Non-linear registration
aka
Spatial normalisation

FMRIB Technial Report TR07JA2

Jesper L. R. Andersson, Mark Jenkinson and Stephen Smith

FMRIB Centre, Oxford, United Kingdom

Correspondence and reprint requests should be sent to:
Jesper Andersson
FMRIB Centre
JR Hospital
Headington
Oxford OX3 9DU
phone: 44 1865 222 782
fax: 44 1865 222 717
mail: jesper@fmrib.ox.ac.uk

28 June 2007

1 ABSTRACT

This document describes the principles behind and the implementation of *fnirt*, the FMRIB tool for small-displacement non-linear registration. The displacement fields are modelled as linear combinations of basis-functions, which may be the discrete Cosine transform (DCT) or cubic B-splines placed on a regular grid. Regularisation of the field is based on membrane energy. At present the registration is based on a weighted sum of scaled sum-of-squared differences and membrane energy. Great effort has been placed on the optimisation, and on providing computational tools to enable robust and rapid convergence even for relatively high resolution of the warps. The results that are presented are based on a multi-scale Levenberg-Marquardt minimisation. The registration is initialised and run to convergence with sub-sampled images, a field of low resolution and a high regularisation weight. The images and the fields from the first step are then up-sampled, the regularisation modified and it is again run to convergence. This is repeated until the required warp-resolution and level of regularisation is achieved. The method has been tested with promising results on T1-weighted structural images and on FA-images from DTI acqusitions.

2 INTRODUCTION

Registering scans of brains from different subjects is a neccesary processing step for many types of analyses such as *e.g.* multi-subject fMRI studies, inter-group comparisons of tissue composition (*e.g.* VBM) or measures derived from diffusion weighted MR (*e.g.* TBSS). There is a family of algorithms that attempt to do this in the native "brain-space" using different sets of transforms between subjects, or more typically between a subject and some template instantiating some standard space. In order of allowing more "non-local" warps the transforms can be divided into linear (affine) transforms, small-deformation non-linear transforms and large-deformation non-linear transforms (often referred to as "viscous fluid registration"). The principal difference between small-deformation and large-deformation method is that in the former case the warps are fully described by three fields of displacements (one for each dimension). In the fluid-type algorithms one would in principle need the history of each location as it is being warped towards it target registration, though in practice this is often achieved through one or more regriddings within the registration. An intuitive example of the difference between the approaches is the matching of two 2D pictures of human faces. For small-deformation methods it would *e.g.* not be possible to map the left eye of subject one to the right eye of subject two and vice versa. This is in contrast to large-deformation methods where it would in principle be possible to start by displacing the right eye upwards and the left downwards, move each eye towards the opposite side and then move them up/down into place.

The fluid-registration algoritms are often considered the most "advanced" and are typically formulated in terms of a set of partial differential equations, and are often solved using corresponding methods such as Gauss-Seidel or multi-grid methods. As outlined above they offer great freedom in matching one image to another and can achieve an almost arbitrarily good matching of intensities between the images. However, the information available in "traditional" structural images such as T1- or T2-eighted images is largely limited to tissue type. This means

that prety much any mapping between two images that map gray matter onto gray matter, white matter onto white matter and CSF onto CSF is equally good in terms of the cost-function and the task of finding the "true" warps can be thought of as finding the most likely set of warps of those that yield that best cost-function value. It can therefore be argued that warps from a small-deformation model that yields a similar or identical cost-function value as those from a large-deformation model represents a more "reasonable" solution to the registration problem. Furthermore, it is clear that there exist no one-to-one mapping as defined by gyri and sulci between different brains. There is a set of sulci that are consistently found across "normal" subjects such as *e.g.* the interhemispheric fissure, the Sylvian fissure, the parietal-occipital fissure and the central sulcus. In contrast there appear to be large inter-subject differences in *e.g.* parietal cortex where it has been reported that not even the number of sulci is consistent across "normal" subjects. Even though it would seem reasonable to assume that there exist at least a functional one-to-one mapping across subjects there is insufficient experimental data to conclude even that. Indeed there exists studies which indicate that even low-level tasks such as *e.g.* odd-ball detection is processed differently in different subjects.

In this work we have therefore opted for a small-deformation model for the warps, and put our efforts into ensuring convergence to a plausible field. We have aimed at achieveing that through a combination of a pyramid, or multi-resolution, scheme along with an afficient optimisation algorithm for each step of the pyramid.

3 THEORY

3.1 Transforms and some definitions

The general transform of coordinates for nonlinear registration is of the form

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \mathbf{M} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} + \begin{bmatrix} d_x(x,y,z) \\ d_y(x,y,z) \\ d_z(x,y,z) \\ 1 \end{bmatrix} \tag{1}$$

where $\mathbf{M}$ is an affine transform matrix that will account for differences in voxel-size, position, etc. For the reminder of this paper we will assume that $\mathbf{M}$ is known. The entities $d_i(x,y,z)$ are knowns as displacement fields, and will for each location $[x\ y\ z]$ tell how far the sampling point should be displaced in the i direction.

We will further assume the existence of some interpolating function such that given samples of some function $g(i,j,k)$ where i, j and k are sampled on some regular grid we can infer values $g(x,y,z)$ as long as x, y and z fall within the original grid. We will then use $g(x',y',z')$ to denote the values of g for a set of coordinates given by 1. This value can be thought of as the value of g at some point $[x\ y\ z]$ under the transform given by $\mathbf{M}$ and the displacement fields $d_i(x,y,z)$.

In this paper we will use basis-functions to model the fields $d_i(x,y,z)$ as a function of some set of parameters $\mathbf{w}$. We can for example model the field as a linear combination of 3D cubic B-splines. Each spline in the 3D set is associated with a spline coordinate lmn specifying the position of the spline in the space given by xyz. We will use $B_{lmn}(x,y,z)$ to denote the value of the spline with coordinate lmn at the location $[x\ y\ z]$. Each of these splines are then multipled

by a coefficient c_{lmn}, specifying how much is needed of that particular spline (see fig. ?). We now vectorise the coefficients c_{lmn} and denote the resulting vector by $\mathbf{w}_i$ where i denotes the direction of the field such that *e.g.* $\mathbf{w}_x$ denotes the vector containing the coefficients defining the $d_x(x,y,z)$ displacement field. Furthermore we denote the concatenation of these vectors by $\mathbf{w}$, *i.e* $\mathbf{w} = [\mathbf{w}^{(x)T}\ \mathbf{w}^{(y)T}\ \mathbf{w}^{(z)T}]^T$. With this notation we can the write $g_{xyz}(\mathbf{w})$ to denote the value of g at $[x'\ y'\ z']$ under the transform given by $\mathbf{w}$ implicit on the (constant) $\mathbf{M}$.

The general notation $g_{xyz}(\mathbf{w})$ is independent of our choice of basis-function and would be equally valid for *e.g* a discrete cosine basis set.

