Retrospective Correction Overview

Speaker Name: Jesper Andersson
Email: jesper.andersson@ndcn.ox.ac.uk
Organization/employer: University of Oxford
Address:
FMRIB Centre
University of Oxford
John Radcliffe Hospital
Oxford OX3 9DU
UK

This talk will give an overview of the principles behind retrospective volumetric motion correction.

To "register" two images involves a small set of tasks/tools.
First of these is the ability to transform an image, i.e. "move" it, given that one knows "by how much" it shall be moved. So for example if we know that the subject moved 4.5mm to the left we need to be able to move it back to the right so the we obtain the image that would have been acquired had the subject not moved.
The second task is to be able to find out how the subject moved as that is often not known. This is typically done by testing many different transforms and evaluating them to find which transform is most "successful". The evaluation is done by calculating a similarity measure (often referred to as a cost-function) between the transformed image and some image that represents the "reference" location, for example the first image in an EPI time-series.
The third task is related to the second in that it concerns the most efficient way of "testing many different" transforms. To naively test transforms at random would take a very long time. Efficient algorithms need to combine an ability to chose a good "next transform to try" with a reasonable computational cost for calculating what the next step should be.

TRANSFORMING IMAGES:
When transforming an image one typically starts by transforming the co-ordinates from the grid of the reference image to the grid of the image one wants to transform (which we may call the *T* image). If for example the subject has moved 3 pixels to the right one would transform the coordinate $\mathbf{x}=[x\ y\ z]^T$ as $\mathbf{x}' = \mathbf{x} + [3\ 0\ 0]^T$, sample the *T* image at $\mathbf{x}'$ and write that value at $\mathbf{x}$ in the transformed image.

Transformation Matrix:
It is practical to code the co-ordinate in a Matrix (let's call it **A**) so that any (rigid body) transform can be described by $\mathbf{x}' = \mathbf{A}\mathbf{x}$. In order to do that one needs to use the "one extended" formulation so that $\mathbf{x}=[x\ y\ z\ 1]^T$, **A** is a 4x4 matrix and $\mathbf{x}'=[x'\ y'\ z'\ 1]^T$. The matrix **A** can in turn be calculated as a function of the rigid body parameters **p**, three rotations and three translations, so that $\mathbf{A} = A(\mathbf{p})$. One little problem that one needs to be aware of is that there are several ways of implementing *A*, which means that one software package may give you one **A** for a given **p** whereas another package might give you a different **A** for the same **p**.

Interpolation:
Typically the transformed co-ordinate will not fall on an integer co-ordinate in the image one wishes to transform, but rather somewhere between a few voxel centres. In three dimensions the immediate surroundings of any non-integer co-ordinate will be a cube where the eight corners consists of original voxel centres. It would seem intuitive to think that our inferred (interpolated) value should be some combination of those corners, and the simplest form of interpolation (tri-linear) is precisely that.
Slightly less intuitively a more "accurate" interpolated value can be obtained by considering also voxel values further away, outside the immediate cube. Such interpolation schemes comes in different flavours, but all have the unfortunate tendency to become computationally quite slow. One

scheme that manages to combine high fidelity with a reasonable computational burden is spline interpolation when implemented as an Infinite Impulse Response (IIR) filter.

COST-FUNCTION:
A cost-function (or objective-function) is a function whose value indicates how well two images are aligned. If we call the images R (for reference) and T (the image we want to transform) we can write the function as $f(\mathbf{p}; R,T)$ which indicates that it is a function of the rigid body parameters $\mathbf{p}$ given the images R and T. One can also be a little less formal and just write $f(\mathbf{p})$, where the dependence on R and T is implicit.
In more "general" image registration one can consider a range of different cost-functions depending on how "similar" one expects the images to be in the first place (consider for example aligning a T1-weighted image to a T2-image). For motion correction the task is typically a little easier in that images are expected to be quite similar and to differ only minimally (the intensity can for example increase slightly in a small activated region compared to the reference image). Popular cost-functions include sum-of-squared differences (SSD) which is predicated on an assumption that the images are identical except for a normal distributed random error. Another often used cost-function is correlation-ratio which makes slightly fewer assumptions than SSD.

SEARCH ALGORITHM:
The aim of a search/optimisation algorithm is to find the "best" value of $f(\mathbf{p})$ (which would mean the smallest value in the case of SSD, or the biggest in the case of correlation-ratio). The actual value of $f(\mathbf{p})$ at that point is incidental, and what really matters is the point (in the space of rigid body parameters) $\mathbf{p}^*$ at which that value occurs. The procedure is to start at some point $\mathbf{p}_0$ and from there take a number of steps, each step taking us closer to the point $\mathbf{p}^*$, until we are there (or we can tell that we are sufficiently close). For each step that is taken the algorithm will spend some time making calculations to determine what would be a good step to take nest.
An ideal algorithm will take as few steps as possible, for each step spending a minimal time on calculating where to go next. But, alas, there is no such thing. Algorithms tend to take a few expensive steps, or many inexpensive ones.
An algorithm in the latter category is the Nelder-Mead downhill simplex method that uses a simplex with one vertex more than there are parameters (i.e. 7 in our case). Each vertex has a location in the $\mathbf{p}$-space as well as a known value for $f(\mathbf{p})$. Using clever heuristics it will replace the vertex with the "worst" $f(\mathbf{p})$ for a better one at each step. Due to its simplicity it is a popular choice. More so than it deserves some might say.
On the other end of the spectrum is the Gauss-Newton method. It is based on a series of linearisations of the problem where at each step the "unknown" function $f(\mathbf{p})$ is replaced by a "known" function base on a Taylor-expansion of $f(\mathbf{p})$. This "proxy-function", which we may call $h(\mathbf{p})$ is a second order multivariate polynomial, and just as with a univariate polynomial it has an analytical solution. Hence, in a single step one can go from $\mathbf{p}$ to $\mathbf{p}'$, where $\mathbf{p}'$ is the point with the "best" $h(\mathbf{p})$. Since $h(\mathbf{p})$ is just a proxy $\mathbf{p}'$ will not necessarily be $\mathbf{p}^*$ but it will typically be a very good step towards it. Unfortunately each step will be rather costly since $h(\mathbf{p})$ will contain the first and second derivatives of $f(\mathbf{p})$ and these takes a little effort to calculate.