Date: 04/26/2008 Version 06
Authors: Ping Fu and Jennifer Staab, UNC-Chapel Hill

Design Document for 3D Neural Activity Mapping

Architecture:

This project will provide both an ImageJ Plugin and Macro that follow the data flow design architecture. The actual design is detailed in the flow diagram on page 13 of this document. The ImageJ plugin will be written in Java and use classes documented in the ImageJ API, as well as the other ImageJ plugins indicated below. The ImageJ macro will use ImageJ's built-in macro language and the ImageJ plugins indicated below.

Both the plugin and the macro will have the same functionality and will follow the flow diagram on page 13. The basic functionality allows users to load a sequence of images, mask them, align them using either TurboReg or a saved transformation file, and finally export this alignment in a SPM-recognizable format. In order to run either the Plugin or the Macro the user needs to load the following software and ImageJ plugins. The plugin will have a more elegant and intuitive interface, while the macro will run faster during the masking, import and other procedures native to the ImageJ built-in macro language. The plugin is intended for most users, especially those new to the process, while the macro is intended for more experienced users running a large batch of images.

The following are ImageJ plugins used in our project:
Used to allow alignment of the imported image sequences:
TurboReg: http://bigwww.epfl.ch/thevenaz/turboreg/
***MultiStackReg: http://www.stanford.edu/~bbusse/work/downloads.html
Used to export image sequence in a SPM readable format:
NIfTi: http://rsb.info.nih.gov/ij/plugins/nifti.html

ImageJ general information, downloads and documentation:
Main information page: http://rsb.info.nih.gov/ij/
Documentation on ImageJ Developer's Resources:
API: http://rsb.info.nih.gov/ij/developer/api/index.html
Built-In Macro Functions: http://rsb.info.nih.gov/ij/developer/macro/functions.html

Statistical Parametric Mapping Software:
SPM software: http://www.fil.ion.ucl.ac.uk/spm/

***MultiStackReg was not directly used. It was revised and named "MultiStackRegFix" and allows the following "additional" functionality:

1. Error checking to guarantee that the alignment plugin, MultiStackRegFix, ran to completion
2. Error checking to guarantee that if a saved Transformation file is used for alignment, the alignment will only occur if the image sequence to be aligned has the same number of images and the same dimensions as the image sequence that originally generated the saved Transformation file.
3. Modified to allow only three types of alignments when run in batch mode from a plugin or macro.

   a. Rigid body alignments that save a Transformation file.
   b. Alignments using a saved Transformation file.
   c. Alignment using a saved Transformation file. Same as **b.** except the program does not prompt the user to select where the saved Transformation file can be found.
4. Modified to prevent warning messages from closing the macro if they appear during execution of the MultiStackRegFix plugin.
5. Allows rigid body alignment and alignment from saved transforms for color images (previously one could only align grayscale images in this way).

Decomposition:

We will have a plugin and a macro with similar functionality. They will both go through the set of screens as specified in the flow diagram on page 13, with the only difference being that the macro runs faster, while the plugin has a more elegant user friendly interface. The macro is mainly for the experienced users that need more speed with their alignments.

AlignWiz Plugin:

The plugin consists of a single .java file with a main class and its subclasses and functions. The plugin is written in Java and uses the Swing library for its interface. The interface consists of a single Swing JFrame object, with JPanels which are added and removed between each panel in the diagram.

1. Main class that calls the first JPanel.
2. Subclasses that create JPanels as specified by flow diagram. There is one class for each JPanel, each with a constructor which adds a new panel to the main JFrame. Each of the subclasses implement ActionListener and include an ActionPerformed() function to process events that happen on its own JPanel buttons.
3. Functions of the main class:
   - **align()—**calls MultiStackReg to align images
   - **maskSection()**—masks brain section appropriately for either cells or grains.
   - **maskBrain()**—masks an entire brain for both grayscale and colored cells. Grains for entire brain are masked by simply importing the existing mask files.
   - **closeImage()**—closes the image if it is open.
   - **deleteDirectory()**—deletes a folder and all files contained in the folder.
   - **run()**—main function that is called when ImageJ runs the plugin.

A detailed view of all subclasses, fields, and methods can be found in the JavaDoc here:

Class AlignWiz_

```
java.lang.Object
       AlignWiz_
```

Date: 04/26/2008 Version 06
Authors: Ping Fu and Jennifer Staab, UNC-Chapel Hill

```
public class AlignWiz_
extends java.lang.Object
```

AlignWiz Class.

