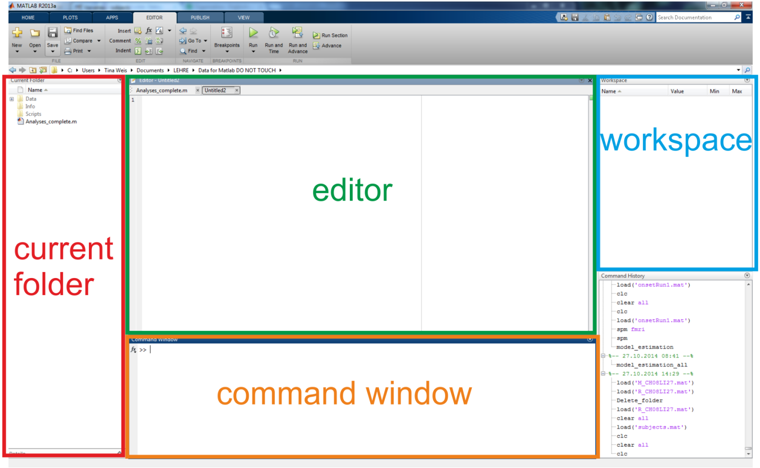
Introduction into Data Analyses with Matlab

Tina Weis

Cognitive and Developmental Psychology
Center for Cognitive Science
University of Kaiserslautern

12. November 2014

Open Matlab



MATLAB R2013a
HOME
PLOTS
APPS
EDITOR
PUBLISH
VIEW
Search Documentation
New
Open
Save
Find Files
Compare
Print
Insert
Comment
Indent
Go To
Find
Breakpoints
Run
Run and Time
Run and Advance
Run Section
Advance
FILE
EDIT
NAVIGATE
BREAKPOINTS
RUN
C: ▸ Users ▸ Tina Weis ▸ Documents ▸ LEHRE ▸ Data for Matlab DO NOT TOUCH ▸
Current Folder
Name
Data
Info
Scripts
Analyses_complete.m
current folder
Editor - Untitled2
Analyses_complete.m
Untitled2
editor
Workspace
Name
Value
Min
Max
workspace
Command Window
command window
Command History
load('onsetRun1.mat')
clc
clear all
clc
load('onsetRun1.mat')
spm fmri
spm
model_estimation
%-- 27.10.2014 08:41 --%
model_estimation_all
%-- 27.10.2014 14:29 --%
load('M_CH08LI27.mat')
load('R_CH08LI27.mat')
Delete_folder
load('R_CH08LI27.mat')
clear all
load('subjects.mat')
clc
clear all
clc

General remarks regarding Matlab

- use *;* to finalize statement
- use *%* to indicate *comments*
- values which are used more often should be defined as *variables*
- clear workspace before running a script (*clear all*)
- use *F9* to run parts of the script
- matlab organizes variable entries into *columns* and *rows*
- use *help* function as much as possible

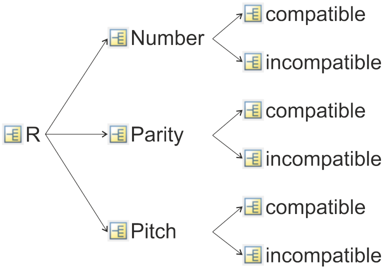
Data set – SNARC meets SPARC

- Spatial Numerical Association of Response Codes (SNARC)
- Spatial Pitch Association of Response Codes (SPARC)
- Mental number line theory

- German number words (1,2,8,9) sung in four different pitches
- Tasks:
  - Magnitude judgment task (smaller / larger than 5)
  - Pitch judgment task (low / high)
  - Parity judgment task (odd / even)
- 2 conditions: compatible vs. incompatible (160 trials each)
- Subjects: 23 right handed participants

Organization of the data set

- Folder *Data* contains one folder for each participant
- Results of participants are stored in *structures* (*struct*)
  ⇒ easy way to store variables in hierarchical order under the need of only one name (saves memory and creativity)
- Inspect the structure of *R* manually: click through the different levels
- To call a entry of a structure within a script: use *dot-operator* (e.g. R.number.compatible)



Organization of the data set - Tables

	1	2	3	4	5	6	7	8	9	10	11
1	12	8	7	530	1	0	566040	566570	33	565742	568808
2	11	8	6	716	1	0	569100	569816	33	568808	570971
3	10	8	4	552	1	0	571256	571808	33	570971	573295
4	6	2	4	543	1	0	573579	574122	2	573295	575622
5	12	8	7	550	1	0	575904	576454	33	575622	577945
6	13	9	3	653	1	0	578227	578880	33	577945	580164
7	3	1	6	609	1	0	580451	581060	2	580164	582431
8	7	2	6	639	1	0	582724	583363	2	582431	584679
9	13	9	3	567	1	0	584964	585531	33	584679	586986
10	11	8	6	703	1	0	587271	587974	33	586986	589158
11	7	2	6	622	1	0	589444	590066	2	589158	591411
12	13	9	3	568	1	0	591701	592269	33	591411	593720
13	16	9	7	634	1	0	594008	594642	33	593720	595963
14	10	8	4	504	1	0	596247	596751	33	595963	598339
15	12	8	7	503	1	0	598621	599124	33	598339	600715
16	3	1	6	631	1	0	600995	601626	2	600715	602958
17	9	8	3	560	1	0	603252	603812	33	602958	605286
18	14	9	4	650	1	0	605576	606226	33	605286	607517
19	5	2	3	484	1	0	607799	608283	2	607517	609907
20	16	9	7	688	1	0	610190	610878	33	609907	612098
21	4	1	7	720	1	0	612379	613099	2	612098	614248
22	5	2	3	778	1	0	614536	615314	2	614248	616348

stimulus RT correct

First steps

- Open a *new script*

First steps

- Open a *new script*
- Start with a *comment* about content, author and date
  $\Rightarrow$ use % for comments

First steps

- Open a *new script*
- Start with a *comment* about content, author and date
  $\Rightarrow$ use % for comments

```
% Comment what you do
% Data analyses with matlab

% Tina Weis (November 2014)
```

First steps

- Open a *new script*
- Start with a *comment* about content, author and date
  $\Rightarrow$ use % for comments

```
% Comment what you do
% Data analyses with matlab

% Tina Weis (November 2014)
```

- *Save script*

First steps

- *Clear workspace*, *close open windows* and *clear command window* to avoid takeover of unintentional data

First steps

- *Clear workspace*, *close open windows* and *clear command window* to avoid takeover of unintentional data

```
% clear
clear all % clear workspace
close all % close open windows
clc % clear command window
```

First steps

- *Clear workspace*, *close open windows* and *clear command window* to avoid takeover of unintentional data

```
% clear
clear all % clear workspace
close all % close open windows
clc % clear command window
```

- Mark rows and *press F9* to run only this part of the script

Define path where your data is stored

- define you first *variable* datapath (*strings* 

(characters) must set in '...' and occur in pink)

Define path where your data is stored

- define you first *variable* datapath (*strings* abc (characters) must set in '... ' and occur in pink)

```
% define datapath as string
datapath = 'C:\Users\Tina Weis\Documents\LEHRE\Data for Matlab DO NOT TOUCH\Data\';
```

Define path where your data is stored

- define you first *variable* *datapath* (*strings* (characters) must set in '...' and occur in pink)

```
% define datapath as string
datapath = 'C:\Users\Tina Weis\Documents\LEHRE\Data for Matlab DO NOT TOUCH\Data\';
```

- ⇒ *F9* ⇒ *datapath* is the first variable occurring in the *workspace* and can be now used instead of the complete path name

Load the names of the participants

- *load subjects* because we need to address individual folder for each participant $\Rightarrow$ to connect two elements use [...]

Load the names of the participants

- *load subjects* because we need to address individual folder for each participant ⇒ to connect two elements use [...]

```
% load subjects
load([datapath 'subjects.mat']);
```

Load the names of the participants

- *load subjects* because we need to address individual folder for each participant ⇒ to connect two elements use [...]

```
% load subjects
load([datapath 'subjects.mat']);
```

- ⇒ *F9* ⇒ *subjects* occurs in the *workspace* as *cell* {}, because names of individual subjects are *strings*

Load data of an individual participant

- *load* data for participant 1 $\Rightarrow$ as before, *strings* can be put together when entered in [...]

Load data of an individual participant

- *load* data for participant 1 $\Rightarrow$ as before, *strings* can be put together when entered in [...]
- to address an entry of a *cell* use the name of the variable *subjects* and specify the *cell* with {...}

Load data of an individual participant

- *load* data for participant 1 ⇒ as before, *strings* can be put together when entered in [...]
- to address an entry of a *cell* use the name of the variable *subjects* and specify the *cell* with {...}

```
% load data for participant 1 into workspace
load([datapath subjects{1} filesep 'R_' subjects{1} '.mat']);
```

Load data of an individual participant

- *load* data for participant 1 ⇒ as before, *strings* can be put together when entered in [...]
- to address an entry of a *cell* use the name of the variable *subjects* and specify the *cell* with {...}

```
% load data for participant 1 into workspace
load([datapath subjects{1} filesep 'R_' subjects{1} '.mat']);
```

- ⇒ *F9* ⇒ *R* occurs in *workspace*
  ⇒ you are able to open this structure and inspect the data by hand

Define specific variable to pic reaction times

- Define variable *rt_raw* for participant 1 (number – compatible) (RT = *column 4* in the table)
  $\Rightarrow$ use *dot-operator* to navigate in *R*

Define specific variable to pic reaction times

- Define variable *rt_raw* for participant 1 (number – compatible) (RT = *column 4* in the table)
  ⇒ use *dot-operator* to navigate in *R*

```
% define variable rt_raw (column 4) for participant 1 (number - compatible)
rt_raw = R.number.compatible(:,4);
```

Define specific variable to pic reaction times

- Define variable *rt_raw* for participant 1 (number – compatible) (RT = *column 4* in the table)
  ⇒ use *dot-operator* to navigate in *R*

```
% define variable rt_raw (column 4) for participant 1 (number - compatible)
rt_raw = R.number.compatible(:,4);
```

- ⇒ *F9* ⇒ *rt_raw* occurs in *workspace*

Use functions to calculate median and standard deviation

- We can use a *function* to calculate the *median* or *standard deviation*
  $\Rightarrow$ *functions* are already implemented, you only have to find them and use them correctly

Use functions to calculate median and standard deviation

- We can use a *function* to calculate the *median* or *standard deviation* ⇒ *functions* are already implemented, you only have to find them and use them correctly
- **help** will help you! ⇒ type *help median* in *command window*

Use functions to calculate median and standard deviation

- We can use a *function* to calculate the *median* or *standard deviation* ⇒ *functions* are already implemented, you only have to find them and use them correctly
- **help** will help you! ⇒ type *help median* in *command window*
- Calculate *median* of the reaction times

