3) we create a field named ‘TotalTrialFeedings’, at the trial level. However, we use a different field name from the one we used in Chapter 10 (TotalTrialFeedings vs. TotalSesFeedings). It is good convention to give every field in the Experiment hierarchy a unique name; if two fields at different levels share the same name, some functions will only act on the higher-up field and/or crash when they encounter a second field of the same name. It also leads to user confusion when a single name refers to two different statistics.

TStrialstat(‘TotalTrialFeedings’,@countevents,[Feed1 Feed2 Feed3])

When this line of code executes, a ‘TotalTrialFeedings’ field appears the Trial level below the active trial type, in each *active* session of each active subject (see Chapter 14 for the function that sets the active subjects, sessions and trials). To see the results for one trial, you may type  
  
Experiment.Subject(#).Session(#).<trialname>.Trial(#).TotalTrialFeedings,  
  
but it is generally much easier to click on TotalTrialFeedings at the trial type level in the Browser. Each indexed trial of that type will appear in the panel to the right. Click on an individual trial, and in the bottom left pane you will see statistics about that trial, including the one we just made. First you will see the start and end times of the trial in terms of the session time, as well as the duration of the trial. Next will be the start and end locations of the trial in the session’s TSData. For example, if the trial’s first event occurred at line 207 and the last event occurred at line 228 in the session’s tsdata, ‘sloc’ will have a value of 207 and ‘eloc’ will have a value of 228. After these five default statistics (StartTime, EndTime, Duration, sloc, and eloc) the new statistic (in this case, TotalTrialFeedings) will appear and it will contain the total amount of times any of the three specified feeding events occurred.

Let’s look at another example to see how TSparse would be executed as an internal step in TStrialstat. Again, the procedure used by TStrialstat is very similar to that used by TSsessionstat; its statistics however only apply to individual trials of the active trial type. We will call TStrialstat to create a new field called ‘InterPokeIntervals’, and use TSparse as the helper function. We provide TSParse with mcode to apply to the data as an extra argument. Here, the mcode creates a two-column vector, with the first column indicating which event code sequences has been matches, and the second indicating the duration between the two events (for more information on the privileged variables ‘match’, ‘time’, and others, see Chapter 9).

TStrialstat('InterPokeIntervals', @TSparse,…  
 ‘result = [match time(2)-Time(1)]',{[PokeOff1 –PokeOn2 PokeOn1] [PokeOff2 –PokeOn1 PokeOn2]}); % Creates a 2-col field at the trial  
% level named ‘InterPokeIntervals’; 1st col indicates which hopper;  
% 2nd col is the interpoke interval

# Chapter 12 Computing Statistics from Statistics- TSapplystat

TSapplystat(newstat, usestats, funhand, funhandArg1, funhandArg2…)

TSapplystat is the workhorse function underneath TSsessionstat (Chapter 10) and TStrialstat (Chapter 11). It takes data from one or more fields at one level in the Experiment structure (*usestats*), applies some function to them (*funhand*), and stores the results in one or more newly created fields (*newstat*) at the same level of the Experiment hierarchy as the usestat fields. TSapplystat also accepts additional arguments to be passed to its helper function.

Say you used TStrialstat or TSsessionstat to calculate the duration of each feeding event either in a session or within the trials of a given type. This would result in a field at either the session or the trial level that contains a vector of feeding durations (the duration of each feeding within a session or within a trial), which could be called something like ‘FeedingDurs’. Now that you have this statistic, you may want to find the average duration of these feeding events. You would use TSapplystat to do this; it takes already existing statistics and applies a function to them that computes one or more further statistics, which are stored in a new, different field *at the same level*. In this case, it would take the vectors of feeding durations, compute their mean and store the result in a new field at the same level of the Experiment hierarchy. The newstat argument (the name of the field in which the means are stored) could be something like ‘AvgFeedingDur’; the usestats field would be ‘FeedingDurs’. The helper function is Matlab’s ‘mean’ function (remember to prepend the ‘@’ to put a handle on it). In order to compute and store that statistic, one would make the following function call:

TSapplystat(‘AvgFeedingDur’,’FeedingDurs’,@mean)

The second argument, the *usestats* argument, can be either a single string that is the name of a field already in the structure or a cell array of strings that are the names of fields already in the structure and at the same level. If it’s a single string (the field name of one already computed statistic), the helper function (funhand) will use the contents of that field as its first argument. Any additional arguments the helper function may require are specified after the 3rd argument of TSapplystat, the argument that specifies the helper function. If *usestats* is a cell array giving the names of the fields in which more than one already computed statistics are stored, the contents of those fields are passed to the helper function *in the order the field names appear in the cell array*. The contents of the first field are passed as the first argument of the helper function, the contents of the second as the second, and so on. Again, any additional arguments that the helper function may need—beyond those specified by the string(s) in *usestats*—are specified after the helper function itself is specified. Because *usestats* can refer to multiple fields, a single call to TSapplystat can compute a statistic or statistics that integrate the information in more than one field. However, if your usestats argument is a cell array of field names, ***all the fields MUST be at the same level***. If you try to use statistics from two different levels in the same call to TSapplystat, it will crash.

*Newstat* can also be either a single string specifying a single output field or a cell array or strings specifying several output fields. In other words, TSapplystat can accept statistics from several different fields—provided always that they are at the same level—and it can put results into several different output fields. The output fields will be at the same level as the input fields. If multiple *newstat* strings are given, a field will be created for each one. ***The helper function must be written so as to accept the number of arguments that TSapplystat will pass to it and so as to output all and only the number of arguments that TSapplystat expects to get back from it*,** which is to say, the same number of outputs as there are strings in *newstat*.

If *newstat* consists of two single quotes (an empty string, ‘ ‘), then the helper function will be called, but no new output statistic will be created. This is useful for calling functions that do not generate outputs. Most commonly, these are functions that create graphs from the data in the usestats field(s).

If *statname* and *usestat* are the same, TSapplystat applies the helper function to the data in *usestat* and replaces the data in *usestat* with the computed statistic. Unless space is a major concern, however, we strongly discourage this, because there is substantial benefit to keeping a record of the computations made in the course of the analysis. If you overwrite the old statistic, the original data is erased and you won’t have a complete record of the intermediate values used. Look at the following TSapplystat function calls and see how they create new statistics:

TSapplystat(‘TSdata’, ‘TSdata’, @Cleanup);  
% Applies the helper function Cleanup, to the contents of the TSData  
% field, and replaces the original uncleaned data in TSData with the cleaned data. NOT A GOOD IDEA!! Breaks the data trail

TSapplystat(‘CleanTSdata’, ‘TSdata’, @Cleanup);  
% Applies the helper function Cleanup to the contents of TSData  
% and stores the cleaned data in CleanTSdata. THE RIGHT WAY TO DO IT

TSapplystat({‘IRI\_BinCounts’ ‘IRI\_BinCenters’},‘IRIs’, @hist,20);  
% Applies Matlab’s hist function to the data in the IRIs field  
% (Inter-Response Intervals) and stores the counts in a field named  
% IRI\_BinCounts and the bin centers in a field named IRI\_BinCenters. % The last argument, 20, is passed to hist as its 2nd input argument, % which specifies the number of bins in the histogram. THIS CODE DOES  
% NOT GENERATE THE GRAPH!

TSapplystat(‘’,‘IRIs’,@hist,20); % Applies Matlab’s hist function to  
% the data in the IRIs field to generate a 20-bin histogram  
% (the graph itself). This graph is the empirical distribution  
% of inter-response intervals. THIS CODE DOES NOT CREATE A NEW FIELD  
% AND IT DOES NOT STORE ANY RESULTS IN THE EXPERIMENT STRUCTURE. When  
% Matlab’s hist function is called without output arguments, it  
% generates the graph; when it is called with output arguments,  
% it outputs the vectors required to generate the graph (the counts  
% and the bin centers) but it does not generate the graph

TSapplystat(‘PokeRate’,{‘NumPokes’ ‘Duration’},@rdivide) % On the  
% assumption that there are NumPokes and Duration fields at the  
% trial level for the active trial type, containing, respectively,  
% the number of pokes and the duration of a trial, this adds  
% a field named PokeRate at the trial level, and puts into it the  
% rate obtained by dividing the number of pokes during a trial  
% by the duration of the trial. The helper function is Matlab’s  
% rdivide function, which implements the •/• operation

Sometimes it is desirable to have TSparse make a match but not actually add any information to the cumulative result record. This can be done by setting the result to []. In the following example, we will find the session times of Feed1 events that happen when Light2 is not on:

TSsessionstat('SesFeed1NoLight2', @TSparse,  
 'if (match==1) result =[]; else result = [time(1)]; end;',…  
 {[LightOn2 –LightOff2 Feed1] [Feed1] 's'});

Here, we are looking for two sequences: 1) when Light2 turns on and there is a Feed1 event before Light2 turns off, and 2) each Feed1 event. Looking at the mcode that gets passed in, you will see that if TSparse is dealing with a match for the first match sequence, it will not add anything to the result vector (result = []); however, if it is a match to the second match sequence, it adds the time it occurred to the results vector (result = [time(1)]). Therefore, the final vector will only contain timestamps for Feed1 events that were not preceded by a LightOn2 event, hence the field name “SesFeed1NoLight2.” You will also see that this function call uses the ‘s’ flag, which is described in Chapter 8.

As you may have noticed, TSapplystat works very similarly to TStrialstat and TSsessionstat. In fact, when computing trial and session statistics, those functions actually call TSapplystat to do most of the work. TStrialstat and TSsessionstat simply pass the trial or session’s TSdata as the usestat field, making it so that the statistic is created directly from the data. When calling TSapplystat directly though, you can indicate any field at all as the usestat argument, as we have seen in the previous examples.

## Using Helper Functions

When TSapplystat creates a new statistic, it is the helper function specified in the call that actually does all of the calculations. Therefore, it is important to know what helper function to use in order to produce your desired statistic. Matlab comes with a wide variety of built-in functions that are very helpful and can be widely used. For example, as we used in the first example in this section, the Matlab function ‘@mean’ takes a vector of numbers and finds the arithmetic mean of all the contents. Perhaps an even more helpful Matlab function, @nanmean, finds the arithmetic mean of the contents of a vector, but first removes any NaN (not a number) values. The @sum function, as you might expect, takes a vector and sums all its contents; similarly, @nansum finds the total sum after removing any NaN members from the vector. The Matlab function @numel evaluates the number of elements in an array, which is often very helpful. @length and @size are similar, but it is important that you know when to use which one. @length takes an array and returns its largest dimension; @length would return 6 for a 1x6 array, and a 6x1 array, and a 4x6 array. The @size function returns both dimensions of the array; a 1x6 array would return [1 6]; a 6x1 array would return [6 1]; and a 4x6 array would return [4 6]. Detailed documentation on all of Matlab’s built-in functions can be found on the MathWorks website (http://www.mathworks.com/help/matlab/).

Often, though, you will wish to compute a statistic that cannot be computed with any single one of Matlab’s built in functions; in that case, you write a custom function. When you then call your custom function as a helper function, you must remember to prepend the handle symbol (‘@’) to the name you have given it. Custom functions are commonly very simple (only one or two lines of code). However, they can be arbitrarily complicated.

*It is important to take all possible inputs into account when writing a custom helper function*. For example, never implicitly assume that the field whose contents get passed to the function always has something in it; always make provision in your function for the case where its input argument(s) is/are empty. Generally speaking, your helper function should check immediately whether an input argument is empty, and, if so, to assign [] to the output and terminate further computation. Failure to provide an output when passed an empty input will cause a crash. Remember an empty output is not the same thing as no output. Generally speaking, when the helper function returns an empty output, the calling function (e.g., TSapplystat) simply enters empty into the specified output field.

Here’s another example of taking all possible inputs into account: Say you have a field that usually contains a lengthy 2-column array. If you want to count how many rows there are, you would use TSapplystat and include a helper function. Your first instinct may be to use Matlab’s ‘length’ function, but, as explained above, the default ‘length’ function returns the length of the longest dimension of an array. If an array, A, is 1x2 (one row and 2 columns) length(A) returns ‘2’, not ‘1.’ Matlab’s ‘size’ function returns both dimensions (rows and columns), but it can takes an optional second argument, either ‘1’ or ‘2’, which is used when you want it to return *only* the first dimension (rows) or *only* the second dimension (columns). In this case, where we want to count the rows, we would include ‘1’ as an argument in TSapplystat, following the call to @size, so that it gets passed to size as a second argument (the first argument being the contents of the usestat field). If we are naming our new statistic ‘NumFeedEvents’, we would make the following function call:

TSapplystat(‘NumFeedEvents’,’FeedEventsAndTimes’,@size,1)

This is a normal call to TSapplystat except for the final argument (‘1’) at the end, after the call to the helper function. This last argument (indeed, any and all arguments after the call to the helper) is/are passed to the helper function as additional input arguments. They are passed in the order they appear in the sequence of TSapplystat arguments (likewise, for all other TS functions that call helper functions).

# Chapter 13 Compiling Statistics (TScombineover)

TScombineover(statname, usestat, [modeflags]);

The function of TScombineover is to aggregate data from one level into a field at the next level up in the hierarchy. If you have data for each trial about how many times an event occurred, but you want to know the *total* for all the trials (the entire session), you would use TScombineover to create a session-level statistic from a trial-level statistic. For example, say you have a statistic in each indexed session that contains an array of the durations of each poke; it may be called something like ‘SesPokesDur’. After using TScombineover, a field would appear at each active subject’s subject level, which could be called something like ‘SubPokesDur’, and which would simply be a concatentation of the data in each session’s SesPokesDur fields. TScombineover accepts up to three arguments, although the third is not required. To produce this statistic, we would make the following function call:

TScombineover(‘SubPokesDur’,‘SesPokeDur’)

This will produce a subject-level statistic that contains all of the poke durations from all of the sessions in a single vertical array. The first argument, *statname*, is the name of the new statistic at the higher level. The second argument, *usestat*, is the name of the statistical field that you want to use to create this new statistic. Because TScombineover creates an array from all the different *usestat* fields, it is important that each array in a field with a given name has the same width. If each array does not have the same width, you must use the ‘c’ flag to create a cell array (see below). The “next higher level” is the session level when the *usestat* fields are at the trial level (NB *not* the Trial-Type level); it is the subject level when the *usestat* fields are at the session level; and it is the Experiment level when they are at the subject level. In other words, it is the next higher *numerically indexed* level (trial types are not numerically indexed).

TScombineover allows for an optional third string argument, the mode flags. This string may contain one of three character "flags" that change the way TScombineover works. The ‘t’ flag is used to tag data; the ‘m’ flag is used to merge data, and the ‘c’ flag is used to create a cell array instead of a regular array.

To build off of the example above, let’s create some sample data for the SesPokesDur and SubPokesDur field. For this particular subject, Subject 5, we’ll make three sessions. We’ll use the following data:

Experiment.Subject(5).Session(1).SesPokesDur =

.03  
 .05  
 .02  
 .04

Experiment.Subject(5).Session(2).SesPokesDur =

.02  
 .02  
 .04  
 .05

Experiment.Subject(5).Session(3).SesPokesDur =

.03  
 .05  
 .04  
 .02

Using the same function call as above (TScombineover(‘SubPokesDur’,’SesPokesDur’)), we would create the following field with the following data:

Experiment.Subject(5).SubPokesDur =

.03  
 .05  
 .02  
 .04  
 .02  
 .02  
 .04  
 .05  
 .03  
 .05  
 .04  
 .02

### Tagging data with the 't' flag

Note that when using TScombineover, there is no way to tell from which trial or session or subject each datum came. In the above example, you are not able to tell which session each poke duration came from simply by looking at the field. Using the ‘t’ flag as the third argument instructs TScombineover to add a column to its output, which column contains the index number of the trial, session, or subject that each datum came from. This extra column will be added on to the right hand side of the returned array. Using the same example as above, we will make the following (slightly different) function call to create the following data field:

TScombineover(‘SubPokesDur’,’SesPokesDur’,’t’);

Experiment.Subject(5).SubPokesDur =

.03 1  
 .05 1  
 .02 1  
 .04 1  
 .02 2  
 .02 2  
 .04 2  
 .05 2  
 .03 3  
 .05 3  
 .04 3  
 .02 3

### Merging times with the 'm' flag

When you use TScombineover to aggregate fields that contain times referenced to the start of a session (session times, time elapsed in a session) or to the start of a trial (trial time, time elapsed in a trial), you may wish to rework the times so that they cumulate across sessions and/or trials. You may want the first time stamp from an event in a later session not be the time elapsed since the start of that later session but rather time elapsed in the course of the entire experiment, cumulating time across all prior sessions. This cumulative session time is the time elapsed on a timer that does not reset itself at the end of a session; rather, it stops where it is and resumes ticking when the next session starts. Similarly, for cumulative trial times; they are the times elapsed on a timer that only runs when a trial is in progress, but does not reset when a trial ends; rather, it stops where it is and resumes ticking when the next trial begins. The ‘m’ flag causes TScombineover to rework the time stamps from successive sessions or successive trials to create such timers (post hoc).

