

Image Registration

Goal

Let's say that we have two images (e.g. images of a human brain).

We now want to find a transformation T that maps image A to image B.

For instance, let's assume that we have an annotated brain ATLAS and a set of new images which are not annotated. By finding the right transformation we can easily annotate the new images.

Note: An ATLAS is typically not constructed from a single patient only. Instead images of multiple patients of a certain age group get averaged.

Rigid Transformation

A rigid transform is a geometric transformation that **preserves distances**.

It includes **rotations**, **translations** and **reflections**.

$$x' = Rx + t$$

where

R ... is the rotation matrix

x ... is a point in R^2 or R^3

t ... is a translation vector

and R is an orthogonal transformation ($R^T = R^{-1}$).

Furthermore, we differentiate between **proper** and **improper** transformations.

A transformation is called **proper**, if the determinant of $\det(R) = +1$ which means that it rotates and translates only.

Improper transformations can reflect, translate and rotate. In such a case $\det(R) = -1$.

We can define a *proper* transformations as a concatenation of multiple rotations.

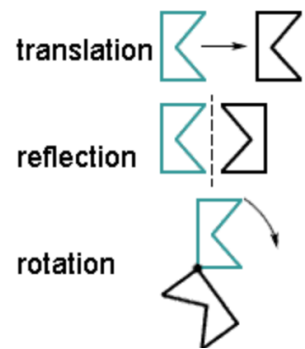
$$R = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \gamma & -\sin \gamma \\ 0 & \sin \gamma & \cos \gamma \end{pmatrix}$$

As we can see the rigid transform has 6 parameters. We have 3 degrees of freedom (α, β, γ) and 3 translation parameters.

Note: Rigid transforms which include translation are NOT linear transforms.

In **homogeneous coordinates** we can write a rigid transformation as **one matrix multiplication**.

Rigid + Scaling Transformation



The simplest non-rigid transformation is rigid + scaling. It's define as follows:

$$x' = RSx + t$$

with

$$S_x = \begin{pmatrix} r_x & 0 & 0 \\ 0 & r_y & 0 \\ 0 & 0 & r_z \end{pmatrix}$$

Obviously, distances are not longer preserved.

Caution: $x' = SRx + t$ produces a different result.

In case scaling is isotropic (equal in all dimensions), we speak of a **similarity transform**.

Affine Transformation

Scaling transformations are special cases of the more general class of **affine transformations**. Affine transformations are defined as follows:

$$x' = Ax + t$$

where there is no restriction on the elements of A. **Affine transformations** preserve

- straight lines (and hence, the planarity of surfaces)
- parallelism

There is no restriction on A. It can be anything.

Projective Transformation

The projective transformation can be written as:

$$x' = \frac{Ax+t}{\langle p, x \rangle + \alpha}.$$

Only straight lines and planar surfaces are preserved.

Image Registration

Algorithm:

Let's say we have two volumes (images) A and B. Volume A is called the *fixed image* since it does not move. Volume B is called the *moving image* since it's the one we apply the transform on.

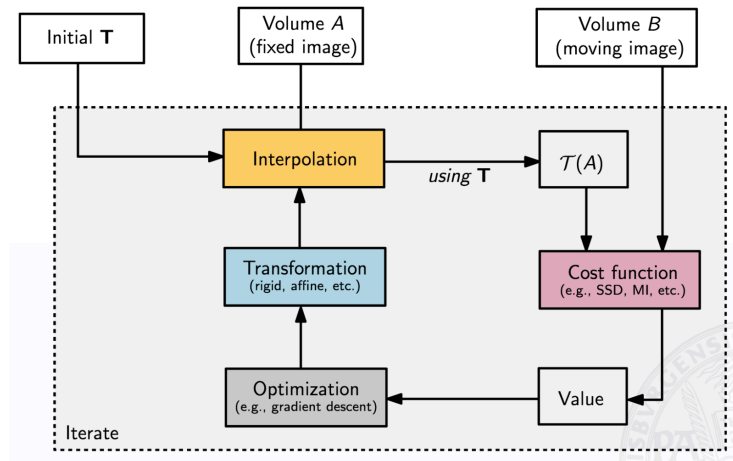
1. Take an initial transform and transform volume A (fixed image).
Perform interpolation if needed (in case we are not on real integer coordinates).
2. Take a cost function (e.g. sum-of-squared-differences) and compute the similarity between the transformed image and volume B.
3. Perform optimization. We take the value of the cost function and adjust the values of our transformation such that the new transformation will be closer to the moving image.

