BE 537 - Assignment 2

November 13, 2022

1 Introduction

We will be implementing the non-symmetric version of the *Log-Domain Demons* algorithm by Vercauteren et al. (2008). Before the advent of deep learning image registration, the approach was arguably the most widely used deformable image registration approach because of its speed and ability to produce diffeomorphic transformations. You may find it easiest to follow the description of the algorithm in Section 3.2.3 (Algorithm 4) in the Non-Parametric Volumetric Registration (NVR) book chapter.

In this assignment, most of the code is already provided. There are six tasks in the base component of the assignment that require answering mathematical questions or coding, indicated in bold below. The total number of points for these seven tasks is 70.

2 Preliminaries

[1]:
```
from google.colab import drive
drive.mount('/content/drive')

import sys
sys.path.insert(0, 'drive/MyDrive/be537/assignments/assignment2/package/')
```

```
Mounted at /content/drive
```

2.1 Load required packages

I have provided an implementation of the functions from the first assignment in a Python module `be537hw1`. You will use some of these functions again in this assignment. There are also a few helpful functions in module `be537hw2` that we will use. To see the listing of functions available in `be537hw1` and `be537hw2`, type

```
help('be537hw1')
help('be537hw2')
```

The cell below loads the required packages and sets the data path. **Update the data path for your environment**.

[2]:
```
# Import required libraries
import os
import numpy as np
import nibabel as nib
import matplotlib.pyplot as plt
import torch
import torch.nn.functional as tfun

# Import the functions from the first assignment
from be537hw1 import *
from be537hw2 import *

# Configure matplotlib options
import matplotlib
%matplotlib inline
matplotlib.rcParams['figure.figsize'] = [10, 9]

%precision %.4f


# Path to the data for this experiment
data_path='drive/MyDrive/be537/assignments/assignment2/package/data/'
```

2.2 Set up GPU capabilities

The cell below sets up a CUDA device to use with torch, if available to you. If you want to use the CPU even if a GPU is present, set `force_cpu` to be `True`.

You can create tensors that reside on the GPU with the command `T = torch.Tensor(..., device=cuda)` and copy existing tensors to the GPU with the command `T.to(cuda)`. If you do not have a GPU, these commands will still work.

- You do not need a GPU for the base component of the assignment, but I recommend using it for the extensions

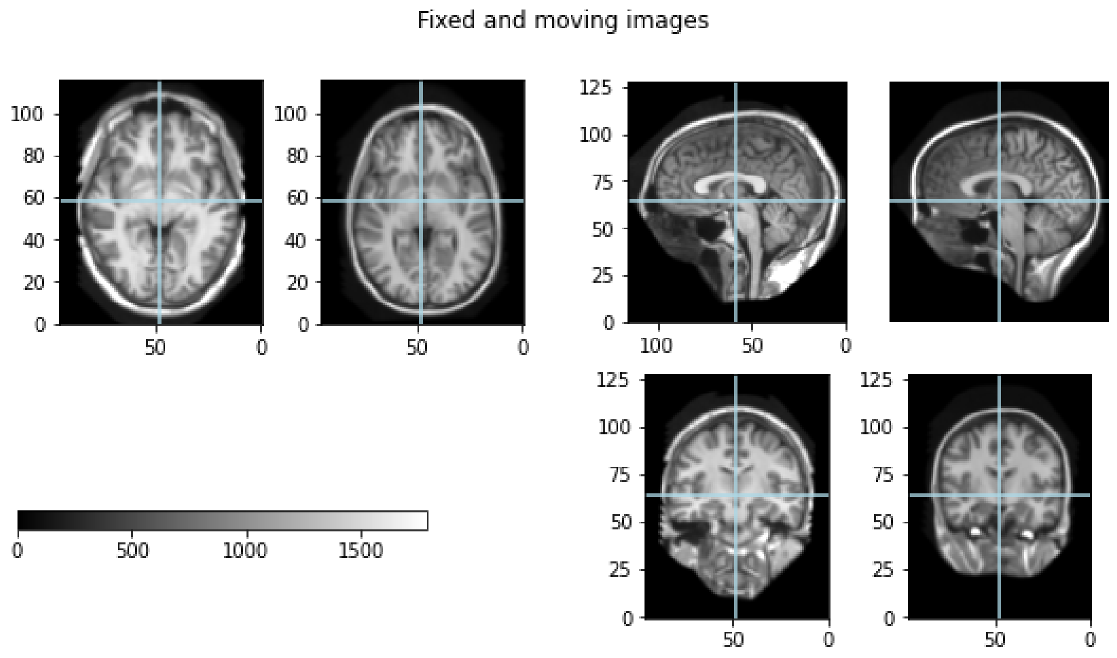
[3]:
```
# Set up CUDA if available
force_cpu = False
if torch.cuda.is_available() and not force_cpu:
  cuda = torch.device('cuda', 0)
  print("Using CUDA device %d named '%s' with %6.2f Gb total memory" %
        (cuda.index, torch.cuda.get_device_name(cuda.index),
         torch.cuda.get_device_properties(cuda.index).total_memory / 2.0**30))
else:
  cuda = torch.device('cpu')
  print("Using CPU")
```

```
Using CUDA device 0 named 'Tesla T4' with  14.76 Gb total memory
```

2.3 New Visualization Code

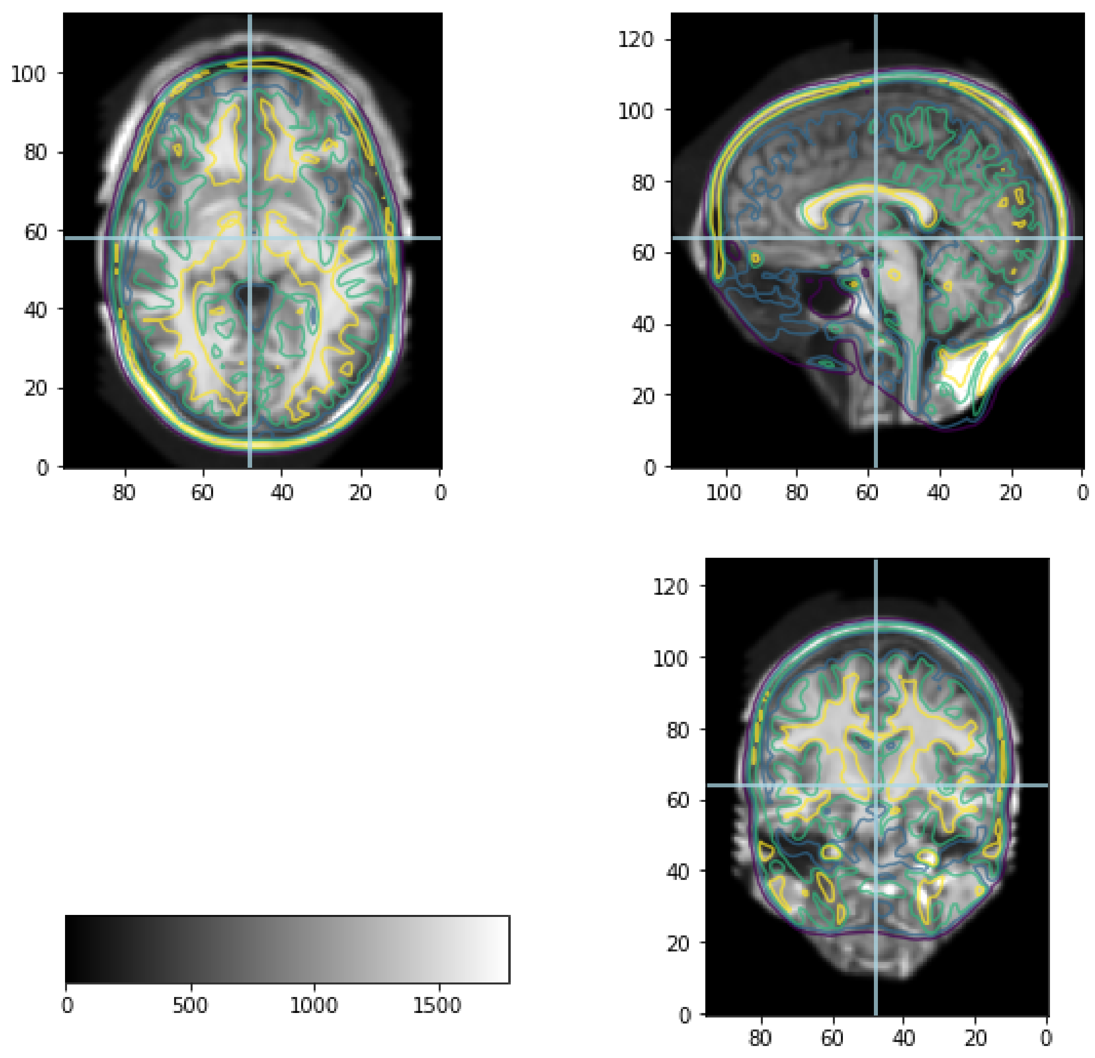
Module `be537hw2` contains two new functions `my_view_multi` and `my_view_ovl` for visualizing multiple images at once. They will be useful for viewing registration results. The code below loads a pair of images from the `testing` directory. The first is the atlas image with ID *1000*, which we will be treating as the *fixed* image. The second is the atlas with ID *1001* that has been aligned to the fixed image using affine registration. We will be treating it as the *moving* image for our deformable registration experiments.

```
[4]: # Read the fixed image
     I_fix, hdr_fix = my_read_pytorch_image_from_nifti(
         os.path.join(data_path,'testing/atlas_2mm_1000_3.nii.gz'),
         dtype=torch.float32, device=cuda)

     # Read the moving image
     I_mov, hdr_mov = my_read_pytorch_image_from_nifti(
         os.path.join(data_path,'testing/reslice_affine_fx_1000_mv_1001.nii.gz'),
         dtype=torch.float32, device=cuda)

     # View the fixed and moving images side by side
     my_view_multi([I_fix, I_mov], hdr_fix, cmap='gray', title='Fixed and moving␣
       ↪images')
```



And here the two images are displayed as overlays

```
[5]: my_view_overlay(I_fix, I_mov, hdr_fix, cmap='gray')
```



2.4 Working with deformable transformations

Before getting started, it is important to understand how spatial transformations are represented in Pytorch.

2.4.1 Order of dimensions

Recall that 3D images are represented in Pytorch as 5D-tensors of dimensions$[N, C, D, H, W]$ where N is the mini-batch size (1 for now), C is the number of channels (1 for grayscale, 3 for an RGB image), and D, H, W are the dimensions of the 3D image.

Spatial transformations are 3D images that contain a coordinate $\phi(x)$ at every voxel x. They can be represented as 3D images with $C = 3$ channels. However, recall from Assignment 1 that the function `grid_sample` expects spatial transformations to be represented as 5D tensors with dimensions $[N, D, H, W, 3]$. Notice that instead of storing the spatial coordinates in the second dimension, the coordinates are stored in the last dimension. In this assignment, we will represent spatial transformations consistent with `grid_sample`.

- You can use function `torch.permute` to go back and forth between $[N, 3, D, H, W]$ and $[N, D, H, W, 3]$ representations of a spatial transformation

2.4.2 Spatial Transformation vs. Displacement Field

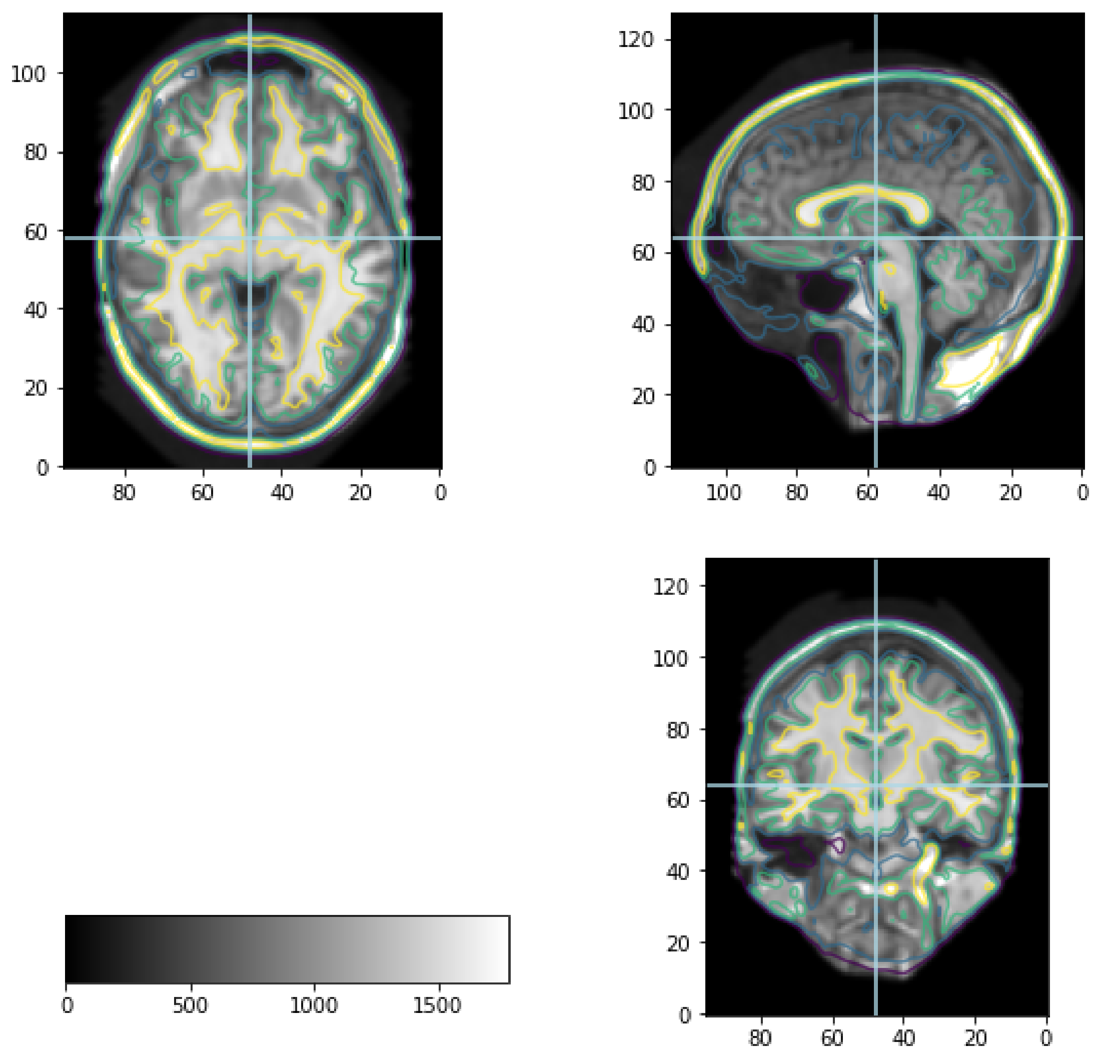
Recall the distinction between a *spatial transformation* and a *displacement field.* A transformation ϕ assigns to each point x in the image domain another point, $\phi(x)$. A displacement field u assigns to each point x a displacement vector $u(x)$. The two are related simply as $\phi(x) = x + u(x)$. Because of this simple relationship, we can think of a displacement field as just another way to represent a spatial transformation. It turns out that in practice representing spatial transformations as displacement fields is easier, particularly when it comes to applying composition operations to spatial transformations. In this assignment, we will represent spatial transformations as displacement fields.

2.4.3 Displacement Field IO

Module `be537hw2` contains the function `my_read_pytorch_warp_from_nifti` that can read a displacement field output by registration tools ANTs and Greedy and return a PyTorch tensor compatible with `grid_sample`.

The next code snippet loads a displacement field (or "warp") computed using Greedy between the fixed and moving images and applies it to the moving image. The transformed (or "resliced") moving image is displayed as an overlay on top of the fixed image. Observe that the contours from the resliced moving image match the structures in the fixed image better than in the previous plot.

[6]:
```python
# Read the spatial transformation (warp for short)
I_warp, hdr_warp = my_read_pytorch_warp_from_nifti(
    os.path.join(data_path,'testing/warp_fx_1000_mv_1001.nii.gz'),
    dtype=torch.float32, device=cuda)

# Create a coordinate grid for using with grid_sample
grid = my_create_pytorch_grid(I_fix.shape).to(cuda)

# Reslice the moving image using the warp
I_res = tfun.grid_sample(I_mov, grid + I_warp, align_corners=False)

# View the fixed and resliced images
my_view_overlay(I_fix, I_res, hdr_fix, cmap='gray')
```



3 Implement Diffeomorphic Demons Registration in PyTorch

In this section, we will implement the diffeomorphic demons image registration algorithm and test it on the imaging data provided. Most code is provided. The portions where you have to write code or answer questions are labeled **TASK 1**, etc.

