

Modular Algorithm Testbed Suite (MATS) User Manual

Naval Surface Warfare Center, Panama City Division



Last modified on: October 26, 2011 (v1.0.1)

Contents

1	Introduction	1
2	MATS Modules	1
3	Data Structures	2
3.1	The Input Structure	2
3.2	Performance Estimation	4
3.3	Contact List	5
4	Configuring the GUI	8
4.1	I/O Data	8
4.2	ATR Modules	10
4.3	Plotting Options	10
4.4	Other Input Parameters	11
4.5	Run-time Elements	13
4.6	Menu Options	13
5	Running MATS	15
6	Adding a Custom Module to MATS	16
6.1	Adding a Detector	16
6.2	Adding a Feature Extraction Method	17
6.3	Adding a Classifier	17
6.4	Adding Other Modules	18
6.5	Summary	18
6.6	Feedback	20
7	Frequently Asked Questions	20
A	Communication Between ATR and Operator Modules	22
A.1	Option #1: Modifying the Contact List with Subroutines	23
A.2	Option #2: Appending to the Feedback File	24
B	Notable Functions	28

1 Introduction

The Modular Algorithm Testbed Suite (MATS) is a software package for the development and testing of automatic target recognition (ATR) algorithms. It employs an open, modular architecture to provide developers with a quick and easy way to test and compare algorithms. MATS is written using the MATLAB[®] programming language and requires version 2009b or later and the Image Processing Toolbox to run properly.

To run MATS, copy the MATS package to a folder on the hard drive. Start MATLAB[®] and make the root folder of the MATS package the current working directory. This can be done either via the MATLAB[®] command prompt (indicated by `>>`) or by using the ‘Current Folder’ panel in the graphical interface. Then run the file `testbed_gui.m` by typing `testbed_gui` into the command prompt. At this point, the user should configure run-time parameters as described in Section 4.

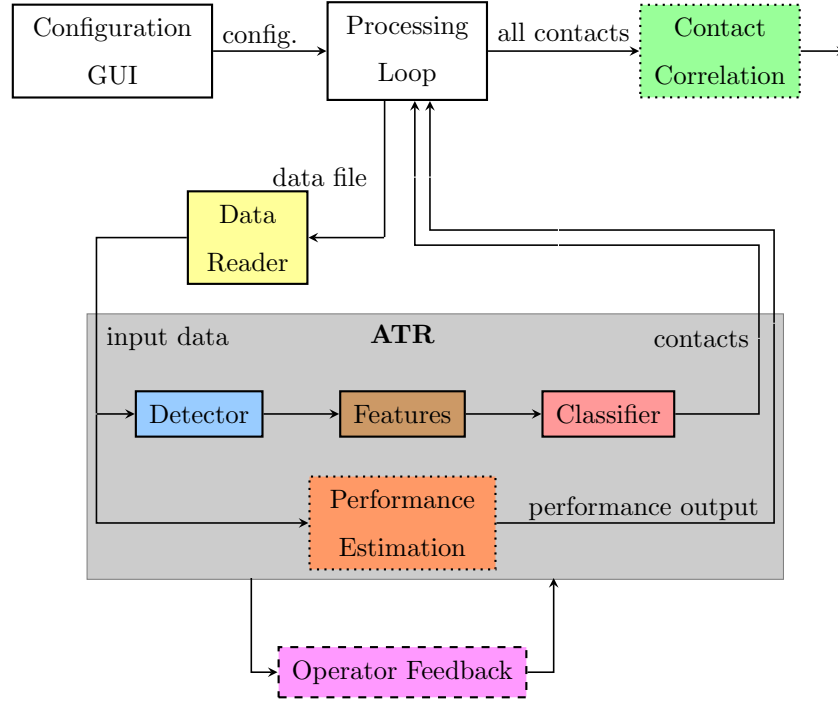
This document is intended to give developers of ATR algorithms an overview of how the MATS system works, how it can be used, and how additional modules can be integrated into it. Sections 2 and 3 describe the basic structure of MATS. Section 4 details the GUI configuration options and Section 5 describes expected run-time behavior. Procedures for integrating custom modules are included in Section 6. Additional information regarding feedback classifier integration is included in Appendix A.

2 MATS Modules

The bulk of the processing occurs in the individual components of the MATS system, which are called *modules*. Modules are organized into various categories, or *pools*, based on their function. Each pool has a specific and unique function to perform within the ATR processing stream:

- A ***data reader*** converts a data file from some format to the standardized MATS input structure detailed in Section 3.
- A ***detector*** takes in an image and returns a list of contacts representing objects of interest in the area covered by the image(s) provided.
- A ***feature extraction*** algorithm creates a feature vector to represent each contact for the purpose of classification.
- A ***classifier*** discriminates between targets and non-targets.
- A ***performance estimation*** algorithm estimates an expected level of performance of the ATR based on properties of the image.
- A ***contact correlation*** algorithm attempts to identify which contacts are actually the same object from different images.

Figure 1: MATS Flow Diagram.



All components within the same type perform similar tasks and are thus completely interchangeable, provided that the data standards defined in Section 3 are met. The specific algorithm to be run is determined at run time via the MATS graphical user interface (GUI) module selection menus, which are detailed in Section 4.2 .

3 Data Structures

The data structures described in this section are the means by which the different modules communicate with each other. The standardization here plays a crucial role in making the MATS suite a modular tool.

3.1 The Input Structure

The input structure contains all of the data that is passed into the ATR module. It represents an image and other data associated with an image (e.g., resolutions, vehicle data, etc.). The complete list of input structure fields is listed in Table 1 .

Table 1: Fields of the Input Structure.

Field	Data Type	Description
.hf	matrix (float)	High frequency (HF) image (complex)
.hf_cres	float	Cross-track (range) resolution of the HF image (in m)
.hf_ares	float	Along-track resolution of the HF image (in m)
.hf_cnum	int	Cross-track (range) dimension of the HF image (in pixels)
.hf_anum	int	Along-track dimension of the HF image (in pixels)
.bb	matrix (float)	Broadband (BB) image (complex)
.bb_cres	float	Cross-track (range) resolution of the BB image (in m)
.bb_ares	float	Along-track resolution of the BB image (in m)
.bb_cnum	int	Cross-track (range) resolution of the BB image (in pixels)
.bb_anum	int	Along-track resolution of the BB image (in pixels)
.side	string	side of vehicle from which the image was taken = {‘PORT’, ‘STBD’}
.lat	array (float)	latitude of vehicle (in degrees)
.long	array (float)	longitude of vehicle (in degrees)
.fn	string	filename of image file/some other image ID
.havegt	int	1 = ground truth data available; 0 = no ground truth available
.gtimage	array (struct)	ground truth info for this image (<i>Refer to Table 2 for details.</i>)
.targettype	string	type of target the detector is trying to detect ¹
.perfparams	struct	parameters required for the performance estimation code (<i>Refer to Table 3 for details.</i>)
.heading	array (float)	nominal heading of the vehicle (in radians)
.time	array (float)	time stamp (UTC)
.sensor	string	sensor ID string
.mode	char	run-time mode: ‘A’ = ATR only; ‘P’ = performance estimation only; ‘B’ = both
.sweetspot	array (int)	range pixel bounds on area of image suitable for ATR processing