4. Keep iterating until convergence

Note: In real world we typically first do a rigid transformation and then, for example, an affine transform. After performing a rigid transform the images will overlap. Rigid transformations are easier to "find" since they have less parameters. Once this is done we perform further "fine tuning" by applying more advanced transformations.

When should we stop optimizing?

- Compare the transformations. For example, compute the distance between two transformations. If they are close, things are not changing anymore
- Look at the cost function. If the cost function does not change anymore (change is very small), we are done.



Optimization

The optimizer computes the gradient of the cost function and adapts the parameters in a way that we move towards the minimum.

Since our problem is usually not convex we might also get stuck in a local minimum.

So, the optimizer computes the following:

$$p_{new} = \operatorname{argmin} C(p)$$

where $C(p)$ is the cost function between the transformed moving image. p are the parameters of our transformation.

Then it updates the parameters by performing the following rule:

$$p^{n+1} = p^n + a_n d^n$$

where a_n is the step size and d^n is the search direction.

In case of gradient descent, the search direction is obviously the following:

$$d^n = -\frac{\partial C}{\partial p} p^n = -g^{n+1}.$$

Since we want to move in the direction of the negative gradient the update rule becomes:

$$p^{n+1} = p^n - a_n g^n$$

In some cases it might also make sense to add an additional generalization term to limit the degree of freedom when

choosing parameters.

This does obviously make sense if we have more advanced transformations (NOT for rigid or affine transformations) since the optimizer might come up with "really crazy" transformations.

By minimizing cost, we maximize similarity.

Nevertheless, the question that remains is when we should stop to optimize. How do we know if we have just reached a minimum?

Well, an indicator that tells us something about the minimum of a function is the 2nd-order derivative. It tells us how steep a function is at a certain point.

In general, we can say that the algorithm that are faster do only take the gradient, while the ones that are better in terms of their behaviour do also consider the 2nd-order derivative.

Note: Just think about a 1000 dimensional vector (parameters). It's gradient has 1000 values. However, the 2nd-order derivative is a matrix with 1000x1000 values.

| Method | Search direction | Type |
|--|---|-------|
| Gradient descent | $\mathbf{d}^{(n)} = -\mathbf{g}^{(n)}$ | — |
| Newton | $\mathbf{d}^{(n)} = -[\mathbf{H}^{(n)}]^{-1}\mathbf{g}^{(n)}$ | Smart |
| "quasi" Newton | $\mathbf{d}^{(n)} = -\mathbf{B}^{(n)}\mathbf{g}^{(n)}$ | Smart |
| Stochastic gradient descent | $\mathbf{d}^{(n)} \approx -\mathbf{g}^{(n)}$ | Cheap |
| $\mathbf{H}^{(n)}$... Hessian matrix, evaluated at $\mathbf{p}^{(n)}$ | | |
| $\mathbf{B}^{(n)}$... Approximation to the Hessian | | |

Derivation of Gaussian-Newton Rule

One loss function that is used commonly is the sum-of-squared-difference (SSD). Hence, the loss function can be written as follows:

$$J(p) = \sum_{i=1}^m r_i^2 = \sum_{i=1}^m (x_i - T(p, x_i))^2$$

The sum-of-squared-differences can be iteratively minimized using the Gauss-Newton algorithm as:

$$p^{n+1} = p^n - (A^T A)^{-1} A^T b$$

But how can we derive this formula? To understand the derivation we need to start by looking at the Newton algorithm.

$$p^{n+1} = p^n - H^{-1} \cdot g$$

As we can see to perform a Newton update step we basically need the Hessian and the gradients of the cost function. So, first, let's compute the gradient of our SSD loss function.

$$g_j = \frac{dJ(p)}{dp_j} = \frac{dJ(p)}{dp_j} \sum_{i=1}^m r_i^2 = 2 \cdot \sum_{i=1}^m r_i \cdot \frac{dr_i}{dp_j}$$

Next, we compute the Hessian H_{jk} (we need to take the derivative of g_j since the Hessian consists of 2nd-order derivatives).

$$H_{jk} = 2 \sum_{i=1}^m \left(\frac{dr_i}{dp_j} \cdot \frac{dr_i}{dp_k} + r_i \cdot \frac{d^2 r_i}{dp_j dp_k} \right)$$

Finally, we approximate the Hessian by ignoring the quadratic terms since they tend to be very small. Of course, we

would have to argue why the term is small. However, for sake of simplicity we now simply assume that this holds.

