

Technical Description of the AutoRegister Components

Oliver Hinds <ohinds@orchardscientific.org>

Contents

1 Overview	2
2 Scout MRI pulse sequence	2
2.1 Repository	2
2.2 Details	2
2.3 Protocol	3
2.3.1 Experiment to determine sequence parameters	3
2.3.2 Protocol parameters	4
3 Registration module	4
3.1 Repository	4
3.2 Environment	4
3.2.1 Python	4
3.2.2 mri_robust_register	5
3.3 File formats	5
3.3.1 NIFTI	5
3.3.2 LTA	5
3.4 Source	5
3.4.1 High-level description	5
3.4.2 auto_register.py	6
3.4.3 external_image.py	7
3.4.4 image_receiver.py	9
3.4.5 registered_image.py	10
3.4.6 tcpip_server.py	11
3.4.7 terminal_input.py	15
3.4.8 transform_sender.py	16
3.5 Tools	17
3.5.1 vsend_nii	17
3.6 Tests	17
4 MR Image reconstruction module	18
4.1 Repository	18
4.2 Details	18
4.2.1 Image sending	18
4.2.2 Transformation handling	18

1 Overview

This document explains the technical details of the components produced for Phase One of the AutoRegister grant. The accompanying document `workflow.md` describes the installation and operation of the AutoRegister system.

The AutoRegister system contains three software modules. The Scout MRI pulse sequence module runs on the MRI scanner and is responsible for playing out the RF pulse sequence that acquires the scout image for registration. The Reconstruction module is Python software running on a laptop external to the MRI system and is responsible for computing transformations based on images received from the scanner, and sending these transforms back to the scanner. The MR image reconstruction module runs on the scanner, reconstructs the scout image, sends it to the Registration module, listens for the transformation sent back, and makes it available for future scans in the session.

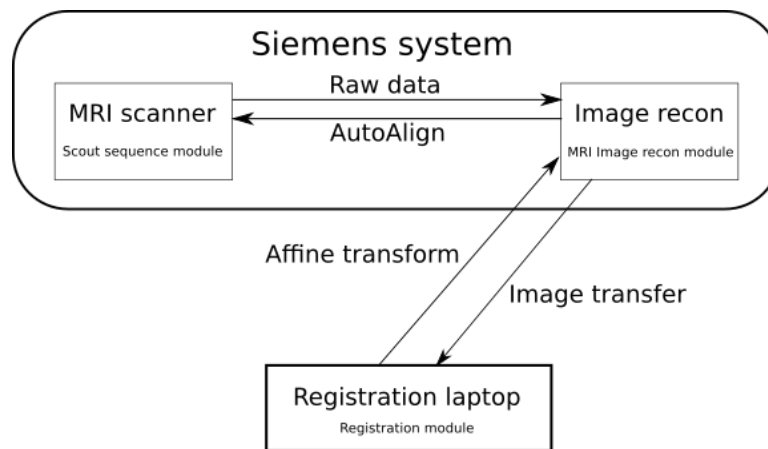


Figure 1: Communication diagram showing the data passed between modules.

2 Scout MRI pulse sequence

The AutoRegister Scout MRI pulse sequence is needed to acquire an image of a patient's head, which is appropriate for computing a spatial transformation between a previous or subsequent image of the same patient.

2.1 Repository

`git@52.72.193.62:ohinds/auto_register_sequence.git`

2.2 Details

The AutoRegister scout pulse sequence is based directly off of the gradient echo pulse sequence distributed by Siemens under the name `a_gre`. The scout has been successfully tested on Siemens

baselines VB17A_20190307 and VD13C_20121124. A scout for baseline VE11B_20150530 is under development.

For each platform version, the only change that needs to be made from the Siemens default pulse sequence is to change the ICE program filename to point to the AutoRegister MR Image reconstruction module. Specifically, clone the `a_gre` pulse sequence code to a new sequence `AutoRegisterScout` and change the line in `a_gre.cpp`: `rSeqExpo.setICEProgramFilename(...)` to point to the AutoRegister ICE Program, e.g. `%CustomerIceProgs%\ohinds\IceProgramAutoRegisterInterface`

2.3 Protocol

2.3.1 Experiment to determine sequence parameters

Early in the AutoRegister project, a series of test scans was conducted to determine a set of protocol parameters appropriate to act as a scout. During these experiments a subject was scanned to simulate two scan sessions: they were removed from the MRI scanner halfway through the scan and repositioned as if they were being scanned in a second session.