¹Valid options include: ‘wedge’, ‘truncated cone’, ‘cylindrical’, ‘torpedo’, ‘sphere’, ‘bomb’, ‘box’, ‘can’, ‘coffin’, ‘disk’, ‘ellipsoidal’, ‘elongated sphere’, ‘fish’, ‘hemisphere’, ‘irregular’, ‘ovoid’, ‘rectangular box’, ‘sphere with horns’, ‘teardrop’, ‘triangular’, ‘unspecified’

Table 2: Fields of the .gtimage Substructure.

Field	Data Type	Description
.x, .y	arrays (int)	location of ground truth object (in pixels)
.side	string	side of vehicle where the ground truth object is located: ‘PORT’, ‘STBD’
.num	array (int)	identifier for ground truth object
.fn	string	filename of image file/some other image ID in which this ground truth object is located

Table 3: Fields of the .perfparams Substructure.

Field	Data Type	Description
.height	array (float)	height of the vehicle at each ping (in m)
.depth	array (float)	depth of the vehicle at each ping (in m)
.maxrange	float	maximum range (in m)

3.2 Performance Estimation

Table 4: Fields of the Performance Estimation Output Structure.

Field	Data Type	Description
.pdpc	matrix (float)	probability of detection and classification at each pixel (same size as input image)
.pfa	matrix (float)	probability of false alarm at pixel
.A	float	average effective range (whole image)
.B	float	average $P_d P_c$ (whole image)
.py	array (float)	average $P_d P_c$ vs. range (for image)

(Continued)

Field	Data Type	Description
.ATRstatus	char	stoplight performance indicator: <ul style="list-style-type: none"> • ‘r’ = red ($P_d P_c < .5$ OR $P_{fa} > .5$ OR number of positive contacts in current image > 10) • ‘y’ = yellow ($P_d P_c < .7$ OR $P_{fa} > .3$ OR number of positive contacts in current image > 5) • ‘g’ = green (otherwise)

3.3 Contact List

The contact list is an array that represents the output of the the ATR processing. Each element of the array represents an individual object and has the fields associated with it as listed in Table 5 . *(Note: the background color of the field entry coordinates with Figure 1 and indicates when the field values are first assigned.)*

Table 5: Fields of Elements the Contact List.

Field	Data Type	Description		
.x, .y	ints	location of contact in the image in pixels (centroid)		
.features	array (float)	features used to classify contact		
.fn	string	filename of image file/some other image ID		
.side	string	side of vehicle: {'PORT', 'STBD'}		
.hfsnippet	matrix (float)	area around contact in h.f. image (complex)		
.bbsnippet	matrix (float)	area around contact in b.b. image (complex)		
.gt	int	ground truth value of contact (if known): 1 = object really is a target; 0 = object really is not a target; empty = ground truth value unknown		
.lat	float	latitude of contact (in degrees)		
.long	float	longitude of contact (in degrees)		
Legend	Values from:	Input	Contact correlation	Feature E.M.
	Classifier	Detector	Performance estimation	Operator feedback

(Continued)

Field	Data Type	Description		
.class	int	1 = classified as relevant; 0 = classified as not relevant		
.classconf	float	probability that classification of contact is correct (0 < .classconf < 1)		
.groupnum	string	ID # of this contact's group; unique for each real object; many images may have objects with the same group #		
.groupclass	int	1 = group classified as relevant; 0 = group not classified as relevant		
.groupclassconf	float	probability that classification of group is correct		
.groupconf	float	probability that this contact belongs in this group (and represents the same object)		
.grouplat	float	latitude of group (in degrees)		
.grouplong	float	longitude of group (in degrees)		
.groupcovmat	matrix (float)	group lat/long covariance matrix (2x2)		
.detector	string	detector identifier		
.featureset	string	feature set identifier		
.classifier	string	classifier identifier		
.contcorr	string	contact correlation identifier		
.opfeedback	struct	operator feedback (see structure below)		
.heading	float	nominal heading of the vehicle (in radians)		
.time	float	timestamp (UTC)		
.alt	float	height of vehicle at contact ping		
.hf_cres	float	HF image cross-track resolution (in <i>m</i>)		
.hf_ares	float	HF image along-track resolution (in <i>m</i>)		
.hf_cnum	int	HF image cross-track dimension (in pixels)		
.hf_anum	int	HF image along-track dimension (in pixels)		
.bb_cres	float	BB image cross-track resolution (in <i>m</i>)		
.bb_ares	float	BB image along-track resolution (in <i>m</i>)		
.bb_cnum	int	BB image cross-track dimension (in pixels)		
.bb_anum	int	BB image along-track dimension (in pixels)		
.veh_lats	array (float)	latitude of vehicle		
Legend	Values from:	Input	Contact correlation	Feature E.M.
	Classifier	Detector	Performance estimation	Operator feedback

(Continued)

Field	Data Type	Description
.veh_long	array (float)	longitude of vehicle
.veh_heights	array (float)	height of the vehicle at each ping (in m)
.normalizer	string	normalizing algorithm used (empty string if none)
.bg_snippet*	matrix (float)	background snippet (complex)
.bg_offset*	array (int)	vector offset indicating where the background snippet is located relative to the contact
.hfraw*	matrix (float)	HF inverse image data (complex)
.bbraw*	matrix (float)	BB inverse image data (complex)
.lb1raw*	matrix (float)	Low Broadband (LB1) raw data (complex)
.hfac*	matrix (float)	HF acoustic color data (complex)
.bbac*	matrix (float)	BB acoustic color data (complex)
.lb1ac*	matrix (float)	LB1 acoustic color data (complex)
Legend	Values from:	Input
	Classifier	Detector
		Contact correlation
		Feature E.M.
		Performance estimation
		Operator feedback

Table 6: Fields of opfeedback Substructure.

Field	Data Type	Description
.opdisplay	int	operator display mode: 0 = don't show this contact 1 = show and confirm (likely a target; operator can veto) 2 = show and ask (unsure; operator must respond)
.opconf	int	operator confidence feedback: 1 = likely clutter 2 = less likely clutter 3 = unsure, could be either 4 = less likely a target 5 = likely a target

4 Configuring the GUI

The configurable GUI front end of the MATS suite (`testbed_gui.m`) allows the user to customize the input arguments of the system. It also allows for partial control over the run-time process via several adjustable parameters. Indented items are dependent on the value of the non-indented items above them. For example, the values selected for the plotting options are irrelevant if showing the images is disabled.