Use functions to calculate median and standard deviation

- We can use a *function* to calculate the *median* or *standard deviation* ⇒ *functions* are already implemented, you only have to find them and use them correctly
- **help** will help you! ⇒ type *help median* in *command window*
- Calculate *median* of the reaction times

```
% calculate median
subjects_median = median(rt_raw);
```

Use functions to calculate median and standard deviation

- We can use a *function* to calculate the *median* or *standard deviation* ⇒ *functions* are already implemented, you only have to find them and use them correctly
- **help** will help you! ⇒ type *help median* in *command window*
- Calculate *median* of the reaction times

```
% calculate median
subjects_median = median(rt_raw);
```

- ⇒ *F9* ⇒ result for *median* will occur in variable *subjects_median*

Use functions to calculate median and standard deviation

- We can use a *function* to calculate the *median* or *standard deviation* ⇒ *functions* are already implemented, you only have to find them and use them correctly
- **help** will help you! ⇒ type *help median* in *command window*
- Calculate *median* of the reaction times

```
% calculate median
subjects_median = median(rt_raw);
```

- ⇒ *F9* ⇒ result for *median* will occur in variable *subjects_median*
- Do the same for *standard deviation* (*std*)

Use functions to calculate median and standard deviation

- We can use a *function* to calculate the *median* or *standard deviation*
  ⇒ *functions* are already implemented, you only have to find them and use them correctly
- **help** will help you! ⇒ type *help median* in *command window*
- Calculate *median* of the reaction times

```
% calculate median and std
subjects_median = median(rt_raw);
subjects_std = std(rt_raw);
```

- ⇒ *F9* ⇒ result for *median* will occur in variable *subjects_median*
- Do the same for *standard deviation* (*std*)

Sort data according to stimulus types

- We have to *sort* the data according to *stimulus types* $\Rightarrow$ check *help sort*

Sort data according to stimulus types

- We have to *sort* the data according to *stimulus types* $\Rightarrow$ check *help sort*
- *Stimulus types* are stored in column *1* of the result tables

Sort data according to stimulus types

- We have to *sort* the data according to *stimulus types* ⇒ check *help sort*
- *Stimulus types* are stored in column *1* of the result tables

```
% sort stimuli
[code, order] = sort(R.number.compatible(:,1));
```

Sort data according to stimulus types

- We have to *sort* the data according to *stimulus types* ⇒ check *help sort*
- *Stimulus types* are stored in column *1* of the result tables

```
% sort stimuli
[code, order] = sort(R.number.compatible(:,1));
```

- *sort* *rt_raw* according to *stimulus types*

Sort data according to stimulus types

- We have to *sort* the data according to *stimulus types* ⇒ check *help sort*
- *Stimulus types* are stored in column *1* of the result tables

```
% sort stimuli
[code, order] = sort(R.number.compatible(:,1));
rt_sort = rt_raw(order);
```

- *sort* *rt_raw* according to *stimulus types*
- ⇒ *F9* ⇒ result for *rt_sort* occurs

Define variable to pic correct responses

- Define variable *correct_raw* (accuracy = column 5) for participant 1 (number – compatible) and *sort* according to stimuli

Define variable to pic correct responses

- Define variable *correct_raw* (accuracy = column 5) for participant 1 (number – compatible) and *sort* according to stimuli

```
% define correct
correct_raw = R.number.compatible(:,5);
```

Define variable to pic correct responses

- Define variable *correct_raw* (accuracy = column 5) for participant 1 (number – compatible) and *sort* according to stimuli

```
% define correct
correct_raw = R.number.compatible(:,5);
correct_sort = correct_raw(order);
```

Define variable to pic correct responses

- Define variable *correct_raw* (accuracy = column 5) for participant 1 (number – compatible) and *sort* according to stimuli

```
% define correct
correct_raw = R.number.compatible(:,5);
correct_sort = correct_raw(order);
```

- ⇒ *F9* ⇒ now we have all the data we need

Sort reaction times by using loops

- We need to reorganize the data for easier addressing in the next steps $\Rightarrow$ instead of having a long column including all RT sorted according to stimuli, we will have a matrix with rows indicating the different stimuli and columns indicating the 10 *repetitions* of each stimulus

Sort reaction times by using loops

- We need to reorganize the data for easier addressing in the next steps $\Rightarrow$ instead of having a long column including all RT sorted according to stimuli, we will have a matrix with rows indicating the different stimuli and columns indicating the 10 *repetitions* of each stimulus
- First we have to define *stimuli* $\Rightarrow$ since we want to use them in a *struct* they have to be defined as *strings* $\Rightarrow$ *strings* have to be included into a *cell* because Matlab can not handle *matrices* with *strings*

Sort reaction times by using loops

- We need to reorganize the data for easier addressing in the next steps $\Rightarrow$ instead of having a long column including all RT sorted according to stimuli, we will have a matrix with rows indicating the different stimuli and columns indicating the 10 *repetitions* of each stimulus
- First we have to define *stimuli* $\Rightarrow$ since we want to use them in a *struct* they have to be defined as *strings* $\Rightarrow$ *strings* have to be included into a *cell* because Matlab can not handle *matrices* with *strings*

```
% define stimuli
stimuli = {'oneLL','oneL','oneH','oneHH',...
           'twoLL','twoL','twoH','twoHH',...
           'eightLL','eightL','eightH','eightHH',...
           'nineLL','nineL','nineH','nineHH'};
```

Sort reaction times by using loops

- We need to reorganize the data for easier addressing in the next steps $\Rightarrow$ instead of having a long column including all RT sorted according to stimuli, we will have a matrix with rows indicating the different stimuli and columns indicating the 10 *repetitions* of each stimulus
- First we have to define *stimuli* $\Rightarrow$ since we want to use them in a *struct* they have to be defined as *strings* $\Rightarrow$ *strings* have to be included into a *cell* because Matlab can not handle *matrices* with *strings*

```
% define stimuli
stimuli = {'oneLL','oneL','oneH','oneHH',...
           'twoLL','twoL','twoH','twoHH',...
           'eightLL','eightL','eightH','eightHH',...
           'nineLL','nineL','nineH','nineHH'};
```

- $\Rightarrow$ *F9* $\Rightarrow$ comparable to *subjects stimuli* occur in *workspace* as *cell*

Sort reaction times by using loops

- We need a *for loop* running through all *stimuli*
  ⇒ a *for loop* needs a count-variable *(st)* and the count-interval *(1:16)*
  ⇒ check *length(stimuli)* with *F9*

Sort reaction times by using loops

- We need a *for* *loop* running through all *stimuli*
  ⇒ a *for* *loop* needs a count-variable *(st)* and the count-interval *(1:16)*
  ⇒ check *length(stimuli)* with *F9*

```
for st = 1:length(stimuli)

end;
```

Sort reaction times by using loops

- We need a *for* *loop* running through all *stimuli*
  ⇒ a *for* *loop* needs a count-variable *(st)* and the count-interval *(1:16)*
  ⇒ check *length(stimuli)* with *F9*

```
for st = 1:length(stimuli)

end;
```

- now we have a loop running from 1 to 16 but doing nothing

Sort reaction times by using loops

- since *rt_sort* is already sorted according to the *stimuli*, 10 consecutive trials belong to one stimulus condition

Sort reaction times by using loops

- since *rt_sort* is already sorted according to the *stimuli*, 10 consecutive trials belong to one stimulus condition

```
for st = 1:length(stimuli)

    rt(st,:) = rt_sort(st*10-9:st*10)';

end;
```

Sort reaction times by using loops

- since *rt_sort* is already sorted according to the *stimuli*, 10 consecutive trials belong to one stimulus condition

```
for st = 1:length(stimuli)

    rt(st,:) = rt_sort(st*10-9:st*10)';

end;
```

- $\Rightarrow$ *F9* $\Rightarrow$ variable *rt* is now organized (16,10)

Sort reaction times by using loops

- since *rt_sort* is already sorted according to the *stimuli*, 10 consecutive trials belong to one stimulus condition

```
for st = 1:length(stimuli)

    rt(st,:) = rt_sort(st*10-9:st*10)';

end;
```

- ⇒ *F9* ⇒ variable *rt* is now organized (16,10)
- do the same for *correct_sort*

Sort reaction times by using loops

- since *rt_sort* is already sorted according to the *stimuli*, 10 consecutive trials belong to one stimulus condition

```
rt(st,:) = rt_sort(st*10-9:st*10)';
correct(st,:) = correct_sort(st*10-9:st*10)';
```

- ⇒ *F9* ⇒ variable *rt* is now organized (16,10)
- do the same for *correct_sort*

Run loop through each repetition of stimuli

- To find trials were participants did something wrong (no button press, wrong button press) we have to run a second loop through the 10 repetitions of each *stimuli*

Run loop through each repetition of stimuli

- To find trials were participants did something wrong (no button press, wrong button press) we have to run a second loop through the 10 repetitions of each *stimuli*
- Instead of *length* (only returns the length of a *vector*) we will use *size* (returns the number of rows and colums)

Run loop through each repetition of stimuli

- To find trials were participants did something wrong (no button press, wrong button press) we have to run a second loop through the 10 repetitions of each *stimuli*
- Instead of *length* (only returns the length of a *vector*) we will use *size* (returns the number of rows and colums)

```
for st = 1:length(stimuli)

    rt(st,:) = rt_sort(st*10-9:st*10)';
    correct(st,:) = correct_sort(st*10-9:st*10)';

    for r = 1:size(rt,2)

    end;
end;
```

Find missed trials

- If participants did not press any button, *rt* is smaller than zero
  $\Rightarrow$ *if* statement (used for comparisons)

Find missed trials

- If participants did not press any button, *rt* is smaller than zero
  $\Rightarrow$ *if* statement (used for comparisons)

```
for r = 1:size(rt,2)

    if rt(st,r) < 0

    end;
end;
```

Find missed trials

- If participants did not press any button, *rt* is smaller than zero $\Rightarrow$ *if* statement (used for comparisons)

```
for r = 1:size(rt,2)

    if rt(st,r) < 0

    end;
end;
```

- Save those values in a new variable *nobutton*

Find missed trials

- If participants did not press any button, *rt* is smaller than zero $\Rightarrow$ *if* statement (used for comparisons)

```
if rt(st,r) < 0

    nobutton = rt(st,r);

end;
```

- Save those values in a new variable *nobutton*

Find missed trials

- If participants did not press any button, *rt* is smaller than zero
  ⇒ *if* statement (used for comparisons)

```
if rt(st,r) < 0

    nobutton = rt(st,r);

end;
```

- Save those values in a new variable *nobutton*
- ⇒ *nobutton* will be overwritten with the actual value ⇒ we need to store all values ⇒ implement a *struct* to save all no button events in each condition separately