This class is a plugin intended for ImageJ. It allows the user to align and mask 2D slice images. The user can select between aligning images of an entire brain or a brain section, and aligning directly with TurboReg or using an existing transformation file. The output format of the aligned images is a .nii file, which utilizes the NIfTi plugin and is intended for analysis with the Statistical Parametric Mapping (SPM) software.

This plugin uses the Swing library to create Panels that lead the user through the process, and each subclass of the main AlignWiz class creates a JPanel and sets appropriate actionListeners.

Plugins used:

-TurboReg plugin, written by PhillipeThevenaz (http://bigwww.epfl.ch/thevenaz/turboreg/)

-MultiStackReg plugin, originally written by Brad Busse (http://www.stanford.edu/~bbusse/work/downloads.html), modified by Jennifer Staab and Ping Fu (http://www.cs.unc.edu/Courses/comp523-s08/3D_Neural/home.html)

-NIfTi plugin, written by Guy Williams (http://rsb.info.nih.gov/ij/plugins/nifti.html) SPM software for data analysis: http://www.fil.ion.ucl.ac.uk/spm/

Version:
1.0
Author:
Ping Fu (corresponding macro written by Jennifer Staab), UNC-Chapel Hill

Nested Class Summary

class	**AlignWiz_.Gap** Gaps for use in GridBagLayout (or any other).
private class	**AlignWiz_.GBHelper** Keeps track of current position in GridBagLayout.
private class	**AlignWiz_.Panel0** Sets up the initial panel with the back, next, and quit buttons
private class	**AlignWiz_.Panel1** Starting panel. Allows user to select to align Entire Brain (grayscale), Entire Brain (color), or Brain Section (grayscale)
private class	**AlignWiz_.Panel2** Allows user to select to align with TurboReg or Transformation file
private class	**AlignWiz_.Panel3** Allows user to import image sequence.

private class	**AlignWiz_.Panel3A** Lets user apply Rotations or Translations to stack pre: useSavedTrans==false
private class	**AlignWiz_.Panel4A** Masking for entire brain.
private class	**AlignWiz_.Panel4C** Allows user to check masking and align pre: useSavedTrans==true
private class	**AlignWiz_.Panel5A** Allows user to check masking and align pre: userSavedTrans==false
private class	**AlignWiz_.Panel5C** Allows user to import transformation file and masks pre: useSavedTrans==true
private class	**AlignWiz_.Panel6A** Allows user to realign, save, view 3D projections, or start over
private class	**AlignWiz_.Panel7** Allows user to import corresponding grains, start over, or exit
private class	**AlignWiz_.PopupListener** ActionListener for popup window after 'Quit' is clicked;

Field Summary

(package private) javax.swing.JButton	**bt01** The 'back' button
(package private) javax.swing.JButton	**bt02** The 'next' button
(package private) javax.swing.JButton	**bt03** The 'quit' button
private boolean	**color** True if images are of colored brain
private java.lang.String	**contains** tring contained in the file names in the image sequence
private int	**endNum** Number of last image in the loaded sequence
private boolean	**entireBrain** True if images are of entire brain in grayscale or color, false if images are of brain section
private javax.swing.JFrame	**f** Main frame that hold all labels and panels
private int	**inc**

	Increment between image numbers in sequence
private boolean	**isGrains** True only after user chooses to align corresponding grains images
private javax.swing.JLabel	**lbl** Main label with directions for each panel
private java.lang.String	**loadPathAndFileName** Path and file name of the first loaded cells image
private java.lang.String	**loadPathC** Path from where cells images were loaded
private java.lang.String	**loadPathG** Path from where grains images were loaded
private int	**numImg** Number of images in the loaded sequence
private javax.swing.JPanel	**p** JPanel containing back, next, and quit buttons
private boolean	**popupOpen** True if popup window is already open
private int	**scale** Percentage of scale of images from original size
private java.lang.String	**slash** File delimiter, either '\' or '/' depending on OS
private int	**startNum** Number of first image in the loaded sequence
private java.io.File	**tempDir** Temporary directory, deleted if user exits by clicking 'exit' at the end of the process
private java.lang.String	**transformationPath** Path where transformation file is loaded from
private boolean	**useSavedTrans** True if using saved transformation file, false if aligning with TurboReg

Constructor Summary

AlignWiz_()	

