4 Part I: Python Imaging Basics

4.1.3

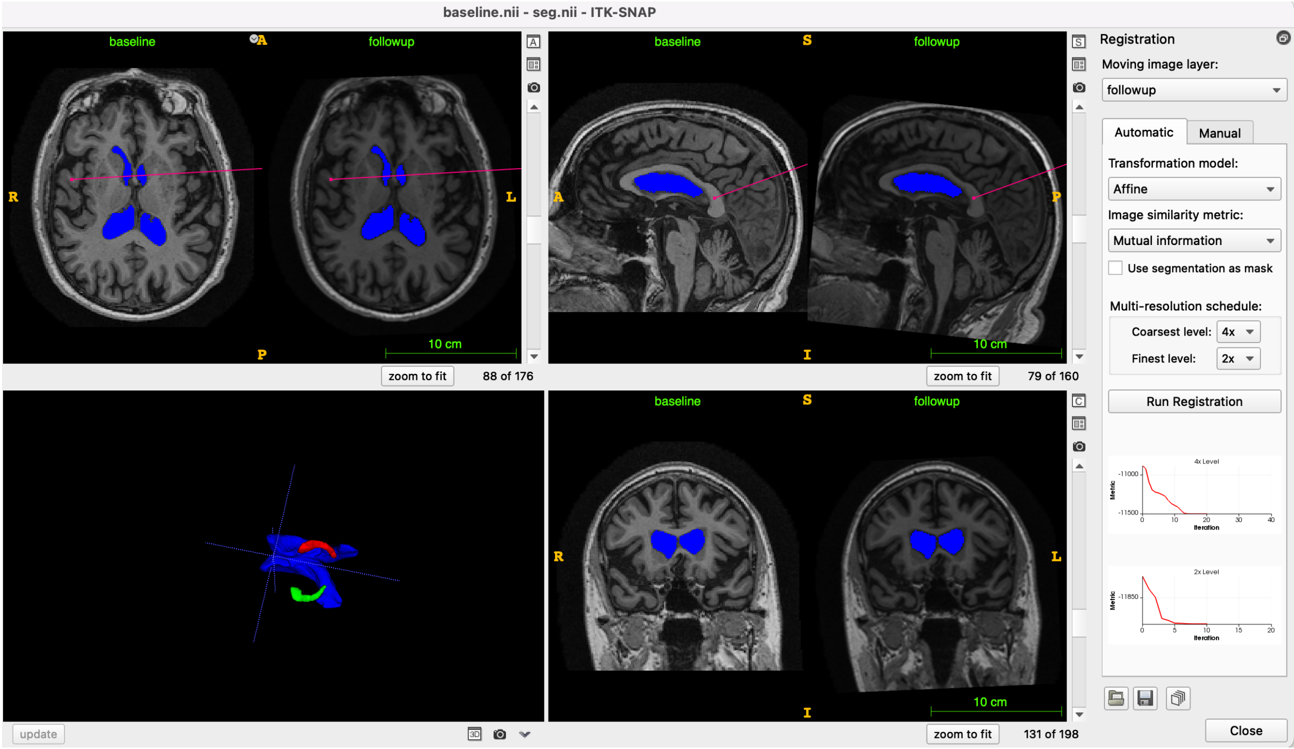
We first import a few packages that'll be used in this assignment.

In [4]:
```
# Import required libraries
import os
import numpy as np
import nibabel as nib
import matplotlib.pyplot as plt
from scipy import interpolate, signal
import math
from IPython import display

# Configure matplotlib options
import matplotlib
%matplotlib inline
matplotlib.rcParams['figure.figsize'] = [12, 8]

# Path to the data for this experiment
#data_dir='/Users/pauly/Documents/penn/BE5370/homework/hw1/data'
```

4.2 Explore the Data in ITK-SNAP

In [5]:
```
display.Image("/Users/feishu/Desktop/Screen_Shot.png")
```

Out[5]:



It seems like affine transformation for registration in ITK-SNAP works the best. As compared to manual manipulations of the registration parameters, which is simply just testing around the parameters, automatic affine transformation works very well.

4.3 Loading and Displaying 3D Images with Python

4.3.1

The functions below will be used to read and write 3D images from/to NIfTI files on disk.

In [6]:
```
def my_read_nifti(filename):
    """ Read NIfTI image voxels and header

    :param filename: path to the image to read
    :returns: tuple (img,hdr) consisting of numpy voxel array and nibabel NIfTI header
    """
    img = nib.load(filename)

    return img.get_fdata(), img.header


def my_write_nifti(filename, img, header = None):
    """ Write NIfTI image voxels and header

    :param filename: path to the image to save
    :param img: numpy voxel array
    :param header: nibabel NIfTI header
    """
    if header is not None:
        nifti_img = nib.Nifti1Image(img, affine=header.get_best_affine(), header=header)
    else:
        nifti_img = nib.Nifti1Image(img, affine=np.ones((len(img.shape),1)))
    nib.save(nifti_img, filename)
```

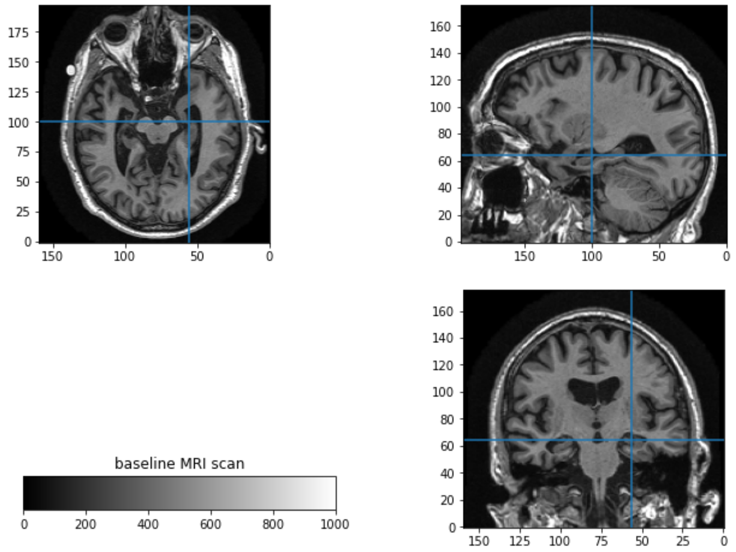
4.3.2 Viewing images

Here we define a `my_view` function to display 3D image in similar layout as ITK-SNAP.

In [7]:
```
def my_view(img, header=None, xhair=None, crange=None, cmap='gray'):
    """Display a 3D image in a layout similar to ITK-SNAP

    :param img: 3D voxel array
    :param header: Image header (returned by my_read_nifti)
    :param xhair: Crosshair position (1D array or tuple)
    :param crange: Intensity range, a tuple with minimum and maximum values
    :param cmap: Colormap (a string, see matplotlib documentation)
    """
    #render printing parameters
    if header == None:
        vx_size = (1,1,1)
    else:
        vx_size = header.get_zooms()

    if xhair == None:
        xhair = np.array(np.array(np.shape(img))/2).astype(int)

    if crange == None:
        crange = (0,1000)


    #create matplotlib figs
    fig, ax = plt.subplots(2,2 )

    im1 = ax[0,0].imshow(img[:,:,xhair[2]].transpose(),
               vmin=crange[0], vmax = crange[1],
               cmap = cmap,
               aspect= vx_size[1]/vx_size[0]
                )
    ax[0,0].invert_yaxis()
    ax[0,0].invert_xaxis()
    ax[0,0].axhline(xhair[1])
    ax[0,0].axvline(xhair[0])


    ax[0,1].imshow(img[xhair[0]].transpose(),
               vmin=crange[0], vmax = crange[1],
               cmap = cmap,
               aspect= vx_size[1]/vx_size[2]
                )
    ax[0,1].invert_yaxis()
    ax[0,1].invert_xaxis()
    ax[0,1].axhline(xhair[2])
    ax[0,1].axvline(xhair[1])

    ax[1,1].imshow(img[:,xhair[1]].transpose(),
               vmin=crange[0], vmax = crange[1],
               cmap = cmap,
               aspect= vx_size[2]/vx_size[0],
                )
    ax[1,1].invert_yaxis()
    ax[1,1].invert_xaxis()
    ax[1,1].axhline(xhair[2])
    ax[1,1].axvline(xhair[0])

    #setting colorbar
    ax[1,0].axis('off')
    #ax[1,0].set_xlabel('baseline MRI scan')
    cax = plt.axes([0.175, 0.15, 0.3, 0.05])

    cbar = plt.colorbar(im1, orientation='horizontal', ax=ax[0,0], cax=cax )

    return plt
```

