Chapter 5
Introducing directionality with diffusion tensors

In this chapter, we focus on how to transfer information from diffusion tensor imaging (DTI) data to our finite element methods. To do so, we will need to overcome a few practical challenges. In particular, the raw DTI data can contain non-physiological data, especially near the CSF. Moreover, the raw DTI data is represented both in terms of a different coordinate system and at a different resolution than the computational mesh. To overcome the first challenge, we will use local extrapolation of nearby valid values; to overcome the second challenge, we will co-register [1] the data with the images used to construct the computational mesh.

Specifically, we will:

- process the diffusion tensor images to extract mean diffusivity and fractional anisotropy [2] data, and
- map the DTI tensor data into a finite element representation created from the T1-weighted images.

[1] See Section 5.2.3.

[2] Mean diffusivity and fractional anisotropy are defined in (5.1) and (5.2), respectively.

© The Author(s) 2022

K.-A. Mardal et al., *Mathematical Modeling of the Human Brain*,
Simula SpringerBriefs on Computing 10,
https://doi.org/10.1007/978-3-030-95136-8_5

5.1 Extracting mean diffusivity and fractional anisotropy

5.1.1 Extracting and converting DTI data

The DTI data must first be extracted from a DICOM dataset. We use DicomBrowser to extract a DTI series from the book data-set in Chapter 2.3, and the resulting files are available in `dicom/ernie/DTI`. Our next task is to convert the extracted DTI images to a single volume image and to produce supplementary information files about the DTI image data for downstream postprocessing. Various open source tools are available for the processing of DTI data [60]. Here, we continue to use FreeSurfer and its associated command-line tools. As in chapter 3.1.2, we can select any of the files extracted from the `DICOM` DTI data (`dicom/ernie/DTI`) to start the process; here, we arbitrarily choose `IM_1496` and launch the FreeSurfer command `mri_convert`:

```
$ cd dicom/ernie/DTI
$ mri_convert IM_1496 dti.mgz
```

This process, when successful, creates three files: `dti.mgz`, `dti.bvals`, and `dti.voxel_space.bvec`. The last two, plain text files, contain information regarding the b-values and b-vectors associated with the DTI data. The b-vectors and b-values are selected as part of the imaging process; they determine the direction (b-vector) and strength (b-value) of the pulsed magnetic diffusion gradient used during the diffusion weighted imaging scan. For instance, Figure 5.1 shows an axial slice measured with the same choice of b-value but different b-vectors. Once the scan has taken place, we can read this information but it cannot be altered without scanning the patient again.

5.1.2 DTI reconstruction with FreeSurfer

Next, we aim to reconstruct comprehensive DTI data from the volume, b-value, and b-vector files using the FreeSurfer command `dt_recon`. The command takes an input volume (following `--i`), b-vector and b-values files (following `--b`), an output directory `--o`, and the `recon-all` subject ID `--s` (see Chapter 3.1.2). Within our book data directory `dicom/ernie/DTI`, we can launch the following commands:

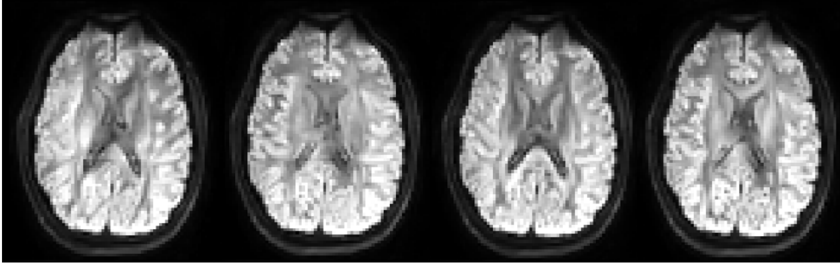

Fig. 5.1 Axial DTI slices measured with different b-vectors. The resolution in the diffusion tensor image is typically lower (here, 96x96x50) compared to that in the T1 images; the latter are, canonically, 256x256x256.

```
$ export SUBJECTS_DIR=my-freesurfer-dir
$ dt_recon --i dti.mgz --b dti.bvals dti.voxel_space.bvecs --s
ernie --o $SUBJECTS_DIR/ernie/dti
```

with `my-freesurfer-dir` replaced by the FreeSurfer subject's directory (e.g. `freesurfer/` from the book data-set).