We will also make use of the gradient, or rather its components, of g at some point $[x'\ y'\ z']$, which we define as

$$\nabla g_{xyz}(\mathbf{w}) = \left[\left.\frac{\partial g_{xyz}}{\partial x}\right|_{\mathbf{w}} \quad \left.\frac{\partial g_{xyz}}{\partial z}\right|_{\mathbf{w}} \quad \left.\frac{\partial g_{xyz}}{\partial z}\right|_{\mathbf{w}} \right] \tag{2}$$

where the partial derivative with respect to *e.g* x, $\partial g_{xyz}/\partial x|_{\mathbf{w}}$, denotes the rate of change of g at $[x'\ y'\ z']$ as one translates the sampling point in the x-direction. It should be noted that the exact form of the partial derivatives $\partial g/\partial x$ is conditional on the interpolating function alluded to above. In the present paper we will ignore this issue and assume the existence of functions/kernels to perform the interpolation and calculating the corresponding partial derivatives. If we now assume that the parameter $w_i^{(x)}$ is the coefficient for the lmnth spline in the x-displacement field d_x we can denote the derivative of g as

$$\left.\frac{\partial g_{xyz}}{\partial w_i^{(x)}}\right|_{\mathbf{w}} = \left.\frac{\partial g_{xyz}}{\partial x}\right|_{\mathbf{w}} B_{lmn}(x,y,z) \tag{3}$$

where the $\mathbf{w}$ subscript indicates that is has been calculated at a point $\mathbf{w}$ in the parameter space.

Let us further define the vector

$$\mathbf{g}(\mathbf{w}) = \begin{bmatrix} g_{111}(\mathbf{w}) \\ g_{211}(\mathbf{w}) \\ g_{X11}(\mathbf{w}) \\ \vdots \\ g_{121}(\mathbf{w}) \\ g_{221}(\mathbf{w}) \\ \vdots \\ g_{X21}(\mathbf{w}) \\ \vdots \\ g_{XY1}(\mathbf{w}) \\ \vdots \\ g_{XYZ}(\mathbf{w}) \end{bmatrix} \tag{4}$$

where x, y and z are defined on the ranges 1–X, 1–Y and 1–Z respectively. In an equivalent

manner we define the vectors

$$\left.\frac{\partial \mathbf{g}}{\partial x}\right|_{\mathbf{w}} = \begin{bmatrix} \left.\frac{\partial g_{111}}{\partial x}\right|_{\mathbf{w}} \\ \left.\frac{\partial g_{211}}{\partial x}\right|_{\mathbf{w}} \\ \vdots \\ \left.\frac{\partial g_{XYZ}}{\partial x}\right|_{\mathbf{w}} \end{bmatrix} \tag{5}$$

and

$$\mathbf{B}_{lmn} = \begin{bmatrix} B_{lmn}(1,1,1) \\ B_{lmn}(2,1,1) \\ \vdots \\ B_{lmn}(X,Y,Z) \end{bmatrix} \tag{6}$$

Noting that there is a direct mapping from the index i into $\mathbf{w}$ to the triplet lmn we can equally well, and more succinctly, write $\mathbf{B}_i$ as $\mathbf{B}_{lmn}$ for the vector representation of the ith basis function. Combining equations 3, 5 and 6 we can further define the matrix

$$\mathbf{J}_x(\mathbf{w}) = \begin{bmatrix} \left.\frac{\partial \mathbf{g}}{\partial x}\right|_{\mathbf{w}} \odot \mathbf{B}_1 & \left.\frac{\partial \mathbf{g}}{\partial x}\right|_{\mathbf{w}} \odot \mathbf{B}_2 & \dots & \left.\frac{\partial \mathbf{g}}{\partial x}\right|_{\mathbf{w}} \odot \mathbf{B}_{LMN} \end{bmatrix} \tag{7}$$

where $\odot$ denotes elementwise (or Hadamard) product. Hopefully it should be clear that each element of $\mathbf{J}_x(\mathbf{w})$ is of the form given by equation 3. Finally by thinking of $\mathbf{g}(\mathbf{w})$ as an $\Re^{3LMN} \rightarrow \Re^{XYZ}$ mapping where L, M and N are the number of basis-functions and X, Y and Z are the number of samples/voxels in the x-, y- and z-directions respectively we can define the Jacobian matrix $\mathbf{J}$ of that mapping as

$$\mathbf{J}(\mathbf{w}) = \underbrace{\begin{bmatrix} \mathbf{J}_x(\mathbf{w}) & \mathbf{J}_y(\mathbf{w}) & \mathbf{J}_z(\mathbf{w}) \end{bmatrix}}_{XYZ \times 3LMN} \tag{8}$$

3.2 Sum of squared differences cost-function

Our task is to find the parameters $\mathbf{w}$ describing the fields $d_i(x,y,z)$ that transforms our function/image g from its native space to some other arbitrary reference space. In this paper we consider intensity-based methods for finding these. This is performed by defining some cost/objective-function in terms of some template $\mathbf{f}$ that defines our reference space and our object image $\mathbf{g}(\mathbf{w})$. Let us call this function O, and let us write it as $O(\mathbf{w})$ as a short for $O(\mathbf{f}, \mathbf{g}(\mathbf{w}))$. An example of such a function is the "mean sum of squared differences" cost-function which we can write as

$$O(\mathbf{w}) = \frac{1}{XYZ} \sum_{z=1}^{Z} \sum_{y=1}^{Y} \sum_{x=1}^{X} (g_{xyz}(\mathbf{w}) - f_{xyz})^2 \tag{9}$$

or equivalently using the notation we defined in equation 4

$$O(\mathbf{w}) = \frac{1}{XYZ} (\mathbf{g}(\mathbf{w}) - \mathbf{f})^T (\mathbf{g}(\mathbf{w}) - \mathbf{f}) \tag{10}$$

In this case O is indeed a cost-function, *i.e* the smaller it is the happier we are. The general strategy of finding an estimate $\hat{\mathbf{w}}$ is then to find

$$\min_{\text{arg } \mathbf{w}} O(\mathbf{w}) \tag{11}$$

3.3 What do we need for minimising the cost-function

Apart from time and patience, that is. Methods for minimsation of functions that are non-linear in the parameters of interest come in various flavors. Some rely solely on the ability to calculate the costfuntion O at any point $\mathbf{w}$ in the parameter space. These methods typically need to calculate O at a *large* number of points are are therefore not practical for problems where $\mathbf{w}$ consists of a large number of parameters and/or when the calculation of O is costly. Methods that require the calculation also of the gradient, and possibly also the Hessian, of O are therefore preferable. So, we need to be able to also calculate these entities. The gradient of O is defined as

$$\nabla O(\mathbf{w}) = \left[\begin{array}{ccc} \frac{\partial O}{\partial w_1}\Big|_{\mathbf{w}} & \frac{\partial O}{\partial w_2}\Big|_{\mathbf{w}} & \cdots \frac{\partial O}{\partial w_{3LMN}}\Big|_{\mathbf{w}} \end{array} \right]^T \tag{12}$$

where the subscript $\mathbf{w}$ indicates that the derivative has been calculated at that point in the parameter space. This is the transpose of how it is mostly defined in the literature, but it will save us from writing some ^{T}s in the remainder of the paper. From the definition of O in equation 9 we see that an element $\partial O/\partial w_j$ of ∇O can be written as

$$\frac{\partial O}{\partial w_i}\Big|_{\mathbf{w}} = \frac{2}{XYZ} \sum_{z=1}^{Z} \sum_{y=1}^{Y} \sum_{x=1}^{X} (g_{xyz}(\mathbf{w}) - f_{xyz}) \left. \frac{\partial g_{xyz}}{\partial w_i} \right|_{\mathbf{w}} \tag{13}$$

If we define a vector $\mathbf{e}$ as

$$\mathbf{e}(\mathbf{w}) = \left[\begin{array}{c} g_{111}(\mathbf{w}) - f_{111} \\ g_{211}(\mathbf{w}) - f_{211} \\ \vdots \\ g_{XYZ}(\mathbf{w}) - f_{XYZ} \end{array} \right] \tag{14}$$

equivalently to equations 4–6, and using the definition in equation 8 we see that the gradient of O can be written as

$$\nabla O(\mathbf{w}) = \frac{2}{XYZ} \mathbf{J}^T(\mathbf{w}) \mathbf{e}(\mathbf{w}) \tag{15}$$