4.1 I/O Data

This portion of the GUI requires the user to choose which data file(s) to process and where the results should be stored.

Source Directory

The source directory is the folder containing the data files to be processed. Each file in the folder should be of the same file format. This is a required field, unless using preprocessed detections.

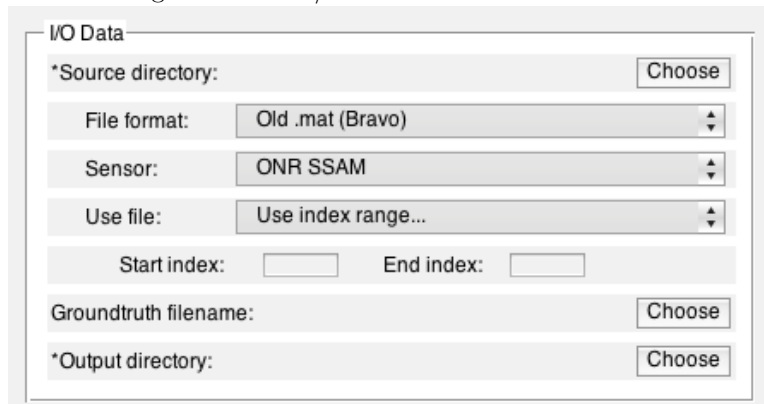
File Format Menu

The selection of the file format determines how MATS will attempt to read the selected file(s). All supported file formats are listed in the format selection menu.

Sensor Menu

The ‘Sensor’ selection modifies the run-time parameters of algorithms. Due to differences in the sensors, a particular configuration for an algorithm may not work on all sensors, as algorithms are incompatible with certain sensors. It is assumed that if the user is trying to use an algorithm, he knows enough about it to make the appropriate selection. As such, the MATS GUI will not prevent you from choosing an algorithm that does not work with the selected sensor format, or vice versa. Algorithm developers should use this value,

Figure 2: The ‘I/O Data’ Section of the GUI.



I/O Data

*Source directory:

File format:

Sensor:

Use file:

Start index: End index:

Groundtruth filename:

*Output directory:

Figure 3: The ‘ATR Modules’ Section of the GUI.

The screenshot shows a window titled "ATR Modules" with the following settings:

- ☒ Manual detector/classifier selection
 - Detector: HFBB_CCA
 - Features: Isaacs
 - Classifier: Isaacs
 - Class. Data File: data_JI_SSAM1_default.mat
- ☒ Skip performance estimation
 - Performance Model: ATRTTSP
- ☐ Use contact correlation
 - Algorithm: Cobb

found in the `.sensor` field of the input structure, to only process data that their algorithm is equipped to properly handle.

File Selection Menu

The file selection menu lists the valid data files found by the MATS GUI. The items shown in this list depend on what file format is selected; changing the file format will update the list of available files. The user may select an individual file for processing, or process a range of files. Choosing ‘Use index range...’ will automatically fill the index selection fields below with the proper indices required to process all of the files in the list, but the user may manually change these if necessary.

(Note: Currently, the file extension determines if a file is of the right “type”. Therefore, it is possible for other files of a different “type” to show up in this list if they have the same file extension. Store only one type of data in a directory for best results.)

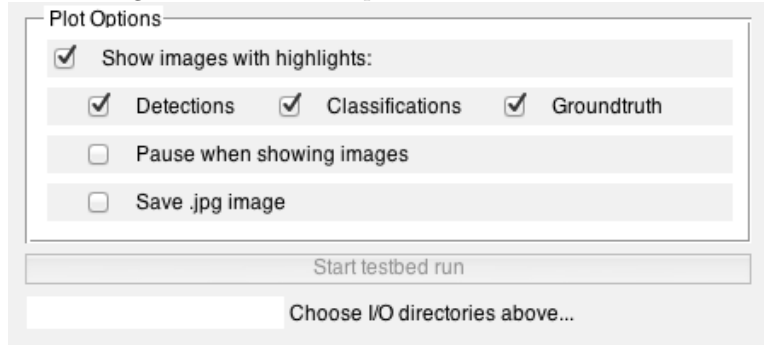
Ground Truth File

The ground truth file contains information relating to the true positions of the objects in the data. Having a ground truth file is not required for MATS execution.

Output Directory

The output directory is where the output files will be stored after the ATR processing of an image has been completed. This is a required field in all circumstances. *(Note: When the new output files are saved, they will overwrite any previous results within this directory that have the same filenames.)*

Figure 4: The ‘Plot Options’ Section of the GUI.



4.2 ATR Modules

In this section of the GUI, the user can choose which modules are executed during the ATR process via several check boxes and selection menus.

The first check box toggles the manual selection of the detector, feature generator, and classifier modules. When this is enabled, the module listed in a given field will be executed during the ATR process; otherwise, the default algorithms will be used. The GUI will automatically detect all modules within the directory structure that follow the specifications for adding a custom module described in Section 6 .

(Note: For the publicly released version of MATS, the default algorithms (i.e., ‘Test’) are placeholders that determine results at random. They are intended for illustrative purposes only.)

There is also a selection menu for classifier parameter files. Different parameter files can be created for different classification scenarios as necessary.

The other check boxes toggle the optional performance estimation and the contact correlation algorithms. When either of those are enabled, the module listed in the selection menu will be executed; otherwise, no algorithm of that type will be executed.

4.3 Plotting Options

This section of the GUI customizes the appearance and behavior of the image display window that MATS produces after processing an image.

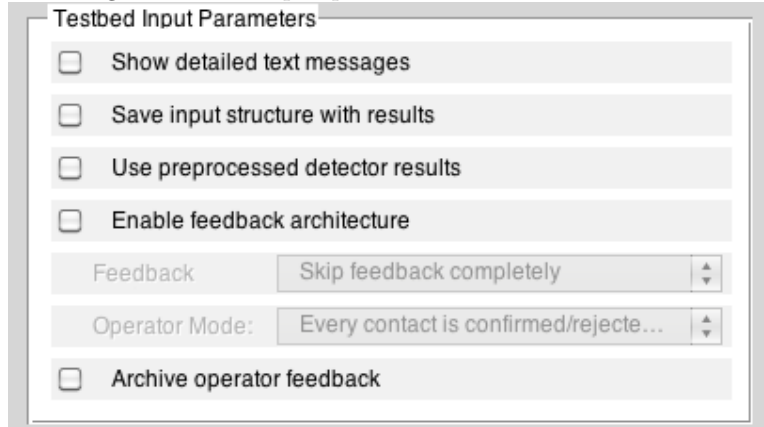
Show Images with Highlights

Enabling this option will display the image after the processing is complete.

