CHAPTER

ONE

Introduction

Image registration is the process of determining the spatial transform that maps points from one image (the moving image) to corresponding points in the target image (the fixed image). The focus of the library is correspondence-based registration. The notion of a correspondence extends from traditional closest-point based matching [2] to recent hybrids of intensity-based and feature-based techniques [6, 15] where the matching of image intensities creates temporary correspondences. Correspondences, however they are generated, are used to refine the transformation estimate.

The most well-developed and widely-known toolkit for medical image registration is the Insight Toolkit (ITK) [12]. ITK contains techniques for medical imaging processing, segmentation, and registration. The focus of the ITK registration framework is intensity-based methods. RGRL complements that of ITK by emphasizing correspondence-based registration. RGRL compiles with ITK, allowing components of each toolkit to be used in any particular program.

RGRL is written in C++ and built on top of the VXL libraries [23] which were designed for development of computer vision applications. To fully understand examples in RGRL, prior knowledge in object-oriented programming and memory management using smart pointer (`vxl/vbl/vbl_smart_ptr`) is required. VXL supports multi-platforms through the use of CMake (http://www.cmake.org), a cross-platform, open-source build system. To compile VXL, the version of CMake should be 1.6 or higher.

For any suggestions and comments, please email Chuck Stewart (stewart@cs.rpi.edu) or Charlene Tsai (tsaic@cs.rpi.edu).

CHAPTER

TWO

General Correspondence-Based Architecture

2.1 Overview

The RGRL library implements a general framework for correspondence-based registration. Components of the framework include initializers, feature sets, correspondence generators, robust transformation estimators, and convergence testers. Multiple feature types may be used either simultaneously or sequentially, and multiresolution is handled naturally. The role of each component is isolated by being designed as an abstract base class, with a wide variety of techniques being implemented as derived classes. Variations on the general framework, including adaptive transformation model selection and registration region growing [22], are facilitated by the notion of a view, first introduced in [21]. The overall design goal of the library is to combine simplicity for education on image registration and flexibility for development of new registration techniques that fall within the general class of correspondence-based approaches.

The architecture has an inner flow loop (Figure 2.1) implementing a basic but flexible correspondence framework and a set of outer loops controlled by a "registration engine" that allow for multiple initial estimates, multiple feature types, and multiple stages (resolutions) to the registration process. We briefly discuss the functionality of each core component of the inner flow loop below:

Feature/Feature set: Features, pre-computed or constructed dynamically, are the basic primitives of correspondence-based registration. A feature set is a collection of features together with representations suitable for spatial queries such as finding the nearest neighbors.

Initialization: Registration is often viewed as a chicken-and-egg problem. If corresponding features are known, an estimate can be obtained by minimizing the errors between the corresponding features. If the estimate is known first, we can generate the matches by transforming one image to another. However, in reality, none of them is known a priori (else, we're done). To break the circle, the application should be able to provide a crude

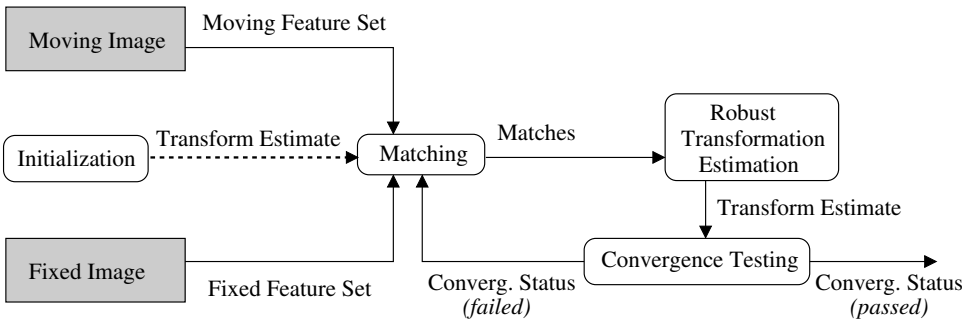


Figure 2.1: Architecture of the inner loop of general correspondence-based registration. Each computational component, enclosed in the box, takes one or more representation objects as input, and produces one representation object as output, indicated by the arrow. All representation objects constructed stay in the loop until re-computed. The dashed arrow indicates the ow of object which is only used once for initialization.

estimate of the transformation and other information, such as the accuracy of the estimate. This is termed initialization. A variety of initialization methods are implemented.

Matching: The matching process generates correspondences from the two image feature sets. The details depend on the feature type as well as on the type of matching, such as nearest neighbor or most similar appearance.

Robust transformation estimation: This process estimates the spatial relationship that maps the moving image to the fixed image using the matches. The goal is to minimize the alignment error of the two feature sets in the presence of outliers/mis-matches.

Convergence testing: The registration process terminates when the estimation has converged to a stable solution or is determined to be oscillating.

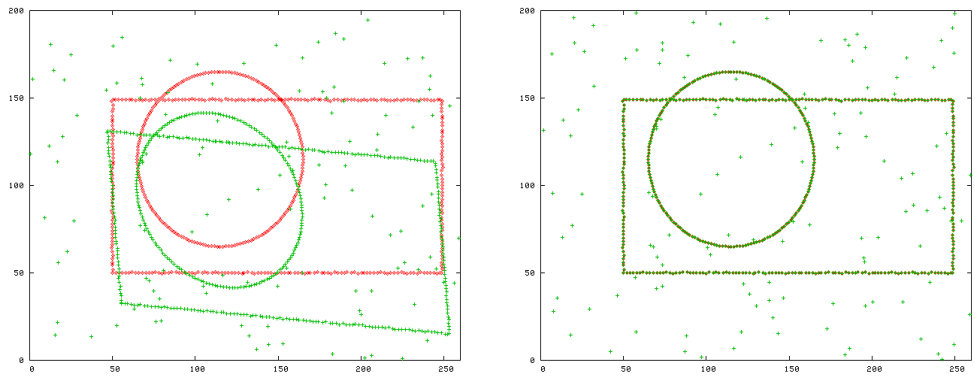
Each process involves computational objects and representation objects. The computational objects take one or more representation objects as input and produce a representation object as output. Due to the clean separation between computational and presentation objects, each object can be removed from the framework and used separately (see section 2.4.1 for an example).