The next entity that we are intersted in is the Hessian of O, *i.e.* the matrix whose ijth element is

$$H_{ij}(\mathbf{w}) = \left. \frac{\partial^2 O}{\partial w_i \partial w_j} \right|_{\mathbf{w}} \tag{16}$$

Again from the definition of O given in equation 9 we see that such an element is of the form

$$\left. \frac{\partial^2 O}{\partial w_i \partial w_j} \right|_{\mathbf{w}} = \frac{2}{XYZ} \sum_{z=1}^{Z} \sum_{y=1}^{Y} \sum_{x=1}^{X} \left. \frac{\partial g_{xyz}}{\partial w_i} \right|_{\mathbf{w}} \left. \frac{\partial g_{xyz}}{\partial w_j} \right|_{\mathbf{w}} + \frac{2}{XYZ} \sum_{z=1}^{Z} \sum_{y=1}^{Y} \sum_{x=1}^{X} (g_{xyz}(\mathbf{w}) - f_{xyz}) \left. \frac{\partial^2 g_{xyz}}{\partial w_i \partial w_j} \right|_{\mathbf{w}} \tag{17}$$

Using equations 8 and 17 we see that we can write the Hessian matrix $\mathbf{H}$ as

$$\mathbf{H}(\mathbf{w}) = \frac{2}{XYZ}\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) + \text{term with weird second derivatives} \tag{18}$$

The first term in equation 18 is known as the Gauss-Newton approximation to the Hessian. In our technical report on optimisation we explain why this works, and often works better than if using the exact Hessian. For the remainder of this paper we will work with an approximation of the hessian given by

$$\mathbf{H}(\mathbf{w}) \approx \frac{2}{XYZ}\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) \tag{19}$$

3.3.1 Relation to my code

In the implementation of fnirt there is a virtual base class called `basisfield` from which two classes, `splinefield` and `dctfield`, have been derived. An object of such a class contains information about the field it implements. So if we were to *e.g* to declare a field of type splinefield we would do it like

```
std::vector<int>  field_size(3), knot_spacing(3);
field_size[0] = 128; field_size[1] = 128; field_size[2] = 96;
knot_spacing[0] = 8; knot_spacing[1] = 8; knot_spacing[1] = 8;
BASISFIELD::splinefield   xfield(field_size,knot_spacing);
```

Thus we have an object called `xfield`, corresponding to the entity $d_x(x, y, z)$ in the text above, implementing a cubic B-spline field of size $128 \times 128 \times 96$, corresponding to $X \times Y \times Z$ in the equations above, with a knot-spacing of 8 voxels. The latter means that the splines are placed on a regular grid with a distance of 8 voxels between adjacent spline kernels. These are placed such that the centre of spline # 2 coincides with the centre of the first voxel and then placed at regular intervals of 8 voxels until the first kernel whose whole support falls outside the field. From this are given the entities L, M and N in the equations above, and we can enquire about these of our field as

```
int L = xfield.CoefSz_x();
int M = xfield.CoefSz_y();
int N = xfield.CoefSz_z();
```

Let us now say we have vectors `f`, `g`, and `dgdx` containing the reference image f, the object image g and the partial derivative w.r.t. x, $\partial\mathbf{g}/\partial x$, both sampled at some point $\mathbf{w}$ in reference space. And let us say the we now want to calculate the gradient and Hessian of O. We can then note that

$$\nabla O(\mathbf{w}) = \frac{2}{XYZ}\mathbf{J}^T(\mathbf{w})\mathbf{e}(\mathbf{w}) = \frac{2}{XYZ}\begin{bmatrix} \mathbf{J}_x^T(\mathbf{w})\mathbf{e}(\mathbf{w}) \\ \mathbf{J}_y^T(\mathbf{w})\mathbf{e}(\mathbf{w}) \\ \mathbf{J}_z^T(\mathbf{w})\mathbf{e}(\mathbf{w}) \end{bmatrix} \tag{20}$$

Thus we can calculate the upper third of ∇O from

```
NEWMAT::ColumnVector  f(xfield.FieldSz()), g(xfield.FieldSz()), dgdx(xfield.FieldSz());

... /* Use NEWIMAGE to calculate values for f, g and dgdx */
```

```
NEWMAT::ColumnVector  e = g-f;
NEWMAT::ColumnVector  nabla0(3*xfield.CoefSz());
nabla0.Rows(1,xfield.CoefSz()) = (2.0/xfield.FieldSz()) * xfield.Jte(dgdx,e);
```

We want of course all of `nabla0` and we would get that from corresponding field objects for $d_y(x,y,z)$ and $d_z(x,y,z)$

```
NEWMAT::ColumnVector  f(xfield.FieldSz());
NEWMAT::ColumnVector  g(f), dgdx(f), dgdy(f), dgdz(f);

... /* Use NEWIMAGE to calculate values for f, g, dgdx, dgdy and dgdz */

NEWMAT::ColumnVector  e = g-f;
int cfsz = xfield.CoefSz();
NEWMAT::ColumnVector  nabla0(3*cfsz);
nabla0.Rows(1,cfsz) = (2.0/xfield.FieldSz()) * xfield.Jte(dgdx,e);
nabla0.Rows(cfsz+1,2*cfsz) = (2.0/yfield.FieldSz()) * yfield.Jte(dgdy,e);
nabla0.Rows(2*cfsz+1,3*cfsz) = (2.0/zfield.FieldSz()) * zfield.Jte(dgdz,e);
```

In the code above we could actually make do with a single field, provided we want to model the x-, y- and z-displacement fields with the same resolution, which we assume we always want. The important thing in the example above is that the member function `Jte` gets called with different partial derivatives. However, in any actual case we will still need to represent the three displacement fields.

The next entity we would like to calculate is the (approximate) Hessian of O. Looking at equations 8 and 19 we see that it can be written as

$$\mathbf{H}(\mathbf{w}) \approx \frac{2}{XYZ}\mathbf{J}^T(\mathbf{w})\mathbf{J}^T(\mathbf{w}) = \frac{2}{XYZ}\begin{bmatrix} \mathbf{J}_x^T(\mathbf{w})\mathbf{J}_x^T(\mathbf{w}) & \mathbf{J}_x^T(\mathbf{w})\mathbf{J}_y^T(\mathbf{w}) & \mathbf{J}_x^T(\mathbf{w})\mathbf{J}_z^T(\mathbf{w}) \\ \mathbf{J}_y^T(\mathbf{w})\mathbf{J}_x^T(\mathbf{w}) & \mathbf{J}_y^T(\mathbf{w})\mathbf{J}_y^T(\mathbf{w}) & \mathbf{J}_y^T(\mathbf{w})\mathbf{J}_z^T(\mathbf{w}) \\ \mathbf{J}_z^T(\mathbf{w})\mathbf{J}_x^T(\mathbf{w}) & \mathbf{J}_z^T(\mathbf{w})\mathbf{J}_y^T(\mathbf{w}) & \mathbf{J}_z^T(\mathbf{w})\mathbf{J}_z^T(\mathbf{w}) \end{bmatrix} \quad (21)$$

Let us now look at how we would calculate two of the submatrices, $\mathbf{J}_x^T(\mathbf{w})\mathbf{J}_x(\mathbf{w})$ and $\mathbf{J}_x^T(\mathbf{w})\mathbf{J}_y(\mathbf{w})$.

```
BASISFIELD::splinefield      xfield(field_size,knot_spacing);
NEWMAT:ColumnVector          dgdx(xfield.CoefSz()), dgdy(xfield.CoefSz());

... /* Use NEWIMAGE to calculate dgdx and dgdy */

boost::shared_ptr<MISCMATHS::BFMatrix>  Hxx = xfield.JtJ(dgdx);
Hxx->MulMeByScalar(2.0/xfield.FieldSz());
boost::shared_ptr<MISCMATHS::BFMatrix>  Hxy = xfield.JtJ(dgdy,dgdx);
Hxy->MulMeByScalar(2.0/xfield.FieldSz());
```

Please don't be confused by the declaration of `Hxx` and `Hxy`. It is basically a pointer to an object of type `BFMatrix`. The class `BFMatrix` in turn is a virual base class with two derived classes `SparseBFMatrix` and `FullBFMatrix` which implements interfaces to sparse and full matrices respectively.