Method Summary

private boolean	**align**() Aligns stack by calling MultiStackRegFix, also checks if alignment was successful
private void	**closeImage**() Closes the image if there is an image currently open
static boolean	**deleteDirectory**(java.io.File path) Deletes all files in a directory
private void	**maskBrain**(ImagePlus threshIm, int iteration) Masking for entire brain pre: entireBrain==true
private void	**maskSection**(ImagePlus image) Masking for brain section pre: entireBrain==false
void	**run**(java.lang.String arg) Main function that runs in ImageJ when plugin is called.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

run

public void **run**(java.lang.String arg)

Main function that runs in ImageJ when plugin is called. It sets the file delimiter 'slash,' and creates the initial panel with Panel0()

---

closeImage

private void **closeImage**()

Closes the image if there is an image currently open

---

deleteDirectory

public static boolean **deleteDirectory**(java.io.File path)

Deletes all files in a directory

Parameters:

path - Path of the directory

---

align

```
private boolean align()
```

Aligns stack by calling MultiStackRegFix, also checks if alignment was successful

Returns:

True if alignment was successful, False if not

---

maskSection

```
private void maskSection(ImagePlus image)
```

Masking for brain section pre: entireBrain==false

Parameters:

`image` - Image to be masked

---

maskBrain

```
private void maskBrain(ImagePlus threshIm,
                       int iteration)
```

Masking for entire brain pre: entireBrain==true

Parameters:

`threshIm` - Thresholded image that is about to be masked

`iteration` - Either 1 or 2. 1 is a rough first masking, 2 is a finer masking which gets the finer details.

Macro:

The macro is made up of multiple functions that allow the programmer a level of abstraction so that they can write a while loop which will drive the primary functionality of the macro. The while loop allows the user to choose one of the 4 different use cases depicted in the functional specifications documentation, performs the selected use case operations for image alignment and loop back to the start to inquire the user's next selection. This allows the user the ability to align multiple animal brains from different use case scenarios in one session.

ImageJ's built-in macro language is rather limited in its scope and user friendliness. Thus a naïve user is encouraged to use the Plugin over the macro as to prevent mistakes in using these tools. Global variables had to be used to allow for the passing of variables between the functions used to create the macro. An example of the limited nature of the macro language is that if the user clicks on the 'cancel' button in any of the dialog windows use in the macro; the macro will be immediately aborted due to the limitations of the macro language.

Below are the details of each of the functions used in the macro:

Global Variables:

The following global variables are set using cells and grains imports as to allow the functions that are used to align these images access to variables. This access is necessary with regards to

masking the imaging and error checking to be certain that the dimensions and number of images match between corresponding cells and grains images. Global variables defined in the ImageJ macro language are designated by the keyword 'var' when they are first defined.

Global variables with default values in parentheses:

Imports Cells:

- Number of Cells Images: ***numImgCells(1)***
- Number of First Image in the Cells Sequence: ***startNumCells(1)***
- Number of last Image in the Cells Sequence: ***endNumCells(2)***
- Increment between adjacent Imported Cells Images: ***incCells(1)***
- Reduction scale of the dimensions of the imported cells image sequence: ***scaleCells(100)***
  * where 100= no reduction in dimensions and 80 = 80% reduction in dimensions of images
- String Common to all the file names of the imported cells image sequence: ***containsCells("")***
- Path of place where the cells image sequence was imported from including the filename of one of the imported images: ***pathCells("")***
- Path including the foldername of the directory that cells image sequences were imported from within: ***folderCells("")***
- Height dimension of imported cells image sequence: ***heightCells(0)***
- Width dimenstion of imported cells image sequence: ***widthCells(0)***
- Global variable for imported cell image sequence ID: ***outImageID(0)***

Imports Grains:

- Number of Grains Images: ***numImgGrains(1)***
- Number of First Image in the Grains Sequence: ***startNumGrains(1)***
- Number of last Image in the Grains Sequence: ***endNumGrains(2)***
- Increment between adjacent Imported Grains Images: ***incGrains(1)***
- Reduction scale of the dimensions of the imported Grains image sequence: ***scaleGrains(100)***
  * where 100= no reduction in dimensions and 80 = 80% reduction in dimensions of images
- String Common to all the file names of the imported Grains image sequence: ***containsGrains("")***
- Path of place where the Grains image sequence was imported from including the filename of one of the imported images: ***pathGrains("")***
- Path including the foldername of the directory that Grains image sequences were imported from within: ***folderGrains("")***
- Height dimension of imported Grains image sequence: ***heightGrains(0)***
- Width dimenstion of imported Grains image sequence: ***widthGrains(0)***

General Global Variables:

- Global variable that defined either forward or backward slash to be used when creating paths based on the operating system (Windows='\' and Mac='/'): ***rSlash("")***
- Global variable defined to indicate that the one needs to return to the 'start' of the program after receiving a 'Start Over' command or some sort of error that requires a start over: ***restart(true)***

Functions:
The following functions were create in the ImageJ macro program to ease the ability of the program to run in a more flexible format than running as straightforward non-branching script. Using functions allowed the programmer to add in error checking and make program more flexible with functionality similar to the plugin alignment wizard. ** Double star indicates the function was used in the primary while loop.