Highlight Options

Here there are three check boxes, one each for detections, classifications, and ground truth elements. Each check box, when enabled, will cause a colored box to appear around the items in the image designated by its

Figure 5: The ‘Input parameters’ Section of the GUI.



label. The color blue represents ground truth, green represents detections, and red represents classifications. *(Note: If a contact is classified as a target, and both detections and classifications are to be highlighted, only the classification box will be visible. This is because the classification box is placed on top of the detection box.)*

Pause While Showing Images

Enabling this option pauses code execution after displaying the image and the selected highlights. MATS will not resume processing until the user clicks on the continue button in the image display window.

Save Images

Enabling this option will save a copy of the image shown, with highlights, into the designated output directory.

4.4 Other Input Parameters

There are several other parameter switches that can be used to modify the behavior of the code:

Show Detailed Text Messages

This option displays more text on the command line and is intended largely for debugging purposes. The amount of text that is displayed without this option enabled is sufficient to monitor the progress of the code’s execution.

Save Input Structure With Results

Enabling this option will save the input structure along with the image’s contact list and the results from the performance estimation. *(Note: This makes the stored output data file large).*

Use Preprocessed Results

When this option is enabled, the user will be prompted for a folder containing previous results after starting the run. For each file in the input folder, the corresponding I/O file will be loaded instead of running the input data through the detector. This will significantly decrease the amount of time required to process an image and can be useful when testing classifiers. Enabling this option will also “gray out” GUI fields relating to data input.

Enable Feedback Architecture

Enabling the feedback architecture is not necessary for non-feedback classifiers. Doing so will only execute unnecessary read/write commands that will slow the processing of data. There are two selection submenus that configure the feedback behavior of the testbed:

Feedback Mode: determines how operator feedback is entered into the testbed

- ‘Use feedback GUI’: uses a [very] basic GUI to view contacts and provide operator feedback.
- ‘Simulate operator feedback w/ archive’: uses a saved file containing operator feedback from a prior run.
- ‘Skip feedback completely’: no feedback input. This is primarily used in situations where some other program (i.e., an external GUI) will collect the feedback from the user.
- ‘Simulate operator feedback w/ ground truth data’: uses ground truth information to determine what a hypothetical operator would say. *(Note: If you haven’t actually selected a ground truth file, this will crash the testbed program.)*

Operator mode: determines how a lack of operator feedback is handled by the testbed

- ‘Every contact is confirmed/rejected by operator’: the operator cannot skip contacts
- ‘Interpret no operator comment as implicit agreement’: when an operator skips a contact, he agrees with the ATR classification
- ‘Use only explicit operator calls’: when an operator skips a contact, no feedback will be used.

(Note: Selecting a classifier from the selection menu will automatically change the state of the check box to its proper value; this generally should not be manually overridden by the user.)

Archive Operator Feedback

This option allows for the reuse of operator feedback. When feedback archiving is enabled, the operator feedback will also be written to a text file where it can be reused in a later run.

4.5 Run-time Elements

The bottom-most region of the MATS GUI is composed of a start button, a progress bar, and a message area.

The start button is initially “grayed out” and will remain unclickable until the output directory and, if not using previously obtained results, the data source directory are selected. The remaining items in the GUI will have usable default values.

The main function of the message area is to track the progress of the run along with the progress bar. However, before the run begins, it displays messages to help guide the user toward a complete configuration.

4.6 Menu Options

The preceding sections have described features of the MATS GUI that affect the way the ATR is performed on an image. The GUI has a few other features that do not, and are located in the menu. They are entirely optional but may still benefit the user.

Save/Load Configuration

The GUI allows you to save your run-time configuration for use at a later time via the menu, **Configuration** → **Save**. Loading a configuration can be done similarly via **Configuration** → **Load**. This feature can be used to establish a consistent run-time environment.

Classifier Training

MATS also has mechanisms in place to facilitate classifier training via **Configuration** → **Retrain Classifier**. Given the results for a data set, the GUI can extract the features and labels from the data and pass them into a training function, which the classifier developer must provide. (*Note: See Section 6.3 for more information about what this entails.*)

Ground Truth Analysis

Select **Analysis** → **Ground Truth Analysis** to launch the ground truth comparison tool. Once a folder containing results files (i.e., of the form `IO_*.mat`) is chosen via the folder selection window, the contacts in that folder will be loaded into memory and its contents will be compared to the selected ground truth file. (*Note: A ground truth file must be selected in the GUI in order to use this function.*)

The final printout lists all of the contacts’ locations from that folder, along with all of the ground truth entries from the designated ground truth file. The data is organized such that each group of text corresponds to one image’s worth of data. When a ground truth entry matches a contact, its information will appear in the same line of text as that of the matching contact. Any unmatched ground truth entries will appear at

Figure 6: Excerpt from a sample ground truth analysis.

```

ImageHF-10-7, PORT

C#163 @ ( 400, 301) ----- .....No match.....
x C#164 @ (2401, 650) ----- GT#122 @ (2401, 650)
x C#165 @ (3208,1151) ----- GT#123 @ (3201,1150)
x C#166 @ (2022,1401) ----- GT#124 @ (2001,1400)
.....No match..... ----- GT#121 @ (1601, 900)

```

the end of the group corresponding to the image in which it is located. Contacts that were classified as mines are indicated by an X at the beginning of the line. A sample excerpt of this process is shown in Figure 6 .

At the end of the printout, all of the ground truth entries from the ground truth file that were not matched with contacts from the ATR are listed, along with some summary statistics. The list of unmatched ground truth entries does not include entries corresponding to data files that were not actually processed.

ROC Curves

Once a run has been completed, a receiver operator characteristic (ROC) curve may be generated by selecting **Analysis** → **ROC Curve Analysis** → **Generate ROC Curve** from the menu, and choosing the folder where the results files were stored. The `ROC_*.mat` files will be read in turn, and a ROC curve will be generated using all of the data up until the current file, for each file in the list. This shows how the ROC curve is changing over time as more data has been processed. The ROC curve generated displays averaged probability of classification versus averaged probability of false alarm. (*Note: This functionality requires the MATLAB[®] Statistics Toolbox, the 'perfcurve' function in particular.*)

Various runs using different detection and classifier algorithms can be compared by generating ROC curves for each run which looks only at performance against targets. This comparison can be generated by selecting **Analysis** → **ROC Curve Analysis** → **ROC Compare** from the menu. It will ask the user to choose the folder where the results files were stored for the first algorithm and then plot the corresponding curve. It will then ask for the results files for the second algorithm and plot the corresponding curve on the same axes. It will continue to ask the users for files and adding curves until 'Cancel' is pressed. In order for correct results the user must select a ground truth file in the main MATS GUI and the proper file format. The ROC curves from this function compare probability of detection / probability of classification versus false alarms per image. The function removes double calls and excessive non-target calls which can falsely boosts an algorithm's performance to accurately compare algorithms.

