ROC Toolbox Manual

Version 1.1.1

Created by Joshua D. Koen, Fred S. Barrett, Iain M. Harlow, and Andrew P. Yonelinas

© The Regents of the University of California, Davis Campus, 2014

Table of Contents

PURPOSE & DISCLAIMER 1

LICENSE 2

INSTALLATION 3

PRIMARY FUNCTIONS & OPTIMIZATION ROUTINE 4

FORMATTING THE DATA FOR ANALYSIS 5

TUTORIAL 1: SINGLE-CONDITION DESIGNS & BASIC USE 9

TUTORIAL 2: MULTIPLE-CONDITION DESIGNS & ADVANCED USE 18

TUTORIAL 3: IMPORTING DATA AND EXTRACTING GROUP DATA 27

APPENDIX A: SIGNAL DETECTION ANALYSIS OF YES/NO DATA 32

APPENDIX B: THE MODELS AND THEIR PARAMETERS 33

APPENDIX C: ADDING TO THE TOOLBOX 39

REFERENCES 41

Purpose & Disclaimer

This toolbox was developed to provide an open source platform for the analysis of receiver operating characteristic (ROC) data. This toolbox is geared towards cognitive psychology researchers who examine ROCs based on ratings (i.e., confidence) data. A primary purpose of this toolbox is to provide a flexible framework to define different models to fit to data from a wide-range of experimental designs.

If you find the ROC toolbox useful for analyzing your data, please cite this manual in your paper.

This code is provided as-is. Although we have tried our best to ensure the code produces accurate results, we cannot and do not guarantee the accuracy of the data obtained from this toolbox.

If you have any questions, find any errors in the code, or would like contribute your own code to the ROC toolbox, please contact koen [dot] joshua [at] gmail [dot] com.

This manual assumes the user has knowledge of signal-detection theory and Matlab programming. If you need to acquire background knowledge on signal-detection theory and ROC analysis, the following sources might be helpful:

Macmillan, N. A., & Creelman, C. D. (2005). Detection theory: A user's guide (2nd Edition). Mahwah, NJ: Lawrence Erlbaum Associates.

Yonelinas, A. P., & Parks, C. M. (2007). Receiver operating characteristics (ROCs) in recognition memory: A review. *Psychological Bulletin*, *133*(5), 800–832.

If you need to develop some experience with programming in Matlab, there are many free online tutorials.

License

The ROC Toolbox is the proprietary property of The Regents of the University of California ("The Regents.")

Copyright © 2014 The Regents of the University of California, Davis campus. All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted by nonprofit, research institutions for research use only, provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name of The Regents may not be used to endorse or promote products derived from this software without specific prior written permission.

The end-user understands that the program was developed for research purposes and is advised not to rely exclusively on the program for any reason.

THE SOFTWARE PROVIDED IS ON AN "AS IS" BASIS, AND THE REGENTS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS. THE REGENTS SPECIFICALLY DISCLAIM ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE REGENTS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, EXEMPLARY OR CONSEQUENTIAL DAMAGES, INCLUDING BUT NOT LIMITED TO PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, LOSS OF USE, DATA OR PROFITS, OR BUSINESS INTERRUPTION, HOWEVER CAUSED AND UNDER ANY THEORY OF LIABILITY WHETHER IN CONTRACT, STRICT LIABILITY OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

If you do not agree to these terms, do not download or use the software. This license may be modified only in writing signed by authorized signatory of both parties.

For commercial license information please contact copyright@ucdavis.edu.

Installation

Prerequisites & Included Redistributions:
This toolbox has been developed in Matlab version 2011a and should be compatible with Matlab version 7.10.0.499 (R2009b) and up. You must have the Statistics and Optimization Toolboxes installed with Matlab for the ROC toolbox to function properly.

The ROC Toolbox includes two code redistributions under the GNU license. The redistributions included in this toolbox are from functions available on the Mathworks File Exchange: (1) loadtxt and sortcell, which can be found on the File Exchange. These functions are necessary for using the `roc_import_data` function.

Getting the toolbox and adding it to the Matlab path
1) Download a copy of the latest release of ROC Toolbox from:
https://github.com/jdkoen/roc_toolbox/releases

2) Unzip the .zip file to the target directory

3) Use Matlab's Set Path interface and add the unzipped directory using the "Add with subfolders" option. Make sure to save the updated path.

Alternatively, you can add the following lines to your 'startup.m' file:

```
addpath('/path/to/roc_toolbox');
roc_startup;
```

These lines add the path to the ROC toolbox distribution to your Matlab environment, and runs a function named 'roc_startup.m' that will add the necessary directories to your path

You can check to see if the ROC toolbox is in your path by running the following command

```
which roc_solver
```

To check the version of the ROC Toolbox you are using, run the following command

```
roc_version;
```

Primary Functions & Optimization Routine

Primary Functions

There are four primary functions in the ROC Toolbox that you will call to analyze your data.

Function	Brief Description
`roc_import_data`	This function can be used to import data stored in a comma-separated (.csv) or tab-delimited (.txt) file. The data is required to be in long format, which is described in more length in the **Formatting the Data for Analysis** section.
`gen_pars`	This function creates the matrices for the starting (x0), lower bounds (LB), and upper bounds (UB) for the parameters of the desired model.
`roc_solver`	This function optimizes the parameters of the desired model to best fit the observed data (with MLE or least-squares optimization) and returns a data structure containing the results from the optimization routine. The optimization routine is briefly described below.
`gen_sim_data`	This function is used to generate bootstrapped data samples from a matrix of parameter values.

Optimization Routine

The ROC Toolbox uses `fmincon` to optimize the parameters to best fit the observed data. The `roc_solver` function sets default options for `fmincon`. However, you are able to control the options for `fmincon` by passing the options structure to the `roc_solver` (see help `roc_solver` for more information). The only option that cannot be controlled at this moment is the search algorithm used by `fmincon`, which is set to `'interior-point'`.

Formatting the Data for Analysis

Overview

The data necessary to run the ROC toolbox are the response frequencies in each rating (i.e., confidence) bin for target and lure items. These data should be stored in matrices. Throughout this manual, we will refer to frequency matrix for targets and lures `targf` and `luref`, respectively. These two matrices provide the primary input to the majority of the functions in this toolbox, although you will mostly use these two variables in conjunction with the `roc_solver` function.

The toolbox requires the data stored in the `targf` and `luref` variables to be formatted such that different conditions occur along the row dimension whereas the rating bins occur along the column dimension. Below is a visual depiction of the required format for the `targf` and `luref` variables. Note that the labels in gray are for illustrative purposes only.

	n	***n*-1**	***n*-2**	**…**	**1**
Condition 1	$Freq_{1,n}$	$Freq_{1,n-1}$	$Freq_{1,n-2}$	…	$Freq_{1,1}$
Condition 2	$Freq_{2,n}$	$Freq_{2,n-2}$	$Freq_{2,n-1}$	…	$Freq_{2,1}$
…	…	…	…	…	…
Condition *i*	$Freq_{i,n}$	$Freq_{i,n-1}$	$Freq_{i,n-1}$	…	$Freq_{i,1}$

There are 3 important things to keep in mind if you are writing your own code to create these matrices:

1. The order of the conditions is arbitrary. You can order them how you wish. However, the only constraint is that Condition *i* for `targf` must be in the same row as Condition *i* in `luref`. The toolbox will not crash if this constraint is not met, but your results will be incorrect when comparing parameter estimates across conditions. There is one situation in which the order of the conditions are important, and that is when you use `'ignoreConds'` optional input to `roc_solver`.

2. The rating bins need to be sorted in descending order such that the 1st column contains the frequencies for the rating bin corresponding to the highest confidence target judgment and that the last column contains the response frequencies for the rating bin corresponding to the highest confidence lure judgment. As with the conditions, an error will not be returned if you incorrectly order the rating bins, but the parameter estimates will be inaccurate.

3. The `targf` and `luref` matrices must be the same size, or an error will be returned. This might seem odd given that some experimental designs might have more target conditions than lure conditions (e.g., two classes of target items and one class of lure

items). In these instances, the lure frequencies should be repeated such that `luref` has the same number of rows as `targf`. This will be covered in more detail in Tutorial 2.

An Example

To provide a more concrete example, say you have data from a recognition memory experiment where participants studied two lists of words. Each list was followed by a recognition test where they were asked to discriminate between studied words (targets) and new words (lures). Attention was divided on one test with a secondary task, but subjects devoted their full attention to the other test. Participants made their judgments using a 6-point confidence scale with the following bins: *6-Sure Target, 5-Maybe Target, 4-Guess Target, 3-Guess Lure, 2-Maybe Lure, 1-Sure Lure*. Here is what the `targf` matrix might look like:

	6	***5***	***4***	***3***	**2**	**1**
Full Attention	*#*	*#*	*#*	*#*	*#*	*#*
Divided Attention	*#*	*#*	*#*	*#*	*#*	*#*

Note that the `luref` matrix would look identical, but contain the response frequencies for the lure items.

Importing Your Data with `roc_import_data`

If you want to import your data from outside Matlab with minimal code, you can use the `roc_import_data` function. The data has to be in a long format such that one row of the file corresponds to one rating frequency in the experimental design. The file formats for the data to be imported can be comma-separated (.csv) or tab-delimited text files (.txt).

Here is an example of the *tutorial3_data.csv* file that is included in the *examples* folder.