2.2 Features and Feature Sets

2.2.1 “Hello World” Registration

The source code of this section can be found in the file
`rgrl/examples/registration_blackbox.cxx`

This example is the ”Hello World” example for correspondence-based registration. The goal of the example is to show steps needed to set up the registration process. For now, we treat



(a) (b)

Figure 2.2: Registration of two simple images. The goal is to register the green to the red image. (a) is the initial alignment, and (b) is the result of correct registration.

the registration engine as a black box (which is not recommended for any but the simplest problems). Please note, most components in the registration library are not templated over the dimension of the image, since the dimension can be easily derived from the feature set.

The problem that we're interested in now is to register two images with some simple shapes. One image is in red and one in green. Both images have Gaussian noise added, and the green image contains some random points which are not present in the red image. Good registration should overlay the green on the red image with identity transformation[1]. To make the problem more interesting, we start with an initial transformation involving slight rotation, shearing and translation (affine transformation), shown in figure 2.2(a). The final alignment with a close to identity transformation is in figure 2.2(b).

The following header files provide declarations for the basic types needed to perform the registration.

```
#include <rgrl/rgrl_feature_trace_pt.h>
#include <rgrl/rgrl_feature_set_location.h>
#include <rgrl/rgrl_trans_affine.h>
#include <rgrl/rgrl_est_affine.h>
#include <rgrl/rgrl_data_manager.h>
#include <rgrl/rgrl_feature_based_registration.h>
```

We start with features. A *feature* is the basic element for correspondence-based registration. It can be as simple as a point in space with a location, or as complex as a region with intensity information and orientation information encapsulated. Features can be constructed before matching is started, or in the case of matching intensities (block matching), constructed as part of the matching process.

[1] Different from ITK, the transformation maps the moving image to the fixed image.

Error Type	point-to-point	point-to-curve	point-to-surface
Error Projector	$\mathbf{M} = \mathbf{I}$	$\mathbf{M} = \mathbf{I} - \hat{\mathbf{t}}\hat{\mathbf{t}}^T$	$\mathbf{M} = \hat{\eta}\hat{\eta}^T$
DoF to Constrain	m	m-1	1

Table 2.1: Error projector matrix for different types of errors. For a point on a curve, $\hat{\mathbf{t}}$ is the tangent to the curve, and for point on a surface, $\hat{\eta}$ is the normal to the surface. *m* is the number of dimensions of the image.

Different feature types should define different alignment error distance measures (see evaluation in [16]). For example, the alignment error of landmarks matched to landmarks is best described as a Euclidean distance, since landmarks are fairly stable 3D point locations. For "non-distinctive" points on curves and surfaces, however, this is too restrictive a measure because the features are often poorly localized in the direction along the curve or on the surface. Thus, the error in the direction normal to the curve or surface is the most appropriate error distance measure. This is achieved most generally by pre-multiplying the standard Euclidean error vector by an *error projector matrix*, and then computing the magnitude of the projected vector. Error projectors of different types are summarized in Table 2.1.

In this example, we prefer point-to-curve as our error measure, since sampled points are less distinctive along the lines and the curve. The feature type is `rgrl_feature_trace_pt` and the code below illustrates the construction of such a feature.

```
vector_2d pt, tangent_dir;
pt[0] = org_x + xi + random.normal()*sigma;
pt[1] = org_y + random.normal()*sigma;
tangent_dir[0] = 1;
tangent_dir[1] = 0;
rgrl_feature_sptr feature_pt = new rgrl_feature_trace_pt(pt, tangent_dir);
feature_points.push_back( feature_pt );
```

Since we're dealing with 2D images, both `pt` for the location of the point and `tangent_dir` for the direction of the tangent are both 2D vectors. Let's not to worry how `pt` and `tangent_dir` are generated, but concentrate on construction of the feature. `rgrl_feature` is the abstract base class of various feature types, and it provides the common interface for interaction with other components of the registration library. `rgrl_feature_sptr` defines a smart pointer of type `rgrl_feature` (`_sptr` stands for smart pointer). `rgrl_feature_sptr` can also store a pointer to a derived class. Here we create an object of type `rgrl_feature_trace_pt` with the point location and the approximated tangent direction. A pointer of such an object is assigned to `rgrl_feature_sptr`, which is stored in a vector, `feature_points`.