To see how this works, suppose we are combining a field named ‘Pokes’ at the trial level. It contains sequences of poke times, where the time in question is the time elapsed since the beginning of the trial. This same field is found in every trial of a given type. The data for the first two trials of type ‘CS’ might look like this:

Experiment.Subject(1).Session(1).TrialCS.Trial(1).Pokes =

2.3  
 3.9  
 5.8  
 8.9

Experiment.Subject(1).Session(1).TrialCS.Trial(2).Pokes =  
 4.6

19.1

First, we call TScombinover without the ‘m’ flag

TScombineover(‘Pokes\_ses’,’Pokes’,’t’) % We have added a ‘t’  
 % flag so we will no which trial each datum came from

The resulting field, which will be at the session level (the first indexed level above the trial level), will look like this:

Experiment.Subject(1).Session(1).Pokes\_ses =

2.3 1  
 3.9 1  
 8.8 1  
 10.9 1  
 4.6 2  
 19.1 2

Notice that the aggregated times do not cumulate; they are not strictly increasing, because the timer in effect resets after each trial. To rework these trial times into the elapsed times on a non-resetting trial timer, we need to know how long each trial lasted. That information will always be found in the Duration field at the trial level. Whenever there are any trials of a given trial type, then a ‘Duration’ field is found under each instance of that trial type, because TStrialstat creates this field and fills it with the computed trial duration when it computes the first user-specified statistic for a given trial type. Suppose that the duration of the Trial 1 was 14 s and the duration of Trial 2 was 20 s and we use the ‘m’ flag to get TScombineover to create a non-resetting trial timer:

TScombineover(‘Pokes\_ses’,’Pokes’,’tm’) % Now we also use the  
 % ‘m’ flag

The resulting field will look like this:

Experiment.Subject(1).Session(1).Pokes\_ses =

2.3 1  
 3.9 1  
 8.8 1  
 10.9 1  
 18.6 2  
 33.1 2

Notice that the duration of the first trial (14 s) has been added to each time that comes from the second trial. If there is a third trial, then the combined duration of the first two trials (14 + 20 = 34 s) is added to every time stamp from the third trial, and so on. When you aggregate from the session level to the Experiment level using the ‘m’ flag, the same thing is done, except that TScombineover looks at the Duration field under each session and uses those durations to make a non-resetting session timer. (The TS system always computes the duration of a session and puts it in a Duration field at the session level, one field under each session for each subject. It does so simply by looking at the time stamp of the last event in tsdata.)

### Combining arrays of different widths using the 'c' flag

Normally, TScombineover takes each array of data and appends it beneath the previous one, creating an ever lengthening array. If the arrays in fields with the same name have different widths (different numbers of columns) for different subjects (or for different sessions within a subject of for different trials within a trial type), then Matlab will not be able to append them to make a single array, because Matlab only accepts rectangular arrays. If you know that some of the fields whose contents are to be aggregated have differing numbers of columns, use the ‘c’ flag as your third argument in TScombineover to create a cell array rather than a simple array. The cell array will have as many rows as there were instances of the aggregated field (that is, as many as the number of subjects, or sessions, or trials across which the aggregation extended). The first cell in each row will contain the contents of the first subject (or session or trial), the second the contents of the second, and so on. The cell array will have only one column. This is a rarely used option, and the browser may not be able to display the resulting field, because the browser assumes that data fields contain only arrays, not cell arrays.

### Using two flags at once

One can use more than one of these flags simulatneously. Simply put the two letters together as one string. For example, say you want to create a non-resetting timer column while also tagging the data to keep track of which trial each set of data came from (that is, use both the ‘m’ flag and the ‘t’ flag). Your third argument would simply be either either the two-character string ‘mt’ or ‘tm’. The order doesn’t matter (‘t’ first or ‘m’ first); it just tells TScombineover that both special functions (merging and tagging) will be applied to the specified fields. Similarly, you could use the cell and tagging flags together (‘ct’ or ‘tc’) to add another column in each array indicating which index the data came from, and then bring all of these arrays (most likely of different widths) together in a cell array.

# Chapter 14. Limiting the Scope of Operations—TSlimit

TSlimit(basis,limits);

The TSlimit command restricts the subjects, sessions, phases or trials operated on by TStrialstat, TSsessionstat, TSapplystat and TScombineover. When you want to analyze only a specific subset of the subjects, or of the sessions, or of the phases (protocols, experimental conditions), or of the trials of a given type, with a sequence of TS commands, you put one or more TSlimit commands at the beginning of that sequence of commands whose operation you want to restrict. The first argument of TSlimit, *basis,* specifies the basis of a restriction (Subjects, Sessions, Phase, or Trials)—about which more later. The second argument, *limits*, specifies which elements within a given basis are to be active (that is, are to operated on by subsequent commands). *There is one special case in which TSlimit required only one argument:* calling *TSlimit(‘all’)* makes all data in all fields available, meaning that there are no restrictions and any subsequent functions will operate on all of the data in the Experiment structure. (However, remember that commands that operate at the trial level always operate only on trials of the active trial type, and only one trial type can be active at any one time.) ‘all’ can also be used as the second argument following a first argument, in other words, as the value of limits. For example, after the call TSlimit(‘Trials’,’all’), all trials of the currently active trial type will be operated on.

When you want to limit on more than one basis, say, only the last 20 trials from certain sessions with certain subjects, you use a sequence of TSlimit calls, one call for each basis.

A simple TSlimit call has as its second argument a vector that specifies the indices or numerical phase names that are to fall within the scope of operation of subsequent commands. Thus, TSlimit(‘Subjects’,1:6) limits the scope of subsequent operations to the first six subjects; that is, it makes them and only them the “active subjects.” TSlimit(‘Sessions’,[2 5 7 10:15]) limits the active sessions to 2, 5, 7, 10, 11, 12,13, 14 and 15.

TSlimit(‘Phases’,[10 600]) limits subsequent operations to those sessions whose Phase field contains either the number 10 or the number 600. The number in the phase field session identifies the experimental protocol in force for that subject during that session. Depending on the context, ‘Phase’ might also be called ‘experimental group’ or ‘experimental condition.’ Phase numbers (that is, numbers that identify experimental group or condition), unlike index numbers, are really names not numbers. They are like the numbers on the jerseys of athletes. It does not generally make sense to say that the ball carrier was tackled by players 16 to 18, because the numbers of jerseys are not numerically ordered indices. Similarly, it does not generally make sense to specify a range of numerically coded phases (e.g., 3:6), whereas it always makes sense to specify a range of numerical indices for subjects, sessions or trials.

Limit specifications for indices can be specified relative to the first index or relative to the last index. If one wants to restrict operations to the second session, one calls: TSlimit(‘Sessions’,2); if one wants to restrict operations to the second to last session, one calls TSlimit(‘Sessions’,-2). To restrict to the last 10 sessions: TSlimit(‘Sessions’,-10:-1). TSlimit(‘Sessions’,Inf) has the same effect as TSlimit(‘Sessions’,-1); both limit analysis to the last session (for each active subject).

When the limits argument is a real number strictly between 0 and 1 (that is, a pure decimal), then the restriction is to the corresponding fraction of the range of indices. For example, TSlimit(‘Trials’,-.5) limits subsequent analyses to the last half of the trials of the active trial type. Suppose one subject has experienced 110 trials of that type while another has experienced only 55, then, for the one subject, subsequent analyses will be restricted to the data from Trials 56:110 while for the other subject, subsequent analyses will be restricted to the data from Trials 28:55. (When the fractional number of trials does not evaluate to an integer, Matlab’s round function rounds it to the nearest integers in accord with the usual rounding rules.) TSlimit(‘Sessions’,.1) limits subsequent analyses to the data from the first 1/10th of the sessions for each active subject. (It is difficult to imagine the circumstances under which it would make sense to use a decimal value for limits when limiting on the basis of subjects, but you can do it if you find it amusing to do so.)

At the Experiment.Info level, you will see—using the Browser—that there are fields titled ‘ActiveSubjects’, ‘ActiveSessions’, ‘ActivePhases’, and ‘ActiveTrials.’ (We here refer to the ‘ActiveTrials’ field, not the ‘ActiveTrialType’ field. The contents of the ‘ActiveTrialType’ field are set by TSsettrialtype, not by TSlimit—see Chapter 11). You can see which indexes are currently active by selecting these fields and viewing their contents. Because the number of subjects is constant throughout the experiment, the index numbers of the active subjects are stored here, in order and without duplication. However, the number of sessions may be different for different subjects, and the number of trials of a given type may differ from session to session, so index numbers of trials and sessions cannot be displayed accurately here. In those fields, there will be a vector that displays as many accurate index numbers as possible (positive integers or ranges that do not depend on the total number of sessions/trials). Any calculations that depend on knowing the total number of instances (total sessions, total trials of a given type, etc), such as Inf, negative integers, or percentiles, will remain in the form that you specified with the limits argument of TSlimit. As with all other fields in the Experiment structure, the contents of these fields can be edited directly from the Browser (or by commands in the Command Window). However, if any of these fields are set to ‘all’, you will see that they are recognized in the structure as a 1x3 char array. If you try to edit the field directly in the Browser once its contents are classified as characters, any numbers or vectors will not be recognized as integers; they will simply be read as another string. For this reason, once a field is defined as ‘all’, you will not be able to edit it directly in the Browser and *must* call TSlimit from the Matlab command window. **We do not recommend setting these fields directly**. **Always use TSlimit to change the contents of these fields**.

Instead of limiting subjects on the basis of their index numbers, TSlimit can limit them on the basis of their ID numbers, using the ‘a’ flag. This flag activates ‘absolute subject’ mode. It is only applicable to subjects (sessions, phases, and trials do not have ID numbers, that is, numerical names; they only have index numbers). In order to use this mode, the second argument, limits, must be a cell array, with two cells. The first cell must contain a vector of subject ID numbers, and the second cell must contain the character ‘a.’ For example, say that you are running an experiment with ten subjects whose ID numbers range from 116 to 125. To limit analyses to the subjects whose ID numbers are 117 and 122, you would make the following function call:

TSlimit(‘Subjects’,{[117 122] ‘a’})

As you can see, the first argument (the value of basis) remains the same. The cell array that is the second argument, however, appears in the ActiveSubjects field at the Experiment.Info level. Any subsequent analyses operate only on the data from the subjects whose ID numbers—NB, *not their index numbers*!—are 117 and 122.

# Chapter 15. Graphing Your Data

It is difficult to exaggerate the importance of effective graphics in revealing the story that your data have to tell and in making it apparent to others. The TS system supplies four graphics commands. They are intended to be simple when used simply but capable of generating complex graphs as your graphic ambitions increase. They are TSplot, TSplotcdfs, TSplotcumrecs, and TSraster. They make it easy to put multiple plots on the same axes and multiple panels in the same figure, thereby facilitating the creation of graphs that allow you to compare visually the results from different subjects and from different experimental conditions.

***With the exception of TSraster, these graphic commands must be called as helper functions by TSapplystat; they cannot be called directly!***

**TSplot** plots an arbitrary number of x-y plots, either as line plots or scatter plots.

Syntax: TSapplystat(‘’,usestat,@TSplot,varargin),

where varargin is a variable number of comma-separated arguments, as described below.

The goal in constructing these plotting functions is to make them very simple so that the new user can use them easily, but to allow the more sophisticated user, or the new user with increasing ambition about what she wants in the figure, to generate increasingly complex figural results in as intuitive way as possible. The additional complexity is achieved by adding arguments after the call to TSplot. The varargin is a place holder for these additional arguments. The sophisticated user who simply wants a reminder of these arguments is referred to the summary at the conclusion of this entry on TSplot.

You use TSplot in its simplest form by the following call:

TSapplystat('',*usestat*,@TSplot)

This call assumes that the *usestat* field contains (at least) 2 columns of data and that you want to plot the second column against the first. Notice that that the *newstat* argument (the first argument in the call) is two single quotes, because this command does not create a new field in the Experiment structure.

Because this call does not specify how many rows and columns of panels are wanted, the resulting figure(s) will have the default arrangement of panels per figure, which is 8 panels, arranged in 4 rows and 2 columns. As many figures are produced as are required to show all the data, given 8 panels per figure.

When the data in each plot come from a field at the Subject level, then each panel is labeled with the Subject’s index number. When the data come from a field at the Session level, each panel is labeled with the Subject index number and the Session index number. When they come from a field at the Trial level, each panel is labeled with the Subject index number, the Session index number, the Trial Type, and the Trial index number.

Because you have not specified labels for the x and y axes, they will be unlabeled. Unlabeled x- and y-axes are bad. You may know now what the axes represent, but other viewers often will not, and you are likely to forget. To get properly labeled graphs you use what Matlab calls ‘Variable-Value pairs. ‘ A Variable-Vale pair is a pair of arguments (inputs to a function) in which the first member of the pair is a text (aka string) identifying the variable and the second member is the value of that variable (which may be either a number or text). In this case, we have two variables for which we want to specify values, namely ‘Xlbl’ and ‘Ylbl.’ For the sake of example, assume that the values that we want to specify for these variables (that is, the labels themselves) are ‘Days’ on the x axis and ‘Weight’ (on the y). Variable-Value pairs may appear in any order, *provided only that for each pair, the value immediately follows the name of the variable.* (Like all input arguments, the value must be separated from the variable by a comma, because a Variable-Value pair is two inputs.) Using Variable-Value pairs, the call would be:

TSapplystat(''*,usestat*,@TSplot,’Xlbl,’Days’,’Ylbl’,’Weight’)

Having the x- and y-values come from the same field guarantees that they are equal in number. However, it often happens that the x-data are in one field and the y-data in another. *Assuming that the data are the same length in both fields* (!) and also, for the moment, that both fields contain only one column of data, then the simplest call would be:

TSapplystat(''*,{usestat1 usestat2}*,@TSplot)

The *usestat* argument in this call is a cell array with two cells containing the names of two different fields. *These fields* *which must at the same level of the hierarchy*. This call plots the data in *usestat2* against the data in *usestat1,* and it labels the axes with the respective field names. If you do not want those labels, override them using Variable-Value pairs, as in the previous example.

It often happens that the x- and y-data are not the first two columns respectively in one field or not the first columns in two fields, in which case you need to tell TSplot which columns are which. You do that using Variable-Value pairs. The variables are ‘Xcol’ and ‘Ycol’. Suppose that the x-values are in the 3rd column of the *usestat* field and the y values in the 2nd column. The call is:

TSapplystat(''*,usestat*,@TSplot,’Xcol’,3,’Ycol’,2,’Xlbl,’Days’,’Ylbl’,’Weight’)

When the x-data are in the 2nd column of *usestat1,* which for this example we assume to be a 3-column array and the y-data in the 5th column of *usestat2*, imagine that the two arrays, the 3-column array from *usestat1* and the 5-column array from *usestat2*, are horizontally concatenated and count the columns accordingly, thus:

TSapplystat(''*,{usestat1 usestat2}*,@TSplot,’Xcol’,2,’Ycol’,8)

(Remember, the arrays must have the same number of rows in order to be horizontally concatenated.)