Subject ID (String/Numeric)	Group ID (String/Numeric)	Condition Label (String)	Item Type (String: target/lure)	Rating Bin (Numeric)	Response Frequencies (Numeric)
101	group1	condition1	target	10	96
101	group1	condition1	target	9	15
101	group1	condition1	target	8	16
101	group1	condition1	target	7	17
101	group1	condition1	target	6	17
101	group1	condition1	target	5	21
101	group1	condition1	target	4	5
101	group1	condition1	target	3	9
101	group1	condition1	target	2	2

101	group1	condition1	target	1	2
101	group1	condition1	lure	10	9
101	group1	condition1	lure	9	13
101	group1	condition1	lure	8	23
101	group1	condition1	lure	7	29
101	group1	condition1	lure	6	22
101	group1	condition1	lure	5	26
101	group1	condition1	lure	4	29
101	group1	condition1	lure	3	25
101	group1	condition1	lure	2	9
101	group1	condition1	lure	1	15
101	group1	condition2	target	10	107
101	group1	condition2	target	9	23
101	group1	condition2	target	8	24
101	group1	condition2	target	7	18
101	group1	condition2	target	6	8
101	group1	condition2	target	5	8
101	group1	condition2	target	4	8
101	group1	condition2	target	3	1
101	group1	condition2	target	2	1
101	group1	condition2	target	1	2
101	group1	condition2	lure	10	11
101	group1	condition2	lure	9	13
101	group1	condition2	lure	8	16
101	group1	condition2	lure	7	34
…	…	…	…	…	…
102	group2	condition1	target	10	105
102	group2	condition1	target	9	11
102	group2	condition1	target	8	19
102	group2	condition1	target	7	17
102	group2	condition1	target	6	15
102	group2	condition1	target	5	16
102	group2	condition1	target	4	13
102	group2	condition1	target	3	1
102	group2	condition1	target	2	1
102	group2	condition1	target	1	2
102	group2	condition1	lure	10	13
102	group2	condition1	lure	9	9
102	group2	condition1	lure	8	17
102	group2	condition1	lure	7	30
102	group2	condition1	lure	6	25

…	…	…	…	…	…
102	group2	Condition2	lure	1	17

Here are some important things to keep in mind when organizing your data to the above format:

1. Do not include the column headers in the file.
2. The item type condition must have target and lure labels. Other types of labels, such as old or new, will cause the function to crash.
3. The order of the rows does not matter as the `roc_import_data` function sorts the data accordingly.
4. All rows and columns must have a value, except the group ID. The group ID column can be left empty as it is only used for between subject conditions. However, the column must be present otherwise the program will return an error.
5. The ratings bin must be ordered such that the frequencies in the 1st column of the matrix represents the highest confidence that the item came from the target distribution and the frequencies in the last column of the matrix represents the highest confidence that the items came from the lure distribution. The code will still run if the rating bins are incorrectly ordered, but the results will be incorrect.
6. Similar to the 3rd issue described previously about formatting your own data, if you have more target item conditions than lure conditions, you will need to repeat the frequencies for the lures with the appropriate condition labels.

`roc_import_data` returns a cell array of structures containing the imported data. Each cell contains an individual subject's data in the following fields:

Field	**Description**
conditions	This is a cell array of strings containing condition names. The first element in this cell array corresponds to first row in the `targf` and `luref` fields.
subID	This is the subject ID string for the current subject.
groupID	This is the group ID for the subject.
targf	This is a matrix that contains the response frequencies for the target items.
luref	This is a matrix that contains the response frequencies for the lure items.

Tutorial 1: Single-Condition Designs & Basic Use

Objectives:

- Use the `gen_pars` function to define the model to be fit to the data.
- Use the `roc_solver` function to fit the model to the data.
- Use the `roc_solver` function options to add additional information/models to the same data structure.

Overview

This tutorial uses data stored in *tutorial1_data.mat* that is located in the *examples* directory of the toolbox. This set of data has a single condition (i.e., one class of target items and one class of lure items) with 1000 target and 1000 lure trials and 10 rating bins. The frequencies were randomly sampled from the unequal-variance signal-detection (UVSD) model with *d'*= 1, Vo = 1.4, and nine criterion values that were equally spaced from -1.5 to +1.5. The code used in this tutorial is saved in the *tutorial1_script.m* file in the *examples* directory.

Load the data and define some information about the design

The .mat file contains many variables that specify how the data were generated, but the only variables that are needed for this tutorial are the `targf` and `luref` matrices. Type the following command to load the data necessary data:

```
load('tutorial1_data.mat','targf','luref');
```

Next, it is useful to define some variables that will be used at a point to define some aspects of the model you want to fit to your data. Specifically, we will define the number of rating bins (`nBins`), number of conditions (`nConds`), and the fit statistic (`fitStat`) that will be optimized to find the best fitting parameters.

```
nBins = size(targf,2);
nConds = size(targf,1);
fitStat = '-LL';
```

The `nBins` and `nConds` variables are helpful when defining the to-be-estimated parameters of a model with the `gen_pars` function. The value of `fitStat` indicates that we will use Maximum Likelihood Estimation (MLE) to find the best fitting parameters. The MLE routine in the ROC Toolbox minimizes the negative log-likelihood value. The other minimization routine uses a least-squares cost function whereby the squared difference between the observed and predicted probabilities in each rating bin are minimized.

Defining the Model

For this example, we will fit the UVSD model to the data. First, we need to define the parameters of the model, which we will do using the `gen_pars` function. The model we want to define will have 11 parameters: *d'*, V_o, and 9 criterion parameters. The `gen_pars` function automatically adds the appropriate number of criterion parameters based on the number of ratings bins (i.e., `nBins - 1`). Thus, the only parameters that need to be specified are the model specific parameters. Let's store the model we want to fit and the parameters of that model in two variables:

```
model = 'uvsd';
parNames = {'Dprime' 'Vo'};
```

The `parNames` variable is a cell array of strings with the parameter names to be estimated. Second, we need to create the matrix of parameter starting values (`x0`), lower bounds (`LB`) and upper bounds (`UB`) using the `gen_pars` function. Type in the following code:

```
[x0,LB,UB] = gen_pars(model,nBins,nConds,parNames);
```

All three of these matrices will be passed to the `roc_solver` function to fit the model to the data, which define the structure of the model to be fit to the data. The `x0` matrix specifies the starting value for each parameter in the optimization routine. The `LB` and `UB` matrices define the range of possible values that each parameter can take. The `LB` and `UB` matrices are important in determining the number of parameters in a model. As will be described later, having parameters where `LB(i) == UB(i)` tells the `roc_solver` that the parameter is not estimated and forces the corresponding parameter to equal the value specified in `LB` and `UB`. Note that the available parameters, the organization of the parameter matrix, and the default values of `x0`, `LB`, and `UB` for each model are covered in Appendix B.

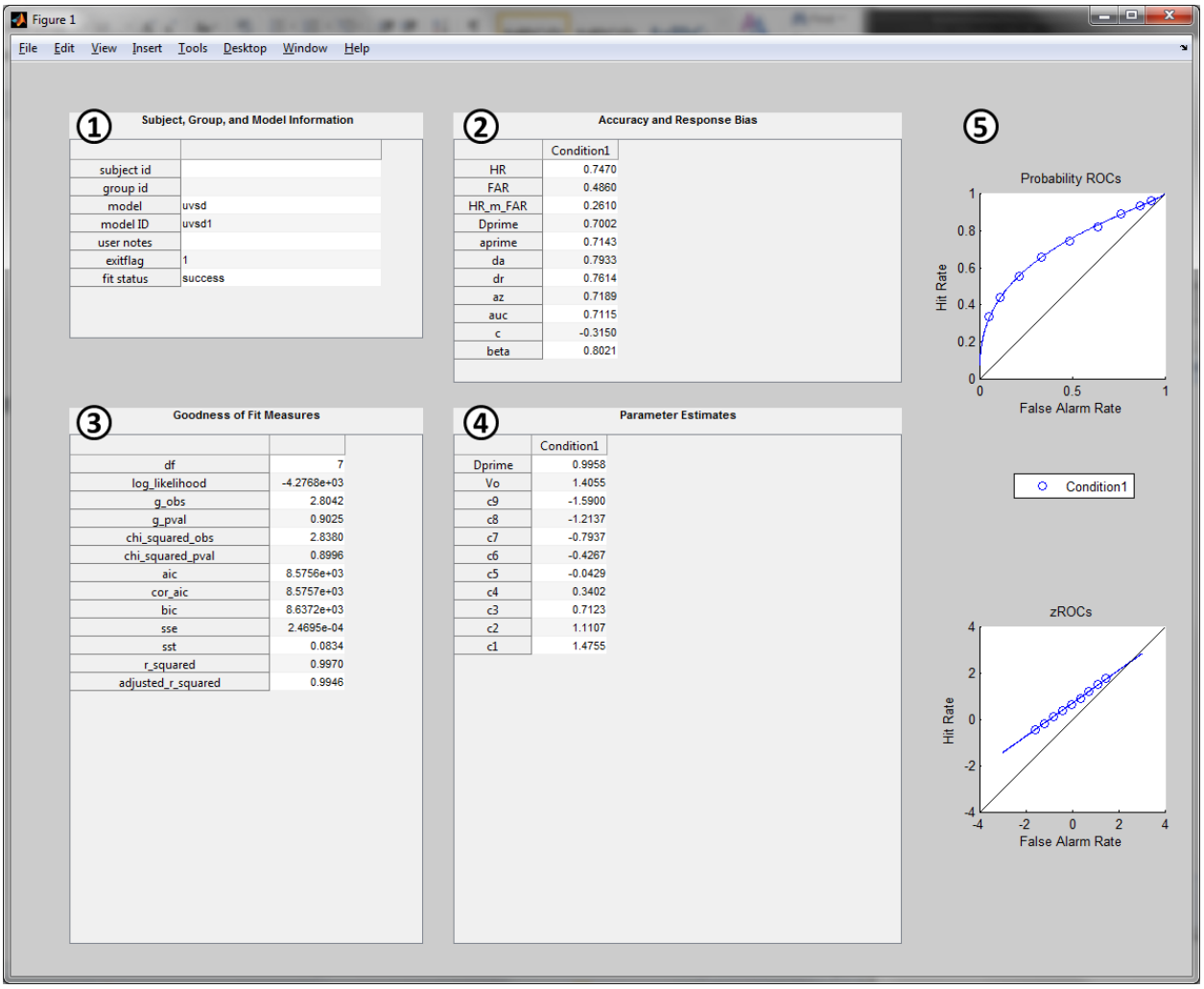
Fitting the model to the data

The last step is to optimize the parameters of the model to best fit the observed data. This is accomplished with the `roc_solver` function. Type in the following code:

```
rocData = roc_solver(targf,luref,model,fitStat,x0,LB,UB);
```

Note that the 6 inputs shown above are required anytime you want to run the `roc_solver`. However, there are many other optional inputs for the `roc_solver`, some of which are explored in more detail below. The `roc_solver` runs the optimization routine and returns the results to a structured variable in `rocData`. After the function is finished, a figure containing summary information from the optimization routine will be printed to the screen. Note that the

`roc_solver` function is still running in the background, and will not exit until the figure is closed (this can be controlled, and is considered in the next part of this tutorial).



Here is a brief description of the information in the figure, which was generated by the `plot_roc_summary` function (called in `roc_solver`):

1. This box contains the information about the subject and model. Note that subject id, group id, and user notes fields are blank. This is because they are optional inputs to `roc_solver`, which we did not supply. The model field is the model that was fit to the data. The model ID field is different from the model field in that it is a user defined label for the model, and thus this is an optional input to `roc_solver`. However, if this is not given, the field is automatically populated to provide distinct IDs for different iterations of the same model. The important thing to pay attention to is the `exitflag` and fit status values. Ideally, these should say 1 and success, respectively, but this will not always be the case because the search algorithm could exit for other reasons (e.g., the

change in the parameters, not the likelihood, becomes too small) or fail completely (e.g., the max number of iterations is reached).