3.1 Building blocks of the algorithm

Implementing the *Log-Domain Demons* algorithm by Vercauteren et al. (2008) will require the following components:

- **Scaling and squaring**, a technique that converts a smooth and bounded velocity field into a diffeomorphic transformation
- **Image gradient computation**, needed to compute optical flow between fixed and moving

images

- **Gaussian smoothing**, used to regularize velocity fields and displacement fields
- **Optical flow computation**,

3.2 Scaling and squaring

In this part of the assignment, you will implement the scaling and squaring operation. The scaling and squaring algorithm is described in Section 2.1.4 of the NVR chapter. We also discuss it in class on the first day of the deformable registration lecture. Scaling and squaring takes as input a vector field called a **stationary velocity field (SVF)** and generates a spatial transformation that is diffeomorphic.

3.2.1 Theory: Scaling and Squaring using Displacement Fields

The scaling and squaring algorithm is described in Section 2.1.4 of the NVR book chapter in terms of compositions of spatial transformations ψ_k. Since we will be working with displacement fields, we need to rewrite the algorithm in terms of displacement fields.

In a Markdown cell in your notebook, prove the following:

- If ψ_1 and ψ_2 are transformations with corresponding displacement fields v_1 and v_2, then the displacement field w corresponding to the transformation $\psi_1 \circ \psi_2$ (the composition of ψ_1 and ψ_2) is given by
$$w(x) = v_2(x) + v_1(x + v_2(x))$$
- Next, rewrite the scaling and squaring algorithm (at the end of section 2.1.4 of the NVR chapter) using displacement fields v instead of spatial transformations ψ.

TASK 1a (5 pts): Your proof goes here

$\psi_1(x) = x + v_1(x)$

$\psi_2(x) = x + v_2(x)$

Let,

$\psi_3(x) = \psi_1(x) \circ \psi_2(x) = \psi_1(\psi_2(x))$

$\psi_3(x) = \psi_1(x + v_2(x))$

$\therefore \psi_3(x) = x + v_2(x) + v_1(x + v_2(x))$

The displacement field of $\psi_3(x)$ is $\psi_3(x) - x$

$\therefore w(x) = v_2(x) + v_1(x + v_2(x))$

TASK 1b (5 pts): Your algorithm goes here

Let, $u(x)$ be a stationary velocity field.

With n iteratons,

$\psi_0(x) = x + \frac{1}{2^n} u(x)$

$v_0(x) + x = x + \frac{1}{2^n} u(x)$

So initialization:

$v_0(x) = \frac{1}{2^n}u(x)$

The algorithm for k-th iteration:

$v_k = v_{k-1}(x) + v_{k-1}(x + v_{k-1}(x))$ for $k = 1, ..., K$

3.2.2 Implementation of Scaling and Squaring

TASK 2 (10 pts): Complete the code for a function that performs scaling and squaring.

Hints: - The only PyTorch functions needed to implement this function are `grid_sample` and `permute`. - When calling `grid_sample`, the first input will need to be of shape `[1,3,D,H,W]` and the output will be of shape `[1,3,D,H,W]`. Since `grid_sample` expects inputs to have shape `[1,D,H,W,3]`, you will need to use `permute` twice inside of this function.

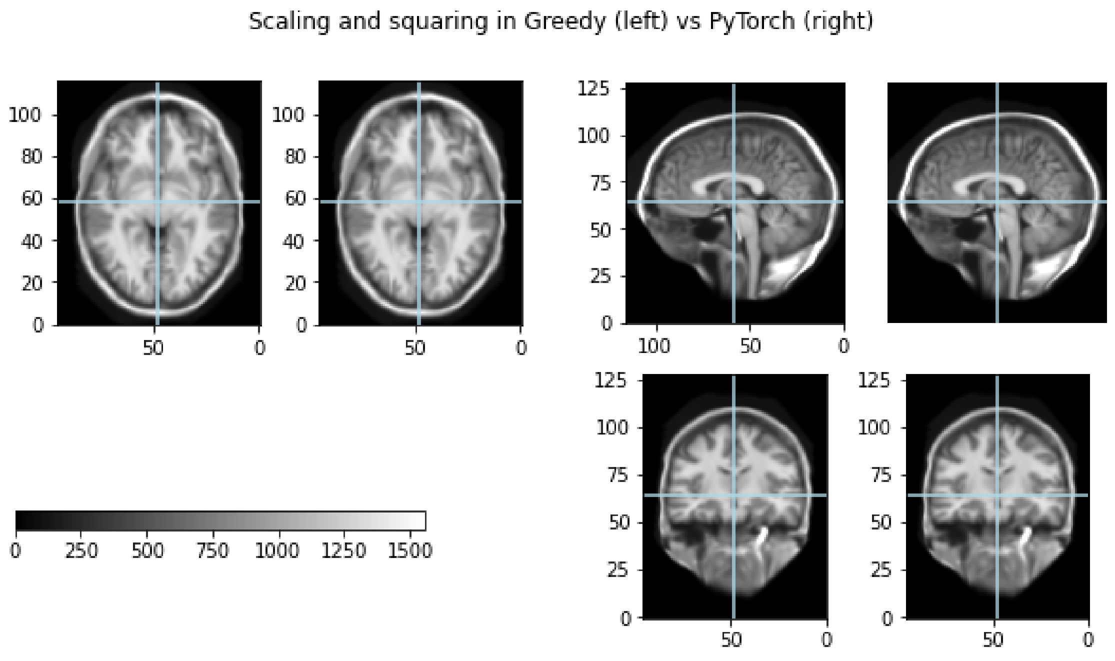
[7]:
```python
def my_scaling_and_squaring(u, grid, n = 6):
    """
    Apply scaling and squaring to a displacement field

    The input and output to this method are displacement fields, v and u,
    respectively. If u is sufficiently smooth and bounded, theoretically,
    the deformation field $\phi(x) = x + u(x)$ will be diffeomorphic

    :param u: Input stationary velocity field, PyTorch tensor of shape␣
  ↪[1,D,H,W,3]
    :param grid: Sampling grid of size [1,D,H,W,3] (created by␣
  ↪my_create_pytorch_grid)
    :param n: Number of iterations of scaling and squaring (default: 6)

    :returns: Output displacement field, v, PyTorch tensor of shape [1,D,H,W,3]
    """
    v0 = u*((0.5)**n) # initialize
    vi = torch.permute(v0, (0,4,1,2,3))

    for _ in range(n):
        curr_x = grid + torch.permute(vi, (0,2,3,4,1))

        # recursion
        vi = vi + tfun.grid_sample(vi, curr_x, align_corners=False)

    v = torch.permute(vi, (0,2,3,4,1))

    return v
```

3.2.3 Test Scaling and Squaring

To confirm that your code is working correctly, we perform the following test:

1. Load a sample stationary velocity field `I_svf` from `testing/svf_fx_1000_mv_1001.nii.gz`
2. Apply scaling and squaring to `I_svf` and use the result to transform the moving image
3. Compare the result to the moving image transformed using `I_warp` above

[8]:
```
# Load the root warp
I_svf, hdr_svf = my_read_pytorch_warp_from_nifti(os.path.
 ↪join(data_path,'testing/svf_fx_1000_mv_1001.nii.gz'), dtype=torch.float32,␣
 ↪device=cuda)

# Apply scaling and squaring to the SVF
I_warp_ss = my_scaling_and_squaring(I_svf, grid, 6)

# Sample the moving image using this transformation
I_res_ss = tfun.grid_sample(I_mov, grid + I_warp_ss, align_corners=False)

# Compare the two images
my_view_multi([I_res, I_res_ss], hdr_fix, cmap='gray',
             title='Scaling and squaring in Greedy (left) vs PyTorch (right)')
```



Here is the quantitative comparison of the two images (you should expect the values to be ~0.1 for mean absolute difference and ~4.0 for max absolute difference

[9]:
```
print('Difference between I_res and I_res_ss')
print('  Mean absolute difference: ' , torch.mean(torch.abs(I_res-I_res_ss)))
print('  Maximum absolute difference: ', torch.max(torch.abs(I_res - I_res_ss)))
```

```
Difference between I_res and I_res_ss
```

```
Mean absolute difference:  tensor(0.1036, device='cuda:0')
Maximum absolute difference:  tensor(4.2585, device='cuda:0')
```

3.3 Gradient computation

We will need to approximate the gradient of the moving image during each step of the deformable registration algorithm. The gradient of an image can be represented as a multi-component image $\nabla I(x) : \mathbb{R}^3 \to \mathbb{R}^3$. At each voxel, this image contains the three partial derivatives of I with respect to the coordinates x_1, x_2, x_3:

$$\nabla I(x) = \left[\frac{\partial I}{\partial x_1}, \frac{\partial I}{\partial x_2}, \frac{\partial I}{\partial x_3}\right]^T$$

It turns out that the images $\frac{\partial I}{\partial x_1}$, $\frac{\partial I}{\partial x_2}$, and $\frac{\partial I}{\partial x_3}$ can each be approximated as discrete convolutions of the image I with simple kernels consisting of three elements.

TASK 3 (10 pts): Your derivation of the convolution kernels for image gradient computation goes here

$$I(y) \simeq I(x) + \sum_{i=1}^{3} \frac{\partial I}{\partial x_i}(x) \cdot (y_i - x_i) + \sum_{i=1}^{3}\sum_{j=1}^{3} \frac{\partial^2 I}{\partial x_i \partial x_j}(x) \cdot \frac{(y_i - x_i)(y_j - x_j)}{2}$$

For $\frac{\partial I}{\partial x_1}$, since we are interested in the $i = 1$ term of x:

$I(x + \delta_1 e_1) = I(x) + \frac{\partial I}{\partial x_1}(x)(x_1 + \delta_1 - x_1) + \frac{\partial^2 I}{\partial x_1^2}(x)\frac{(x_1+\delta_1-x_1)(x_1+\delta_1-x_1)}{2}$

$\therefore I(x + \delta_1 e_1) = I(x) + \delta_1 \frac{\partial I}{\partial x_1}(x) + \frac{\delta_1^2}{2}\frac{\partial^2 I}{\partial x_1^2}(x)$

And,

$I(x - \delta_1 e_1) = I(x) + \frac{\partial I}{\partial x_1}(x)(x_1 - \delta_1 - x_1) + \frac{\partial^2 I}{\partial x_1^2}(x)\frac{(x_1-\delta_1-x_1)(x_1-\delta_1-x_1)}{2}$

$\therefore I(x - \delta_1 e_1) = I(x) - \delta_1 \frac{\partial I}{\partial x_1}(x) + \frac{\delta_1^2}{2}\frac{\partial^2 I}{\partial x_1^2}(x)$

So,

$I(x + \delta_1 e_1) - I(x - \delta_1 e_1) = I(x) + \delta_1 \frac{\partial I}{\partial x_1}(x) + \frac{\delta_1^2}{2}\frac{\partial^2 I}{\partial x_1^2}(x) - I(x) + \delta_1 \frac{\partial I}{\partial x_1}(x) - \frac{\delta_1^2}{2}\frac{\partial^2 I}{\partial x_1^2}(x)$

$I(x + \delta_1 e_1) - I(x - \delta_1 e_1) = 2\delta_1 \frac{\partial I}{\partial x_1}(x)$

$\therefore \frac{\partial I}{\partial x_1}(x) = -\frac{1}{2\delta_1} I(x - \delta_1 e_1) + \frac{1}{2\delta_1} I(x + \delta_1 e_1)$

If we consider $d_1 = -\frac{1}{2\delta_1}$ and $d_2 = \frac{1}{2\delta_1}$, this relationship between the derivative of $I(x)$ at each pixel and the value of $I(x)$ at adjacent pixels can also be expressed as a discrete convolution between $I(x)$ and a size (3,1,1) kernel $[[[d_1]], [[0]], [[d_2]]]$.

Similarly for $\frac{\partial I}{\partial x_2}$, we have:

$\therefore \frac{\partial I}{\partial x_2}(x) = -\frac{1}{2\delta_2} I(x - \delta_2 e_2) + \frac{1}{2\delta_2} I(x + \delta_2 e_2)$

$h_1 = -\frac{1}{2\delta_2}$ and $h_2 = \frac{1}{2\delta_2}$

The relationship between the derivative of $I(x)$ at each pixel and the value of $I(x)$ at adjacent pixels can also be expressed as a discrete convolution between $I(x)$ and a size (1,3,1) kernel $[[[h_1]], [0], [[h_2]]]$

For $\frac{\partial I}{\partial x_3}$, we have:

$\therefore \frac{\partial I}{\partial x_3}(x) = -\frac{1}{2\delta_3} I(x - \delta_3 e_3) + \frac{1}{2\delta_3} I(x + \delta_3 e_3)$

$w_1 = -\frac{1}{2\delta_3}$ and $w_2 = \frac{1}{2\delta_3}$

The relationship between the derivative of $I(x)$ at each pixel and the value of $I(x)$ at adjacent pixels can also be expressed as a discrete convolution between $I(x)$ and a size (1,1,3) kernel $[[[w_1, 0, w_2]]]$

3.3.1 Implementation

TASK 4 (10 pts): Write a function with a signature below that computes the gradient of a 3D image

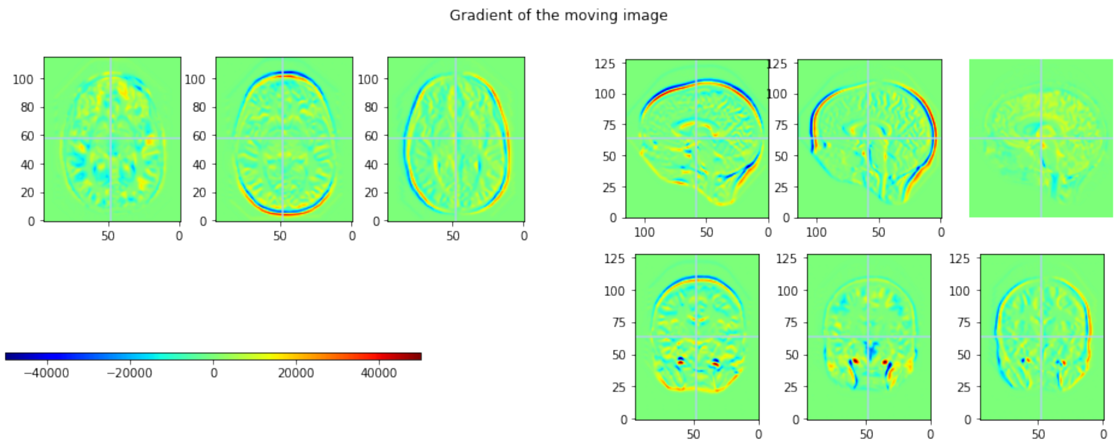
Hints: * The gradient computation can be accomplished using a single call to `torch.nn.functional.conv3d` with `I` as the first parameter and the second parameter a tensor of size $[3, 1, 3, 3, 3]$ that contains your three kernels. * The spacings $\delta_1, \delta_2, \delta_3$ should be with respect to the PyTorch coordinate system, in which the image domain is the cube $[-1, 1] \times [-1, 1] \times [-1, 1]$. So for an image of size \$ N__1 ×N__2 ×N__3\$, we will have $\delta_1 = \frac{2}{N_1}$, etc.