All features of the same type from one image are stored in a *feature set*. Fundamentally, feature sets answer queries in support of correspondence generation. Examples include finding the nearest feature in the fixed image to a given feature (mapped from the moving image), finding the k-nearest neighbors, or even finding all features in a given region when multiple correspondences are required for a feature (see for example [10]). Spatial data structures

such as k-d trees and binning support fast querying. In this example, the type of feature set is `rgrl_feature_set_location`, which answers the queries based on the feature location. The construction of a feature set `rgrl_feature_set_location` takes a vector of smart pointers to features, which we previously constructed.

```
const unsigned int  dimension = 2;
rgrl_feature_set_sptr moving_feature_set =
  new rgrl_feature_set_location<dimension>(moving_feature_points);
rgrl_feature_set_sptr fixed_feature_set =
  new rgrl_feature_set_location<dimension>(fixed_feature_points);
rgrl_mask_box image_roi = moving_feature_set->bounding_box();
```

Next, we initialize the registration process with a prior affine transformation, and define the affine estimator. More detailed discussion on other common initialization schemes will be discussed in section 2.4.

```
vnl_matrix<double> A(2,2);
A(0,0) = 0.996;   A(0,1) = -0.087;
A(1,0) = -0.087;  A(1,1) = 0.996;
vector_2d t( 10, -13 );
rgrl_transformation_sptr init_transform = new rgrl_trans_affine(A, t);
rgrl_estimator_sptr      estimator = new rgrl_est_affine();
```

All the components required for registration are stored in a data structure, `rgrl_data_manager`. This is useful for illustration purposes, but not for many real problems. To make it more useful, the data manager also supports robust estimation, multi-resolution and multi-model registration (topics to be discussed later). In the simplest form, the data manager takes only the two feature sets. The data manager is passed to the registration engine, `rgrl_feature_based_registration` during construction.

```
rgrl_data_manager_sptr data = new rgrl_data_manager();
data->add_data( moving_feature_set, fixed_feature_set );
rgrl_feature_based_registration reg( data );
```

Now, we're ready to perform the registration. With the same sets of features, we may obtain different results depending on the region of interest (`image_roi`), transformation estimator (`estimator`) and initial transformation estimate (`init_transform`). Please note, the initial estimate is only for generation of the first set of matches. Therefore, it can belong to a model different from that of the transformation estimator.

```
reg.run( image_roi, estimator, init_transform );
```

2.2.2 Range Data Registration - Another Example

The source code of this section can be found in the file
`rgrl/examples/registration_range_data_blackbox.cxx`

The goal of this example is to demonstrate registration of 3D range data using `rgrl_feature_face_pt` as the feature type. Features are points located on 3D surfaces. Each point is associated with a direction normal to the surface at the given point. If normal directions are not provided, they can be easily estimated at each point location by gathering m closest points and fitting a plane. Implementation of triangularization algorithms is part of the future work.

The set of header files included is very similar to the previous example. Since we're interested in point-to-surface error, the only difference is the feature type — `rgrl_feature_face_pt`, instead of `rgrl_feature_trace_pt`.

```
#include <rgrl/rgrl_feature_face_pt.h>
#include <rgrl/rgrl_feature_set_location.h>
#include <rgrl/rgrl_trans_affine.h>
#include <rgrl/rgrl_est_affine.h>
#include <rgrl/rgrl_data_manager.h>
#include <rgrl/rgrl_feature_based_registration.h>
```

In this example, feature locations and associated normals are imported from a file. The construction of `rgrl_feature_face_pt` takes a location and a normal (both are 3D vectors).

```
    rgrl_feature_sptr feature_pt = new rgrl_feature_face_pt(location, normal);
    feature_points.push_back( feature_pt );
```

The default underlying data structure for `rgrl_feature_set_location` is N-D bins, which, on average, is quite fast for sparse and scattered data, but inadequate for clustered data (unless the bin size is adjusted accordingly). In the case of range data, points are normalized to the range of $[-1,1]$. To better handle the clustered data, kd-tree should be the choice of data structure for faster performance. During the construction of `rgrl_feature_set_location`, the kd-tree is selected if the 2nd input parameter is set to false (`!use_bins`).

```
  const unsigned int dimension = 3;
  bool use_bins = true;
  rgrl_feature_set_sptr moving_feature_set =
    new rgrl_feature_set_location<dimension>(moving_feature_points, !use_bins);
  rgrl_feature_set_sptr fixed_feature_set =
    new rgrl_feature_set_location<dimension>(fixed_feature_points, !use_bins);
  rgrl_mask_box image_roi = moving_feature_set->bounding_box();
```

The rest of the program is extremely similar to `registration_blackbox.cxx`. To execute the example, one may try the feature file `rgrl/examples/dragonStandRight_0_normal.txt`. The original data is from http://graphics.stanford.edu/data/3Dscanrep/. Normals were estimated separately and stored in `dragonStandRight_0_normal.txt` with the point locations. The result of the execution is a transformation very close to identity.

2.3 Robust Registration

The source code of this section can be found in the file
`rgrl/examples/registration_simple_shapes_outliers.cxx`

It is never a good idea to treat the registration engine as a black box, and expect the default techniques to work robustly for images with noise. In this section, we'll look at components which were previously hidden in the registration engine.

Back to the same registration problem as for section 2.2.1. The purpose of the example is to demonstrate customized robust registration. The focus of the discussion is on matching, robust transformation estimation, and convergence testing. We'll skip the part on construction of feature sets and initialization of the transformation (which were explained in section 2.2.1).

The *matcher* is responsible for computing the correspondences and constructing a *match set* data structure to store them. One of the derived class of the matcher is `rgrl_matcher_k_nearest`. This matcher can use different "nearness" measures, such as Euclidean distance and appearance-based similarity measures [6, 18], to determine the k nearest neighbours (allowing multiple correspondences per feature). When $k = 1$ and "nearness" is defined by Euclidean distance, this becomes the classical ICP matcher [2, 5, 24], as used in this example.

```
unsigned int k = 1;
rgrl_matcher_sptr cp_matcher = new rgrl_matcher_k_nearest( k );
```

To robustly handle outliers (wrong matches) generated by the matcher, proper weights should be assigned to the matches, such that matches with higher errors have less weight. "Weighter" objects compute the weights for each match and store them back with the matches in the match set. Several types of weight calculations are implemented, including standard M-estimator weighting, weighting based on feature similarities, and competitive weighting, if there are multiple matches for each feature.