In each "session", several candidate scout images were acquired with different pulse sequence parameters for comparison (see Table 1). In addition, several relatively high-resolution MPRAGE images were acquired to serve as ground truth. Images for the subjects `ar_scout_param_test_1`, `ar_scout_param_test_3`, and `ar_scout_param_test_3`, on which sequence parameter choices were made, are located in the directory `/autofs/eris/cmet/autoReg/DevScans/DevVolunteers/`

Table 1: Table 1: Pulse sequence parameters that were varied in candidate scout sequences.

Candidate	Voxel size	Matrix size	Acceleration	Acquisition time
1	1.33 mm ³	192	2	TODO
2	1.33 mm ³	192	3	
3	1.33 mm ³	192	4	
4	2 mm ³	128	0	
5	2 mm ³	128	2	
6	2 mm ³	128	3	
7	2 mm ³	128	4	
8	4 mm ³	64	0	
9	4 mm ³	64	2	
10	4 mm ³	64	3	
11	4 mm ³	64	4	

These data were analyzed by coregistering corresponding candidate and ground truth scans across sessions using `mri_robust_register`. The residual registration error compared to the ground truth was computed for each candidate scout using `lta_diff`. A summary of the residual registration error for each subject is contained in the file `Registration/summary.txt`. Based on this experiment, we decided to use candidate (5), as it has relatively low error (about 0.8 mm for the subject shown in Figure 2), and a reasonable acquisition time.

For one subject, `nu_correct` was used to correct intensity non-uniformities prior to registration. No substantial difference in residual error was observed, so the decision was made to forgo `nu_correct`.

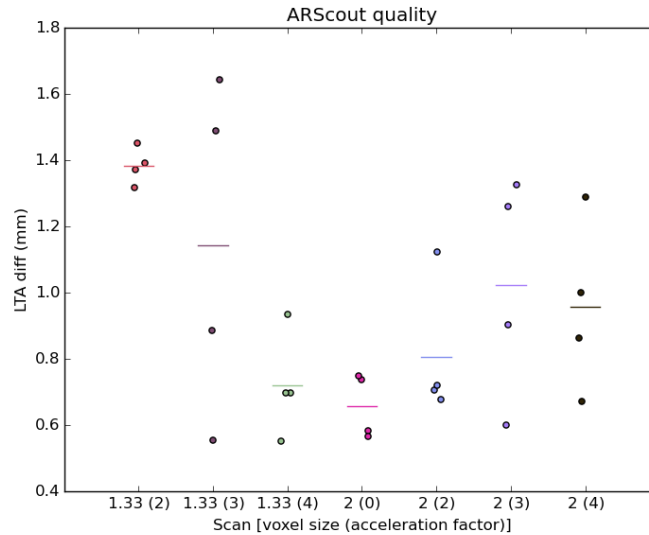


Figure 2: Comparison of the residual registration error of the candidate scout sequences.

2.3.2 Protocol parameters

The AutoRegister Scout captures images with a 3D gradient echo sequence with a TE of 3.82ms, a TR of 7.9ms, a flip angle of 25 degrees, an isotropic voxel size of 2mm, and an acquisition matrix size of 128x128 with 128 slices. With a GRAPPA acceleration factor of 2, this sequence takes TODO TIME to acquire.

3 Registration module

The registration module is python software that runs on a computer external to the MRI system: currently a laptop in the scanner control room. The software receives an image from the MR Image reconstruction module, computes a spatial transformation, and sends the transformation back to the Image reconstruction module.

3.1 Repository

`git@52.72.193.62:ohinds/auto_register.git`

3.2 Environment

3.2.1 Python

To avoid version and package conflicts, the python software runs in a dedicated virtual environment produced by the `virtualenv` software. The `workflow.md` file describes the process of setting up the virtual environment, which is very simple.

The python libraries on which the Registration module depends are listed below.