Find missed trials

- If participants did not press any button, *rt* is smaller than zero
  ⇒ *if* statement (used for comparisons)

```
for st = 1:length(stimuli)

    n = 1;

    rt(st,:) = rt_sort(st*10-9:st*10)';
    correct(st,:) = correct_sort(st*10-9:st*10)';

    for r = 1:size(rt,2)

        if rt(st,r) < 0

            RT.nobutton(n) = rt(st,r);
            n = n+1;

        end;
    end;
end;
```

- Save those values in a new variable *nobutton*
- ⇒ *nobutton* will be overwritten with the actual value ⇒ we need to store all values ⇒ implement a *struct* to save all no button events in each condition separately

Find missed trials

- If participants did not press any button, *rt* is smaller than zero
  ⇒ *if* statement (used for comparisons)

```
for st = 1:length(stimuli)

    n = 1;

    rt(st,:) = rt_sort(st*10-9:st*10)';
    correct(st,:) = correct_sort(st*10-9:st*10)';

    for r = 1:size(rt,2)

        if rt(st,r) < 0

            RT.nobutton.(stimuli{st})(n) = rt(st,r);
            n = n+1;

        end;
    end;
end;
```

- Save those values in a new variable *nobutton*
- ⇒ *nobutton* will be overwritten with the actual value ⇒ we need to store all values ⇒ implement a *struct* to save all no button events in each condition separately

Find slow trials

- We also may want to exclude trials with *rt* smaller than 100 ms, because they seem to be unnatural

Find slow trials

- We also may want to exclude trials with *rt* smaller than 100 ms, because they seem to be unnatural
- To exclude those *rt* in addition to the previous ones, we expand the *if* statement with *elseif*

Find slow trials

- We also may want to exclude trials with *rt* smaller than 100 ms, because they seem to be unnatural
- To exclude those *rt* in addition to the previous ones, we expand the *if* statement with *elseif*

```
if rt(st,r) < 0

    RT.nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

end;
```

Find slow trials

- We also may want to exclude trials with *rt* smaller than 100 ms, because they seem to be unnatural
- To exclude those *rt* in addition to the previous ones, we expand the *if* statement with *elseif*

```
if rt(st,r) < 0

    RT.nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

end;
```

- write results in same *struct* with new name *shorter*

Find slow trials

- We also may want to exclude trials with *rt* smaller than 100 ms, because they seem to be unnatural
- To exclude those *rt* in addition to the previous ones, we expand the *if* statement with *elseif*

```
if rt(st,r) < 0

    RT.nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

    RT.shorter.(stimuli{st})(s) = rt(st,r);
    s = s+1;

end;
```

- write results in same *struct* with new name *shorter*

Find outlier trials

- Outlier criterium: *rt* below and above 2 *std* from *median* should be excluded

Find outlier trials

- Outlier criterium: *rt* below and above 2 *std* from *median* should be excluded
- Use another *elseif* and define the outlier criterium and save in *struct*

Find outlier trials

- Outlier criterium: *rt* below and above 2 *std* from *median* should be excluded
- Use another *elseif* and define the outlier criterium and save in *struct*

```
if rt(st,r) < 0

    RT.nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

    RT.shorter.(stimuli{st})(s) = rt(st,r);
    s = s+1;

elseif rt(st,r) < subjects_median - 2*subjects_std

end;
```

Find outlier trials

- Outlier criterium: *rt* below and above 2 *std* from *median* should be excluded
- Use another *elseif* and define the outlier criterium and save in *struct*

```
if rt(st,r) < 0

    RT.nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

    RT.shorter.(stimuli{st})(s) = rt(st,r);
    s = s+1;

elseif rt(st,r) < subjects_median - 2*subjects_std || ...
        rt(st,r) > subjects_median + 2*subjects_std

end;
```

Find outlier trials

- Outlier criterium: *rt* below and above 2 *std* from *median* should be excluded
- Use another *elseif* and define the outlier criterium and save in *struct*

```
if rt(st,r) < 0

    RT.nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

    RT.shorter.(stimuli{st})(s) = rt(st,r);
    s = s+1;

elseif rt(st,r) < subjects_median - 2*subjects_std || ...
        rt(st,r) > subjects_median + 2*subjects_std

    RT.outlier.(stimuli{st})(o) = rt(st,r);
    o = o+1;
end;
```

- ⇒ still need to exclude errors, where participants pressed wrong button

Find error trials

- To end *if* statement you can use *else* to include all further cases

Find error trials

- To end *if* statement you can use *else* to include all further cases

```
if rt(st,r) < 0

    RT.nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

    RT.shorter.(stimuli{st})(s) = rt(st,r);
    s = s+1;

elseif rt(st,r) < subjects_median - 2*subjects_std || ...
        rt(st,r) > subjects_median + 2*subjects_std

    RT.outlier.(stimuli{st})(o) = rt(st,r);
    o = o+1;

else

end;
```

Find error trials

- To end *if* statement you can use *else* to include all further cases

```
if rt(st,r) < 0

    RT.nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

    RT.shorter.(stimuli{st})(s) = rt(st,r);
    s = s+1;

elseif rt(st,r) < subjects_median - 2*subjects_std || ...
        rt(st,r) > subjects_median + 2*subjects_std

    RT.outlier.(stimuli{st})(o) = rt(st,r);
    o = o+1;

else

end;
```

- We need to find error trials (variable *correct* $= 0$) $\Rightarrow$ new *if* statement

Find error trials

- To end *if* statement you can use *else* to include all further cases

```
else
    if

    end;
end;
```

- We need to find error trials (variable *correct* $= 0$) $\Rightarrow$ new *if* statement

Find error trials

- To end *if* statement you can use *else* to include all further cases

```
else
    if correct(st,r) == 0

    end;
end;
```

- We need to find error trials (variable *correct* $= 0$) $\Rightarrow$ new *if* statement

Find error trials

- find *errors* with appropriate comparison condition and store results into *struct*

Find error trials

- find *errors* with appropriate comparison condition and store results into *struct*

```
else
    if correct(st,r) == 0

        RT.errors.(stimuli{st})(e) = rt(st,r);
        e = e+1;

    end;
end;
```

Find error trials

- find *errors* with appropriate comparison condition and store results into *struct*

```
else
    if correct(st,r) == 0

        RT.errors.(stimuli{st})(e) = rt(st,r);
        e = e+1;

    end;
end;
```

- all other *rt* should be stored in *correct*

Find error trials

- find *errors* with appropriate comparison condition and store results into *struct*

```
else
    if correct(st,r) == 0

        RT.errors.(stimuli{st})(e) = rt(st,r);
        e = e+1;

    else

        RT.correct.(stimuli{st})(c) = rt(st,r);
        c = c+1;

    end;
end;
```

- all other *rt* should be stored in *correct*

Find error trials

- find *errors* with appropriate comparison condition and store results into *struct*

```
else
    if correct(st,r) == 0

        RT.errors.(stimuli{st})(e) = rt(st,r);
        e = e+1;

    else

        RT.correct.(stimuli{st})(c) = rt(st,r);
        c = c+1;

    end;
end;
```

- all other *rt* should be stored in *correct*
- now we can run ▷ the whole script which we wrote so far and see what happens ⇒ inspect variable *RT* in workspace

What we did so far...

- We loaded the data of participant 1 and concentrated on one *task* (number) and the one *run* (compatible)

What we did so far...

- We loaded the data of participant 1 and concentrated on one *task* (number) and the one *run* (compatible)
- We calculated *median* and *std* for definition of the outliers criterium

What we did so far...

- We loaded the data of participant 1 and concentrated on one *task* (number) and the one *run* (compatible)
- We calculated *median* and *std* for definition of the outliers criterium
- We found reaction times *smaller than 0*, *smaller than 100* and *outside of the outlier criterium*

What we did so far...

- We loaded the data of participant 1 and concentrated on one *task* (number) and the one *run* (compatible)
- We calculated *median* and *std* for definition of the outliers criterium
- We found reaction times *smaller than 0*, *smaller than 100* and *outside of the outlier criterium*
- We sorted the remaining reaction times according the *correct answers* of the participant

What we did so far...

- We loaded the data of participant 1 and concentrated on one *task* (number) and the one *run* (compatible)
- We calculated *median* and *std* for definition of the outliers criterium
- We found reaction times *smaller than 0*, *smaller than 100* and *outside of the outlier criterium*
- We sorted the remaining reaction times according the *correct answers* of the participant
- ⇒ All those results are stored in a *struct* named *RT*

What we did so far...

- We loaded the data of participant 1 and concentrated on one *task* (number) and the one *run* (compatible)
- We calculated *median* and *std* for definition of the outliers criterium
- We found reaction times *smaller than 0*, *smaller than 100* and *outside of the outlier criterium*
- We sorted the remaining reaction times according the *correct answers* of the participant
- $\Rightarrow$ All those results are stored in a *struct* named *RT*

- Next step: do the same with *incompatible* run

Find error trials for incompatible run

- We can use the same loops which we already had but with a few changes

Find error trials for incompatible run

- We can use the same loops which we already had but with a few changes
- Since we need a few more variables to define now, we will put the definition of the *stimuli* more on the beginning of the script behind *load([subject...])*

Find error trials for incompatible run

- We can use the same loops which we already had but with a few changes
- Since we need a few more variables to define now, we will put the definition of the *stimuli* more on the beginning of the script behind *load([subject...])*

```
% load subjects
load([datapath 'subjects.mat']);

% VARIABLES
% define stimuli
stimuli = {'oneLL','oneL','oneH','oneHH',...
           'twoLL','twoL','twoH','twoHH',...
           'eightLL','eightL','eightH','eightHH',...
           'nineLL','nineL','nineH','nineHH'};
```

Find error trials for incompatible run

- Define a variable which allows to call either *compatible* or *incompatible* run named *compatibility*

Find error trials for incompatible run

- Define a variable which allows to call either *compatible* or *incompatible* run named *compatibility*

```
% load subjects
load([datapath 'subjects.mat']);

% VARIABLES
% define stimuli
stimuli = {'oneLL','oneL','oneH','oneHH',...
           'twoLL','twoL','twoH','twoHH',...
           'eightLL','eightL','eightH','eightHH',...
           'nineLL','nineL','nineH','nineHH'};

compatibility = {'compatible','incompatible'};
```

Find error trials for incompatible run

- Define a variable which allows to call either *compatible* or *incompatible* run named *compatibility*

```
% load subjects
load([datapath 'subjects.mat']);

% VARIABLES
% define stimuli
stimuli = {'oneLL','oneL','oneH','oneHH',...
           'twoLL','twoL','twoH','twoHH',...
           'eightLL','eightL','eightH','eightHH',...
           'nineLL','nineL','nineH','nineHH'};

compatibility = {'compatible','incompatible'};
```

- We need a new *for* loop to run through the two entries in *compatibility*
  $\Rightarrow$ but where to start the loop in the script?