[10]:
```python
# Compute the gradient of a 3D image in PyTorch
def my_image_gradient(I):
    """
    Compute the gradient of an image using central difference approximation

    :param I: input image, represented as a [N,1,D,H,W] tensor
    :returns: gradient of the input image, represented as a [N,3,D,H,W] tensor
    """
    (N, C, D, H, W) = I.shape

    k1 = torch.tensor([-0.25*D, 0, 0.25*D], device=cuda).reshape((3,1,1))
    k2 = torch.tensor([-0.25*H, 0, 0.25*H], device=cuda).reshape((1,3,1))
    k3 = torch.tensor([-0.25*W, 0, 0.25*W], device=cuda).reshape((1,1,3))

    k1 = tfun.pad(k1, (1, 1, 1, 1, 0, 0))
    k2 = tfun.pad(k2, (1, 1, 0, 0, 1, 1))
    k3 = tfun.pad(k3, (0, 0, 1, 1, 1, 1))

    kernel = torch.stack((k3, k2, k1)).unsqueeze(0).reshape((3,1,3,3,3))

    I_grad = tfun.conv3d(I, kernel, padding="same")

    return I_grad
```

3.3.2 Check Gradient Computation

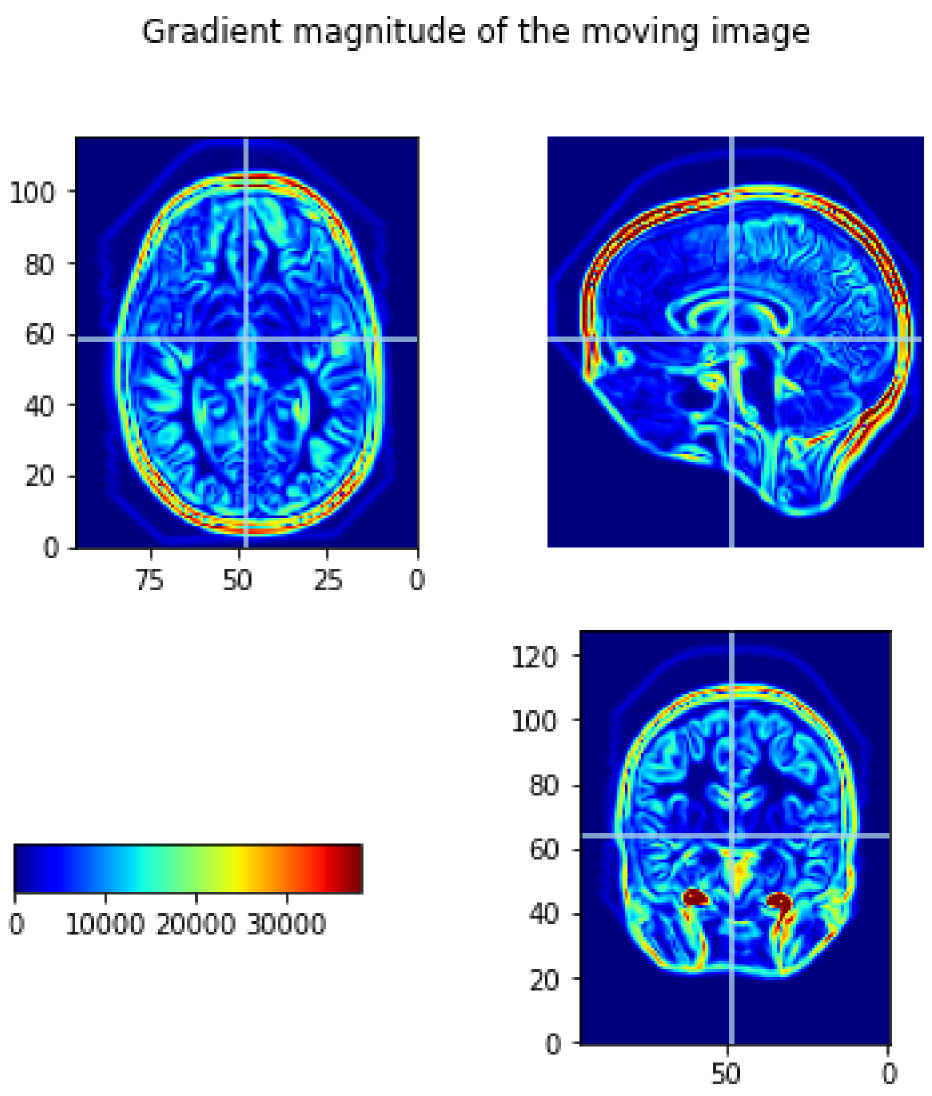
Plot the image gradient using the commands. The expected output is shown in `files/gradient_example.png`.

[11]:
```
I_mov_grad = my_image_gradient(I_mov)
my_view_multi(I_mov_grad, hdr_mov, cmap='jet', crange=[-50000,50000],
              transpose=True, title='Gradient of the moving image',
              figwidth=16);
```



Also plot the gradient magnitude image i.e. an image where at every voxel the intensity is equal to $|\nabla I|$. The gradient magnitude is large at edges in the image.

[12]:
```
my_view_multi(torch.sqrt(I_mov_grad.square().sum(1)), hdr_mov, cmap='jet',
              title='Gradient magnitude of the moving image', figwidth=6)
```



3.4 Gaussian Smoothing

Implementing Gaussian smoothing in Pytorch efficiently requires computing the fast Fourier transform (FFT) of the input image and the Gaussian kernel, multiplication in the frequency domain, and taking the inverse FFT. Getting this to work correctly is a little tricky, so the code for Gaussian smoothing is provided for you in functions `my_pytorch_gaussian_lpf` and `my_pytorch_gaussian_lpf_transform`.

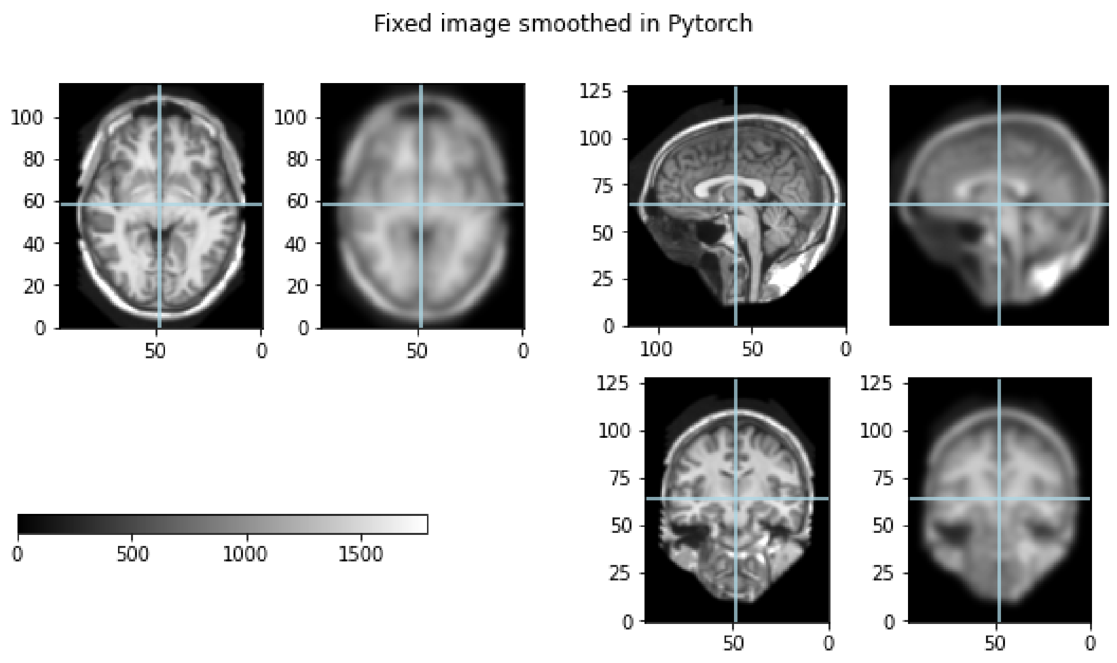
```
[13]: # Function to generate FFT of a Gaussian kernel for use in PyTorch FFT
      def my_gaussian_fft_kernel(I_ref, sigma):
          """
          Generate the FFT of a Gaussian kernel that can be used for FFT-based␣
       ↪Gaussian smoothing
```

```
    :param I_ref: 5D tensor that will be convolved with the kernel, used to␣
 ↪infer the
                  shape, data type and device for the output kernel
    :param sigma: standard deviation of the Gaussian kernel
    :param dtype: data type of the output tensor (default: torch.float32)
    :returns: 5D tensor that can be multiplied by the image RFFT for FFT-based␣
 ↪smoothing
    """

    # Generate a Gaussian kernel, convert to PyTorch
    K = torch.tensor(my_gaussian_3d(sigma),
                     dtype=I_ref.dtype,
                     device=I_ref.device).unsqueeze(0).unsqueeze(0)

    # Normalize the kernel
    K = K / K.sum()

    # Pad the kernel so it is centered in an image of size I_ref.shape
    ab_pad = np.array(I_ref.shape[2:5]) - np.array(K.shape[2:5])
    a_pad = np.floor(ab_pad / 2)
    b_pad = ab_pad - a_pad
    pad_5d = tuple(np.flip(np.vstack((a_pad, b_pad)).T.flatten()).
 ↪astype(int))+(0,0,0,0)
    K_pad = tfun.pad(K,pad_5d)

    # Return the real FFT of the kernel
    return torch.fft.rfftn(K_pad)


# Function to smooth a scalar image with a Gaussian using PyTorch FFT
def my_pytorch_gaussian_lpf(img, sigma=None, kernel=None):
    """
    Apply Gaussian smoothing to a 3D image represented as a PyTorch tensor

    You can pass in the sigma of the Gaussian or a 5D tensor representing the␣
 ↪Gaussian
    kernel (generated using `my_gaussian_fft_kernel`). The latter is faster if␣
 ↪you will
    be making repeated calls to this function.

    :param img: Input image, represented as a 5D tensor
    :param sigma: Standard deviation of the Gaussian kernel. If `kernel`␣
 ↪parameter
        is not supplied, the kernel will be generated by calling␣
 ↪`my_gaussian_fft_kernel`
```

```
    :param kernel: Gaussian kernel generated by `my_gaussian_fft_kernel`.
    """

    # Either sigma or kernel must be provided
    assert sigma is not None or kernel is not None
    if kernel is None:
        kernel = my_gaussian_fft_kernel(img, sigma)

    # Create a Gaussian kernel
    img_fft = torch.fft.rfftn(img)
    return torch.fft.fftshift(torch.fft.irfftn(img_fft * kernel))


# Function to smooth a deformation field with a Gaussian
def my_pytorch_gaussian_lpf_transform(v, sigma=None, kernel=None):
    """
    Apply Gaussian smoothing to a transform represented as a PyTorch tensor

    You can pass in the sigma of the Gaussian or a 5D tensor representing the␣
 ↪Gaussian
    kernel (generated using `my_gaussian_fft_kernel`). The latter is faster if␣
 ↪you will
    be making repeated calls to this function.

    :param img: Input spatial transformation, represented as a [N,D,H,W,3]␣
 ↪tensor
    :param sigma: Standard deviation of the Gaussian kernel. If `kernel`␣
 ↪parameter
        is not supplied, the kernel will be generated by calling␣
 ↪`my_gaussian_fft_kernel`
    :param kernel: Gaussian kernel generated by `my_gaussian_fft_kernel`.
    """

    # Make sure the image is formatted as a transform
    assert v.shape[4]==3

    # Convert the transform into a multi-channel image of shape [1,3,D,H,W]
    v_chan = v.permute(0,4,1,2,3)

    # Either sigma or kernel must be provided
    assert sigma is not None or kernel is not None
    if kernel is None:
        kernel = my_gaussian_fft_kernel(v_chan, sigma)

    # Perform FFT convolution but only along image dimensions
    v_fft = torch.fft.rfftn(v_chan, dim=(-3,-2,-1))
    v_sm_chan = torch.fft.fftshift(
        torch.fft.irfftn(v_fft * kernel, dim=(-3,-2,-1)),
        dim=(-3,-2,-1))

    # Reorder dimensions again
    return v_sm_chan.permute(0,2,3,4,1)
```

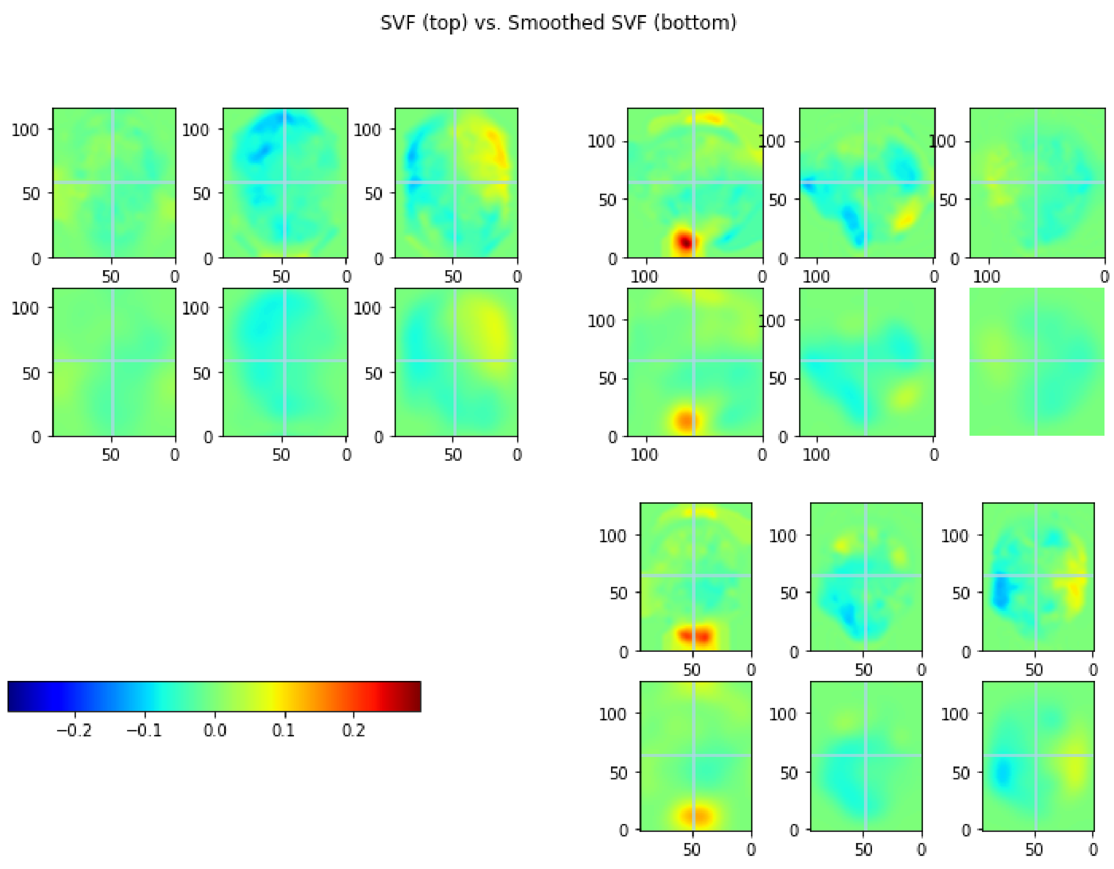
Here is an example of the fixed image smoothed in Pytorch

[14]:
```
I_sm = my_pytorch_gaussian_lpf(I_fix, sigma=2.0)
my_view_multi([I_fix, I_sm], hdr_fix, title="Fixed image smoothed in Pytorch")
```



Here is an example of a deformation field smoothed in PyTorch

[15]:
```
I_svf_sm = my_pytorch_gaussian_lpf_transform(I_svf, sigma=6.0)
my_view_multi(torch.cat((I_svf, I_svf_sm)), hdr_fix, is_warp=True,
              transpose=True, cmap='jet', figwidth=12,
              title='SVF (top) vs. Smoothed SVF (bottom)')
```



3.5 Optical flow computation

Optical flow will be discussed in the Deformable Registration lectures and is covered in Section 3.2.1 of the NVR Chapter. Recall that optical flow infers a velocity field from a pair of images. When these images are frames from a video sequence, the optical flow is an estimate of how fast objects or particles are moving in the video. When doing inter-subject registration, optical flow does not have the same clear physical meaning, but it can be thought of as an infinitecimal displacement that, when applied to the moving image, makes it a little more similar to the fixed image.

3.5.1 Optical Flow Algorithms

Your task is to implement **two versions** of the optical flow approximation.

One of them will be a simple approximation that is obtained if you differentiate the mean square difference metric (MSD, also known as the sum of squared differences metric, or SSD) between $I(x)$ and $J^i(x + v(x))$ with respect to v. This approximation is referred to as `grad_msd` in the code below.

$$v = G_\sigma * (I - J^i)\nabla J^i$$

The other one can be one of the following: * The original *Demons* approximation, given in Algorithm

3 (line 2b) of the NVR Chapter (Section 3.2.2). * The Lukas-Kanade algorithm (advanced) * The Horn and Schunk algorithm (advanced)

3.5.2 Implementation as a Python Class

When working with PyTorch to train neural networks we typically not only write Python functions, but also Python **classes**. A class is an object-oriented programming concept that allows data and functions to be abstracted away into a single entity. We will write a Python class to capture the logic of optical flow computation. The class will have a constructor (`__init__` function) that users call to configure the class (e.g., specify which optical flow method to use or how much smoothing to perform) and a method called `forward` that will perform the actual computation. Once we create an instance of this class, we can call `forward` multiple times with different image inputs.

TASK 5 (20 pts): complete the class definition below by writing the code for the `forward` function

Notes/Hints: * You may find the function `torch.where` useful when implementing the conditional expression in the Demons formula * Other functions that you will need here are `torch.square`, `torch.sum` and `torch.permute` * For the `grad_msd` method, the magnitude of the velocity field retured by `forward` will be very large (on the order of 10^7). This is okay because during registration we will normalize the velocity field so that the maximum displacement is on the order of one voxel.