This command produces multiple output files,[3] including `tensor.nii.gz`, `register.dat`, and `register.lta`. The registration in `dt_recon` uses the registration command `bbregister`[4] to register the DTI data [3]. Files with the suffix .nii are in the NIfTI format. Of these, `tensor.nii.gz` is the spatially varying diffusion tensor. Further, an eigendecomposition of this tensor in terms of spatially varying eigenvalues λ_1, λ_2, and λ_3 and eigenvectors v_1, v_2, and v_3 is given in the files `eigvals.nii.gz` and `eigvec1.nii.gz`,`eigvec2.nii.gz`, and `eigvec3.nii.gz`.

[3] The command above will store the files in $SUBJECTS_DIR/ernie/dti. Alternatively, you can run `mri2fem/chp5/all.sh` which will create a directory `mri2fem/chp5/ernie-dti` that includes the same set of the files as well. You will need FSL installed to use `dt_recon` (see Chapter 2.4.1).

[4] This registration step is done automatically by FreeSurfer using the subject's previously FreeSurfer-processed data that is assumed to be available at this stage of the book; see Chapter 3.1.2 for the necessary steps. The mathematical details of coregistration are further discussed in Section 5.2.3.

5.1.3 Mean diffusivity and fractional anisotropy

In addition, `dt_recon` produces the NIfTI files `adc.nii.gz` and `fa.nii.gz` for the mean (or apparent) diffusivity (MD) and fractional anisotropy (FA), respectively. The mean diffusivity is given by

$$\mathrm{MD} = \frac{1}{3}(\lambda_1 + \lambda_2 + \lambda_3), \tag{5.1}$$

and fractional anisotropy is defined [39] by

$$\mathrm{FA}^2 = \frac{1}{2}\frac{(\lambda_1 - \lambda_2)^2 + (\lambda_2 - \lambda_3)^2 + (\lambda_3 - \lambda_1)^2}{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}. \tag{5.2}$$

NIfTI files can be viewed in ParaView. You might first need to enable

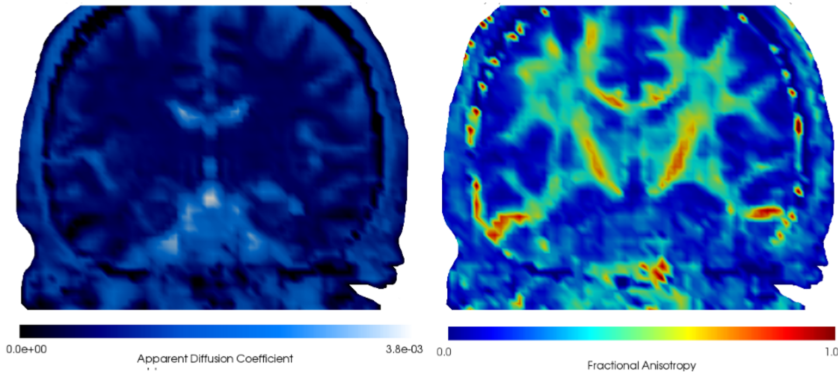

Fig. 5.2 Mean diffusivity (left) and fractional anisotropy (right) as shown in `ParaView`.

the NIfTI viewer plugin by selecting the ParaView menu option labeled [Tools→Manage Plugins], selecting [5] [AnalyzeNifTIReaderWriter] and then clicking [Load Selected]. You can then open and view .nii files in ParaView,

[5] The correct plugin may also be named AnalyzeNifTIIO in earlier versions (i.e. before 5.7.0) of ParaView.

just as you would any other file. To unzip .nii.gz to .nii, one can use `mri_convert`:

```
$ mri_convert adc.nii.gz adc.nii
$ mri_convert fa.nii.gz fa.nii
```

Let us open `adc.nii` and verify that we can reproduce Figure 5.2 (left); the process will be the same for `fa.nii`. After loading the AnalyzeNifTIIO plugin, described above, and opening `adc.nii` click Apply. You will likely see an empty three-dimensional cube in the view window. In the left pane, find the `Representation` option and change this to `Volume`. You should now see something that looks similar to a 'brain in a box' viewed from the top. Now click Filters→Alphabetical and select `Slice`. In the left pane, find the option labeled `Normal`; it should be under the option labeled `Origin`. Change the `Normal` from `1 0 0` to `0 1 0` and click Apply.