Find error trials for incompatible run

- Define a variable which allows to call either *compatible* or *incompatible* run named *compatibility*

```
% load data for participant 1 into workspace
load([datapath subjects{1} filesep 'R_' subjects{1} '.mat']);

for comp = 1:length(compatibility)

    % define variable rt_raw (column 4) for participant 1 (number - compati
    rt_raw = R.number.compatible(:,4);

    % calculate median
    subjects_median = median(rt_raw);
    subjects_std = std(rt_raw);

    % sort stimuli
    [code, order] = sort(R.number.compatible(:,1));
    rt_sort = rt_raw(order);
```

- We need a new *for* loop to run through the two entries in *compatibility*
  $\Rightarrow$ but where to start the loop in the script?

Find error trials for incompatible run

- Modify the existing loop, that it calls either compatible or incompatible run, depending on the value of counter *comp*

Find error trials for incompatible run

- Modify the existing loop, that it calls either compatible or incompatible run, depending on the value of counter *comp*

```
for comp = 1:length(compatibility)

    % define variable rt_raw (column 4) for participant 1 (number - compati
    rt_raw = R.number.(compatibility{comp})(:,4);

    % calculate median
    subjects_median = median(rt_raw);
    subjects_std = std(rt_raw);

    % sort stimuli
    [code, order] = sort(R.number.(compatibility{comp})(:,1));
    rt_sort = rt_raw(order);

    % define correct
    correct_raw = R.number.(compatibility{comp})(:,5);
    correct_sort = correct_raw(order);
```

Find error trials for incompatible run

- Modify the existing loop, that it calls either compatible or incompatible run, depending on the value of counter *comp*

```
for comp = 1:length(compatibility)

    % define variable rt_raw (column 4) for participant 1 (number - compati
    rt_raw = R.number.(compatibility{comp})(:,4);

    % calculate median
    subjects_median = median(rt_raw);
    subjects_std = std(rt_raw);

    % sort stimuli
    [code, order] = sort(R.number.(compatibility{comp})(:,1));
    rt_sort = rt_raw(order);

    % define correct
    correct_raw = R.number.(compatibility{comp})(:,5);
    correct_sort = correct_raw(order);
```

- ⇒ now we load *rt* and *correct* data of either the compatible or incompatible run

Find error trials for incompatible run

- We have to enlarge our resulting *struct* RT by a new level; otherwise, results will be overwritten

Find error trials for incompatible run

- We have to enlarge our resulting *struct* *RT* by a new level; otherwise, results will be overwritten

```
if rt(st,r) < 0

    RT.(compatibility{comp}).nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

    RT.(compatibility{comp}).shorter.(stimuli{st})(s) = rt(st,r);
    s = s+1;

elseif rt(st,r) < subjects_median - 2*subjects_std || ...
        rt(st,r) > subjects_median + 2*subjects_std

    RT.(compatibility{comp}).outlier.(stimuli{st})(o) = rt(st,r);
    o = o+1;

else
    if correct(st,r) == 0

        RT.(compatibility{comp}).errors.(stimuli{st})(e) = rt(st,r);
        e = e+1;

    else

        RT.(compatibility{comp}).correct.(stimuli{st})(c) = rt(st,r);
        c = c+1;

    end;

end;
```

Find error trials for incompatible run

- We have to enlarge our resulting *struct* *RT* by a new level; otherwise, results will be overwritten

```
if rt(st,r) < 0

    RT.(compatibility{comp}).nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

    RT.(compatibility{comp}).shorter.(stimuli{st})(s) = rt(st,r);
    s = s+1;

elseif rt(st,r) < subjects_median - 2*subjects_std || ...
        rt(st,r) > subjects_median + 2*subjects_std

    RT.(compatibility{comp}).outlier.(stimuli{st})(o) = rt(st,r);
    o = o+1;

else
    if correct(st,r) == 0

        RT.(compatibility{comp}).errors.(stimuli{st})(e) = rt(st,r);
        e = e+1;

    else

        RT.(compatibility{comp}).correct.(stimuli{st})(c) = rt(st,r);
        c = c+1;

    end;

end;
```

Different tasks

- We need another loop allowing for entering the different tasks in participant 1 $\Rightarrow$ define new variable *task*

Different tasks

- We need another loop allowing for entering the different tasks in participant 1 $\Rightarrow$ define new variable *task*

```
% VARIABLES
% define stimuli
stimuli = {'oneLL','oneL','oneH','oneHH',...
           'twoLL','twoL','twoH','twoHH',...
           'eightLL','eightL','eightH','eightHH',...
           'nineLL','nineL','nineH','nineHH'};

compatibility = {'compatible','incompatible'};
task = {'number', 'pitch', 'parity'};
```

Different tasks

- We need another loop allowing for entering the different tasks in participant 1 $\Rightarrow$ define new variable *task*

```
% VARIABLES
% define stimuli
stimuli = {'oneLL','oneL','oneH','oneHH',...
           'twoLL','twoL','twoH','twoHH',...
           'eightLL','eightL','eightH','eightHH',...
           'nineLL','nineL','nineH','nineHH'};

compatibility = {'compatible','incompatible'};
task = {'number', 'pitch', 'parity'};
```

- build the *for* loop for the *task*

Different tasks

- We need another loop allowing for entering the different tasks in participant 1 $\Rightarrow$ define new variable *task*

```
% load data for participant 1 into workspace
load([datapath subjects{1} filesep 'R_' subjects{1} '.mat']);

for ta = 1:length(task)

    for comp = 1:length(compatibility)

        % define variable rt_raw (column 4) for participant 1 (number - com
        rt_raw = R.number.(compatibility{comp})(:,4);
```

- build the *for* loop for the *task*

Different tasks

- We need another loop allowing for entering the different tasks in participant 1 ⇒ define new variable *task*

```
% load data for participant 1 into workspace
load([datapath subjects{1} filesep 'R_' subjects{1} '.mat']);

for ta = 1:length(task)

    for comp = 1:length(compatibility)

        % define variable rt_raw (column 4) for participant 1 (number - com
        rt_raw = R.number.(compatibility{comp})(:,4);
```

- build the *for* loop for the *task*
- ⇒ let the program know which data to load in which task

Different tasks

- We need another loop allowing for entering the different tasks in participant 1 ⇒ define new variable *task*

```
for ta = 1:length(task)

    for comp = 1:length(compatibility)

        % define variable rt_raw (column 4) for participant 1 (number - com
        rt_raw = R.(task{ta}).(compatibility{comp})(:,4);

        % calculate median
        subjects_median = median(rt_raw);
        subjects_std = std(rt_raw);

        % sort stimuli
        [code, order] = sort(R.(task{ta}).(compatibility{comp})(:,1));
        rt_sort = rt_raw(order);

        % define correct
        correct_raw = R.(task{ta}).(compatibility{comp})(:,5);
        correct_sort = correct_raw(order);
```

- build the *for* loop for the *task*
- ⇒ let the program know which data to load in which task

Different tasks

- We need another loop allowing for entering the different tasks in participant 1 ⇒ define new variable *task*

```
for ta = 1:length(task)

    for comp = 1:length(compatibility)

        % define variable rt_raw (column 4) for participant 1 (number - com
        rt_raw = R.(task{ta}).(compatibility{comp})(:,4);

        % calculate median
        subjects_median = median(rt_raw);
        subjects_std = std(rt_raw);

        % sort stimuli
        [code, order] = sort(R.(task{ta}).(compatibility{comp})(:,1));
        rt_sort = rt_raw(order);

        % define correct
        correct_raw = R.(task{ta}).(compatibility{comp})(:,5);
        correct_sort = correct_raw(order);
```

- build the *for* loop for the *task*
- ⇒ let the program know which data to load in which task
- ⇒ ... and where to store the RTs

Different tasks

- We need another loop allowing for entering the different tasks in participant 1 ⇒ define new variable *task*

```
if rt(st,r) < 0

    RT.(task{ta}).(compatibility{comp}).nobutton.(stimuli{st})(n) = rt(st,r);
    n = n+1;

elseif rt(st,r) < 100

    RT.(task{ta}).(compatibility{comp}).shorter.(stimuli{st})(s) = rt(st,r);
    s = s+1;

elseif rt(st,r) < subjects_median - 2*subjects_std || ...
        rt(st,r) > subjects_median + 2*subjects_std

    RT.(task{ta}).(compatibility{comp}).outlier.(stimuli{st})(o) = rt(st,r);
    o = o+1;

else
    if correct(st,r) == 0

        RT.(task{ta}).(compatibility{comp}).errors.(stimuli{st})(e) = rt(st,r);
        e = e+1;

    else

        RT.(task{ta}).(compatibility{comp}).correct.(stimuli{st})(c) = rt(st,r);
        c = c+1;

    end;

end;
```

Save the preprocessed data

- For later use, we save *RT* using the *save* function ⇒ *help save*

Save the preprocessed data

- For later use, we save *RT* using the *save* function ⇒ *help save*

```
end;

save([datapath filesep subjects{1} filesep 'RT_' subjects{1} '.mat'], 'RT');
```

Save the preprocessed data

- For later use, we save *RT* using the *save* function ⇒ *help save*

```
end;

save([datapath filesep subjects{1} filesep 'RT_' subjects{1} '.mat'], 'RT');
```

- 

More than one participant...

- We have to run the same preprocessing *for all participants* $\Rightarrow$ in contrast to other loops we have to start loop before individual data is loaded into workspace (close loop behind the *save* command, because we need to save the results for each subject individually)

More than one participant...

- We have to run the same preprocessing *for all participants* ⇒ in contrast to other loops we have to start loop before individual data is loaded into workspace (close loop behind the *save* command, because we need to save the results for each subject individually)

```
    compatibility = {'compatible','incompatible'};
    task = {'number', 'pitch', 'parity'};

for sub = 1:length(subjects)

        % load data for participant 1 into workspace
        load([datapath subjects{1} filesep 'R_' subjects{1} '.mat']);

        for ta = 1:length(task)
```

```

        save([datapath filesep subjects{1} filesep 'RT_' subjects{1} '.mat'], '

end;
```

More than one participant...

- We have to run the same preprocessing *for all participants* ⇒ in contrast to other loops we have to start loop before individual data is loaded into workspace (close loop behind the *save* command, because we need to save the results for each subject individually)

```
compatibility = {'compatible','incompatible'};
task = {'number', 'pitch', 'parity'};

for sub = 1:length(subjects)

    % load data for participant 1 into workspace
    load([datapath subjects{1} filesep 'R_' subjects{1} '.mat']);

    for ta = 1:length(task)
```

```

    save([datapath filesep subjects{1} filesep 'RT_' subjects{1} '.mat'], '

end;
```

- We have to replace *subjects*{1} with *subjects*{*sub*}

More than one participant...