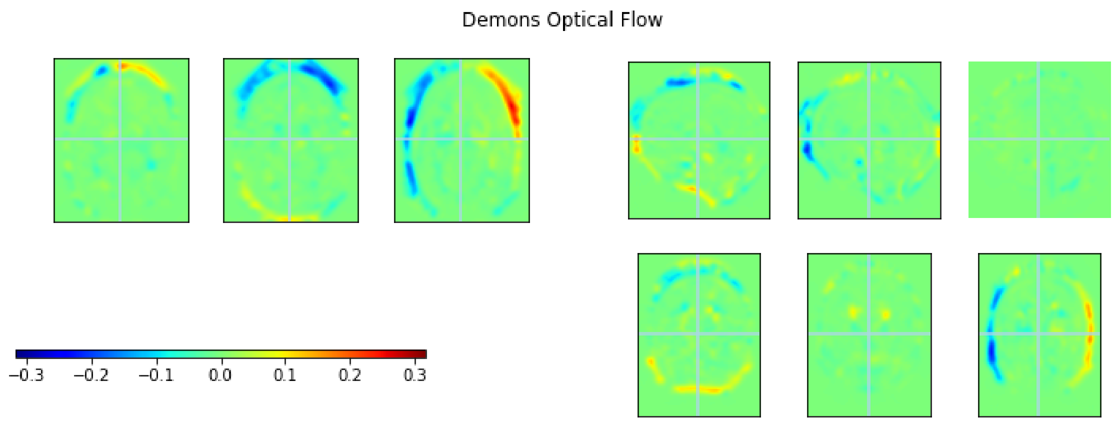
[16]:
```
class MyOpticalFlow:
    """A class that implements multiple optical flow algorithms"""

    def __init__(self, I, method='demons', sigma=2.0, demons_eps=0.001):
        """
        Initialize the optical flow computation

        :param I: Fixed image, represented as [N,1,D,H,W] tensor
        :param method:
            String, specifies which method is used for the computation.␣
↪Available
            options are 'demons' (default) and 'grad_msd'
        :param sigma:
            Standard deviation for Gaussian smoothing in 'demons' and␣
↪'grad_msd'
            methods (passed to `my_gaussian_fft_kernel`)
        :param demons_eps:
            Only used if `method=='demons'`, sets the threshold on the␣
↪denominator
            below which the flow is set to zero. See Algorithm 2 step 2b
        """
        self.method = method
        self.demons_eps = demons_eps
        self.kernel = my_gaussian_fft_kernel(I, sigma)

    def forward(self, I, J):
        """
        Compute optical flow for a given pair of fixed/moving images

        :param I: fixed image, represented as [N,1,D,H,W] tensor
        :param J: moving image, represented as [N,1,D,H,W] tensor
        :returns: optical flow velocity field, represented as a [N,D,H,W,3]␣
↪tensor
        """
        if self.method=="grad_msd":
            vbg = (I-J) * my_image_gradient(J) # v before Gaussian

            vbg = torch.permute(vbg, (0,2,3,4,1))

            v = my_pytorch_gaussian_lpf_transform(vbg, kernel=self.kernel)

            return v

        elif self.method=="demons":
            numerator = (I-J) * my_image_gradient(J)
            denominator = torch.sum(torch.square(my_image_gradient(J)), dim=1)\
                + torch.sum(torch.square(J-I), dim=1)

            vbg = numerator/denominator
            vbg = torch.where(denominator>self.demons_eps, vbg, torch.zeros(vbg.
↪shape, device=cuda))

            vbg = torch.permute(vbg, (0,2,3,4,1))

            v = my_pytorch_gaussian_lpf_transform(vbg, kernel=self.kernel).
↪float()

            return v
```

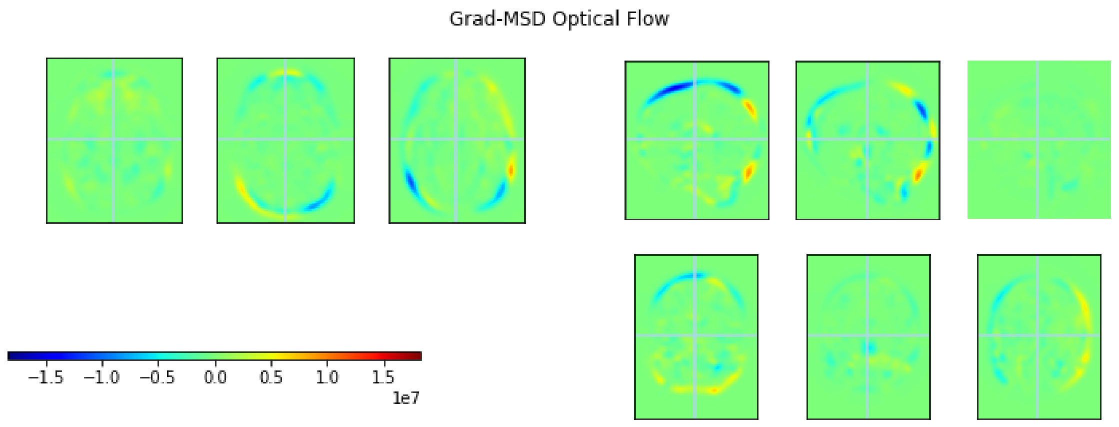
3.5.3 Check Optical Flow Computation

Visualize the optical flow fields that you obtain for the two methods using `my_view_multi` using code below. The output should look like `files/oflow_example.png`.

[17]:
```
of_demons = MyOpticalFlow(I_fix, method='demons', sigma=2.0)
v_demons = of_demons.forward(I_fix, I_mov)

of_gradmsd = MyOpticalFlow(I_fix, method='grad_msd', sigma=2.0)
v_gradmsd = of_gradmsd.forward(I_fix, I_mov)

my_view_multi(v_demons, hdr_fix, cmap='jet', is_warp=True, transpose=True,
              figwidth=12, axis_visible=False, title='Demons Optical Flow')
```

```
my_view_multi(v_gradmsd, hdr_fix, cmap='jet', is_warp=True, transpose=True,
              figwidth=12, axis_visible=False, title='Grad-MSD Optical Flow')
```





3.6 Log-Demons Algorithm

3.6.1 Full-Resolution Method

Finally, we are ready to implement the log-Demons algorithm, described in Algorithm 4 in the NVR chapter (Section 3.2.3). Again, we will implement it as a Python class `MyLogDemonsAlgorithm`

The method `__init__` will be responsible for storing the variables needed by the class (i.e., lines of code line `self.I = I`) and initializing the sampling grid and other objects that are repeatedly used during the algorithm.

The method `step` is the portion of the algorithm that is iterated. At every call to `step`, the spatial transformation ϕ^i and the stationary velocity field u^i are updated. Note that consisently with how we implemented scaling and squaring, we use the displacement field w^i to represent the spatial transformation ϕ^i in our code. The two are related by $\phi^i(x) = x + w^i(x)$.

The code for `MyLogDemonsAlgorithm` is provided to you, but it requires that gradient computation and scaling and squaring algorithms be implemented correctly.

```
[18]: class MyLogDemonsAlgorithm:
          """A class that performs log-Demons registration between two images"""

          def __init__(self, I, J, of, tau=0.5, eps_prime = 0.5):
              """
              Initialize the Log Demons algorithm

              :param I: Fixed image, represented as an [N,1,D,H,W] tensor
              :param J: Moving image, represented as an [N,1,D,H,W] tensor
              :param of:
                  Instance of class MyOpticalFlow used to perform optical flow␣
      ↪computations
              :param tau:
                  Standard deviation for 'diffusion-like' regularization (Algorithm␣
      ↪4, Line 2c)
              :param eps_prime:
                  Normalization factor. The velocity field computed by optical flow␣
      ↪will
                  be scaled so that the longest velocity vector is equal to eps_prime
                  voxel widths. See second bullet point after Algorithm 3.
              """
              self.I = I
              self.J = J
              self.of = of
              self.kernel_tau = my_gaussian_fft_kernel(I, tau)
              self.eps_prime = eps_prime
              self.grid = my_create_pytorch_grid(I.shape, dtype=I.dtype, device=I.
      ↪device)

          def step(self, w_i, u_i):
              """
              Execute one iteration of the Log Demons algorithm (Algorithm 4, Line 2)

              :param w_i:
                  Displacement field corresponding to the current estimate of the␣
      ↪spatial
                  transformation. Transformation $\phi_i$ is given by $\phi_i(x) = x␣
      ↪+ w_i(x)$.
                  Represented as an [N,1,D,H,W] tensor.
              :param u_i:
                  Current estimate of the stationary velocity field $u_i$
                  Represented as an [N,1,D,H,W] tensor.

              :returns:
```

```
            Tuple (Ji, rms, w_i1, u_i1) where

            - `Ji` is the moving image transformed by the spatial␣
↪transformation $\phi_i$
            - `msd` is the mean square intensity difference between I and J^i
            - `w_i1` is the updated displacement field corresponding to␣
↪$\phi_{i+1}$
            - `u_i1` is the updated stationary velocity field
        """

        # Resample the moving image using current displacement field w_i
        J_i = tfun.grid_sample(self.J, self.grid + w_i, align_corners=False)

        # Compute the RMS metric between fixed image I and resampled moving␣
↪image
        msd = torch.mean((self.I - J_i).square())

        # Compute the optical flow field between fixed image I and resampled␣
↪moving image
        v_i = self.of.forward(self.I, J_i)

        # Scale the optical flow field so the longest vector does not exceed␣
↪eps_prime
        # (eps_prime is specified in voxel units, e.g., 0.5 voxel).
        scale = self.eps_prime * np.min(2.0 / np.array(self.I.shape[2:5])) /␣
↪torch.max(v_i)

        # Compute the updated SVF using the approximation of the Lie bracket␣
↪term in Algorithm 4 line 2c
        u_i1 = my_pytorch_gaussian_lpf_transform(u_i + scale * v_i, kernel=self.
↪kernel_tau)

        # Perform scaling and squaring to get a new transformation
        w_i1 = my_scaling_and_squaring(u_i1, self.grid)

        # Return the updated w and u
        return J_i, msd, w_i1, u_i1
```

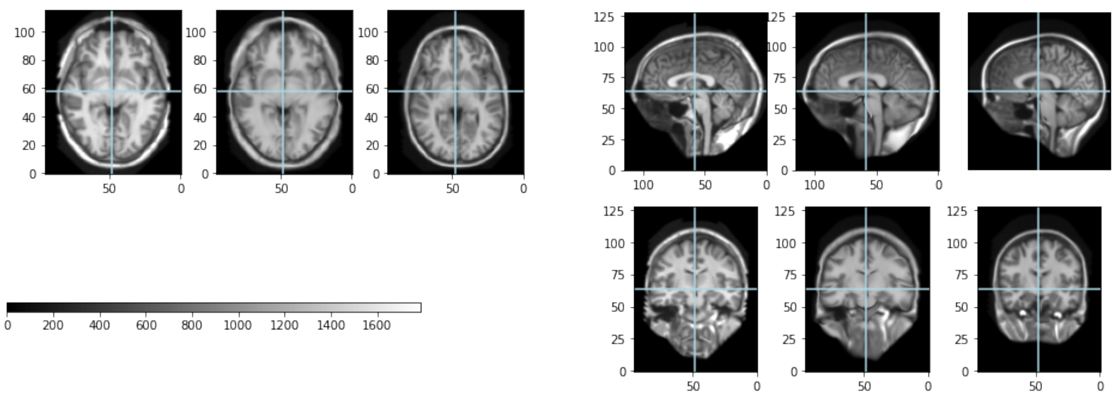
The code below should be used to test out your algorithm. It should produce output similar to what is shown in `files/reg_example.png`. The objective function should start at ~96000 and go down to ~13000. It might take a minute or two on a CPU to complete the 200 iterations.

[19]:
```
log_demons = MyLogDemonsAlgorithm(I_fix, I_mov, of_gradmsd)
w_i = torch.zeros_like(I_fix).repeat((1,3,1,1,1)).permute(0,2,3,4,1)
u_i = w_i.clone()
for k in range(200):
    J_i, msd, w_i, u_i = log_demons.step(w_i, u_i)
    if k % 20 == 0:
        print('Iter %03d, MSD = %8.4f' % (k, msd))
```

```
Iter 000, MSD = 96846.3906
Iter 020, MSD = 52207.1992
Iter 040, MSD = 31559.2520
Iter 060, MSD = 22472.2324
Iter 080, MSD = 18530.4863
Iter 100, MSD = 15748.2754
Iter 120, MSD = 14349.0088
Iter 140, MSD = 14096.1768
Iter 160, MSD = 13312.1777
Iter 180, MSD = 13139.9023
```

[20]:
```
my_view_multi([I_fix, J_i, I_mov], hdr_fix, figwidth=16,
              title='Fixed, registered, moving images')
```

Fixed, registered, moving images



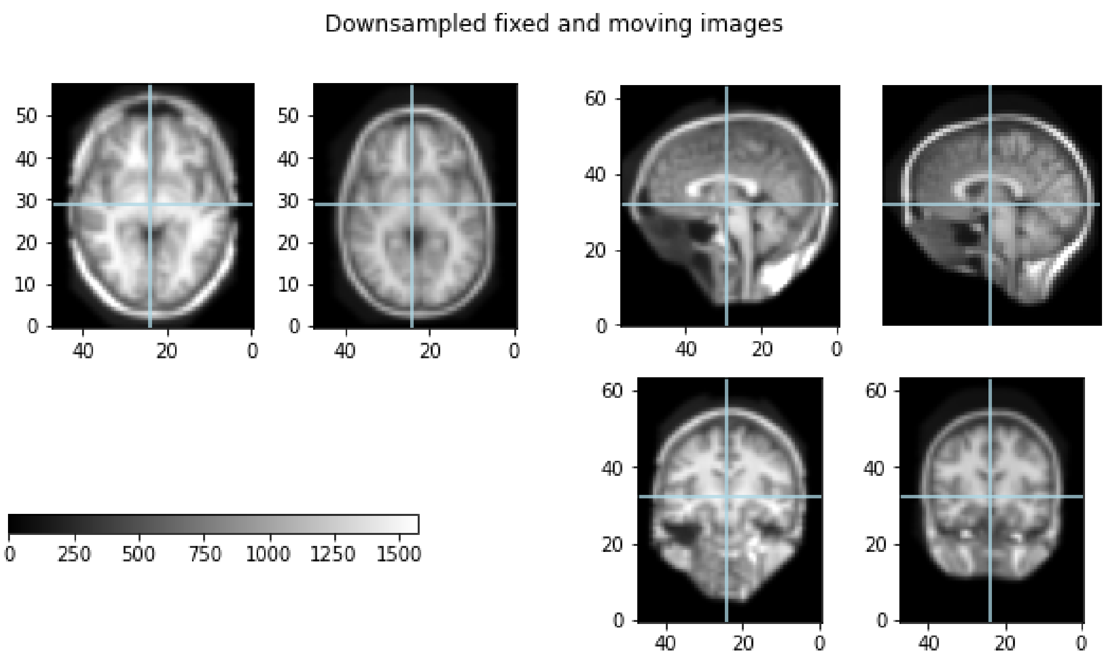
3.6.2 Multi-Resolution Log Demons

A major difference between the registration method above and tools like ANTS is that the latter use multi-resolution schemes for significantly faster optimization. For example, when registering images of 256x256x256 voxels, one would typically subsample to 64x64x64 and run registration at that resolution; then use the result to initialize registration at 128x128x128 resolution, and run that for some number of iterations and use the result to initialize full-resolution registration. This has a dramatic effect on registration quality. This section provides code for multi-resolution Log Demons registration. There is no code for you to write in this section.

First, we coded a function to downsample an image represented as a PyTorch tensor (`my_image_downsample` in `be537_hw2`).