In [8]:
```
I_bl,hdr_bl = my_read_nifti(os.path.join( 'data/baseline.nii'))
```

In [9]:
```
my_view(img = I_bl,
        header=hdr_bl,
        xhair=(56,100,64),
        #crange=(0,1000),
        cmap='gray')
plt.title('baseline MRI scan')
plt.show()
```



5 Part II: Effects of Low-Pass Filtering on Longitudinal Measures

In this section we implement functions to preform registration on the follow up image, as well as how low pass filtering affect image displays.

5.1.2 Implementation

We first perform transformation on the followup image. `my_read_transform` reads the A, b affine transformation matrix.

In [10]:
```
def my_read_transform(filename):
    """Read Greedy-style 3D transform (4x4 matrix) from file

    :param filename: File name containing transform file
    :returns: tuple (A,b) where A is the 3x3 affine matrix, b is the translation vector
    """
    raw = np.loadtxt(filename)

    A = raw[0:3, 0:3]
    b = raw[0:3,3]

    return A, b
```

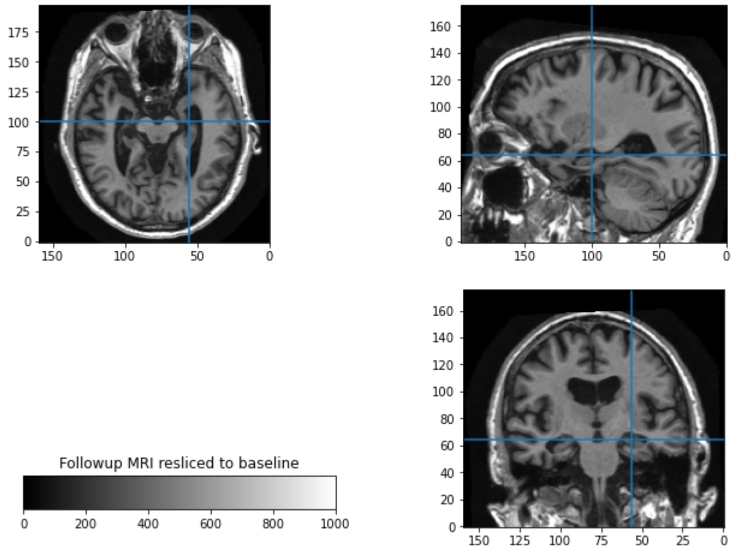
In [11]:
```
A, b = my_read_transform("data/f2b.txt")
```

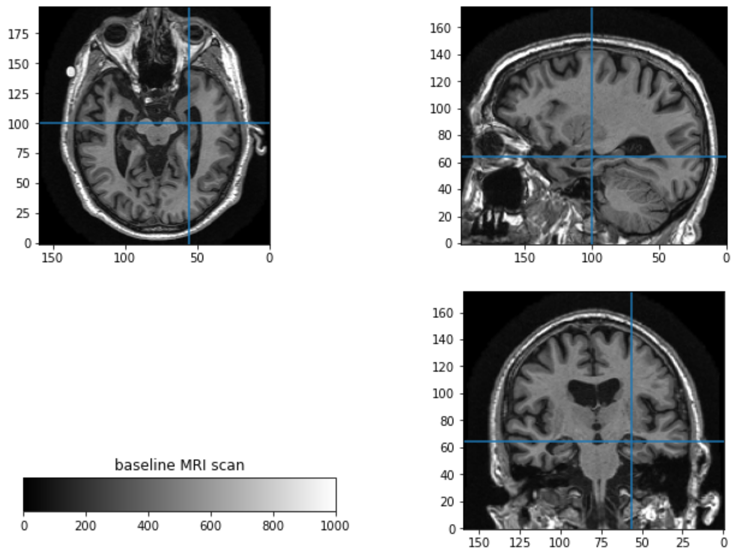
Next, we define `my_transform_image` that apply A and b to the follow up image

In [12]:
```
def my_transform_image(I_ref, I_mov, A, b, method='linear', fill_value=0):
    """Transform a moving image into the space of the fixed image

    :param I_ref: 3D voxel array of the fixed (reference) image
    :param I_mov: 3D voxel array of the moving image
    :param A: 3x3 affine transformation matrix
    :param b: 3x1 translation vector
    :param method: Interpolation method (e.g., 'linear', 'nearest')
    :param fill_value: Value with which to replace missing values (e.g., 0)
    :returns: 3D voxel array of the affine-transformed moving image
    """
    Row, Column, Slice = I_ref.shape
    x = np.linspace(0, Row-1, Row)
    y = np.linspace(0, Column-1, Column)
    z = np.linspace(0, Slice-1, Slice)

    x_m, y_m, z_m  = np.meshgrid(x,y,z,
                       indexing='ij')

    # map all points in space of follow up image
    x_m = x_m.reshape(Row* Column*Slice)
    y_m = y_m.reshape(Row* Column*Slice)
    z_m = z_m.reshape(Row* Column*Slice)

    f = np.dot( A , np.stack([x_m, y_m, z_m] ))  + b.reshape(3,1)

    x_, y_, z_ = f #, 3, 1)#.reshape(160,198,176)

    x_t = x_.reshape((160,198,176))
    y_t = y_.reshape((160,198,176))
    z_t = z_.reshape((160,198,176))

    interpolated = interpolate.interpn((x, y, z), I_mov, (x_t, y_t, z_t),
                               method = method, bounds_error=False, fill_value = fill_value )
    return interpolated
```

In [13]:
```
I_fu,hdr_fu = my_read_nifti(os.path.join( 'data/followup.nii'))
```

In [14]:
```
my_view(img = my_transform_image(I_bl, I_fu, A, b, method='linear', fill_value=0),
        header=hdr_bl,
        xhair=(56,100,64),
        crange=(0,1000),
        cmap='gray')
plt.title('Followup MRI resliced to baseline')
plt.show()
my_view(img = I_bl,
        header=hdr_bl,
        xhair=(56,100,64),
        crange=(0,1000),
        cmap='gray',)
plt.title('baseline MRI scan')
plt.show()
```





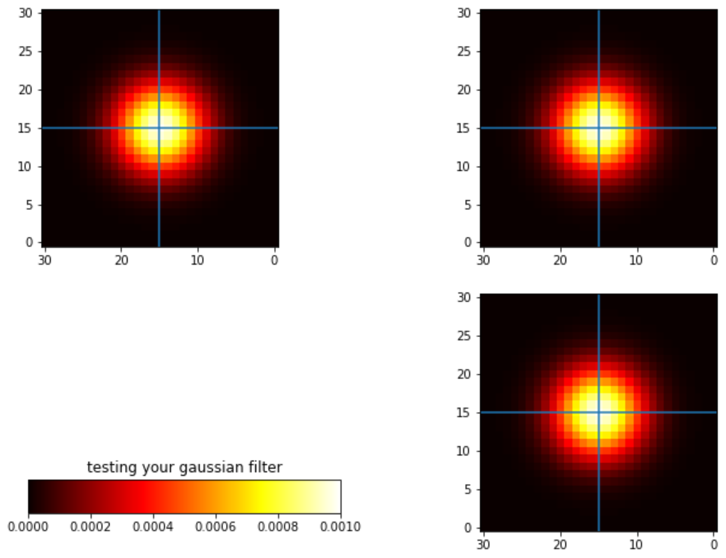
5.2 Effects of Low Pass Filtering on Difference Image Computation

Then we perform guassian low pass filtering.