Functions:

- ****startMenuBrain() –** Ask the user what type of alignment they are doing Entire Brain (Grayscale), Entire Brain (Color) or Brain Section.
  No input variables
  Returns the user selection.
- ****resetGlobals() –** Resets global all variables used in the macro (as defined above) to their default values.
  No input variables
  Returns Nothing.
- **resetGlobalCells() –** Resets global 'cells' variables as define above to their default values.
  No input variables
  Returns Nothing.
- **resetGlobalGrains() –** Resets global 'grains' variables as define above to their default values.
  No input variables
  Returns Nothing.
- ****startMenuTransform() –** Ask the user if they are aligning using Saved Transforms or using TurboReg.
  No input variables
  Returns user selection.
- ****importSequence() –** Imports cells image sequence into ImageJ and set global variables pertaining to the cells images. Does error checking for real values, negative values, and impossible startNumCells, endNumCells and incCells combinations. Set global variables based on user input. Will keep asking for import until user enters in acceptable inputs into the import window.
  No input variables
  Returns imported Image Sequence ID
- **failImportWarning(errText) –** code for displaying an error message for errors in the Sequence Imports.
  Input variables: errText = error message to be viewed by user.
  Returns nothing
- **importSequenceGrains(xGlobals) –** Imports grains image sequence into ImageJ and set global variables pertaining to the grains images. Does error checking for real values, negative values, and impossible startNumGrains, endNumGrains and incGrains combinations. Will also check to make sure imported Grains matches existing imported Cells in dimension and number if imported corresponding cells previous to the grains

images. Sets global variables based on user input. Will keep asking for import until user enters in acceptable inputs into the import window.
Input variables: xGlobals = that inquires whether or not the program should check cells and grains image compatibility.
Returns: Grains Image ID

- **importSequenceSaveTrans() –** imports grains sequence for grains imported when the user will align using a Saved Transformation file. Use function above and findMaskDir() function to import the grains sequence (xGlobals==false) because no existing grains file.
No input variables
Returns: Grains Image ID
- ****checkOS() –** determine which types of slashes are use (back or forward) dependent on the operating system. Sets global variable rSlash appropriately
No input variables
Returns Nothing.
- ****blurAndThreshold() –** creates masks based on the imported cell sequence using ***Gaussian blur and threshold tools of ImageJ***. These masks, mask out the background in the imported sequence.
No input variables
Returns: Image Id of image that contains the initial masks.
- ****createMasks(inImageID,inFolder,inrSlash,inStart,inInc) –** creates and saves the masks as individual mask*.tif files for each of the images in the mask image sequence. Properly increments masks if the global increment value is not 1 or starts from a number greater than one.
Input variables: inImageID = input sequence of masks,
inFolder= folder where masks should be saved
inrSlash=correct slash given OS,
inStart=input image sequence start number
inInc=input image increment value.
Returns: nothing but has the side effect of creating the masks.
- ****maskOutBackground(inPath,inFolder,inNum,inStart,inInc,inScale,inContains,inrSlash) –** masks out the background of the imported cells sequence images using the masks created and saved in the previous steps.
Input variables: inPath=path of input sequence
inFolder= folder where masks were saved
inNum= number of input images in the sequence
inStart=input image sequence start number
inInc=input image increment value
inScale=Scale of input sequence
inContains= the common string in the input sequence if any exists
inrSlash=correct slash given OS.
Returns: image ID of masked images.
- ****xAndFixMasks(inImageID,inrSlash,inFolder,inStart,inInc,isColorBrain) –** allows the user to check the masking and fix the masks using the brush tool if necessary
Input variables: inImageID=input image sequence whose masking to be checked
inrSlash=correct slash given OS.
inFolder= folder where masks were saved

inStart=input image sequence start number
inInc=input image increment value
isColorBrain=1 if input image sequence is of a color brain.