The M-estimator weight function depends on the loss function for the estimation of the transformation [20]. The choice of loss function, `m_est_obj`, is motivated by looking at several different loss functions. and their associated weight functions illustrated in Figure 2.3. The least-squares loss function has a constant weight (unit weight), the Cauchy weight function descends and asymptotes at 0, while the Beaton-Tukey biweight function has a hard limit beyond which the weight is 0. This limit is set to about 4 error scale (robust standard deviation of the alignment errors) in the statistics literature [11]. This is important for rejecting errors due to mismatches, and therefore it is the one we choose here. Implementation of a variety of loss functions can be found in `vxl/contrib/rpl/rrel`.

```
vcl_auto_ptr<rrel_m_est_obj>  m_est_obj( new rrel_tukey_obj(4) );
rgrl_weighter_sptr wgter = new rgrl_weighter_m_est(m_est_obj, false, false);
```

The formulation of a weight function is $w(u) = \rho'(u)/u$, where $\rho(u)$ is the loss function and u is the registration error measured in error scale (scale normalized residual). There are several

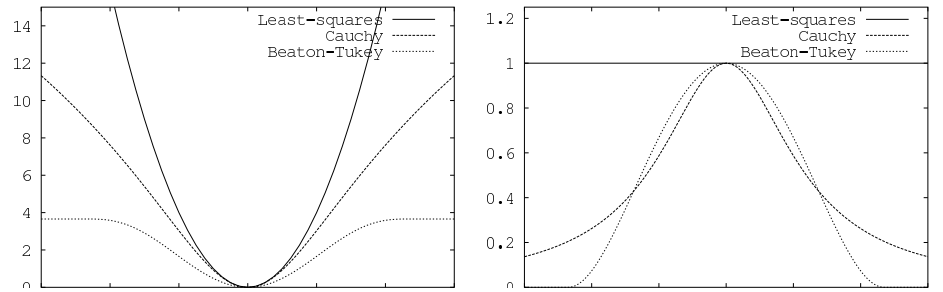


Figure 2.3: Plots of (a) the robust loss function $\rho(u)$ and (b) weight function $w(u) = \rho'(u)/u$ for the Beaton-Tukey biweight loss function, the Cauchy loss function and the quadratic loss function, which equates to least-squares estimation. The Beaton-Tukey is chosen because it most aggressively rejects outliers, providing no weight to matches with normalized distances greater than about 4 standard deviations.

ways to compute the scale, and the methods can be classified into weighted and unweighted. Weighted scale estimator has precedence over unweighted, but it requires initialization (to get the first set of weights). This can be done either by providing a prior scale to the initializer, or by using an unweighted scale estimator. In the example, we take the second approach by declaring both types of scale estimator.

```
vcl_auto_ptr<rrel_objective> muset_obj( new rrel_muset_obj(0, false) );

rgrl_scale_estimator_unwgted_sptr unwgted_scale_est =
  new rgrl_scale_est_closest( muset_obj );
rgrl_scale_estimator_wgted_sptr wgted_scale_est =
  new rgrl_scale_est_all_weights();
```

The unweighted scale estimator uses MUSE [13], which is similar to MAD (Median Absolute Deviation), but much more robust. It automatically determines the fraction of (approximately) correct matches.

A *convergence tester* determines whether the registration process should terminate for the current initial estimate. The result of testing is stored in a *convergence status* object. Two termination criteria are currently implemented: (1) the estimation has converged with the change of alignment error below certain threshold, and (2) the estimation is oscillating. Derived classes of the convergence tester include weighted and unweighted. The alignment error computed by the weighted tester is the weighted average of the errors, while the unweighted tester returns the median error. The input parameter, `tolerance`, to `rgrl_convergence_on_weighted_error` does not affect the termination of the registration process, but allows the result to be labeled as acceptable or not.

```
double tolerance = 1.5;
```

Component	RGRL Class	Technique
matcher	rgrl_matcher_k_nearest	ICP (k=1)
weighter	rgrl_weighter_m_est	Beaton-Tukey Bi-weight
unweighted scale estimator	rgrl_scale_est_closest	MAD (Median Abs. Dev.)
convergence tester	rgrl_convergence_on_median_error	Median error

Table 2.2: Default techniques for major components if not set

```
rgrl_convergence_tester_sptr conv_test =
  new rgrl_convergence_on_weighted_error( tolerance );
```

As before, all the components, except the convergence tester, are stored in the data manager. Table 2.2 contains the list of default techniques for the major components if not set.

```
rgrl_data_manager_sptr data = new rgrl_data_manager();
data->add_data( moving_feature_set, // data from moving image
                fixed_feature_set,  // data from fixed image
                cp_matcher,         // matcher for this data
                wgter,              // weighter
                unwgted_scale_est,  // unweighted scale estimator
                wgted_scale_est);   // weighted scale estimator
```

Construction of the registration engine takes the data manager and the convergence tester. The Engine also allows adjustment of some parameters, such as `set_max_icp_iter()` to limit the maximum amount of looping in Figure 2.1.

```
rgrl_feature_based_registration reg( data, conv_test );
reg.set_expected_min_geometric_scale( 0.1 ); //scale cannot go lower than 0.1
reg.set_max_icp_iter( 10 );
```

2.4 Initialization

Initialization plays an important role in registration. A reliable initialization technique alleviates the problem of narrow domain of convergence when numerous local minima of the objective function are present. Up to now, we have been only dealing with registration problems with manual initialization. RGRL provides mechanisms to perform automatic initialization via the use of an *initializer*. The interface to the initializer object defines only the request to provide the next initial estimate at the start of the iterative process illustrated in Figure 2.1. This isolates initialization from the rest of the algorithm. Currently, two types of initializers have been implemented: a random-sampling initializer, and an invariant-indexing initializer that matches invariant image features [22]. We demonstrate both initializers using retinal images.