In [15]:
```
def my_gaussian_lpf(image, sigma):
    """
    Apply 3D Gaussian low-pass filtering to an image
    :param image: 3D voxel array of the input image
    :param sigma: Standard deviation of the Gaussian kernel, in voxel units
    :returns: 3D voxel array of the filtered iamge
    """
    size = 2* 3.5 * sigma +1
    t = np.concatenate( (-np.array( range(int(size/2) ,0,-1) ), np.array( range(int((size+1)/2))) ) )

    g_ = np.exp(-t**2 / (2 * (sigma**2)) ) / (sigma* np.sqrt(2*math.pi) )
    g_x, g_y, g_z = np.meshgrid(g_,g_,g_)
    G = g_x *g_y* g_z
    #G = (g_,g_,g_)

    return signal.fftconvolve(image, G, mode = 'same')
```

In [16]:
```
test_gaussian = np.zeros((31,31,31))
test_gaussian[15][15][15] =1
```

In [65]:
```
my_view(img = my_gaussian_lpf(test_gaussian, sigma = 4),
        header=None,
        xhair=None,
        crange=(0,0.001),
        cmap='hot')
plt.title('testing your gaussian filter')
plt.show()
```



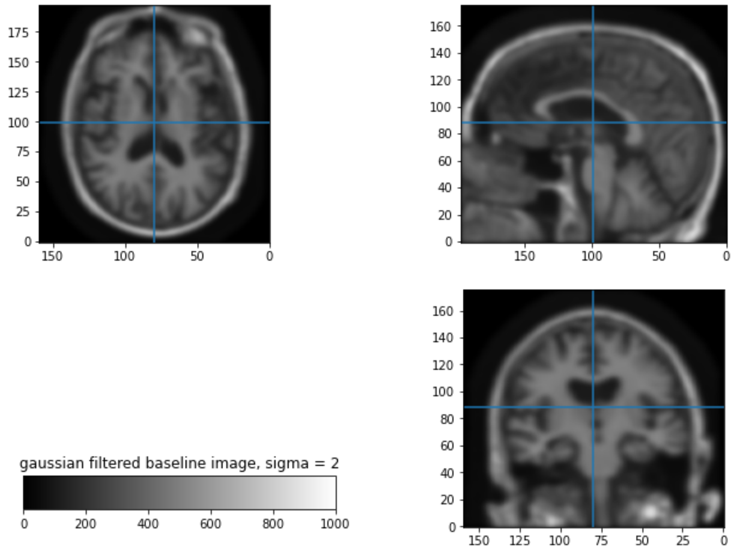
5.2.2 Mean Filter

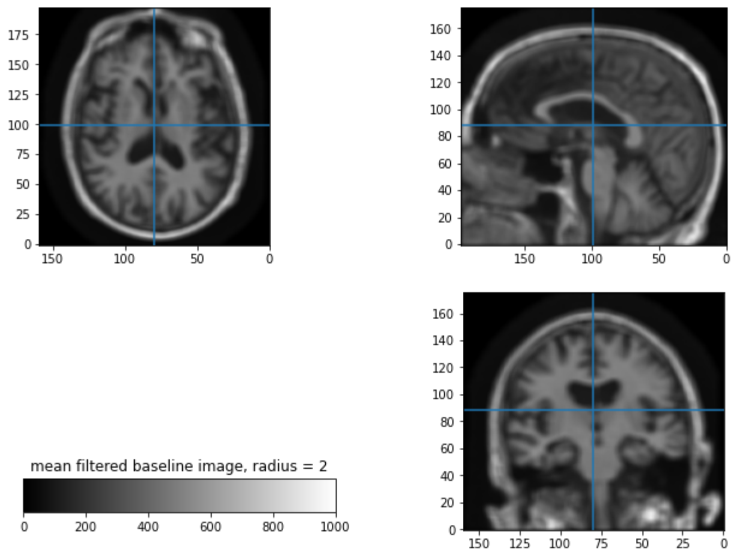
We also try mean filtering.

In [18]:
```
def my_mean_lpf(image, radius):
    """
    Apply 3D mean filtering to an image
    """
    kern=np.ones((2*radius+1)**3) / ((2*radius+1)**3)
    kern = kern.reshape((2*radius+1,2*radius+1,2*radius+1))

    return signal.fftconvolve(image, kern, mode = 'same')
```

Now we try out Gaussian filter and mean filter with the baseline image.

In [66]:
```
my_view(img = my_gaussian_lpf(I_bl, sigma = 2),
        header=hdr_bl,
        xhair=None,
        crange=(0,1000),
        cmap='gray')
plt.title('gaussian filtered baseline image, sigma = 2')
plt.show()

my_view(img = my_mean_lpf(I_bl, 2),
        header=hdr_bl,
        xhair=None,
        crange=(0,1000),
        cmap='gray')
plt.title('mean filtered baseline image, radius = 2')
plt.show()
```



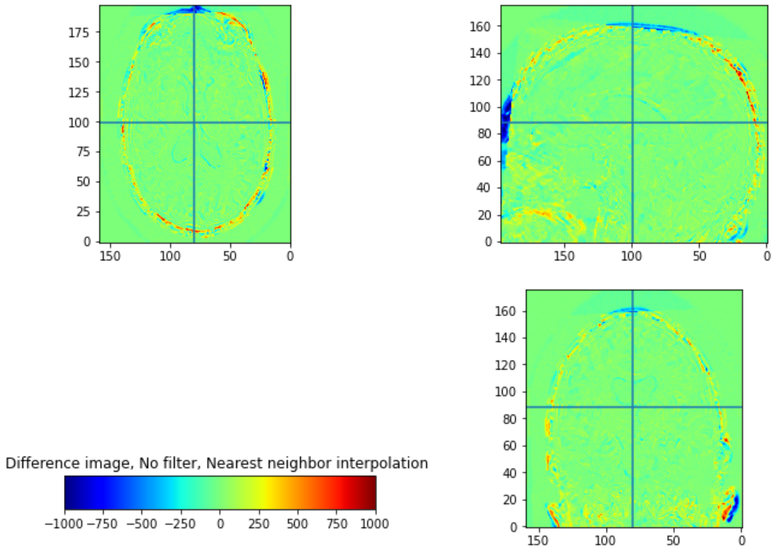


5.2.3 Compute and Show Difference Images

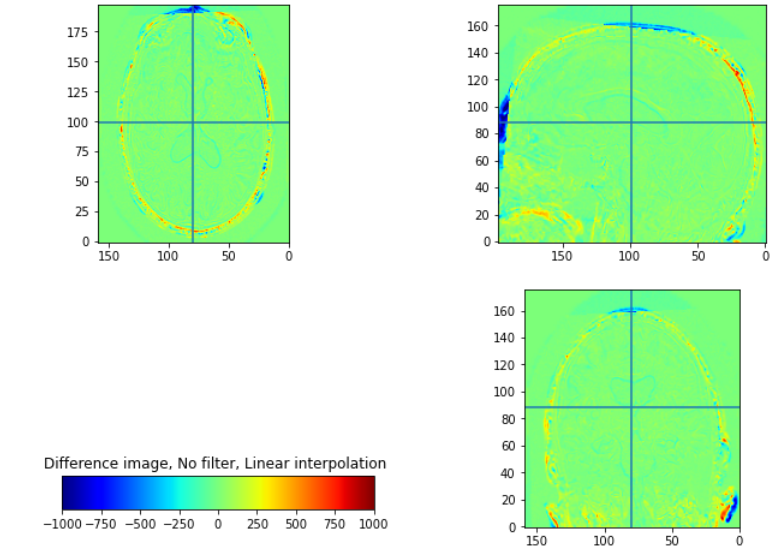
We the transformation function and the filters defined, in this section we visualize the effects of the transformation by plotting difference between baseline and registered followup. We as well visualize different combinations of filters.