If you want more (or fewer) rows of panels in each figure and/or more (or fewer) columns, you again have records to Variable-Value pairs. The variables are ‘Rows’ and ‘Cols’.

TSapplystat('',*usestat*,@TSplot,’Rows’,6)

produces figures with 6 rows of panels and 2 columns, while

TSapplystat('',*usestat*,@TSplot,’Cols’,1)

produces figures with 4 panels one beneath the other, and

TSapplystat('',*usestat*,@TSplot,’Rows’,7,’Cols’,3)

produces figures with 21 panels, arranged in 7 rows and 3 columns, while

TSapplystat('',*usestat*,@TSplot,’Rows’,1,’Cols’,1)

produces a figure with only one panel, as would be likely if the data plotted come from a field at the Experiment level.

Depending on the level of the hierarchy at which the field name(s) in *usestat* is/are found, each panel is labeled with ‘S#’ if the field is at the Subject level (# will be the index number of the subject, not the subject’s ID #), with ‘S#,s#’ if the field is found at the session level, and with ‘S#,s#,TT,T#’ if the field is found at the trial level (in each case, # is the index number; TT is for trial type). If you want the Phase to be included in the panel label, include ‘Phase’ in the sequence of input arguments after the call to TSplot. If the *usestat* field(s) is/are at the Session level or the Trial level, the phase information will be taken from the Phase field of the session from which the data come. If the *usestat* field is at the Subject level, then the Phase information will be taken from the Phase field of the first of the currently active sessions.

Real data often contain gross outliers that radically distort the graph, making it impossible to see most of the data. For this and other reasons, one may want to limit the y-axis range and/or the x-axis range. To do that, use the Variable-Value pairs ‘Xlm’ and ‘Ylm’. The values for either of these must be the same 2D vector you would use in Matlab’s xlim and ylim commands: the first value in each vector specifies the lower limit; the second, the upper limit.

In summary, TSplot syntax is:

TSapplystat(‘’,*usestats,*@TSplot,varargin)

The possible arguments in varargin are:  
 Xcol  
 Ycol  
 Xlbl  
 Ylbl  
 Xlm  
 Ylm  
 Scat  
 Phase

All but the last varargin arguments are Variable-Value pairs: the name of the variable, enclosed in single quotes, must be followed by a comma and then by its value, which is in single quotes only if it is text.

**TSplotcdfs**

In the experimental literature on behavior, the most common data graphing and analysis operation is the comparison of two or more sample distributions. The most common graphic form of this comparison is a bar plot, in which the heights of the bars portray the means of the samples and error bars indicate the standard errors of the means (or, more rarely, 95% confidence intervals). Bar graphs for comparing distributions flunk Tufte’s criterion for a good scientific graph, namely, that it display as much information as possible in as intelligible a way as possible in as little space as possible.[[2]](#footnote-2) In the same amount of space, plots of the cumulative empirical distributions convey much more information in a more readily intelligible form. TSplotcdfs creates graphs of with one or more plots of cumulative distributions on the same panel and with multiple panels per figure.

The plot of the cumulative distribution is the most informative way of examining the data in a sample. It is made by plotting the data on the x-axis after an ascending sort and, on the y-axis, the cumulative fraction of the sample data (see Table 1 and Figure 15-1). It conveys all the information in the raw data in a visually intelligible way. A histogram is the most common way of examining the distribution of data in a sample, but the appearance of a histogram depends critically on the arbitrary choice of a bin width. There are no choices involved in the plotting of the cumulative distribution. It is a maximally informative way of viewing your raw data.

The data samples in Table 15.1, which are plotted in figure 15.1, are drawn from two distributions with the same mean, so it is not surprising that a 2-sample t-test for a difference in the means—the most commonly reported statistic in the experimental literature—gives no significant difference. However, from the plots in Figure 15.1, it is obvious that these samples do not come from the same distributions. If these samples were compared with the usual bar graphs, there would be no evidence of this. One would not know that the data in the D2 sample spanned a much larger range than the data in the D1 sample. One would also have no idea of the sample sizes. In Figure 15.1, by contrast, it is visually obvious that the sample sizes were small, because the steps are prominent. A cumulative distribution makes an upward step at each datum. When the sample size is small, the steps are large; as the sample size increases, the steps become small and the curves become smooth. It is also obvious in Figure 1 that, although these two samples come from different distributions, the central tendencies of those distributions are similar. The median of a sample plotted as a cumulative distribution is the point on the x axis at which the plot crosses .5 on the y axis, in other words, the value such that half the data are less than that value. One sees at a glance that at their midpoints, these two samples are almost the same, which means that the sample medians are about the same. The forms of the distributions from which these data come are not unambiguously apparent with samples this small, but the educated eye would realize that the data in the black plot plausibly come from a normal distribution and those in the red plot from an exponential distribution.

*Table 15.1. Illustrating Computation of Cumulative Distribution*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **D1** | **D1 sorted** | **D2** | **D2 sorted** | **Cumulative Fractrion of D** | | |
| 15.105 | 6.8446 | 13.612 | 1.3305 | 0.09 | 1/11 |
| 6.8446 | 8.2688 | 8.6709 | 8.6709 | 0.18 | 2/11 |
| 11.938 | 8.999 | 11.853 | 11.362 | 0.27 | 3/11 |
| 10.529 | 10.529 | 33.507 | 11.853 | 0.36 | 4/11 |
| 8.2688 | 11.938 | 18.557 | 13.612 | 0.45 | 5/11 |
| 14.524 | 13.1 | 27.209 | 18.555 | 0.55 | 6/11 |
| 13.846 | 13.846 | 22.013 | 18.557 | 0.64 | 7/11 |
| 13.1 | 14.051 | 18.555 | 22.013 | 0.73 | 8/11 |
| 8.999 | 14.524 | 11.362 | 27.209 | 0.82 | 9/11 |
| 16.382 | 15.105 | 39.035 | 33.507 | 0.91 | 10/11 |
| 14.051 | 16.382 | 1.3305 | 39.035 | 1.00 | 11/11 |

**CumDistsIllustration.pdfFigure 15.1** *The empirical cumulative distributions for the illustrative data samples in Table 1. The fraction in the last column of Table 1 (given in decimal form in the 2nd to last column) is plotted against an ascending sort of the data, that is, against the data as displayed in Columns 2 and 4 of Table 15.1. Notice that there is an upward step in the cumulative distribution at each data value, so the approximate n in the samples is apparent simply from the size of these steps. Notice also that the central tendencies are similar but the ranges radically different. None of this would be apparent in a bar-graph comparison of the sample means.*

Like TSplot, TScdfplots must be called from TSapplystat. The syntax is:

TSapplystat(‘’,usestat,@TScdfplot,varargin)

where, usestat is the name of a data field or a cell array of names for data fields, *all of which are at the same level of the Experiment hierarchy*. The possible varargin are:

‘Rows’ ‘Cols’ ‘DataCols’ ‘Xlbl’ ’Xlm’ ‘Ylm’ ‘Phase’  
  
All of them are Variable-Value Pairs except ‘Phase’. Thus, if usestat is a cell array specifying two different fields named ‘Switch1’ and ’Switch2’ and one wants to plot the 1st and 3rd column in Switch1 and the 2nd col in Switch 2, one would use the ‘DataCols’ Variable-Value pair as follows:

TSapplystat(‘’,{‘Switch1’ ‘Switch2’},@TSplotcdfs,’DataCols’,…  
 {[1 3] 2})

When usestat is a cell array of field names and ‘DataCols’ is used to specify which columns are to be plotted, then the value of the ‘DataCols’ variable must be a cell array with the same number of cells as usestat, and each cell must contain a row vector specifying the numbers of the to-be-plotted columns from the corresponding field in the usestat cell array. If all the columns from both fields are to be plotted, then ‘DataCols’ need not be among the varargin.

If one wanted to specify a label for the x axis, one would add the ‘Xlbl’ and its value to the varargin, as follows:

TSapplystat(‘’,{‘Switch1’ ‘Switch2’},@TSplotcdfs,’DataCols’,…  
 {[1 3] 2},’Xlbl’,’Switch Latencies (s)’)

The y-axis in the plot of a cumulative distribution is always the cumulative fraction of the data, so ‘Ylbl’ is not included among the possible varargin. The ‘Rows’ and ‘Cols’ varargin allow the specification of the numbers of rows and columns of panels per figure, as in TSplot. The ‘Xlm’ and ‘Ylm’ varargin allow specification of the limits of the x and y axes, as in TSplot. Their values must be 2D row vectors with the first value in the vector <= the second, as in Matlab’s xlim and ylim commands.

The panels are automatically labeled when the data come from any level of the Experiment structure below the Experiment level. If they come from the Subject level, each panel is labeled with ‘S#’ where # is the subject’s index number. If they come from the Session level, each panel is labeled with ‘S#,s#’, where the 2nd # is the index number of the session. If they come from the Trial level, each panel is labeled with ‘S#,s#,<TT>,T#’, where <TT> is the name of the trial type and the 3rd # is the trial number. If ‘Phase’ is included among the varargin, then the number identifying the phase (aka Group or Condition) is included in the labels, prefaced by ‘P’. When the data come from the Session level or the Trial level, the phase information is taken from the Phase field of the session from which the data come. When they come from the Subject level, then the Phase information is taken from the Phase field of the first of the currently active sessions.

**TSplotcumrecs**

Behavioral and electrophysiological experiments that involve learning often focus on changes in behavioral and/or electrophysiological measures over time. The best plots for rendering such changes apparent and for comparing the times at which changes in two different measures occur are cumulative records. They were popularized for behavioral work by B.F. Skinner, who plotted the cumulative count of pigeons’ pecks or rats’ lever presses on a chart moving at a constant speed, thereby obtaining a plot of the response count as a function of time. The slope of such a plot is pecks/(unit time), which is to say, rate. More generally, the slope of a cumulative record is the derivative of the y-variable with respect to the x-variable: how much of a change in y is observed for a given change in x. Whenever one is interested in changes in central tendencies, either over time, or over successive trials, the cumulative record is the plot of choice. It accomplishes something almost paradoxical: it reveals changes in central tendency (that is, in the average value) without averaging. Thus, it reveals both when such changes occur and whether they are abrupt or gradual.

Figure 15.2 shows the utility of such records in showing behavioral changes in individual subjects.

CumRecs.pdf

**Figure 15.2**. *Cumulative records from individual subjects in different experimental learning paradigms.* **A.** *Cumulative records of the pecks delivered by each of 9 pigeons in an autoshaping experiment. (Data from the laboratory of John Gibbon.) Note the abrupt onset of pecking in each case.* **B.** *Cumulative records of the anticipatory blinks from 10 rabbits in an eye-blink conditioning experiment. Note the abrupt onset of conditioned blinking in each case and the steady maintenance of a blink probability somewhat less than 1 for many trials thereafter. (Data from the laboratory of E. James Kehoe.)* **C.** *Cumulative correct choices by a rat in a + maze (solid line). For first 20 trials bait placed in a randomly chosen one of the three non-start arms, so true probability of correct was .33 (chance). End of this phase is marked with a thin vertical line. Subsequently, there were 30 trials per session, with the location of the bait reversed after Trial 15 of each session. Locations of abrupt changes in probability of correct choice, as determined by a change-point detecting algorithm, marked with ovals. Dashed line plots the slopes between these change points, that is, the probability of a correct choice. (Data from the laboratory of Sheri Mitzumori.)* **D.** *Cumulative efficiency of the swim trajectories of a mouse learning to find the hidden platform in a water maze. The efficiency of a trajectory is the straight-line distance between the start location and the platform divided by the distance traveled by the rat in swimming to the platform. Note, here too, the abrupt onset of efficient swims (at Trial 3). For comparison, the thin solid line has a slope equal to the mean first-trial efficiency of all the mice tested—when they could not know the location on the first trial—and the thin dashed line has a slope equal to this mean + twice its standard error. (Data from the laboratory of Louis Matzel.)* **E.** *The cumulative number of cycles between two feeding hoppers as a function of elapsed session time in a naïve mouse placed in a matching paradigm with concurrent variable interval schedules running independently on the two hoppers. (Data from the Gallistel laboratory.)* **F.** *The cumulative duration of the visits to Lever 1 versus the cumulative duration of the visits to Lever 2 in a rat responding for brain stimulation reward on two levers that deliver rewards on concurrent variable interval schedules. The slopes of these plots show the extent to which the rat “prefers” (tends to stay at) one lever versus the other. Up to the point indicated by the vertical gray bar, Lever 1 delivered 9 rewards for every 1 delivered by Lever 2; at the gray bar, the relative rates of reward delivery abruptly changed to 1:1. The plot shows the abruptness of the rat’s adjustment to this change. The slope of the thin dashed comparison line is 9:1, so prior to the change the ratio of the average duration of the rat’s visits (“stays”) at Lever 1 to the average duration of its visits to Lever 2 approximately matched the ratio of the rates of reward it was getting from the two levers. When the ratio of the rates of reward abruptly changed, there was a rapid and abrupt comparable adjustment in the ratio of its average visit durations. The thin solid line shows the 1:1 slope. Because the slope of the plot approximately matches the slopes of these comparison lines before and after the change, the rat was matching both before and after the change, and it adapted to the change very rapidly. (Data from the Gallistel laboratory.)*

TSplotcumrecs makes the creation of plots like these simple, and it makes it simple to have multiple panels of plots like these on a single figure, so that one can see the results from many subjects at once. The syntax is:

TSapplystat(‘’,usestat,@TSplotcumrecs,varargin)

The recognized varargin are:  
'Dcols' 'XY' 'Xlbl' 'LeftYlbl' 'RightYlbl' 'Rows' 'Cols' 'LeftYlm' 'RigthYlm' ‘Phase’

All of the varargin are Variable-Value pairs, except ‘Phase’.

In the simplest case, usestat is a field containing a single column of measures. These measures may be successive event times, in which case they will be strictly increasing. Or, they may be measures made on successive “trials” (e.g, number of pecks, durations of visits). In both cases, the a second column vector is implicit, namely, the vector that counts the events or trials [1 2 3 4 …]. In the first case, where the times at which these events occurred are in the vector passed in, one wants a plot of the cumulative number of events on the y axis (e.g., cumulative pecks), as a function of elapsed time of the x axis, as in the cumulative records that Skinner popularized. In the second case, one wants the cumulative sum of the vector passed in plotted on the y axis (e.g., cumulative pecks) and the count vector on the x axis (e.g, Trial Count). TSplotcumrecs determines which is wanted by examining the single column vector of data that is passed to it: if the values in the vector are strictly increasing (Case 1), then they are plotted on the x axis and the implicit count vector is plotted on the y axis. If they are not strictly increasing (Case 2), then the cumsum of this vector is plotted on the y axis and the implicit count vector is plotted on the x axis.

This principle (strictly ascending data plotted against the x-axis with implicit count vector plotted against the y-axis; cumsum of data that are not strictly ascending plotted against y-axis with implicit count vector plotted against x-axis) applies in all cases except when one cumsum is plotted against another, as for example, in Figure 15.2F, where the cumsum of successive visit durations to one lever are plotted against the cumsum of successive visit durations to the other lever. To produce that cumulative record, the value of the variable ‘XY’ is 0.

Another possibility when two to-be-plotted data columns are passed in through usestats is a double-y plot where there are two y axes, the usual one on the left, and a separately scaled one on the right. If the two data columns contain not strictly ascending data, then the cumsum of the first is plotted against the left y axis and the cumsum of the second against the right y axis, with the corresponding count vectors plotted against the x axis. When both data sequences are strictly ascending, then they are plotted against a common x axis and the two corresponding count vectors are plotted against the left and right y axes, respectively.