[21]:
```
I_fix_ds = my_image_downsample(I_fix, 2.0)
I_mov_ds = my_image_downsample(I_mov, 2.0)
```

```
hdr_fix_ds = my_adjust_nifti_header_for_resample(hdr_fix, I_fix_ds.shape[2:5])
my_view_multi([I_fix_ds,I_mov_ds], hdr_fix_ds,
              title="Downsampled fixed and moving images")
```



This function runs the log-Demons algorithm in a multi-resolution setting

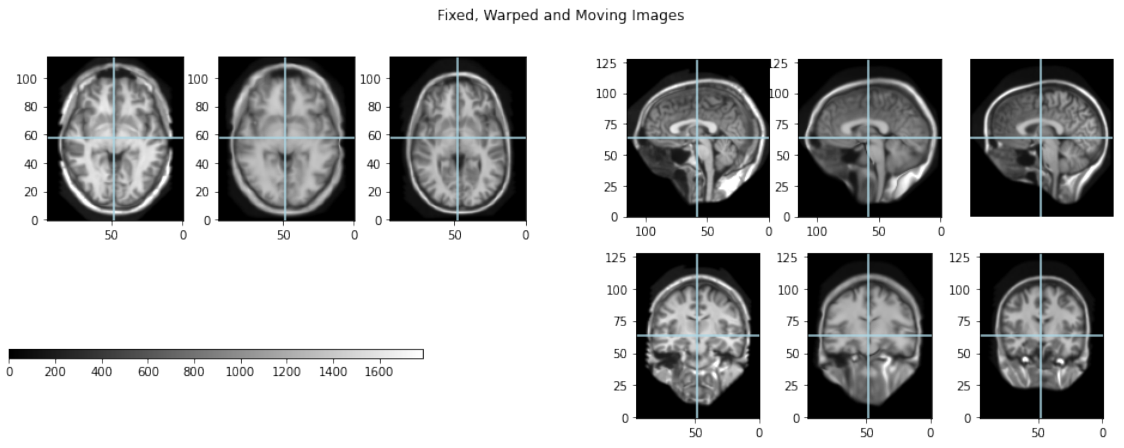
[22]:
```
def my_log_demons_multires(I_fix, I_mov, iter_vec,
                           sigma=2.0, tau=0.5, eps_prime=0.5,
                           of_method='grad_msd', of_param = {},
                           silent=False):
    """
    Multi-resolution log Demons algorithm

    :param I_fix: Fixed image, represented as an [N,1,D,H,W] tensor
    :param I_mov: Moving image, represented as an [N,1,D,H,W] tensor
    :param iter_vec:
        Number of iterations at each resolution level, e.g., [100,40] will
        do 100 iterations at 2x downsampling, 40 at full resolution
    :param sigma: See MyOpticalFlow
    :param tau: See MyLogDemonsAlgorithm
    :param eps_prime: See MyLogDemonsAlgorithm
    :param of_method: See MyOpticalFlow
    :param of_param: See MyOpticalFlow

    :returns:
        Tuple (I_res, rms, w, u) where
        - `I_res` is the moving image transformed by the spatial transformation␣
↪$\phi$
        - `rms` is the root mean square intensity difference between I and I_res
        - `w` is the displacement field corresponding to $\phi$
        - `u` is the stationary velocity field corresponding to $\phi$
    """

    # Current u
    u = None

    # Iterate over resolution levels
    for level, n_iter in enumerate(iter_vec):

        # Downsample images by current factor
        factor = 2 ** (len(iter_vec) - level - 1)
        I_fix_ds = my_image_downsample(I_fix, factor)
        I_mov_ds = my_image_downsample(I_mov, factor)
        dim_ds = I_fix_ds.shape[2:5]

        # Resample u to the current iteration
        if u is not None:
            u = tfun.interpolate(u.permute(0,4,1,2,3),
                                 tuple(dim_ds)).permute(0,2,3,4,1)
        else:
            u = torch.zeros_like(I_fix_ds).permute(0,2,3,4,1).repeat(1,1,1,1,3)

        # Create optical flow for this iteration
        of = MyOpticalFlow(I_fix_ds, sigma=sigma, method=of_method, **of_param)

        # Create the algorithm for this iteration
        log_demons = MyLogDemonsAlgorithm(I_fix_ds, I_mov_ds, of, tau=tau,
                                          eps_prime=eps_prime)

        # Compute the displacement field w
        w = my_scaling_and_squaring(u, log_demons.grid)

        # Perform iterations
        for k in range(n_iter):
            I_res, msd, w, u = log_demons.step(w, u)
            if k % 10 == 0:
                if not silent:
                    print('Level %01d  Iter %03d  RMS %8.4f' % (level, k,
                                                                msd.item()))

    return I_res, msd, w, u
```

Run a multi-resolution registration. This registration should run faster than the one above and get the final RMS that is under 13, 000.

```
[23]: I_res, rms, phi, u = my_log_demons_multires(I_fix, I_mov, [80,80,40],
                                                   of_method='grad_msd', sigma=1.0)
```

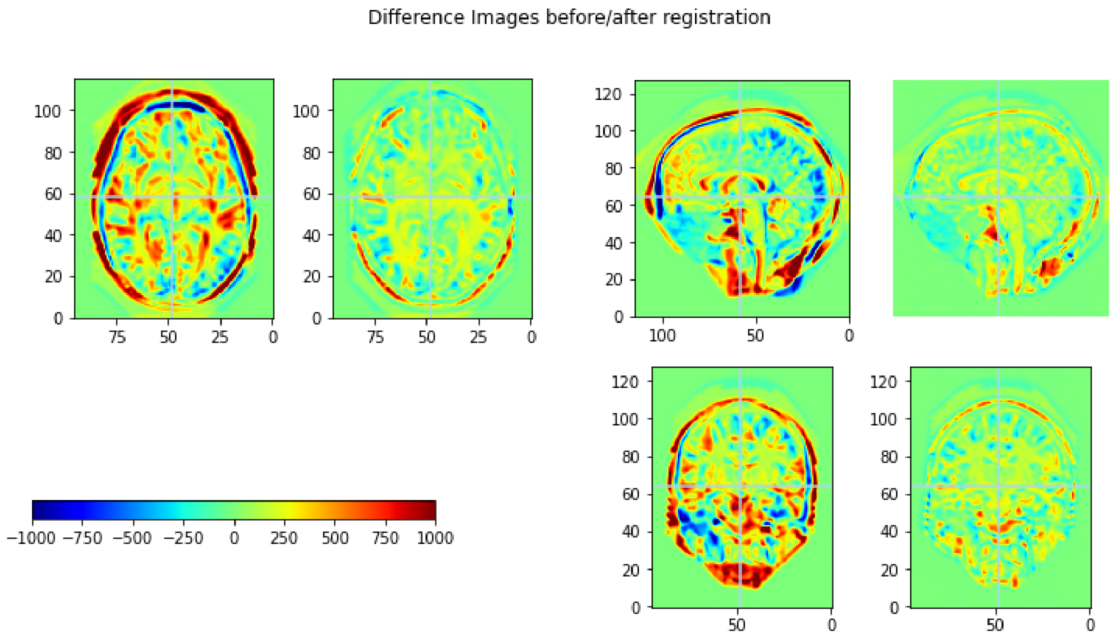
```
Level 0  Iter 000  RMS 38671.7969
Level 0  Iter 010  RMS 25605.3965
Level 0  Iter 020  RMS 23156.2090
Level 0  Iter 030  RMS 20229.6172
Level 0  Iter 040  RMS 19559.4141
Level 0  Iter 050  RMS 19037.0918
Level 0  Iter 060  RMS 18722.2539
Level 0  Iter 070  RMS 18559.9395
Level 1  Iter 000  RMS 36157.4648
Level 1  Iter 010  RMS 18300.3711
Level 1  Iter 020  RMS 13684.6992
Level 1  Iter 030  RMS 11146.5049
Level 1  Iter 040  RMS 10658.3428
Level 1  Iter 050  RMS 10496.4746
Level 1  Iter 060  RMS 10224.4717
Level 1  Iter 070  RMS 10139.5508
Level 2  Iter 000  RMS 16456.1543
Level 2  Iter 010  RMS 13733.7529
Level 2  Iter 020  RMS 13152.0605
Level 2  Iter 030  RMS 12773.3506
```

```
[24]: my_view_multi([I_fix,I_res,I_mov], hdr_fix, figwidth=16,
                    title='Fixed, Warped and Moving Images')
```



Plot difference images before and after registration

```
[25]: my_view_multi([I_fix-I_mov,I_fix-I_res], hdr_fix, cmap='jet',
                    crange=[-1000,1000], figwidth=12,
                    title='Difference Images before/after registration')
```



3.7 Compare Affine and Deformable Registration for 10 Subject Pairs

In this final sub-task of the base component of the assignment, you will perform deformable registration between 10 randomly selected pairs of subjects in the dataset and evaluate how well these registrations align anatomical labels between images compared to affine registration. To make things simpler, we computed the affine transformations between all image pairs ahead of time.

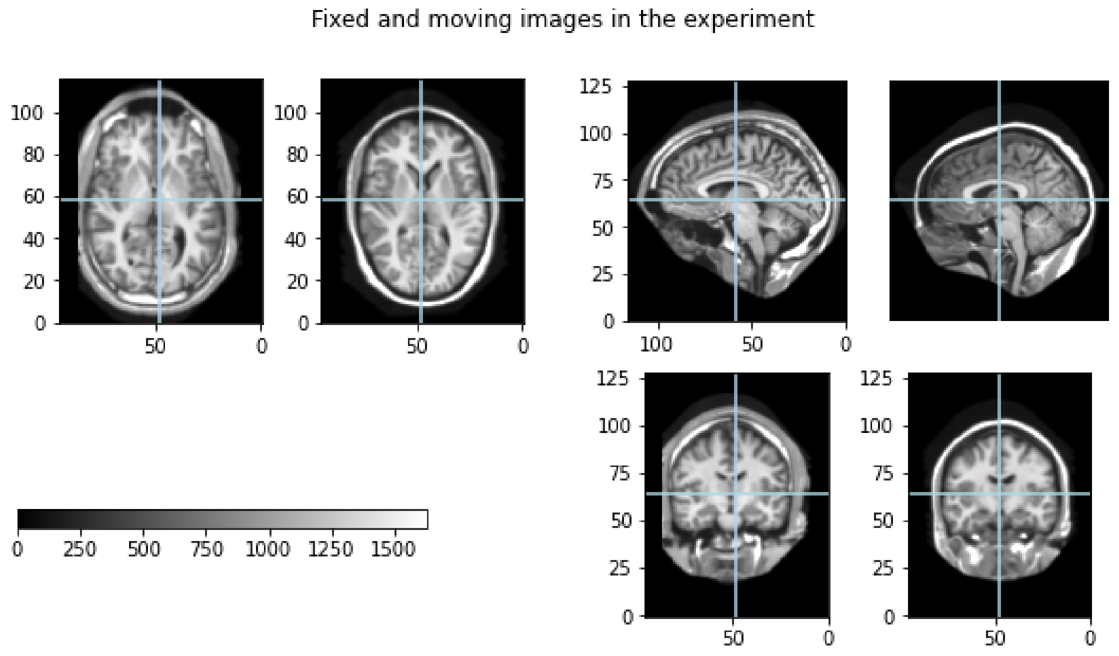
3.7.1 Class to Load Registration Experiments

We also provide some helper code to save you some tedious work. The class `RegistrationExperiment` provided in the `be537hw2.py` lets you easily load and access all the data associated with each registration experiment. This class provides access to the following variables:

- `exp.I_fix`: Fixed image, represented as a PyTorch tensor
- `exp.S_fix`: Fixed image segmentation
- `exp.I_mov`: Moving image
- `exp.S_mov`: Moving image segmentation
- `exp.hdr_fix`: Fixed image header (for calling `my_view`)
- `exp.grid`: Fixed image coordinate grid (for calling `grid_sample`)
- `exp.A`, `exp.b`: Affine transform used to map the moving image into fixed image space.

The code below loads a registration experiment with subject `1002` as fixed and `1014` as moving.

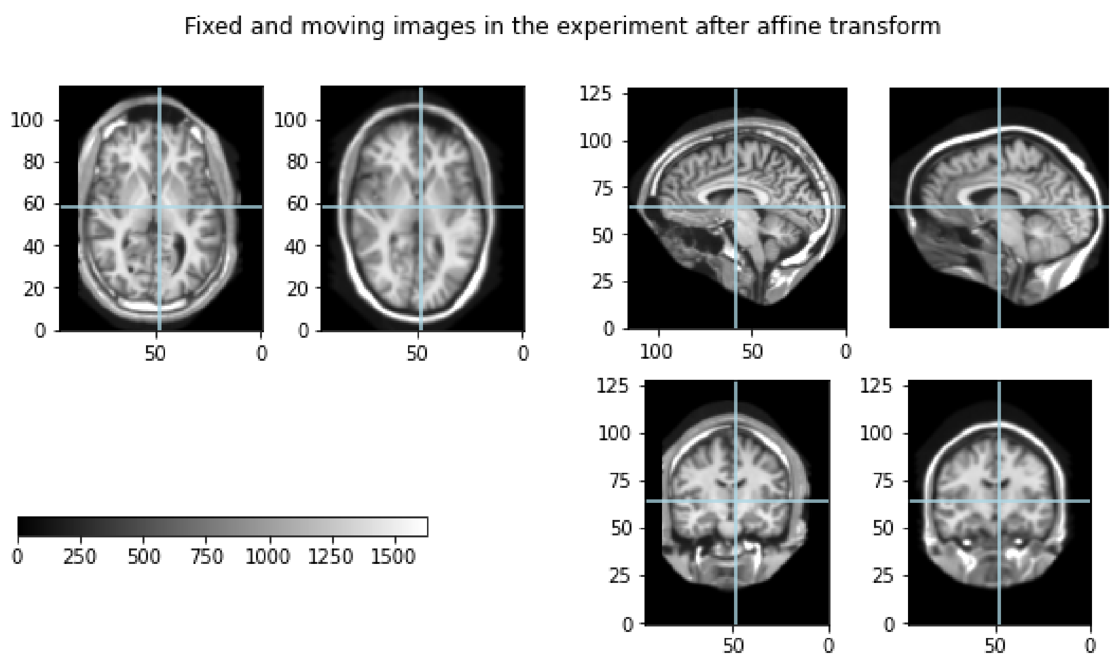
[26]:
```
exp = RegistrationExperiment('1002','1014', data_path, dtype=torch.float32,
                              device=cuda)
my_view_multi([exp.I_fix, exp.I_mov], exp.hdr_fix,
               title='Fixed and moving images in the experiment')
```



3.7.2 Function to apply affine and deformable transformations jointly

Another useful function we provided is `my_transform_affine_and_warp`. It allows you to apply the a combination of affine and deformable transformations to a moving image. If the deformable transformation is not specified (as in the line below), only the affine transformation is applied.

[27]:
```
I_aff = my_transform_affine_and_warp(exp.I_mov, exp.grid, exp.A, exp.b)
my_view_multi([exp.I_fix, I_aff], exp.hdr_fix,
               title='Fixed and moving images in the experiment after affine␣
 ↪transform')
```

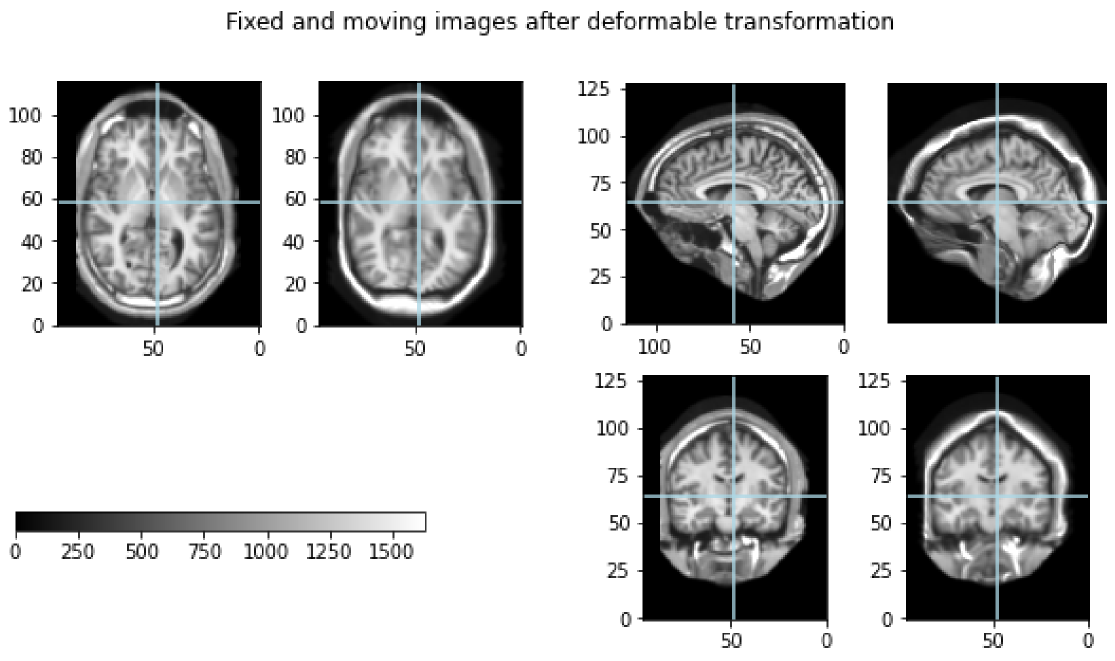

Fixed and moving images in the experiment after affine transform
100
80
60
40
20
0
50
0
125
75
25
0
250
500
750
1000
1250
1500

Now that we have the fixed image `exp.I_fix` and the affine-transformed moving image `I_aff`, we can apply your log-Demons algorithm to perform deformable registration.