In [20]:
```
I_fu,hdr_fu = my_read_nifti(os.path.join( 'data/followup.nii'))
```

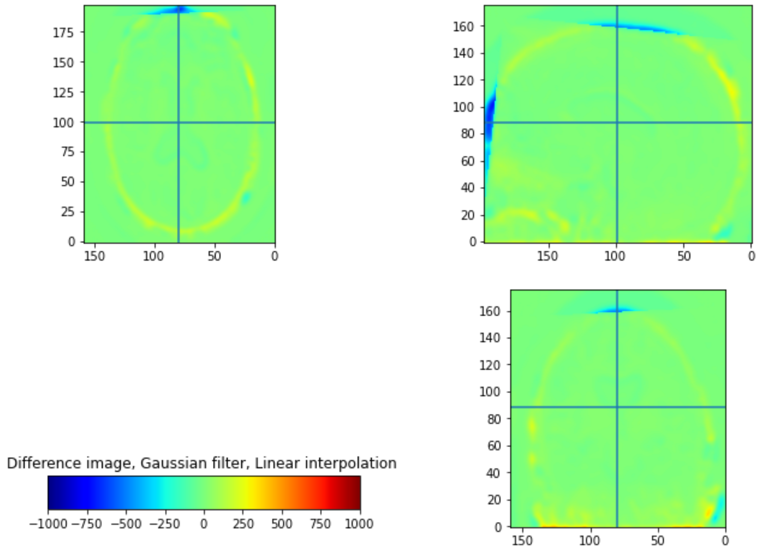
In [21]:
```
none_none_nearest = my_transform_image(I_bl, I_fu, A, b, method='nearest', fill_value=0) - I_bl
my_view(img = none_none_nearest,
        header=None,
        xhair=None,
        crange=(-1000,1000),
        cmap='jet')
plt.title('Difference image, No filter, Nearest neighbor interpolation ')
plt.show()
```



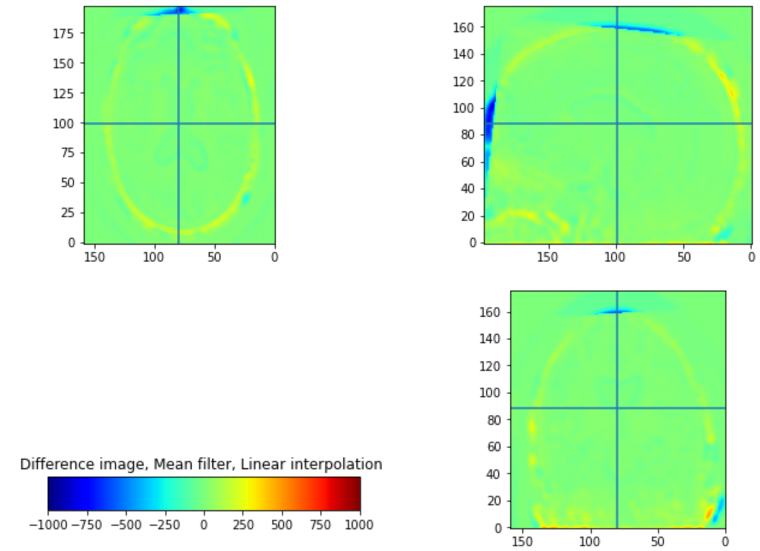
In [22]:
```
none_none_linear = my_transform_image(I_bl, I_fu, A, b, method='linear', fill_value=0) - I_bl
my_view(img = none_none_linear,
        header=None,
        xhair=None,
        crange=(-1000,1000),
        cmap='jet')
plt.title('Difference image, No filter, Linear interpolation ')
plt.show()
```



In [23]:
```
I_ref_g = my_gaussian_lpf(I_bl, 2)
I_mov_g = my_gaussian_lpf(I_fu, 2)
gaussian_linear = my_transform_image(I_ref_g , I_mov_g , A, b, method='linear', fill_value=0) - I_ref_g
my_view(img = gaussian_linear,
        header=None,
        xhair=None,
        crange=(-1000,1000),
        cmap='jet')
plt.title('Difference image, Gaussian filter, Linear interpolation ')
plt.show()
```



In [24]:
```
I_ref_m = my_mean_lpf(I_bl, 2)
I_mov_m = my_mean_lpf(I_fu, 2)
mean_linear = my_transform_image(I_ref_m , I_mov_m , A, b, method='linear', fill_value=0) - I_ref_m
my_view(img = mean_linear,
        header=None,
        xhair=None,
        crange=(-1000,1000),
        cmap='jet')
plt.title('Difference image, Mean filter, Linear interpolation ')
plt.show()
```



5.3 Quantify Intensity Difference over Regions of Interest

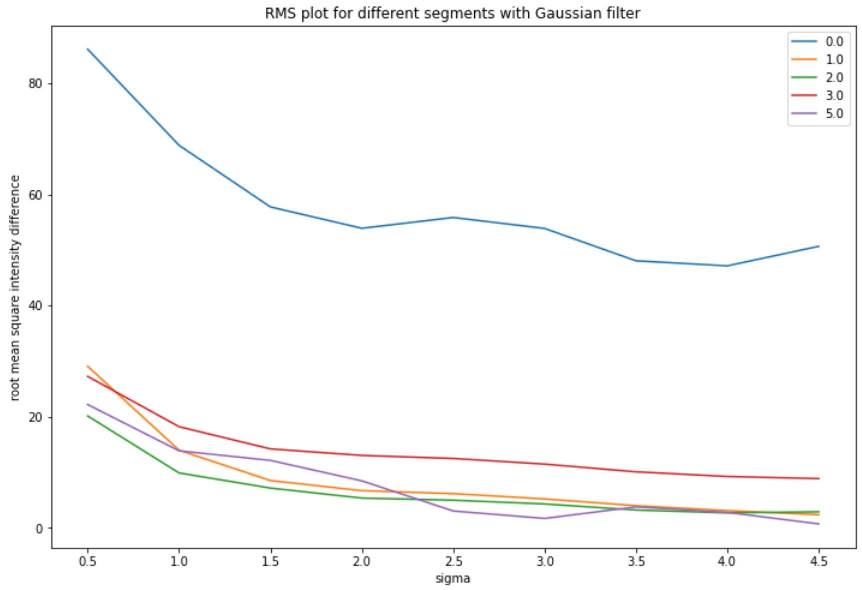
In this section we calculate the root mean square of differences between registered followup and baseline over defined roi regions.