- `numpy` for matrix math
- `nibabel` for reading and writing NIFTI images
- `nosetest` for running tests

3.2.2 `mri_robust_register`

Co-registration of scout images is accomplished using the tool `mri_robust_register` from the FreeSurfer software package. Instructions for installing FreeSurfer and configuring a shell environment suitable for running `mri_robust_register` are available at <http://freesurfer.net/>.

3.3 File formats

3.3.1 NIFTI

The NIFTI file format is widely used to store MR images, and `mri_robust_register` inherits NIFTI compatibility from FreeSurfer. The Registration module uses the `nibabel` python package to write out NIFTI files when MR images are received from the Image reconstruction module.

3.3.2 LTA

The LTA file format stores a linear transformation in text format. This is the format in which `mri_robust_register` stores computed transformations. The Registration module contains custom code to load a transformation from an LTA file.

3.4 Source

The source code for the Registration module is written in Python. It has been tested with Python version 2.7.

3.4.1 High-level description

The main file, `auto_register.py`, controls operation of the AutoRegister Registration module. It is executed from the command line while in the python environment described above. It's main duty is to configure an instance of the `AutoRegister` class and to execute it's main loop.

In the main loop of the `AutoRegister` class, four operations are carried out. First, a class member instance of `ImageReceiver` is queried to check if new images are available from the MRI scanner or simulator. Second, if a new image is available, it is either saved as the reference image for registration with subsequent images, or it is taken as subsequent image that is registered with the reference image using a newly created instance of the `RegisteredImage` class. Third, once the registration is completed,

a class member instance of `TransformSender` is notified that a new transformation is ready. This transformation is sent when a request is made for it from an external client. Fourth, a member instance of `TerminalInput` is queried to determine if a new character has been entered by the user. If so, and if this character is 'q', the main loop exits.

The sections below give a high-level description of each source file and list the output of the `pydoc` Python source code documentation generation tool.

3.4.2 `auto_register.py`

The top-level file in the Registration module. It contains the `__main__` entry point, and thus is the file that is executed to run the entire Registration module.

Help on module `auto_register`:

NAME

`auto_register` - Main file and class for the autoregister application.

FILE

`/home/ohinds/projects/auto_register/src/auto_register.py`

CLASSES

`__builtin__.object`
`AutoRegister`

```
class AutoRegister(__builtin__.object)
|   Methods defined here:
|
|   __init__(self, args)
|       Initialize the autoregister application and helper modules.
|
|   check_for_input(self)
|       Return the last character input, or None. If 'q' is seen, the
|       autoregister application shuts down.
|
|   run(self)
|       Main loop of the autoregister application.
|
|   shutdown(self)
|       Shutdown the autoregister application. Stops the mainloop and
|       tears down helper modules.
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
```

```
| __weakref__
|     list of weak references to the object (if defined)
```

FUNCTIONS

```
main(args)
    Main entry point
```

3.4.3 external_image.py

The data structure to hold an image received from an external source. Adapted from code originally developed by Satra Ghosh for the MURFI project.

Help on module external_image:

NAME

```
external_image - Storage and I/O for an image received from an external sender.
```

FILE

```
/home/ohinds/projects/auto_register/src/external_image.py
```

CLASSES

```
__builtin__.object
    ExternalImage
```

```
class ExternalImage(__builtin__.object)
|   Datastructure representing an image that has been sent to us by an
|   external sending application, usually an MRI scanner or test tool
|   simulating one.
|
|   Methods defined here:
|
|   __init__(self, typename, format_def=[('magic', '5s'), ('headerVersion', 'i'), ('series
|       Initialize the image data structure and helper variables.
|
|   create_header(self, img, idx, nt, mosaic)
|       Create a default dummy header.
|
|   from_image(self, img, idx, nt, mosaic=True)
|       Convert an ExternalImage instance into a header/image pair, both in
|       byte strings suitable for sending to an external receiver.
|
|   get_header_size(self)
|
|   get_image_size(self)
|
|   hdr_from_bytes(self, byte_str)
|       Unpack a byte string received from an external source and fill in
|       the image header info it represents.
```

```

|
|  hdr_to_bytes(self, hdr_info)
|      Convert the image header data into a string of bytes suitable for
|      sending to an external receiver.
|
|  make_img(self, in_bytes)
|      Convert a byte string received from an external sender into image
|      data.
|
|  process_header(self, in_bytes)
|      Convenience function to convert a string of bytes into an image
|      header. Performs rudimentary validation on a received byte
|      string to make sure it's from a source we recognize.
|
|  process_image(self, in_bytes)
|      Convenience function to convert a string of bytes into
|      image data. Merely passes through to 'make_img'.
|
|  -----
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  -----
|  Data and other attributes defined here:
|
|  struct_def = [('magic', '5s'), ('headerVersion', 'i'), ('seriesUID', '...