[28]:
```python
_, _, w_opt, _ = my_log_demons_multires(exp.I_fix, I_aff, [80,80,40],
                                        of_method='grad_msd', sigma=1.0,
                                        silent=True)
```

To apply the combination of deformable and affine transformations to the moving image, we use the command below. This helps avoid double interpolation and associated aliasing, that would happen if we first applied the affine transform and then separately applied the deformable transformation

[29]:
```python
I_def = my_transform_affine_and_warp(exp.I_mov, exp.grid, exp.A, exp.b, w_opt)
my_view_multi([exp.I_fix, I_def], exp.hdr_fix,
              title='Fixed and moving images after deformable transformation')
```

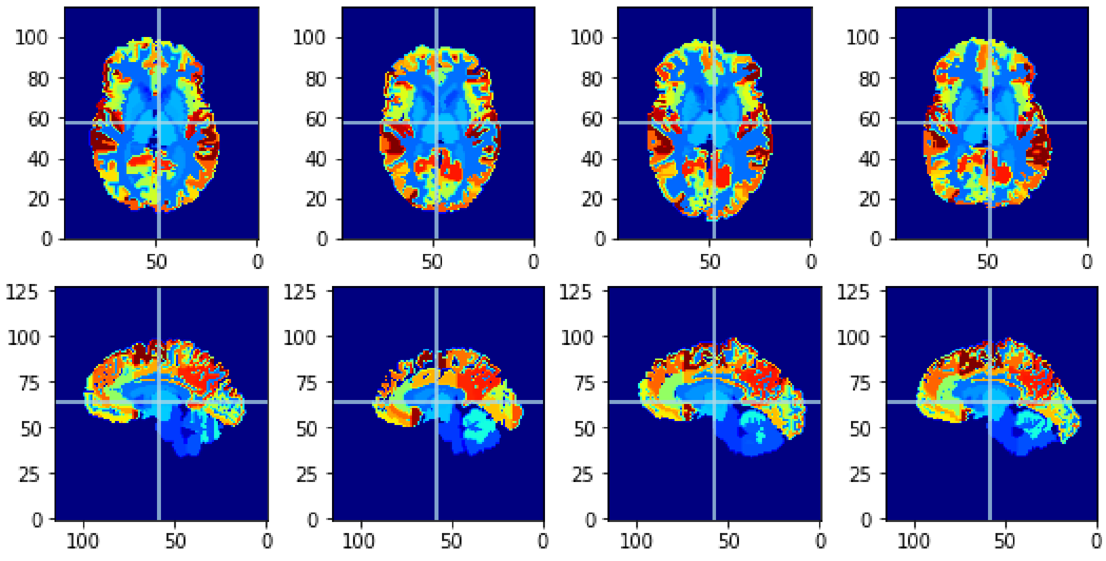


3.7.3 Applying transformations to segmentation images

In order to apply your affine and deformable transformations to the anatomical segmentation of the moving image, we can use the following code, which uses nearest neighbor interpolation to avoid interpolating linearly between adjacent anatomical labels.

```
[30]: S_aff = my_transform_affine_and_warp(exp.S_mov, exp.grid, exp.A, exp.b,
                                          mode='nearest')
      S_def = my_transform_affine_and_warp(exp.S_mov, exp.grid, exp.A, exp.b, w_opt,
                                          mode='nearest')
      my_view_multi([exp.S_fix, exp.S_mov, S_aff, S_def], exp.hdr_fix,
                    cmap='jet', layout='A;S',
                    title='Segmentations: fixed, moving, moving after affine, moving␣
        ↪after deformable.')
```

Segmentations: fixed, moving, moving after affine, moving after deformable.



3.7.4 Measuring alignment between segmentations

Finally, to measure how well the registration aligned anatomical structures between the fixed and moving images, use the function `my_generalized_dice` to measure the generalized Dice similarity coefficient between the segmentations. GDSC is a measure between 0 and 1 with 0 indicating no overlap and 1 indicating perfect overlap. The code below computes GDSC at different stages of the registration.

[31]:
```
print('Alignment between segmentations images %s and %s' % (exp.fixed_id, exp.
 ↪moving_id))
print('  GDSC before registration: ', my_generalized_dice(exp.S_fix, exp.S_mov))
print('  GDSC after affine registration', my_generalized_dice(exp.S_fix, S_aff))
print('  GDSC after deformable registration', my_generalized_dice(exp.S_fix,␣
 ↪S_def))
```

```
Alignment between segmentations images 1002 and 1014
  GDSC before registration:  tensor(0.1664, device='cuda:0')
  GDSC after affine registration tensor(0.3797, device='cuda:0')
  GDSC after deformable registration tensor(0.4727, device='cuda:0')
```

3.7.5 Evaluate GDSC for 10 subject pairs

[32]:
```
def evaluate_gdsc(id_fix, id_mov, grad_msd=False, demons=False):
    """_summary_

    :param id_fix: ID of the fixed image
    :param id_mov: ID of the moving image
    :param grad_msd: whether to do grad_msd deformable registration, defaults␣
↪to False
    :param demons: whether to do demons deformable registration, defaults to␣
↪False

    :returns:
        Tuple (gdsc_all, segmentations) where

        - `gdsc_all` is the list of GDSC for none, affine, deformable␣
↪registration
        - `segmentations` is the list of affine and deformable registered␣
↪segmentations
    """

    gdsc_all = []
    segmentations = []

    exp = RegistrationExperiment(str(id_fix),str(id_mov), data_path,
                                 dtype=torch.float32, device=cuda)

    gdsc_no_reg = my_generalized_dice(exp.S_fix, exp.S_mov).cpu().numpy()
    gdsc_all.append(gdsc_no_reg)

    print('  GDSC before registration: ', gdsc_no_reg)

    # get affine registered image
    I_aff = my_transform_affine_and_warp(exp.I_mov, exp.grid, exp.A, exp.b)
    S_aff = my_transform_affine_and_warp(exp.S_mov, exp.grid, exp.A, exp.b,
                                         mode='nearest')
    gdsc_affine = my_generalized_dice(exp.S_fix, S_aff).cpu().numpy()

    segmentations.append(S_aff)
    gdsc_all.append(gdsc_affine)

    print('  GDSC after affine registration', gdsc_affine)

    if grad_msd:
        # grad msd deformable registration
        _, _, w_gradmsd, _ = my_log_demons_multires(exp.I_fix, I_aff,
                                                    [80,80,40],
                                                    of_method='grad_msd',
                                                    sigma=1.0, silent=True)
        S_gradmsd = my_transform_affine_and_warp(exp.S_mov, exp.grid, exp.A,
                                                 exp.b, w_gradmsd,
                                                 mode='nearest')
        gdsc_def_gradmsd = my_generalized_dice(exp.S_fix, S_gradmsd).cpu().
↪numpy()
```

```
        segmentations.append(S_gradmsd)
        gdsc_all.append(gdsc_def_gradmsd)

        print('  GDSC after grad_msd deformable registration', gdsc_def_gradmsd)

    if demons:
        # demons deformable registration
        _, _, w_demons, _ = my_log_demons_multires(exp.I_fix, I_aff, [80,80,40],
                                                   of_method='demons',
                                                   sigma=1.0, silent=True)
        S_demons = my_transform_affine_and_warp(exp.S_mov, exp.grid, exp.A,
                                                exp.b, w_demons, mode='nearest')
        gdsc_def_demons = my_generalized_dice(exp.S_fix, S_demons).cpu().numpy()
        segmentations.append(S_demons)
        gdsc_all.append(gdsc_def_demons)

        print('  GDSC after demons deformable registration', gdsc_def_demons)

    return gdsc_all, segmentations
```

[33]:
```
def print_mean_std(my_list):
    my_array = np.array(my_list)
    mean = np.mean(my_array)
    std = np.std(my_array)

    return mean, std
```

TASK 6 (10 points): Pick 10 random pairs of fixed/moving subjects, perform registration, and record GDSC before registration, after affine registration, after deformable registration using Demons method, and after deformable registration using one of the other optical flow methods you implemented. Report summary statistics and conclude which technique performed best.

Method	Average GDSC
No registration	0.2461 ± 0.0743
Affine only	0.3769 ± 0.0328
Affine + Grad MSD	0.4940 ± 0.0224
Affine + Demons	0.4463 ± 0.0312

[34]:
```
list_atlas = [1000, 1001, 1002, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1013,
              1014, 1015, 1017, 1036]

gdsc_no_reg = []
gdsc_affine = []
gdsc_gradmsd = []
gdsc_demons = []
```

```python
for i in range(10):
    # pick two IDs randomly from list above
    id_fix, id_mov = np.random.choice(list_atlas, 2, replace=False)
    print(id_fix, id_mov)

    gdsc_all, _ = evaluate_gdsc(id_fix, id_mov, grad_msd=True, demons=True)

    gdsc_no_reg.append(gdsc_all[0])
    gdsc_affine.append(gdsc_all[1])
    gdsc_gradmsd.append(gdsc_all[2])
    gdsc_demons.append(gdsc_all[3])
```

```
1013 1036
  GDSC before registration:  0.22458068
  GDSC after affine registration 0.41836646
  GDSC after grad_msd deformable registration 0.5040101
  GDSC after demons deformable registration 0.44491392
1014 1036
  GDSC before registration:  0.13437201
  GDSC after affine registration 0.32930702
  GDSC after grad_msd deformable registration 0.47471428
  GDSC after demons deformable registration 0.4444901
1000 1006
  GDSC before registration:  0.16061912
  GDSC after affine registration 0.34549358
  GDSC after grad_msd deformable registration 0.4864284
  GDSC after demons deformable registration 0.4590709
1009 1010
  GDSC before registration:  0.34810483
  GDSC after affine registration 0.3859839
  GDSC after grad_msd deformable registration 0.5215652
  GDSC after demons deformable registration 0.4297017
1001 1014
  GDSC before registration:  0.29346314
  GDSC after affine registration 0.367543
  GDSC after grad_msd deformable registration 0.48808372
  GDSC after demons deformable registration 0.45490834
1006 1000
  GDSC before registration:  0.16061912
  GDSC after affine registration 0.34566307
  GDSC after grad_msd deformable registration 0.4872934
  GDSC after demons deformable registration 0.42142546
1013 1007
  GDSC before registration:  0.22376765
  GDSC after affine registration 0.35946524
  GDSC after grad_msd deformable registration 0.49983105
  GDSC after demons deformable registration 0.4653676
1008 1011
  GDSC before registration:  0.2757263
  GDSC after affine registration 0.43848667
  GDSC after grad_msd deformable registration 0.5365502
  GDSC after demons deformable registration 0.51401544
1008 1009
  GDSC before registration:  0.3564495
  GDSC after affine registration 0.37907672
  GDSC after grad_msd deformable registration 0.45155773
  GDSC after demons deformable registration 0.38594896
1013 1006
  GDSC before registration:  0.28333756
  GDSC after affine registration 0.39988133
  GDSC after grad_msd deformable registration 0.48959446
  GDSC after demons deformable registration 0.44302535
```

[35]:
```
print('GDSC before registration: ', print_mean_std(gdsc_no_reg))
print('GDSC after affine registration', print_mean_std(gdsc_affine))
print('GDSC after grad_msd deformable registration',␣
 ↪print_mean_std(gdsc_gradmsd))
print('GDSC after demons deformable registration', print_mean_std(gdsc_demons))
```

```
GDSC before registration:  (0.246104, 0.07426206)
GDSC after affine registration (0.37692672, 0.032781854)
GDSC after grad_msd deformable registration (0.49396285, 0.022437815)
GDSC after demons deformable registration (0.44628677, 0.031206675)
```

4 Extension: Multi-Atlas Segmentation

[36]:
```
list_atlas = [1000, 1001, 1002, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1013,
              1014, 1015, 1017, 1036]

# randomly choose the 10 atlas images
atlas_imgs = np.random.choice(list_atlas, 10)
```

[37]:
```
def get_strong_segmentation(id_fix, atlas_imgs=atlas_imgs):
    """Generate strong segmentation for a given image ID

    :param id_fix: ID for the target image
    :param atlas_imgs: list of IDs for the atlas images, defaults to atlas_imgs
    :return:
        Tuple (strong_seg_aff, strong_seg_def, aff_gdsc, def_gdsc, exp.S_fix)␣
 ↪where

        - `strong_seg_aff` is the affine registered strong segmentation
        - `strong_seg_def` is the deformable registered strong segmentation
        - `aff_gdsc` is the GDSC between the ground truth and affine registered␣
↪strong segmentation
        - `def_gdsc` is the GDSC between the ground truth and deformable␣
↪registered strong segmentation
        - `exp.S_fix` is the ground truth segmentation
    """
    gdsc_no_reg = []
    gdsc_affine = []
    gdsc_def = []

    weak_segmentations_aff = []
    weak_segmentations_def = []

    for id_mov in atlas_imgs:
        print(id_fix, id_mov)
        gdsc_all, segmentations = evaluate_gdsc(id_fix, id_mov, grad_msd=True)

        gdsc_no_reg.append(gdsc_all[0])
        gdsc_affine.append(gdsc_all[1])
        gdsc_def.append(gdsc_all[2])

        weak_segmentations_aff.append(segmentations[0])
        weak_segmentations_def.append(segmentations[1])

    print('\nWEAK SEGMENTATIONS')
    print('  Average GDSC before registration: ', print_mean_std(gdsc_no_reg))
    print('  Average GDSC after affine registration: ',␣
↪print_mean_std(gdsc_affine))
    print('  Average GDSC after deformable registration: ',␣
↪print_mean_std(gdsc_def))

    strong_seg_aff = my_majority_vote(weak_segmentations_aff)
    strong_seg_def = my_majority_vote(weak_segmentations_def)

    exp = RegistrationExperiment(str(id_fix),str(id_mov), data_path,
                                 dtype=torch.float32, device=cuda)

    aff_gdsc = my_generalized_dice(exp.S_fix, strong_seg_aff).cpu().numpy()
    def_gdsc = my_generalized_dice(exp.S_fix, strong_seg_def).cpu().numpy()

    print('\nSTRONG SEGMENTATIONS')
    print('  GDSC with only affine registration: ', aff_gdsc)
    print('  GDSC with deformable registration: ', def_gdsc)

    return strong_seg_aff, strong_seg_def, aff_gdsc, def_gdsc, exp.S_fix
```