- We have to run the same preprocessing *for all participants* $\Rightarrow$ in contrast to other loops we have to start loop before individual data is loaded into workspace (close loop behind the *save* command, because we need to save the results for each subject individually)

```
for sub = 1:length(subjects)

    % load data for participant 1 into workspace
    load([datapath subjects{sub} filesep 'R_' subjects{sub} '.mat']);
```

```
    save([datapath filesep subjects{sub} filesep 'RT_' subjects{sub} '.mat'], 'RT');

end;
```

- We have to replace $subjects\{1\}$ with $subjects\{sub\}$

More than one participant...

- We have to run the same preprocessing *for all participants* ⇒ in contrast to other loops we have to start loop before individual data is loaded into workspace (close loop behind the *save* command, because we need to save the results for each subject individually)

```
for sub = 1:length(subjects)

    % load data for participant 1 into workspace
    load([datapath subjects{sub} filesep 'R_' subjects{sub} '.mat']);

    save([datapath filesep subjects{sub} filesep 'RT_' subjects{sub} '.mat'], 'RT');

end;
```

- We have to replace *subjects*{1} with *subjects*{*sub*}
- ⇒ Preprocessing for all participants is finished!

Build mean for each condition

- in the further steps we only work with the *mean* of each condition
  $\Rightarrow$ we have to build the *mean* of each condition at the end of the *for* loop for the repetitions

Build mean for each condition

- in the further steps we only work with the *mean* of each condition
  $\Rightarrow$ we have to build the *mean* of each condition at the end of the *for* loop for the repetitions

```
RT.(task{ta}).(compatibility{comp}).mean(st) = mean(RT.(task{ta}).(compatibility{comp}).correct.(stimuli{st}));
```

Build mean for each condition

- in the further steps we only work with the *mean* of each condition $\Rightarrow$ we have to build the *mean* of each condition at the end of the *for* loop for the repetitions

```
RT.(task{ta}).(compatibility{comp}).mean(st) = mean(RT.(task{ta}).(compatibility{comp}).correct.(stimuli{st}));
```

- *clear* *RT* after each participant to omit overwriting

Build mean for each condition

- in the further steps we only work with the *mean* of each condition
  $\Rightarrow$ we have to build the *mean* of each condition at the end of the *for* loop for the repetitions

```
save([datapath filesep subjects{sub} filesep 'RT_' subjects{sub} '.mat'], 'RT');
clear RT
```

- *clear* *RT* after each participant to omit overwriting

...

Sorting the data

- We open a *new script* and start with the general beginning
  $\Rightarrow$ *comments* and *clear*

Sorting the data

- We open a *new script* and start with the general beginning
  ⇒ *comments* and *clear*

```
% Analyses of SNARC and SPARC

% Tina Weis (November 2014)

clear all
close all
clc
```

Sorting the data

- We define the path were our data is stored

Sorting the data

- We define the path were our data is stored

```
% define datapath as string
datapath = 'C:\Data for Matlab DO NOT TOUCH\Data\';
```

Sorting the data

- We define the path were our data is stored

```
% define datapath as string
datapath = 'C:\Data for Matlab DO NOT TOUCH\Data\';
```

- .. and load the names of our participants

Sorting the data

- We define the path were our data is stored

```
% load subjects
load([datapath 'subjects.mat']);
```

- .. and load the names of our participants

Sorting the data

- We now load the sorted RT data of participant 1

Sorting the data

- We now load the sorted RT data of participant 1

```
% load data of participant 1
load([datapath filesep subjects{1} filesep 'RT_' subjects{1} '.mat']);
```

hier ein bild wie die daten sortiert werden sollten

Left vs. right hand

- we need to initialize two empty *vectors*, which are filled with *zeros*, having the number of columns according to the *conditions* (16)

Left vs. right hand

- we need to initialize two empty *vectors*, which are filled with *zeros*, having the number of columns according to the *conditions* (16)

```
left = zeros(1,16);
right = zeros(1,16);
```

Left vs. right hand

- we need to initialize two empty *vectors*, which are filled with *zeros*, having the number of columns according to the *conditions* (16)

```
left = zeros(1,16);
right = zeros(1,16);
```

- we have to differentiate between *compatible* and *incompatible* condition, therefore we need a variable *compatibility*

Left vs. right hand

- we need to initialize two empty *vectors*, which are filled with *zeros*, having the number of columns according to the *conditions* (16)

```
% load subjects
load([datapath 'subjects.mat']);

% Variable
compatibility = {'compatible', 'incompatible'};

% load data of participant 1
load([datapath filesep subjects{1} filesep 'RT_' subjects{1} '.mat']);

left = zeros(1,16);
right = zeros(1,16);
```

- we have to differentiate between *compatible* and *incompatible* condition, therefore we need a variable *compatibility*

Left vs. right hand

- we need to initialize two empty *vectors*, which are filled with *zeros*, having the number of columns according to the *conditions* (16)

```
% load subjects
load([datapath 'subjects.mat']);

% Variable
compatibility = {'compatible', 'incompatible'};

% load data of participant 1
load([datapath filesep subjects{1} filesep 'RT_' subjects{1} '.mat']);

left = zeros(1,16);
right = zeros(1,16);
```

- we have to differentiate between *compatible* and *incompatible* condition, therefore we need a variable *compatibility*
- ⇒ this variable has to be assessed by a *for* loop

Left vs. right hand

- we need to initialize two empty *vectors*, which are filled with *zeros*, having the number of columns according to the *conditions* (16)

```
    left = zeros(1,16);
    right = zeros(1,16);

for c = 1:length(compatibility)

end;
```

- we have to differentiate between *compatible* and *incompatible* condition, therefore we need a variable *compatibility*
- $\Rightarrow$ this variable has to be assessed by a *for* loop

Left vs. right hand

- if it is a *compatible* run *small* numbers should be organized in the *left* variable because answered with the *left* hand and *large* numbers should be organized in the *right* variable because answered with the *right* hand
$\Rightarrow$ use *if* statement to assess the two conditions of *compatibility*

Left vs. right hand

- if it is a *compatible* run *small* numbers should be organized in the *left* variable because answered with the *left* hand and *large* numbers should be organized in the *right* variable because answered with the *right* hand $\Rightarrow$ use *if* statement to assess the two conditions of *compatibility*

```
for c = 1:length(compatibility)

    if c == 1 % compatible

    else % incompatible

    end;

end;
```

Left vs. right hand

- if it is a *compatible* run *small* numbers should be organized in the *left* variable because answered with the *left* hand and *large* numbers should be organized in the *right* variable because answered with the *right* hand
  $\Rightarrow$ use *if* statement to assess the two conditions of *compatibility*

```
for c = 1:length(compatibility)

    if c == 1 % compatible

        left(1, 1:8) = RT.number.(compatibility{c}).mean(1:8);
        right(1, 9:16) = RT.number.(compatibility{c}).mean(9:16);

    else % incompatible

    end;

end;
```

Left vs. right hand

- if it is a *compatible* run *small* numbers should be organized in the *left* variable because answered with the *left* hand and *large* numbers should be organized in the *right* variable because answered with the *right* hand $\Rightarrow$ use *if* statement to assess the two conditions of *compatibility*

```
for c = 1:length(compatibility)

    if c == 1 % compatible

        left(1, 1:8) = RT.number.(compatibility{c}).mean(1:8);
        right(1, 9:16) = RT.number.(compatibility{c}).mean(9:16);

    else % incompatible

    end;

end;
```

- vice versa in the incompatible run

Left vs. right hand

- if it is a *compatible* run *small* numbers should be organized in the *left* variable because answered with the *left* hand and *large* numbers should be organized in the *right* variable because answered with the *right* hand ⇒ use *if* statement to assess the two conditions of *compatibility*

```
for c = 1:length(compatibility)

    if c == 1 % compatible

        left(1, 1:8) = RT.number.(compatibility{c}).mean(1:8);
        right(1, 9:16) = RT.number.(compatibility{c}).mean(9:16);

    else % incompatible

        left(1, 9:16) = RT.number.(compatibility{c}).mean(9:16);
        right(1, 1:8) = RT.number.(compatibility{c}).mean(1:8);

    end;

end;
```

- vice versa in the incompatible run

Group comparisons

- Since we want to compare between the individual participants, it is better to store the results for all participants into one variable

Group comparisons

- Since we want to compare between the individual participants, it is better to store the results for all participants into one variable
- We can access the data of the individual participants again via a *for* loop $\Rightarrow$ start, before data of participant 1 is loaded (indent and end loop) and replace $\{1\}$ by $\{sub\}$

Group comparisons

- Since we want to compare between the individual participants, it is better to store the results for all participants into one variable
- We can access the data of the individual participants again via a *for* loop ⇒ start, before data of participant 1 is loaded (indent and end loop) and replace {1} by {*sub*}

```
% Variable
compatibility = {'compatible', 'incompatible'};

for sub = 1:length(subjects)

    % load data of participant 1
    load([datapath filesep subjects{1} filesep 'RT_' subjects{1} '.mat']);
```

Group comparisons

- Since we want to compare between the individual participants, it is better to store the results for all participants into one variable
- We can access the data of the individual participants again via a *for* loop ⇒ start, before data of participant 1 is loaded (indent and end loop) and replace {1} by {*sub*}

```
for sub = 1:length(subjects)

    % load data of participant 1
    load([datapath filesep subjects{sub} filesep 'RT_' subjects{sub} '.mat']);
```

Group comparisons

- Since we want to compare between the individual participants, it is better to store the results for all participants into one variable
- We can access the data of the individual participants again via a *for* loop $\Rightarrow$ start, before data of participant 1 is loaded (indent and end loop) and replace {1} by {*sub*}

```
for sub = 1:length(subjects)

    % load data of participant 1
    load([datapath filesep subjects{sub} filesep 'RT_' subjects{sub} '.mat']);
```

- variable *left* and *right* should be extended by a row for each participant

Group comparisons

- Since we want to compare between the individual participants, it is better to store the results for all participants into one variable
- We can access the data of the individual participants again via a *for* loop ⇒ start, before data of participant 1 is loaded (indent and end loop) and replace {1} by {*sub*}

```
left = zeros(sub,16);
right = zeros(sub,16);

for c = 1:length(compatibility)

    if c == 1 % compatible

        left(sub, 1:8) = RT.number.(compatibility{c}).mean(1:8);
        right(sub, 9:16) = RT.number.(compatibility{c}).mean(9:16);

    else % incompatible

        left(sub, 9:16) = RT.number.(compatibility{c}).mean(9:16);
        right(sub, 1:8) = RT.number.(compatibility{c}).mean(1:8);

    end;

end;
```

- variable *left* and *right* should be extended by a row for each participant

Different tasks

- again we not only have one task but three $\Rightarrow$ define variable *task*

Different tasks

- again we not only have one task but three ⇒ define variable *task*

```
% Variable
compatibility = {'compatible', 'incompatible'};
task = {'number', 'pitch', 'parity'};
```