2. This table contains numerous signal-detection measures of accuracy and response bias for each condition in the experiment.
3. This table contains the goodness of fit measures for the optimized parameters.
4. This table contains the optimized parameter estimates for each condition. It is important to point out that the criterion location (i.e., c#) parameters are not estimated in absolute units, but instead measured relative to the previous criterion location. This was done to avoid issues when estimating the criterion parameters (see Appendix B for more details). Also, you will probably notice that there is a bunch of not-a-number (NaN) in parentheses next to each parameter estimates (although this is currently not shown in the figure above). This because the call to the `roc_solver` did not include the `bootIter` property-value pair input. This optional input, shown below, will use a non-parametric bootstrap routine to estimate the standard errors of the parameter estimates. If the `bootIter` input is given to `roc_solver`, then the NaN's will be replaced with estimates of the standard errors.
5. Figures of the probability ROC (top) and *z*-transformed ROC (bottom) with the observed data points (open circles) and the hypothetical function relating the hit and false alarm rate given the parameter estimates (solid line) for each condition. Note that the condition labels are optional. However, if it is not provided, `roc_solver` will automatically populate this (condition#).

Estimating the standard error of the parameter estimates with `roc_solver`

The `roc_solver` function can estimate the standard error of the parameter estimates using a non-parametric bootstrap procedure. This procedure can be initialized by using the `bootIter` property-value optional input to `roc_solver`. The non-parametric routine randomly samples with replacement trials from the overserved `targf` and `luref` matrices to create bootstrapped `targf` and `luref` matrices. The model fit to the observed data is also fit to each one of the bootstrapped `targf` and `luref` randomizations. By Default, the standard errors are not estimated because it can take a long time to do so. The amount of time depends on the complexity of your model, the number of trials in the data set, and the number of iterations. The following code uses the `bootIter` option with a small number of iterations. On real data you would want to increase this number to probably no less than 1000 iterations.

```
bootIter = 100;
rocData = roc_solver(targf,luref,model,fitStat,x0,LB,UB,'bootIter',bootIter);
```

After the bootstrapped procedure is finished, you should see a figure similar to that shown above with the NaN values replaced by the bootstrapped estimated standard errors. Given the amount of time, the `bootIter` option is not used in the remainder of the tutorials.

Exploring the other optional inputs of `roc_solver`

Next, we will explore some of the optional property-value inputs that can be passed to `roc_solver` (see the help file for all of the optional inputs). The options that will be used here relate to passing information about the subject ID and experiment/model information to the data structure returned by `roc_solver`. First, clear the `rocData` variable:

```
clear rocData;
```

Now we need to define some information for our "subject" and "experiment". Type in the following:

```
subID = 'tutorial1_subject';
groupID = 'group1';
condLabels = {'item recognition'};
modelID = 'uvsd model 1';
outpath = fileparts(which('tutorial1_data.mat'));
```

The above commands will be used to define our subject ID (`subID`), the group classification (`groupID`), the names of the conditions (`condLabels`), the user-specified name for the model (`modelID`), and the directory to write a .pdf of the summary figure that appears after running the `roc_solver`. Now, we are ready to fit the same model as we did above to the data again (the results should be identical, so the figure will not be shown in this document). Type the following command:

```
rocData = roc_solver(targf,luref,model,fitStat,x0,LB,UB, ...
    'subID',subID, ...
    'groupID',groupID, ...
    'condLabels',condLabels, ...
    'modelID',modelID, ...
    'saveFig',outpath, ...
    'figure',true, ...
    'figTimeout',5);
```

The only difference between this command and the previous command using the roc_solver is the inclusion of the property-value optional inputs. The `'subID', 'groupID', 'conLabels',` and `'modelID'` options are used to store the subject ID, group ID, condition labels, and model ID, respectively, in the appropriate places in the data structure. As you should have noticed while the figure was on the screen, the information that was empty in area (1) in the above figure should now have values present.

The `'saveFig'` option tells the `roc_solver` where to save a .pdf of the summary figure. Open this .pdf file in the *examples* directory. The `'figure'` option can be used to turn off plotting the summary figure (by setting the value to false). This might be useful in you are running a large number of simulations, or if you are annoyed by the figure and feel you have the expertise to diagnose model fit just from fit statistics. However, the default value of the `'figure'` option is `true`, and we strongly suggest not setting this value to `false`. A better option might be to use `'figTimeout'`, which will automatically close the figure after a specified amount of time (in the example here, 5 seconds). The `'saveFig'` and `'figTimeout'` options can be used together to automatically save and close the figure, and the `'figure'` option is automatically set to `true` if `'saveFig'` is provided.

Before continuing, let's take a look at the structure of `rocData`.

```
>> rocData

rocData =

              subID: 'tutorial1_subject'
            groupID: 'group1'
    condition_labels: {'item recognition'}
       observed_data: [1x1 struct]
          uvsd_model: [1x1 struct]
```

Notice how the `uvsd_model` field is a 1x1 structure, and how there are no `dpsd_model` and `msd_model` fields.

Next, we fill fit three different models to the same data, and store it in the same data structure that we have been using (i.e., `rocData`). To accomplish this we will use the `'append'` optional input for the `roc_solver`. First, we will fit an equal-variance signal-detection (EVSD) model to the data. This is very similar to the UVSD model, with the difference that the V_o parameter is constrained to equal 1. First, let's create the appropriate `x0`, `LB`, and `UB` matrices to pass to the solver:

```
modelID = 'evsd model 1';
model = 'uvsd';
evsdParNames = {'Dprime'};
[x0,LB,UB] = gen_pars(model,nBins,nConds,evsdParNames);
```

We still have to use the UVSD model, but the EVSD model is nested within (i.e., a special case of) the full UVSD model. Additionally, you will notice that the 2nd column of the `LB` and `UB`

matrices (which corresponds to the V_o parameter) are equal, which indicates that the parameter is not estimated. This was automatically done by omitting the parameter name from the `evsdParNames` cell string, but it can be set after the fact by directly manipulating the `LB` and `UB` matrices. Now, run the `roc_solver` again:

```
rocData = roc_solver(targf,luref,model,fitStat,x0,LB,UB, ...
    'subID',subID, ...
    'groupID',groupID, ...
    'condLabels',condLabels, ...
    'modelID',modelID, ...
    'saveFig',outpath, ...
    'append',rocData);
```

Note in the summary figure the fit of this model to the data is not as good as the previous model, which is caused by the parameter constraint on V_o. The `'append'` option adds the results from current model to the structure data contained in `rocData`. Note that the first 4 optional inputs do not necessarily need to be included if this is not the first model stored in the data structure (and if this information has not changed). Because the model is still technically the UVSD model, albeit with a parameter constraint, the data will be stored in a new element of the `uvsd_model` field of the `rocData` structure. Again, examine the structure of `rocData`:

```
>> rocData

rocData =

              subID: 'tutorial1_subject'
            groupID: 'group1'
    condition_labels: {'item recognition'}
       observed_data: [1x1 struct]
          uvsd_model: [1x2 struct]
```

The `uvsd_model` field is now a 1x2 structure array. Next, examine each element of the `uvsd_model` field separately, and pay attention to the `modelID` field:

```
>> rocData.uvsd_model(1)

ans =

             modelID: 'uvsd model 1'
         model_notes: ''
      fit_statistics: [1x1 struct]
          parameters: [1x1 struct]
      predicted_data: [1x1 struct]
      predicted_rocs: [1x1 struct]
    optimization_info: [1x1 struct]
```

```
>> rocData.uvsd_model(2)

ans =

              modelID: 'evsd model 1'
          model_notes: ''
       fit_statistics: [1x1 struct]
           parameters: [1x1 struct]
       predicted_data: [1x1 struct]
       predicted_rocs: [1x1 struct]
    optimization_info: [1x1 struct]
```

Now we will fit a DPSD and MSD model to the same data. For the DPSD model, type the following code:

```
model = 'dpsd';
dpsdParNames = {'Ro' 'F'};
modelID = 'dpsd model 1';
[x0,LB,UB] = gen_pars(model,nBins,nConds,dpsdParNames);
rocData = roc_solver(targf,luref,model,fitStat,x0,LB,UB, ...
    'subID',subID, ...
    'groupID',groupID, ...
    'condLabels',condLabels, ...
    'modelID',modelID, ...
    'saveFig',outpath, ...
    'append',rocData);
```

This segment of code fill fit the DPSD model to the data with the recollection of oldness (`Ro`) and familiarity (`F`) parameters of the DPSD model. Notice now that there is a `dpsd_model` field added to `rocData`:

```
>> rocData

rocData =

               subID: 'tutorial1_subject'
             groupID: 'group1'
    condition_labels: {'item recognition'}
       observed_data: [1x1 struct]
          uvsd_model: [1x2 struct]
          dpsd_model: [1x1 struct]
```

Finally, we will fit a version of the MSD model to the data using the mixing parameter for target items (`lambda_targ`) and the strength of the items in the first strength distribution (`Dprime1_targ`; i.e., the strength of items above `lambda_targ`).

```
model = 'msd';
msdParNames = {'lambda_targ' 'Dprime1_targ'};
modelID = 'msd model 1';
[x0,LB,UB] = gen_pars(model,nBins,nConds,msdParNames);
```

```
rocData = roc_solver(targf,luref,model,fitStat,x0,LB,UB, ...
    'subID',subID, ...
    'groupID',groupID, ...
    'condLabels',condLabels, ...
    'modelID',modelID, ...
    'saveFig',outpath, ...
    'append',rocData);
```

Check the structure of the `rocData` one last time, and notice the addition of the `msd_model` field.

```
>> rocData

rocData =

               subID: 'tutorial1_subject'
             groupID: 'group1'
    condition_labels: {'item recognition'}
       observed_data: [1x1 struct]
          uvsd_model: [1x2 struct]
          dpsd_model: [1x1 struct]
           msd_model: [1x1 struct]
```

Tutorial 2: Multiple-Condition Designs & Parameter Constraints

Objectives:

- Use the `roc_solver` with multiple-condition data.
- Include parameter contraints to limit to value and range of values a parameter can take.
- Explore the `ignoreConds` and `constrfun` options of the `roc_solver`.