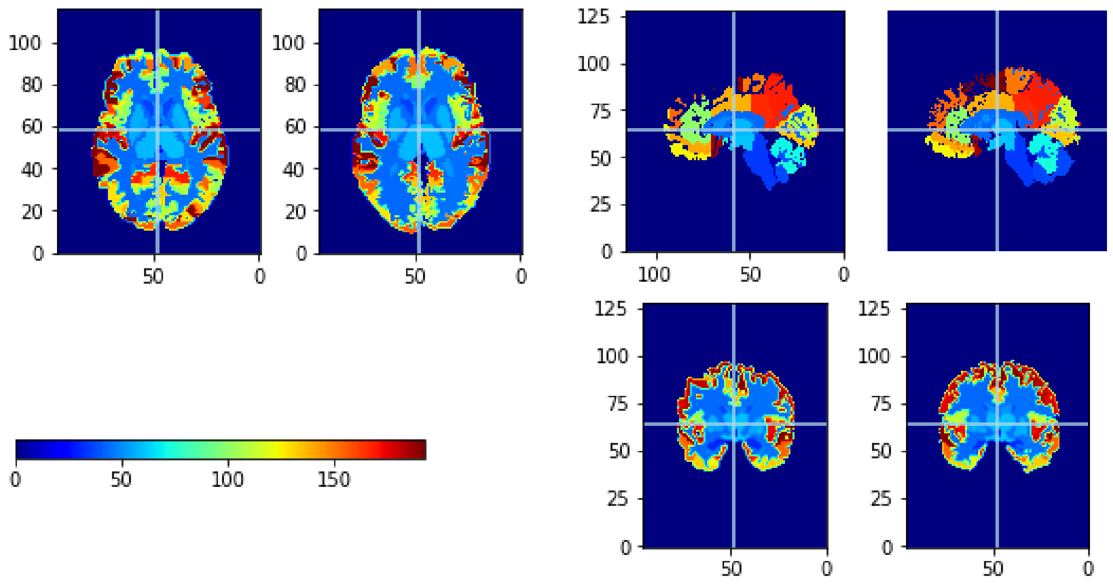
4.1 Target Image 1003

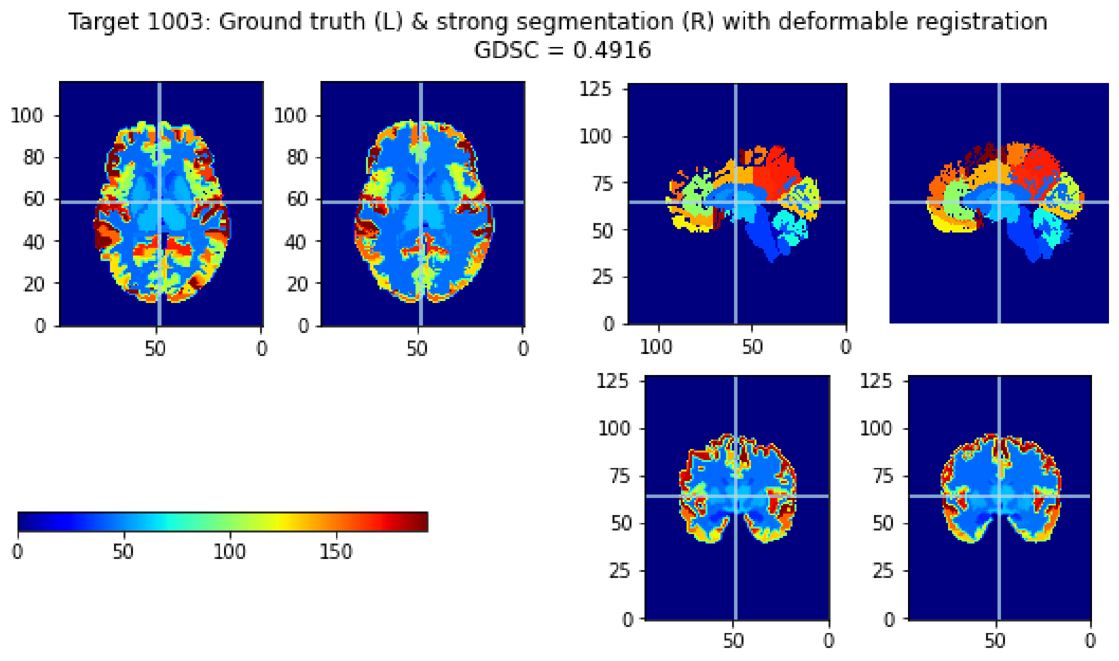
```
[38]: strong_seg_aff, strong_seg_def, aff_gdsc, def_gdsc, S_fix =␣
        ↪get_strong_segmentation(1003)
```

```
1003 1000
  GDSC before registration:  0.1316802
  GDSC after affine registration 0.3823198
  GDSC after grad_msd deformable registration 0.4354621
1003 1036
  GDSC before registration:  0.15730593
  GDSC after affine registration 0.37248668
  GDSC after grad_msd deformable registration 0.39088637
1003 1001
  GDSC before registration:  0.30236235
  GDSC after affine registration 0.42282417
  GDSC after grad_msd deformable registration 0.34501716
1003 1001
  GDSC before registration:  0.30236235
  GDSC after affine registration 0.42282417
  GDSC after grad_msd deformable registration 0.34501716
1003 1036
  GDSC before registration:  0.15730593
  GDSC after affine registration 0.37248668
  GDSC after grad_msd deformable registration 0.39088637
1003 1014
  GDSC before registration:  0.31679967
  GDSC after affine registration 0.3829238
  GDSC after grad_msd deformable registration 0.39848256
1003 1014
  GDSC before registration:  0.31679967
  GDSC after affine registration 0.3829238
  GDSC after grad_msd deformable registration 0.39848256
1003 1009
  GDSC before registration:  0.35558972
  GDSC after affine registration 0.3873193
  GDSC after grad_msd deformable registration 0.5135692
1003 1007
  GDSC before registration:  0.38080025
  GDSC after affine registration 0.43082297
  GDSC after grad_msd deformable registration 0.49873152
1003 1012
  GDSC before registration:  0.20964095
  GDSC after affine registration 0.4415992
  GDSC after grad_msd deformable registration 0.5050789

WEAK SEGMENTATIONS
  Average GDSC before registration:  (0.2630647, 0.08589055)
  Average GDSC after affine registration:  (0.39985305, 0.025085386)
  Average GDSC after deformable registration:  (0.4221614, 0.06020854)

STRONG SEGMENTATIONS
  GDSC with only affine registration:  0.4895567
  GDSC with deformable registration:  0.4916204
```

[39]:
```
my_view_multi([S_fix, strong_seg_aff], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1003: Ground truth (L) & strong segmentation (R)␣
 ↪with only affine registration \nGDSC = {aff_gdsc:.4f}")

my_view_multi([S_fix, strong_seg_def], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1003: Ground truth (L) & strong segmentation (R)␣
 ↪with deformable registration \nGDSC = {def_gdsc:.4f}")
```

Target 1003: Ground truth (L) & strong segmentation (R) with only affine registration
GDSC = 0.4896





Target 1003: Ground truth (L) & strong segmentation (R) with deformable registration
GDSC = 0.4916

4.2 Target Image 1004

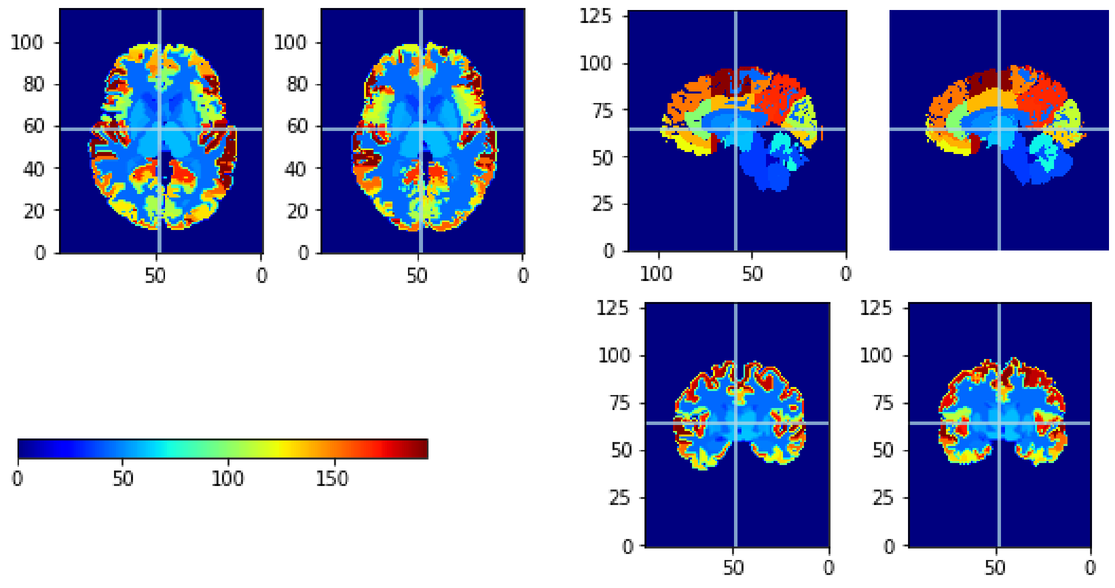
```
[40]: strong_seg_aff, strong_seg_def, aff_gdsc, def_gdsc, S_fix =␣
       ↪get_strong_segmentation(1004)
```

```
1004 1000
  GDSC before registration:  0.1809921
  GDSC after affine registration 0.40627903
  GDSC after grad_msd deformable registration 0.5095027
1004 1036
  GDSC before registration:  0.22732563
  GDSC after affine registration 0.3882557
  GDSC after grad_msd deformable registration 0.5001966
1004 1001
  GDSC before registration:  0.31397113
  GDSC after affine registration 0.40873292
  GDSC after grad_msd deformable registration 0.49865496
1004 1001
  GDSC before registration:  0.31397113
  GDSC after affine registration 0.40873292
  GDSC after grad_msd deformable registration 0.49865496
1004 1036
  GDSC before registration:  0.22732563
  GDSC after affine registration 0.3882557
  GDSC after grad_msd deformable registration 0.5001966
```

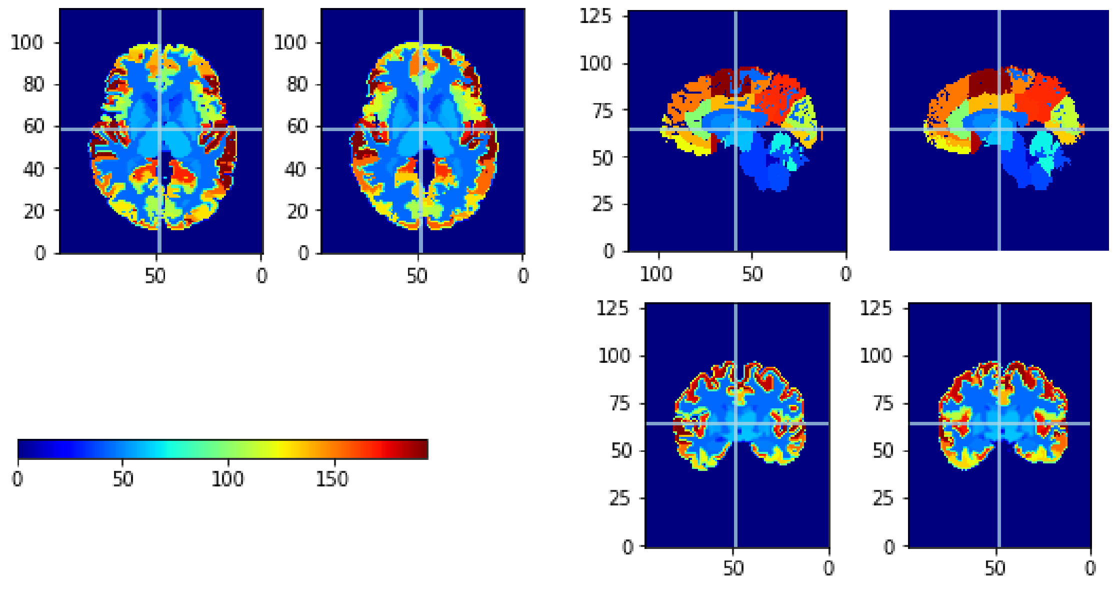
```
1004 1014
  GDSC before registration:  0.26888025
  GDSC after affine registration 0.37708142
  GDSC after grad_msd deformable registration 0.5004797
1004 1014
  GDSC before registration:  0.26888025
  GDSC after affine registration 0.37708142
  GDSC after grad_msd deformable registration 0.5004797
1004 1009
  GDSC before registration:  0.29996473
  GDSC after affine registration 0.36913952
  GDSC after grad_msd deformable registration 0.4326929
1004 1007
  GDSC before registration:  0.28079155
  GDSC after affine registration 0.38683254
  GDSC after grad_msd deformable registration 0.49758324
1004 1012
  GDSC before registration:  0.28710777
  GDSC after affine registration 0.44979498
  GDSC after grad_msd deformable registration 0.4808879

WEAK SEGMENTATIONS
  Average GDSC before registration:  (0.26692098, 0.040799823)
  Average GDSC after affine registration:  (0.3960186, 0.022282528)
  Average GDSC after deformable registration:  (0.49193293, 0.020844763)

STRONG SEGMENTATIONS
  GDSC with only affine registration:  0.48550487
  GDSC with deformable registration:  0.5952851
```

[41]:
```
my_view_multi([S_fix, strong_seg_aff], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1004: Ground truth (L) & strong segmentation (R)␣
 ↪with only affine registration \nGDSC = {aff_gdsc:.4f}")

my_view_multi([S_fix, strong_seg_def], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1004: Ground truth (L) & strong segmentation (R)␣
 ↪with deformable registration \nGDSC = {def_gdsc:.4f}")
```

Target 1004: Ground truth (L) & strong segmentation (R) with only affine registration
GDSC = 0.4855



Target 1004: Ground truth (L) & strong segmentation (R) with deformable registration
GDSC = 0.5953

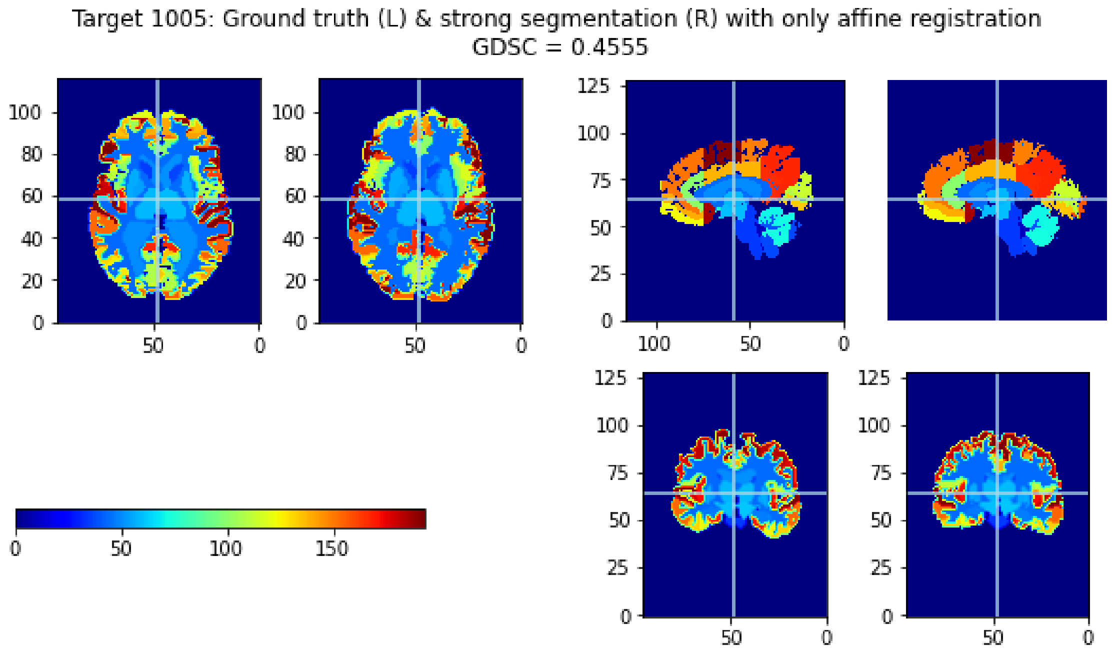


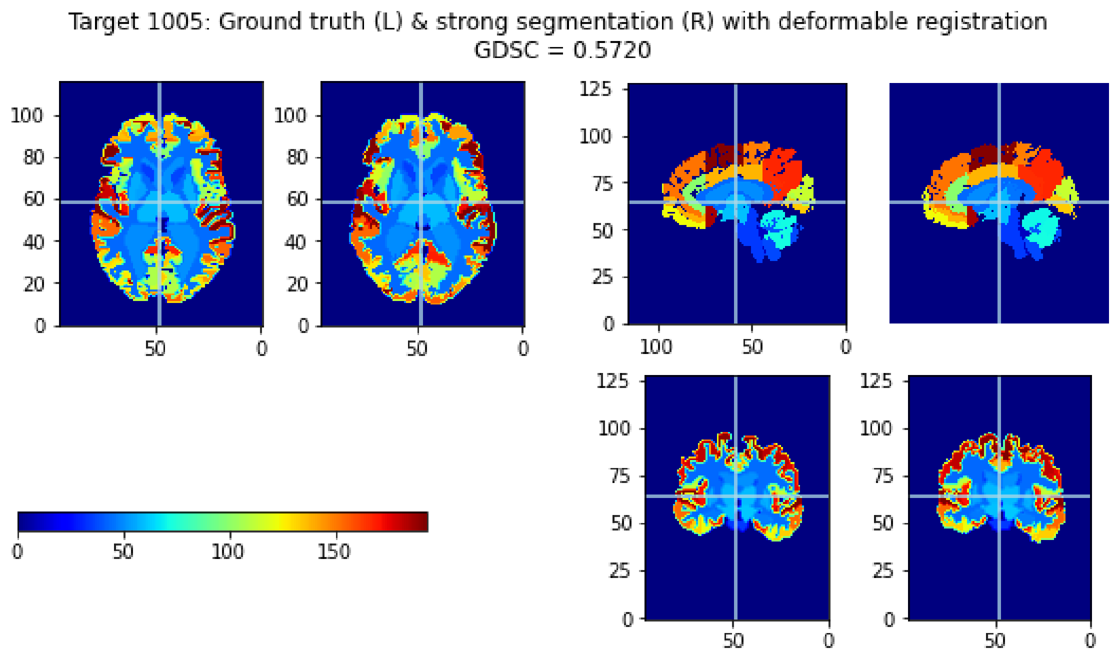
4.3 Target Image 1005

```
[42]: strong_seg_aff, strong_seg_def, aff_gdsc, def_gdsc, S_fix =␣
        ↪get_strong_segmentation(1005)
```

```
1005 1000
  GDSC before registration:  0.20155624
  GDSC after affine registration 0.37697098
  GDSC after grad_msd deformable registration 0.4967663
1005 1036
  GDSC before registration:  0.2049951
  GDSC after affine registration 0.36643764
  GDSC after grad_msd deformable registration 0.4970466
1005 1001
  GDSC before registration:  0.28162566
  GDSC after affine registration 0.36923528
  GDSC after grad_msd deformable registration 0.46596432
1005 1001
  GDSC before registration:  0.28162566
  GDSC after affine registration 0.36923528
  GDSC after grad_msd deformable registration 0.46596432
1005 1036
  GDSC before registration:  0.2049951
  GDSC after affine registration 0.36643764
  GDSC after grad_msd deformable registration 0.4970466
1005 1014
  GDSC before registration:  0.28053266
  GDSC after affine registration 0.39449316
  GDSC after grad_msd deformable registration 0.48526144
1005 1014
  GDSC before registration:  0.28053266
  GDSC after affine registration 0.39449316
  GDSC after grad_msd deformable registration 0.48526144
1005 1009
  GDSC before registration:  0.2820854
  GDSC after affine registration 0.34989637
  GDSC after grad_msd deformable registration 0.42476666
1005 1007
  GDSC before registration:  0.28492218
  GDSC after affine registration 0.35587177
  GDSC after grad_msd deformable registration 0.48642522
1005 1012
  GDSC before registration:  0.25037813
  GDSC after affine registration 0.4019997
  GDSC after grad_msd deformable registration 0.45774922

WEAK SEGMENTATIONS
  Average GDSC before registration:  (0.25532487, 0.03496879)
  Average GDSC after affine registration:  (0.3745071, 0.016430398)
  Average GDSC after deformable registration:  (0.47622523, 0.021808852)

STRONG SEGMENTATIONS
  GDSC with only affine registration:  0.45553008
  GDSC with deformable registration:  0.5719938
```