Hence, the Hessian becomes:

$$H_{jk} \approx 2 \sum_{i=1}^m \left(\frac{dr_i}{dp_j} \cdot \frac{dr_i}{dp_k} \right)$$

Additionally, we recognize that this comes in since we can easily rewrite it as:

$$H_{jk} \approx 2 \sum_{i=1}^m \left(\frac{dr_i}{dp_j} \cdot \frac{dr_i}{dp_k} \right) = 2 \sum_{i=1}^m J_{ij} \cdot J_{ik} = 2 \cdot J_r^T \cdot J_r$$

So, let's summarize what we have:

$$J_{ij} = \begin{pmatrix} \frac{dr_1}{dp_1} & \frac{dr_1}{dp_2} & \cdots & \frac{dr_1}{dp_j} \\ \frac{dr_2}{dp_1} & \frac{dr_2}{dp_2} & \cdots & \frac{dr_2}{dp_j} \\ \cdots & & & \\ \frac{dr_m}{dp_1} & \frac{dr_m}{dp_2} & \cdots & \frac{dr_m}{dp_j} \end{pmatrix}$$

$$J_{ij}^T = \begin{pmatrix} \frac{dr_1}{dp_1} & \frac{dr_2}{dp_1} & \cdots & \frac{dr_m}{dp_1} \\ \frac{dr_1}{dp_2} & \frac{dr_2}{dp_2} & \cdots & \frac{dr_m}{dp_2} \\ \cdots & & & \\ \frac{dr_1}{dp_j} & \frac{dr_2}{dp_j} & \cdots & \frac{dr_m}{dp_j} \end{pmatrix}$$

$$J_{ik} = \begin{pmatrix} \frac{dr_1}{dp_1} & \frac{dr_1}{dp_2} & \cdots & \frac{dr_1}{dp_k} \\ \frac{dr_2}{dp_1} & \frac{dr_2}{dp_2} & \cdots & \frac{dr_2}{dp_k} \\ \cdots & & & \\ \frac{dr_m}{dp_1} & \frac{dr_m}{dp_2} & \cdots & \frac{dr_m}{dp_k} \end{pmatrix}$$

$$H_{jk} = 2 \cdot J_{ij}^T \cdot J_{ik} = 2 \cdot \begin{pmatrix} \frac{dr_1}{dp_1} \frac{dr_1}{dp_1} + \frac{dr_2}{dp_1} \frac{dr_2}{dp_1} + \frac{dr_3}{dp_1} \frac{dr_3}{dp_1} + \cdots + \frac{dr_m}{dp_1} \frac{dr_m}{dp_1} & \frac{dr_1}{dp_1} \frac{dr_1}{dp_2} + \frac{dr_2}{dp_1} \frac{dr_2}{dp_2} + \frac{dr_3}{dp_1} \frac{dr_3}{dp_2} + \cdots + \frac{dr_m}{dp_1} \frac{dr_m}{dp_2} & \cdots \\ \frac{dr_1}{dp_2} \frac{dr_1}{dp_1} + \frac{dr_2}{dp_2} \frac{dr_2}{dp_1} + \frac{dr_3}{dp_2} \frac{dr_3}{dp_1} + \cdots + \frac{dr_m}{dp_2} \frac{dr_m}{dp_1} & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ \frac{dr_1}{dp_j} \frac{dr_1}{dp_1} + \frac{dr_2}{dp_j} \frac{dr_2}{dp_1} + \frac{dr_3}{dp_j} \frac{dr_3}{dp_1} + \cdots + \frac{dr_m}{dp_j} \frac{dr_m}{dp_1} & \frac{dr_1}{dp_j} \frac{dr_1}{dp_2} + \frac{dr_2}{dp_j} \frac{dr_2}{dp_2} + \frac{dr_3}{dp_j} \frac{dr_3}{dp_2} + \cdots + \frac{dr_m}{dp_j} \frac{dr_m}{dp_2} & \cdots \end{pmatrix}$$

$$J_j = \begin{pmatrix} \frac{dr_1}{dp_1} & \frac{dr_1}{dp_2} & \cdots & \frac{dr_1}{dp_j} \\ \frac{dr_2}{dp_1} & \frac{dr_2}{dp_2} & \cdots & \frac{dr_2}{dp_j} \\ \cdots & & & \\ \frac{dr_m}{dp_1} & \frac{dr_m}{dp_2} & \cdots & \frac{dr_m}{dp_j} \end{pmatrix}$$