Returns: Nothing

- **maskColoredImages(inPathSeq,inFolderMask,inNum,inStart,inInc,inScale,inContains,isGrains,isSavedTransform,RA_Out) –** masked colored images in the proper manner handling the fact that the client prefers a white background.
  Input variables: inPathSeq=path of input image sequence
  inFolderMask= folder where masks were saved
  inNum= number of input images in the sequence
  inStart=input image sequence start number
  inInc=input image increment value
  inScale=Scale of input sequence
  inContains= the common string in the imput sequence if any exists,
  isGrains=1 if the input image is a grains image
  isSavedTransform=1 if the input image is to be aligned with saved transforms
  RA_Out= number of times user incremented through realignment if one previously aligned a cells image sequence.
  Returns: Image ID of masked Image Sequence
- **transformMenu(titleText,messText)—** Creates dialog box that disappears when 'Ok' is selected. This dialog box will either prompt the user to load or save transforms
  Input variables: titleText=title of dialog window (should be Load or Save)
  messText=text message of dialog window (should be Load or Save)
  Returns: Nothing
- **alignWithStackReg(inrSlash,inFolder,isColorBrain) –** Aligns input image sequence using TurboReg, aligns correctly if brain image is color brain where inverts before and after alignment. Also checks that MultiStackRegFix ran to completion by checking for 1 in xAlign.txt file.
  Input variables: inFolder= folder where cells images were imported (xAlign.txt file saved)
  inrSlash=correct slash given OS
  isColorBrain=1 if image to be aligned is a color image sequence**.**
  Returns: returns whether or not the images were aligned (e.g. multiStackRegFix ran to Completion)
- **checkAlignment(inImageID,isBrainRegion,isColorBrain) –** Prompts user to check the alignment to see that it was successful. Shows 3D projections in correct manner (correct for color brains) if the images are entire brains, does not give 3D projections of brain sections.
  Input variables: inImageID= sequence of images to have their alignment checked
  isBrainRegion=1 if the sequence of images is a brain section
  ,isColorBrain=1 if sequence of input images I a color brian.
  Returns: returns Image ID of sequence images that were checked
- ****alignWithStackRegLoop(inrSlash,inFolder,isBrainRegion,isColorBrain) –** aligns cells images, saves the transforms from the alignment, allows user to check the alignment

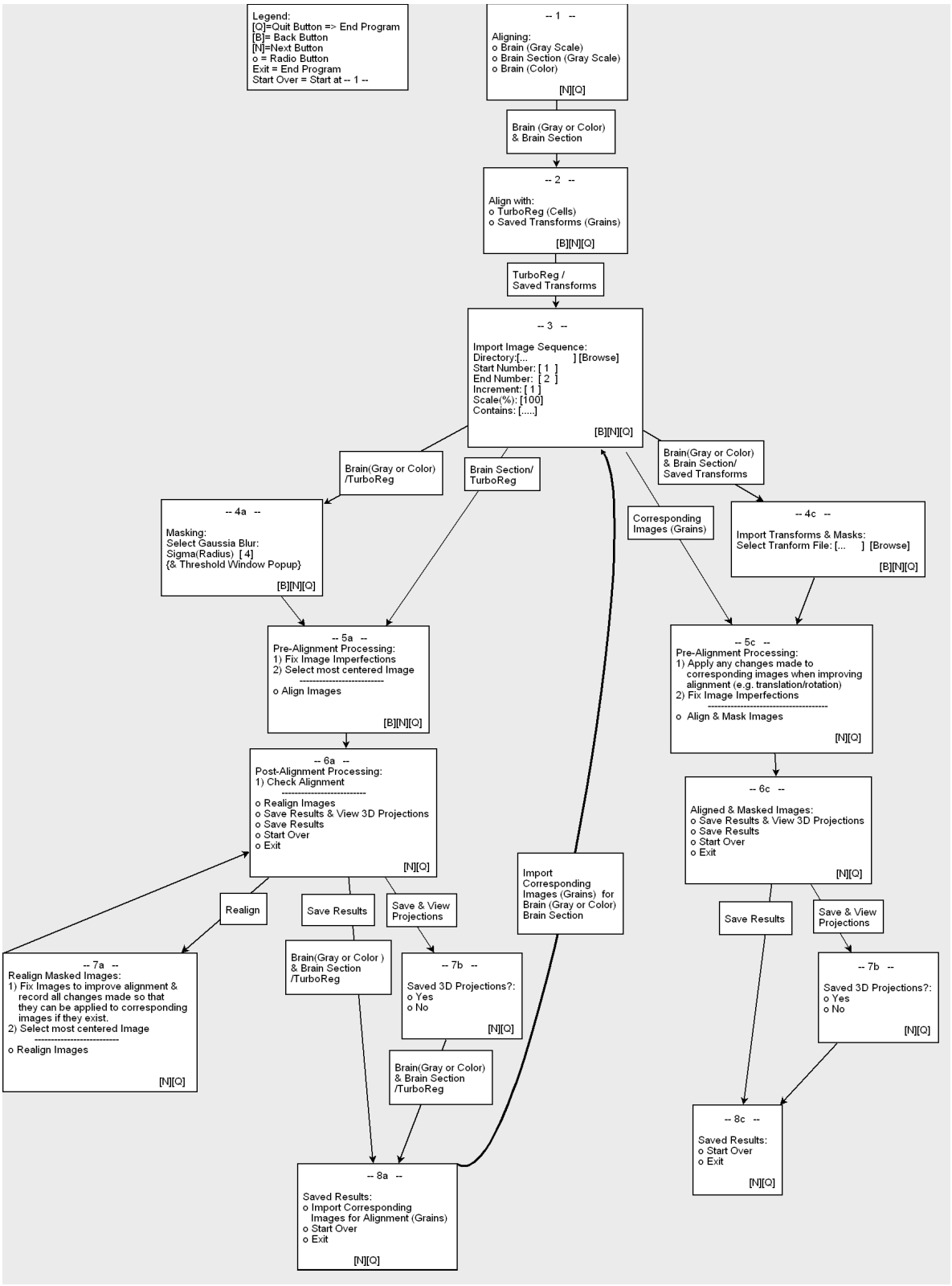
and realign if necessary, saves the results from the alignment, and allows user to view and save the 3D projections.
Input variables: inFolder= folder where images to be aligned are saved
inrSlash=correct slash given OS
isBrainRegion=1 if image sequence to be aligned is a brain section,
isColorBrain=1 if image to be aligned is a color image sequence.
Returns: The number of times the cells images were re-aligned if any
****alignGrains() –** Asks user if they would like to align corresponding images (grains).
No input variables
Returns: the user selection.