When three or more data vector are passed in, cumulative records of each are plotted. Which axis is which is determined by the already specified principle: if the data in each column are not strictly ascending, then the cumsums are plotted against the y axis and the corresponding count vectors against the x axis; if they are strictly ascending, then the reverse is done. *In all cases, the to-be-plotted data columns must be either all strictly ascending or all not strictly ascending.*

The uses of the other varargin are the same as in the previous two plot functions. Notice that when double-y plots are called for, both the labels and the limits of the two y axes may be separately specified (by giving appropriate values for ‘LeftYlbl’, ‘RightYlbl’, ‘LeftYlm’ and ‘RightYlm’).

The panels are automatically labeled when the data come from any level below the Experiment level of the Experiment structure. When they come from the Subject level, each panel is labeled with ‘S#’ where # is the subject’s index number. When they come from the Session level, each panel is labeled with ‘S#,s#’, where the 2nd # is the index number of the session. When they come from the Trial level, each panel is labeled with ‘S#,s#,<TT>,T#’, where <TT> is the name of the trial type and the 3rd # is the trial number. When ‘Phase’ is included among the varargin, then the number identifying the phase (aka Group or Condition) is included in the labels, prefaced by ‘P’. When the data come from the Session level or the Trial level, the phase information is taken from the Phase field of the session from which the data come. When they come from the Subject level, then the Phase information is taken from the Phase field of the first of the currently active sessions.

**TSraster**

One of the most general and useful visualizations of frequently recurring events like action potentials or pecks or pokes is a raster plot. It plots point events as points and events with non-negligible durations as line segments on a 1 dimensional time axis. Since all time stamped data consists of times and events, this means that any TSdata can be plotted this way.

TSraster creates raster plots. Because it is most commonly used to make figures with only a single panel, it is the only plot function that is ordinarily called directly. Like any function, however, it CAN be called by TSapplystat as a helper function when one wants more than one raster plot on a single figure.

The syntax is:

[H,Ax] = TSraster(tsdata,trialdef,events,varargin)

It has 3 mandatory arguments: the *tsdata*, the *trialdef* to use to break up the TSdata into chunks to be plotted on successively higher horizontal lines, and the *events* to plot on each line. The *tsdata* is a 2-col array with time stamps in the first column and event codes in the second. The *trialdef* is any legal matchcode sequence that could be used with TSmatch or TSdefinetrialtype. The *events* argument is a 2 column matrix of event codes, in which each row of the matrix defines an event. An example of an event might be when a feeding occurs. In this case, there might be a code, Feed1, which indicates that feeder 1 dispensed a pellet at a certain time. This is a point event, an event negligible duration; therefore, it is plotted as a point in a raster plot. A duration event might be CS such as a noise or light that comes on and then goes off after some non-negligible duration. It is defined by 2 event codes, one denoting the onset of the event, the other the offset, for example, LightOn1 and LightOff1. Duration events, which are defined by an onset and offset event are plotted as horizontal line segments in a raster plot, indicating when (for example) the light was on and when it was not.

For duration events, the On code goes in the 1st column, and the Off code goes in the 2nd column of the 2-column *events* array. For point events, place the event code goes in the 1st column and a 0 in the 2nd column.

The optional additional arguments allow you to specify the colors, markers, (vertical) offsets, labels for the different point and duration events, and the axes into which the plot is to be made. When these arguments are not given, default values are chosen. These arguments can be given in any order and included or excluded in any combination. Unlike in the previous plot commands, these additional arguments are not Variable-Value pairs.

The colors-specifying argument is a string with the same number of characters as the number of events (that is, as the number of rows in the 2-col *events* array. Each character specifies the color or marker used for the corresponding event. The options for these characters can be viewed if you type help plot in the command window. To specify more than one code for some of the events, pass in a character array with the same number of rows as events, and then each row of characters will correspond to an event. Use the strvcat function to construct this character array; it fills in blanks where there are characters so that all rows of the array have the same number of “characters’, thereby honoring Matlab’s rectangularity constraint. For example, if you wanted the first event to be a red line, the second to be a blue square, and the third to be a green dotted line, the following command would create the desired character array: strvact(‘r’ ‘bs’ ‘g:’)

Vertical offsets, which forestall coincident events from being plotted on top of one another, are specified by a vector of integers the same length as the number of events. This vector allows you to offset the horizontal lines on which the events are plotted so that they are not all on top of each other. The integer does not correspond to an exact number of pixels; the integer offsets are all evenly spaced so that the distance from the largest offset number to the smallest is the same in every raster plot. This allows you to make however many offset lines you need. By default every event is placed on a separate line.

To add a legend, pass a cell array of strings with as many cells as events (that is, as rows in the event-specifying array), with the string in cell as the legend for the corresponding event. If this argument is included in the varargin, a legend will be added to the plot using these labels.

If the an axes handle is included among the varargin, TSrasterplot plots into those axes rather than creating a de novo figure. This is useful when you want to make more than one raster plot in a single figure. For example, suppose you want to make 4 raster plots on one figure arranged in two rows and two columns. You create the 4 sets of axes using subplot: Ax(1) = subplot(2,2,1); Ax(2) = subplot(2,2,2); Ax(3) = subplot(2,2,3); Ax(4)=subplot(2,2,4). Then, you create a raster plot in the first subplot by a call to TSraster that includes Ax(1) among the varargin, a raster plot in the second subplot by a call that includes Ax(2), and so on.

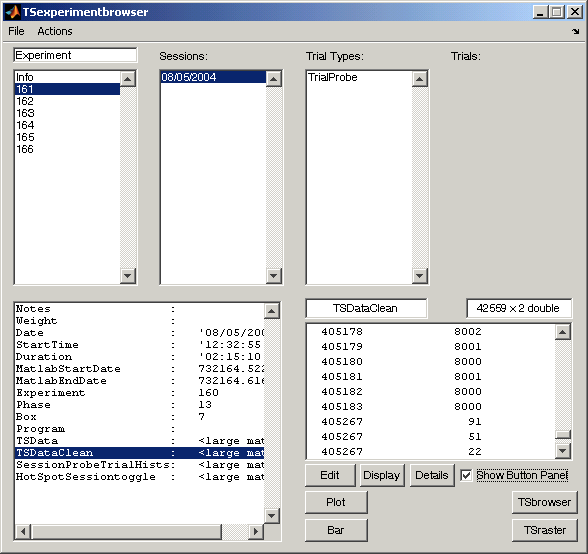
The optional output arguments, H & Ax, are handles to the figure and the axes that Matlab creates when it makes a de novo figure.

TSraster is a powerful tool, but it can be tedious to enter all the information into the varargin. The Raster Gui is a graphical tool that facilitates the creation of raster plots. It acts as a graphical interface to the TSraster function. When using the TSrastergui, you can select Trial definitions from a drop down menu, or enter your own, you can select event codes from drop down menus, or enter your own, you can select colors, markers, and linestyles from a drop down menu, and you can add labels. You can add these event definitions to the current raster group, delete them, or modify them. You can also maintain multiple raster groups at once, and you can load or save raster groups to .mat files for backups or sharing with others.

## Loading the Raster Gui

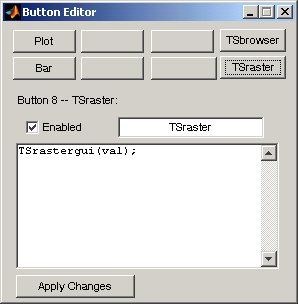
Syntax Tsrastergui(*tsdata*)

This command opens a GUI that takes tsdata as an initial argument and provides a graphical user interface for generating raster plots depicting events as they occurred during trials for any given session. The easiest way to call the Raster GUI is through the Experiment Browser. Start the Experiment Browser (TSexperimentbrowser) and select a TSData set. Then select the check box to display the button panel.

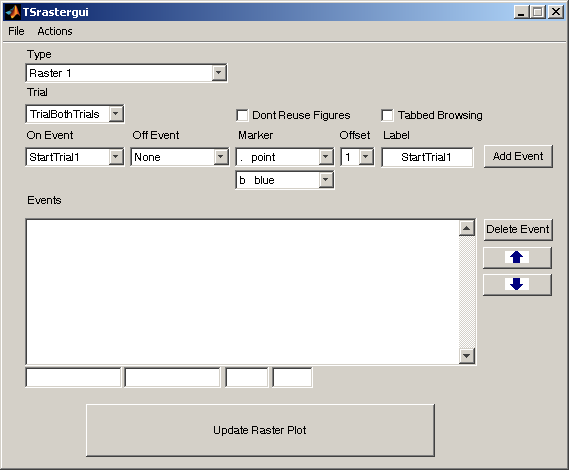


For further help and instructions refer to the Browser GUI documentation (Chapter 3).

With TS Data selected, click on the TSraster button to display the Raster GUI. If the TSraster button is not currently in your button panel you can add it by using the “Action” menu and selecting “Edit Buttons” (or by using the keyboard shortcut Ctrl-B).

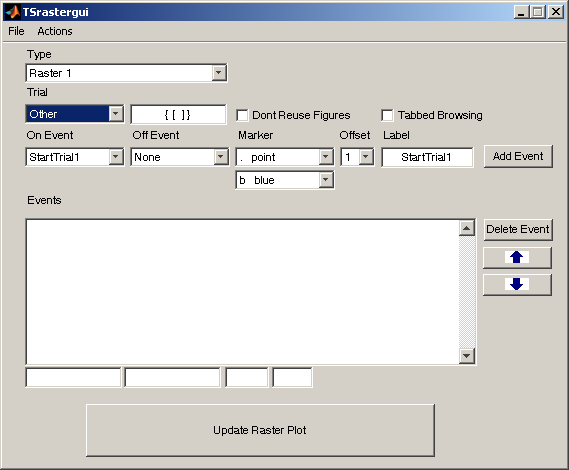
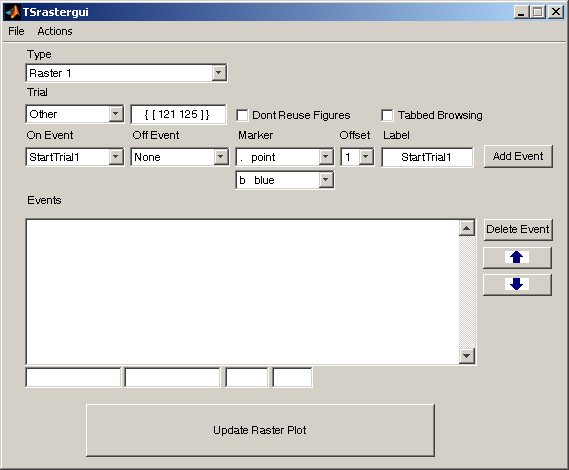


Using the button editor, create a new button with the value “Tsrastergui(val);” and click the “Apply Changes” button. This will ensure that the Raster GUI will be passed the selected TS Data when it is called. Then go to the Session data or other TSdata that you would like to raster, and press your TSrastergui button. When you have started the Raster GUI you see the following interface:



At the upper left is a drop down menu that indicates the raster you have selected. It should say “Raster 1” at the start, (unless someone else has been using TSrastergui before you. If they have, go to file and select “New Raster Settings”). This drop down menu keeps track of all of the raster settings you currently have loaded. If you have more than one, you can switch between them using this menu.

Directly beneath it, you will see another drop down menu, marked Trial. This list is populated with the names of trials that you have already defined using TSdefinetrialtype. You can select them here and you won’t have to retype them. If you want to make up a trial definition now to use, you can do that by selecting other, and typing the trial definition in the field that appears to the right.

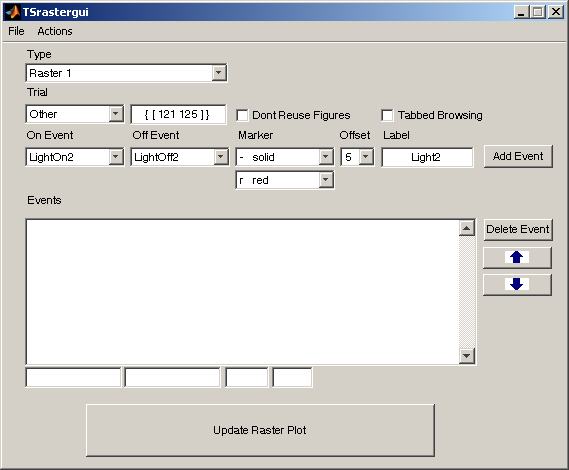
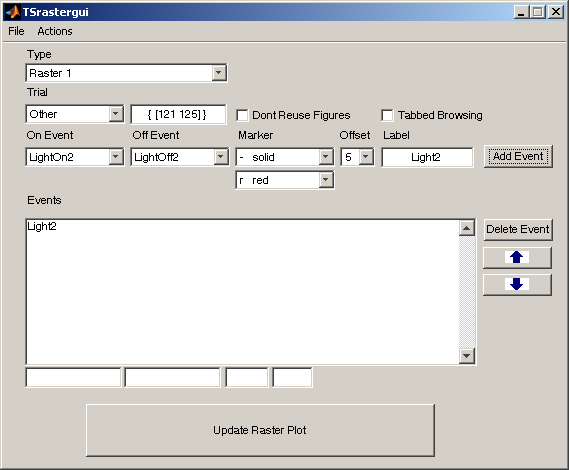
 

Now that you have a trial definition for your data, you need to start defining events. The rest of the drop down menus are all for that purpose. First you set them to the appropriate settings, then you select “add event” to add this new event to your list is of event definitions.

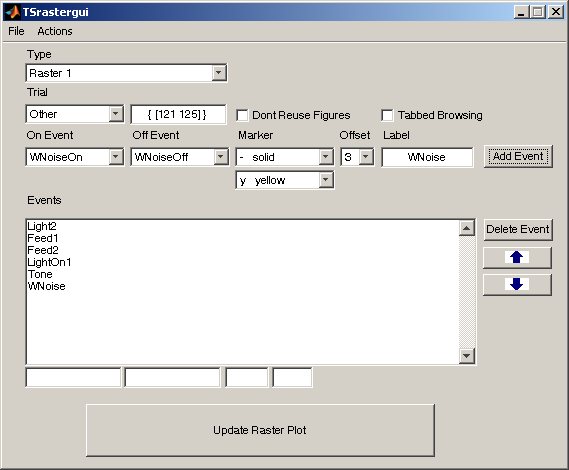
The first 2 drop down menus are marked On event and Off event. They are populated with your event code definitions for this Experiment. They also can be marked other, in which case a field will appear where you can enter the number code you wish to use. The Off event can be marked None, which will indicate a point event.

You can also select your colors and marker very easily. If you have selected a point event, then the marker drop downs will allow you to choose a color and a point marker. If you have selected a duration event, then you can select a color and a line style. Be warned, it may be unadvisable to use a dotted line or dashed line marker if the event you are plotting tends to go on and off frequently, as you will be unable to tell the dashes from when the event actually went on or off.

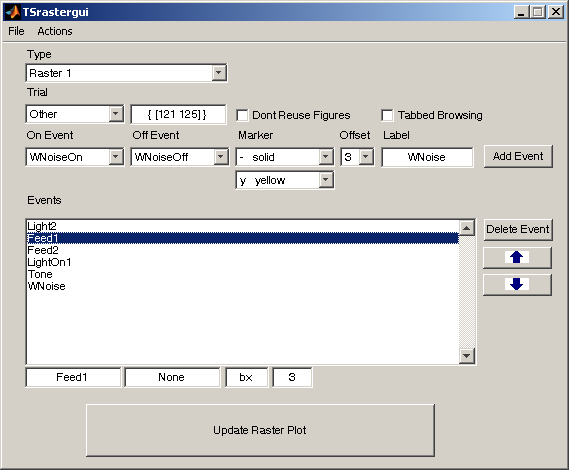
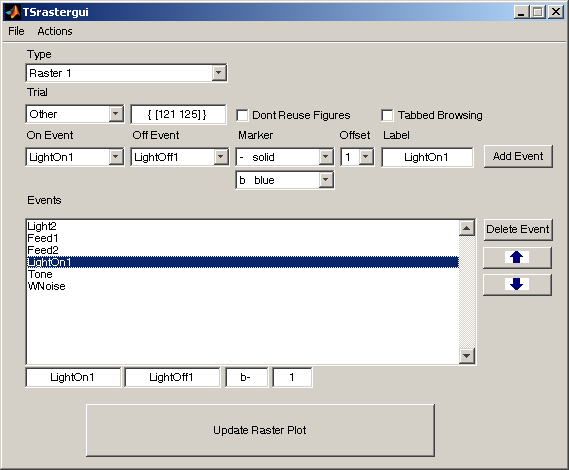
You can select the offset number from the offset dropdown. You can select a number between 1 and 10. Finally, you can select your label. It defaults to the name of the On event when you set it, but you can type whatever you like.