```

FUNCTIONS

```

demosaic(mosaic, x, y, z)
    Convert a mosaic 2D image into a 3D volume

```

```

mosaic(data)
    Convert a 3D volume into a mosaic 2D image

```

```

sleep(...)
    sleep(seconds)

```

Delay execution for a given number of seconds. The argument may be a floating point number for subsecond precision.

3.4.4 image_receiver.py

The helper module that runs a TCP/IP server to listen for incoming images from an external computer. When it receives an image, it is saved and the filename stored so other modules can request it. Adapted from code originally developed by Satra Ghosh for the MURFI project.

Help on module image_receiver:

NAME

image_receiver

FILE

/home/ohinds/projects/auto_register/src/image_receiver.py

DESCRIPTION

Receive images sent to a TCP server. Also save the images and make the filename available through an external interface.

CLASSES

`__builtin__.object`
ImageReceiver

```
class ImageReceiver(__builtin__.object)
|   Run a TCP server, receive images, and save them.
|
|   Methods defined here:
|
|   __init__(self, args)
|       Store arguments, determine first available image name so we don't
|       overwrite existing images, and create a template image header for saving.
|
|   get_next_filename(self)
|       If there is a new image file available, return it and remove the
|       filename from those available.
|
|   is_running(self)
|
|   process_data(self, sock)
|       Callback to receive image data when it arrives.
|
|   save_nifti(self, img)
|       Save a received image to a file.
|
|   start(self)
|
|   stop(self)
|
|   -----
|   Data descriptors defined here:
```

```
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

FUNCTIONS

```
main(argv)
    Main entry. Just starts the server and waits for it to finish.
```

USED IN STANDALONE MODE ONLY

```
parse_args(args)
    Parse command line arguments.
```

USED IN STANDALONE MODE ONLY

```
sleep(...)
    sleep(seconds)
```

Delay execution for a given number of seconds. The argument may be a floating point number for subsecond precision.

3.4.5 registered_image.py

The helper module that calls out to the external registration program (`mri_robust_register`) and parses the resulting transformation file. It makes both the filename and an affine matrix representation available to other modules.

Help on module `registered_image`:

```
NAME
    registered_image
```

```
FILE
    /home/ohinds/projects/auto_register/src/registered_image.py
```

DESCRIPTION

Register two images by calling out to a helper executable (default is `mri_robust_register`). The resulting transformation is read from the output file generated by the registration executable, and made available to other modules.

This file can also be used in standalone mode for testing or reproducing the `AutoRegister` behavior.

CLASSES

RegisteredImage

```
class RegisteredImage
|   Class that registers two images and stores info about the registration.
|
|   Methods defined here:
|
|   __init__(self, reference, movable, opts=None, verbose=False)
|       Setup for performing registrations.
|
|   get_transform(self)
|       Retrieve a 4x4 numpy matrix representing the most recently computed
|       transform.
|
|   get_transform_filename(self)
|       Return the name of the most recently computed transform.
|
|   register(self)
|       Call out to the external program to register the reference and
|       movable images.
|
|   -----
|   Class methods defined here:
|
|   check_environment(cls) from __builtin__.classobj
|       Make sure that our environment is able to execute the registration
|       program.
|
|   read_transform_file(cls, filename) from __builtin__.classobj
|       Read and LTA file and parse the transformation it contains.
```

FUNCTIONS

```
main(argv)
    During standalone operation, build all the arguments that would
    normally be passed in from the calling module and pass them into a
    RegisteredImage class instance.
```

3.4.6 tcpip_server.py

A simple, generic TCP/IP server. This is used by both ImageReceiver and TransformSender to send and receive data from the MRI scanner or simulator. Adapted from code originally developed by Satra Ghosh for the MURFI project.

Help on module tcpip_server:

NAME

tcpip_server - Threaded TCP/IP server that will notify a callback on a new connection.

FILE

/home/ohinds/projects/auto_register/src/tcpip_server.py

CLASSES