Overview

This tutorial uses the data in *tutorial2_data.mat*. There are 2 sets of frequencies, `targf1` and `luref1`, and `targf2` and `luref2`. These data were generated from the DPSD model, and there are two conditions (i.e., rows) for each data set. For the 1st data set, the parameters for Condition 1 (i.e., row 1) were: Ro = .3, Rn = .3, F = .85. For Condition 2, the parameters were: Ro = .3, Rn = .1, F = .75. There are six rating bins, thus there are five criterion values for each condition equally spaced between -1.5 to +1.5.

The second data set was also generated from the DPSD model, but with different parameters. For Condition 1, the parameters were: Ro = .5, Rn = 0, F = .9. The parameters for Condition 2 were: Ro = .25, Rn = 0, F = .9. Again, there are six rating bins, and the five criterion values for each condition equally spaced between -1.5 to +1.5. However, note that frequencies for lures items Condition 1 and Condition 2 were identical. This is meant to reflect a design where there are multiple types of target trials, but only a single type of lure trials.

It is assumed that you have completed Tutorial 1, and have a good understanding of the basic functionality of the toolbox. The variable names used in the first tutorial are also used here for continuity. This tutorial is more advanced and will utilize different optional inputs for the `roc_solver`. In this tutorial you will learn how to fit multiple condition data when you have unique Target/Lure conditions (i.e., target types with a unique corresponding lure type) as well as designs with multiple target conditions and only a single type of lure item. Moreover, you will be shown how to constrain parameters using the lower (`LB`) and upper (`UB`) bound matrices as well as non-linear equality constraint functions.

Load the data and define some initial information

First, load the *tutorial2_data.*mat file by typing the following:

```
load('tutorial2_data.mat');
```

This file contains the target and lure frequency matrices for the two data sets.

Next, define the fit statistic that will be used to optimize the parameters, the model you want to fit, and some information about the deisgn.

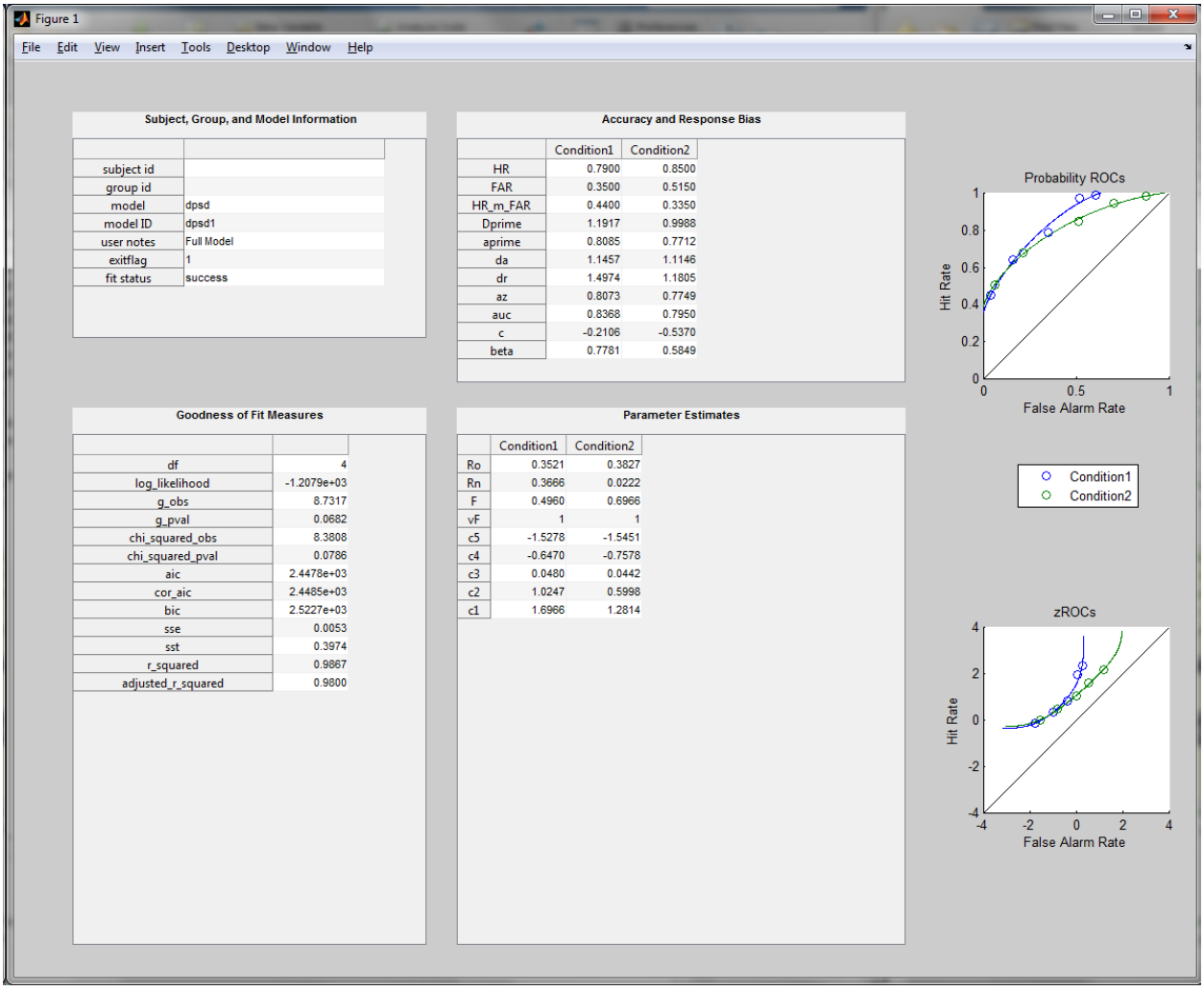
```
fitStat = '-LL';
model = 'dpsd';
[nConds,nBins] = size(targf1);
parNames = {'Ro' 'Rn' 'F'};
```

Fit different variants of the same model to the first data set (`targf1` and `luref1`)

For this tutorial we will focus on the first data set. A total of four different iterations of the DPSD model will be fit to this data. First, we will simply fit a "Full Model" to the data. Recall that the current data was generated from a DPSD model with the Ro, Rn, and F parameters. Use the following code to fit the model to the data:

```
rocData1 = [];
[x0, LB, UB] = gen_pars(model,nBins,nConds,parNames);
modelNotes = 'Full Model';
rocData1 = roc_solver(targf1,luref1,model,fitStat,x0,LB,UB, ...
    'notes',modelNotes, ...
    'append',rocData1);
```

This model is the "Full Model" because all of the parameters we want to estimate are allowed to vary independently of each other. This is what the summary information figure should look like:



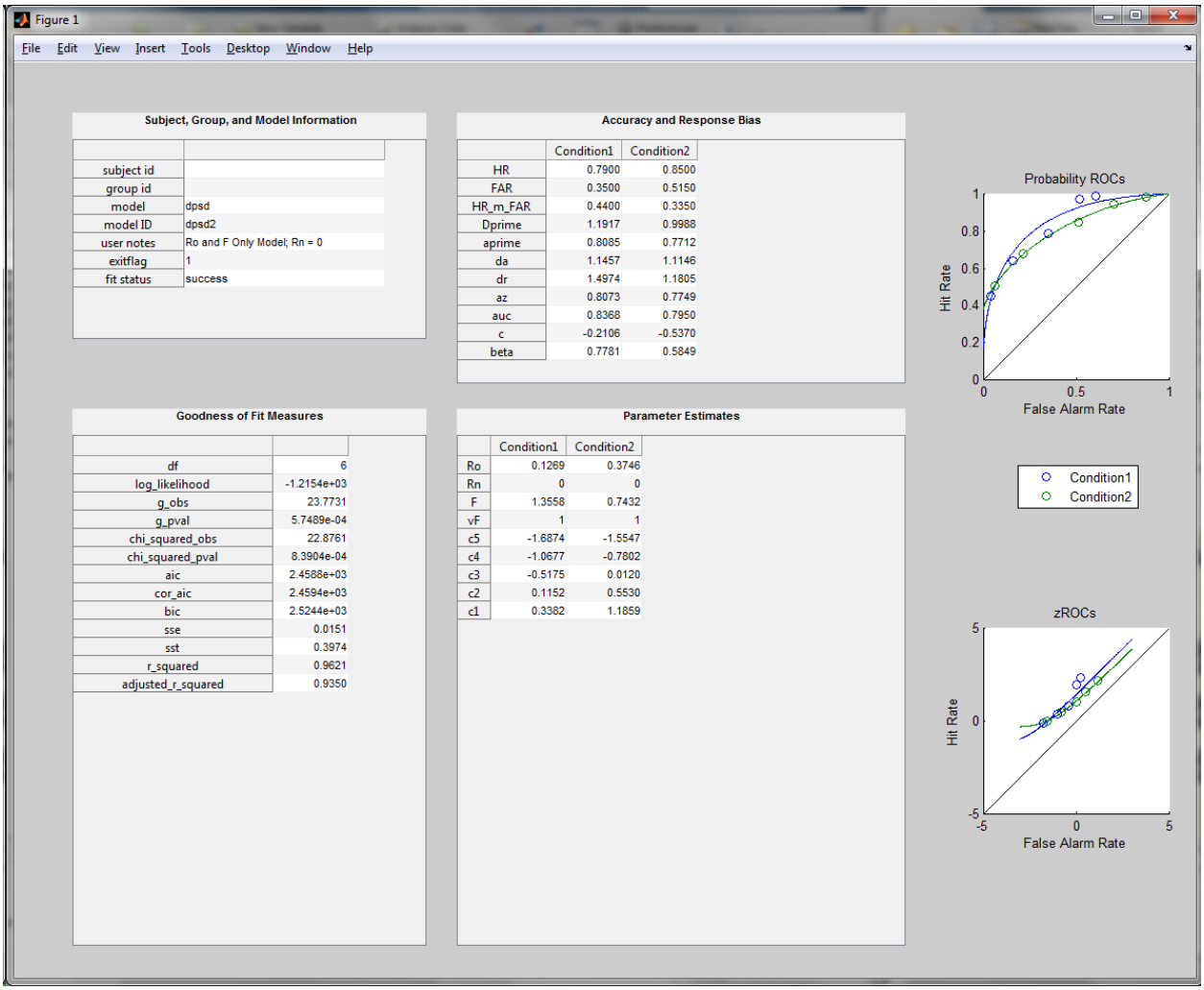
The next model that will be fit to the data is a model where we do not estimate Rn. In this case, we want to constrain Rn to equal 0. This can be achieved using the `gen_pars` function as we did above with the exception that Rn is not included in the `parNames` variable (see Tutorial 1 for an example of a DPSD model with just Ro and F). However, you can also manually change the `x0`, `LB`, and `UB` variables. This latter approach might be beneficial in situations where you don't want to estimate a parameter such as Rn but you want the parameter to have a value other than 0 (e.g., .5). The following will force Rn to equal 0 in the next model.

```
x0(:,2) = 0;
LB(:,2) = 0;
UB(:,2) = 0;
```

For the DPSD model, the Rn parameter corresponds to the 2nd column in the above matrices. Next, run the `roc_solver` on this new (i.e., constrained) model.

```
modelNotes = 'Ro and F Only Model; Rn = 0';
rocData1 = roc_solver(targf1,luref1,model,fitStat,x0,LB,UB, ...
    'notes',modelNotes, ...
    'append',rocData1);
```

Note in the summary figure that Rn in both conditions is now equal to 0. Additionally, looking at the fit statistics table you should notice that there are more degrees of freedom (df) for this model where Rn is forced to equal 0 (i.e., it is not estimated).