Confusion Matrix

A confusion matrix can be generated for a chosen folder via **Analysis** → **Generate Confusion Matrix**. This will summarize the numbers of correctly identified objects, correctly rejected non-objects, misses, and false alarms and the column sums are targets and non-targets. (*Note: This functionality requires the MATLAB[®] Statistics Toolbox.*)

Display Formatted Contact Data

This tool provides a convenient way to display the locations of a list of contacts in an easily legible format. Select **Analysis** → **Print Sorted Contact List**, and then one of the remaining options (pixel coordinates or latitude/longitude pairs). When prompted, choose a folder containing results files. The contacts from all of these files will be aggregated into a master contact list, and the contacts' locations will be displayed along with classification and classifier confidence at the MATLAB[®] prompt. The entries will be sorted with decreasing values of classifier confidence (shown as **CLASSCONF**).

Documentation

This file may be accessed via MATS by selecting **Help** → **MATS Documentation** from the GUI menu.

5 Running MATS

Once the run button is clickable and not “grayed out”, MATS is ready to run. Clicking the run button will initiate a loop that will process all of the selected images in turn. As the code runs, status messages will be printed to the MATLAB[®] prompt. Before each module is run, there will be a message announcing which module is starting. When the module finishes, another message will display the amount of time elapsed within that module. These two sets of messages can help the user monitor the progress of an individual data file through the ATR. The progress bar in the GUI will advance as the files are processed in turn.

When an image has completed processing, the results (and possibly the input structure, if enabled via the GUI) are saved in a file called `IO_[INPUT_FNAME].mat` where `[INPUT_FNAME]` is the name of the source data file without its extension. Similarly, a smaller file `ROC_[INPUT_FNAME].mat` will be created that can be used in the ROC curve feature discussed in Section 4.6.

After all of the selected images have been processed, contact correlation will be run if contact correlation is enabled in the GUI configuration. Finally, if a ground truth file had been entered into the GUI before execution, the ground truth analysis function prints a summary of the final contact list and ground truth information. The ground truth analysis function is also discussed in more detail in Section 4.6.

6 Adding a Custom Module to MATS

The MATS suite has subroutines to automatically import available modules from the testbed directory structure when the GUI initializes. To effectively integrate a new algorithm, files must be created in the proper location so that when the GUI looks for available modules, the new algorithm will be found. The integration process is similar for all module classes, although there are some notable differences which will be covered in the subsequent sections. The next section will detail the integration of a detector.

6.1 Adding a Detector

Image Normalization

A subroutine for image normalization (`ATR_Core\normalize_images.m`) is included to assist in module development. If this function is called, the `.normalizer` field of the contact list should be set to `'NSWC'`. If a different function is used, a short string identifier should be used instead. If no image normalization is performed, use an empty string (i.e., `''`).

Copy Code into the Directory Structure

In order for the new module to appear in the appropriate selection menu, it must reside in the proper place in the directory structure. Each module type (e.g., detectors) has a dedicated subdirectory which contains all of the modules of that class, contained in their own subdirectories. In order to access the new module from the GUI, create a new folder (the *module root directory*) of the form `[MATS_ROOT]\Detectors\[SUBDIR]`, where `[MATS_ROOT]` is the root-level folder of the MATS package (i.e., the folder in which the GUI is located), and `[SUBDIR]` is a descriptive string identifying the module. Because this string is part of the path of the module's files, it should be kept fairly short for convenience.

Create the Main Function

In the module root directory, create a file `det_[SUBDIR].m`. This function is the entry point into the new module. The GUI will search for this file specifically, and execute it when necessary if it is selected in the detector module menu. Note that the file name must include the same spelling, including case, as the subdirectory in which it resides. This file must take in one input (an input structure) and return one output (a contact list). The new detector should only operate on familiar sensor types; if an unrecognized sensor is chosen, it should produce an error to prevent producing potentially misleading results. A detector should also initialize all of the fields in the contact list to default values before returning them. An example of this is shown in the file `Detectors\Test\det_Test.m`.

Create a Description File

Also in the root directory, create a *description file* named `about.txt`. Each detector, feature generator, and classifier has a file like this that describes what is required for the successful execution of the module. For an example of how to create this for a detector, refer to `Detectors\Test\about.txt`. Entry descriptions can be found in Table 8 in Section 6.5 . Detectors only require the `module_tag` field.

The integration is now complete. Run `testbed_gui.m` and your new detector should automatically be in the Detector drop down menu on the GUI.

6.2 Adding a Feature Extraction Method

Adding a feature extraction method is slightly more complicated in that the description file has more required fields; however, the basic process is the same. In this case, the module root directory should be created in the **Features** subdirectory, and the main function to be called should be `feat_[SUBDIR].m` which has one input and one output, both of which are the contact list.

6.3 Adding a Classifier

Similar modifications to the procedure are required for classifier integration. The main function `cls_[SUBDIR].m` should be created in the new folder `[MATS_ROOT]\Classifiers\[SUBDIR]`. The parameters to the function should be the same as for the feature case: one input, one output, both as a contact list. However, since classifiers can be trained on different feature sets, an additional step is required to accommodate this.

Classifier Parameter Files

At least one *classifier parameter file* must also be created. Each of these files should contain a version of all the configurable parameters that could vary among parameter files. There are no constraints placed on the internal organization of this file; MATS will only pass the file name selected from the GUI selection menu entitled ‘Class. data file’ into the classifier upon execution. (Refer to Section 4.2 for more information about the GUI). The default data file should be indicated with `default` in the file name somewhere.

Training Functions

In addition, a classifier may also have a training function to assist in creating the classifier parameter files. The training function should have a function definition of the form `trn_[SUBDIR](labels, features, data_fname)`, where `labels` is a $1 \times N$ row vector containing ground truth information for N contacts, `features` is an $L \times N$ matrix containing the feature vectors of these contacts as columns, and `data_fname` is the name that

Figure 7: Sample Description File.

```
module_name: Test Classifier
module_tag: TEST
uses_feedback: 0
```

the new classifier data file will have. `labels` and `features` will be extracted from prior results (i.e., the `I0_*.mat` files from a prior run).

6.4 Adding Other Modules

To integrate performance estimation and contact correlation algorithms, follow the procedure outlined in Section 6.1 with the changes listed in Table 7 . A description file is not required for either of these module classes.