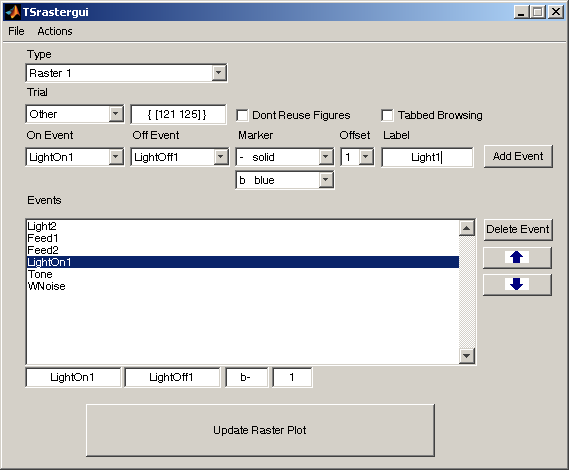
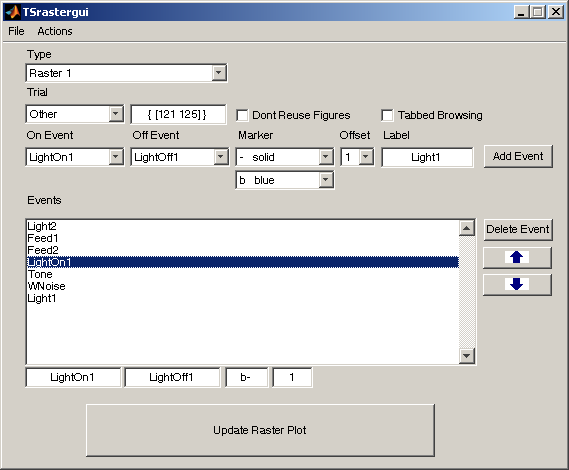
Now you can add your event. Click Add Event, and its label will appear in the “events” list. Add a few more events using different codes and markers.

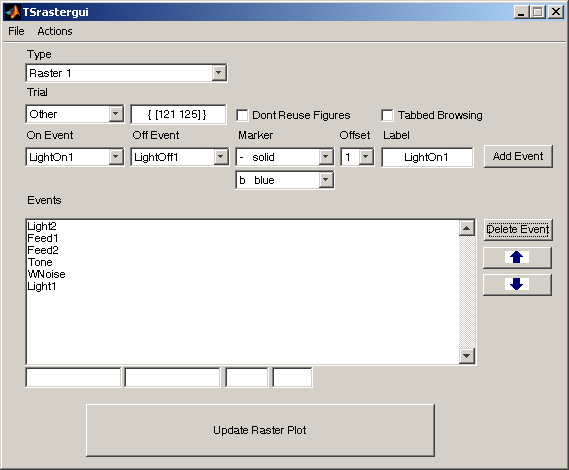
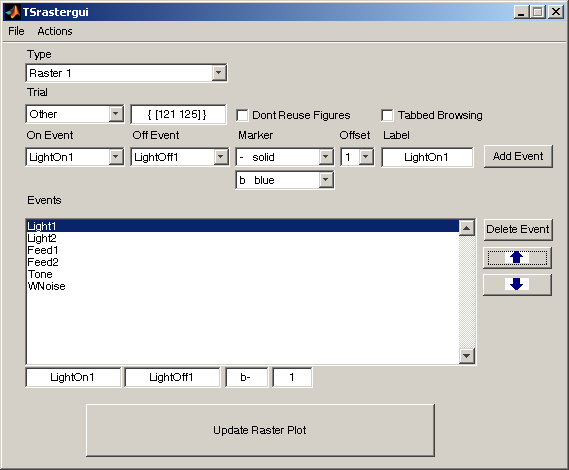


Now you have the makings of a good raster plot. To review your settings, you can select one of the events, and its configuration will appear in the text boxes below the events box. You can quickly browse through your current events this way.

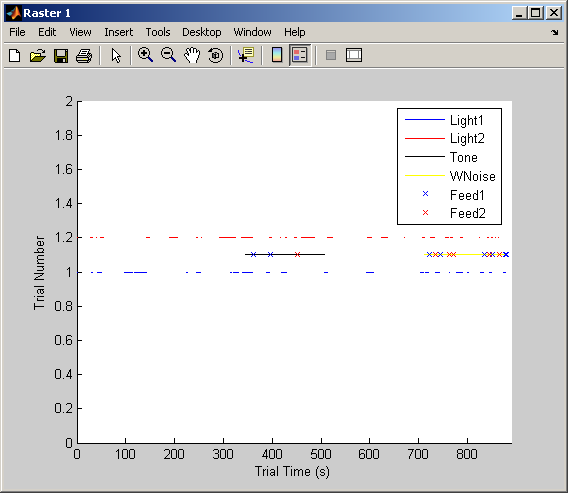
 

If you double click on an event, then the popup menus for On/Off events, Markers, Offsets, and the Labels textbox will all be reset to reflect that event’s current state. If you need to add several similar events, you can use this to quickly select the right settings. This is also very useful if you discover an error in one of your events. You can double click it, fix the error and add the changed version, and then select the flawed event and press delete. You can use the arrows to move around one or more of the events into order.

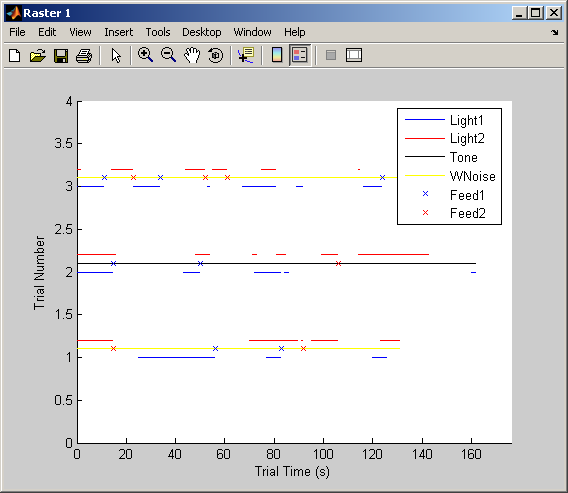
 

Now that your events are all correct, you are ready to make your plot. Press the large button at the bottom to make your plot appear.

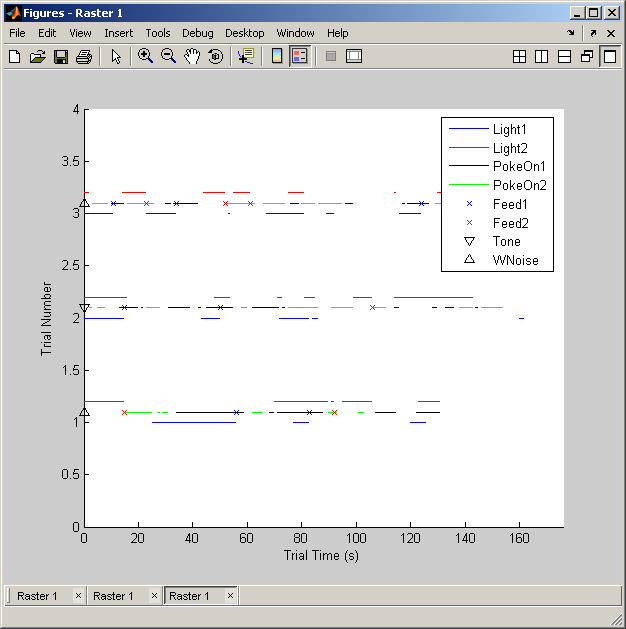


Notice the 2 checkboxes at the top, labeled “Don’t Reuse Figures” and “Tabbed Browsing”. These checkboxes change what happens when you press that button. The reuse figures option lets you select whether you want a new figure window made every time you press the button or if you want to reuse the windows for each type of raster. Reusing can be quite handy when you are setting up your events; you can press the update button every time you add an event to see if it turned out the way you thought it would, then you can quickly tweak it and update again to see what effect your changes had.

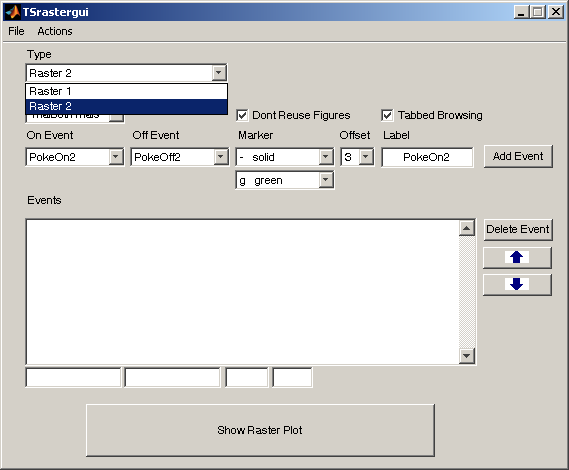
In this example, the Trial selection was changed to produce a different (and more interesting) breakdown of the session. With 2 clicks this change can be instantly updated to the same figure window we used above to reflect the change.



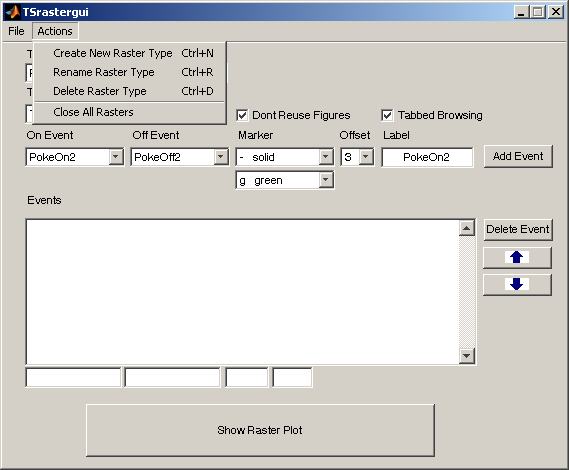
The tabbed browsing check starts the figure windows in “Docked” mode. Since you can quickly accumulate many figure windows when making raster plots, it can be handy to check this and keep them all organized.



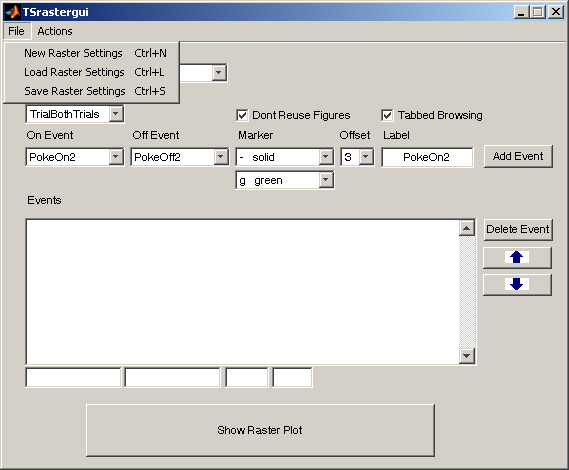
To make a second raster definition, go to the “Actions” menu, and select “Create New Raster Type” You will be asked to give the name of the new type, and then a blank event list will be shown. You can now switch back and forth between the 2 rasters using the upper left popup menu. Events will be added to the currently selected raster. You can make plots of either, and even if you have set it to reuse figure windows, raster types will not overwrite each other’s figures.



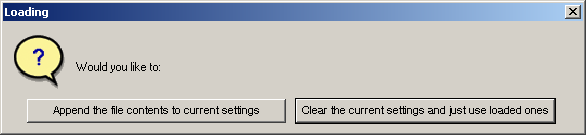
Also in the actions menu, you can rename or delete a raster type group. If you have accumulated too many figures, you can quickly delete all the raster figures using the “Close all rasters” button.



If you would like to save a collection of raster types, you can save it from the file menu.



You can also load a collection, and then you must select whether to append the current settings with the file contents or to overwrite the current settings and just use the file contents. So, if you just opened the raster gui and don’t have any important raster types made, you can just clear it. If you just made a new one and you also want 2 old ones you made, then you should append.



If you have raster plot settings that are particularly useful for interpreting your data, you should distribute these settings files with your Experiment.mat, and then other TSlib users can make the rasters and examine your data on their own.

# Chapter 16. Less Commonly Used Functions

## TSaddPrograms

In the spirit of our most important design principles—keep it all together!—we urge users to store their experiment-control program code in the Experiment structure along with all the data and all the results derived from the data. TSaddPrograms compares the program file you tell it to load to the program files it has already loaded. If the text in the to-be-loaded program file matches exactly the text of an already stored program, that program is not loaded again; the number of the program it matches is entered into the Program field of the Subject and Session. When the specified program code does not exactly match any stored program, another ‘Program(#) field is added to the Programs structure and the code file is read into its ‘Code’ subfield.

Syntax: TSaddPrograms(*filename,sub,ses*),  
  
where *filename* is the name of the file containing the program code (e.g., ‘InhibitonAcq.MPC’), *sub* is the subject’s index number, and *ses* is the index number of the session.

When the Experiment structure is created (see TSinitExperiment) an indexible Programs field is created on the Experimental level to contain program code. (In the Browser, it appears at the bottom of the pane immediately below “Experiment,” which contains the indexed Subject list. If there are many subject, you need to scroll down to see it.) The Programs field is itself a structure. It has indexed ‘Program’ subfields. Each ‘Program’ subfield has two fields subordinate to it:

Name: filename of the program  
 Code: Displays the MedPC code

TSinitexperiment defaults both these subfields to 'No Programs Loaded'

TSaddPrograms calls Matlab's readtextfile function, which will throw an error if the file being read contains any lines with only a line return. There are two remedies: 1) Open the program file in a word processor or text editor and do a find (^p) and replace (^p space); or 2) open Matlab's readtextfile command, which is editable, find all instances of the fgetl command and replace with fgets.

## TSwriteMedPC

Syntax TSwriteMedPC(*Filename*,*ProgramNum*)

Creates a file with name *Filename* and writes into it the MedPC program code stored in the Code field of the Programs structure indexed by *ProgramNum* . In other words, it exports stored MedPC experiment-control code.

## TSaddsubjects

Interactive function: Asks the user a series a questions and then adds one or more subjects to the Experiment . If some or all of the new subjects have ID numbers less than those of subjects already in the structure, those new subjects will appear before those old ones in the indexed list of subjects, **so those old subjects will now have new index numbers!**

Syntax TSaddsubjects [no arguments, because it’s interactive]

## TSsetoverwritemode

TSsetoverwritemode(true); % Turns on overwrite mode

TSsetoverwritemode(false); % Turns off overwrite mode

When loading in data or computing statistics, TSloadsessions may attempt to load in data it has already loaded, and other TS functions may attempt to compute statistics they have already computed. For example, you may have used TSsessionstat on sessions 1-10. If you now load in a session 11, calling the same TSsessionstat could cause that statistic to be re-computed for sessions 1-10. If you keep all of your data files in one folder, use of TSloadsessions could cause many files to be unnecessarily reloaded.

Re-computation or reloading of files won't cause any damage, but it will result in TSlib running much slower then need be. On the other hand, some times it may be necessary to re-compute statistics, as is the case if the statistic definition has been changed or when the new (additional) data from an ongoing session alter statistics already computed (e.g., means and variances).