The final thing that should be pointed out in this section is that it is very easy to write code that is independent of what basis-set we actually want to use. The following code would for example work

```
boost::shared_ptr<basisfield>   myfield;
if (I_like_splines == true) {
  myfield = boost::shared_ptr<splinefield>(new splinefield(fsize,ksp));
}
else {
  myfield = boost::shared_ptr<dctfield>(new dctfield(fsize,order));
}

ColumnVector nabla0 = myfield->Jte(dgdx,e);
nabla0 &= myfield->Jte(dgdy,e);
nabla0 &= myfield->Jte(dgdz,e);
nabla0 *= 2.0/myfield.FieldSz();
boost::shared_ptr<BFMatrix> = myfield->JtJ(dgdx);
etc etc
```

So, you can see that after the initial creation of the field we can do all we need to do with it without actually having to know what type of field it is.

3.4 Sum of squared differences cost-function with scaling

This is a tiny change compared to the "sum of squared differences" described above. But it is of interest because it is the first (most basic) actual cost-function implemented in fnirt, and as such will allow us further insights into the structure of the code. For the more general cases later on it is now useful to define the vector $\boldsymbol{\theta}$ as the concatenation of $\mathbf{w}$, the parameters pertaining to the displacement field, and any other parameters we might want to introduce. These other parameters will typically be used to model variations/differences in intensity that does not pertain to structure and can hence be seen as confounds within the present problem. The very simplest form would be a scalar scaling factor that models global intensity differences between f and g. So, let us now define $\boldsymbol{\theta}$ as

$$\boldsymbol{\theta} = \begin{bmatrix} \mathbf{w} \\ \alpha \end{bmatrix} \tag{22}$$

and the cost-function as

$$O(\boldsymbol{\theta}) = \frac{1}{XYZ}\sum_{z=1}^{Z}\sum_{y=1}^{Y}\sum_{x=1}^{X}(g_{xyz}(\mathbf{w}) - \alpha f_{xyz})^2 \tag{23}$$

or

$$O(\boldsymbol{\theta}) = \frac{1}{XYZ}(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f})^T(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f}) \tag{24}$$

which yields the gradient

$$\nabla O(\boldsymbol{\theta}) = \frac{2}{XYZ}\begin{bmatrix} \mathbf{J}^T(\mathbf{w})(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f}) \\ -\mathbf{f}^T(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f}) \end{bmatrix} \tag{25}$$

and the (approximate) Hessian

$$\mathbf{H}(\boldsymbol{\theta}) = \frac{2}{XYZ}\begin{bmatrix} \mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) & -\mathbf{J}^T(\mathbf{w})\mathbf{f} \\ -\mathbf{f}^T\mathbf{J}(\mathbf{w}) & \mathbf{f}^T\mathbf{f}^T \end{bmatrix} \tag{26}$$

3.4.1 So, how do we calculate that using the `basisfield` class?

The upper left block is identical to before, so we have already seen how to calculate that. It can further be seen that the block consisting of $-\mathbf{J}^T(\mathbf{w})\mathbf{f}$ is of the same general form as the upper portion of the gradient, only with the vector $(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f})$ replaced by $\mathbf{f}$.

```
splinefield  xfield(fsize,ksp), yfield(fsize,ksp), ...
ColumnVector f, g, e, dgdx, dgdy, dgdz;

.../* Calculate g, e, dgdx etc */

ColumnVector nabla0 = xfield.Jte(dgdx,e) & xfield.Jte(dgdy,e) & xfield.Jte(dgdz,e);
nabla0 &= DotProduct(f,e);
nabla0 *= (2.0/xfield.FieldSz());

boost::shared_ptr<BFMatrix>  JtJ;

.../* Calculate the 9 components of JtJ, and put them together */

ColumnVector toprightbit = xfield.Jte(dgdx,f) & xfield.Jte(dgdy,f) & xfield.Jte(dgdz,f);
boost::shared_ptr<BFMatrix>  H = JtJ;
H->HorConcatToMyRight( -toprightbit)
ColumnVector bottomrow = ( -toprightbit.t()) | DotProduct(f,f);
H->VerConcatBelowMe(bottomrow);
H->MulMeBySCalar(2.0/xfield.FieldSz());

/* Gradient and Hessian now ready to use */
```

As tempting as it might seem to try this out straight away, we should mention that there is yet another layer of abstraction though.

3.4.2 The cost-function class and the `nonlinear` toolbox

As part of implementing a toolbox for nonlinear optimisation we defined a cost-function class with the very simple interface

```
class NonlinCF
{
public:
  /* Constructors, destructor and all that malarky */
  virtual double cf(ColumnVector& p) const = 0;
  virtual ColumnVector grad(ColumnVector& p) const;
  virtual boost::shared_ptr<BFMatrix> hess(ColumnVector& p) const;
private:
  /* Stuff */
};
```

It can be seen that the member function `cf` is pure virtual, which means that `NonlinCF` is a virtual base class. Its purpose is to provide a consistent interface for obtaining the cost-function,

its gradient and Hessian. The parameter `p` corresponds to $\boldsymbol{\theta}$ in the paragraph above and *e.g* a call like `mygrad = cf.grad(p)` is literally identical to $\nabla O(\mathbf{theta})$.

Being a virtual base class there will never be any instances of `NonlinCF` but rather of classes derived from it. To make it really concrete let us imagine we want to fit a mono-exponential function to some data under the assumption that errors are normal distributed. So, our model is given by

$$y_i = \theta_1 e^{-\theta_2 x_i} + e_i, \quad e_i \sim N(0, \sigma^2) \tag{27}$$

where we are interested in finding $\boldsymbol{\theta} = [\theta_1\ \theta_2]^T$. We do so by defining a cost-function which is the sum of squared errors between the model predictions and the observed data. To accomplish this we create class which we may call `OneExpCF`, and which down to its bare bones might look something like

```
class OneExpCF: public NonlinCF
{
public:
  OneExpCF(const ColumnVector& px, const ColumnVector& py) : x(px), y(py) {
    /* Should do some error checking here */
  }
  ~OneExpCF() ;
  virtual double cf(const ColumnVector& p) const;
private:
  ColumnVector   x;    // Independent data (times) goes here
  ColumnVector   y;    // "Measured" data goes here
};

double OneExpCF::cf(const ColumnVector& p) const
{
  double cfv = 0.0;
  for (int i=1; i<=x.Nrows(); i++) {
    double err = y(i) - p(1)*exp(-p(2)*x(i));
    cfv += err*err;
  }
  return(cfv);
}
```