2.4.1 Random Sampling

The source code of this section can be found in the file
`rgrl/examples/registration_landmark.cxx`

The registration problem is to align the retinal images using the pre-computed landmarks. A landmark feature is defined as a branching or cross-over point of a vascular-like structure. It is characterized by a number of outgoing direction vectors. To make it more general, any feature with a center location and a set of outgoing direction vectors can be a "landmark", such as a corner. The implementation of a landmark is `rgrl_feature_landmark`.

This example demonstrates a simplified implementation of the hierarchical pairwise registration algorithm in [4]. Characteristics of the algorithm include the use of landmarks, multiple matches enhanced by signature similarity, and random sampling [8] for initialization, which is the focus of this section.

The simplified hierarchical pairwise registration algorithm consists of the following steps:

1. Manually guess the initial translation (need not be very accurate).
2. Based on the initial translation, generate 2 matches for each landmark in the moving image. A distance threshold maybe applied to further reduce the number of matches.
3. Run random sampling on the candidate match set to obtain the best affine transformation. Since the estimate is computed using only 3 matches, refinement of the transformation is necessary.
4. Refine the affine transformation using Iteratively-Reweighted Least-Squares (IRLS). IRLS is handled by the registration engine, and it takes the original feature sets. The transformation model for the engine is quadratic, which is the final model of the application.

Landmarks are first imported from files. With the initial translation estimate (`init_translation`), we generate 2 closest matches, in terms of the Euclidean distance, for each landmark in the moving image. This task is handled by the k-nearest matcher, which is one of the major components in the registration engine. In this example, the matcher operates outside the engine. The `pruned_match_set`, returned by the matcher, contains matches for random sampling.

```
int k = 2;
rgrl_matcher_sptr cp_matcher = new rgrl_matcher_k_nearest( k );
rgrl_match_set_sptr pruned_match_set =
  cp_matcher->compute_matches(*moving_feature_set,
                              *fixed_feature_set,
                              *init_translation,
                              moving_image_region,
                              *dummy_scale);
```

Random sampling [19] is performed by `rgrl_initializer_ran_sam` . It is crucial to provide the transformation estimator with the dimension of the image. This allows the initializer to compute the minimum number of matches to constrain a transformation. Another necessary component for random sampling is the unweighted scale estimator (`unwgted_scale_est`), which is needed to measure the error for each fit. All elements are passed into the initializer via a function call `set_data`. The current implementation only handles a pre-computed match set.

```
int dim = 2;

rgrl_estimator_sptr affine_estimator = new rgrl_est_affine(dim);

vcl_auto_ptr<rrel_objective> obj_fun( new rrel_lms_obj(1) );
rgrl_scale_estimator_unwgted_sptr unwgted_scale_est =
  new rgrl_scale_est_closest( obj_fun );

rgrl_initializer_ran_sam* ran_sam = new rgrl_initializer_ran_sam();
ran_sam->set_data(pruned_match_set,
                  unwgted_scale_est,
                  moving_image_region,
                  affine_estimator,
                  0 );
rgrl_initializer_sptr initializer = ran_sam;
```

The estimation of the best affine transformation is triggered when the registration engine asks for an initial transformation. Only one initial estimate is provided by random sampling initializer, since the estimation can only be triggered once.

We skip the part on the setting of the registration engine for robust estimation of the final quadratic transformation. Different from previous examples, the initializer, instead of the initial estimate, initiates the registration process.

```
reg.run( initializer );
```

Since the initial translation prior to random sampling was specified manually, this example only works with specific feature files, `rgrl/examples/IMG0004.pgm.landmarks.txt` for the fixed image, and `rgrl/examples/IMG0002.pgm.landmarks.txt` for the moving image. The mask image is `rgrl/examples/mask.png`. Running the example with the given input files, the final result is

```
Final xform:
Q =
0.000101208 0.000103199 9.30317e-06
1.73496e-05 2.33522e-05 -5.01135e-06
A = 0.858602 -0.0974608
-0.0434711 0.972142
t = -208.821 -10.0152
```

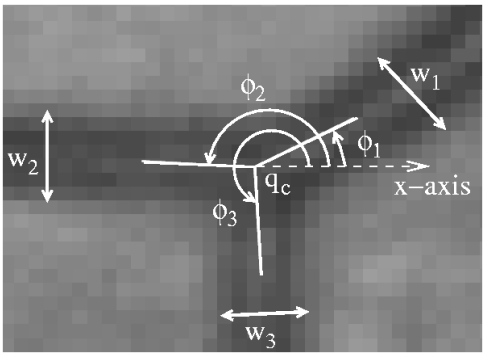


Figure 2.4: A landmark is characterized by a center location **c**, the orientations of the three blood vessels that meet to form it and the widths w_j of the blood vessel. Differences in orientations and the ratios of the blood vessel widths are invariant to rotation, translation and scaling of the image. The orientations themselves are invariant to translation and scaling.

```
Final alignment error = 1.00458
```

2.4.2 Invariant Indexing

The source code of this section can be found in the file
`rgrl/examples/registration_retina_no_view.cxx`

With the same set of images as in section 2.4.1, we now align the images using both the landmarks and vessel center points. Different from the previous example, the landmarks are only for initialization using invariant indexing, which is the focus of this section. The initial transformation from the initializer is refined by the registration engine using the vessel center points as features.