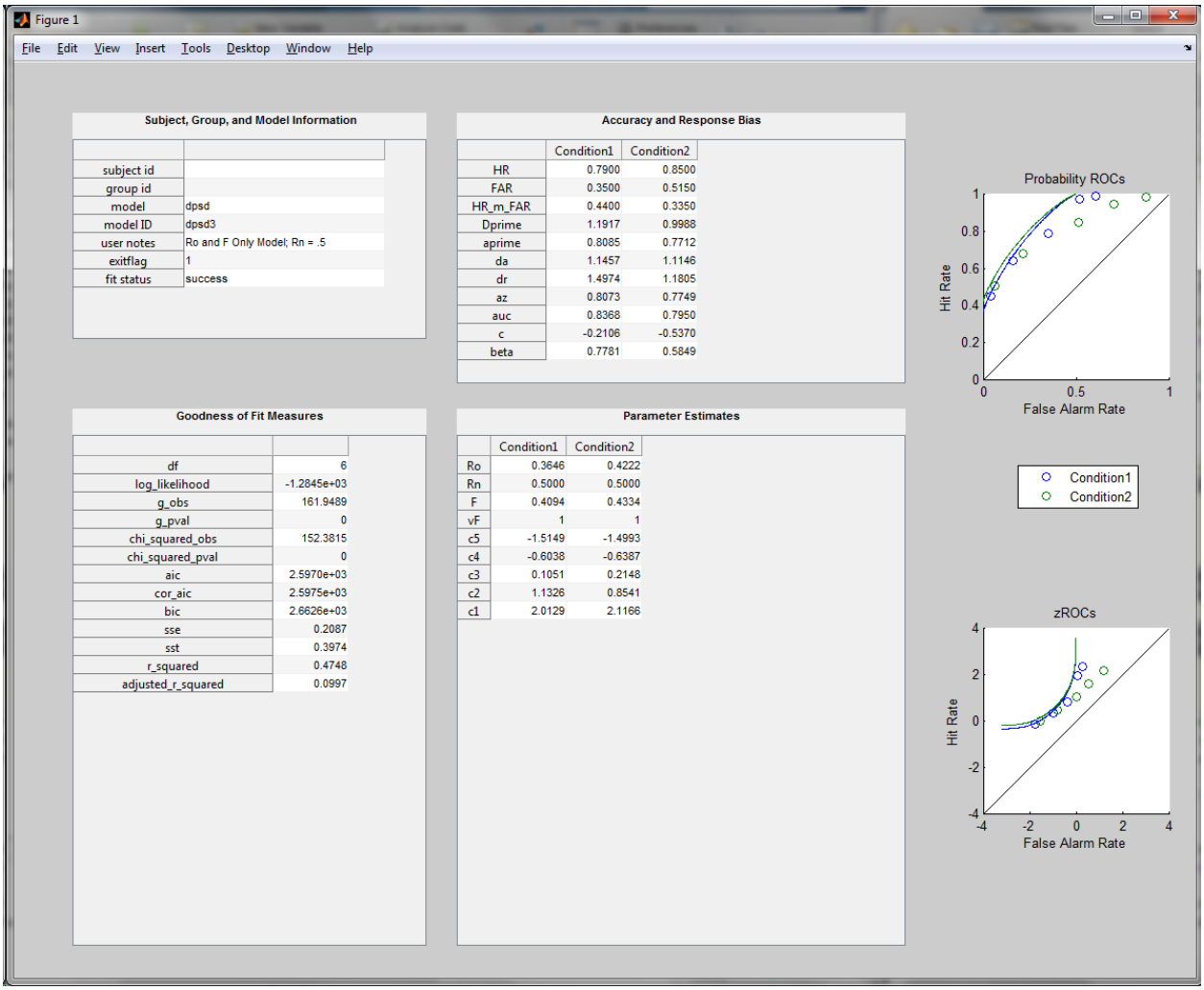


The next model is nearly identical to the above, except now we will force Rn to equal .5 instead of 0.

```
x0(:,2) = .5;
LB(:,2) = .5;
UB(:,2) = .5;
modelNotes = 'Ro and F Only Model; Rn = .5';
rocData1 = roc_solver(targf1,luref1,model,fitStat,x0,LB,UB, ...
    'notes',modelNotes, ...
```

```
'append',rocData1);
```

As with the model where Rn = 0, you should now see that Rn = .5 in both conditions. Also note that this model and the previous model have the same number of degrees of freedom, but they do not provide the same numerical fit to the data (see e.g., *G*, or g_obs).



The previous two models illustrate how to constrain parameters in the model to a specific value. This was accomplished by an element in `LB` being set equal to the corresponding element in `UB`. The primary use of such a constraint is to determine which parameters of a model are estimated and which are not, or to limit the range of potential values a parameter can take.

However, this type of constraint is not helpful when one wants to place an equality constraint on a parameter across conditions, or even within a condition. This is accomplished in the ROC toolbox by using nonlinear constraint functions that are passed to the `roc_solver`. These functions are very useful in that essentially any parameter can be constrained to equal any

other parameter. However, the difficulty arises in that you will have to write your own constraint function that is specific to your model (see Appendix C for more information). We have provided three nonlinear constraint functions with the ROC toolbox. One function that constrains the criterion parameters to be equal across conditions will be used later on in this tutorial. The other two functions (for the DPSD and MSD models) are referred to as symmetry constraint functions. The function for the DPSD (`dpsd_sym_constr`) makes Ro = Rn within each condition, whereas the function for the MSD model (`msd_sym_constr`) makes lambda_targ = lambda_lure within each condition. Here, we will explore how to use constraint functions using the `dpsd_sym_constr` function.

First, we need to make the `x0`, `LB`, and `UB` matrices, and define some model notes.

```
modelNotes = 'Symmetrical Model [Ro(i) = Rn(i)]';
[x0, LB, UB] = gen_pars(model,nBins,nConds,parNames);
```

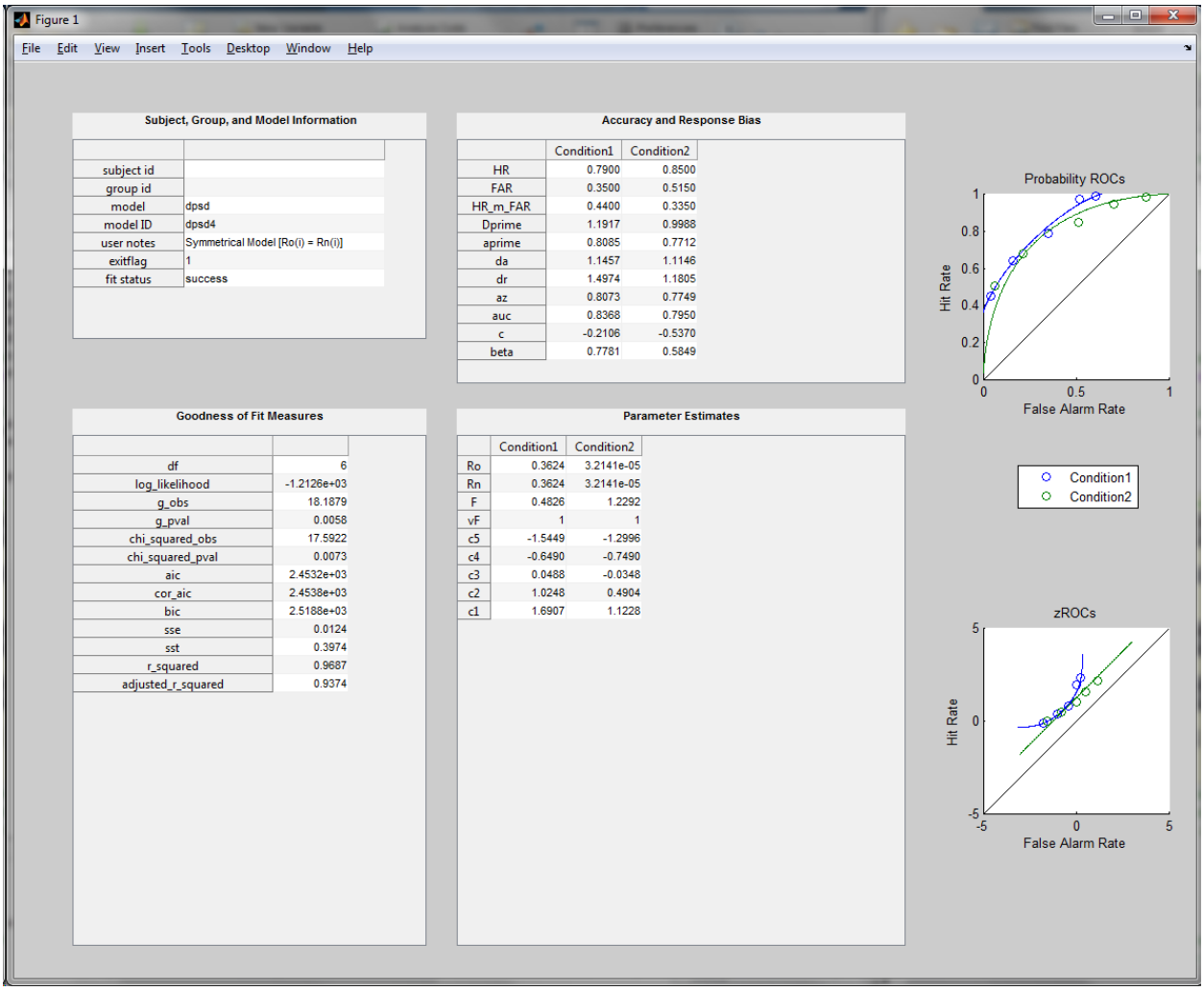
To use a constraint function, the function needs to be passed as a function handle to the `roc_solver`.

```
constrfun = @dpsd_sym_constr;
```

This function handle will be passed to `roc_solver` using the `'constrfun'` flag.

```
rocData1 = roc_solver(targf1,luref1,model,fitStat,x0,LB,UB, ...
    'notes',modelNotes, ...
    'constrfun',constrfun, ...
    'append',rocData1);
```

What you should notice in the summary information is that the value for Ro and Rn are equal within each condition, but are different between the two conditions.