TSlib maintains an overwrite flag that determines whether previously computed statistics should be re-computed and whether previously loaded files should be re-loaded. If the overwrite mode is true, then all TSlib functions will re-compute their values and files will be re-loaded by TSloadsessions. If this mode is false, then the will only load in data that has not been loaded before and they will only compute statistics that have not been computed before.

The overwrite mode is set to false as its default. To turn the overwrite mode on, use the command TSsetoverwritemode(true). To turn it back off, use the command TSsetoverwritemode(false).

## TSrmfield

This function deletes unwanted fields, often ones you have produced by mistake

Syntax TSrmfield*(Level,’field’)*

The first argument, *Level*, is the level where the field you want to delete is located in the Experiment structure. The four possible values for this argument are the strings: ‘Experiment’, ‘Subject’, ‘Session’, and ‘Trial,’. The second argument, ‘*field*’ is the name of the field you want to remove (enclosed in single quotes). Once you call this function, a text prompt will appear asking, “Do you want to remove the field *fieldname* from the *Level* level? [y/n]”. If you enter ‘n’, the function quits without doing anything; if you enter ‘y’, the function removes the field from the Experiment structure. When you delete a field from the Session level, it removes it from the Session level of every subject; when you delete a field from the Trial level, it removes it from the Trial level of every session of every single subject. **So be careful—and make a backup before you use this function! There is no restore!**

## TSremovesession

Removes a single session from the Experiment Structure

Syntax: TSremovesession (*sub,ses*),

where *sub* is a subject ID number from the Experiment.Subjects list (NB, the subject’s ID#, not its index #), *ses* is the index number of the session to be removed .This function removes individual sessions that were added erroneously. It removes the specified session from the sequence of indexed sessions for the specified subject, decrease that subject's NumSessions count, and recalculate the start and end dates for the Experiment. It will NOT remove the session file from the FilesLoaded record in Experiment.Info, so to reload this Session, you must turn on overwrite mode (TSsetoverwritemode(true)).

***TSrmeventcodes***

Syntax TSrmeventcodes(*Text* or # or *cellarray*)

Removes entries from the event-code dictionary in the Experiment structure. Entries to be removed may be specified either by name or number. To remove multiple entries at once, enter a cell array with one entry per cell, specified either as a string or a number. These codes are removed from the current Experiment and cleared from the global workspace as well. If the code is not found in the dictionary, it will not be cleared.

## TSsetdata

Syntax TSsetdata(‘newTSDataField’)

TSsetdata sets the field containing the active tsdata. The default setting is the TSdata field created at the Session level when the data from a session are loaded. However, the user may copy some portion of the data into another field and use this command to make that the active data field, so that subsequent TS commands like TStrialstat, TSsessionstat, operate only on the data in this field. Useful when there are different experimental conditions within the same session.

## TSedit\*

TSEdit allows the user to flexibly edit TSData by means of searching for matchcodes and either replacing these matches or adding new information around the match. For example :

TSedit(TSData , MatchcodeArrays , EditcodeArrays)

searches the TSData for instances of MatchcodeArrays and edits matched data based on instructions from EditcodeArrays. EditcodeArrays is a set of arrays of the form [Matchnum Matchpos Newcode Newlocationoffset]. Matchnum is the index of MatchcodeArrays in which edit codes should be executed if found. For instance, MatchcodeArrays may contain multiple matchcodes specified by the user, such as {[LightOn1, LightOff1] [LightOn1, Feed1, LightOff1]}. In this case, Matchnum of 1 will refer to [LightOn1. LightOff1], and 2 will refer to [LightOn1, Feed1, LightOff1]. This allows the user to flexibly specify code to be executed upon different matches. Matchpos refers to the index of the matchcode array specified by Matchnum that is to become the base of the offset of the desired data manipulation. In the above example of Matchnum 1, a Matchpos of 1 will center the offset on the time in which the LightOn1 is matched, and a Matchpos of 2 will do the same for LightOn2. NewCode is self-explanatory, as it refers to the event code that is to be inserted or replaced over the old event code. Finally, Newlocationoffset refers to the above-mentioned offset that is the time after, before, or during which the user’s data manipulation will take place. An offset of Inf will replace the code at that time. If a NewCode of 0 is given with an offset of Inf, it will delete the code at that time. Consider this simple TSData using matlabcodes and singular digit timestamps:

[ 1 00113

2 00041

3 00021

4 00031

5 00114 ]

The code :

TSEdit(TSData, {[LightOn1, LightOff1] [Feed1, EndSession]}, {[1, 2, LightOff2, Inf] [2, 1, PokeOn1, 1] [1 1 StartTrial, -1]})

will have the following effect on TSData:

[ 1 00113

1 00111

2 00041

3 00021

4 01011

4 00032

5 00114 ]

The matched instance of LightOff1 is changed to LightOff2, the instance of PokeOn1 is inserted into the time of Feed1 + 1 (Time 4), and the instance of StartTrial is inserted into the time of LightOn1 – 1 (Time 1).

## TSwaitbar

TSwaitbar('text'); % Turns on text reporting of progress

TSwaitbar ('graphics'); % Turns on graphical reporting of progress

TSwaitbar ('off'); % Turns off reporting of progress

Sometimes in can take a while to compute statistics in TSlib over all of your data and it can be useful to see how for it has to go. By default, TSlib reports its progress in the MATLAB command window. If you would like to see the progress with a graphical wait bar, use the command TSwaitbar('graphics'). To turn off reporting of progress, use the command TSwaitbar('off'). To go back to using the text reporting, use TSwaitbar('text').

Advanced Use:

TSwaitbar is an expansion upon Matlab’s built-in waitbar system. It acts as an interface to a stack of waitbars, which can provide progress updates on nested processes. The interface is very easy to use and to implement for your own custom functions so that they will provide waitbar updates as well.

The functions TSapplystat, TStrialstat, TSsessionstat, and TScombineover all use the TSwaitbar interface.

If you turn on graphical waitbars, and run an ApplyStat or one of the other functions just listed, you will see several waitbars appear in a figure window, each of which indicate progress at the Subject, Session, and Trial level.

If you write a custom function for use with ApplyStat / TrialStat / etc., and it iterates over a large for loop and takes a long time to execute, it might be desirable for it to create a waitbar as well. This can be done quite easily. Simply add the following commands at the appropriate places in your function with respect to the for loop:

TSwaitbar('add',0,'Example Process');

TSwaitbar('inc', .01);

for x = 1:100

...

...

TSwaitbar('update', x/100);

end

TSwaitbar('remove');

For more information, see help TSwaitbar.

## TSaddlog

Adds a dated daily log field to Experiment level so that user can keep dated notes

## TSdeclareeventcodes

Puts the event-code dictionary stored an Experiment structure into the workspace of the calling function.

## TSeventname

Finds the name (in the event-code dictionary given an event code number as input)

## TSexporteventcodes

Exports the event-code dictionary to a text file

## TSimporteventcodes

Imports event codes to the Experiment structure from an ascii textfile. This an be a file made with TSexporteventcodes or made in a text editor. Each line in the file must be of the form:  
 *text*=#,  
where *text* is the name of the event (used when writing code) and # is the numerical code for that event.  
This codeset will overwrite whatever codes were already in the Experiment! THERE MUST NOT BE ANY SPACES WITHIN THE EVENT CODES THEMSELVES!! (Spaces before and after '=' are no problem)

## TSfunctions

Prints to the screen a list of the TSfunctions with brief descriptions of what they do

# Section II. Partially & Fully Automated Analysis

## Chapter 17: Introduction to the Fully Automated System

In comparison to many other parts of modern biological science, the experimental study of animal behavior has an artisanal character: Everything is done by hand; the power of modern computing is only minimally used. Behavioral testing is an indispensable feature of the quest to reveal the neurobiological mechanisms of cognition and behavior. But, to bring its power fully to bear within contemporary neuroscience, it needs to become more automated, which will almost automatically make it more quantitative as well.

When properly done, quantitatively oriented behavioral testing places strong qualitative and quantitative constraints on the underlying mechanisms. This is most dramatically illustrated by the central role that psychophysical results have played in directing the experimental investigation of the neurobiological mechanisms of vision. In the domain of cognition, the most dramatic demonstration of the power and value of quantitative behavioral experiment to inspire and direct experimentation at the cellular and molecular levels of neurobiological analysis is the behavioral genetics approach to the cellular and molecular biology of the circadian clock, pioneered by Seymour Benzer. He and his students searched for drosophila with heritable variation in the behaviorally measured period of their clock. Prior to that work, what the mechanism of this clock might be was so obscure that more than a few leading neuroscientists believed there was not in reality a clock, despite a vast body of results from behavioral experiments implying that there was a clock. Many neuroscientists thought that talk of a clock was metaphorical at best. Now, we know a great deal about the molecular biological mechanism. Most of this rknowledge flows from the behavioral measurements of free-running (unentrained) clocks made by Benzer and his students and the many other experimentalists inspired by their success with drosophila.

A fundamental feature of a highly automated approach to behavioral testing is that the data be analyzed in quasi real time, as they are being generated. This is in contrast to the common practice of analyzing the data only months or even years after they have been acquired.

The Gallistel lab has developed a testing system that uses off-the shelf equipment from Med Associates. This equipment comes with a proprietary process-control programming language, MedPC™. MedAssociates mouse test chambers are connected to polypropylene mouse nest tubs by a short length of acrylic tube to create a test environment in which a mouse lives undisturbed throughout the testing. It nests in the tub and forages for food in the test chamber. The conditions under which it obtains food are controlled by MedPC process-control code running on PCs connected to the test environments by the Med Associates experiment control equipment.

The data generated by a mouse are written to a text file on the hard disk of the controlling PC every 10 minutes. The PCs controlling the test environments are form a local area network (LAN), together with an offsite server. TSlib code running in Matlab on the server harvests the data from the hard disks of the PCs every few hours, imports the data into an Experiment structure and analyzes and graphs it using the already described TSlib functions. The Experiment structure and the graphs live in Dropbox™ folders, which provides state-of-the art offsite data backup and makes the emerging results accessible to any invitee at any time from anywhere in the world.

The MedPC code may be written in such a way that a single file contains code for many different experimental protocols. Which of these protocols is operative at any given time is determined by the values in an internal MedPC array, to which Matlab has indirect access. Matlab code can alter the contents of the MedPC array by writing values to an external text file, which the MedPC program reads every 15 minutes. When the running MedPC program reads the external file it compares the values therein to the values currently in the internal MedPC array. If they disagree, then the new values, the values in the external text file, are loaded into MedPCs internal array, thereby changing the operative protocol. This arrangement enables Matlab to advance individual mice automatically from protocol to the next in a programmed sequence of protocols, when the behavior data from the mouse in a currently operative protocol show that it has satisfied user-specified performance criteria. The sequence of tests proceeds automatically on a mouse-by-mouse basis, with the results available to the investigators in graphic form every few hours.

In partially automated testing, the automatic data harvesting and analysis is implemented without the automated progression from protocol to protocol. The experimenter advances mice from one protocol to the next by stopping the current session when a mouse has met a performance criterion, then starting a new session with a new protocol in force.

### Functions Specific to Quasi-Real-Time, Automated Analysis

**TSbegin**  A graphical user interface (GUI) that creates an Experiment structure and fills it with basic information (Lab name, Subject IDs, strains, sexes, weights, Supplier(s), input and output time units, Paths to files, names of protocol-specific analysis functions, and input arguments for them). This information is read from an Excel spreadsheet template, where the user has entered it. See Chapter 18

**TSstartsession** A graphical user interface (GUI) that reads from the same Excel spreadsheet read by TSbegin the information needed to create the MedPC macro, which the user then calls to start a session. The required information is: i) the letter that identifies the hard disk of the control computer on the local area network (LAN); ii) the Box # for each subject; iii) the “Group” or “Phase” (i.e. experimental condition initially in force); and iv) the complete path to the MedPC code file, the compiled version of which will control the experimental environment and record the data (the time-stamped events). See Chapter 18

**TSaddprotocol** Adds a protocol to the sequence of automatically executed experimental protocols in the fully automated system. Takes from its input arguments or elicits from user via prompts a set of parameters that define the to-be-added protocol, the names of the fields within the Experiment structure where the data will be found on which the decision to terminate the protocol will be based, the Matlab decision code that implements the decision, and the decision criteria (one or more performance-threshold values that must be exceeded).

**TSchangeprotparams** Implements the advancement from one protocol to the next in the fully automated system.

**TSparameterarray** GUI that walks the user through the process of setting the protocol parameters

**TSdiagnostics** Scrutinizes the tsdata for evidence of faults in the MedPC code (for example, failures to turn a signal light off after turning it on, failures to turn the house lights off or on, and so on.

**TSmakeemptyfield** Early in training, the data that define a trial type may not yet have been generated by a given mouse, in which case there will be no trials of that type and, therefore, no data fields within trials of that type, hence also no session-level fields that get their data from those trial-level fields. This leads to crashes, particularly in calls to graphics functions, specifying non-existent fields. TSmakeempty field creates the fields whose existence is pre-supposed by those calls.

**Sequencer** This function is called at intervals of a few hours to once a day by a Matlab timer object (analysistimer) that lives in Matlab’s base workspace. The sequencer loads the ActiveExperiments file, which contains the ActiveExperiments cell array, which has one row for each active experiment. The Sequencer steps through this cell array row by row, calling DailyAnalysisFA to analyze the data of the experiment in the first row, then in the second row, and so on, until all the experiments have been analyzed.

**DailyAnalysisFA**  Called by the sequencer to perform the a quasi real time analysis of the data coming in from a currently running fully automated experiment. This function loads the harvested raw data into the Experiment structure does a basic analysis of it, focusing on variables like number of pellets fed and number of pellets observed to drop properly, which are crucial to animal health. Also calls protocol-specific functions that do analyses appropriate to specific protocols (e.g., the concurrent VI matching protocol, the 2-hopper autoshaping protocol, the switch protocol). Finally it calls TSchangeprotparameters, which decides whether a given mouse has met the criteria for advancement to the next protocol, and, if so, advances it by writing new protocol-control parameters to the external parameter file read by the running MedPC program

**CreateAnalysisTimer** Script that prompts user for the information needed to create a timer object that will call the Sequencer function at intervals specified by the user

**DataGrabberTimer** Script that prompts user for the information needed to create a timer object that will call NewDataGrabber at regular intervals. NewDataGrabber reads the MedPC backup files and copies them to the server and renames them so that TSloadsessions can load them into the Experiment structure when TSloadsessions is called by DailyAnalysisFA.

Chapter 18: TSbegin and TSstartsession

In the partially and fully automated procedures, the metadata for the experiment are entered into and Excel spreadsheet, which is then read by the TSbegin function in order to set up the Experiment structure and then by the TSstartsession function in order to create the MedPC Macro that the user calls in order to start the MedPC program for each mouse (each test apparatus). For this to work, the user must use the *StartExperimentTemplate.xlsx* spreadsheet (see Tables 18.1 and 18.2).