A brief description of the invariant signature computation, matching, and initial estimation is as follows. Matches between two landmarks, one in each image, are generated by computing and comparing invariants [3, 14]. Invariants are blood vessel width ratios and blood vessel orientations (Figure 2.4), giving a five-component invariant signature vector. The first 2 elements are cartesian and the last 3 are angular invariants. The closest match is found for each signature vector. Additional matches are determined when the Mahalanobis distance between signature vectors is within a 95% confidence chi-squared uncertainty bound. The set is ordered by the chi-squared confidence levels. For each match, a similarity transformation is estimated from the landmark locations and the orientations and widths of the vessels that meet to form the landmarks.

The base class of a feature with invariant properties (also refer to as invariant feature) is `rgrl_invariant`. The invariant indexing initializer interacts with such features via calls for

cartesian and angular invariants. An invariant feature can also estimate the transformation from another feature of the same type. The derived class for a landmark invariant feature is `rgrl_invariant_single_landmark`.

```
rgrl_invariant_single_landmark* single =
  new rgrl_invariant_single_landmark( location, direction1,
                                      direction2, direction3,
                                      width1, width2, width3,
                                      angular_std, width_ratio_std );
```

Having all landmark invariant features stored in two lists, one for each image, we are ready to construct the initializer, which performs the matching and ordering of the matches. `add_data(.)` can be called multiple times if there are multiple invariant lists for an image. All matches are ordered by chi-squared confidence levels. Unlike `rgrl_initializer_ran_sam`, `rgrl_initializer_inv_indexing` can return more than one initial estimate. Each call to the initializer for the next initial estimate takes the next unexplored match from the list.

```
rgrl_initializer_inv_indexing* inv_initializer =
  new rgrl_initializer_inv_indexing( moving_image_region,
                                     fixed_image_region );
double angular_std = 5.5*vnl_math::pi/180;
double nn_radius = angular_std * vcl_sqrt(11.0704);//95% chi-sqr uncertainty bound
inv_initializer->add_data( fixed_landmark_set,
                           moving_landmark_set,
                           nn_radius );
rgrl_initializer_sptr initializer = inv_initializer;
```

Let's execute this example with the following feature files. `rgrl/examples/IMG0002.pgm.txt` contains traces and `rgrl/examples/IMG0002.pgm.landmarks.invariant.txt` contains landmarks of IMG0002.pgm. Similarly, `rgrl/examples/IMG0004.pgm.txt` and `rgr/examples/IMG0004.pgm.landmarks.invariant.txt` for IMG0004.pgm. It does not matter which image is the fixed and which is the moving, since the initialization process is automatic. This executable also takes a binary image, called mask image. Such an image is used to define the valid region of the retina image. The mask image is `examples/mask.png`. When mapping IMG0002 to IMG0004 (the same pair as in section 2.4.1), the registration process should succeed on the second initial estimate with the final result:

```
Final xform:
Q =
0.000101439 0.000101008 8.86866e-06
1.85556e-05 2.149e-05 -4.64715e-07
A = 0.859269 -0.0949512
-0.0495303 0.971695
t =-209.843 -7.68227
```

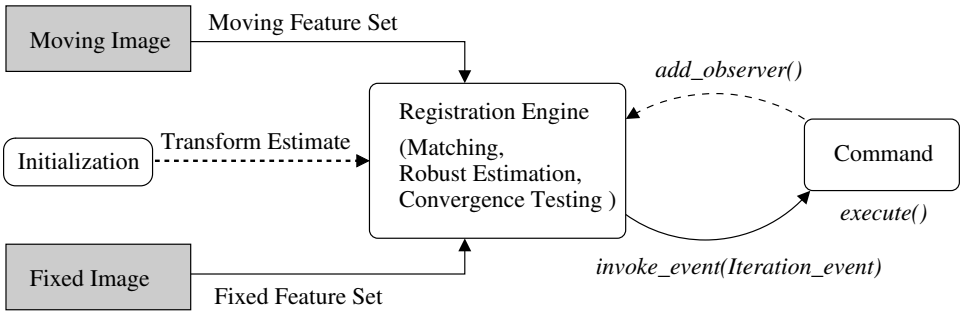


Figure 2.5: Interaction between the registration engine and the command object. The command is added to the engine via a call to add_observer() before the registration process begins, and is associated to the iteration event. In each iteration, the execute() function of the command object is triggered.

```
Final alignment error = 0.576973
```

The improvement in accuracy over the previous example is due to the use of vesel center points for refinement of the transformation.

2.5 Monitoring Registration

The source code of this section can be found in the file
`rgrl/examples/registration2.cxx`

It is a good practice to monitor the progress of the registration process. There are two ways of doing it. The first is the use of command/observer mechanism[9]. The second is the setting of the debug flag, which is a technique used by the developers of RGRL library. We'll only discuss the first approach in this section.

The command/observer design pattern depicts the interaction among 3 classes: `rgrl_object`, `rgrl_command`, and `rgrl_event`. `rgrl_object` is the base class of most objects in RGRL. This class maintains a list of event observers. Each event observer bundles a `rgrl_event` and `rgrl_command`, and is created via the call to `rgrl_object::add_observer()`. By doing so, the command registers itself with the object, declaring that it is interested in receiving notification when a particular event happens. Events defined in RGRL are `rgrl_event_start`, `rgrl_event_end`, `rgrl_event_iteration`.