- **saveProjections(inFolder,inrSlash,inImageID,isColorBrain)—**
Asks User if they want to view 3D projections, if they say yes then asks if they want to save 3D projections after the 3D projections are shown. Correctly treats color brain images with inverting before and after 3D projection creation.
Input variables: inFolder= folder where aligned images 3D projects should be saved
inrSlash=correct slash given OS
inImageID=image ID of aligned image sequence results.
isColorBrain=1 if image to be aligned is a color image sequence
Returns: Nothing
- **saveResultsAskProjections(inImageID,inFolder,inrSlash,isColorBrain)—**
Asks User where to save the results of the image sequence alignment, saves the resulting image there and then runs the saved projects function (above) that asks about viewing & saving 3D projections. Correctly treats color brain images with inverting before and after 3D projection creation.
Input variables: inImageID=image ID of aligned image sequence results.
inFolder= folder where the aligned images to be aligned are saved
inrSlash=correct slash given OS
isColorBrain=1 if image to be aligned is a color image sequence
Returns: Nothing
- **xDimensions() –** checks the dimensions of the grains images as compared to the cells images and if off displays a error message that asks what action the user wants to take next Import Grains again, restart or quit
No input variables
Returns: the result of the user selection
- ****importMaskGrains(inCellsFolder,inrSlash,inNum,inStart,inInc,inScale,inContains,isBrainRegion,isSavedTransform,isColorBrain,RA_Out) –** imports the corresponding images (grains images) and masks them.
Input variables: inCellsFolder=path of input cells sequence(or masks location)
inrSlash= correct slash given OS
inNum= number of input grains images in the sequence
inStart=input grains image sequence start number
inInc=input grains image increment value
inScale=Scale of input grains sequence
inContains= the common string in the input grains sequence if any exists,
isBrainRegion=1 if input grains image is a brain section
isSavedTransform=1 if the input grains image is to be aligned with saved

transforms
isColorBrain=1 if Brain is a color image
RA_Out= number of times user incremented through realignment if one previously aligned a cells image sequence

Returns: the Image ID of the masked grains image or 9999 if function requires a restart