As you see we have now added "space" for the data in the class, and defined the member function `cf`. We are a little lazy so we don't override neither `grad` or `hess`. This is OK since the base class `NonlinCF` defines these using numerical differentiation based on the `cf` function that we have just defined. To use this we will further need to create an instance of the class `NonlinParam` which is a glorified struct that contains information about what algorithm we want to use for the minimisation, convergence critera etc. So, all the code we need to write is

```
ColumnVector   x, y;

.../* Get x and y from a file, the user or something. */

OneExpCF      mycf(x,y);
NonlinParam   mypar(2,NL_LM);  // 2 -> We have two parameters
                               // NL_LM -> Use Levenberg-Marquardt
```

```
NonlinOut status = nonlin(mypar,mycf);

if (status != NL_MAXITER) {
  cout % << ``Found values theta1 = `` << (mypar.Par())[0]
       % << ``, theta2 = `` << (mypar.Par()[1] << ``'', found in ``
       % << par.NIter() << `` iterations'';
}
else {
  mypar.SetGaussNewtonType(LM_L)   // Maybe pure Levenberg is better
  mypar.Reset();
  status = nonlin(mypar,mycf);
  if (status != NL_MAXITER) {
    cout % << ``Found values theta1 = `` << (mypar.Par())[0]
         % << ``, theta2 = `` << (mypar.Par()[1] << ``'', found in ``
         % << par.NIter() << `` iterations'';
  }
  else {
    cout % << ``Maximum # of iterations exceeded'';
}
```

As can be deduced from the above `nonlin` is a global function that takes as parameters a `NonlinPar` and a `NonlinCF` (or any class derived from `NonlinCF`) object and finds a set of parameters that minimise the value of the cost-function. There is a *rich* set of parameters that can be set for objects of type `NonlinParam` that should be sufficient to ensure convergence for most types of models and data.

3.4.3 The `fnirt_CF` class

The `fnirt_CF` class is intended as a base-class from which to derive cost-function classes for use in `fnirt` (or `fnirt`-like applications). It is derived from `NonlinCF`, but contrary to what its name implies it doesn't actually implement any cost-function as such. Its purpose is instead to implement functionality that transcends any particular cost-function. It does so through a set of `protected` member functions (*i.e.* functions that are only available from within subclasses of `fnirt_CF`). It may all seem a little esoteric so let us dive straight into an example

```
class fnirt_CF : public NonlinCF
{
public:
  fnirt_CF(const volume<float>&              ref,       // Reference image
           const volume<float>&              obj,       // Object image
           Matrix&                           M,         // Affine matrix
           vector<shared_ptr<basisfield> >   dfield);   // x-, y- and z-fields

.../* Stuff */

  virtual void SetRefMask(const volume<char>&  refm);   // Set a mask in ref-space
  virtual void SetObjMask(const volume<char>&  objm);   // Set a mask in object-space
```

```
.../* More stuff */

protected:
  virtual void SetDefFieldParams(const ColumnVector& p); // Set field coefficients
  virtual const voulme<char>& Mask() const;              // Get mask
  virtual const volume<float>& Robj() const;             // Get warped object image (g)
  virtual const volume4D<float> RobjDeriv() const;       // Get dgdx, dgdy and dgdz
  virtual const volume<float> Ref() const;               // Get ref (f)

.../* Even more stuff */
};
```

To see why this might all be useful let us look at a class derived from `fnirt_CF` that implements the "sum of squared differences with scaling" cost-function described above.

```
class SSD_fnirt_CF: public fnirt_CF
{
public
  SSD_fnirt_CF(/* Super-set of parameters for fnirt_CF constructor */)
  : fnirt_CF(/* Sub-set of all parameters)
  {/* Stuff */}

.../* Stuff */

  virtual double cf(const ColumnVector&  p) const;

.../* More stuff */
};

/* Bare bones version of cf .*/

double SSD_fnirt_CF::cf(const ColumnVector& p) const
{
  SetDefFieldParams(p.Rows(1,3*DefCoefSz()));     // Make fields reflect p
  double sf = p(3*DefCoefSz()+1);                 // Last element is scale-factor
  const NEWIMAGE::volume<float>& ref = Ref();     // Reference image
  const NEWIMAGE::volume<float>& obj = Robj();    // Object image warped according to p
  const NEWIMAGE::volume<char>& mask = Mask();    // Total mask in reference space
  double ssd = 0.0;
  int n = 0;
  for (int k=0; k<RefSz_z(); k++) {
    for (int j=0; j<RefSz_y(); j++) {
      for (int i=0; i<RefSz_x(); i++) {
        if (mask(i,j,k)) {n++; ssd += SQR(obj(i,j,k)-sf*ref(i,j,k));}
      }
    }
  }
  ssd /= double(n);                               // Mean SSD
```

```
  return(ssd);
}
```

and `SSD_fnirt_CF` may be used in an applictaion like *e.g.* fnirt

```
volume<float>   ref, obj; // Reference and Object image
volume<char>    objm;     // Used to mask away that nasty tumor in obj
Matrix          M;        // Affine matrix from fnirt

.../* Read images, mask and affine matrix. */

vector<int>     ksp(3,8); // 8 voxels knot-spacing
vector<int>     isz(3);   // Size of field/ref
isz[0] = ref.xsize(); isz[1] = ref.ysize(); isz[2] = ref.zsize();

vector<shared_ptr<basisfield> >   dfield(3);  // field
for (int i=0; i<3; i++) {
  dfield[i] = shared_ptr<splinefield>(new splinefield(isz,ksp));
}

SSD_fnirt_CF  mycf(ref,obj,M,dfield);
mycf.SetObjMask(objm);
NonlinPar     mypar(mycf.NPar(),NL_LM);  // Levenberg-Marquardt often a good choice

if ((NonlinOut status = nonlin(mypar,mycf)) == NL_MAXITER) {
  cout % << "Rats!" << endl;
}
```

So, making ones cost-function a subclass of `fnirt_CF` makes it relatively easy to implement the cost-function. The functionality inherited from `NonlinCF` makes it easy to perform the actual minimisation and the functionality from `fnirt_CF` facilitates implementing the actual cost-function (and its derivative and Hessian). So in the example above we used a mask in object space because we didn't want the warping to be confused by some ghastly growth that happens to be in poor `obj`. Because the mask is in the space of the image we attempt to warp it means that it will have to be warped along with `obj` in each iteration. This is facilitated by `fnirt_CF` offering a `public` interface `.SetObjMask(mask)` that allows us to specify the mask and the `protected` function `.Mask()` that returns a mask that is the intersection of all the masks (in object and/or reference space) that has been set by the user transformed into reference space.

This is just one example of how the functionality in `fnirt_CF` may facilitate the implementation of new cost-functions.

3.5 Regularisation of the field

The chosen model for non-linear registration offers considerable freedom in terms of different constellations of warps. It is frequently the case that a large set of different warps (as parametrised by different instances of $\mathbf{w}$) yields similar values for the cost-function, and in these cases we need

some way to decide between them. Furthermore, representing the field as a linear combination of basis-fuctions ensures that it is smooth and continous but does *not* not guarantee that it is "one-to-one" and "onto".

The term "one-to-one" means that no two points in the original space $\mathbf{x}$ can map onto the same point in the warped space $\mathbf{x}'$, a condition which is indicated by the Jacobian determinant of the $[x\ y\ z] \rightarrow [x'\ y'\ z']$ transformation becoming zero or negative at one or more grid points. The significance of "onto" is that each point in the transformed space $\mathbf{x}'$ should have a mapping onto some point in the original space $\mathbf{x}$, *i.e.* there must not be any points in the transformed space that "cannot be reached". The "onto" requirement is not really meaningful (*i.e.* we can never hope to enforce it) for a discretely sampled function, but the "one-to-one" condition is widely thought to be important if we are to consider a transform/field as reasonable.