Inside the registration engine, the iteration event (`rgrl_event_iteration`) is invoked at the end of each iteration. When an event is invoked by an object, this object goes through its list of observers and checks whether any one of them has expressed interest in the current event type. Whenever such an observer is found, its corresponding `execute()` method is invoked. In this context, `execute()` methods should be considered callbacks, and are assumed

to execute rapidly, for example, printing out a transformation. The following code illustrates a simple way of creating a command to monitor a registration process. This new class, called `command_iteration_update`, derives from `rgrl_command` and provides a specific implementation of the `execute()` methods.

```
class command_iteration_update: public rgrl_command
{
public:
  void execute(rgrl_object* caller, const rgrl_event & event )
  {
    execute( (const rgrl_object*) caller, event );
  }

  void execute(const rgrl_object* caller, const rgrl_event & event )
  {
    const rgrl_feature_based_registration* reg_engine =
      dynamic_cast<const rgrl_feature_based_registration*>(caller);

    if ( !reg_engine ) return;

    rgrl_transformation_sptr trans = reg_engine->current_transformation();

    if ( trans->is_type( rgrl_trans_quadratic::type_id() ) ) {
      rgrl_trans_quadratic* q_xform = rgrl_cast<rgrl_trans_quadratic*>(trans);
      vcl_cout<<"xform: Q \n"<<q_xform->Q()<<"A = "<<q_xform->A()<<
        "t = "<<q_xform->t()<<vcl_endl;
    }
    else vcl_cout<<"ERROR: Incorrect type!"<<vcl_endl;
  }
};
```

Before the registration process begins, the `command_iteration_update` is added to the observer list of the engine and specifies its interest in catching `rgrl_event_iteration`.

```
  reg.add_observer( new rgrl_event_iteration(), new command_iteration_update());
```

One may try running this example with the feature files `rgrl/examples/IMG0002.pgm.txt` and `rgrl/examples/IMG0003.pgm.txt`.

CHAPTER

THREE

Advanced Topics

3.1 Multi-Resolution Registration

The source code of this section can be found in the file
`rgrl/examples/registration3.cxx`

Multi-resolution approach is widely used to improve the domain of convergence, since there are fewer local minima present at the lower resolution. Different models can also be used at different resolutions for better stability [1, 7, 17]. Registration is first performed at a coarse scale (low resolution), and the result is used to initialize the process at the next finer scale.

Multi-resolution is handled naturally by the registration engine, as illustrated in Figure 3.1. A resolution is represented as a "stage". Components for each stage are stored in the data manager, same as previous examples, and the data manager handles data of multiple stages. The higher the index of the stage, the lower the resolution is. Stage 0 represents the original resolution.

In this example, we register the retinal images using 2 resolutions: the original resolution and the down-sampled. The image of the lower resolution is generated from the original using a down-sample factor of 2. We also assign different transformation models to different resolutions. An affine transformation model (6 parameters) is for the lower resolution and a quadratic model (12 parameters) is for the original. Starting at the lower resolution, the registration process is initialized with an *identity* transform. In this section, we'll only discuss steps that are relevant to multi-resolution registration.

We add data to each stage the same way as before. However, a few extra operations on the data manager are required for multi-resolution registration. First, it is necessary to specify the transition going from the lower to the higher resolution using `set_dimension_increase_for_next_stage` method. This method sets the parameter that allows the internal data to be scaled properly when moving to the next resolution level if feature locations are in image coordinate[1]. In this example, since the lower resolution image is half the size for each dimension of the original, the `dimension_increase` is set to 2 for the lower resolution. Second, the estimator for each stage is specified explicitly, since different models

[1] In ITK, such a parameter is not necessary, since the estimation is performed in physical coordinate.

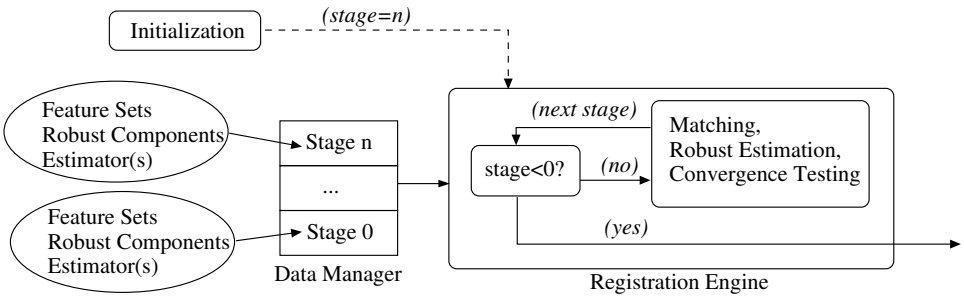


Figure 3.1: Multi-resolution registration. The diagram depicts the replationship among the registration engine, the initializer and the data manager. The initializer specifies the stage at which the registration begins. At each stage/resolution, the engine iterates among matching, robust estimation and convergence testing as illustrated in Figure 2.1. The registration process terminates when the stage is below 0, which is the highest resolution.

are desirable at different resolutions. If the estimators are not set, the one provided by the initialization is the default for the lowest resolution, which in turn passes the estimator to the next stage.

```
bool multi_resol = true;
rgrl_data_manager_sptr data = new rgrl_data_manager( multi_resol );

unsigned resolution = 1;                    //lower resolution
double dimension_increase = 2;
data->add_data( resolution,
                moving_feature_set_low_res,
                fixed_feature_set_low_res );
rgrl_estimator_sptr affine_model = new rgrl_est_affine();
data->add_estimator( resolution, affine_model);
data->set_dimension_increase_for_next_stage( resolution, dimension_increase);

resolution = 0;              //original resolution
data->add_data( resolution,
                moving_feature_set_high_res,
                fixed_feature_set_high_res );
rgrl_estimator_sptr quadratic_model = new rgrl_est_quadratic();
data->add_estimator( resolution, quadratic_model);

rgrl_feature_based_registration reg( data );
```

To run the registration engine, the starting resolution has to be specified. If an `rgrl_initializer` handles initialization, the initializer has to have the knowledge of the resolution level at which the registration begins. Again, the higher the number is for the resolution

level, the lower the image resolution. The highest resolution is always for 0^{th} stage[2].