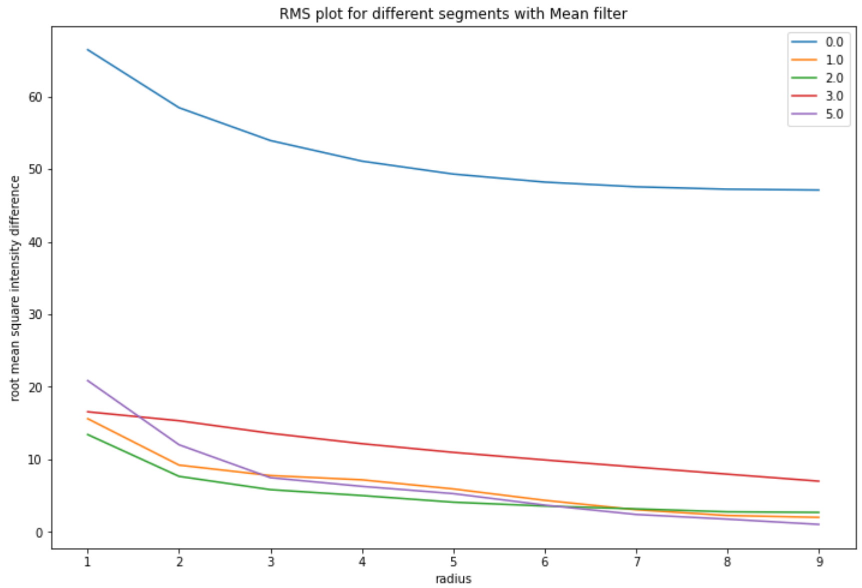
In [25]:
```
I_seg,hdr_seg = my_read_nifti(os.path.join( 'data/seg.nii'))
```

In [26]:
```
def my_rms_over_roi(image, seg, label):
    """Compute RMS of a difference image over a label in the segmentation
    """
    labels, counts = np.unique(seg, return_counts=True)
    d = dict(zip(labels, counts) )

    diff_squared = image **2
    summed = np.sum( np.where(seg == label, diff_squared, 0) )

    #print(d)
    return np.sqrt(summed / d[label] )
```

In [27]:
```
labels, counts = np.unique(I_seg, return_counts=True)
```

In [28]:
```
gaussian_rms = []
cur_rms = []
#fig, ax = plt.subplots()

for i in np.arange(0.5,5,0.5):
    cur_rms = []
    I_ref_g = my_gaussian_lpf(I_bl, i)
    I_mov_g = my_gaussian_lpf(I_fu, i)
    gaussianed = my_transform_image(I_ref_g , I_mov_g , A, b, method='linear', fill_value=0) - I_ref_g
    for j in labels:
        cur_rms.append(my_rms_over_roi(gaussianed, I_seg, j))
        #j = ax.plot([1, 2, 3], label='label1')
        #line2, = ax.plot([1, 2, 3], label='label2')
    gaussian_rms.append(cur_rms)

fig, ax = plt.subplots()
plt.plot(np.arange(0.5,5,0.5), gaussian_rms)#, label = )
ax.legend(labels)
plt.title( 'RMS plot for different segments with Gaussian filter')
plt.xlabel("sigma")
plt.ylabel("root mean square intensity difference")
plt.show()

mean_rms = []
cur_mean_rms = []
for i in np.arange(1,10,1):
    cur_mean_rms = []
    I_ref_m = my_mean_lpf(I_bl, i)
    I_mov_m = my_mean_lpf(I_fu, i)
    meaned = my_transform_image(I_ref_m , I_mov_m , A, b, method='linear', fill_value=0) - I_ref_m
    for j in labels:
        cur_mean_rms.append(my_rms_over_roi(meaned, I_seg, j))
    mean_rms.append(cur_mean_rms)

fig, ax = plt.subplots()
plt.plot(np.arange(1,10,1), mean_rms)#, label = )
ax.legend(labels)
plt.title( 'RMS plot for different segments with Mean filter')
plt.xlabel("radius")
plt.ylabel("root mean square intensity difference")
plt.show()
```





Plots demonstrates a tread that as sigma in Gaussian filter and radius in mean filter increases, the calculated RMS value is smaller. This indicates that increasing sigma and radiu might cause more smoothing to the images, thus cutting down some more features and resulting in less significant differences.

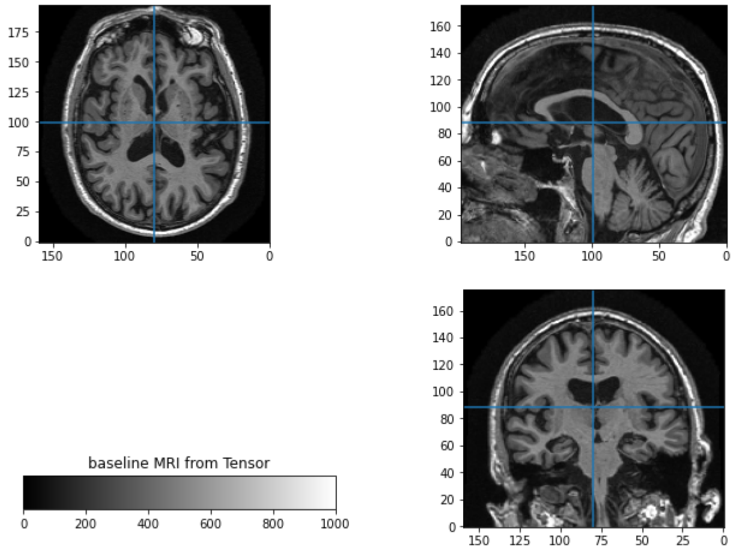
6 Part III: Affine Registration with PyTorch

In this section we perform the above functions in PyTorch, utilizing Pytorch inherent functions.

6.1 Loading 3D images into PyTorch tensors

In [29]:
```
import torch
import torch.nn.functional as tfun
```

In [33]:
```
T_bl = torch.from_numpy(I_bl).unsqueeze(0).unsqueeze(0)
my_view(T_bl.squeeze().detach().cpu().numpy(), hdr_bl, cmap='gray')
plt.title("baseline MRI from Tensor")
plt.show()
```



6.2 Applying Affine Transformations in PyTorch

Since pytorch and numpy have different coordinate systems, we implement functions to transform the affine parameters A, b from/to torch to/from numpy.

We first derive A', b' from A, b with this equation: $W(Ax+b)+z = A'(Wx+z)+b'$

In [44]:
```
display.Image("/Users/feishu/Desktop/derive.png")
```

Out[44]:

$$x' = Wx + z$$

$$W(Ax+b) + z = A'(Wx+z) + b'$$

$$WAx + Wb + z = A'Wx + A'z + b' \quad ①$$

$$WAx = A'Wx$$

$$A' = WAW^{-1} \quad ②$$

take ② into ①

$$WAx + Wb + z = WAW^{-1}Wx + WAW^{-1}z + b'$$

$$= WAx + Az + b'$$

$$Wb + z = WAW^{-1}z + b'$$

$$b' = Wb + z - A'z$$

$$\begin{cases} A' = WA * \text{inverse}(W) \\ b' = Wb + z - A'z \end{cases}$$

In [71]:
```
def my_numpy_affine_to_pytorch_affine(A, b, img_size):
    """Convert affine transform (A,b) from NumPy to PyTorch coordinates
    :param A: affine matrix, represented as a shape (3,3) NumPy array
    :param b: translation vector, represented as a shape (3) NumPy array
    :returns: tuple of NumPy arrays (A',b') holding affine transform in PyTorch coords """
    Sx, Sy, Sz = img_size
    W = [[0,0,2/Sz],[0,2/Sy,0], [2/Sx, 0, 0]]
    z = [1/Sz - 1, 1/Sy - 1, 1/Sx - 1]

    A_ =  np.matmul( np.matmul(W, A) ,np.linalg.inv(W) )
    b_ =  np.matmul( W , b ) + z -  np.matmul( A_, z)

    return A_, b_
```

In [79]:
```
def my_pytorch_affine_to_numpy_affine(A, b, img_size):
    """Convert affine transform (A,b) from PyTorch to NumPy coordinates
    :param A: affine matrix, represented as a shape (3,3) NumPy array
    :param b: translation vector, represented as a shape (3)NumPy array
    :returns: tuple of NumPy arrays (A',b') holding affine transform in NumPy coords """
    Sx, Sy, Sz = img_size
    W = [[0,0,2/Sz],[0,2/Sy,0], [2/Sx, 0, 0]]
    z = [1/Sz - 1, 1/Sy - 1, 1/Sx - 1]

    Aprime = A
    bprime = b

    A_ = np.matmul( np.linalg.inv(W), np.matmul( Aprime, W)  )
    b_ = np.matmul( np.linalg.inv(W) , np.matmul(Aprime, z)  + bprime - z )

    return A_, b_
```

In [73]:
```
A_prime, b_prime = my_numpy_affine_to_pytorch_affine(A, b, I_bl.shape)
A_prime, b_prime
```