- **allowForGrainsEdit(inImageID,inFolder,inrSlash,isSavedTransform) –** allows the user to adjust the corresponding image sequence (grains images) if they adjusted the cells images prior to realignment.
  Input Variables: inImageID=input Grains image ID
  inFolder = folder that contains cells image (or masks)
  inrSlash=correct slash given OS
  isSavedTransforms=1 if image is to be aligned with saved transforms.
  Returns: the Image ID of the resulting edited Grains image
- ****alignGrainsXAndSave(inImageID,inFolder,inGrainsFolder,inrSlash,isBrainRegion,isColorBrain) –** aligns the corresponding (grains) images using saved transformation file, save the results and prompts the user about viewing and saving 3D projections of the alignment.
  Input Variables: inImageID=input grains image ID,
  inFolder = folder that contains cells images (or masks)
  inGrainsFolder=folder that contains the grains images,
  inrSlash=correct slash given OS
  isBrainRegion=1 if is brain section
  isColorBrain=1 if is color brain.
  Returns: Nothing
- **createRegionMask(inImageStack,inrSlash)—** Creates the correct masks for brain regions and returns nothing.
  Input Variables: inrSlash=correct slash given OS
  inImageStack=image ID of image sequence to be masked.
  Returns: Nothing
- **maskBrainRegion(inImageStack,inrSlash)—** Using function above correctly masks brain regions images and returns results.
  Input Variables: inrSlash=correct slash given OS
  inImageStack=image ID of image sequence to be masked.
  Returns: Image ID of the resulting masked brains image
- **waitWindow(title,msg)—**Creates a wait window with input title and msg, returns nothing.
  Input Variables: title = title of wait window
  msg=message text inside wait window
  Returns: Nothing


Legend:
[Q]=Quit Button => End Program
[B]= Back Button
[N]=Next Button
o = Radio Button
Exit = End Program
Start Over = Start at -- 1 --
-- 1 --
Aligning:
o Brain (Gray Scale)
o Brain Section (Gray Scale)
o Brain (Color)
[N][Q]
Brain (Gray or Color) & Brain Section
-- 2 --
Align with:
o TurboReg (Cells)
o Saved Transforms (Grains)
[B][N][Q]
TurboReg / Saved Transforms
-- 3 --
Import Image Sequence:
Directory:[...] [Browse]
Start Number: [1]
End Number: [2]
Increment: [1]
Scale(%): [100]
Contains: [.....]
[B][N][Q]
Brain(Gray or Color) /TurboReg
Brain Section/ TurboReg
Brain(Gray or Color) & Brain Section/ Saved Transforms
Corresponding Images (Grains)
-- 4a --
Masking:
Select Gaussia Blur:
Sigma(Radius) [4]
{& Threshold Window Popup}
[B][N][Q]
-- 4c --
Import Transforms & Masks:
Select Tranform File: [...] [Browse]
[B][N][Q]
-- 5a --
Pre-Alignment Processing:
1) Fix Image Imperfections
2) Select most centered Image
o Align Images
[B][N][Q]
-- 5c --
Pre-Alignment Processing:
1) Apply any changes made to corresponding images when improving alignment (e.g. translation/rotation)
2) Fix Image Imperfections
o Align & Mask Images
[N][Q]
-- 6a --
Post-Alignment Processing:
1) Check Alignment
o Realign Images
o Save Results & View 3D Projections
o Save Results
o Start Over
o Exit
[N][Q]
-- 6c --
Aligned & Masked Images:
o Save Results & View 3D Projections
o Save Results
o Start Over
o Exit
[N][Q]
Import Corresponding Images (Grains) for Brain (Gray or Color) Brain Section
Realign
Save Results
Save & View Projections
Save Results
Save & View Projections
-- 7a --
Realign Masked Images:
1) Fix Images to improve alignment & record all changes made so that they can be applied to corresponding images if they exist.
2) Select most centered Image
o Realign Images
[N][Q]
Brain(Gray or Color) & Brain Section /TurboReg
-- 7b --
Saved 3D Projections?:
o Yes
o No
[N][Q]
-- 7b --
Saved 3D Projections?:
o Yes
o No
[N][Q]
Brain(Gray or Color) & Brain Section /TurboReg
-- 8c --
Saved Results:
o Start Over
o Exit
[N][Q]
-- 8a --
Saved Results:
o Import Corresponding Images for Alignment (Grains)
o Start Over
o Exit
[N][Q]

Data:

Image input data that are valid must be files recognizable by ImageJ. A non-exhaustive list of file formats supported by ImageJ can be found at < http://imagejdocu.tudor.lu/imagej-documentation-wiki/faq/which-file-types-are-supported-by-imagej > (Valid as of February 7, 2008).

The 3D reconstruction output will have NIfTi data format, which is supported by the SPM (Statistical Parametric Mapping) software. Details about the NIfTi file format can be found at < http://nifti.nimh.nih.gov/nifti-1 > (Valid as of February 7, 2008).