*Table 18.1 The Portion of StartExperimentTemplate.xlsx Read by TSbegin*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G |
| 1 | **Path To Parent Directory (e.g. /Users/galliste/Dropbox)** | | | | | | |
| 2 | C:\Users\hkourtev\Dropbox | | | | | | |
| 3 | **Experiment ID #** | | | | | | |
| 4 | 360 |  |  |  |  |  |  |
| 5 | **Full Name of Event Codes File (i.e., including path and extension) OR list of event names and codes** | | | | | | |
| 6 | C:\Users\hkourtev\Dropbox\CanonicalMatlabCode\eventcodes.txt | | | | | | |
| 7 | **Lab Name (e.g., Gallistel)** | | | | | | |
| 8 | Gallistel | | | | | | |
| 9 | **Path To Folder With Helper Functions** | | | | | | |
| 10 | C:\Users\hkourtev\Dropbox\canonicalmatlabcode\FAhelperfunctions | | | | | | |
| 11 | **Load Function** | | | |  |  |  |
| 12 | TSloadMEDPCFA | | | |  |  |  |
| 13 | **Data File Recognition Prefix Character** | | | | | | |
| 14 | ! |  |  |  |  |  |  |
| 15 | **Data File Extension** | | | |  |  |  |
| 16 |  |  |  |  |  |  |  |
| 17 | **Time Unit (in seconds) for Time Stamps in Raw Data File** | | | | | | |
| 18 | 0.02 |  |  |  |  |  |  |
| 19 | **Time Unit for Time Stamps in TSData** | | | |  |  |  |
| 20 | 1 |  |  |  |  |  |  |
| 21 | **Data Analysis Info (for ActiveExperiments)** | | | | |  |  |
| 22 |  | | | | |  |  |
| 23 |  |  |  |  |  |  |  |
| 24 | **Protocol Specific Analysis Functions** | | | |  |  |  |
| 25 | MatchingAnalysisFA | | AutoshapeAnalysis2HFA | | SwitchAnalysisFA | |  |
| 26 | **Input Arguments for those Functions** | | | | |  |  |
| 27 | PlotPath,Rows,Format,{[StartEarly StopEarly] [StartLate StopLate]} | | PlotPath,Rows,Format | | PlotPath,Rows | |  |
| 28 | **Phase(s) for which function to be called** | | | |  |  |  |
| 29 | 1 | 2 | 3 |  |  |  |  |
| 30 | **Figure format** | |  |  |  |  |  |
| 31 | .fig |  |  |  |  |  |  |
| 32 | **# rows of plots on each figure** | | |  |  |  |  |
| 33 | 4 |  |  |  |  |  |  |
| 34 | **Password & email addresses** | | |  |  |  |  |
| 35 | ServerEmailPassword | | galliste@ruccs.rutgers.edu | | keithanacker@verizon.net |  |  |
| 36 | **--------------------Subject Information---------------------------** | | | | | | |
| 37 | **ID** | **Species** | **Strain** | **Sex** | **Arrival Weigth** | **Arrival Date** | **Source** |
| 38 | 1397 | Mouse | CD-1 | M | 25-30 | 3/14/15 | Harlan |
| 39 | 1398 | Mouse | CD-1 | M | 25-30 | 3/14/15 | Harlan |
| 40 | 1399 | Mouse | CD-1 | M | 25-30 | 3/14/15 | Harlan |
| 41 | 1400 | Mouse | CD-1 | M | 25-30 | 3/14/15 | Harlan |
| 42 | 1401 | Mouse | CD-1 | M | 25-30 | 3/14/15 | Harlan |
| 43 | 1402 | Mouse | CD-1 | M | 25-30 | 3/14/15 | Harlan |
| 44 | 1403 | Mouse | CD-1 | M | 25-30 | 3/14/15 | Harlan |
| 45 | 1404 | Mouse | CD-1 | M | 25-30 | 3/14/15 | Harlan |
| 46 | etc | etc | etc | etc | etc | etc | etc |

*Note that the Excel row numbers and column letters are shown. TSbegin looks for a specific piece of information in a specific row and column(s). Therefore, the user must not alter where information is entered in this spreadsheet. For example, the ID# for the experiment must be in Row 4, Column A and the ID# of the first subject must appear in Row 38, Column A. There can, however, be arbitrarily many subjects listed in successive rows after this first subject.*

**TSbegin**

When you call TSbegin, it asks whether you have entered the required information into the Excel Template and saved the filled-out template under, say, a file name like “Experiment500Info.xlsx.” When you answer, ‘y’, it brings up a GUI that allows you to browse for that file. When you find and select the spreadsheet file, TSbegin reads the information from the spreadsheet and creates an Experiment structure for this experiment. It saves the Experiment structure inside a folder named ‘Experiment#’. Within that same folder, it creates subfolders named ‘MatlabCode,’ ‘DataArchive,’ ‘Plots,’ ‘DataTemp’, and ‘Transfer.’

The information needed for TSbegin to set up a fully automated experiment is supplied by entries in the template show in Table 18.1, to wit:

**Row 2, Column A**: The complete path to the “Parent Directory.” This is the ‘directory’ (aka folder) that will contain the folders for the individual experiments. The parent directory should be a Dropbox™ folder, that is, a folder in a cloud-synchronization system. The folder can, of course, be given any name one likes, but we refer to it hereafter as the Dropbox folder.

The TSlib folder should be one of the subfolders in the Dropbox folder. This is not essential, but it is generally desirable. *What is essential is that* *the TSlib folder be on Matlab’s search path!* The advantages of putting everything in a cloud-synchronizing folder are great. First, it provides state-of-the-art off-site backup. You will never loose your data, even if the laboratory burns down or is destroyed by an earthquake. Second, everyone in the lab can access the data and the graphic reports any time, anywhere in the world where there is internet access.

**Row 4, Column A:** The ID# for the experiment. Every experiment should have a different ID number. (The fully automated system can run more than one experiment at a time.) TS begin will create a folder for this experiment within the parent directory. The folder will be named ‘Experiment[#].’ TSbegin will put the Experiment structure for this experiment in that folder, and, as already noted, it will create several subfolders (to be explained).

**Row 6, Column A:** The complete path to the file that contains the event codes (see Chapter 7 for an explanation of event codes).  
 OR, if there are only a very few events, you may list the textual event code for the first event in Row 6, Column A, the corresponding event number in Row 6, Column B, the textual event code for the second event in Row 6, Column C, its corresponding event number in Row 6, Column D, and so on. TS begin reads these event codes into the eventcode “dictionary” in the Experiment structure that it creates.

**Row 8, Column A:** The name of the lab in which the experiment is to be run. This looks toward the day when the raw data and intermediate results from experiments are routinely placed in publically accessible data bases. The name of the lab together with the ID number for the experiment will uniquely identify the experiment.

**Row 10, Column A:** Complete path to a folder containing the routinely used “helper functions.” A helper function is a function that is called by one of the functions in TSlib to help the TSlib function carry out some, usually simple, computation on some data. However, a helper function can be itself arbitrarily complex, and it may itself call helper functions (helpers to the helper). The protocol-specific analysis functions, such as MatchingAnalysisFA, are complex and lengthy helper functions, as are most of the graphics functions. For more on helper functions, see Chapter 12 (TSapplystat). Users will tend to build up a library of custom helper functions, which they employ from one experiment to the next. TSbegin copies the folder of helper functions into a subfolder within the folder for this experiment. The subfolder containing the helper functions is named “MatlabCode.” The DailyAnalysisFA function puts this folder and the other subfolders on Matlab’s search path when it is called to analyze the data from a given experiment, and it takes it off Matlab’s search path when it finishes its analysis of that experiment. Thus, if one has modified a helper function used in an earlier experiment, but not changed its name, the version of this function called when analyzing the data from the new experiment will be the modified version of that function in the MatlabCode folder within the folder for the new experiment. It will not be the function with the same name lurking elsewhere.

**Row 12, Column A:** The complete path to the load function. The load function is a helper function for TSstartsession. It understands the structure of *your* raw data files. It extracts from them the header information that TSloadsessions requires—see Chapter 5.

**Row 14, Column A:** The initial character that distinguishes the name of a raw data file from the name of any other kind of file—see Chapter 5. If using an extension to distinguish raw data files instead of a prefatory character, leave this cell blank.

**Row 16, Column A:** In addition to, or instead of, distinguishing raw data files from other files by means of a prefatory character, one may distinguish them by an extension—see Chapter 5. If they are so distinguished, enter that extension in this cell; otherwise leave this cell blank.

**Row 18, Column A:** Enter here the time unit for your raw data in seconds—see Chapter 5 for further explanation.

**Row 20, Column A:** Enter here the time unit for the TSData field within the Experiment structure—see Chapter 5 for further explanation.

**Row 25, several columns:** In the fully automated procedure, each mouse is advanced from protocol to protocol automatically, as its performance satisfies the performance criteri(on/a) for the current protocol. For each protocol, there may be data analyses that only make sense for that protocol and that should, therefore, only be applied to the data generated by that protocol. In this row, the user supplies the names of those functions protocol-specific data-analysis functions.

**Row 27, several columns:** Protocol-specific analysis functions may have input arguments specific to a given such function. In this row, the user gives those input arguments *exactly as they would be given in a call to the function* (see example in Table 18.1). TSbegin creates a cell array called ActiveExperiments, which contains the names of the protocol-specific analysis functions and also their input arguments. DailyAnalysisFA uses this information to construct calls to those functions (see Figure 18.1)

**Row 29, several columns:**  The different protocols define different experimental “phases” or “conditions,” through which a subject progresses. In the Experiment structure, at the Session level, there is for each subject a Phase field. The number in this field specifies the current condition for that subject. That, in turn, determines which protocol-specific analysis function is applied to the most recent data from that subject. In these columns, the user specifies which phase numbers correspond to which protocol-specific analysis functions. In the example shown in Table 18.1, Phase 1 designates the matching protocol, Phase 2 the 2-hole autoshaping protocol, and Phase 3 the switch protocol. *The phase # for a given protocol must be entered in the same column as is the corresponding analysis function in Row 27*.

**Row 31, Column A:** Both DailyAnalysisFA and the protocol-specific analysis functions generate graphs. The graphs are saved into dated subfolders within the “Plots” subfolder that TSbegin creates in the folder for the new experiment. Here, the user specifies the format in which those figures should be saved. ‘.fig’ and ‘.pdf’ are the most common formats, but Matlab allows many different formats.

**Row 33, Column A:**  Many of the figures generated by the data-analysis functions have multiple panels on each figure, usually one panel per subject. The panels are arranged in 2 columns. Here, the user specifies how many rows of panels are to be put on each figure. Values between 2 and 8 are appropriate. When they exist, values that divide the number of subjects an integer number of times are preferred.

**Row 35, several columns:** DailyAnalysisFA sends emails with user-specified data graphs and text file alerts to data patterns, such as too few pellets fed, which indicate a possible problem requiring rapid attention. The the password for the account by which the server will send the emails and the email addresses of the recipients are specified in successive columns of this row.

**Rows 38 and after, Columns A-G:** These contain subject-specific information, as per the self-explanatory column headings (ID, Species, Strain, Sex, Arrival Weight, Arrival Date, Source/Supplier)

Fig 18.1 ActiveExperiments.pdf

**Figure 18.1** *The ActiveExperiments cell array. It has one row per active experiment. Sequencer loads this cell array and analyses the active experiments using the information in the 5 cells in a given row. The first cell in a row gives the ID number of the experiment to be analyzed; the second cell gives the format in which the figures are to be saved; the 3rd cell specifies how many rows of panels are desired in those figures with multiple panels (there are always 2 columns of panels); the 4th cell contains a cell array with 3 columns and n1 rows, where n1 is different protocols in the sequence of protocols to be run. The 1st cell in one of these rows gives the number that identifies a protoco1 (aka ‘phase’, ‘condition’ or ‘group’); the 2nd cell gives the name of the function that analyzes the data from that protocol; the 3rd cell gives the arguments that are to be passed in to that function. The 5th cell in a row of the Active Experiments cell array is a 1 x n2 cell array, where n2 is the number of different email addresses +1. The first cell of this cell array contains the password required to send an email from the server. The remaining cells are the email addresses. The ActiveExperiments cell array is created by TSbegin if and when the user answers ‘y’ to the prompt whether this is a fully automated experiment.*

**TSstartsession**

Sessions in MedPC are most conveniently started using MedPC macros. A MedPC macro is a simple text file. Each line gives the information required to start one subject, for example:  
LOAD BOX 1 SUBJ 1367 EXPT 358 GROUP 1 PROGRAM AUTOSHAPEACQ

The function TSstartsession gets this information from the second portion of the Excel spreadsheet, the portion that gives the information necessary to start a session Table 18.2. It creates the requisite macros, writes them to the appropriate hard drives, and enters this same information plus additional information into appropriate places in the Experiment structure.

*Table 18.2 The Portion of the Template Giving the Information Needed to Start a Session*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | H | I | J | K | L | M |
| Rows | ID | HD | Box# | Group# | .MPC | Protocol# | Weight |
| 38 | 1397 | Q | 1 | 1 | Q:\MED-PC IV\MPC\MatchingAutoShapeSwitch.MPC | 1 | 29.3 |
| 39 | 1398 | Q | 2 | 2 | Q:\MED-PC IV\MPC\MatchingAutoShapeSwitch.MPC | 1 | 31.5 |
| 40 | 1399 | Q | 3 | 1 | Q:\MED-PC IV\MPC\MatchingAutoShapeSwitch.MPC | 1 | 32 |
| 41 | 1400 | Q | 4 | 1 | Q:\MED-PC IV\MPC\MatchingAutoShapeSwitch.MPC | 1 | 33.2 |
| 42 | 1401 | Q | 5 | 1 | Q:\MED-PC IV\MPC\MatchingAutoShapeSwitch.MPC | 1 | 30.9 |
| 43 | 1402 | Q | 6 | 2 | Q:\MED-PC IV\MPC\MatchingAutoShapeSwitch.MPC | 1 | 31.5 |
| 44 | 1403 | Q | 7 | 2 | Q:\MED-PC IV\MPC\MatchingAutoShapeSwitch.MPC | 1 | 32.4 |
| 45 | 1404 | Q | 8 | 2 | Q:\MED-PC IV\MPC\MatchingAutoShapeSwitch.MPC | 1 | 35.3 |
| 46 | 1405 | K | 1 | 1 | K:\MED-PC IV\MPC\MatchingAutoShapeSwitch.MPC | 1 | 29.9 |

# Raw Session Data File Formats

If you have data in a custom format not supported by our functions, then you need to write a custom load routine. This is not difficult, but there are certain guarantees your code must make in order for it to be compatible with our code. The TSlibrary functions expect certain information about Sessions, all of which must be passed back by your function. After getting all this information, our session loading routines will place it into the appropriate places in the Experiment structure.

There are a large number of fields in Session designed to accommodate whatever peripheral data may be stored in your Session files. Much of it is not required, and can be set to the empty array [] without any problems.

The job of your function is to take a filename and get as much of this information as is available out of the file.

Your function must have the following syntax. It takes one argument, the filename to be loaded. It returns:

[ SUCCESS , ExperimentID, SubjectID, Phase, Box, MATLABStartDate, Duration, TSdata, Notes, Weight, Program, FileReportedUnits ]

All fields are required to be returned by your function.

All fields should default to [] if they are not found in the file, or not supported in your custom data format.

Success - This is a flag which should be true if the load was successful and false if it was a failure.

ExperimentID - This is the identification number of the Experiment. This is not a stringent requirement for your function to meet. However, if you do not provide it, you will recieve a warning, and we will assume that it belongs to the current Experiment. If you provide the number but it does not match the current Experiment, you will receive a warning. We generally assume that you wont load the wrong Experiment's data.

SubjectID - This is the identification number of the Subject this data is for. This is a required field, because if we do not know what subject it is we dont know where to put the Session. If it is empty, it is an error. If the number returned is not listed in this Experiment's list of subjects, that is also an error.

Phase - This is the phase number that this Session represents. This is not stringently required, if the empty matrix is returned you will receive a warning and it will be set to 1.

Box - This is the number of the box in which the session was recorded. This is not stringently required, if the empty matrix is returned you will recieve a warning and it will be set to 1.

MATLABStartDate - This is the date and time at which the Session started. This is a stringent requirement, if it is missing you will recieve an error. This should be a matlab date number, e.g., something returned from the matlab function "datenum". Datenum will accept a vector containing [Y, M, D, H, M, S] (year month date hours minutes seconds), or a string representing a date. See help datenum for more information about this.

Duration - This is a totally optional field representing the duration of the trial, also as a matlab date number. If you do not record this in your data file, set it to [], and we will determine it by using the last time stamp in the TSdata and converting that time into seconds.