6.5 Summary

For any module, a new subdirectory (the *module root directory*) must be created within the proper folder for its module class. All of the module’s code must be copied into the module root directory. In addition, a main function with the appropriate prefix must be created in order for the GUI to recognize the new module as a valid choice. Furthermore, each detector, feature generator, and classifier has a *description file* called `about.txt` located in its root directory. This file describes what is required for the successful execution of the module in the manner shown in Figure 7 . The GUI parses this file, extracts the necessary information from it, and modifies the run-time configuration appropriately. All of these options are summarized in Table 7 and Table 8 . (Note: For the binary flags in Table 8 , either 1/0 or ‘yes’/‘no’ will be properly interpreted by the GUI.)

Table 7: Summary of Integration Parameters.

Class	Subdirectory	Prefix	Need DF?
Detector	Detectors\	det_	X
Feature Ext. Method	Features\	feat_	X
Classifier	Classifiers\	cls_	X
Performance estimation	Performance_Estimation\	perf_	
Contact correlation	Contact_Correlation\	cor_	

Table 8: Required Entries in Description Files.

Det.	Feat.	Class.	Description File Entry
X	X	X	<code>module_name</code> : character string indicating the name of the module.
X	X	X	<code>module_tag</code> : short character string (3-4 alphanumeric characters) that will appear in the corresponding module identifier strings (e.g., <code>.detector</code>).
	X		<code>reqs_inv_img</code> : binary flag indicating whether this feature extraction method requires inverse imaging.
	X		<code>reqs_bg_snippet</code> : binary flag indicating whether this feature extraction method requires background snippets.
	X		<code>feat_mode</code> : string indicating how features are handled. ‘append’ adds onto the existing features, if any were created by the detector; ‘overwrite’ overwrites them.
		X	<code>uses_feedback</code> : binary flag indicating whether this classifier can use operator feedback.

6.6 Feedback

If the classifier to be added to the testbed has a feedback element (i.e., it requests data from the user/operator in order to improve its performance), additional steps must be taken to ensure successful execution. In this scenario, it is assumed that an operator process is running in parallel, so that the operator and ATR processes do not have to wait on each other.

The classifier can query the operator via the `.opfeedback.opdisplay` fields of the contacts in the contact list. By setting these to a positive value the classifier can indicate that the operator should confirm or reject a given mine. In response, the operator rates the contacts in turn on a five-point scale, with a value of five corresponding to a confident mine classification; three, an ambiguous case; and one, a confident non-mine classification. These ratings are stored in `opfeedback.opconf`.

Once the operator has rated a contact, this information needs to be relayed back to the testbed where it can inform the classifier. Since the testbed is already busy processing the next image at this point, this communication must happen indirectly. The operator decisions are written to a dedicated *feedback file*, the contents of which will be read during the next iteration of the ATR processing. The information contained in the feedback file will be incorporated into the contact list, and then the feedback file will be cleared of any data.

A classifier should never attempt to reclassify a contact that has been viewed by the operator. After the operator has evaluated an image worth of contacts, it is assumed that he has made any necessary corrections to erroneous calls. The last contact that the operator has viewed can be easily determined by finding the latest negative value for `.opdisplay`.

7 Frequently Asked Questions

1. *I see messages saying that 'bkuplock.txt' and/or 'bkupedit.txt' cannot be opened. Is that bad?*

Probably not. At the beginning of the run, `bkuplock.txt` and `bkupedit.txt` will not exist. `bkuplock.txt` will not be written to until an operator feedback action takes place. If you aren't using a feedback classifier, or just haven't submitted feedback on contacts yet, messages indicating that those files do not exist or cannot be opened are expected.

Appendices

A Communication Between ATR and Operator Modules

In order to successfully integrate a feedback classifier, some additional details about the inner workings of the MATS code must be understood. Otherwise, unexpected behavior may result, and the causes may be difficult to identify.

The mechanism by which the feedback is acquired from the operator is called the *operator process* and runs in parallel with the main ATR process. The two processes must communicate with each other via common files. The operator process can check periodically for new output files (i.e., new `IO_*.mat` files located in the output directory) to get information about new contacts that have been recently created by the ATR process. After obtaining the operator's feedback on the new contacts, the operator process should record the feedback in order to pass it on to the classifier. This can be done in one of two ways: 1.) by running a MATLAB[®] function to edit the saved contact lists directly (Section A.1), or 2.) by appending the feedback data to the feedback file (Section A.2).

The cumulative contact list is saved across two files. The *locked file* contains all of the contacts for which the operator has given feedback, or has had the opportunity to give feedback but has declined to do so. The remaining contacts are stored in the *unlocked file*. Thus the unlocked file contains new contacts as well as backlogged contacts that the operator has not yet seen. At the end of the ATR function, newly evaluated contacts (as determined by changes in their `.opfeedback.opconf` values) migrate to the locked file, and the unlocked file is updated to both reflect this movement and to include new contacts from the current image.

Internally, the main ATR function tracks two indices that determine how a contact's data may be modified. The *last locked index* is the index of the last contact residing in the locked file. This index can be found by locating the last contact with a non-negative `.opfeedback.opconf` value towards the beginning of the contact list. Because normally data is only appended to the locked file, any further changes to the contacts stored in the locked file will not be recognized by the ATR; instead, the ATR will continue to read the version that was previously written in the locked file.

The *last viewed index* is the index of the last contact that the operator has had the opportunity to see. What this means depends on how the feedback is acquired. If the contacts are presented sequentially, then the last viewed index corresponds to the index of the last contact presented. If the operator sees all of the contacts from one image at a time, then this would be the latest index of the contacts on the screen, regardless if the operator has really examined or commented on that contact. The contact at the last viewed index is the last contact with a negative value of `.opfeedback.opdisplay` towards the front of the list. (Alternatively, it is the contact before the earliest one with a non-negative value of `.opfeedback.opdisplay`.) When the ATR runs, any contact that has not been viewed will be passed into the classifier for classification.

Figure 8 illustrates these concepts for a sample contact list. In this scenario, all contacts from the same image are viewed simultaneously. Each column in the array represents a contact, and each row represents an operator feedback field (i.e., `.opdisplay` or `.opconf`). The contents of each cell indicates the sign of the

Figure 8: MATS Operator Feedback Scheme.

	last viewed index							
.opdisplay	-	-	-	-	-	-	+	0
.opconf	+	0	-	0	+	+	-	-
	last locked index							

feedback field listed on its row for that contact. In this example, `contacts(2).opfeedback.opconf` is zero, and `contacts(2).opfeedback.opdisplay` is negative. The locked contacts are shaded red; the unlocked contacts are shaded either white or blue. The blue contacts represent contacts from future images that have already been processed by the ATR, but have not been viewed by the operator, and are hence able to be reclassified. The white contacts are currently on the operator's screen. Note that if the contacts were being evaluated sequentially, the white region would consist of at most one contact. The last locked index and last viewed index are the indices of the last contact in the red and white regions, respectively. By modifying the operator feedback fields, these indices advance and the contacts' data becomes more protected.