```
SocketServer.BaseRequestHandler
    ThreadedTCPRequestHandler
SocketServer.TCPServer(SocketServer.BaseServer)
    ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer)
SocketServer.ThreadingMixIn
    ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer)
```

```
class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler)
|   Simply passes received data through to the specified callback
|   function
|
|   Methods defined here:
|
|   __init__(self, callback, *args, **keys)
|
|   handle(self)
|
|   -----
|   Methods inherited from SocketServer.BaseRequestHandler:
|
|   finish(self)
|
|   setup(self)
|
class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer)
|   TCPIP server. Simply spins up on init and spawns a thread to handle
|   new connections.
|
|   Method resolution order:
|       ThreadedTCPServer
|       SocketServer.ThreadingMixIn
|       SocketServer.TCPServer
|       SocketServer.BaseServer
|
|   Methods defined here:
|
|   __init__(self, address, callback)
|
|   is_running(self)
|
|   -----
|   Methods inherited from SocketServer.ThreadingMixIn:
|
|   process_request(self, request, client_address)
|       Start a new thread to process the request.
|
```

`process_request_thread(self, request, client_address)`

Same as in `BaseServer` but as a thread.

In addition, exception handling is done here.

Data and other attributes inherited from `SocketServer.ThreadingMixIn`:

`daemon_threads = False`

Methods inherited from `SocketServer.TCPServer`:

`close_request(self, request)`

Called to clean up an individual request.

`fileno(self)`

Return socket file number.

Interface required by `select()`.

`get_request(self)`

Get the request and client address from the socket.

May be overridden.

`server_activate(self)`

Called by constructor to activate the server.

May be overridden.

`server_bind(self)`

Called by constructor to bind the socket.

May be overridden.

`server_close(self)`

Called to clean-up the server.

May be overridden.

`shutdown_request(self, request)`

Called to shutdown and close an individual request.

Data and other attributes inherited from `SocketServer.TCPServer`:

`address_family = 2`

`allow_reuse_address = True`

request_queue_size = 5

socket_type = 1

Methods inherited from SocketServer.BaseServer:

finish_request(self, request, client_address)
 Finish one request by instantiating RequestHandlerClass.

handle_error(self, request, client_address)
 Handle an error gracefully. May be overridden.

 The default is to print a traceback and continue.

handle_request(self)
 Handle one request, possibly blocking.

 Respects self.timeout.

handle_timeout(self)
 Called if no new request arrives within self.timeout.

 Overridden by ForkingMixIn.

serve_forever(self, poll_interval=0.5)
 Handle one request at a time until shutdown.

 Polls for shutdown every poll_interval seconds. Ignores
 self.timeout. If you need to do periodic tasks, do them in
 another thread.

shutdown(self)
 Stops the serve_forever loop.

 Blocks until the loop has finished. This must be called while
 serve_forever() is running in another thread, or it will
 deadlock.

verify_request(self, request, client_address)
 Verify the request. May be overridden.

 Return True if we should proceed with this request.

Data and other attributes inherited from SocketServer.BaseServer:

timeout = None

FUNCTIONS

`handler_factory(callback)`

Standalone function to serve as a callback proxy for spawning a thread to handle a new connection.

3.4.7 `terminal_input.py`

The helper module that allows sane user input. At the moment, this module is only used to listen for a shutdown signal.

Help on module `terminal_input`:

NAME

`terminal_input`

FILE

`/home/ohinds/projects/auto_register/src/terminal_input.py`

CLASSES

`--builtin--.object`

`TerminalInput`

`class TerminalInput(--builtin--.object)`

| Allow the user to interact with a terminal in sane ways.

|

| Methods defined here:

|

| `--init__(self, disabled)`

| Perform initialization by setting up threading and setting
| attributes of the terminal to stop echo and exit canonical
| mode. Setting the disabled flag to True will stop the terminal
| attributes from being set. This is useful for debugging, as
| you can't input sanely to a debugger when the attributes are
| set.

|

| `get_char(self)`

| Retrieve the next available character that was input by the user,
| or None if there was no character since the last call.

|

| `run(self)`

|

| `start(self)`

|

| `stop(self)`

|

| -----
| Data descriptors defined here:

|