Different tasks

- again we not only have one task but three ⇒ define variable *task*

```
% Variable
compatibility = {'compatible', 'incompatible'};
task = {'number', 'pitch', 'parity'};
```

- introduce a *for* loop running about all participants

Different tasks

- again we not only have one task but three ⇒ define variable *task*

```
% Variable
compatibility = {'compatible', 'incompatible'};
task = {'number', 'pitch', 'parity'};

for t = 1:length(task)

    for sub = 1:length(subjects)
```

- introduce a *for* loop running about all participants

Different tasks

- again we not only have one task but three ⇒ define variable *task*

```
    % Variable
    compatibility = {'compatible', 'incompatible'};
    task = {'number', 'pitch', 'parity'};

for t = 1:length(task)

        for sub = 1:length(subjects)
```

- introduce a *for* loop running about all participants
- use *elseif* statement for addressing *task* in *compatible* and *incompatible*

Different tasks

- again we not only have one task but three ⇒ define variable *task*

```
            if c == 1 % compatible

                if t == 1 % if number task

                    left(sub, 1:8) = RT.number.(compatibility{c}).mean(1:8);
                    right(sub, 9:16) = RT.number.(compatibility{c}).mean(9:16);

                elseif t == 2 % if pitch task

                end;

```

- introduce a *for* loop running about all participants
- use *elseif* statement for addressing *task* in *compatible* and *incompatible*

Different tasks

- again we not only have one task but three ⇒ define variable *task*

```
else % incompatible

    if t == 1 % if number task

        left(sub, 9:16) = RT.number.(compatibility{c}).mean(9:16);
        right(sub, 1:8) = RT.number.(compatibility{c}).mean(1:8);

    elseif t == 2

    end;

end;
```

- introduce a *for* loop running about all participants
- use *elseif* statement for addressing *task* in *compatible* and *incompatible*

Different tasks

- sort *pitch* data according to hand: *compatible*

Different tasks

- sort *pitch* data according to hand: *compatible*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
left	1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH
right	1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH

Different tasks

- sort *pitch* data according to hand: *compatible* and *incompatible*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
left	1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH
right	1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH

Pitch judgment

- sort pitch data according to hand for *compatible*

Pitch judgment

- sort pitch data according to hand for *compatible*

```
            if c == 1 % compatible

                if t == 1 % if number task

                    left(sub, 1:8) = RT.number.(compatibility{c}).mean(1:8);
                    right(sub, 9:16) = RT.number.(compatibility{c}).mean(9:16);

                elseif t == 2 % if pitch task

                    left(sub, [1:2 5:6 9:10 13:14]) = RT.number.(compatibility{c}).mean([1:2 5:6 9:10 13:14]);
                    right(sub, [3:4 7:8 11:12 15:16]) = RT.number.(compatibility{c}).mean([3:4 7:8 11:12 15:16]);

                end;
```

Pitch judgment

- sort pitch data according to hand for *compatible* and *incompatible* run

```
                    if t == 1 % if number task

                        left(sub, 9:16) = RT.number.(compatibility{c}).mean(9:16);
                        right(sub, 1:8) = RT.number.(compatibility{c}).mean(1:8);

                    elseif t == 2

                        left(sub, [3:4 7:8 11:12 15:16]) = RT.number.(compatibility{c}).mean([3:4 7:8 11:12 15:16]);
                        right(sub, [1:2 5:6 9:10 13:14]) = RT.number.(compatibility{c}).mean([1:2 5:6 9:10 13:14]);

                    end;
```

Parity judgment

- sort parity data according to hand for *compatible*

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Parity judgment

- sort parity data according to hand for *compatible*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
left	1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH
right	1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH

Parity judgment

- sort parity data according to hand for *compatible* and *incompatible* run

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
left	1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH
right	1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH

Parity judgment

- sort parity data according to hand for *compatible*

```
if t == 1 % if number task

    left(sub, 1:8) = RT.number.(compatibility{c}).mean(1:8);
    right(sub, 9:16) = RT.number.(compatibility{c}).mean(9:16);

elseif t == 2 % if pitch task

    left(sub, [1:2 5:6 9:10 13:14]) = RT.number.(compatibility{c}).mean([1:2 5:6 9:10 13:14]);
    right(sub, [3:4 7:8 11:12 15:16]) = RT.number.(compatibility{c}).mean([3:4 7:8 11:12 15:16]);

else % parity task

    left(sub, [1:4 13:16]) = RT.number.(compatibility{c}).mean([1:4 13:16]);
    right(sub, 5:12) = RT.number.(compatibility{c}).mean(5:12);

end;
```

Parity judgment

- sort parity data according to hand for *compatible* and *incompatible* run

```
            if t == 1 % if number task

                left(sub, 9:16) = RT.number.(compatibility{c}).mean(9:16);
                right(sub, 1:8) = RT.number.(compatibility{c}).mean(1:8);

            elseif t == 2

                left(sub, [3:4 7:8 11:12 15:16]) = RT.number.(compatibility{c}).mean([3:4 7:8 11:12 15:16]);
                right(sub, [1:2 5:6 9:10 13:14]) = RT.number.(compatibility{c}).mean([1:2 5:6 9:10 13:14]);

            else % parity task

                left(sub, 5:12) = RT.number.(compatibility{c}).mean(5:12);
                right(sub, [1:4 13:16]) = RT.number.(compatibility{c}).mean([1:4 13:16]);

            end;
```

Find correct task

- At the moment data is only taken from *number* task, also in *parity* and *pitch* task $\Rightarrow$ choose the correct task with the loop counter, also for *compatible* and *incompatible* runs

Find correct task

- At the moment data is only taken from *number* task, also in *parity* and *pitch* task ⇒ choose the correct task with the loop counter, also for *compatible* and *incompatible* runs

```
if t == 1 % if number task

    left(sub, 1:8) = RT.number.(compatibility{c}).mean(1:8);
    right(sub, 9:16) = RT.number.(compatibility{c}).mean(9:16);

elseif t == 2 % if pitch task

    left(sub, [1:2 5:6 9:10 13:14]) = RT.number.(compatibility{c}).mean([1:2 5:6 9:10 13:14]);
    right(sub, [3:4 7:8 11:12 15:16]) = RT.number.(compatibility{c}).mean([3:4 7:8 11:12 15:16]);

else % parity task

    left(sub, [1:4 13:16]) = RT.number.(compatibility{c}).mean([1:4 13:16]);
    right(sub, 5:12) = RT.number.(compatibility{c}).mean(5:12);

end;
```

Find correct task

- At the moment data is only taken from *number* task, also in *parity* and *pitch* task ⇒ choose the correct task with the loop counter, also for *compatible* and *incompatible* runs

```
        if t == 1 % if number task

            left(sub, 1:8) = RT.(task{t}).(compatibility{c}).mean(1:8);
            right(sub, 9:16) = RT.(task{t}).(compatibility{c}).mean(9:16);

        elseif t == 2 % if pitch task

            left(sub, [1:2 5:6 9:10 13:14]) = RT.(task{t}).(compatibility{c}).mean([1:2 5:6 9:10 13:14]);
            right(sub, [3:4 7:8 11:12 15:16]) = RT.(task{t}).(compatibility{c}).mean([3:4 7:8 11:12 15:16]);

        else % parity task

            left(sub, [1:4 13:16]) = RT.(task{t}).(compatibility{c}).mean([1:4 13:16]);
            right(sub, 5:12) = RT.(task{t}).(compatibility{c}).mean(5:12);

        end;
```

Find correct task

- At the moment data is only taken from *number* task, also in *parity* and *pitch* task ⇒ choose the correct task with the loop counter, also for *compatible* and *incompatible* runs

```
if t == 1 % if number task

    left(sub, 9:16) = RT.(task{t}).(compatibility{c}).mean(9:16);
    right(sub, 1:8) = RT.(task{t}).(compatibility{c}).mean(1:8);

elseif t == 2

    left(sub, [3:4 7:8 11:12 15:16]) = RT.(task{t}).(compatibility{c}).mean([3:4 7:8 11:12 15:16]);
    right(sub, [1:2 5:6 9:10 13:14]) = RT.(task{t}).(compatibility{c}).mean([1:2 5:6 9:10 13:14]);

else % parity task

    left(sub, 5:12) = RT.(task{t}).(compatibility{c}).mean(5:12);
    right(sub, [1:4 13:16]) = RT.(task{t}).(compatibility{c}).mean([1:4 13:16]);

end;
```

Find correct task

- At the moment data is only taken from *number* task, also in *parity* and *pitch* task ⇒ choose the correct task with the loop counter, also for *compatible* and *incompatible* runs

```
if t == 1 % if number task

    left(sub, 9:16) = RT.(task{t}).(compatibility{c}).mean(9:16);
    right(sub, 1:8) = RT.(task{t}).(compatibility{c}).mean(1:8);

elseif t == 2

    left(sub, [3:4 7:8 11:12 15:16]) = RT.(task{t}).(compatibility{c}).mean([3:4 7:8 11:12 15:16]);
    right(sub, [1:2 5:6 9:10 13:14]) = RT.(task{t}).(compatibility{c}).mean([1:2 5:6 9:10 13:14]);

else % parity task

    left(sub, 5:12) = RT.(task{t}).(compatibility{c}).mean(5:12);
    right(sub, [1:4 13:16]) = RT.(task{t}).(compatibility{c}).mean([1:4 13:16]);

end;
```

Find correct task

- At the moment data is only taken from *number* task, also in *parity* and *pitch* task ⇒ choose the correct task with the loop counter, also for *compatible* and *incompatible* runs

```
if t == 1 % if number task

    left(sub, 9:16) = RT.(task{t}).(compatibility{c}).mean(9:16);
    right(sub, 1:8) = RT.(task{t}).(compatibility{c}).mean(1:8);

elseif t == 2

    left(sub, [3:4 7:8 11:12 15:16]) = RT.(task{t}).(compatibility{c}).mean([3:4 7:8 11:12 15:16]);
    right(sub, [1:2 5:6 9:10 13:14]) = RT.(task{t}).(compatibility{c}).mean([1:2 5:6 9:10 13:14]);

else % parity task

    left(sub, 5:12) = RT.(task{t}).(compatibility{c}).mean(5:12);
    right(sub, [1:4 13:16]) = RT.(task{t}).(compatibility{c}).mean([1:4 13:16]);

end;
```

- 
- you will see, that everything will be overwritten by the last task ⇒ we have to save the data for individual tasks

But, where to save the data?

- The data in *left* and *right* should be stored in the *struct S*

But, where to save the data?

- The data in *left* and *right* should be stored in the *struct* *S*
- found the *end* belonging to the *sub* loop

But, where to save the data?