Let's execute this example with the feature files `rgrl/examples/IMG0002.pgm.txt` and `rgrl/examples/IMG0003.pgm.txt`. The corresponding low resolution files are `rgrl/examples/IMG0002_low.pgm.txt` and `rgrl/examples/IMG0003_low.pgm.txt`, respectively. The first transformation below is the result at the lower resolution. The second transformation is the final estimate using a quadratic model.

```
xform: A =
0.999887 -0.00609213
0.00332669 0.998528
t = -10.1254 -0.63757


Final xform:
Q =
1.69245e-05 1.00902e-05 2.49065e-06
2.81146e-07 3.95073e-06 8.30134e-06
A = 0.979931 -0.018122
-0.00138591 0.989724
t = -13.0714 2.17319
Final alignment error = 0.430113
```

3.2 View-Based Registration

The code for view-based registration will be made open-source soon.

[2] In ITK, it is the opposite. The highest index represents the finest resolution.

BIBLIOGRAPHY

[1] J.R. Bergen, P. Anandan, K.J. Hanna, and R. Hingorani. Hierarchical model-based motion estimation. In *Proceedings of the Second European Conference on Computer Vision*, pages 237–252, 1992. 3.1

[2] P.J. Besl and N.D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992. 1, 2.3

[3] T.O. Binford and T.S. Levitt. Quasi-invariants: Theory and exploitation. In *Proceedings of the DARPA Image Undertanding Workshop*, pages 819–829, 1993. 2.4.2

[4] A. Can, C.V. Stewart, B. Roysam, and H.L. Tanenbaum. A feature-based, robust, hierarchical algorithm for registering pairs of images of the curved human retina. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(3):347–364, 2002. 2.4.1

[5] Y. Chen and G.G. Medioni. Object modeling by registration of multiple range images. *Image and Vision Computing*, 10(3):145–155, 1992. 2.3

[6] J. Feldmar and N. Ayache. Rigid, affine and locally affine registration of free-form surfaces. *International Journal of Computer Vision*, 18(2):99–119, 1996. 1, 2.3

[7] J. Feldmar, J. Declerck, G. Malandain, and N. Ayache. Extension of the ICP algorithm to nonrigid intensity-based registration of 3d volumes. *Computer Vision and Image Understanding*, 66(2):193–206, May 1997. 3.1

[8] Martin A. Fischler and Robert C. Bolles. Random Sample Consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24:381–395, 1981. 2.4.1

[9] E. Gamma, R. Helm, R Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995. 2.5

[10] Sébastien Granger and Xavier Pennec. Multi-scale EM-ICP: A fast and robust approach for surface registration. In *Proceedings of the Seventh European Conference on Computer Vision*, pages 418–432, 2002. 2.2.1

[11] Paul W. Holland and Roy E. Welsch. Robust regression using iteratively reweighted least-squares. *Commun. Statist.-Theor. Meth.*, A6:813–827, 1977. 2.3

[12] L. Ibáñez, W. Schroeder, L. Ng, and J. Cates. *The ITK Software Guide: The Insight Segmentation and Registration Toolkit (version 1.4)*. Kitware Inc., 2003. 1

[13] J. V. Miller and C. V. Stewart. MUSE: Robust surface fitting using unbiased scale estimates. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 300–306, 18–20 June 1996. 2.3

[14] J.L. Mundy and A. Zisserman, editors. *Geometric Invariance in Computer Vision*. MIT Press, 1992. 2.4.2

[15] S. Ourselin, A. Rocher, G. Subsol, X. Pennec, and N. Ayache. Reconstructing a 3D structure from serial histological sections. *Image and Vision Computing*, 19:25–31, 2000. 1

[16] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the ICP algorithm. In *Proceedings of the Third International Conference on 3-D Digital Imaging and Modeling*, pages 224–231, 2001. 2.2.1

[17] H.S. Sawhney and R. Kumar. True multi-image alignment and its application to mosaicing and lens distortion correction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(3):235–243, 1999. 3.1

[18] C. Schutz, T. Jost, and H. Hugli. Multi-feature matching algorithm for free-form 3d surface registration. In *International conference on pattern recognition (ICPR 98)*, pages 16–20, 1998. 2.3

[19] Charles V. Stewart. MINPRAN: A new robust estimator for computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:925–938, 1995. 2.4.1

[20] Charles V. Stewart. Robust parameter estimation in computer vision. *SIAM Reviews*, 41(3):513–537, 1999. 2.3

[21] Charles V. Stewart, Chai-Ling Tsai, and Amitha Perera. A view-based approach to registration: Theory and application to vascular image registration. In *Proceedings of the International Conference on Information Processing in Medical Imaging*, Ambleside, UK, 20–25July 2003. 2.1

[22] C.V. Stewart, C-L. Tsai, and B. Roysam. The dual-bootstrap iterative closest point algorithm with application to retinal image registration. *IEEE Transactions on Medical Imaging*, 22(11):1379–1394, 2003. 2.1, 2.4

[23] The TargetJr Consortium. *The VXL Book*. http://vxl.sourceforge.net/, 2000. 1

[24] Z.Y. Zhang. Iterative point matching for registration of free-form curves and surfaces. *International Journal of Computer Vision*, 13(2):119–152, 1994. 2.3