As alluded to above the "one-to-one" requirement is not guaranteed, or even helped, by representing the field by some basis-set. All that guarantees is that when the Jacobian goes negative it does so gradually (over space). A common solution to these problems is to use some form of "regularisation" on the field. This is simply some differentiable function of the field, or the parameters of the field, whose value indicates how "likely" we consider that field to be. It is quite common to use some mechanical analogy such that in the choice between two fields we will consider *e.g.* that with a smaller membrane energy to be the more likely. The mechanical analogy results in the same set of equations as one obtains from considering the sum-of-squared differences cost-function as a likelihood (assuming normal distributed errors) and postulating a multinormal prior on the coefficients of the field. There are forms for the variance-covariance matrix of the prior that corresponds to *e.g.* membrane energy and bending energy.

At present the membrane energy is used for regularisation. Given the definition in equation 1 the membrane energy is defined as

$$E_m = \lambda \sum_{i=1}^{XYZ} \sum_{j=1}^{3} \sum_{k=1}^{3} \left(\left[\frac{\partial d_j}{\partial x_k} \right]_i \right)^2 \tag{28}$$

where the i subscript denotes the ith voxel and where λ is a material dependent constant. If we define a vector $\mathbf{B}^{(i)}_{lnm}$ as a vector containing an "unravelled" version of the lmnth basis-function having been differentiated in the ith direction (*cf* equation 6), the matrix $\mathbf{B}^{(i)}$ as the concatenation of the LMN basis-functions (*cf* equation 7) and the vector $\mathbf{w}_x$ containing the coefficients pertaining to the x-component ($d_x(x,y,z)$) of the field then the membrane energy for the x-component can be expressed as

$$\mathbf{w}_x^T \mathbf{S}^{(\mathbf{x})} \mathbf{w}_x = \mathbf{w}_x^T \left(\mathbf{B}^{(\mathbf{x})\mathbf{T}} \mathbf{B}^{(\mathbf{x})} + \mathbf{B}^{(\mathbf{y})\mathbf{T}} \mathbf{B}^{(\mathbf{y})} + \mathbf{B}^{(\mathbf{z})\mathbf{T}} \mathbf{B}^{(\mathbf{z})} \right) \mathbf{w}_x = \sum_{i=1}^{XYZ} \sum_{k=1}^{3} \left(\left[\frac{\partial d_x}{\partial x_k} \right]_i \right)^2 \tag{29}$$

and the total membrane energy as

$$E_m(\mathbf{w}) = \mathbf{w}^T \mathbf{S} \mathbf{w} = \begin{bmatrix} \mathbf{w}_x^T & \mathbf{w}_y^T & \mathbf{w}_z^T \end{bmatrix} \begin{bmatrix} \mathbf{S}^{(x)} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^{(y)} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{S}^{(z)} \end{bmatrix} \begin{bmatrix} \mathbf{w}_x \\ \mathbf{w}_y \\ \mathbf{w}_z \end{bmatrix} \tag{30}$$

Consequently the gradient and the Hessian of $E_m(\mathbf{w})$ are

$$\nabla E_m = \mathbf{S}\mathbf{w} \tag{31}$$

and

$$\mathbf{H}_{E_m} = \mathbf{S} \tag{32}$$

Hence the cost-function including regularisation, its derivatives and Hessian are

$$O(\boldsymbol{\theta}) = \frac{1}{XYZ}(g(\mathbf{w}) - \alpha f)^T(g(\mathbf{w}) - \alpha f) + \lambda \mathbf{w}^T\mathbf{S}\mathbf{w} \tag{33}$$

,

$$\nabla O(\boldsymbol{\theta}) = \frac{2}{XYZ}\begin{bmatrix} \mathbf{J}^T(\mathbf{w})(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f}) \\ -\mathbf{f}^T(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f}) \end{bmatrix} + \lambda \begin{bmatrix} \mathbf{S}\mathbf{w} \\ 0 \end{bmatrix} \tag{34}$$

and

$$\mathbf{H}(\boldsymbol{\theta}) = \frac{2}{XYZ}\begin{bmatrix} \mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) & -\mathbf{J}^T(\mathbf{w})\mathbf{f} \\ -\mathbf{f}^T\mathbf{J}(\mathbf{w}) & \mathbf{f}^T\mathbf{f}^T \end{bmatrix} + \lambda \begin{bmatrix} \mathbf{S} & \mathbf{0} \\ \mathbf{0}^T & 0 \end{bmatrix} \tag{35}$$

where λ can be seen as an arbitrary weighting of the regularisation versus the sum-of-squared differences, or between the likelihood and the prior if one prefers that perspective.

3.5.1 Implementation in the `basisfield` class

The contribution of the membrane energy is a function only of the field, or rather the parameters defining the field, and can hence be calculated and returned by the basisfield class. Let us for example say that we have declared and defined a `splinefield` like

```
NEWMAT::ColumnVector        w = ...;                        // Whichever way we get them
BASISFIELD::splinefield     xfield(size,knot_spacing);
xfield.SetCoef(w);
```

we can then obtain the mebrane energy and its gradient and hessian at this point in parameter space as

```
double                                  membrane_energy = xfield.MemEnergy();
NEWMAT::ColumnVector                    gradient = xfield.MemEnergyGrad();
boost::shared_ptr<MISCMATHS::BFMatrix> hessian = MemEnergyHess();
```

3.5.2 What is going on with that `BFMatrix` class?

The Hessian $\mathbf{H}$ will rapidly become very large and costly to store and calculate (more about the latter below). A reasonable "bread-and-butter" resolution may be, in the case of using a spline-basis, a $4 \times 4 \times 4$ voxel knot-spacing over the $91 \times 109 \times 91$ matrix of the MNI template. This would result in $25 \times 30 \times 25$ splines requiring 18750 parameters for each of the x-, y- and z-displacement fields. This means that $\mathbf{H}$ is a 56251×56251 matrix, requiring on the order of 25GB to store and represent it, something which is beyond most computers today. This is the reason why for example the DCT-based registration in SPM is limited to a resolution of roughly 20mm (or 10 voxels) isotropically. The spline basis set has a great advantage here in that the Hessian is sparse, meaning that a large proportion of the entries are zero. And that proportion

increases with increased resolution of the warps. No row or column of the Hessian contains more than 343 ($7 \times 7 \times 7$) non-zero elements, and many contain fewer. It is therefore crucial that we have an efficient storage format for the Hessian (or for sparse matrices in general) if we are to reap the full benefits of the spline basis-set. At the same time we do not wish to represent all Hessian matrices as sparse matrices (*e.g.* the full Hessian matrices from the DCT basis set) since that will typically increase storage needs by more than 50% and incurr an execution time penalty when the matrix is indeed full.

This is the reason why our `basisfield` classes, and also the classes derived from `NonlinCF`, return an object of type `BFMatrix`. This is a virtual wrapper-class with two derived classes `FullBFMatrix` and `SparseBFMatrix` which has an API that is sufficient for the needs of the different sub-classes of `NonlinCF`. This is a way to obtain the required polymorphism given the restriction that the matrix-class we commonly use, Newmat, do not have a sparse representation nor has it been designed such that it would be easy to sub-class a sparse-matrix class from it.

Why do we then bother to include the DCT basis-set in the first place? When performing low-resolution (20mm warp resolution or poorer) non-linear warping the full Hessian for the DCT set is feasible to represent and *very* efficient to calculate. This has to do with the properties of the basis and slightly simplified it can be though of as an FFT for Hessian calulation. In contrast the Hessian for the spline set is not very sparse at that resolution, and each element is quite costly to calculate because of the large overlap (in units of voxels) of neighbouring splines. The DCT-set will therefore yield similar results as the spline-set for an order of magnitude shorter execution time.