A.1 Option #1: Modifying the Contact List with Subroutines

The simplest way to communicate the operator feedback to the ATR process is to call several MATLAB[®] subroutines located in the `ATR_Core` subdirectory of the MATS library. This method will edit the saved contact lists directly.

1. Read value of `contacts(k).opfeedback.opdisplay`. If this value is positive, then feedback is requested for this contact. (*Note: Refer to Table 6 for more information.*)
2. Query the operator to acquire feedback (i.e., `contacts(k).opfeedback.opconf`). MATS currently supports a five-point scale, with a five likely being a target and a one likely being clutter.
3. Wait while the file `bkup_atr_busy.flag` exists using the `wait_if_flag` function. When `bkup_atr_busy.flag` is present, the ATR process is busy modifying the locked and unlocked files.
4. Create `bkup_gui_busy.flag` with the `write_flag` function. Similarly, the presence of `bkup_gui_busy.flag` informs the ATR process that the operator is about to change the contents of the locked file and/or the unlocked file. The ATR process will then wait until the contact list files are again available before trying to access them.
5. Check if any new contacts are available by searching for new IO files in the output directory. A new file in the directory indicates that the contact list may have been changed.

6. If new contacts are available, load the latest version of the contact list using the `read_backup` function.
7. Edit the contact list to incorporate operator confidence (`.opconf`).
8. Save contact list via the `write_backups` function.
9. Delete `bkup_gui_busy.flag` with `delete_flag`. Failure to delete `bkup_gui_busy.flag` will cause the ATR to hang indefinitely!

Notes:

- When a contact is viewed, 10 should be subtracted from its `.opfeedback.opdisplay` value, and the saved lists should be updated using Steps 3-9. This will prevent contacts that are currently being viewed from being reclassified.
- Details of the functions mentioned above can be found in Appendix B.
- The ATR process stores the locked contacts in `[TB_ROOT]\bkuplock.txt` and the unlocked contacts in `[TB_ROOT]\bkupedit.txt`. Use these values when calling `read_backups` or `write_backups`.

A.2 Option #2: Appending to the Feedback File

An alternative approach is to write the feedback to the feedback file. During each execution of the ATR, all contents of the feedback file are read, incorporated into the contact list, and finally purged from the feedback file. Thus, the feedback file contains all of the operator feedback that has accumulated since the last time the ATR checked that file.

Writing to this file involves low-level writing operations and requires the writing operations to perfectly match future read operations, whereas Option #1 handles this automatically behind the scenes. However, the benefit of this approach is that it does not require any particular programming language in order to work.

The process for effectively relaying feedback for the k th contact to the ATR process via the feedback file may be summarized as follows:

1. Read value of `contacts(k).opfeedback.opdisplay`. If this value is positive, then feedback is requested for this contact. (*Note: Refer to Table 6 for more information.*)
2. Query the operator to acquire feedback (i.e., `contacts(k).opfeedback.opconf`). MATS currently supports a five-point scale, with a five likely being a target and a one likely being clutter.
3. Wait while the file `opfb_atr_busy.flag` exists using the `wait_if_flag` function. When `opfb_atr_busy.flag` is present, the ATR process is busy modifying the contents of the feedback file.

4. Create `opfb_op_busy.flag` with the `write_flag` function. Similarly, the presence of `opfb_op_busy.flag` informs the ATR process that the operator is about to change the contents of the feedback file and will cause it to wait until the feedback file is again available.
5. Append operator confidence data to feedback file `feedback.txt` in the proper format. This is detailed in the next section.
6. (Optional) Increment counter stored in `feedback_cnt.txt`. This is only necessary if periodic updating of the classifier is desired. The ATR will reset this value automatically after incorporating feedback from the feedback file into the list.
7. Delete `opfb_op_busy.flag` with `delete_flag`. Failure to remove `opfb_op_busy.flag` will cause the ATR to hang indefinitely!
8. (Optional) Periodically update the classifier.

Appending Data to the Feedback File

In order to properly relay the operator feedback to the classifier, data must be appended to the feedback file in the proper format. The amount of data to be appended depends on what kind of action the operator has taken. In the case of a *verify operation*, where the operator rates an existing contact, not much information is needed because an entry for this object already exists in the contact list. However, for *add operations*, when the operator manually adds a contact to the contact list that the ATR did not detect, additional data is required to fill in the data fields that would have been filled by the ATR. The following data for an arbitrary contact are always written in this order:

- **ID** (uint16) – unique identifier for each contact in the contact list. Starts at 1 and increments with each new contact added to the list.
- **est_index** (uint16) – *For verify case:* current index at the time of review; in most cases, this is the index, but in the event of a manual contact insertion, this contact may have shifted to a later position. *For add case:* index of insertion into the array
- **type** (uchar) – either ‘V’ (verify contact) or ‘A’ (add contact manually)
- **len_fname** (uint8) – length of filename
- **filename** (1 x `len_fname`, uchar) – name of the file in which this contact is located
- **side** (1 x 4, uchar) – either ‘PORT’ or ‘STBD’
- **len_sensor** (uint8) – length of sensor name

- `sensor` (1 x `len_sensor`, uchar) – name of sensor used (e.g., ‘ONR SSAM’)
- `x` (uint16) – x location of contact (in pixels)
- `y` (uint16) – y location of contact (in pixels)
- `opdisplay` (int8) – new value of `opdisplay`. Values are shifted by -10 to indicate that this contact has been viewed by the operator (i.e., `contacts(index).opfeedback.opdisplay - 10`).
- `opconf` (int8) – operator confidence rating (1-5)

In addition, these fields must be written when *adding* contacts:

- `hf_rows` (uint16) – number of rows in high frequency snippet
- `hf_cols` (uint16) – number of columns in high frequency snippet
- `hf_reals` (`hf_rows` x `hf_cols`, float32) – real part of complex HF snippet
- `hf_imags` (`hf_rows` x `hf_cols`, float32) – imaginary part of complex HF snippet
- `bb_rows` (uint16) – number of rows in broadband snippet
- `bb_cols` (uint16) – number of columns in broadband snippet
- `bb_reals` (`bb_rows` x `bb_cols`, float32) – real part of complex BB snippet
- `bb_imags` (`bb_rows` x `bb_cols`, float32) – imaginary part of complex BB snippet
- `lat` (float32) – latitude of added contact
- `long` (float32) – longitude of added contact
- `heading` (float32) – heading of added contact
- `time` (float32) – time stamp of added contact
- `altitude` (float32) – altitude of added contact
- `hf_ares` (float32) – along-track resolution of original HF image
- `hf_cres` (float32) – cross-track/range resolution of original HF image
- `hf_anum` (uint16) – along-track size of original HF image
- `hf_cnum` (uint16) – cross-track/range size of original HF image
- `bb_ares` (float32) – along-track resolution of original BB image

- `bb_cres` (float32) – cross-track/range resolution of original BB image
- `bb_anum` (uint16) – along-track size of original BB image
- `bb_cnum` (uint16) – cross-track/range size of original BB image
- `len_vlats` (uint16) – length of vehicle latitude data
- `veh_lats` (`len_vlats` x 1, float32) – vehicle latitude data
- `len_vlongs` (uint16) – length of vehicle longitude data
- `veh_long` (`len_vlongs` x 1, float32) – vehicle longitude data
- `len_vheights` (uint16) – length of vehicle height data
- `veh_heights` (`len_vheights` x 1, float32) – vehicle height data

Furthermore, extra data can be filled in for these optional fields. These can be bypassed (as in NSAM) by writing zeros for the dimension values and skipping the write operations for the matrices.