Designs with multiple types of targets and a single type of lure item (`targf2` and `luref2`)

The above examples focused on data from multiple condition designs where there are unique target/lure conditions. However, there might be situations where there are multiple types of target items with only one type of lure item. A classic example in the memory literature is the levels of processing procedure. In this task, participants study a list of words and make a deep semantic judgment (i.e., pleasantness) about half of the words and make a shallow judgment (i.e., number of vowels) about the other half of the words. At test, these deep and shallow encoded (target) words are intermixed with a single set of new (lure) words. The new items cannot be attributed to the deep and shallow conditions, so the model needs to be on such that there is only a single set of criterion parameters estimated across all conditions (i.e., the criterion parameters must be constrained across the conditions).

Importantly, because there is only one set of lure items, we do not want to count the contribution of the lure items multiple times when calculating the goodness of fit measure that

is minimized. To avoid this, we can use the `'ignoreConds'` optional input to the `roc_solver`. The data passed through this option is a vector of row (i.e., conditions) numbers whose lure items should be excluded from the calculation of the fit statistic. This cannot contain the first condition because the criterion parameters for the row that will be ignored are set to equal the previous row. For example, by passing a value of 2 to the `'ignoreConds'` option, you are telling the `roc_solver` function that the criteria parameters for condition 2 should equal condition 1. Thus, when using this option, the order of conditions is very important.

To illustrate this, we will work through an example using the second data set included in this tutorial. First, examine the contents of `luref`:

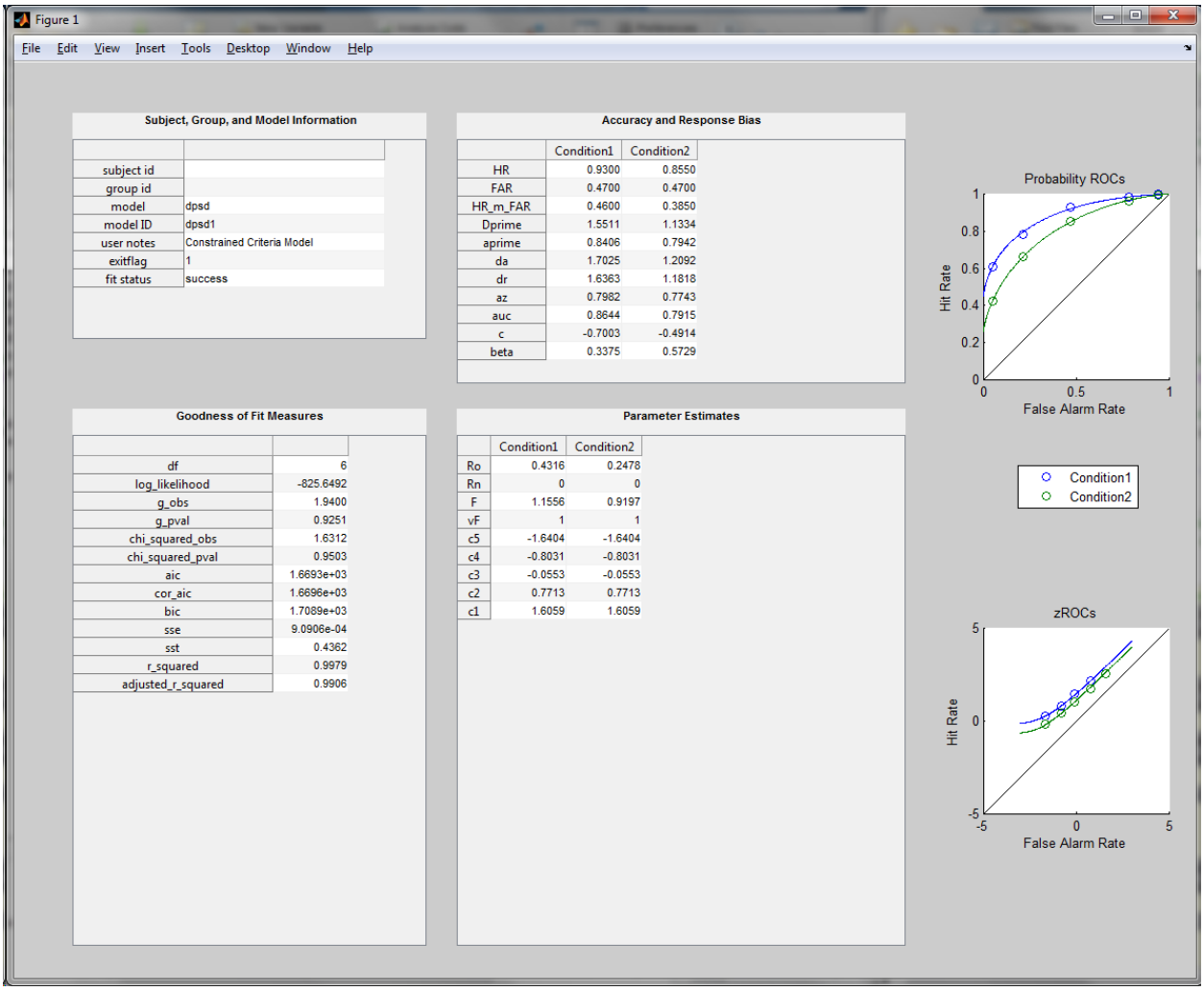
```
>> luref2

luref2 =

    10    33    51    63    32    11
    10    33    51    63    32    11
```

Take note that both rows of `luref` have identical values. This is a necessity for the toolbox because `targf` and `luref` have to be the same size; else many of the functions will return an error. Thus, when organizing your own data in an experimental design such as this, you will have to repeat the same frequencies in `luref`.

Next, we will generate the staring value and bounding parameters, and fit the model to the data. Type the following:

```
parNames = {'Ro' 'F'};
[x0, LB, UB] = gen_pars(model,nBins,nConds,parNames);
modelNotes = 'Constrained Criteria Model';
rocData2 = [];
rocData2 = roc_solver(targf2,luref2,model,fitStat,x0,LB,UB, ...
    'notes',modelNotes, ...
    'ignoreConds',ignoreConds, ...
    'append',rocData2);
```

The use of the `'ignoreConds'` option automatically tells the `roc_solver` to constrain the criterion parameters between condition 2 and condition 1 to be equal. This equality constraint is achieved with the `criteria_constraint` function that is included in the toolbox. This function forces the criterion parameters for row M in the `x0` (i.e., parameter matrix) to equal the same parameter in row M – 1. This is the output you should get from the above model:



Importantly, the `criteria_constraint` function is written such that you can also use an additional criterion constraint function with the `'constrfun'` optional in the `roc_solver`.

Tutorial 3: Importing Data and Extracting Group Data

Objectives:

- Use the `roc_import_data` function to import data.
- Use the `get_group_data` to extract group data.

Overview

This tutorial uses the data in *tutorial3_data.csv*. This file contains the long format of a simulated data set with two subjects, each with two conditions. There are 10 rating bins in the two conditions, and the data was generated from a DPSD model with only the Ro and F parameters. The code for this tutorial is in *tutorial3_script.m*.

Import the data with the `roc_import_data` function

You can import data from .txt or .csv files into the proper format using the `roc_import_data` function. See the Formatting the Data section for a discussion on how to create this file. Before importing the data, we will first define a directory to write our data to (the `saveDir` variable).

```
saveDir = fullfile(fileparts(which('tutorial3_data.csv')),'tutorial3');
mkdir(saveDir);
```

This will create a *tutorial3* folder in the *examples* directory of the ROC Toolbox. Next, we will import the data and store it in a variable called `rawData`:

```
rawData = roc_import_data('tutorial3_data.csv');
```

`rawData` is a cell array of structures, where each cell contains the data for one subject ID. To get an idea of what the fields are, let's examine the contents of the first subject:

```
>> rawData{1}

ans =

    condLabels: {2x1 cell}
         subID: '101'
       groupID: 'group1'
         targf: [2x10 double]
         luref: [2x10 double]
```

The frequency data used for the `roc_solver` is in the `targf` and `luref` fields. Other important information that can be passed as optional input to the `roc_solver` is contained in the

`condLabels` (condition labels), `subID` (subject ID), and `groupID` fields. Note that the `groupID` field is mainly intended for between-subject variables (e.g., controls vs. patients).

Before writing a `for` loop to fit the model to each subjects data, let's first define some information about the model similar to Tutorials 1 and 2:

```
[nConds,nBins] = size(rawData{1}.targf);
nSubs = length(rawData);
fitStat = '-LL';
```

Next, we will fit the EVSD (i.e., UVSD model with Vo = 1), UVSD, DPSD, and MSD models to each subjects data in a `for` loop. For each subject and each model, the output from `roc_solver` will be stored in `rocData`, and `rocData` will be saved to a .mat file in the *tutorial3* folder, along with the summary figure for each model for a given subject's data.