The advantage of the spline-set becomes obvious when going to medium-resolution registration (10mm warp resolution), when it is no longer possible to represent the full Hessian for the DCT-set. Even if it was possible to represent it the Hessian contains 64 times more elements, each element taking as long to compute as for the low-resolution case. For the spline set the number of elements grow much more slowly with increasing reolution since sparsity also increases with resolution. In addition the cost of calculating each element decreases since the overlap between neighbouring splines decreases. In fact there is but limited difference in the time it takes to calculate the 5616×5616 Hessian for a knot-spacing of 20mm and the 31752×31752 Hessian for a 10mm knot-spacing.

3.6 Approximations for speed

When using Gauss-Newton style optimisation (Levenberg or Levenberg-Marquardt) the costliest operations are the calculation and inversion of $\mathbf{H}$. The example in the preceeding section shows that a 56251×56251 Hessian is in no way unreasonable, and will take a little while both to compute and invert. For this reason the `.JtJ()` routines of the derived classes of `basisfield` are the real bottle necks, and efforts at optimisation should focus on these.

If we look at *e.g.* equation 19 we see that $\mathbf{H}$ is a function of $\mathbf{w}$ and will thus need to be recalculated at each iteration. If we further look at equations 7 and 8 we see that that dependence comes from caculating $\partial\mathbf{g}/\partial x$ (and y and z) at the point $\mathbf{w}$ in the parameter space. What then do we think $\partial\mathbf{g}/\partial x$ would look like when we are reasonable close to convergence? I think it would look very similar to $\partial\mathbf{f}/\partial x$, and that doesn't depend on $\mathbf{w}$. So one trick, that has been played successfully in *e.g.* SPM, is to calculate $\mathbf{H}$ once and for all using $\partial\mathbf{f}/\partial x$. The

approximation is of course worse for the first few iterations when $\mathbf{g}$ and $\mathbf{f}$ are still quite dissimilar. However, at that stage the second order Taylor expansion that underlies Gauss-Newton style minimisation is likely to be a poor approximation anyway. It should also be noted that getting $\mathbf{H}$ "wrong" in a Levenberg/Levenberg-Marquardt minimisation should not affect the end result, just the way and time it takes to get there. A poor approximation to $\mathbf{H}$ would mean that it would take more iterations to converge, but on the other hand that should be weighed against the time and computational effort of each iteration which will be *much* smaller if we can reuse $\mathbf{H}$. Even better, of course, would be if we could in the first iteration calculate the Cholesky decomposition of $\mathbf{H}$ which would greatly facilitate solving for $\boldsymbol{\theta}$ in subsequent iterations.

These are all things that we need to look into empirically. It is neither clear that we will end up using Gauss-Newton style optimisation as our default. In particular the Scaled Conjugate Gradient method seems to do well on the model examples I have looked at, and that does away with calulating $\mathbf{H}$ altogether (though it calculates the gradient twice at each iteration). However, see my reservations below regarding different scales of different sets of parameters.

3.7 Scaling of the gradient vector

Many algorithms for non-linear optimisation suffers quite badly when different parts of the gradient vector (∇O) have vastly different scales. Consider ∇O given by equation 25. It is a $3LMN + 1 \times 1$ vector where the upper $3LMN$ elements are of the form $\mathbf{j}_i^T(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f})$ where $\mathbf{j}_i$ is a vector which is non-zero only over small portion corresponding to the support of spline that it pertains to. This is in contrast to the final element $-\mathbf{f}^T(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f})$ where more or less all elements of $\mathbf{f}$ are non-zero. Intuitively it is also obvious that a unity change of α (say from 1 to 2) will change O massively, whereas a unity change of one of the spline coefficients will move the samling point for a few hundred voxels to varying degrees and will have much smaller effect on O. So we have a vector where $3LMN$ elements are of one scale, and then one element that is hundreds, or even thousands, of times larger.

Let us then consider what would happen when a method that bases its step (in parameter space) on the gradient, *i.e.* attempts to take a step in the $-\nabla O(\boldsymbol{\theta})$ direction. That direction will be completely dominated by the α direction, *i.e.* we attempt to take a long step in the α direction and a short step in all other directions. The problem though is that most likely we only need to take a very short step in the α direction. If we have done an intial global normalisation (as one does) a realistic range might be $0.97 < \alpha < 1.03$. So that means that for the step $\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \lambda\nabla O(\boldsymbol{\theta})$ we will find a very small λ, and therefore it is likely that we will only have taken a *tiny* step of that we would need to take for the other direction. We should note here that although the search directions for *e.g.* Variable-Metric and Conjugate-Gradient methods are not the gradient, they both start ot with that as their initial direction. And because of the way they construct new directions based on previous directions and the present gradient each new directions will have a large proportion of the present gradient in them.

In practice what will tend to happen is that for the first few (can be a small few, but sometimes a large few) attempted steps the gradient will be dominated by $-\mathbf{f}^T(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f})$ so the step lengths become very small and we practicaly do not move at all in any of the $\mathbf{w}$ directions. Then comes a step where we are literally at the bottom of the pass in the α direction and $\partial O/\partial\alpha$ becomes (very close to) zero. The algorithm will then be able to take a sizeable

step along the $\mathbf{w}$ directions and the cost-function will decrease. After that step α is typically non-zero again and there will be another few steps of neglible length until $\partial O/\partial\alpha$ is again zero and the algorithm will again be able to take a "proper" step. So, the problem produces very typical symptoms of slow and jerky convergence. For the particular case of a single scale-factor whose scale is vastly different from the warp parameters ($\mathbf{w}$) it is relatively easy to "fudge" it by redefining O as a summation over $(g_{xyz}(\mathbf{w}) - (1 + c\alpha)f_{xyz})^2$ where c is some suitably small constant (10^{-3} seems to work reasonably well). However the problem is more "general" than that and for example for my top-down-bottm-up method the parameter vector $\boldsymbol{\theta} = [\mathbf{w}^T\mathbf{p}^T]^T$ where $\mathbf{w}$ is a set of spline-coefficients used to model the (magnetic) field and $\mathbf{p}$ are a set of rigid body movement parameters describing any change in subject position between the two acquisitions. The effects on the cost-function is of course much larger for any of the movement parameters than for the warp parameters, and we have again a problem of different scales of parameters. We will see similar problems also if/when we want to include a model for RF inhomogeneity. I suspect/hope it shall be possible to find reasonable fudge factors also for those parameters, but it won't be pretty. It is hardly realistic that we shall be able to find a general solution to it since this is a long standing issue in "non-Newton" non-linear optimisation.

In contrast, for the Newton/Gauss-Newton style algorithms this is a non-issue. If we take again the example of the gradient described by equation 25, remember that the step for a Newton style algorithm is of the general form $\boldsymbol{\theta}^{(k+1)} = \mathbf{H}(\boldsymbol{\theta})^{-1}\nabla O(\boldsymbol{\theta})$ and look at the element in the bottom-right corner of $\mathbf{H}$ we see that that, just as the large element of ∇O, that too is much larger than the other elements of $\mathbf{H}$. A, very crude, approximation to the α component of the Newton-step is $-\mathbf{f}^T(\mathbf{g}(\mathbf{w}) - \alpha\mathbf{f})/\mathbf{f}^T\mathbf{f}$, and in practice this typically turns out to be one of the smallest components of the step. The information in the Hessian will for all practical purposes "calibrate" any akward scalings in the gradients without any need for finding ones own fudgy factors. This means that convergence tend to be steady and reliable, which makes it easy to implement convergence criteria. Non-Newton type algorithms, as described above, will for poorly scaled parameters exhibit long stretches of sometimes tens of iterations during which changes in cost-function are very small, and may then all of a sudden make another "jerk" and change by maybe a factor of two (I have seen it happen). How, then, does one implement a test for convergence?