- `bg_snippet_rows` (uint16) – number of rows in background snippet
- `bg_snippet_cols` (uint16) – number of columns in background snippet
- `bg_snippet_reals` (`bg_snippet_rows` x `bg_snippet_cols`, float32) – real part of background snippet
- `bg_snippet_imags` (`bg_snippet_rows` x `bg_snippet_cols`, float32) – imaginary part of background snippet
- `bg_offset`(1 x 2, int16) – vector offset indicating where the background snippet is located relative to the contact
- `hfraw_rows` (uint16) – number of rows in HF inverse image data
- `hfraw_cols` (uint16) – number of columns in HF inverse image data
- `hfraw_reals` (`hfraw_rows` x `hfraw_cols`, float32) – real part of HF inverse image data
- `hfraw_imags` (`hfraw_rows` x `hfraw_cols`, float32) – imaginary part of HF inverse image data
- `bbraw_rows` (uint16) – number of rows in BB inverse image data
- `bbraw_cols` (uint16) – number of columns in BB inverse image data
- `bbraw_reals` (`bbraw_rows` x `bbraw_cols`, float32) – real part of BB inverse image data
- `bbraw_imags` (`bbraw_rows` x `bbraw_cols`, float32) – imaginary part of BB inverse image data

- `lb1raw_rows` (uint16) – number of rows in lb1 raw data
- `lb1raw_cols` (uint16) – number of columns in lb1 raw data
- `lb1raw_reals` (lb1raw_rows x lb1raw_cols, float32) – real part of lb1 raw data
- `lb1raw_imags` (lb1raw_rows x lb1raw_cols, float32) – imaginary part of lb1 raw data
- `hfacs_rows` (uint16) – number of rows in hf acoustic color data
- `hfacs_cols` (uint16) – number of columns in hf acoustic color data
- `hfacs_reals` (hfacs_rows x hfacs_cols, float32) – real part of hf acoustic color data
- `hfacs_imags` (hfacs_rows x hfacs_cols, float32) – imaginary part of hf acoustic color data
- `bbacs_rows` (uint16) – number of rows in bb acoustic color data
- `bbacs_cols` (uint16) – number of columns in bb acoustic color data
- `bbacs_reals` (bbacs_rows x bbacs_cols, float32) – real part of bb acoustic color data
- `bbacs_imags` (bbacs_rows x bbacs_cols, float32) – imaginary part of bb acoustic color data
- `lb1acs_rows` (uint16) – number of rows in lb1 acoustic color data
- `lb1acs_cols` (uint16) – number of columns in lb1 acoustic color data
- `lb1acs_reals` (lb1acs_rows x lb1acs_cols, float32) – real part of lb1 acoustic color data
- `lb1acs_imags` (lb1acs_rows x lb1acs_cols, float32) – imaginary part of lb1 acoustic color data

B Notable Functions

`testbed_gui`

`function testbed_gui()` - Main function for the MATS GUI. No inputs or outputs.

`wait_if_flag`

`function wait_if_flag(flagname)` - Causes execution to wait while the file(s) defined in `flagname` exist.

- `flagname` = either a string representing the filename of the flag to wait for, or a cell array containing multiple such strings.

write_flag

`function write_flag(filename, msg_on)` - Writes a tiny, empty file whose sole purpose is merely to exist. This is used to inform processes of which files are currently being used.

- `filename` = the name of the file to be created.
- `msg_on` = 0 or 1, enables debugging messages.

delete_flag

`function delete_flag(filename, msg_on)` - Deletes a flag file.

- `filename` = the name of the file to be deleted.
- `msg_on` = 0 or 1, enables debugging messages.

read_backups

`function c_list = read_backups(fn_l, fn_u, msg_on)` - Reads the backup files specified in `fn_l` and `fn_u` and imports the data into a contact list.

- `fn_l` = file name of the backup file used to store unchangeable (locked) contacts.
- `fn_u` = file name of the backup file used to store contacts that can still be changed (unlocked).
- `msg_on` = 0 or 1, enables debugging messages.
- `c_list` = a contact list containing all of the data contained in the input.

write_backups

`function write_backups(c_list, lock_ind_old, lock_ind_new, fn_l, fn_u, msg_on)` - Writes the contact list (`c_list`) over two files: `fn_l`, for past contacts that cannot be changed again, and `fn_u`, for more recent contacts that may be reclassified again.

Normally, the locked list is only appended to since, by definition, the contacts within the locked list cannot change. However, it may be desirable to manually remove a contact that causes problems for the ATR, which would also require removing it from the locked backup. In this case, call this function with `lock_ind_old == 0`. This will mimic the first call to this function and overwrite the locked backup with all contacts with indices up to and including `lock_ind_new`.

- `c_list` = a contact list.
- `lock_ind_old` = index of last locked contact from previous call to `write_backups`.

- `lock_ind_new` = index of last locked contact from current call to `write_backups`.
- `fn_l` = file name of the backup file used to store locked contacts.
- `fn_u` = file name of the backup file used to store contacts that can still be updated.
- `msg_on` = 0 or 1, enables debugging messages.

find_last_locked

`function lastlock_index = find_last_locked(contacts, show_details)` – Determines the last contact to be evaluated by the operator by scanning the `.opconf` values.

- `contacts` = a contact list
- `show_details` = 0 or 1, enables debugging messages.
- `lastlock_index` = index of the last contact to be evaluated by the operator

find_last_viewed

`function lastview_index = find_last_viewed(contacts, new_img_ind, show_details)` – Determines the last contact to be seen by the operator by scanning the `.opdisplay` values.

- `contacts` = a contact list.
- `new_img_ind` = first index of current image; search will start at the contact before this location.
- `show_details` = 0 or 1, enables debugging messages.
- `lastview_index` = index of the last contact that was seen by the operator