The loop also stores the name of the saved .mat file in a cell array of strings (`subFiles`), which will be passed to the `get_group_data` function to extract the data. Also, the `'figTimeout'` option is used in the `roc_solver` commands, and is set to 2 seconds so that you do not have to manually close each window. Also, the figures are saved so you can examine them at a later point.

```
subFiles = {};
for i = 1:length(rawData)
    % Initialize rocData anew for each subject
    rocData = [];

    % Fit the UVSD model to the data
    model = 'uvsd';
    modelID = 'evsd_model';
    parNames = {'Dprime'};
    [x0, LB, UB] = gen_pars(model,nBins,nConds,parNames);
    rocData = roc_solver(rawData{i}.targf,rawData{i}.luref, ...
        model,fitStat,x0,LB,UB, ...
        'subID',rawData{i}.subID, ...
        'groupID',rawData{i}.groupID, ...
        'condLabels',rawData{i}.condLabels, ...
        'modelID',modelID, ...
        'saveFig',saveDir, ...
        'figTimeout',2, ...
        'append',rocData);

    % Fit the UVSD model to the data
    model = 'uvsd';
    modelID = 'uvsd_model';
    parNames = {'Dprime' 'Vo'};
    [x0, LB, UB] = gen_pars(model,nBins,nConds,parNames);
    rocData = roc_solver(rawData{i}.targf,rawData{i}.luref, ...
        model,fitStat,x0,LB,UB, ...
```

```
        'subID',rawData{i}.subID, ...
        'groupID',rawData{i}.groupID, ...
        'condLabels',rawData{i}.condLabels, ...
        'modelID',modelID, ...
        'saveFig',saveDir, ...
        'figTimeout',2, ...
        'append',rocData);

    % Fit the DPSD model to the data
    model = 'dpsd';
    modelID = 'dpsd_model';
    parNames = {'Ro' 'F'};
    [x0, LB, UB] = gen_pars(model,nBins,nConds,parNames);
    rocData = roc_solver(rawData{i}.targf,rawData{i}.luref, ...
        model,fitStat,x0,LB,UB, ...
        'subID',rawData{i}.subID, ...
        'groupID',rawData{i}.groupID, ...
        'condLabels',rawData{i}.condLabels, ...
        'modelID',modelID, ...
        'saveFig',saveDir, ...
        'figTimeout',2, ...
        'append',rocData);

    % Fit the MSD model to the data
    model = 'msd';
    modelID = 'msd_model';
    parNames = {'lambda_targ' 'Dprime1_targ'};
    [x0, LB, UB] = gen_pars(model,nBins,nConds,parNames);
    rocData = roc_solver(rawData{i}.targf,rawData{i}.luref, ...
        model,fitStat,x0,LB,UB, ...
        'subID',rawData{i}.subID, ...
        'groupID',rawData{i}.groupID, ...
        'condLabels',rawData{i}.condLabels, ...
        'modelID',modelID, ...
        'saveFig',saveDir, ...
        'figTimeout',2, ...
        'append',rocData);

    % Save rocData to a .mat file for the current subject
    matFile = fullfile(saveDir,strcat(rawData{i}.subID,'_rocData.mat'));
    subFiles{i} = matFile;
    save(matFile,'rocData');
end
```

After all of the models are fit to the data, we are ready to combine each individual's data into a format that will be helpful for group analyses. First, we will use the `get_group_data` function to extract the information for the EVSD model in each subject. The `get_group_data` function returns a structure variable that contains the data for each subject (rows) and, if applicable, each condition (columns). Moreover, this function can write the group data to a .csv or a .txt file with the `'saveCSV'` or `'saveTXT'` optional inputs. Type in the following:

```
evsdPrefix = fullfile(saveDir,'evsd_group_data');
evsdData = get_group_data(subFiles,'uvsd',1,'rocData','saveCSV',evsdPrefix);
```

The evsdPrefix is a string of the path and file name (excluding the extension) of the .csv/.txt file that you want the data written to. The first 4 inputs to the get_group_data function are required, and the last two inputs are optional. The first input is a cell array of strings with each subject's .mat file where the data is stored. The second and third inputs are the model that you want to extract the group data from, and the iteration (i.e., array index) of that model, respectively. As you will see below, we will use the value of 2 to extract data from the UVSD model (instead of the EVSD model, which is a constrained version of the UVSD model). The fourth input is a string indicating the variable that the output from the roc_solver function was stored in. Here, we stored the data in rocData, but you can store the output in any variable you want. Let's explore the structure of the evsdData to see how the group data is organized:

```
>> evsdData =

                subID: {2x1 cell}
              groupID: {2x1 cell}
           conditions: {2x1 cell}
     rating_frequency: [1x1 struct]
    acc_bias_measures: [1x1 struct]
        par_estimates: [1x1 struct]
            fit_stats: [1x1 struct]
         model_verify: [1x1 struct]
```

The subID and groupID fields contain the subject IDs and between-subject group labels, respectively. The conditions fields contain the condition labels for the models (but not for each subject, since they are the same). The rating_frequency field contains the target and lure frequencies for each subject and each condition. Note that the conditions here are in different columns of a cell array. Thus, the first cell of the evsdData.rating_frequency.target field is a matrix of frequencies for all subjects in the first condition (i.e., evsdData.conditions{1}). The acc_bias_measures contains the data for the different signal detection measures of accuracy and response bias. The par_estimates field contains the best fitting parameter estimates for the different parameters in the model. The parameter (i.e., field) names of the par_estimates structure will differ depending on the model that is fit to the data. In the criterion field of par_estimates, each cell corresponds to a matrix where the rows are the subjects and the columns are different conditions. Each cell corresponds to a different criterion parameter in a descending fashion, such that evsdData.par_estimates.criterion{1} is the most conservative criterion parameter, evsdData.par_estimates.criterion{2} is the next most conservative, and so on. The fit_stats field contains the different goodness of fit measures. Finally, the model_verify field contains diagnostic information that can be used to

check that the data you have extracted for each subject is from the same model. This is important if, as in Tutorial 2, you have multiple iterations of the same model. Now, extract the group data for the other three models, and save the group data to a .mat file:

```
% Extract the UVSD data
uvsdPrefix = fullfile(saveDir,'uvsd_group_data');
uvsdData = get_group_data(subFiles,'uvsd',2,'rocData','saveCSV',uvsdPrefix);

% Extract the DPSD data
dpsdPrefix = fullfile(saveDir,'dpsd_group_data');
dpsdData = get_group_data(subFiles,'dpsd',1,'rocData','saveCSV',dpsdPrefix);

% Extract the MSD data
msdPrefix = fullfile(saveDir,'msd_group_data');
msdData = get_group_data(subFiles,'msd',1,'rocData','saveCSV',msdPrefix);

% Save the group data structures to a .mat file
save(fullfile(saveDir,'all_group_data.mat'),'evsdData','uvsdData', ...
    'dpsdData','msdData');
```

If you explore the *tutorial3* directory that was created earlier, you will notice a bunch of .csv files that contain the data from the `get_group_data` function. Four files are created for each model: **freq_data.csv*, **acc_bias_data.csv*, **pars.csv*, and **fit_stats.csv*. These files contain, respectively, the rating frequency, accuracy/response bias, parameter estimates, and fit statistic data. Open these up and explore how they are organized. This ends the last tutorial.

Appendix A: Signal Detection Analysis of Yes/No Data

The ROC toolbox is capable of analyzing simple yes/no data with signal detection theory metrics of accuracy and response bias. This type of analysis could be applied to data from recognition experiments that have a single hit rate (HR) and a single false alarm rate (FAR) for each participant/condition. The accuracy measures that can be calculated with functions in this toolbox are *d*' and A', and the bias measures are *c* and β. The functions that calculate these are:

`calc_acc_dprime(HR, FAR)` → Calculates *d*'
`calc_acc_aprime(HR, FAR)` → Calculates A'
`calc_bias_c(HR, FAR)` → Calculates *c*
`calc_bias_beta(HR, FAR)` → Calculates b

Each function takes as input a vector of hit rates (HRs) and false alarm rates (FARs) such that the first element of each vector constitutes a HR/FAR pair from one participant/condition. These values must be proportions (e.g., # Hits/# Target Items). Note that HRs and FARs of 1's or 0's will lead to undefined values for *d*', *c*, and β. Corrections to your raw data can be applied to avoid this (see Macmillan & Creelman, 2005; Snodgrass & Corwin, 1988, for further discussion).

Likewise, if you have some sort of ratings (i.e., confidence) data, there are additional signal detection summary indices that can be calculated. These include d_a, d_r, *A*z, and area under the curve (AUC). These formulas, as well as the ones mentioned above, are discussed in Macmillan and Creelman (2005), with the exception of the d_r index. The formula of the d_r index can be found in Mickes, Wais, & Wixted (2007). To calculate these metrics, as well as the three presented above, you can use the following function:

```
calc_rating_acc_bias(targf, luref)
```

where `targf` and `luref` refer to a matrix of response frequencies for target and lure items, respectively (see the **Formatting Your Data** section).

Appendix B: The Models and Their Parameters

The ROC Toolbox currently contains three different models that can be fit to ROC data. These include the unequal-variance signal detection (USVD) model, the dual-process signal detection model (DPSD), and the mixture signal detection (MSD) model. Here is a brief outline of the parameters of each model:

The UVSD Model

The parameters of this model are:

Parameter	Description
d'	Distance between the target and lure distributions in standard deviation units. This is typically referred to as the "strength" of the target distribution. Note that the mean and standard deviation of the lure distribution are 0 and 1, respectively. If estimated, the starting value is 1 with bounds of $\pm\infty$. If this parameter is not estimated, it is constrained to equal 0.
V_o	This is the variance of the target distribution in standard deviation units. If estimated, the starting value is 1.4 with bounds of 0 and $+\infty$. If not estimated, this parameter is constrained to equal 1. This constraint results in an equal-variance signal detection model.
$\{c_1, c_2, \ldots, c_k\}$	These are the criterion placement parameters of the model. The number of criterion parameters is determined by the number of rating bins (B_i) minus 1 (i.e., $B_i - 1$). By default, the first (i.e., most conservative) criterion parameter is can vary between $\pm\infty$, whereas the remaining criterion points have a lower bound of 1 x 10^{-10} and an upper bound of ∞. Thus, the criterion locations after the most conservative criteria are estimated as the interval between criteria. This is done avoid estimation errors that can result in more liberal response bins have a more conservative criteria estimate than more conservative response bins (e.g., $c_2 < c_1$). These parameters are always estimated, and the starting values are equally spaced between ±1.5. The bounds on the 1st criterion parameter are $\pm\infty$, whereas the bounds on the remaining criterion parameters are 0 and $+\infty$.

Here is the formula for the UVSD model:

$$p(B_i|\text{Target}) = \Phi(c_k - d', \text{V}_\text{o})$$
$$p(B_i|\text{lure}) = \Phi(c_k, 1)$$

B_i represents the i^{th} rating bin, $p(B_i|\text{target})$ is the proportion of responses in each bin for target items, $p(B_i|\text{lure})$ is the proportion of responses in each bin for lure items, and Φ is the Gaussian cumulative distribution function.