Intrestingly the Variable-Metric methods are actually supposed to be able to handle this type of problem (hence the name), but in my practical tests I have found nothing to indicate that would actually be the case.

For this reason I tend to favor Gauss-Newton type algorithms over the various flavours of Variable-Metric/Conjugate-Gradient methods out there that doesn't require calculation or representation of the Hessian. Although the price of course is that it complicates the implementation quite considerably and represents a large proportion of the total calculations.

4 EXPERIMENTS

Experiments have been performed registering different structural images against each other and against templates acquired with similar tissue contrasts. This has been performed both for T1-weighted and FA (Fractional anisotropy) images. Tests have been aimed at finding

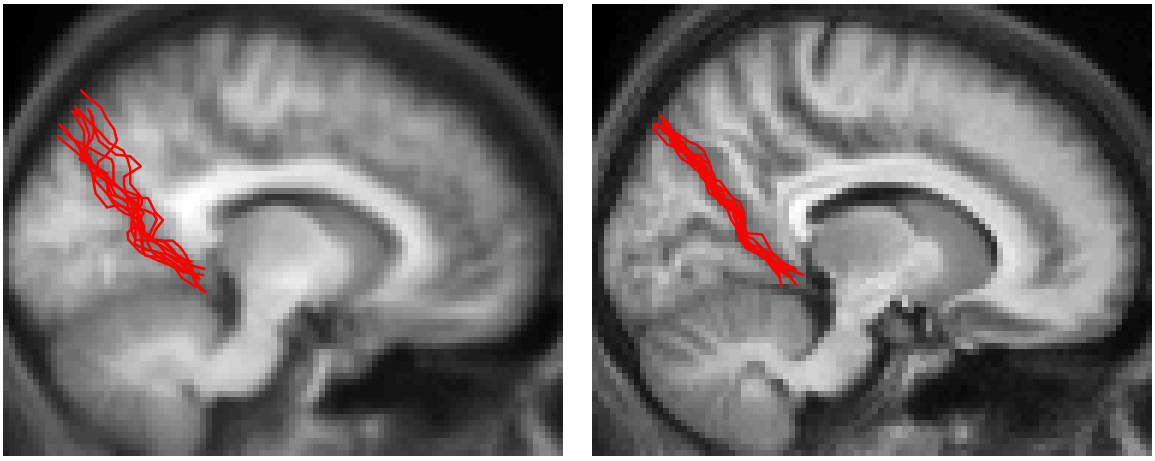

Figure 1: Sagittal slice at y=6mm in the MNI-space. On the left is an average across 10 subjects after affine and on the right after non-linear registration. The parietal-occipital fissure was manually traced in each subjects normalised scan and all ten traces were overlayed on the average images. Not the considerably higher level of detail in the non-linear mean, indicating better alignment of structure, and the considerably smaller inter-subject variability of the manual traces.

schemes/schedules of combinations of subsampling of images and field, regularisation weight and smoothing of images.

5 RESULTS

An example of the results from the non-linear registration is demonstrated in figures 1 and 2. These are T1-weighted structural images that were registered to the non-linear avg152 template using a three step procedure. Sub-sampling of the images were performed at factors 4,2,1 with knot-spacings 40, 20 and 10mm. Regularisation was set at $\lambda = 0.2$, 0.2 and 0.4, and within each optimisation-step λ was a product of the tentative λ and the mean-sum-of-squared deviations, leading to a succesive relaxation of the regularisation. Running the method to converegence at full image resolution and a knot-spacing of 10mm reguires almost an hour on an intel Mac laptop. In this example the values of the Jacobian matrix was between 0.3 and 3 for all voxels within all subjects, demonstrating that the inter-subjecr agreement was achieved without any "extreme" volume changes. We believe that this is largely attributed to the pyramid approach, and if attempting to register to the full resolution in a single step the warps tend to more "extreme".

6 CONCLUSIONS

We have described the implementation of a method/framework for small-displacement non-linear registration of brain MR images. The results look promising and a first release of the

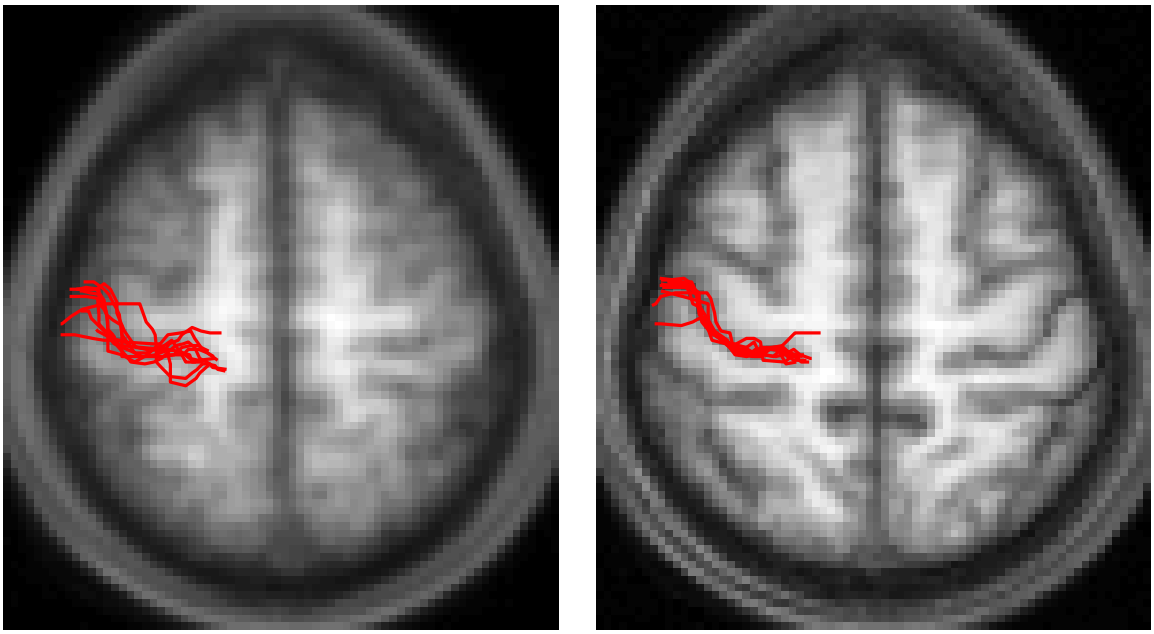

Figure 2: Transversal slice at z=56mm in the MNI-space of the same ten subjects as in figure 1. Average across affine registered images to the left and of non-linearly registered to the right. Note how the superior frontal sulcus, the precentral, central and postcentral sulci, the intraparietal sulcus and the cingulate gyrus are all clearly discernible in the non-linear average indicating a high degree of overlap. In the affine average in contrast it is really only the central and the cingulate gyrus that are discernible, and even those with large amount of blurring. This is further demonstrated by the manual traces of the central sulcus.

software is scheduled to July 2007. Future work will aim at validation, finding the best set of parameters/schedules for different types of images and at a better modelling of the signal by including bias-field, and a physics based mapping of intensities between the two image (or between the template and the input image).