- The data in *left* and *right* should be stored in the *struct S*
- found the *end* belonging to the *sub* loop

```
        S.(task{t}) = struct('left', struct('all', left), 'right', struct('all', right));

end;
```

But, where to save the data?

- The data in *left* and *right* should be stored in the *struct S*
- found the *end* belonging to the *sub* loop

```
        S.(task{t}) = struct('left', struct('all', left), 'right', struct('all', right));

end;
```

- 

Summarizing conditions

- for easier analysis we can group some conditions

1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH

Small Low	Small High	Large Low	Large High

Summarizing conditions

- for easier analysis we can group some conditions

1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH

Small Low	Small High	Large Low	Large High

Summarizing conditions

- for easier analysis we can group some conditions

1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH

Small Low	Small High	Large Low	Large High

Summarizing conditions

- for easier analysis we can group some conditions

1LL	1L	1H	1HH	2LL	2L	2H	2HH	8LL	8L	8H	8HH	9LL	9L	9H	9HH

Small Low	Small High	Large Low	Large High

Define groups

- new variable for 4 *conditions*

Define groups

- new variable for 4 *conditions*

```
% Variable
compatibility = {'compatible', 'incompatible'};
task = {'number', 'pitch', 'parity'};
conditions = {'small_low', 'small_high', 'large_low', 'large_high'};
```

Define groups

- new variable for 4 *conditions*

```
% Variable
compatibility = {'compatible', 'incompatible'};
task = {'number', 'pitch', 'parity'};
conditions = {'small_low', 'small_high', 'large_low', 'large_high'};
```

- give positions which belong to each condition

Define groups

- new variable for 4 *conditions*

```
% Variable
compatibility = {'compatible', 'incompatible'};
task = {'number', 'pitch', 'parity'};
conditions = {'small_low', 'small_high', 'large_low', 'large_high'};
positions = [1,2,5,6; 3,4,7,8; 9,10,13,14; 11,12,15,16];
```

- give positions which belong to each condition

Define groups

- enhance struct *S* for each hand into each of the conditions and calculate the *mean* of the four conditions $\Rightarrow$ check *help mean* for calculating mean for the correct row or column

Define groups

- enhance struct *S* for each hand into each of the conditions and calculate the *mean* of the four conditions ⇒ check *help mean* for calculating mean for the correct row or column

```
for i = 1:length(conditions)

    S.(task{t}).left.(conditions{i}) = mean(S.(task{t}).left.all(:,position(i,:)),2);

end;
```

Define groups

- enhance struct *S* for each hand into each of the conditions and calculate the *mean* of the four conditions ⇒ check *help mean* for calculating mean for the correct row or column

```
for i = 1:length(conditions)

    S.(task{t}).left.(conditions{i}) = mean(S.(task{t}).left.all(:,position(i,:)),2);

end;
```

- do the same for the right hand

Define groups

- enhance struct *S* for each hand into each of the conditions and calculate the *mean* of the four conditions ⇒ check *help mean* for calculating mean for the correct row or column

```
for i = 1:length(conditions)

    S.(task{t}).left.(conditions{i}) = mean(S.(task{t}).left.all(:,position(i,:)),2);
    S.(task{t}).right.(conditions{i}) = mean(S.(task{t}).right.all(:,position(i,:)),2);

end;
```

- do the same for the right hand

Define groups

- enhance struct *S* for each hand into each of the conditions and calculate the *mean* of the four conditions ⇒ check *help mean* for calculating mean for the correct row or column

```
for i = 1:length(conditions)

    S.(task{t}).left.(conditions{i}) = mean(S.(task{t}).left.all(:,position(i,:)),2);
    S.(task{t}).right.(conditions{i}) = mean(S.(task{t}).right.all(:,position(i,:)),2);

end;
```

- do the same for the right hand
- and inspect S

Summarize both hands according to SNARC and SPARC

- we have to sort hands according to four conditions: *SNcSPc, SNcSPi, SNiSPc, SNiSPi*

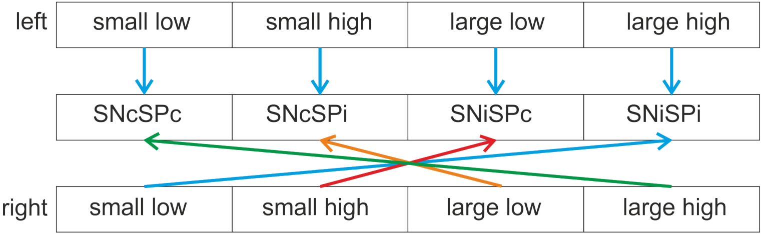
Summarize both hands according to SNARC and SPARC

- we have to sort hands according to four conditions: *SNcSPc, SNcSPi, SNiSPc, SNiSPi*

left	small low	small high	large low	large high
	SNcSPc	SNcSPi	SNiSPc	SNiSPi

Summarize both hands according to SNARC and SPARC

- we have to sort hands according to four conditions: *SNcSPc, SNcSPi, SNiSPc, SNiSPi*



SNARC and SPARC

- define variable *SNSP*

SNARC and SPARC

- define variable *SNSP*

```
% Variable
compatibility = {'compatible', 'incompatible'};
task = {'number', 'pitch', 'parity'};
conditions = {'small_low', 'small_high', 'large_low', 'large_high'};
positions = [1,2,5,6; 3,4,7,8; 9,10,13,14; 11,12,15,16];
SNSP = {'SNcSPc', 'SNcSPi', 'SNiSPc', 'SNiSPi'};
```

SNARC and SPARC

- define variable *SNSP*

```
% Variable
compatibility = {'compatible', 'incompatible'};
task = {'number', 'pitch', 'parity'};
conditions = {'small_low', 'small_high', 'large_low', 'large_high'};
positions = [1,2,5,6; 3,4,7,8; 9,10,13,14; 11,12,15,16];
SNSP = {'SNcSPc', 'SNcSPi', 'SNiSPc', 'SNiSPi'};
```

- build *for* loop

SNARC and SPARC

- define variable *SNSP*

```
for s = 1:length(SNSP)

end;
```

- build *for* loop

SNARC and SPARC

- define variable *SNSP*

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});

end;
```

- build *for* loop

SNARC and SPARC

- define variable *SNSP*

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});

end;
```

- build *for* loop

SNARC and SPARC

- define variable *SNSP*

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});

end;
```

- build *for* loop
- 

SNARC and SPARC

- Summarize hands by calculating *mean* $\Rightarrow$ you will need to use [...] and check the results!

SNARC and SPARC

- Summarize hands by calculating *mean* $\Rightarrow$ you will need to use [...] and check the results!

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})]);
end;
```

SNARC and SPARC

- Summarize hands by calculating *mean* ⇒ you will need to use [...] and check the results!

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})]);
end;
```

- be careful to calculate the *mean* for each individual participant, so check help mean and see how to enter

SNARC and SPARC

- Summarize hands by calculating *mean* ⇒ you will need to use [...] and check the results!

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})],2);
end;
```

- be careful to calculate the *mean* for each individual participant, so check help mean and see how to enter

SNARC and SPARC

- Summarize hands by calculating *mean* $\Rightarrow$ you will need to use [...] and check the results!

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})],2);
end;
```

- be careful to calculate the *mean* for each individual participant, so check help mean and see how to enter
-

SNARC and SPARC

- for later plotting it is nice to also have the total *mean* in each of the *SNARC-SPARC* conditions

SNARC and SPARC

- for later plotting it is nice to also have the total *mean* in each of the *SNARC-SPARC* conditions

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})],2);
    S.(task{t}).(SNSP{s}).mean = mean(S.(task{t}).(SNSP{s}).both);
end;
```

SNARC and SPARC

- for later plotting it is nice to also have the total *mean* in each of the *SNARC-SPARC* conditions

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})],2);
    S.(task{t}).(SNSP{s}).mean = mean(S.(task{t}).(SNSP{s}).both);
end;
```

- as well as the *standard deviation (std)*

SNARC and SPARC

- for later plotting it is nice to also have the total *mean* in each of the *SNARC-SPARC* conditions

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})],2);
    S.(task{t}).(SNSP{s}).mean = mean(S.(task{t}).(SNSP{s}).both);
    S.(task{t}).(SNSP{s}).std = std(S.(task{t}).(SNSP{s}).both);

end;
```

- as well as the *standard deviation (std)*

SNARC and SPARC

- for later plotting it is nice to also have the total *mean* in each of the *SNARC-SPARC* conditions

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})],2);
    S.(task{t}).(SNSP{s}).mean = mean(S.(task{t}).(SNSP{s}).both);
    S.(task{t}).(SNSP{s}).std = std(S.(task{t}).(SNSP{s}).both);

end;
```

- as well as the *standard deviation (std)*
- you may wish to have *standard error* instead of *standard deviation* by dividing std by the square root of the number of the participants ⇒ try yourself by asking google for the *square root* and how to enter it into matlab

SNARC and SPARC

- for later plotting it is nice to also have the total *mean* in each of the *SNARC-SPARC* conditions

```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})],2);
    S.(task{t}).(SNSP{s}).mean = mean(S.(task{t}).(SNSP{s}).both);
    S.(task{t}).(SNSP{s}).std = std(S.(task{t}).(SNSP{s}).both)/sqrt(length(subjects));

end;
```

- as well as the *standard deviation (std)*
- you may wish to have *standard error* instead of *standard deviation* by dividing std by the square root of the number of the participants ⇒ try yourself by asking google for the *square root* and how to enter it into matlab

SNARC and SPARC

- for later plotting it is nice to also have the total *mean* in each of the *SNARC-SPARC* conditions

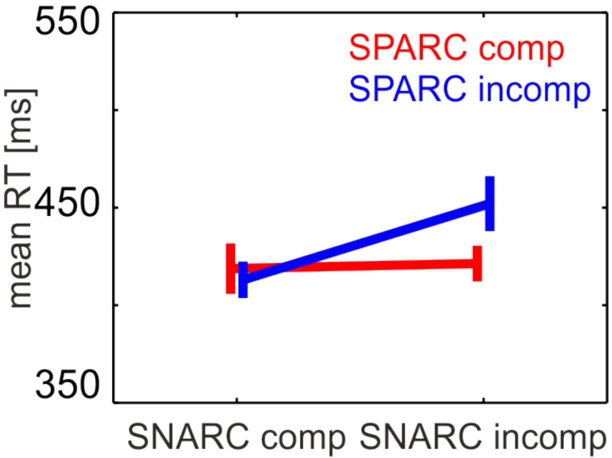
```
for s = 1:length(SNSP)