The DPSD Model

The parameters of the DPSD model, developed by Yonelinas (1994, 1999) are:

Parameter	Description
R_o	This is the recollection of "oldness" parameter, and it only influences target distribution (see the formulas below). When this parameter is estimated, it is constrained to have a value between 0 and 1 and a starting value of .2. Thus, this parameter is estimated as a probability. If not estimated, the parameter is set to 0.
R_n	This is the recollection of "newness" parameter, and it only influences the lure distribution (see the formulas below). When this parameter is estimated, it is constrained to have a value between 0 and 1, and a starting value of .2. Thus, this parameter is estimated as a probability. If not estimated, the parameter is set to 0.
F	This is the familiarity parameter of the DPSD model, and is estimated in *d'* units (similar to the *d'* parameter of the UVSD model). In other words, this parameter reflects distance between the familiarity strengths of the target and lure distributions. Again, note that the mean and standard deviation of the lure distribution are 0 and 1, respectively. If estimated, the parameter has a starting value of 1 with bounds of $\pm\infty$. If this parameter is not estimated, it is constrained to equal 0.
vF	This is the variance of the familiarity strength distribution for target items, and is estimated in standard deviation units.

	If estimated, the starting value is 1 with bounds of 0 and $+\infty$. If not estimated, this parameter is constrained to equal 1.
$\{c_1, c_2, \ldots, c_k\}$	These are the criterion placement parameters of the model. The number of criterion parameters is determined by the number of rating bins (B_i) minus 1 (i.e., $B_i - 1$). By default, the first (i.e., most conservative) criterion parameter is can vary between $\pm\infty$, whereas the remaining criterion points have a lower bound of 1 x 10^{-10} and an upper bound of ∞. Thus, the criterion locations after the most conservative criteria are estimated as the interval between criteria. This is done avoid estimation errors that can result in more liberal response bins have a more conservative criteria estimate than more conservative response bins (e.g., $c_2 < c_1$). These parameters are always estimated, and the starting values are equally spaced between ±1.5. The bounds on the 1st criterion parameter are $\pm\infty$, whereas the bounds on the remaining criterion parameters are 0 and $+\infty$.

The formulas for the DPSD are:

$$p(B_i|\text{target}) = R_o + (1 - R_o)\Phi(c_k - F, \text{v}F)$$
$$p(B_i|\text{lure}) = (1 - R_n)\Phi(c_k, 1)$$

B_i represents the i^{th} rating bin, $p(B_i|\text{target})$ is the proportion of responses in each bin for target items, $p(B_i|\text{lure})$ is the proportion of responses in each bin for lure items, and Φ is the Gaussian cumulative distribution function.

The MSD Model

The parameters of the MSD model, developed by DeCarlo (2002,2003), are:

Parameter	Description
λ_{targ} lambda_targ	This is the mixing parameter defining how the two latent class target distributions are mixed together. This parameter only exerts an influence on the target distributions (see the formulas below). When this parameter is estimated, it is constrained to have a value between 0 and 1, and a starting value of .2. Thus, this parameter is estimated as a probability.

	If not estimated, the parameter is set to 1, which means that there is only 1 distribution representing target items.
d'_{1targ} Dprime1_targ	This is the memory strength of the target distribution that exceeds, or is scaled by, λ_{targ}. This parameter is estimated as the mean difference, in standard deviation units, between d'_{1targ} and d'_{2targ} (see the formula). Importantly, however, the value output by the `roc_solver`, which the user sees, is $d'_{1targ} + d'_{2targ}$. When this parameter is estimated, it is constrained to have a value between 0 and $+\infty$, and has a starting value of 1.5. The lower bound of the estimated parameter is 0 because of the way the parameter is estimated (discussed above). If not estimated, the parameter is set to 0.
V_{1targ} var1_targ	This is the variance (i.e., standard deviation) the target distribution that is scaled by, λ_{targ}. If estimated, the starting value is 1 with bounds of 0 and $+\infty$. If not estimated, this parameter is constrained to equal 1. When estimated, the parameter can take any positive value.
d'_{2targ} Dprime2_targ	This is the memory strength of the target distribution that exceeds, or is scaled by, $1-\lambda_{targ}$. If estimated, the parameter has a starting value of 1 with bounds of $\pm\infty$. If this parameter is not estimated, it is constrained to equal 0.
V_{2targ} var2_targ	This is the variance (i.e., standard deviation) the target distribution that is scaled by, $1-\lambda_{targ}$. If estimated, the starting value is 1 with bounds of 0 and $+\infty$. If not estimated, this parameter is constrained to equal 1. When estimated, the parameter can take any positive value.
λ_{lure} lambda_lure	This is the mixing parameter defining how the two latent class lure distributions are mixed together. This parameter only exerts an influence on the lure distributions (see the formulas below). When this parameter is estimated, it is constrained to have a value between 0 and 1, and a starting value of .2. Thus, this parameter is estimated as a probability.

	If not estimated, the parameter is set to 1, which means that there is only 1 distribution representing target items.
d'_{1lure} Dprime1_lure	This is the memory strength of the lure distribution that exceeds, or is scaled by, λ_{lure}. This parameter is estimated as the mean difference, in standard deviation units, between d'_{1lure} and d'_{2lure} (see the formula). Importantly, however, the value output by the roc_solver, which the user sees, is $d'_{1lure} + d'_{2lure}$. When this parameter is estimated, it is constrained to have a value between 0 and $+\infty$, and has a starting value of 1.5. The lower bound of the estimated parameter is 0 because of the way the parameter is estimated (discussed above). If not estimated, the parameter is set to 0.
V_{1lure} var1_lure	This is the variance (i.e., standard deviation) the lure distribution that is scaled by, λ_{lure}. If estimated, the starting value is 1 with bounds of 0 and $+\infty$. If not estimated, this parameter is constrained to equal 1. When estimated, the parameter can take any positive value.
$\{c_1, c_2, \dots, c_k\}$	These are the criterion placement parameters of the model. The number of criterion parameters is determined by the number of rating bins (B_i) minus 1 (i.e., $B_i - 1$). By default, the first (i.e., most conservative) criterion parameter is can vary between $\pm\infty$, whereas the remaining criterion points have a lower bound of 1 x 10^{-10} and an upper bound of ∞. Thus, the criterion locations after the most conservative criteria are estimated as the interval between criteria. This is done avoid estimation errors that can result in more liberal response bins have a more conservative criteria estimate than more conservative response bins (e.g., $c_2 < c_1$). These parameters are always estimated, and the starting values are equally spaced between ±1.5. The bounds on the 1st criterion parameter are ±∞, whereas the bounds on the remaining criterion parameters are 0 and +∞.

The formulas for the MSD are:

$$p(B_i|\text{target}) = \lambda_{targ}\Phi\left(c_k - \left[d'_{1targ} + d'_{2targ}\right], \mathrm{V}_{1targ}\right) + \left(1 - \lambda_{targ}\right)\Phi\left(c_k - d'_{2targ}, \mathrm{V}_{2targ}\right)$$

$$p(B_i|\text{lure}) = \lambda_{lure}\Phi(c_k + \left[d'_{2lure} - d'_{1lure}\right], \mathrm{V}_{1lure}) + (1 - \lambda_{lure})\Phi(c_k, 1)$$

B_i represents the i^{th} rating bin, $p(B_i|\text{target})$ is the proportion of responses in each bin for target items, $p(B_i|\text{lure})$ is the proportion of responses in each bin for lure items, and Φ is the Gaussian cumulative distribution function.

Appendix C: Adding to the toolbox

Nonlinear constraint functions and other models can be added to the ROC toolbox to fit your specific needs. Below, we provide a very brief overview of how to add to the toolbox. If you have any questions, you should contact koen [dot] joshua [at] gmail [dot] com.

Constraint Functions

A constraint function must have four puts and accepts a matrix of parameter estimates (`pars`, or `x0`) as input. For more help with the `c` and `ceq` outputs (see below), see help `fmincon`. The four outputs are:

1. `c` → A vector for the inequality constraints.
2. `ceq` → A vector for the equality constraints.
3. `n` → The number of equality constraints. This is used for calculating the degrees of freedom appropriately.
4. `constrModel` → This is a string output that specifies what model the equality constraint is for. This is used by `roc_solver` to check that the equality constraint can be used with the model that is currently being fit to the data.

The primary input is the pars matrix, but there can be others. See the .m files in the *constraint_funcs* folder of the toolbox.

Adding New Models

New signal detection models can be added to the toolbox assuming the model can accommodate cumulative confidence data. A new model requires 4 functions, and the modification of an existing function. Say we want to add a new model, which we will refer to as "modelX" below.

First, to add modelX to the toolbox, we need to at it to the `get_all_models` function, which contains a cell array of string of all the valid models in the toolbox. This function is in the *utilities* directory of the ROC toolbox.

Create a folder in the *models* directory named modelX. This step isn't necessary, but is helpful for organization. The four functions for modelX will be saved in this directory. The name of the four functions to add are listed below, along with the primary computation of the function:

1. `modelX_calc_pred_data` → This function calculates the predicted data given a set of parameters.

2. `modelX_gen_pars` → This function creates the `x0`, `LB`, and `UB` parameters to be used with `roc_solver`.
3. `modelX_gen_sim_data` → This function generates simulated data from the model given a set of parameters.
4. `modelX_info` → This function contains various information about the model that is called by other functions.

There is more complexity to these functions than is listed above, and if you want to add your own model, we strongly encourage you to look at the same functions for other models as a template. Note that modelX should be replaced with whatever you want to call your model.

More will be added to this later and possibly template functions to create new models, if there is enough interest.

References

DeCarlo, L. T. (2002). Signal detection theory with finite mixture distributions: Theoretical developments with applications to recognition memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *109*(4), 710–721.

DeCarlo, L. T. (2003). An application of signal detection theory with finite mixture distributions to source discrimination. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *29*(5), 767–778.

Macmillan, N. A., & Creelman, C. D. (2005). Detection theory: A user's guide (2nd Edition). Mahwah, NJ: Lawrence Erlbaum Associates.

Mickes, L., Wixted, J. T., & Wais, P. E. (2007). A direct test of the unequal-variance signal detection model of recognition memory. *Psychonomic Bulletin & Review*, *14*(5), 858–865.

Snodgrass, J. G., & Corwin, J. (1988). Pragmatics of measuring recognition memory: applications to dementia and amnesia. *Journal of Experimental Psychology: General*, *117*(1), 34–50.

Yonelinas, A. P. (1994). Receiver-operating characteristics in recognition memory: Evidence for a dual-process model. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *20*(6), 1341–1354.

Yonelinas, A. P. (1999). The contribution of recollection and familiarity to recognition and source-memory judgments: A formal dual-process model and an analysis of receiver operating characteristics. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *25*(6), 1415–1434.

Yonelinas, A. P., & Parks, C. M. (2007). Receiver operating characteristics (ROCs) in recognition memory: A review. *Psychological Bulletin*, *133*(5), 800–832.