User input:

Below are the typical inputs that are required of the user when running the macro and plugin programs. The specific dialog windows for each of these inputs can be viewed in the User's Manual.

The user is required to give the following inputs:

1. Import Image (cells or grains) Sequence:
    a. Requires the path of where the image sequence can be found
    b. Requires Start Number of the first image in the sequence
    c. End number of the last image in the sequence
    d. Increment to be used to import the image sequence
    e. Scale indicates the amount to reduce the scale of the image sequence (i.e. 100= no scale reduction of the image sequence)
    f. 'Contains' if the images sequence contains a common a common string (like 'cells' or 'grains').

2. User selects what type of brain images they are going to use
    a. Brain (grayscale)
    b. Brain (Color)
    c. Brain Section

3. User selects what type of alignment they will perform
    a. Align with TurboReg
    b. Align using Save Transforms

4. Place to save the following output items:
    a. Transformation file (that results from the alignment)
    b. SPM readable format of the alignment results (both cells & grains)

5. Place to find (load) the following items:
    a. Transformation file (so can align images with saved transforms)
    b. Masks (when aligning with saved transforms)

6. User provides choices like 'next' or 'back' to move forward or backward in the plugin. User provides 'ok' to proceed forward in the macro.

7. User helps with the masking of the 'Brain' images by selecting an appropriate Gaussian blur (Gaussian blur is an ImageJ macro function) and Thresholding (threshold is an ImageJ macro function) to mask out the background.
8. The user checks and fixes imperfections in the masking (i.e. uses the paint brush tool to cover up imperfections with the background color as they use the image scrollbar to process through the entire image sequence)

9. The user checks the alignment and decide whether or not they need to realign the image sequence. If the user decides to re-align the images they are allowed to use the 'rotate' and 'translate' tools of ImageJ to attempt to make the image sequence more consistent and centered between adjacent image slices as to improve the alignment that TurboReg will produce.

10. The user decides whether or not to view 3D projections of the alignment and whether or not they need to save these 3D projections.

Design decisions:

We decided to use TurboReg for alignment and the NifTI plugin for our I/O requirements. The TurboReg algorithm can align images, color or grayscale, using a rigid body transformation, where one image is rotated, translated vertically and/or translated horizontally to align it to the other image. Specifically, we used a plugin named 'MultiStackReg' that uses TurboReg to align a sequence of images and provides the transformation file from that alignment. In addition, MultiStackReg will also align images using a previously output Transformation file. To add some necessary functionality as outlined in the beginning of this document we augmented MultiStackReg to create 'MultiStackRegFix' which is used to perform the image sequence alignments for both the plugin and the macro. To output the aligned image sequence, we use the plugin NifTI which outputs the aligned image sequence in a SPM readable format. In addition, NifTI files can be imported directly into ImageJ, so the SPM readable format provides a good way to store the aligned image sequence in a single file.

We made certain design decisions to make the final software product as flexible as possible. Because there is so much variability in brain image sequences we attempted to add features to the software as to enable it to be as robust as possible. Some of those features are to allow the user input in the masking of the images so that the parameters chosen during masking will fit all images in the image sequence. In addition, we allow the user the ability to check and fix masking during the alignment process as to allow maximum flexibility for all types of brain images. Moreover, we ask the user to check the alignment during the alignment process as to allow the user the ability to edit the images and attempt a re-alignment if the first alignment fails during the process. This increases user flexibility since the user doesn't have continue the process if the alignment fails. In addition, the user is allowed to adjust the images in the software product to prevent the inconvenience of having to exit the software product to edit the images and then re-start the software product to re-attempt another alignment.

To increase the flexibility of our software tool, we offer our software product as both an ImageJ Plugin and an ImageJ macro that have similar functionality. The plugin is much more elegant and user friendly since it is created as an alignment wizard. The plugin is suggested for the naïve user. Due to the limitations of ImageJ's macro language, the macro could not be converted into an alignment wizard. The macro does offer the advantage of speed, the macro runs much more quickly for certain functions that are native to the ImageJ macro language, like masking and sequence import. Thus, for the experienced user the macro might be the optimal choice because of its speed with regards to aligning large numbers of high resolution images.

Finally we suggest that the user chose pixel resolution on the order of 512x512 pixels, due to possible memory requirements in SPM, if the user plans to import the aligned images into SPM for statistical analysis. Our program will produce output that can be recognized and analyzed by SPM, but will not deal directly with the SPM software.