    S.(task{t}).(SNSP{s}).left = S.(task{t}).left.(conditions{s});
    S.(task{t}).(SNSP{s}).right = S.(task{t}).right.(conditions{5-s});
    S.(task{t}).(SNSP{s}).both = mean([S.(task{t}).left.(conditions{s}) S.(task{t}).right.(conditions{s})],2);
    S.(task{t}).(SNSP{s}).mean = mean(S.(task{t}).(SNSP{s}).both);
    S.(task{t}).(SNSP{s}).std = std(S.(task{t}).(SNSP{s}).both)/sqrt(length(subjects));

end;
```

- as well as the *standard deviation (std)*
- you may wish to have *standard error* instead of *standard deviation* by dividing std by the square root of the number of the participants ⇒ try yourself by asking google for the *square root* and how to enter it into matlab
-

Example


550
450
350
mean RT [ms]
SPARC comp
SPARC incomp
SNARC comp
SNARC incomp

Plot

- To open a new figure we need *figure*

Plot

- To open a new figure we need *figure*

```
% Plot the results
figure;
```

Plot

- To open a new figure we need *figure*
- On the x-axis we need to points, so we initialize the x-axis with $[1, 2]$

```
% Plot the results
figure;
```

Plot

- To open a new figure we need *figure*
- On the x-axis we need to points, so we initialize the x-axis with $[1, 2]$

```
% Plot the results
h = figure;
plot([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean]);
```

- We first want to plot the SPARC compatible red line ⇒ therefore we need the mean of the group for SNcSPc positioned at 1 on the x-axis and SNiSPc positioned at 2 on the x-axis ⇒ we already calculated those means and stored them in the structure *S* ⇒ since we are still in the *task* loop and can therefore use *(task{t})* in the structure *S*

Plot

- To open a new figure we need *figure*
- On the x-axis we need to points, so we initialize the x-axis with $[1, 2]$

```
% Plot the results
h = figure;
plot([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],'r-');
```

- We first want to plot the SPARC compatible red line ⇒ therefore we need the mean of the group for SNcSPc positioned at 1 on the x-axis and SNiSPc positioned at 2 on the x-axis ⇒ we already calculated those means and stored them in the structure *S* ⇒ since we are still in the *task* loop and can therefore use $(task\{t\})$ in the structure *S*
- F9 ⇒ so far our plot is blue instead of red, we can change all properties of the figure, see *help plot* ⇒ color, linewidth, linetype,...

Plot

- To open a new figure we need *figure*
- On the x-axis we need to points, so we initialize the x-axis with $[1, 2]$

```
% Plot the results
h = figure;
plot([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],'r-','Linewidth', 2);
```

- We first want to plot the SPARC compatible red line ⇒ therefore we need the mean of the group for SNcSPc positioned at 1 on the x-axis and SNiSPc positioned at 2 on the x-axis ⇒ we already calculated those means and stored them in the structure *S* ⇒ since we are still in the *task* loop and can therefore use $(task\{t\})$ in the structure *S*
- F9 ⇒ so far our plot is blue instead of red, we can change all properties of the figure, see *help plot* ⇒ color, linewidth, linetype,...

Plot

- We now need the second line $\Rightarrow$ to plot them into the same *figure* we need the command *hold on*, otherwise it will just overwrite the first figure

Plot

- We now need the second line ⇒ to plot them into the same *figure* we need the command *hold on*, otherwise it will just overwrite the first figure

```
% Plot the results
h = figure;
plot([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],'r-','Linewidth', 2);
hold on;
```

Plot

- We now need the second line ⇒ to plot them into the same *figure* we need the command *hold on*, otherwise it will just overwrite the first figure

```
% Plot the results
h = figure;
plot([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],'r-','Linewidth', 2);
hold on;
```

- insert second line for SPi

Plot

- We now need the second line ⇒ to plot them into the same *figure* we need the command *hold on*, otherwise it will just overwrite the first figure

```
% Plot the results
h = figure;
plot([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],'r-','Linewidth', 2);
hold on;
plot([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],'b-','Linewidth', 2);
```

- insert second line for SPi

Plot

- We now need the second line ⇒ to plot them into the same *figure* we need the command *hold on*, otherwise it will just overwrite the first figure

```
% Plot the results
h = figure;
plot([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],'r-','Linewidth', 2);
hold on;
plot([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],'b-','Linewidth', 2);
```

- insert second line for SPi
- F9 ⇒ plot might not be the right *function*, because we are still missing the errorbars in the figure ⇒ check how *errorbar* works

Plot

- We now need the second line ⇒ to plot them into the same *figure* we need the command *hold on*, otherwise it will just overwrite the first figure

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
                [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
                [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
```

- insert second line for SPi
- F9 ⇒ plot might not be the right *function*, because we are still missing the errorbars in the figure ⇒ check how *errorbar* works

Label axis

- We finally need to insert axis names $\Rightarrow$ *help plot*

Label axis

- We finally need to insert axis names ⇒ *help plot*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
```

Label axis

- We finally need to insert axis names ⇒ *help plot*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
```

Label axis

- We finally need to insert axis names ⇒ *help plot*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
```

- We also need a *title* ⇒ *help plot* (be careful, title should be named that it is specific to each task)

Label axis

- We finally need to insert axis names ⇒ *help plot*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
```

- We also need a *title* ⇒ *help plot* (be careful, title should be named that it is specific to each task)

Label axis

- We finally need to insert axis names ⇒ *help plot*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
```

- We also need a *title* ⇒ *help plot* (be careful, title should be named that it is specific to each task)
-

Specify axis

- It seems that we have different y-axis in the different tasks, for better comparison between task choose the same y-axis on all figures $\Rightarrow$ ask *google*

Specify axis

- It seems that we have different y-axis in the different tasks, for better comparison between task choose the same y-axis on all figures ⇒ ask *google*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
ylim([500 800]);
```

Specify axis

- It seems that we have different y-axis in the different tasks, for better comparison between task choose the same y-axis on all figures ⇒ ask *google*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
                [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
                [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
ylim([500 800]);
```

- we need to define which color indicates which condition ⇒ *help legend*

Specify axis

- It seems that we have different y-axis in the different tasks, for better comparison between task choose the same y-axis on all figures $\Rightarrow$ ask *google*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
                [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
                [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
ylim([500 800]);
legend('SPc', 'SPi', 'Location', 'SouthEast');
```

- we need to define which color indicates which condition $\Rightarrow$ *help legend*

Specify axis

- It seems that we have different y-axis in the different tasks, for better comparison between task choose the same y-axis on all figures ⇒ ask *google*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
ylim([500 800]);
legend('SPc', 'SPi', 'Location', 'SouthEast');
```

- we need to define which color indicates which condition ⇒ *help legend*
-

Rename x-axis and save plot and results

- instead of having *1* and *2* on x-axis we might want to have *SNc* and *SNi*

Rename x-axis and save plot and results

- instead of having *1* and *2* on x-axis we might want to have *SNc* and *SNi*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
ylim([500 800]);
legend('SPc', 'SPi', 'Location', 'SouthEast');
set(gca, 'xtick', [1,2], 'xTickLabel', 'SNc|SNi');
```

Rename x-axis and save plot and results

- instead of having *1* and *2* on x-axis we might want to have *SNc* and *SNi*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
ylim([500 800]);
legend('SPc', 'SPi', 'Location', 'SouthEast');
set(gca, 'xtick', [1,2], 'xTickLabel', 'SNc|SNi');
```

- we also want to *save* the figure

Rename x-axis and save plot and results

- instead of having *1* and *2* on x-axis we might want to have *SNc* and *SNi*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
                [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
                [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
ylim([500 800]);
legend('SPc', 'SPi', 'Location', 'SouthEast');
set(gca, 'xtick', [1,2], 'xTickLabel', 'SNc|SNi');
saveas(h, [datapath task{t} '.jpg']);
```

- we also want to *save* the figure

Rename x-axis and save plot and results

- instead of having *1* and *2* on x-axis we might want to have *SNc* and *SNi*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
               [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
               [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
ylim([500 800]);
legend('SPc', 'SPi', 'Location', 'SouthEast');
set(gca, 'xtick', [1,2], 'xTickLabel', 'SNc|SNi');
saveas(h, [datapath task{t} '.jpg']);
```

- we also want to *save* the figure
-

Rename x-axis and save plot and results

- instead of having *1* and *2* on x-axis we might want to have *SNc* and *SNi*

```
% Plot the results
h = figure;
errorbar([1,2],[S.(task{t}).SNcSPc.mean, S.(task{t}).SNiSPc.mean],...
                [S.(task{t}).SNcSPc.std, S.(task{t}).SNiSPc.std],'r-','Linewidth', 2);
hold on;
errorbar([1,2],[S.(task{t}).SNcSPi.mean, S.(task{t}).SNiSPi.mean],...
                [S.(task{t}).SNcSPi.std, S.(task{t}).SNiSPi.std],'b-','Linewidth', 2);
xlabel('SNARC compatibility');
ylabel('mean RT [ms]');
title(['SNARC SPARC ' task{t}]);
ylim([500 800]);
legend('SPc', 'SPi', 'Location', 'SouthEast');
set(gca, 'xtick', [1,2], 'xTickLabel', 'SNc|SNi');
saveas(h, [datapath task{t} '.jpg']);
```

- we also want to *save* the figure
-
- We also may want to *save* all the results you have or especially only one variable (e.g. S, where all your information is stored)

Rename x-axis and save plot and results

- instead of having *1* and *2* on x-axis we might want to have *SNc* and *SNi*

```
end;

save([datapath filesep 'S.mat'], 'S')
```

- we also want to *save* the figure
- 
- We also may want to *save* all the results you have or especially only one variable (e.g. S, where all your information is stored)

Make tables

- now we have a figure of our data but we also have to do some *statistics*

Make tables

- now we have a figure of our data but we also have to do some *statistics*
- we have to prepare the data for statistical analysis e.g. with *SPSS*, and organize the data into a *table*

Make tables

- now we have a figure of our data but we also have to do some *statistics*
- we have to prepare the data for statistical analysis e.g. with *SPSS*, and organize the data into a *table*

```
Table_SPSS = [S.number.SNcSPc.both S.number.SNcSPi.both S.number.SNiSPc.both S.number.SNiSPi.both...
              S.pitch.SNcSPc.both S.pitch.SNcSPi.both S.pitch.SNiSPc.both S.pitch.SNiSPi.both...
              S.parity.SNcSPc.both S.parity.SNcSPi.both S.parity.SNiSPc.both S.parity.SNiSPi.both];
```

Make tables

- now we have a figure of our data but we also have to do some *statistics*
- we have to prepare the data for statistical analysis e.g. with *SPSS*, and organize the data into a *table*

```
Table_SPSS = [S.number.SNcSPc.both S.number.SNcSPi.both S.number.SNiSPc.both S.number.SNiSPi.both...
              S.pitch.SNcSPc.both S.pitch.SNcSPi.both S.pitch.SNiSPc.both S.pitch.SNiSPi.both...
              S.parity.SNcSPc.both S.parity.SNcSPi.both S.parity.SNiSPc.both S.parity.SNiSPi.both];
```

The end...