```
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)
```

3.4.8 transform_sender.py

The helper module that sends transformation data to the MRI scanner or simulator. It runs a TCP/IP server that listens for "ping" requests from the external client, and sends a transformation on receiving a ping only if there are any transformations in the send queue.

Help on module transform_sender:

NAME

transform_sender

FILE

/home/ohinds/projects/auto_register/src/transform_sender.py

CLASSES

```
--builtin__.object
    TransformSender
```

```
class TransformSender(--builtin__.object)
| Sends a transformation to an external computer. At the moment, this
| transformation is only suitable for use in the Siemens AutoAlign
| system (or other systems that use a compatible coordinate system).
|
| This "sender" is actually a TCP server that listens for "ping"
| requests from an external receiver. When a transform is available
| to send, the server responds to the ping request by transmitting
| the transform.
|
| Methods defined here:
|
| __init__(self, host, port)
|     Initialize the networking parameters and internal state.
|
| clear_state(self)
|     Set the internal state to an empty string (allowing transformations
|     to be sent).
|
| process_data(self, sock)
|     Callback when a "ping" request is received. This checks that the
|     request bytes are actually "ping", and replies with the string
|     "none" if there are no transforms queued for sending, and
|     otherwise sends the first transform available.
```



```

|
| send(self, transform)
|     Queue a transform for sending to an external requestor.
|
| set_state(self, state)
|     Set the internal state of the sender. If the state is anything
|     other than an empty string, no transforms will be sent. This
|     is useful for disabling sending old transforms while an image
|     registration or network transfer is in progress.
|
| start(self)
|
| stop(self)
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)

```

3.5 Tools

3.5.1 vsend_nii

The C++ tool `vsend_nii` simulates the image sending functionality of the MR Image reconstruction module. This was used in development and testing of the AutoRegister Registration module, and can be used to reproduce the results of executing `auto_register.py` while not at an MRI scanner.

`vsend_nii` depends on the libraries ACE (<http://www.cs.wustl.edu/~schmidt/ACE.html>) and niftilib (<http://niftilib.sourceforge.net/>). Both of these packages are available in the Ubuntu repositories.

To build the `vsend_nii` executable, change to the directory `tools/scanner_sim` and execute `make`. To execute it, run the binary from the command line, passing the path to a NIFTI image file as an argument.

3.6 Tests

TODO

4 MR Image reconstruction module

The MR Image reconstruction module reconstructs the raw k-space data from the MRI scanner into the AutoRegister Scout image, sends this image to the Registration module, listens for a transformation in response, and makes the transformation available for application to future scans through the Siemens AutoAlign system.

4.1 Repository

`git@52.72.193.62:ohinds/ice_auto_register_interface.git`

4.2 Details

The image reconstruction module is implemented as an ICE functor called `IceAutoRegisterInterface`. This functor's two main responsibilities are to send an AutoRegister Scout to the Registration module, and to receive and pass on a resulting transformation.

4.2.1 Image sending

The image sending functionality in the `IceAutoRegisterInterface` is based on the `VSend` functor, which was developed to send EPI volumes to a real-time fMRI system. `VSend` was in turn based on the `SendExternal` Siemens example program.

The `IceAutoRegisterInterface` builds up a volume slice-by-slice, each time the functor method `ComputeImage` is called. The last time `ComputeImage` is called (just after the last slice is acquired), the functor adds the final slice to the volume and sends the volume over a TCP/IP socket to the server running in the Registration module.

4.2.2 Transformation handling

After an image is sent, a wait loop is entered where a "ping" response is sent to the Registration module every second. While the response "registering" is received, the loop continues. When a string containing a free text version of a 4x4 matrix representation of an affine transformation is received, the transformation is saved, and `ComputeImage` is allowed to proceed.

The transformation is passed along to the Siemens AutoAlign system in the functor method `foreignDllCall`, which is called after `ComputeImage` returns. All that's required here is to copy the received transformation into a data structure provided by the AutoAlign system, and passed into `foreignDllCall`.

If this process completes without error, the Registration module-computed transformation will be available for application to further sequence protocols on the scanner host by choosing the AutoAlign -> Head Brain option in the AutoAlign setup.