Out[73]:
```
(array([[ 0.9939    , -0.120375  ,  0.04881818],
        [ 0.08791111,  1.001     ,  0.03361616],
        [-0.04499   , -0.0314325 ,  1.0003    ]]),
 array([ 0.18924318,  0.06270253, -0.00460125]))
```

In [80]:
```
A_test, b_test = my_pytorch_affine_to_numpy_affine(A_prime, b_prime, I_bl.shape)
A_test, b_test
```

Out[80]:
```
(array([[ 1.0003, -0.0254, -0.0409],
        [ 0.0416,  1.001 ,  0.0989],
        [ 0.0537, -0.107 ,  0.9939]]),
 array([ 5.6887, -5.8519, 23.4575]))
```

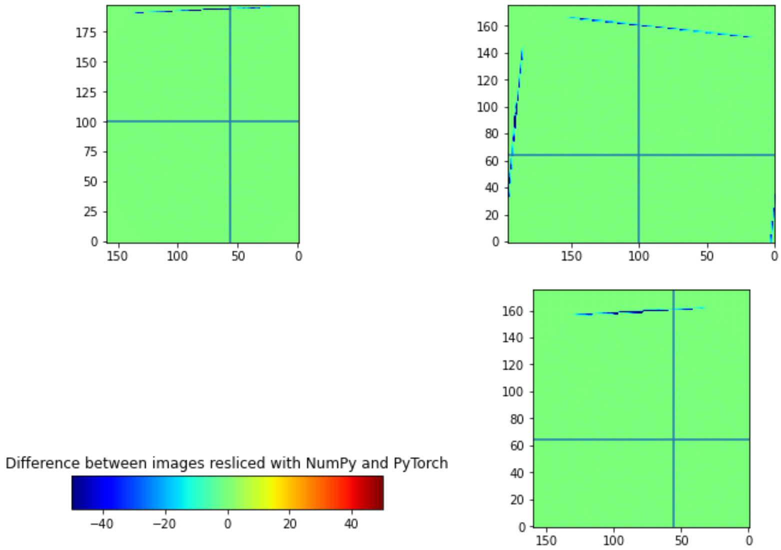
Now we are ready to transform image with pytorch A, b in python.

In [38]:
```
def my_transform_image_pytorch(T_ref, T_mov, T_A, T_b, mode='bilinear', padding_mode='zeros'):
    """Apply an affine transform to 3D images represented as PyTorch tensors
    :param T_ref: Fixed (reference) image, represented as a 5D tensor
    :param T_mov: Moving image, represented as a 5D tensor
    :param T_A: affine matrix in PyTorch coordinate space, represented as a shape (3,3) tensor
    :param T_b: translation vector in PyTorch coordinate space, represented as a shape (3) tensor
    :param mode: Interpolation mode, see grid_sample
    :param padding_mode: Padding mode, see grid_sample
    :returns: Transformed moving image, represented as a 5D tensor
    """
    x = T_ref.shape[2]
    y = T_ref.shape[3]
    z = T_ref.shape[4]

    grid = tfun.affine_grid(torch.eye(3,4).unsqueeze(0), torch.Size((1,1,x,y,z)), align_corners=False).double()
    #grid.double()
    grid_ = grid.squeeze(0).reshape(x*y*z, 3)
    x_g = grid_[:,0]
    y_g = grid_[:,1]
    z_g = grid_[:,2]

    x_gt,y_gt,z_gt = torch.matmul( T_A , torch.stack([x_g, y_g, z_g] ) ) + T_b.reshape(3,1)
    x_gt = x_gt.reshape((160,198,176))
    y_gt = y_gt.reshape((160,198,176))
    z_gt = z_gt.reshape((160,198,176))
    #np.stack([x_gt,y_gt,z_gt] ,axis = -1)
    grid_tensor = (torch.stack([x_gt,y_gt,z_gt] ,axis = -1))

    return tfun.grid_sample( T_mov.double(),
                 grid_tensor.unsqueeze(0).double() ,mode =mode, padding_mode = padding_mode, align_corners = False)
```

We check the performance of our pytorch transform function by plotting the difference of ptorch and numpy transformation functions.

In [40]:
```
# Convert the affine transform (A,b) to a PyTorch affine transform
A_prime, b_prime = my_numpy_affine_to_pytorch_affine(A, b, I_bl.shape)
T_A, T_b = torch.tensor(A_prime), torch.tensor(b_prime)

# Convert the baseline and follow-up images to PyTorch tensors
T_bl = torch.from_numpy(I_bl).unsqueeze(0).unsqueeze(0)
T_fu = torch.from_numpy(I_fu).unsqueeze(0).unsqueeze(0)

# Apply the transform to the follow-up image using PyTorch
T_fu_reslice = my_transform_image_pytorch(T_bl, T_fu, T_A, T_b)
# Apply the transform to the follow-up image using NumPy
I_fu_reslice = my_transform_image(I_bl, I_fu, A, b)

# Compute the difference between two ways of resampling
I_diff = I_fu_reslice - T_fu_reslice.squeeze().detach().cpu().numpy()
# Compute RMS difference between interpolated images over our labels

for label in (1,2,3):
    print('RMS over label %d is %f' % (label, my_rms_over_roi(I_diff, I_seg, label)))
# Visualize the difference image
my_view(I_diff, xhair=(56,100,64), crange=[-50,50], cmap='jet')
plt.title('Difference between images resliced with NumPy and PyTorch')
plt.show()
```

```
RMS over label 1 is 0.000197
RMS over label 2 is 0.000173
RMS over label 3 is 0.000134
```



6.3 Affine Registration using LBFGS Optimizer

Finally, we apply affine registration with pytorch optimizer.

6.3.1 Objective Function for Affine Registration

The apply the optimizer, we define an objective function that calculates RMS between affine registrated follow up and baseline MRI image as tensors.

In [41]:
```
def my_affine_objective_fn(T_ref, T_mov, T_A, T_b):
    """Compute the affine registration objective function
    :param T_ref: Fixed (reference) image, represented as a 5D tensor
    :param T_mov: Moving image, represented as a 5D tensor
    :param T_A: affine matrix in PyTorch coordinate space, represented as a shape (3,3) tensor :param T_b: translation vector in P
    """
    #transform reference:
    transformed_mov = my_transform_image_pytorch(T_ref, T_mov, T_A, T_b, mode='bilinear', padding_mode='zeros')

    diff = transformed_mov - T_ref
    diff_squared = diff **2.0
    summed = torch.sum( diff_squared )
    size = T_ref.shape[2]*T_ref.shape[3]*T_ref.shape[4]

    return torch.sqrt(summed / float(size))
```

In [45]:
```
my_affine_objective_fn(T_bl, T_fu, T_A, T_b)
```

Out[45]:
```
tensor(87.0636, dtype=torch.float64)
```

We test out partial derivatives of our objective function:

In [82]:
```
# Create tensors T_A and T_b and track their partial derivatives
T_A = torch.tensor(A_prime, requires_grad=True)
T_b = torch.tensor(b_prime, requires_grad=True)
# Compute the objective function (forward pass)
obj = my_affine_objective_fn(T_bl, T_fu, T_A, T_b)
# Compute the partial derivatives of the objective function with respect to
# elements of T_A and T_b automatically (backward pass)
obj.backward()
# Print the objective function value and partial derivatives
obj, T_A.grad, T_b.grad
```

Out[82]:
```
(tensor(87.0636, dtype=torch.float64, grad_fn=<SqrtBackward0>),
 tensor([[-15.0390, -78.2039,  29.2966],
         [ 52.9872, 164.1667,   3.8247],
         [-37.4327,  42.8411, -68.0883]], dtype=torch.float64),
 tensor([ 105.7288, -576.0979,  329.4931], dtype=torch.float64))
```

6.3.2 Minimizing the Objective Function

We use iterative approach to minimize the objective function.