[43]:
```
my_view_multi([S_fix, strong_seg_aff], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1005: Ground truth (L) & strong segmentation (R)␣
 ↪with only affine registration \nGDSC = {aff_gdsc:.4f}")

my_view_multi([S_fix, strong_seg_def], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1005: Ground truth (L) & strong segmentation (R)␣
 ↪with deformable registration \nGDSC = {def_gdsc:.4f}")
```


Target 1005: Ground truth (L) & strong segmentation (R) with only affine registration
GDSC = 0.4555


Target 1005: Ground truth (L) & strong segmentation (R) with deformable registration
GDSC = 0.5720

4.4 Target Image 1018

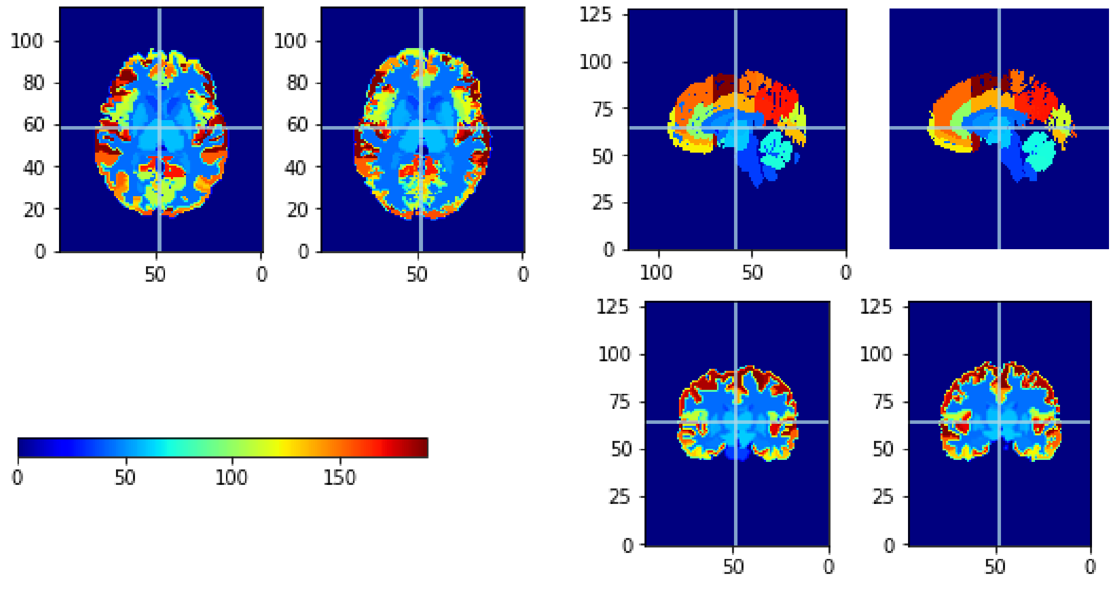
[44]:
```
strong_seg_aff, strong_seg_def, aff_gdsc, def_gdsc, S_fix =␣
↪get_strong_segmentation(1018)
```

```
1018 1000
  GDSC before registration:  0.1224667
  GDSC after affine registration 0.3658713
  GDSC after grad_msd deformable registration 0.51392365
1018 1036
  GDSC before registration:  0.1339297
  GDSC after affine registration 0.36231488
  GDSC after grad_msd deformable registration 0.46119422
1018 1001
  GDSC before registration:  0.26013175
  GDSC after affine registration 0.40272295
  GDSC after grad_msd deformable registration 0.44714528
1018 1001
  GDSC before registration:  0.26013175
  GDSC after affine registration 0.40272295
  GDSC after grad_msd deformable registration 0.44714528
1018 1036
  GDSC before registration:  0.1339297
  GDSC after affine registration 0.36231488
  GDSC after grad_msd deformable registration 0.46119422
```

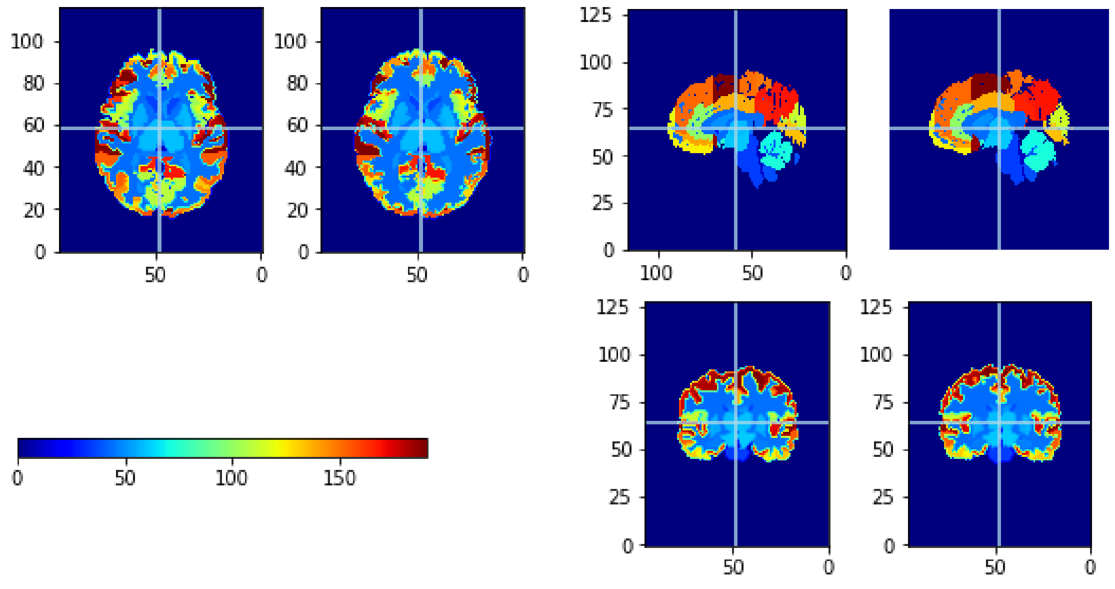
```
1018 1014
  GDSC before registration:  0.38094145
  GDSC after affine registration 0.41549993
  GDSC after grad_msd deformable registration 0.50124466
1018 1014
  GDSC before registration:  0.38094145
  GDSC after affine registration 0.41549993
  GDSC after grad_msd deformable registration 0.50124466
1018 1009
  GDSC before registration:  0.32738164
  GDSC after affine registration 0.38014004
  GDSC after grad_msd deformable registration 0.5111169
1018 1007
  GDSC before registration:  0.33075002
  GDSC after affine registration 0.4072966
  GDSC after grad_msd deformable registration 0.5107064
1018 1012
  GDSC before registration:  0.2794172
  GDSC after affine registration 0.42794567
  GDSC after grad_msd deformable registration 0.5113146

WEAK SEGMENTATIONS
  Average GDSC before registration:  (0.26100212, 0.09479947)
  Average GDSC after affine registration:  (0.39423293, 0.023230271)
  Average GDSC after deformable registration:  (0.48662296, 0.027152855)

STRONG SEGMENTATIONS
  GDSC with only affine registration:  0.48106903
  GDSC with deformable registration:  0.5740177
```

[45]:
```
my_view_multi([S_fix, strong_seg_aff], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1018: Ground truth (L) & strong segmentation (R)␣
↪with only affine registration \nGDSC = {aff_gdsc:.4f}")

my_view_multi([S_fix, strong_seg_def], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1018: Ground truth (L) & strong segmentation (R)␣
↪with deformable registration \nGDSC = {def_gdsc:.4f}")
```

Target 1018: Ground truth (L) & strong segmentation (R) with only affine registration
GDSC = 0.4811



Target 1018: Ground truth (L) & strong segmentation (R) with deformable registration
GDSC = 0.5740



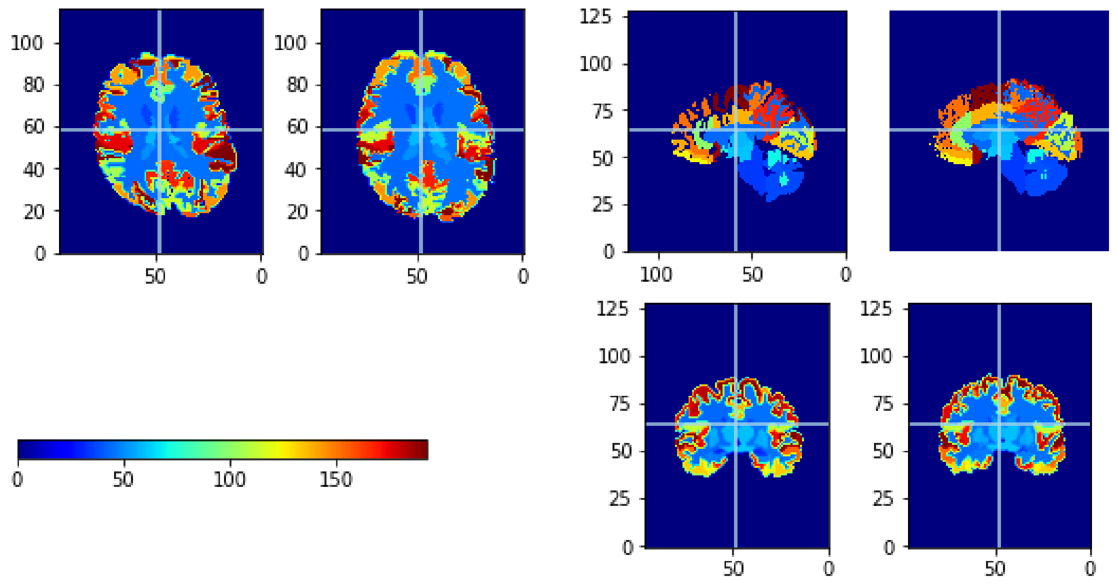
4.5 Target Image 1019

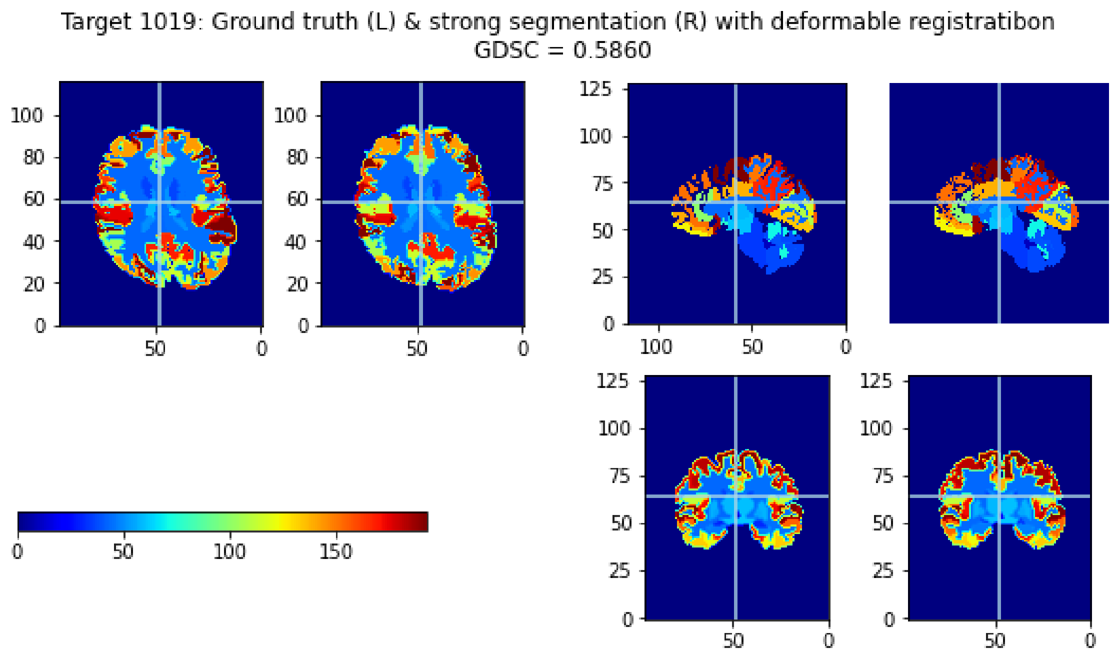
```
[46]: strong_seg_aff, strong_seg_def, aff_gdsc, def_gdsc, S_fix =␣
       ↪get_strong_segmentation(1019)
```

```
1019 1000
  GDSC before registration:  0.027663497
  GDSC after affine registration 0.3542557
  GDSC after grad_msd deformable registration 0.4780594
1019 1036
  GDSC before registration:  0.033684824
  GDSC after affine registration 0.34974688
  GDSC after grad_msd deformable registration 0.51479894
1019 1001
  GDSC before registration:  0.08404672
  GDSC after affine registration 0.37625387
  GDSC after grad_msd deformable registration 0.49218595
1019 1001
  GDSC before registration:  0.08404672
  GDSC after affine registration 0.37625387
  GDSC after grad_msd deformable registration 0.49218595
1019 1036
  GDSC before registration:  0.033684824
  GDSC after affine registration 0.34974688
  GDSC after grad_msd deformable registration 0.51479894
1019 1014
  GDSC before registration:  0.120412685
  GDSC after affine registration 0.3365032
  GDSC after grad_msd deformable registration 0.476658
1019 1014
  GDSC before registration:  0.120412685
  GDSC after affine registration 0.3365032
  GDSC after grad_msd deformable registration 0.476658
1019 1009
  GDSC before registration:  0.10763356
  GDSC after affine registration 0.36100587
  GDSC after grad_msd deformable registration 0.43466574
1019 1007
  GDSC before registration:  0.14144461
  GDSC after affine registration 0.3597839
  GDSC after grad_msd deformable registration 0.4779459
1019 1012
  GDSC before registration:  0.0719865
  GDSC after affine registration 0.37483683
  GDSC after grad_msd deformable registration 0.49617532

WEAK SEGMENTATIONS
  Average GDSC before registration:  (0.082501665, 0.038560484)
  Average GDSC after affine registration:  (0.35748905, 0.014261714)
  Average GDSC after deformable registration:  (0.48541322, 0.021847328)

STRONG SEGMENTATIONS
  GDSC with only affine registration:  0.42312393
  GDSC with deformable registration:  0.5860492
```

[47]:
```
my_view_multi([S_fix, strong_seg_aff], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1019: Ground truth (L) & strong segmentation (R)␣
 ↪with only affine registration \nGDSC = {aff_gdsc:.4f}")

my_view_multi([S_fix, strong_seg_def], header=exp.hdr_fix, cmap='jet',
              title=f"Target 1019: Ground truth (L) & strong segmentation (R)␣
 ↪with deformable registratibon \nGDSC = {def_gdsc:.4f}")
```

Target 1019: Ground truth (L) & strong segmentation (R) with only affine registration
GDSC = 0.4231




Target 1019: Ground truth (L) & strong segmentation (R) with deformable registratibon
GDSC = 0.5860

4.6 Analysis

Target image	Avg GDSC weak affine	Avg GDSC weak deformable	GDSC strong affine	GDSC strong deformable
1003	0.3999 ± 0.0251	0.4221 ± 0.0602	0.4896	0.4916
1004	0.3960 ± 0.0223	0.4919 ± 0.0208	0.4855	0.5953
1005	0.3745 ± 0.0164	0.4762 ± 0.0218	0.4555	0.5720
1018	0.3942 ± 0.0232	0.4866 ± 0.0272	0.4811	0.5740
1019	0.3575 ± 0.0143	0.4854 ± 0.0218	0.4231	0.5860

Change from affine to deformable (strong)	Change from weak to strong (affine)	Change from weak to strong (deformable)
0.0020	0.0897	0.0695
0.1098	0.0895	0.1034
0.1165	0.0810	0.0958
0.0929	0.0869	0.0874
0.1629	0.0656	0.1006

For all images, we see an upward trend in the GDSC from affine to deformable registration. There is also gain in GDSC when we use the strong segmentation as compared to each of the weak segmentations.