TSdata - This field should contain the actual raw data of your Experiment, in 2 column time stamped data format. Most of our functions assume this data format, but if you really have a need for it, you could use a different format and most things would continue to work just fine, especially if you still have a double matrix with time-stamps in the left column. In reality, no checks are made on this field, so you can put your data in whatever form is convenient for you. If you do this, be sure to provide a duration, or else there will be an error.

Notes - This totally optional field should hold a character array or cell string containing notes on this session. This is not critical in any way to the functioning of the TS library, and if you do not store it in your data file, you should return an empty character array (''), and it is easy to add your notes to the Sessions using the TSexperimentbrowser.

Weight - This, like notes, is a totally optional field, which should hold a number representing the weight of the subject at the time of the session. It is easy to add it later, return an [] if you do not want to add it.

Program - This is another optional field meant to keep a record of the program used to collect this data. It should hold a string or cell string. Return '' if you do not want to add it.

FileReportedUnits - This field is used if the datafile has a record of what units its timestamps are in. It is analogous to the InputTimeUnit field in the TSsetloadparameters function. If these 2 values do not match, a warning will be given, and the file will override the load parameter setting.

Your function will not need to access the Experiment structure at all, nor will it need to access or apply the loading parameters. You should NOT perform any unit conversions in this function, TSloadsessions will take care of that.

You should throw warnings and errors if the data file is flawed.

In general you can craft this function around the custom format you choose to use. MATLAB has numerous routines for reading different formats, and is capable of reading just about anything. As long as you can get the Subject ID, Date, and TSData at a bare minimum, you can pretty much ignore the other fields and either hardcode them as empty or [1]. In fact, it is possible to encode the Subject ID, Date, and even other fields into the filename, and have the file just contain ascii, or binary, TSdata.

That said, these other fields are here for later convenience, and we encourage you to use a format that has enough information so that years later you will be able to figure out what the file was and what session it represents. If you provide them, then they will be readily accessible from the Experiment Browser window, and your records of the Experiment will be that much more complete.

## Standard Format

We have developed a standard session file format which TSlib has built in support for. The assumption is that the datafile forms a 2 column matrix of integer or floating point values.

At the start, there is a header section containing peripheral information in the first column, with identifier flags in the second column attached to each number. Then there is a row of two 0’s, which forms a separator between the header and the actual TSdata, which continues until the end of the file.

Example:

<Month>, 1

<Date>, 2

<Year>, 3

<Hours>, 4

<Minutes>, 5

<Seconds>, 6

<Experiment>, 7

<Subject>, 8

<Phase>, 9

<Box>, 10

<TimeUnit>, 11

<Weight>, 12

0, 0

... (tsdata)...

... ...

... ...

In this example, <…> should be replaced by their numeric values.

Note that the rows in the header can be in any order (except for the seperator obviously) because they are identified by the integers 1-12 in the right hand column. They can even be absent, and functions that support this format will set them to [].

TSlib has 3 functions that support this format: TSloadstdtab, TSloadstdxls, and TSloadstdcsv

Standard Tab = Standard format above, using Tab delimiting for columns and end-lines for rows

Standard CSV = Standard format above, with Comma delimiting for columns, end-lines for rows

Standard XLS = Standard format above, in an Excel spreadsheet file.

The functions themselves are quite simple: Here is the source for TSloadstdxls, as an example for the reader.

function [ SUCCESS , ExperimentID, SubjectID, Phase, Box, MATLABStartDate, Duration, TSdata, Notes, Weight, Program, FileReportedUnits ] = TSloadstdxls ( filename )

%

% Assumptions:

% File is tab delimited

% Dlm file forms a 2 column matrix

% First 12 rows are various session fields with indicator numbers on the

% right column which tell the program which field it is. These do not

% have to be in any specific order. At the end, a row of 0,0 seperates

% the tsdata from the codes.

% Month, 1

% Date, 2

% Year, 3

% Hours, 4

% Minutes, 5

% Seconds, 6

% Experiment, 7

% Subject, 8

% Phase, 9

% Box, 10

% TimeUnit, 11

% Weight, 12

% 0, 0

% ... (tsdata)...

% ... ...

% ... ...

%

% Only Subject and date are really required. Other things are not

% strictly required but are good to have. TimeUnit is strictly optional, and

% represents the unit the time stamps are measured in, using

% seconds e.g. if your unit is 50ths of a seconds, this field should be

% .02. If this field is not provided the load parameters will be used. If

% this field does not match the load parameters setting, a warning will

% be passed and the data file will override the load parameters setting.

%

% Weight is strictly optional and if your apparatus does not take note of

% this, you are encouraged to use the TSexperimentbrowser to enter this

% data at the end of the Experiment.

SUCCESS = 0;

raw = xlsread(filename);

seperator = find(raw(:,2) == 0, 1, 'first');

%Default values for everything

month = [];

date = [];

year = [];

hours = [];

minutes = [];

seconds = [];

ExperimentID = [];

SubjectID = [];

Phase = [];

Box = [];

FileReportedUnit = [];

Notes = '';

Weight = [];

Program = '';

FileReportedUnits = [];

Duration = [];

TSdata = [];

%Parse the header

x = 1;

while x < seperator

switch raw(x,2)

case 1

month = raw(x,1);

case 2

date = raw(x,1);

case 3

year = raw(x,1);

case 4

hours = raw(x,1);

case 5

minutes = raw(x,1);

case 6

seconds = raw(x,1);

case 7

ExperimentID = raw(x,1);

case 8

SubjectID = raw(x,1);

case 9

Phase = raw(x,1);

case 10

Box = raw(x,1);

case 11

FileReportedUnit = raw(x,1);

case 12

Weight = raw(x,1);

end

x = x + 1;

end

%Throw warnings if date parts are missing

if isempty(month)

warning('TSload:NoMonth', 'This sesssion data file %s did not provide the month of the session.', filename);

end

if isempty(date)

warning('TSload:NoDate', 'This sesssion data file %s did not provide the date of the session.', filename);

end

if isempty(year)

warning('TSload:NoYear', 'This sesssion data file %s did not provide the year of the session.', filename);

end

if isempty(hours)

warning('TSload:NoHour', 'This sesssion data file %s did not provide the hour of the session.', filename);

hours = 12;

end

if isempty(minutes)

warning('TSload:NoMinute', 'This sesssion data file %s did not provide the minutes of the session.', filename);

minutes = 0;

end

if isempty(seconds)

seconds = 0;

end

MATLABStartDate = datenum(year, month, date, hours, minutes, seconds); %Get matlabstartdate from time variables

TSdata = raw(seperator+1:end, :); %TSdata is everything after the seperator

TSdata = sortrows(TSdata, 1); %make sure it sorted by timestamp

TSdata(all((TSdata == circshift(TSdata, 1))'),:) = []; %eliminate repeats

SUCCESS = 1;

## MEDPC format

We have developed a custom MedPC format which is suitable for storing large quantities of time stamped data. MedPC does not support a 2 column output, so we encode the timestamps and event codes into one integer, by multiplying the times by 10^5. Since event codes range from 11-99999, there cannot be a collision here. MedPC has their own header syntax so this function parses theirs.

It is recommended, if you want to use this format, that you use MedPC format 3 “stripped, C array only” variant apprended all, and store the encoded TSdata in the ‘C’ array so that it will be the only thing besides the headers stored to the file.

TSloadMEDPC can then load this session data correctly. It parses the headers assuming that row 1 contains Month, row 2 contains Date, row 3 contains Year, etc. as specified by the MedPC standard.

Then, it removes all lines less than 10^5 and keeps the remainder as the TSdata lines. This will remove the header, which actually is reinserted periodically by MEDPC and needs to be detected and removed unless you take care of it this way. The column vector of encoded information is then expanded into full 2 column TSdata, duplicates codes are checked for and removed, and then this is returned.

You should see the MedPC Users Manual for more information about how to do this, or see (ref).

Notes on setting up a LAN with a remote server. I decided to make this a separate Word document: SettingUpTheLAN.docx

The Pascal code called by BKGRND [Must change this because MedPC Support redid it to solve our startup problem ]

Library BackProc;

{$C+}

Uses MedTypes, User, SysUtils;

Type

Real = Extended;

Var

SomeUserData: UserDataType;

BackProcGlobal: Real;

{$F+} {BE SURE NOT TO REMOVE THE FAR DIRECTIVE TO THE LEFT!!}

{THIS PROC DEMONSTRATES WRITING TO A DATA FILE IN THE BACKGROUND}

Procedure BackProc1(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

St: String[50];

OutF: Text;

I: Word;

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

St := LowerCase(SubjectSt) + '.txt';

While St[1] = ' ' Do

Delete(St, 1, 1);

AssignFile(OutF, '.\Data\' + St);

Rewrite(OutF);

{

Vars['C'].Size EXEMPLIFIES HOW TO DETERMINE THE NUMBER OF ELEMENTS

ASSOCIATED WITH ANY ARRAY OR VARIABLE. THE FIRST ELEMENT OF ARRAYS

OR VARIABLES IS REFERENCED AS 0 AND THE SIZE FIELD GIVES THE HIGHEST

SUBSCRIPT. THE FOLLOWING LOOP WRITES ALL ELEMENTS OF ARRAY C TO THE

OUTPUT FILE.

}

For I := 0 To Vars['C'].Size Do

Writeln(OutF, BigArr[Vars['C'].StartOffset+I]:9:2);

Close(OutF);

{

THE FOLLOWING LINE SET VARIABLE D SO THAT THE CALLING BOX CAN KNOW THAT THE

JOB IS DONE. ANY VARIABLE OR ARRAY ELEMENT COULD BE SUBSTITUTED.

}

BigArr[Vars['D'].StartOffset] := 1; {SYNTAX FOR ADDRESSING A SIMPLE VARIABLE}

End; {With}

I := IORESULT; {REMOVES ANY ERROR CONDITIONS -- IMPORTANT IF YOU'RE DOING FILE I/O}

BackRequest[Box][1] := False; {THIS MUST ALWAYS BE DONE TO LET MED-PC KNOW THAT JOB IS DONE.}

End; {With}

End; {BackProc1}

{$F-}

{$F+}{$I-} {THE $I- DIRECTIVE PREVENTS THE PROGRAM FROM CRASHING IF THE FILE DOESN'T EXIST}

{THIS PROC DEMONSTRATES READING FROM A DATA FILE IN THE BACKGROUND}

Procedure BackProc2(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

InF: Text;

InF1: Text;

I: Word;

D: Word;

St: String[50];

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

Str(Box, St);

AssignFile(InF, '.\Data\Box' + St + 'CurrentParameters.txt');

Reset(InF);

AssignFile(InF1, '.\Data\Box' + St + 'OldParameters.txt');

Reset(InF1);

D := 0;

If (IORESULT = 0) Then

Begin

{

Vars['C'].Size EXEMPLIFIES HOW TO DETERMINE THE NUMBER OF ELEMENTS

ASSOCIATED WITH ANY ARRAY OR VARIABLE. THE FIRST ELEMENT OF ARRAYS

OR VARIABLES IS REFERENCED AS 0 AND THE SIZE FIELD GIVES THE HIGHEST

SUBSCRIPT. THE FOLLOWING LOOP READS ALL ELEMENTS OF ARRAY C IN FROM

THE INPUT FILE.

}

For I := 0 To Vars['O'].Size Do

Begin

Readln(InF, BigArr[Vars['O'].StartOffset+I]);

End;

Close(InF);

For I := 0 To Vars['X'].Size Do

Begin

Readln(InF1, BigArr[Vars['X'].StartOffset+I]);

End;

Close(InF1);

For I := 0 To Vars['O'].Size Do

Begin

If (BigArr[Vars['O'].StartOffset+I] <> BigArr[Vars['X'].StartOffset+I]) Then

Begin

D := 1;

End;

End;

If (D = 0) Then

Begin

BigArr[Vars['J'].StartOffset] := 1; {SYNTAX FOR ADDRESSING A SIMPLE VARIABLE}

End;

If (D = 1) Then

Begin

AssignFile(InF1, '.\Data\Box' + St + 'OldParameters.txt');

Rewrite(InF1);

BigArr[Vars['J'].StartOffset] := 2; {SYNTAX FOR ADDRESSING A SIMPLE VARIABLE}

For I := 0 To Vars['O'].Size Do

Begin

Writeln(InF1, BigArr[Vars['O'].StartOffset+I]);

End;

Close(InF1);

End;

End; {If}

{

THE FOLLOWING LINE SET VARIABLE D SO THAT THE CALLING BOX CAN KNOW THAT THE

JOB IS DONE. ANY VARIABLE OR ARRAY ELEMENT COULD BE SUBSTITUTED.

}

End; {With}

I := IORESULT; {REMOVES ANY ERROR CONDITIONS -- IMPORTANT IF YOU'RE DOING FILE I/O}

BackRequest[Box][2] := False; {THIS MUST ALWAYS BE DONE TO LET MED-PC KNOW THAT JOB IS DONE.}

End; {With}

End; {BackProc2}

{$F-}{$I+}

{$F+}

{THE BACKPROC THAT FOLLOWS DEMONSTRATES WRITING A MACRO FROM WITHIN

AN MSN PGM. THE MACRO MAY BE PLAYED BACK FROM WITHIN MED-PC.}

Procedure BackProc3(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

BigArr[Vars['A'].StartOffset] := BackProcGlobal;

End; {With}

BackRequest[Box][3] := False;

End; {With}

End; {BackProc3}

{$F-}

{$F+}

Procedure BackProc4(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

End; {With}

BackRequest[Box][4] := False;

End; {With}

End; {BackProc4}

{$F-}

{$F+}

Procedure BackProc5(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

End; {With}

BackRequest[Box][5] := False;

End; {With}

End; {BackProc5}

{$F-}

{$F+}

Procedure BackProc6(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

End; {With}

BackRequest[Box][6] := False;

End; {With}

End; {BackProc6}

{$F-}

{$F+}

Procedure BackProc7(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

End; {With}

BackRequest[Box][7] := False;

End; {With}

End; {BackProc7}

{$F-}

{$F+}

Procedure BackProc8(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

End; {With}

BackRequest[Box][8] := False;

End; {With}

End; {BackProc8}

{$F-}

{$F+}

Procedure BackProc9(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

End; {With}

BackRequest[Box][9] := False;

End; {With}

End; {BackProc9}

{$F-}

{$F+}

Procedure BackProc10(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var

Data: ^DataRec;

Begin

With MPCGlobal^ Do

Begin

Data := BoxPointer[Box];

With Data^, Data^.Description, Data^.Header Do

Begin

BigArr[Vars['A'].StartOffset] := UserDataType(UserData^).Areal;

End; {With}

BackRequest[Box][10] := False;

End; {With}

End; {BackProc10}

{$F-}

{$F+}

Procedure InitializeUserData(MPCGlobal: MPCGlobalPtr); Export;

Begin

With MPCGlobal^ Do

Begin

UserData := @SomeUserData;

End; {With}

End; {InitializeUserData}

{$F-}

Exports BackProc1;

Exports BackProc2;

Exports BackProc3;

Exports BackProc4;

Exports BackProc5;

Exports BackProc6;

Exports BackProc7;

Exports BackProc8;

Exports BackProc9;

Exports BackProc10;

Exports InitializeUserData;

Begin

End.

Sample Macro file

**Computer Names And IPs**

'Q nel7-glab1 172.17.39.13'

'K nel7-galistel5 172.17.39.16'

'U nel7x32-glab4 172.17.39.17'

'O nel7-glab3 172.17.39.19'

'I nel7-galllab5 172.17.39.15'

'R GALLDATAAN 172.17.34.66'

'Z nel-gallistelpc 172.17.39.18'

1. A global variable in Matlab is accessible from the workspace of any function, provided it is declared as a global variable in the opening statements of that function [↑](#footnote-ref-1)
2. Tufte, E. R. (2001). The visual display of quantitative information. Edition 2. Cheshire, CT, Graphics Press. [↑](#footnote-ref-2)