In [47]:
```
# Starting point for optimization - identity affine transform. Note that
# the LBFGS implementation in PyTorch requires all the parameters (i.e.
# variables that we are optimizing over) to be contained in a single
# tensor, which we call T_opt
T_opt = torch.tensor(np.eye(4,4), requires_grad=True)
# Objective function for optimization, a wrapper around my_affine_objective_fn
f_opt = lambda : my_affine_objective_fn(T_bl, T_fu, T_opt[0:3,0:3], T_opt[0:3,3])
# Initialize the LBFGS optimizer with a line search routine
optimizer = torch.optim.LBFGS([T_opt],
                              history_size=10,
                              max_iter=4,
                              line_search_fn="strong_wolfe")
# Keep track of the objective function values over the course of optimization
opt_history = []
# Run for a few iterations
for i in range(50):
    optimizer.zero_grad()
    objective = f_opt()
    objective.backward()
    optimizer.step(f_opt)
    opt_history.append(objective.item())
    print('Iter %03d Obj %8.4f' % (i, objective.item()))
```

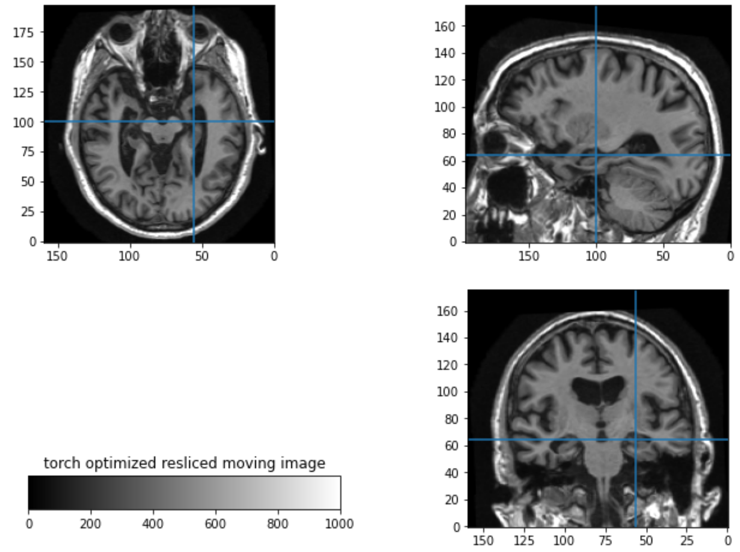
```
Iter 031 Obj  86.4243
Iter 032 Obj  86.4236
Iter 033 Obj  86.4235
Iter 034 Obj  86.4234
Iter 035 Obj  86.4234
Iter 036 Obj  86.4234
Iter 037 Obj  86.4234
Iter 038 Obj  86.4234
Iter 039 Obj  86.4234
Iter 040 Obj  86.4234
Iter 041 Obj  86.4234
Iter 042 Obj  86.4234
Iter 043 Obj  86.4234
Iter 044 Obj  86.4234
Iter 045 Obj  86.4234
Iter 046 Obj  86.4234
Iter 047 Obj  86.4234

Iter 048 Obj  86.4234
Iter 049 Obj  86.4234
```

In [49]:
```
T_opt[0:3,0:3], T_opt[0:3,3]
```

Out[49]:
```
(tensor([[ 0.9942, -0.1207,  0.0486],
         [ 0.0884,  1.0002,  0.0342],
         [-0.0450, -0.0318,  1.0011]], dtype=torch.float64,
        grad_fn=<SliceBackward0>),
 tensor([ 0.1888,  0.0642, -0.0053], dtype=torch.float64,
        grad_fn=<SelectBackward0>))
```

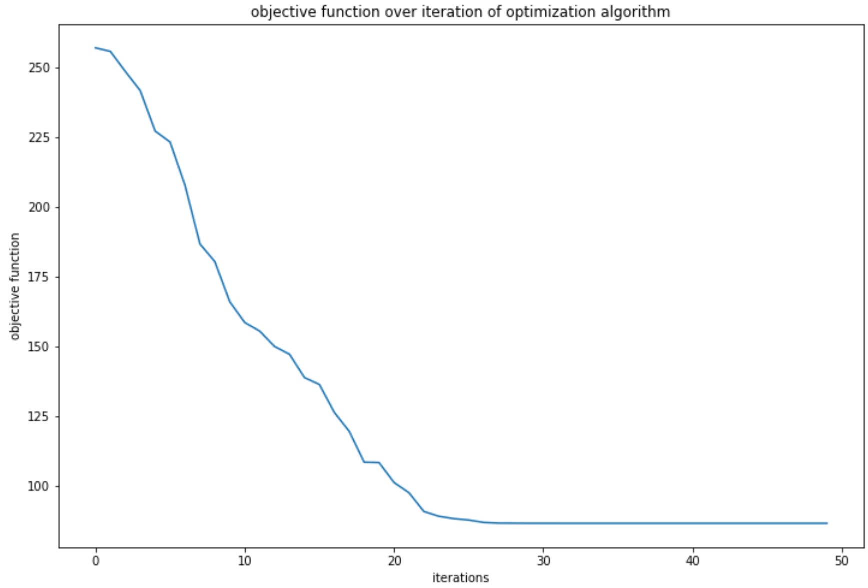
...And plot the optimized resliced moving image:

In [55]:
```
T_fu_opted = my_transform_image_pytorch(T_bl, T_fu, T_opt[0:3,0:3], T_opt[0:3,3],
                                        mode='bilinear', padding_mode='zeros')

my_view(T_fu_opted.squeeze().detach().cpu().numpy(), header = hdr_bl,
        xhair=(56,100,64), crange=[0,1000], cmap='gray')

plt.title('torch optimized resliced moving image')
plt.show()
```



...And the objective function through iterations as well:

In [56]:
```
# plot objective function:
fig, ax = plt.subplots()
plt.plot(range(50), opt_history)
plt.title("objective function over iteration of optimization algorithm")
plt.xlabel("iterations")
plt.ylabel("objective function")
plt.show()
```



cited from Assignment 1 instructions: "Convert the matrix A' and vector b′ computed by the optimization to a NumPy space matrix and compare to the affine transform loaded earlier from f2b.txt. They should be similar, since the latter was obtained by performing affine registration in ITK-SNAP. ":

In [57]:
```
A_opt, b_opt = my_pytorch_affine_to_numpy_affine(T_opt[0:3,0:3].detach().cpu().numpy(),
                                                 T_opt[0:3,3].detach().cpu().numpy(),
                                                 I_bl.shape)
A_opt, b_opt
```

Out[57]:
```
(array([[ 1.00105298, -0.03582787, -0.04094684],
        [ 0.03044359,  1.00019641,  0.07140776],
        [ 0.05351462, -0.14931126,  0.99416957]]),
 array([ 5.60948973, -5.73212395, 23.43297672]))
```

In [58]:
```
A, b
```

Out[58]:
```
(array([[ 1.0003, -0.0254, -0.0409],
        [ 0.0416,  1.001 ,  0.0989],
        [ 0.0537, -0.107 ,  0.9939]]),
 array([ 5.6887, -5.8519, 23.4575]))
```

they are indeed similar!