$$r_i = \begin{pmatrix} r_1 \\ r_2 \\ \dots \\ r_m \end{pmatrix}$$

$$J_j^T = \begin{pmatrix} \frac{dr_1}{dp_1} & \frac{dr_2}{dp_1} & \dots & \frac{dr_m}{dp_1} \\ \frac{dr_1}{dp_2} & \frac{dr_2}{dp_2} & \dots & \frac{dr_m}{dp_2} \\ \dots & \dots & \dots & \dots \\ \frac{dr_1}{dp_j} & \frac{dr_2}{dp_j} & \dots & \frac{dr_m}{dp_j} \end{pmatrix}$$

$$g_j = 2 \cdot J_j^T \cdot r_i = 2 \cdot \begin{pmatrix} r_1 \frac{dr_1}{dp_1} + r_2 \frac{dr_2}{dp_1} + \dots + r_m \frac{dr_m}{dp_1} \\ r_1 \frac{dr_1}{dp_2} + r_2 \frac{dr_2}{dp_2} + \dots + r_m \frac{dr_m}{dp_2} \\ \dots \\ r_m \frac{dr_1}{dp_j} + r_2 \frac{dr_2}{dp_j} + \dots + r_m \frac{dr_m}{dp_j} \end{pmatrix}$$

Therefore, we can now write:

$$H_{jk}^{-1} \cdot g = (2 \cdot J_{ij}^T \cdot J_{ik})^{-1} \cdot 2 \cdot J_j^T \cdot r_i.$$

Since J_{ij} is the same as J_j we finally get for the Gauss-Newton update rule:

$$p^n + 1 = p^n - (J^T \cdot J)^{-1} \cdot J^T \cdot r_i$$

Cost functions

In this section we are going to compare different cost functions.

Sum of squared differences (SSD)

$$SSD(A, B') = \frac{1}{N} \sum_i (A(i) - B'(i))^2 \quad \forall A \cap B'$$

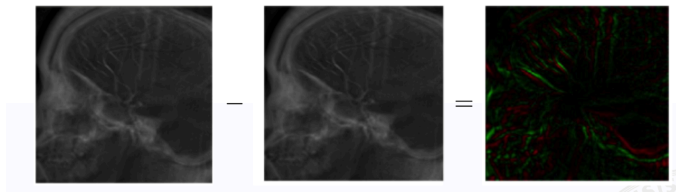
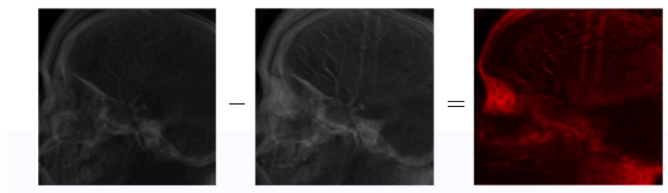
Sum of absolute differences (SAD)

$$SAD(A, B') = \frac{1}{N} \sum_i |A(i) - B'(i)| \quad \forall A \cap B'$$

SSD and SAD are probably the most common cost functions. However, there's one major problem. They only work if there is an initial overlap between A and B'. One way to handle such cases is to compute the center of both brains and move the images on the top of each other.

Furthermore, both images need to have the same modality. We can't compare, for instance, T1 and T2 images since the color range is completely different.

The following images illustrate the problem.

| Disalignment | Contrast Change |
|---|--|
| <p>Good example for using SSD: same image, only misalignment</p>  | <p>Bad example for using SSD: same image, but contrast change!</p>  |

Correlation coefficient

Another strategy that typically performs slightly better when it comes to different image modelity is to compute a correlation coefficient CC .

$$CC(A, B') = \frac{\sum_i [A(i) - \bar{A}][B'(i) - \bar{B}']}{\sqrt{\sum_i [A(i) - \bar{A}]^2 \sum_i [B'(i) - \bar{B}']^2}} \quad \forall A \cap B'$$

As we can see, if both images are perfectly aligned, CC should be close to 1.

Information-theoretic approaches

Joint histogram and joint entropy

Other techniques that had a huge impact in the last couple of years are information-theoretic approaches.

For instance, we might compute the **joint histogram** of two images.

Why does it work?

The histogram can be seen as discrete version of the probability density. So, if the images are well-aligned, then the joint histogram starts to **form clusters**.

Let's say we have two images with intensity ranges normalized to [0,1].

$$J(A, B')[i, j] = \sum_{x \in A} \delta(x)$$

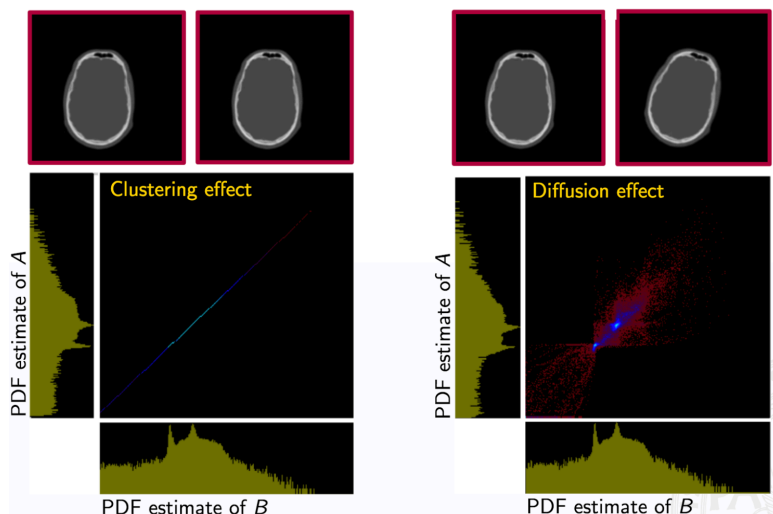
with

$$\delta(x) = \begin{cases} 1, & \text{if } A(x) \in [\frac{i}{B_A}, \frac{i+1}{B_A}] \text{ and } B(x) \in [\frac{j}{B_{B'}}, \frac{j+1}{B_{B'}}] \\ 0, & \text{otherwise} \end{cases}$$

Note: B_A and B_B are bins of the (marginal) histogram A and B'

As mentioned before, the $J(A, B')$ can be used as an estimator for the joint probability density (PDF) of A and B' and as an estimate for the **joint entropy**.

$$JE(A, B') = - \sum_{i,j} J(A, B')[i, j] \cdot \log J(A, B')[i, j]$$



Note: The JE is invariant to scaling and different intensity ranges. Also it's important to note that the PDF is only defined on the **region of overlap**.

Mutal information

Since the PDF is only defined on the region of overlap, a better strategy is to look at the **mutal information (MI)**.

$MI(A, B') = H(A) + H(B') - JE(A, B')$ where $H(A)$ and $H(B')$ are the marginal histograms of A and B'

We don't run into issue anymore since we only consider the histogram and the joint entropy. So, we don't depend on the overlap anymore. However, there still need to overlap slightly. Registration won't work if they are totally off.

Finally, we try to **maximize the mutal information**.

Note: MI is the de-facto standard in image registration today.

Interpolation

We basically differentiate between two ways of interpolation:

- Pushing
- Pulling

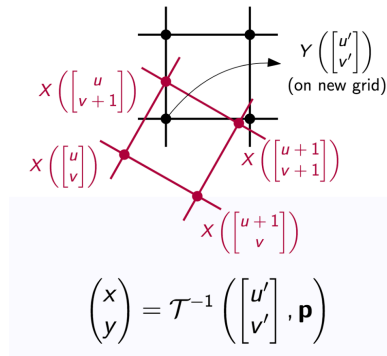
But what is the intuition behind it?

Let's say that we have a transform T that maps $T(x, y)$ to $T(x', y')$.

Depending on the transformation some points may get squeezed together (get very close to each other) in the transformed image and we might end up with "holes". One way to overcome this problem would be to interpolate by it's closest values, but this would require nearest-neighbour interpolation which is computationally intensive. Also, if we have many points in a region, the interpolation would be dominated by these points. This principle is called **pushing** and is usually not done in practice due to the hole problem.

The better we to deal with it would be to start with the target image and **apply the inverse transform**. This does still require interpolation since it's unlikely that the transformation maps to exactly a point on the discrete grid. This approach is called **pulling**.

Note: Finding the inverse is not always easy. For more complex transform we might have to solve it numerically.



One really simple way is to do nearest-neighbour interpolation:

$$Y(u', v') = X(\text{round}(x), \text{round}(y))$$

Other choices are bilinear or polynomial interpolation.

Physical vs. voxel space

Usually, before we do the actual registration we first convert the image to physical space, perform the registration and finally map it back to voxel space. This needs to be done to handle images with different voxel spacings.