In the left pane, once more, hide the object `adc.nii` by clicking the picture of the eye next to its name. Now, rotate the view window so that you can see the X–Z plane; the result should look similar to Figure 5.2. We can make it look more similar by changing the color scheme. In the left pane, find the section labeled `Coloring`. Mouse over the buttons here until you find the button labeled `Choose preset`. Click this and select the `Black, blue and white` color scheme, click Apply and then close the color scheme preset window. The image you see now was saved and post-processed to remove the border outside the skull to produce Figure 5.2 (left). You can repeat these steps with `fa.nii`, this time using the `jet` color scheme, to reproduce Figure 5.2 (right).

The average FA value is generally around 0.5 and changes by around 2% between day and night [68]. Anisotropy decreases with age, declining around 14% between 30 to 80 years [40] and can change by up to 50% in certain areas of the brain of a person with Alzheimer's disease compared with healthy subjects [49]. In the `ernie` data (Figure 5.2), the median white matter FA value is 0.3, with a minimum of 0.009 and a maximum of 0.9998.

5.2 Finite element representation of the diffusion tensor

In this section, we:

- ensure that the DTI data have a valid eigendecomposition (with positive eigenvalues),

- map the DTI tensor into a finite element tensor function defined on a finite element mesh, and
- briefly discuss co-registration.

5.2.1 Preprocessing the diffusion tensor data

The DTI data can be quite rough compared to the T1 data and our corresponding finite element meshes; DTI data is typically at a low resolution of 96x96x50 while T1 resolution is typically much higher at 256x256x256. [6] Moreover, the signal can be disturbed near the cerebrospinal fluid (CSF), which makes the data in certain areas of the cortical gray matter and in regions near the ventricle system less reliable. Indeed, inspection of the eigenvalues of the DTI tensor shows non-physiological (zero and/or negative) eigenvalues. To ensure a physiologically (and mathematically) reasonable diffusion tensor, we recommend preprocessing the diffusion tensor prior to numerical simulation. In particular, in this chapter we present two scripts that:

- check the DTI tensor data for non-physiological values and
- replace non-physiological with physiological values in the DTI tensor,

respectively.

Creating brain masks

First, we will use FreeSurfer to create masks of the brain. A mask is a type of filter where voxels (significantly) outside the brain are set to zero and all other voxels are set to a value of one. Using our white matter parcellation data (included in `freesurfer/ernie/mri/wmparc.mgz`), we can create brain masks as follows:

```
$ mri_binarize --i wmparc.mgz --gm --dilate 2 --o mask.mgz
```

The `dilate` flag determines the extent to which the mask should be extended outside the brain surface provided by `wmparc.mgz`. Examples of such masks are shown in Figure 5.3.

[6] See Figure 2.3 (left) versus Figure 5.1.

Fig. 5.3 Brain masks created using `mri_binarize` with `dilate` ranging from zero to three.

Examining the DTI data values

We can work with the DTI data in a very similar manner as we did for the parcellation (image) data in Chapter 4.4.1. We will again use NiBabel to load the image data, use the `vox2ras` functions for the mapping between the different image coordinate systems (DTI voxel space and T1 voxel space), and process the data as NumPy arrays. The complete script can be run as

```
$ cd mri2fem/chp5
$ python3 check_dti.py --dti tensor.nii.gz --mask mask.mgz
```

We import the key packages:

```
import argparse
import numpy
import nibabel

from nibabel.processing import resample_from_to
numpy.seterr(divide='ignore', invalid='ignore')
```

We define the function `check_dti_data` that takes the DTI tensor and mask files as input:

```
def check_dti_data(dti_file, mask_file, order=0):
    # Load the DTI image data and mask:
    dti_image = nibabel.load(dti_file)
    dti_data = dti_image.get_fdata()

    mask_image = nibabel.load(mask_file)
    mask = mask_image.get_fdata().astype(bool)

    # Examine the differences in shape
    print("dti shape  ", dti_data.shape)
    print("mask shape ", mask.shape)
    M1, M2, M3 = mask.shape
```

Now, the important coordinate transformations can be handled as follows:

```
    # Create an empty image as a helper for mapping
    # from DTI voxel space to T1 voxel space:
    shape = numpy.zeros((M1, M2, M3, 9))
    vox2ras = mask_image.header.get_vox2ras()
    Nii = nibabel.nifti1.Nifti1Image
    helper = Nii(shape, vox2ras)

    # Resample the DTI data in the T1 voxel space:
    image = resample_from_to(dti_image, helper, order=order)
    D = image.get_fdata()
```

Before computing eigenvalues, we run

```
    # Reshape D from M1 x M2 x M3 x 9 into a N x 3 x 3:
    D = D.reshape(-1, 3, 3)

    # Compute eigenvalues and eigenvectors
    lmbdas, v = numpy.linalg.eigh(D)
```

and we compute the fractional anisotropy and check the validity of each voxel value, as follows:

```
    # Compute fractional anisotropy (FA)
    FA = compute_FA(lmbdas)

    # Define valid entries as those where all eigenvalues are
    # positive and FA is between 0 and 1
    positives = (lmbdas[:,0]>0)*(lmbdas[:,1]>0)*(lmbdas[:,2]>0)
    valid = positives*(FA < 1.0)*(FA > 0.0)
    valid = valid.reshape((M1, M2, M3))
```

```
    # Find all voxels with invalid tensors within the mask
    ii, jj, kk = numpy.where((~valid)*mask)
    print("Number of invalid tensor voxels within the mask ROI:
                                ", len(ii))

    # Reshape D from N x 3 x 3 to M1 x M2 x M3 x 9
    D = D.reshape((M1,M2,M3,9))

    return valid, mask, D
```

The above snippet makes use of the function `compute_FA`, which is also defined in `check_dti.py`, to compute (5.2). The result is a vector `FA` whose entries contain the fractional anisotropy computed at each available DTI data location. The term `positives` is a binary vector, with the same number of entries as `FA`; it has a value of one if all three of the eigenvalues for the region corresponding to the array index are positive, and zero otherwise. The vector `valid` is therefore a second binary vector whose indices correspond to the locations where DTI data are available. The value at each index of `valid` is one precisely when all of the eigenvalues are positive and the fractional anisotropy there is larger than zero but less than one. The `valid` vector is therefore a mask that indicates where the DTI tensor contains physically admissible values. The `valid` mask is then reshaped [7] to fit the dimensions of the original `mask`, created from the `mask_file`, and the number of zeros, corresponding to invalid entries, is computed and reported in the final lines.

Improving DTI values by extrapolation and resampling to T1 space

If numerous invalid DTI voxel data are reported, by `check_dti.py` as discussed above, we can attempt to improve the DTI data by extrapolating from adjacent valid voxel locations to correct nearby invalid data. The correction script is `mri2fem/chp5/clean_dti_data.py` and can be run as

```
$ cd mri2fem/chp5
$ python3 clean_dti_data.py --dti tensor.nii.gz --mask mask.mgz
--out tensor-clean.nii
```

[7] The term *reshaped* here means that the (tensor) data is reorganized into an expected form. An example would be reshaping a 1×9 (row) tensor to a 3×3 (matrix) tensor by putting the first three entries of the 1×9 tensor in the first row, the next three in the second row and the last three in the final row of the 3×3 tensor.

and the main functionality reads

```
def clean_dti_data(dti_file, mask_file, out_file, order=3,
                   max_search=9):
    valid, mask, D = check_dti_data(dti_file, mask_file,
                                    order=order)
    # Zero out "invalid" tensor entries outside mask,
    # and extrapolate from valid neighbors
    D[~mask] = numpy.zeros(9)
    D[(~valid)*mask] = numpy.zeros(9)
    ii, jj, kk = numpy.where((~valid)*mask)
    for i, j, k in zip(ii, jj, kk):
        D[i, j, k, :] = \
            find_valid_adjacent_tensor(D, i, j, k, max_search)

    # Create and save clean DTI image in T1 voxel space:
    mask_image = nibabel.load(mask_file)
    M1, M2, M3 = mask.shape
    shape = numpy.zeros((M1, M2, M3, 9))

    vox2ras = mask_image.header.get_vox2ras()
    Nii = nibabel.nifti1.Nifti1Image
    dti_image = Nii(D, vox2ras)

    nibabel.save(dti_image, out_file)
```

The first operation carried out by the `clean_dti_data` function is to call `check_dti_data`, which we discussed in the previous section. Recall that, among other things, the `check_dti_data` function returns a tensor representation (`D`) of the DTI data that has been converted from DTI voxel space coordinates to T1 voxel space coordinates. [8] Next, we will search for a valid tensor in directly adjacent voxels using the function `find_valid_adjacent_tensor` defined in `clean_dti_data.py`. If no valid tensor is found nearby, the search range is iteratively increased.

The script determines that a nearby tensor, in the valid region, contains valid data if a non-zero mean diffusivity (MD) is also calculated there. Once one or more, nearby valid value(s) are found, replacement data is chosen. If only one valid value is found, it is directly used. If there are multiple valid

[8] T1 coordinates are the same coordinates used by the computational meshes that were constructed, in previous chapters, from the surfaces extracted from FreeSurfer segmented T1 data. Thus, `D` is now expressed in terms of coordinates that make sense when used alongside the computational meshes

tensors within the search range, the tensor data[9] with MD value closest to the median of the non-zero MD is chosen as a replacement:

```
def find_valid_adjacent_tensor(data, i, j, k ,max_iter):
    # Start at 1, since 0 is an invalid tensor
    for m in range(1, max_iter+1) :
        # Extract the adjacent data to voxel i, j, k
        # and compute the mean diffusivity.
        A = data[i-m:i+m+1, j-m:j+m+1, k-m:k+m+1,:]
        A = A.reshape(-1, 9)
        MD = (A[:, 0]+ A[:, 4] + A[:,8])/3.

        # If valid tensor is found:
        if MD.sum() > 0.0:
            # Find index of the median valid tensor, and return
            # corresponding tensor.
            index = (numpy.abs(MD - numpy.median(MD[MD>0]))).
                                               argmin()
            return A[index]

    print("Failed to find valid tensor")
    return data[i, j, k]
```

5.2.2 Representing the DTI tensor in FEniCS

With the DTI data checked and potentially improved, we are now ready to map our preprocessed DTI image (now in T1 voxel space) onto a FEniCS mesh. We will use the code located in `mri2fem/chp5/dti_data_to_mesh.py` to accomplish this task. To begin, we assume that we have a `mesh` available (e.g. `ernie-brain-32.h5` from Chapter 4.4.2), that we have loaded the clean DTI image and data in `dti_image` and `dti_data`, respectively, and that we have the `ras2vox` transform associated with this image. We can retrieve the `vox2ras` and `ras2vox` transformations associated with the data by

[9] Because of the way the `valid` mask is constructed, a tensor with invalid data can violate either the required condition that all of the eigenvalues must satisfy $\lambda_i > 0$ or the required condition that the FA must satisfy $0 <$FA< 1. In either case, the search for a nearby valid tensor identifies a nearby candidate and replaces the whole of the tensor information at the invalid tensor location. Thus, all of required conditions are satisfied, at the previously invalid location, after the data replacement.

```
# Transformation to voxel space from mesh coordinates
vox2ras = dti_image.header.get_vox2ras_tkr()
ras2vox = numpy.linalg.inv(vox2ras)
```

To represent the diffusion tensor in FEniCS, we create a FEniCS `Function` over a `TensorFunctionSpace` of (discontinuous) piecewise constant polynomial fields (`"DG", 0`):

```
# Create a FEniCS tensor field:
DG09 = TensorFunctionSpace(mesh, "DG", 0)
D = Function(DG09)
```

For each cell, we need to associate an identifying coordinate value so that we can associate the cells of our mesh to the voxel data. One possibility is to extract the cell midpoints as we have done before; here, we opt to extract the coordinates of the degrees of freedom associated with a DG `FunctionSpace` object that we will define on our mesh and convert these to voxel indices:

```
# Get the coordinates xyz of each degree of freedom
DG0 = FunctionSpace(mesh, "DG", 0)
imap = DG0.dofmap().index_map()
num_dofs_local = (imap.local_range()[1] \
                  - imap.local_range()[0])
xyz = DG0.tabulate_dof_coordinates()
xyz = xyz.reshape((num_dofs_local, -1))

# Convert to voxel space and round off to find
# voxel indices
ijk = apply_affine(ras2vox, xyz).T
i, j, k = numpy.rint(ijk).astype('int')
```

The above snippet first retrieves the coordinates of the `TensorFunctionSpace` degrees of freedom on our mesh and applies the `ras2vox` transformation to determine coordinates in voxel space.

We can now reshape the DTI data into a cell-wise structure based on the extracted indices[10] (now in voxel space):

```
# Create a matrix from the DTI representation
D1 = dti_data[i, j, k]
```

[10] Voxels are located based on the degree of freedom (DOF) coordinates from the `FunctionSpace` object. This approach guarantees that there are no missing values as every coordinate maps to some voxel. However, some voxels may correspond to more than one mesh cell as there may be more cells in the mesh than there are voxels e.g. if the mesh has a lower resolution than the resolution of the T1 (voxel) image space.

```
print(D1.shape)
```

With the reshaped DTI data in hand, we assign these to a FEniCS tensor field, `D`, allowing the data to be saved alongside the mesh data.

```
# Assign the output to the tensor function
D.vector()[:] = D1.reshape(-1)
```

The FEniCS tensor field DTI data can be saved alongside the mesh for later use in FEniCS simulations with

```
# Now store everything to a new file - ready for use!
hdf = HDF5File(mesh.mpi_comm(), outfile, 'w')
hdf.write(mesh,"/mesh")
hdf.write(D, "/DTI")
```

The resulting fiber directions, shown in Figure 5.4, can be inspected visually.

5.2.3 A note on co-registering DTI and T1 data

As we have seen, FreeSurfer uses several different coordinate systems to label the position of data in its various output files. Thus, to combine different types of data into something we can use in FEniCS simulations, we need to extract information about the different coordinate systems used in the files and be able to map between these different coordinate systems. This process is known as co-registration. The scripts we have presented use NiBabel functionality to handle co-registration; this section provides additional information regarding co-registration, for both context and completeness.

In short, let $x_1 = (x_1, y_1, z_1)$ and $x_2 = (x_2, y_2, z_2)$ represent the same physiological point in $\mathbb{R}^3$ but represented with respect to two different coordinate systems (bases). Then, there is an affine transformation such that

$$x_2 = A\,x_1 + b, \tag{5.3}$$

for $A \in \mathbb{R}^{3\times3}$ and $b \in \mathbb{R}^3$. The mapping is often stored instead as a 4×4 matrix, where the last row can be ignored. As this equivalent 4×4 representation often appears in the discussions, and software documentation, within the neuroimaging community, we also show it here; the above affine transformation (5.3) can also be written as:

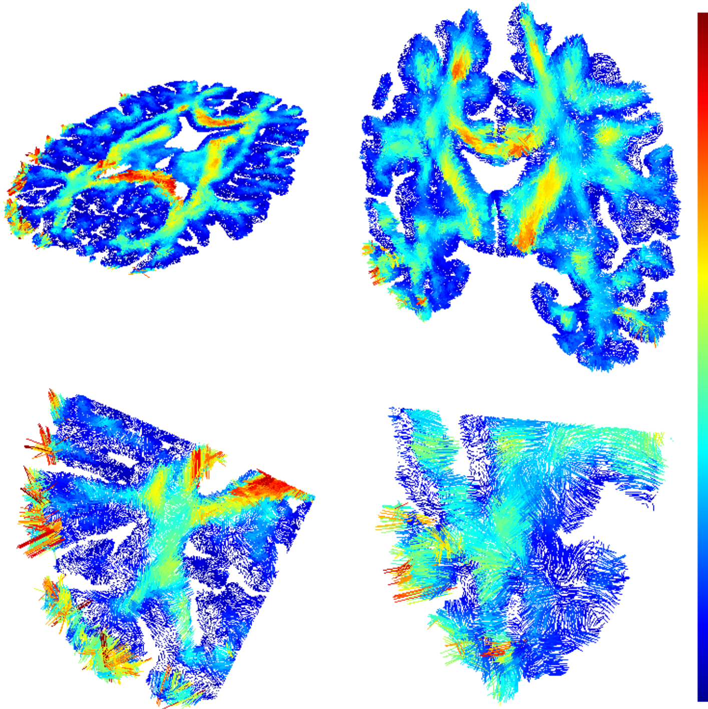

Fig. 5.4 Upper panels show fiber directions (DTI eigenvectors) colored by the fractional anisotropy in the axial and coronal planes. The lower panels show a zoom focusing on the boundary between gray matter and the cerebrospinal fluid. Note that the vector nature of the data can be seen more clearly in bottom panel images where the fibers can be seen to have clear directionality.

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix},$$

where the a_{ij} are the entries of the matrix A and the b_i are the entries of the vector b.

The term *co-registration* specifically refers to the determination of the transformation matrix A and vector b corresponding to a pair of files. A key step in the co-registration of T1 and DTI images, or any pair of images in general, is to ascertain the type of coordinate system used when initially storing these images. Towards this end, we can make use of the `mri_info` command. Coordinate system information regarding the FreeSurfer-processed T1 images is stored in the file `orig.mgz`. We can interrogate this file by:

```
$ cd $SUBJECTS_DIR/ernie/mri
$ mri_info orig.mgz --orientation
LIA
```

The output `LIA` means that the T1 image files were generated with respect to the 'Left Inferior Anterior' coordinate system (see [3] for details). Coordinate system information regarding the FreeSurfer-processed DTI images is stored in the file `tensor.nii.gz`. We can interrogate this file, once more using `mri_info`, by:

```
$ cd $SUBJECTS_DIR/ernie/dti
$ mri_info tensor.nii.mgz --orientation
LPS
```

The coordinate systems can be understood as follows: the positive direction in the sagittal plane can be either (L)eft or (R)ight, the positive direction in the coronal plane can be either (P)osterior or (A)nterior, and the positive direction in the axial plane can be either (I)nferior or (S)uperior. Furthermore, the order of the planes can be different, that is, the third axis might not correspond to the axial plane. For instance, let us examine the coordinate systems described by the abbreviations *LIA* and *LPS*. We see that the coronal plane corresponds to the third axes (A) in *LIA* and second axes (P) in *LPS*, and we have the opposite for the axial plane (I vs. S). Thus, these coordinate systems differ by the choice of a positive direction in the coronal and axial planes, in addition to their order.

Both coordinate systems describe voxel spaces, and we thus need to take into account any difference in voxel sizes. We can obtain voxel sizes (in millimeters) by further using `mri_info`:

```
$ cd $SUBJECTS_DIR/ernie/dti
$ mri_info tensor.nii.gz | grep voxel\ sizes
voxel sizes: 2.500000, 2.500000, 2.500000
```

```
$ cd $SUBJECTS_DIR/ernie/mri
$ mri_info orig.mgz | grep voxel\ sizes
voxel sizes: 1.000000, 1.000000, 1.000000
```

We observe that the voxel sizes differ, and therefore the transformation matrix needs to be scaled from 2.5 mm to 1.0 mm. Thus, the matrix transformation will have the form:

$$A = \begin{bmatrix} 0.4 & 0 & 0 \\ 0 & 0 & -0.4 \\ 0 & -0.4 & 0 \end{bmatrix}.$$

The vector b gives the difference between the origins of the two coordinate systems.

Note, however, that this affine transformation matrix is not quite realistic. First, it assumes that there is no rotational difference between the brains. Second, due to the lack of offset vector b as in (5.3), this transformation assumes that the origins have the same anatomical position. This is unlikely to be the case, since the magnetic resonance images differ in modality or occurrence (i.e. taken at different times). Therefore, to find the affine transformation matrix, we need to find the optimal overlap of the brain contour in the magnetic resonance images. This can be done manually, but it is preferable to do this using registration tools such as `bbregister`, which was used with `dt_recon` in Chapter 5.1.2. In our example, the affine transformation matrix can computed by taking the inverse of the augmented matrix found in `register.lta`[11] located in folder `$SUBJECTS_DIR/ernie/dti`. The augmented matrix is a combination of the matrix A and the vector b with the following structure:

$$\begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix}$$

The approximated transformation matrix becomes

$$A = \begin{bmatrix} 0.4 & 0.0 & -0.1 \\ -0.1 & 0.0 & -0.4 \\ 0.0 & -0.4 & 0.0 \end{bmatrix},$$

and the translation vector

$$b = \begin{bmatrix} 9.0 \\ 106.4 \\ 7.7 \end{bmatrix}.$$

[11] This file was created by bbregister as part of the dt_recon command discussed in Section 5.1.2.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.