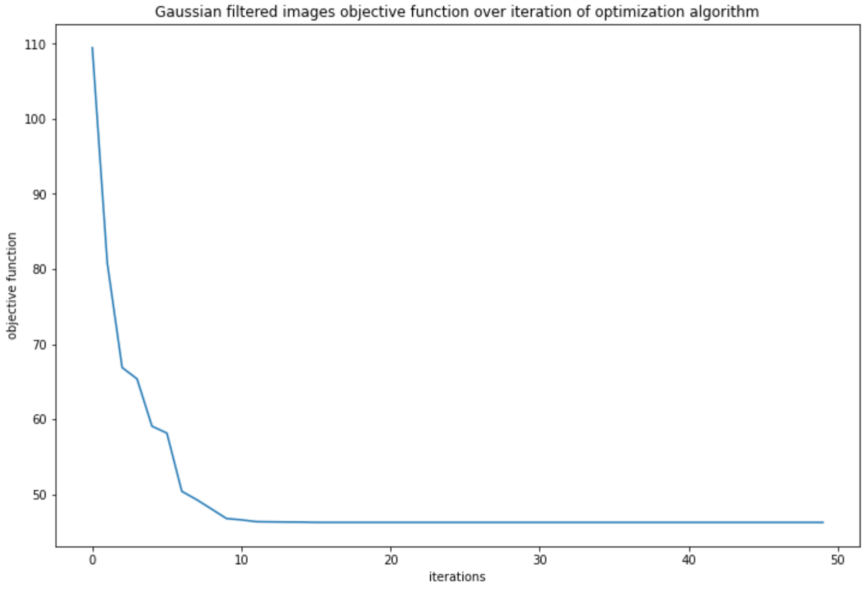
Next we examine whether applying Gaussian smoothing to the baseline and moving images before doing optimization impacts affine registration.

In [59]:
```
I_ref_g = my_gaussian_lpf(I_bl, sigma = 5)
I_mov_g = my_gaussian_lpf(I_fu, sigma = 5)
T_bl_g = torch.from_numpy(I_ref_g).unsqueeze(0).unsqueeze(0)
T_fu_g = torch.from_numpy(I_mov_g).unsqueeze(0).unsqueeze(0)
#gaussianed = my_transform_image(I_ref_g , I_mov_g , A, b, method='linear', fill_value=0) - I_ref_g
```

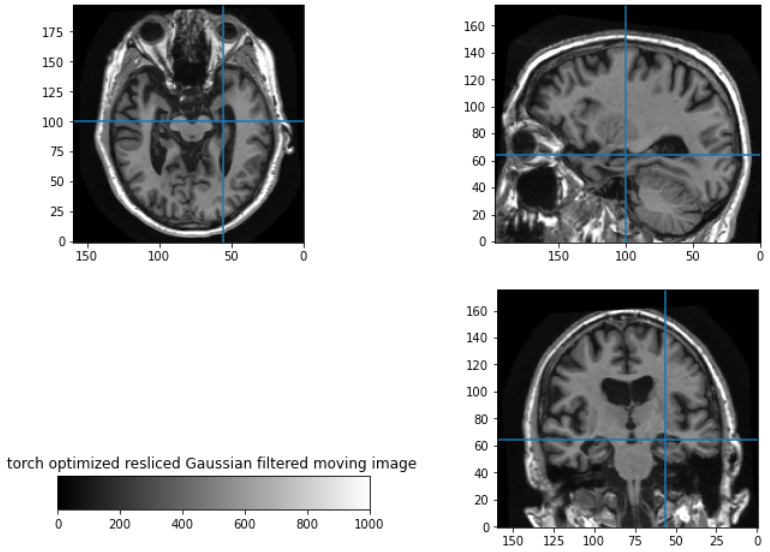
In [60]:
```
# Starting point for optimization - identity affine transform. Note that
# the LBFGS implementation in PyTorch requires all the parameters (i.e.
# variables that we are optimizing over) to be contained in a single
# tensor, which we call T_opt
T_opt_g = torch.tensor(np.eye(4,4), requires_grad=True)
# Objective function for optimization, a wrapper around my_affine_objective_fn
f_opt_g = lambda : my_affine_objective_fn(T_bl_g, T_fu_g, T_opt_g[0:3,0:3], T_opt_g[0:3,3])
# Initialize the LBFGS optimizer with a line search routine
optimizer = torch.optim.LBFGS([T_opt_g],
                              history_size=10,
                              max_iter=4,
                              line_search_fn="strong_wolfe")
# Keep track of the objective function values over the course of optimization
opt_history_g = []
# Run for a few iterations
for i in range(50):
    optimizer.zero_grad()
    objective = f_opt_g()
    objective.backward()
    optimizer.step(f_opt_g)
    opt_history_g.append(objective.item())
    print('Iter %03d Obj %8.4f' % (i, objective.item()))
```

```
Iter 000 Obj 109.4604
Iter 001 Obj  80.8611
Iter 002 Obj  66.8921
Iter 003 Obj  65.3758
Iter 004 Obj  59.0799
Iter 005 Obj  58.1616
Iter 006 Obj  50.4160
Iter 007 Obj  49.2966
Iter 008 Obj  48.0486
Iter 009 Obj  46.7843
Iter 010 Obj  46.6120
Iter 011 Obj  46.3723
Iter 012 Obj  46.3447
Iter 013 Obj  46.3225
Iter 014 Obj  46.3157
Iter 015 Obj  46.2760
Iter 016 Obj  46.2743
Iter 017 Obj  46.2742
Iter 018 Obj  46.2741
Iter 019 Obj  46.2741
Iter 020 Obj  46.2741
Iter 021 Obj  46.2741
Iter 022 Obj  46.2741
Iter 023 Obj  46.2741
Iter 024 Obj  46.2741
Iter 025 Obj  46.2741
Iter 026 Obj  46.2741
Iter 027 Obj  46.2741
Iter 028 Obj  46.2741
Iter 029 Obj  46.2741
Iter 030 Obj  46.2741
Iter 031 Obj  46.2741
Iter 032 Obj  46.2741
Iter 033 Obj  46.2741
Iter 034 Obj  46.2741
Iter 035 Obj  46.2741
Iter 036 Obj  46.2741
Iter 037 Obj  46.2741
Iter 038 Obj  46.2741
Iter 039 Obj  46.2741
Iter 040 Obj  46.2741
Iter 041 Obj  46.2741
Iter 042 Obj  46.2741
Iter 043 Obj  46.2741
Iter 044 Obj  46.2741
Iter 045 Obj  46.2741
Iter 046 Obj  46.2741
Iter 047 Obj  46.2741
Iter 048 Obj  46.2741
Iter 049 Obj  46.2741
```

In [61]:
```
# plot objective function:
fig, ax = plt.subplots()
plt.plot(range(50), opt_history_g)
plt.title("Gaussian filtered images objective function over iteration of optimization algorithm")
plt.xlabel("iterations")
plt.ylabel("objective function")
plt.show()
```



In [63]:
```
T_fu_opted_g = my_transform_image_pytorch(T_bl, T_fu, T_opt_g[0:3,0:3], T_opt_g[0:3,3],
                                          mode='bilinear', padding_mode='zeros')

my_view(T_fu_opted_g.squeeze().detach().cpu().numpy(), header = hdr_bl,
        xhair=(56,100,64), crange=[0,1000], cmap='gray')

plt.title('torch optimized resliced Gaussian filtered moving image ')
plt.show()
```



In [64]:
```
A_opt_g, b_opt_g = my_pytorch_affine_to_numpy_affine(T_opt_g[0:3,0:3].detach().cpu().numpy(),
                                                     T_opt_g[0:3,3].detach().cpu().numpy(),
                                                     I_bl.shape)
A_opt_g, b_opt_g, A, b
```

Out[64]:
```
(array([[ 1.0201995 , -0.0317349 , -0.0491458 ],
        [ 0.02622579,  1.00564565,  0.06185408],
        [ 0.06429508, -0.15731659,  0.96918139]]),
 array([ 4.4907457 , -5.59184598, 25.80193722]),
 array([[ 1.0003, -0.0254, -0.0409],
        [ 0.0416,  1.001 ,  0.0989],
        [ 0.0537, -0.107 ,  0.9939]]),
 array([ 5.6887, -5.8519, 23.4575]))
```

The registration on images applied with Gaussian filters indeed converge faster. The affine parameters optimized with image that have Gaussian filter applied is not that similar to that given by affine registration in ITK-SNAP and stored in f2b.txt. Yet the plotted difference from `my_view` demonstrates relatively undistinguishable differences.

In [ ]: