In [1]:
```
%matplotlib inline

# some useful imports
import os
import copy
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib import animation
from scipy.interpolate import interp1d
from scipy.signal import convolve2d
from scipy.optimize import curve_fit
import pandas as pd
```

In [2]:
```
mpl.rcParams['font.size'] = 12
mpl.rcParams['axes.labelsize'] = 14
mpl.rcParams['axes.linewidth'] = 1.5

mpl.rcParams.update({'errorbar.capsize': 4})

mpl.rcParams['xtick.top'] = True
mpl.rcParams['xtick.major.width'] = 1.5
mpl.rcParams['ytick.right'] = True
mpl.rcParams['ytick.major.width'] = 1.5
mpl.rcParams['xtick.direction'] = 'in'
mpl.rcParams['ytick.direction'] = 'in'

mpl.rcParams["font.family"] = "Times New Roman"
mpl.rcParams["mathtext.fontset"] = "stix"

plt.rcParams["animation.html"] = "html5"

# set the color of figure background
plt.rcParams.update({
    "figure.facecolor":  (0.0, 0.0, 0.0, 0.0),  # red   with alpha = 30%
    "axes.facecolor":    (1.0, 1.0, 1.0, 1.0),  # green with alpha = 50%
    "savefig.facecolor": (0.0, 0.0, 0.0, 0.0),  # blue  with alpha = 20%
})
```

Forward Radon Transform

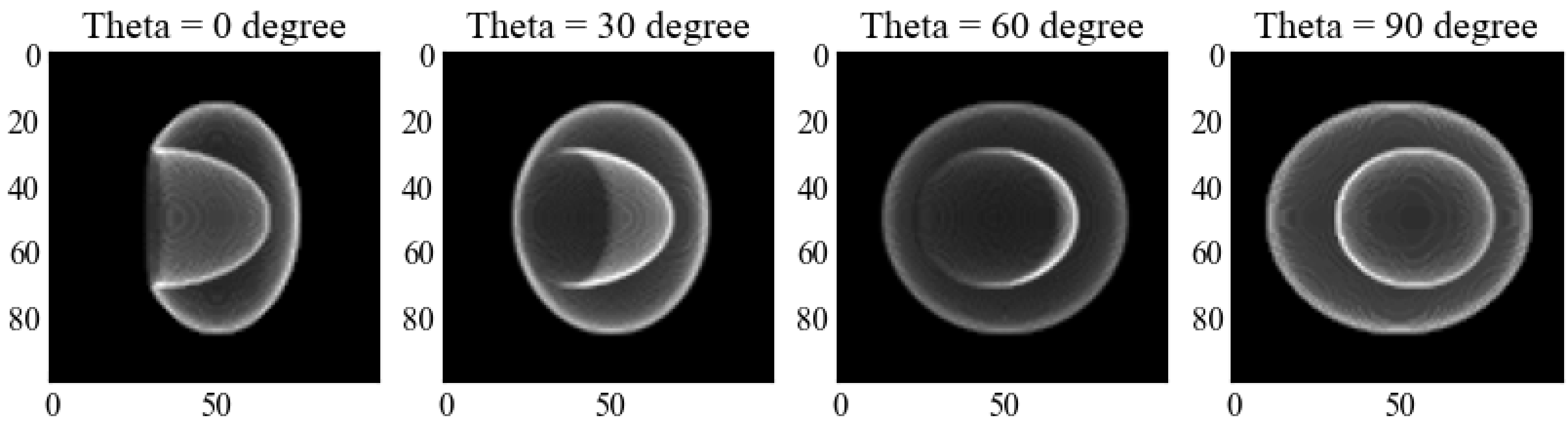
In [3]:
```
# define a function to perform 3D Radon transform
from skimage.transform import radon

def radon3D(image_cube, theta = np.array([90])):
    '''
    Method to calculate 3D radon transform
    image_cube: 3D object with dimension nXmXP
    theta: projection angle in the xy plane
    '''
    image_proj = np.zeros((image_cube.shape[0], image_cube.shape[1]))
    for yi in range(image_cube.shape[0]):
        image_slice = image_cube[yi, ...]
        image_proj[yi, :] = radon(image_slice, theta=theta).reshape((image_cube.shape[1], ))

    return image_proj
```

In [4]:
```
# construct a dummy 3D object

def make_a_ellipse(image_cube = None, R = 30,
                   x_scale_factor = 1,
                   y_scale_factor = 1,
                   z_scale_factor = 1,
                   z_cutoff = False,
                   intensity = 1,
                   bkg_intensity = 0,
                   shape = (100, 100, 100), center = [50, 50, 50]):
    '''
    Method to construct a elliptical object in 3D
    '''
    if image_cube is None:
        image_cube = np.ones(shape)*bkg_intensity


    for i in range(image_cube.shape[0]):
        for j in range(image_cube.shape[1]):
            for k in range(image_cube.shape[2]):
                if z_cutoff:
                    if ((i - center[0])/y_scale_factor)**2 +\
                       ((j - center[1])/x_scale_factor)**2 +\
                       ((k - center[2])/z_scale_factor)**2 < R**2 and k >= center[2]:
                        image_cube[i, j, k] = intensity
                else:
                    if ((i - center[0])/y_scale_factor)**2 +\
                       ((j - center[1])/x_scale_factor)**2 +\
                       ((k - center[2])/z_scale_factor)**2 < R**2:
                        image_cube[i, j, k] = intensity

    return image_cube
```

In [5]:
```
# make an 3D object

shape = (100, 100, 100)
scale_factors = [0.8, 0.7, 0.5]
center =[50, 50, 50]


image_cube = make_a_ellipse(image_cube = None, R = 50,
                            x_scale_factor = scale_factors[0],
                            y_scale_factor = scale_factors[1],
                            z_scale_factor = scale_factors[2],
                            bkg_intensity = 1,
                            intensity = 0.5, shape = shape)
image_cube = make_a_ellipse(image_cube = image_cube, R = 48,
                            x_scale_factor = scale_factors[0],
                            y_scale_factor = scale_factors[1],
                            z_scale_factor = scale_factors[2],
                            center = center,
                            intensity = 0.33, shape = shape)

center_invert = [50, 45, 30]
scale_factors_invert = [0.8, 0.7, 1.2]

image_cube = make_a_ellipse(image_cube = image_cube, R = 30,
                            x_scale_factor = scale_factors_invert[0],
                            y_scale_factor = scale_factors_invert[1],
                            z_scale_factor = scale_factors_invert[2],
                            z_cutoff=False,
                            center = center_invert,
                            intensity = 1, shape = shape)
```

In [6]:
```
image_cube_grad0 = np.gradient(image_cube, axis = 0)
image_cube_grad1 = np.gradient(image_cube, axis = 1)
image_cube_grad2 = np.gradient(image_cube, axis = 2)

image_cube_grad = np.sqrt(image_cube_grad0**2 + image_cube_grad1**2 + image_cube_grad2**2)
```

In [7]:
```
image_projs = dict()
image_projs['angle_list'] = [0, 30,60, 90]
fig, axes = plt.subplots(1, len(image_projs['angle_list']), figsize=(10, 4.5))

image_projs['image_list'] = tuple()
for ai, angle in enumerate(image_projs['angle_list']):
    image_projs['image_list'] += (radon3D(image_cube_grad, theta = np.array([angle])), )

    axes[ai].imshow(image_projs['image_list'][-1], cmap = 'gray')
    axes[ai].set_title(f"Theta = {angle} degree")
```



Inverse Radon Transform (Filtered Back Projection)

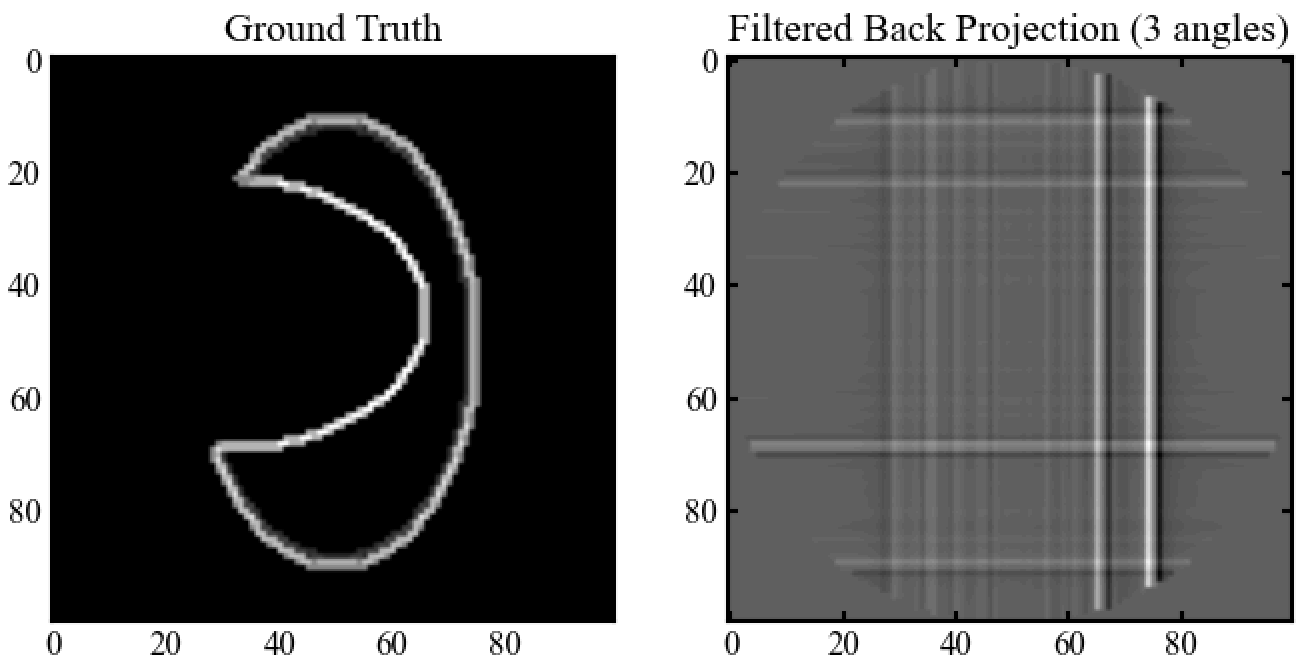
In [8]:
```
from skimage.transform import iradon

def iradon3d_fbp(image_projs, filter_type = 'ramp'):
    '''
    Method to recover 3D image cube using inverse radon transform

    '''

    num_layer = image_projs['image_list'][0].shape[0]
    layer_width = image_projs['image_list'][0].shape[1]
    num_anlge = len(image_projs['angle_list'])

    for li in range(num_layer):
        sg = np.zeros((layer_width, num_anlge))
        for ai in range(num_anlge):
            sg[:, ai] = image_projs['image_list'][ai][li, ...].reshape((layer_width, ))

        fbp = iradon(sg, theta=image_projs['angle_list'], filter_name=filter_type)
        if li == 0:
            image_cube = fbp.reshape((1, *fbp.shape)).copy()
        else:
            image_cube = np.vstack((image_cube, fbp.reshape((1, *fbp.shape))))

    return image_cube
```

In [9]:
```
image_projs = dict()
image_projs['angle_list'] = np.linspace(0, 180, 3) #[0, 30,60, 90]
# fig, axes = plt.subplots(1, len(angle_list), figsize=(10, 4.5))

image_projs['image_list'] = tuple()
for ai, angle in enumerate(image_projs['angle_list']):
    image_projs['image_list'] += (radon3D(image_cube_grad, theta = np.array([angle])), )

#     axes[ai].imshow(image_projs['image_list'][-1], cmap = 'gray')
#     axes[ai].set_title(f"Theta = {angle} degree")
```

In [10]:
```
image_cube_fbp = iradon3d_fbp(image_projs, filter_type = 'ramp')
```

In [1125]:
```
fig, axes = plt.subplots(1, 2, figsize=(8, 4.5))

axes[0].imshow(image_cube_grad[50, ...], cmap = 'gray')
axes[0].set_title('Ground Truth')

axes[1].imshow(image_cube_fbp[50, ...], cmap = 'gray')
axes[1].set_title(f"Filtered Back Projection ({len(image_projs['angle_list'])} angles)")
```

Out[1125]:
```
Text(0.5, 1.0, 'Filtered Back Projection (3 angles)')
```



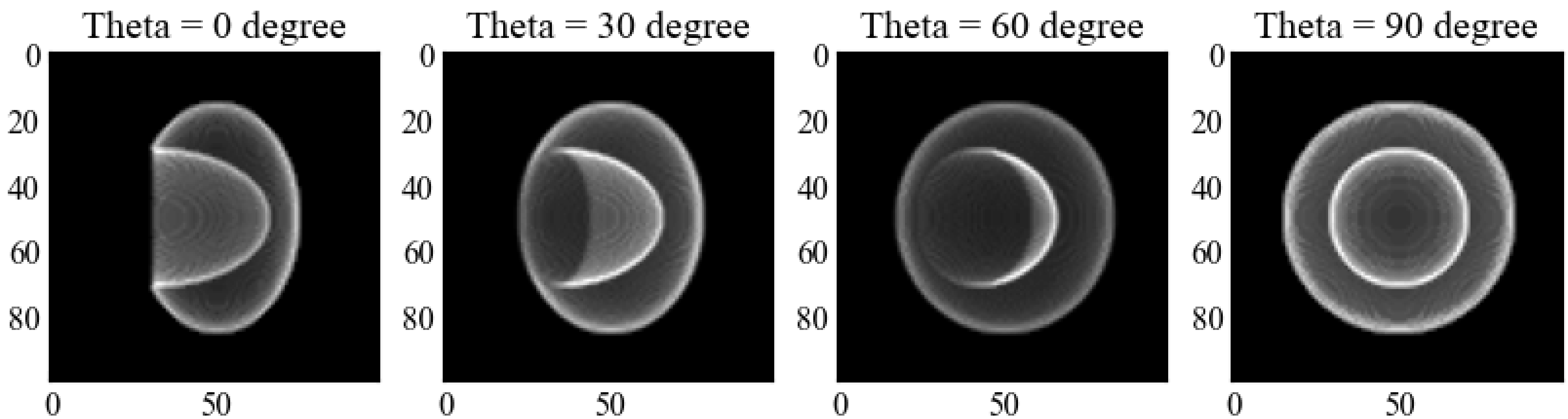
Algebric Method via Iterative Solver

In [12]:
```
from skimage.transform import iradon_sart

def iradon3d_sart(image_projs, prior = None):
    '''
    Method to recover 3D image cube using inverse radon transform

    '''

    num_layer = image_projs['image_list'][0].shape[0]
    layer_width = image_projs['image_list'][0].shape[1]
    num_anlge = len(image_projs['angle_list'])

    for li in range(num_layer):
        sg = np.zeros((layer_width, num_anlge))
        for ai in range(num_anlge):
            sg[:, ai] = image_projs['image_list'][ai][li, ...].reshape((layer_width, ))

        if prior is None:
            sart = iradon_sart(sg, theta=image_projs['angle_list'])
        else:
            sart = iradon_sart(sg, theta=image_projs['angle_list'], image=prior[li, ...])
        if li == 0:
            image_cube = sart.reshape((1, *sart.shape)).copy()
        else:
            image_cube = np.vstack((image_cube, sart.reshape((1, *sart.shape))))

    return image_cube
```

In [13]:
```
# make an 3D object

shape = (100, 100, 100)
scale_factors = [0.7, 0.7, 0.5]
center =[50, 50, 50]


image_cube_sym = make_a_ellipse(image_cube = None, R = 50,
                            x_scale_factor = scale_factors[0],
                            y_scale_factor = scale_factors[1],
                            z_scale_factor = scale_factors[2],
                            bkg_intensity = 1,
                            intensity = 0.5, shape = shape)
image_cube_sym = make_a_ellipse(image_cube = image_cube_sym, R = 48,
                            x_scale_factor = scale_factors[0],
                            y_scale_factor = scale_factors[1],
                            z_scale_factor = scale_factors[2],
                            center = center,
                            intensity = 0.33, shape = shape)

center_invert = [50, 50, 30]
scale_factors_invert = [0.7, 0.7, 1.2]

image_cube_sym = make_a_ellipse(image_cube = image_cube_sym, R = 30,
                            x_scale_factor = scale_factors_invert[0],
                            y_scale_factor = scale_factors_invert[1],
                            z_scale_factor = scale_factors_invert[2],
                            z_cutoff=False,
                            center = center_invert,
                            intensity = 1, shape = shape)
```

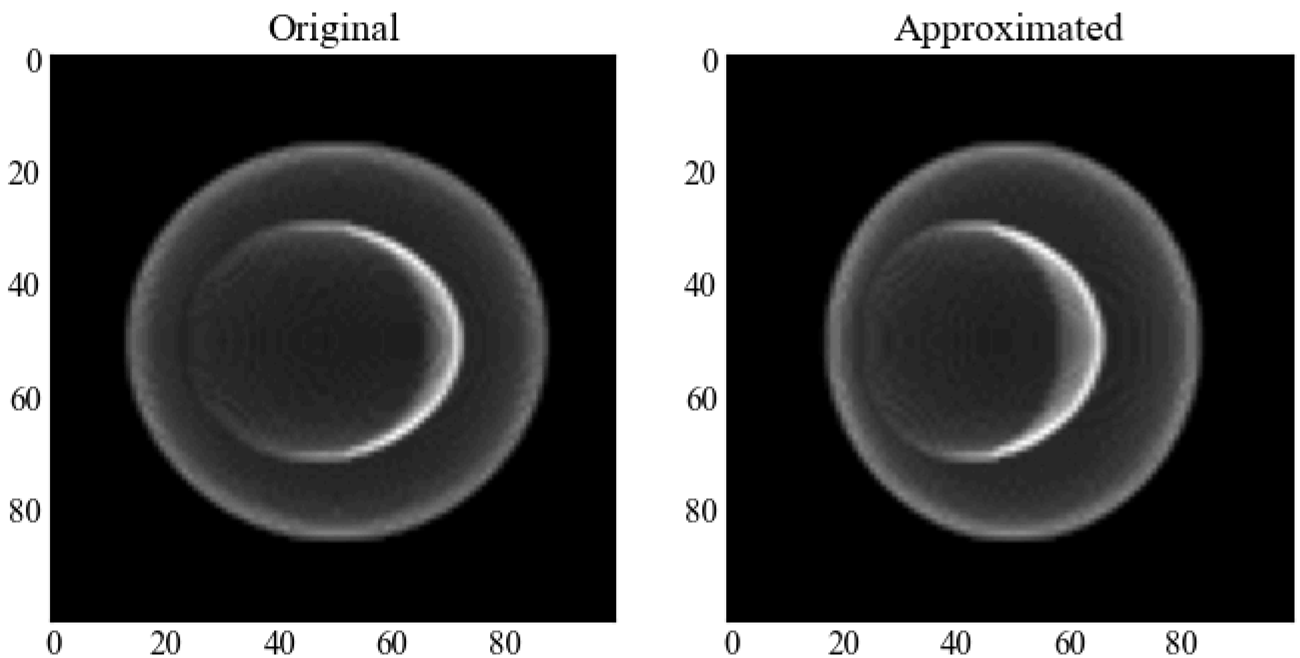
In [14]:
```
image_cube_grad_sym0 = np.gradient(image_cube_sym, axis = 0)
image_cube_grad_sym1 = np.gradient(image_cube_sym, axis = 1)
image_cube_grad_sym2 = np.gradient(image_cube_sym, axis = 2)

image_cube_grad_sym = np.sqrt(image_cube_grad_sym0**2 + image_cube_grad_sym1**2 + image_cube_grad_sym2**2)
```

In [15]:
```
image_projs = dict()
image_projs['angle_list'] = [0, 30,60, 90]
fig, axes = plt.subplots(1, len(image_projs['angle_list']), figsize=(10, 4.5))

image_projs['image_list'] = tuple()
for ai, angle in enumerate(image_projs['angle_list']):
    image_projs['image_list'] += (radon3D(image_cube_grad_sym, theta = np.array([angle])), )

    axes[ai].imshow(image_projs['image_list'][-1], cmap = 'gray')
    axes[ai].set_title(f"Theta = {angle} degree")
```



In [16]:
```
# construct a axial symmetric rotation image prior
from skimage.filters import threshold_otsu
from skimage.filters import gaussian

angle_compare = 60

side_proj = radon3D(image_cube_grad, theta = np.array([angle_compare]))
side_proj_sym = radon3D(image_cube_grad_sym, theta = np.array([angle_compare]))

side_proj_bin = gaussian(side_proj>threshold_otsu(side_proj)*3, 0.75, preserve_range=False)

fig, axes = plt.subplots(1, 2, figsize=(8, 4.5))

axes[0].imshow(side_proj, cmap = 'gray')
axes[1].imshow(side_proj_sym, cmap = 'gray')

axes[0].set_title('Original')
axes[1].set_title('Approximated')
```

Out[16]:
```
Text(0.5, 1.0, 'Approximated')
```

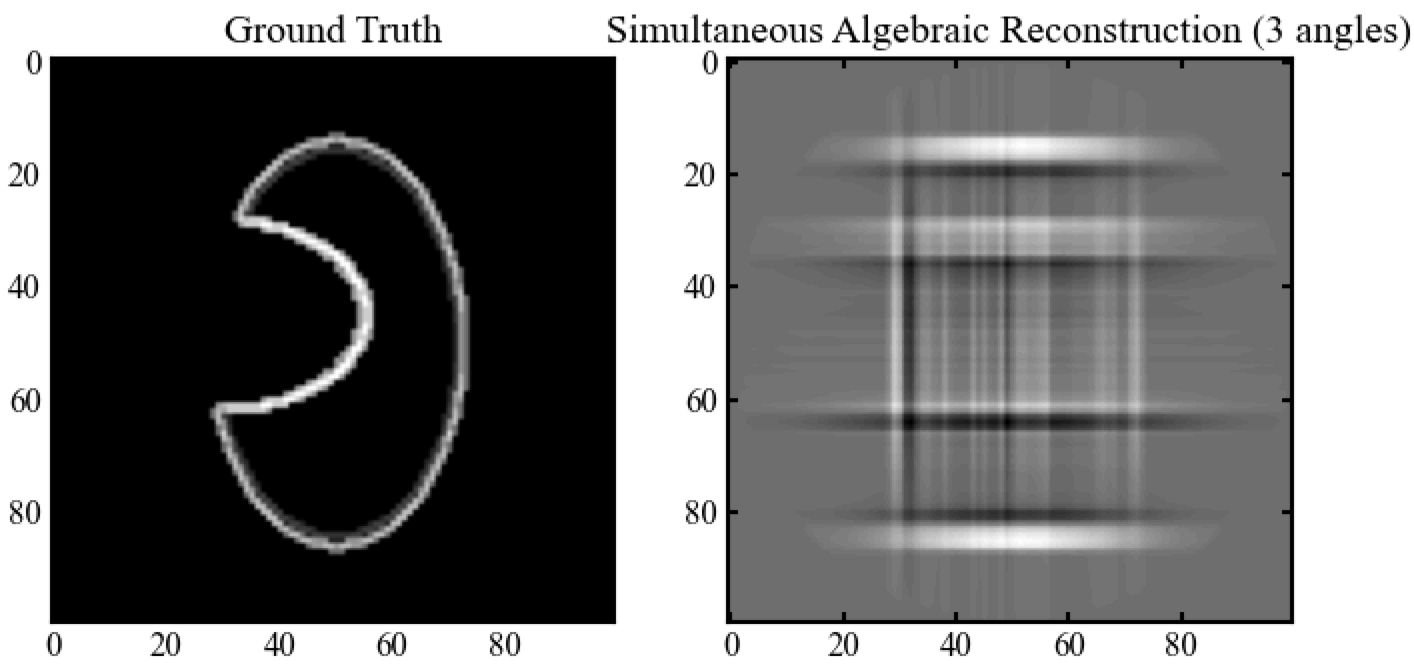


In [17]:
```
image_projs = dict()
image_projs['angle_list'] = np.linspace(0, 180, 3) #[0, 30,60, 90]
# fig, axes = plt.subplots(1, len(angle_list), figsize=(10, 4.5))

image_projs['image_list'] = tuple()
for ai, angle in enumerate(image_projs['angle_list']):
    image_projs['image_list'] += (radon3D(image_cube_grad, theta = np.array([angle])), )

#     axes[ai].imshow(image_projs['image_list'][-1], cmap = 'gray')
#     axes[ai].set_title(f"Theta = {angle} degree")
```

In [18]:
```
image_cube_sart = iradon3d_sart(image_projs, prior = image_cube_grad_sym.copy())
```

In [19]:
```
fig, axes = plt.subplots(1, 2, figsize=(8, 4.5))

layger_to_probe = 35

subtracted = image_cube_sart[layger_to_probe, ...] - image_cube_grad_sym[layger_to_probe, ...]
# subtracted = subtracted>threshold_otsu(subtracted)*1.2

axes[0].imshow(image_cube_grad[layger_to_probe, ...], cmap = 'gray')
axes[0].set_title('Ground Truth')

# axes[1].imshow(subtracted, cmap = 'gray')
axes[1].imshow(subtracted, cmap ='gray')
axes[1].set_title(f"Simultaneous Algebraic Reconstruction ({len(image_projs['angle_list'])} angles)")
```

Out[19]:
```
Text(0.5, 1.0, 'Simultaneous Algebraic Reconstruction (3 angles)')
```



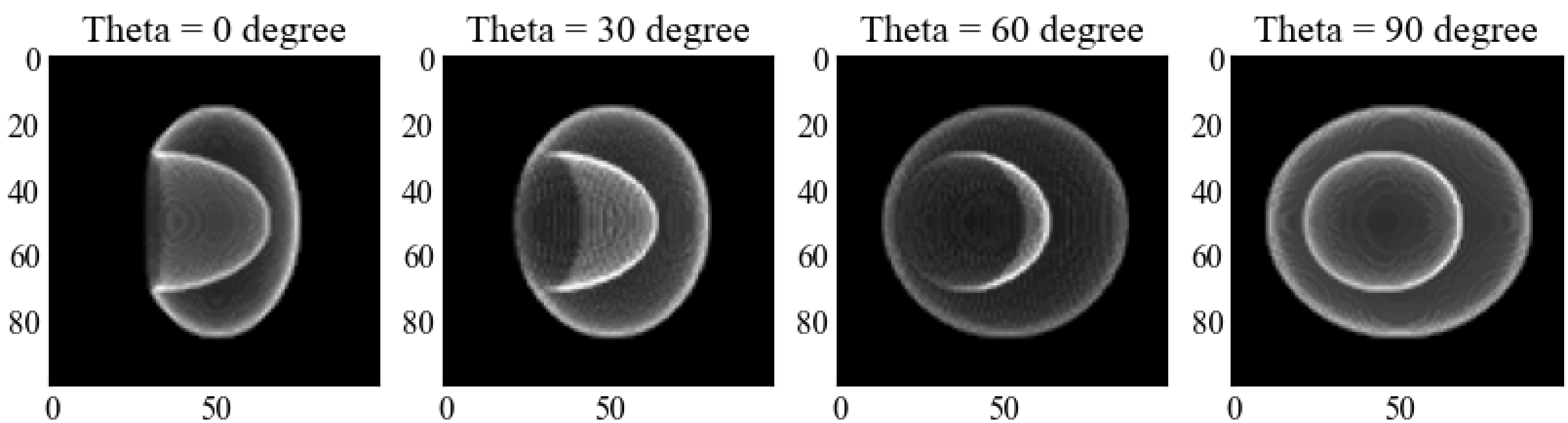
Adam Solver + Axially Symmetric Regularizer

In [311]:
```
import torch
import torchvision.transforms.functional as TF

def radon_torch(image, angle):
    """
    Compute the Radon transform of an image for a given angle.

    Args:
        image (torch.Tensor): Input image.
        angle (float): Angle (in degrees) for the Radon transform.

    Returns:
        torch.Tensor: Radon transform (sinogram) of the input image.
    """
    # Convert angle to radians
    theta = torch.tensor([np.deg2rad(angle)])

    # Apply rotation to the image
    try:
        rotated_image = TF.rotate(image, angle)
    except:
        rotated_image = TF.rotate(image.reshape((1, *image.shape)), angle)

    # Compute the projection (sum along columns)
    projection = torch.sum(rotated_image, dim=1)

    return projection

def radon3D_torch(image_cube_torch, theta = 90):
    '''
    Method to calculate 3D radon transform
    image_cube: 3D object with dimension nXmXP
    theta: projection angle in the xy plane
    '''
    image_proj_torch = torch.zeros((image_cube_torch.shape[0], image_cube_torch.shape[1]))
    for yi in range(image_cube_torch.shape[0]):
        image_slice_torch = image_cube_torch[yi, ...]
        image_proj_torch[yi, :] = radon_torch(image_slice_torch, angle=theta).reshape((image_cube_torch.shape[

    return image_proj_torch

def gaussian_fun(x, mu, sigma):
```

```
    return 1/(sigma * np.sqrt(2 * np.pi)) *np.exp( - (x - mu)**2 / (2 * sigma**2))
```

In [312]:
```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# convert the numpy arrays to pytorch tensors
image_cube_grad_torch = torch.from_numpy(image_cube_grad).to(device)
image_cube_grad_sym_torch = torch.from_numpy(image_cube_grad_sym).to(device)
```

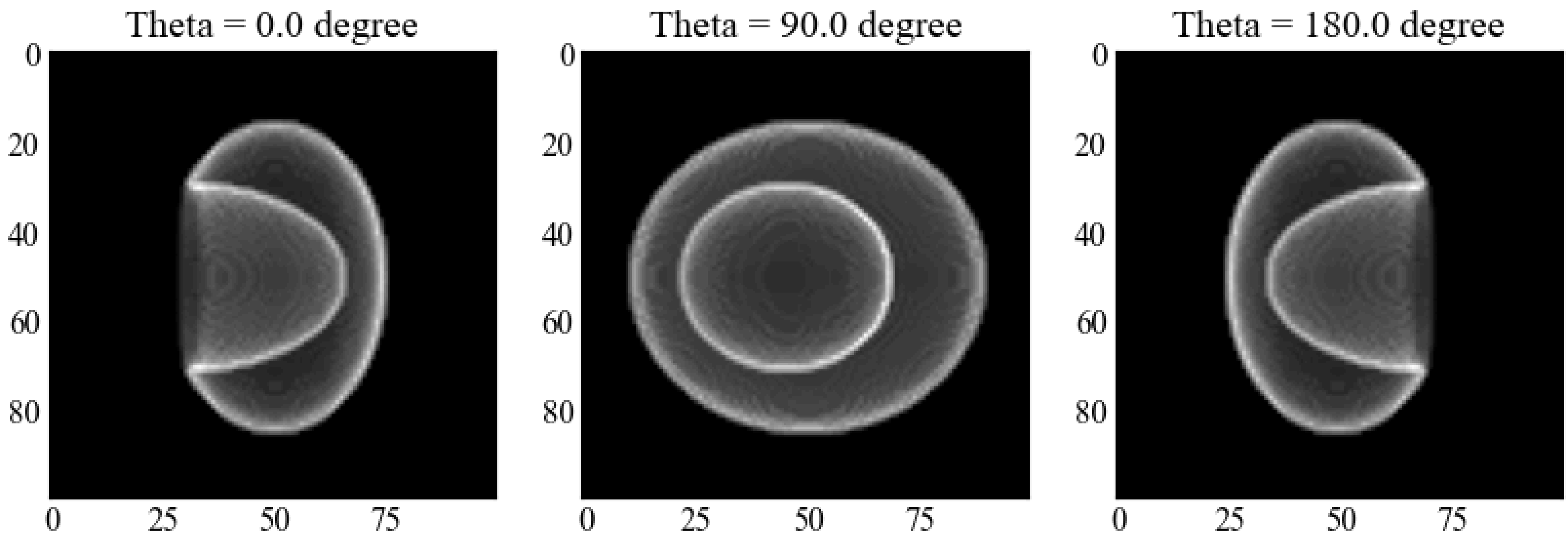
In [313]:
```
image_projs_torch = dict()
image_projs_torch['angle_list'] = [0, 30,60, 90]
fig, axes = plt.subplots(1, len(image_projs_torch['angle_list']), figsize=(10, 4.5))

image_projs_torch['image_list'] = tuple()
for ai, angle in enumerate(image_projs_torch['angle_list']):
    image_projs_torch['image_list'] += (radon3D_torch(image_cube_grad_torch, theta = angle), )

    axes[ai].imshow(image_projs_torch['image_list'][-1], cmap = 'gray')
    axes[ai].set_title(f"Theta = {angle} degree")
```



In [314]:
```
from tqdm import tqdm
import torch

def iradon_adam(image_projs_torch, image_cube_reg, reg_angles = np.linspace(0, 180, 15),
                lam = np.ones((15, )), num_iters =75, learning_rate=5e-2):

    # check if GPU is available, otherwise use CPU
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

    # create regularizer images
    reg_list = tuple()
    for reg_angle in reg_angles:
        reg_list += (radon3D_torch(image_cube_reg, theta = reg_angle), )

    # initialize the solution
    x = torch.zeros_like(image_cube_reg,
                         requires_grad=True).to(device) #np.zeros_like(image_cube_reg)

    # initialize Adam optimizer
    optim = torch.optim.Adam(params=[x], lr=learning_rate)

    for it in tqdm(range(num_iters)):

        # set all gradients of the computational graph to 0
        optim.zero_grad()

        # this term computes the data fidelity term of the loss function
        loss_data = 0
        for ai, theta in enumerate(image_projs_torch['angle_list']):

            loss_data += (radon3D_torch(x, theta = theta) -\
                          image_projs_torch['image_list'][ai]).pow(2).sum()

        # regularizer term
        loss_regularizer = 0
        for ai, theta in enumerate(reg_angles):
            loss_regularizer += lam[ai] *(radon3D_torch(x, theta = theta) - reg_list[ai]).pow(2).sum()

        # compute weighted sum of data fidelity and regularization term
        loss = loss_data +  loss_regularizer

        # compute backwards pass
        loss.backward()
```

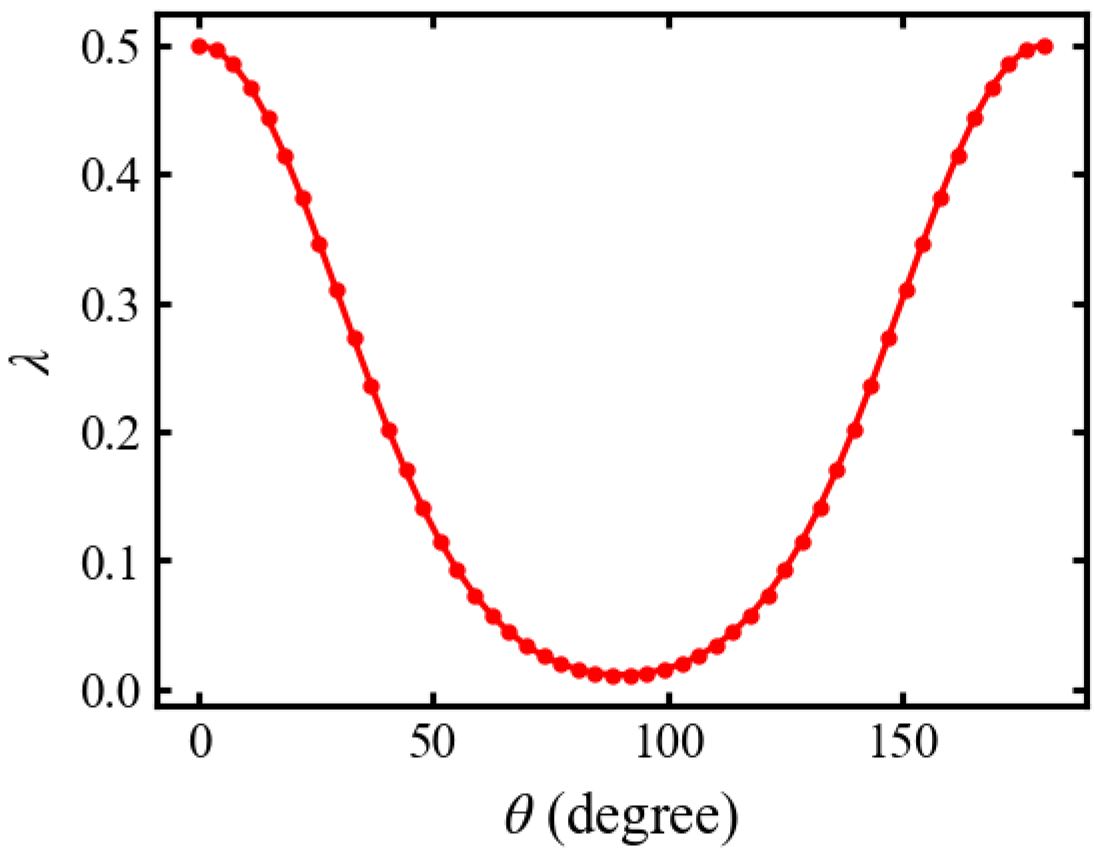
```
        # take a step with the Adam optimizer
        optim.step()

    # return the result as a numpy array
    return x.detach().cpu().numpy()
```

In [315]:
```
image_projs_torch = dict()
image_projs_torch['angle_list'] = np.linspace(0, 180, 3)
fig, axes = plt.subplots(1, len(image_projs_torch['angle_list']), figsize=(10, 4.5))

image_projs_torch['image_list'] = tuple()
for ai, angle in enumerate(image_projs_torch['angle_list']):
    image_projs_torch['image_list'] += (radon3D_torch(image_cube_grad_torch, theta = angle), )

    axes[ai].imshow(image_projs_torch['image_list'][-1], cmap = 'gray')
    axes[ai].set_title(f"Theta = {angle} degree")
```



In [446]:
```
reg_angles = np.linspace(0, 180, 50)
lam_gaussian = gaussian_fun(reg_angles, 0, 30)  + gaussian_fun(reg_angles, 180, 30)
lam_gaussian = 0.5*lam_gaussian/lam_gaussian.max()

fig, ax = plt.subplots(dpi = 150, figsize=(4, 3))

ax.plot(reg_angles, lam_gaussian, 'r.-')
ax.set_xlabel(r'$\theta$ (degree)')
ax.set_ylabel(r'$\lambda$')
```

Out[446]:
```
Text(0, 0.5, '$\\lambda$')
```

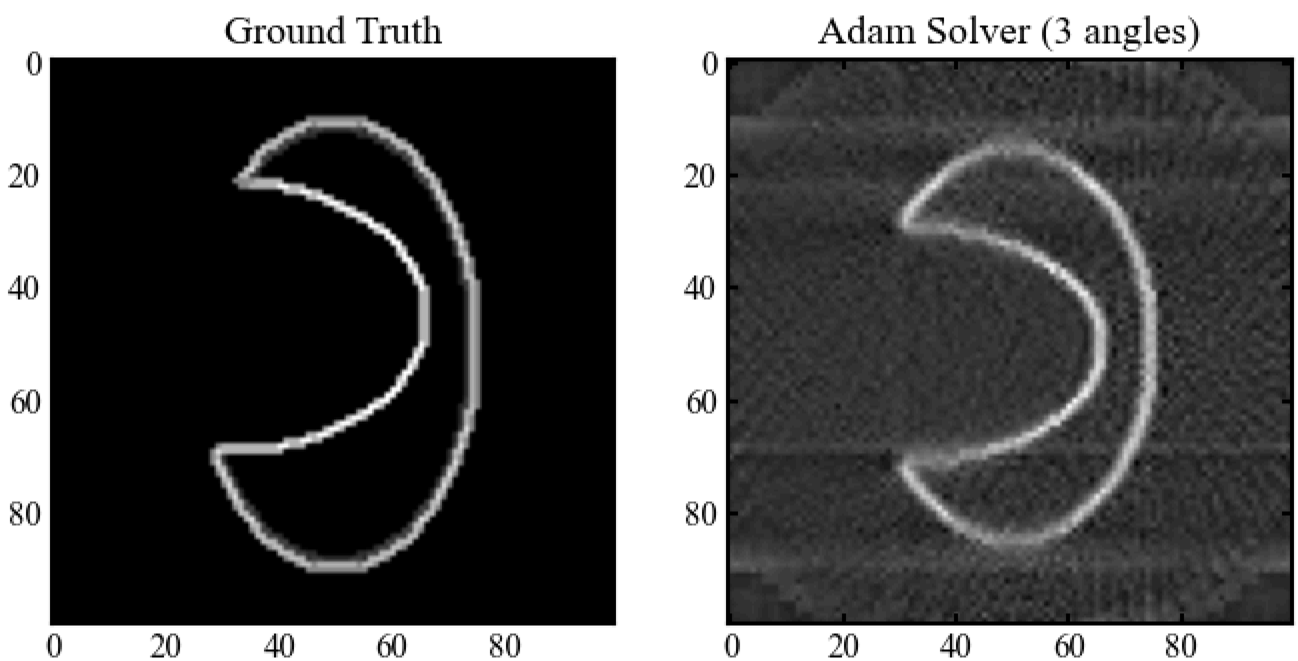


In [447]:
```
image_cube_adam = iradon_adam(image_projs_torch, image_cube_grad_sym_torch,
                              reg_angles = reg_angles,
                              lam = lam_gaussian, num_iters =75, learning_rate=5e-2)
```

```
100%|██████████| 75/75 [10:49<00:00,  8.66s/it]
```

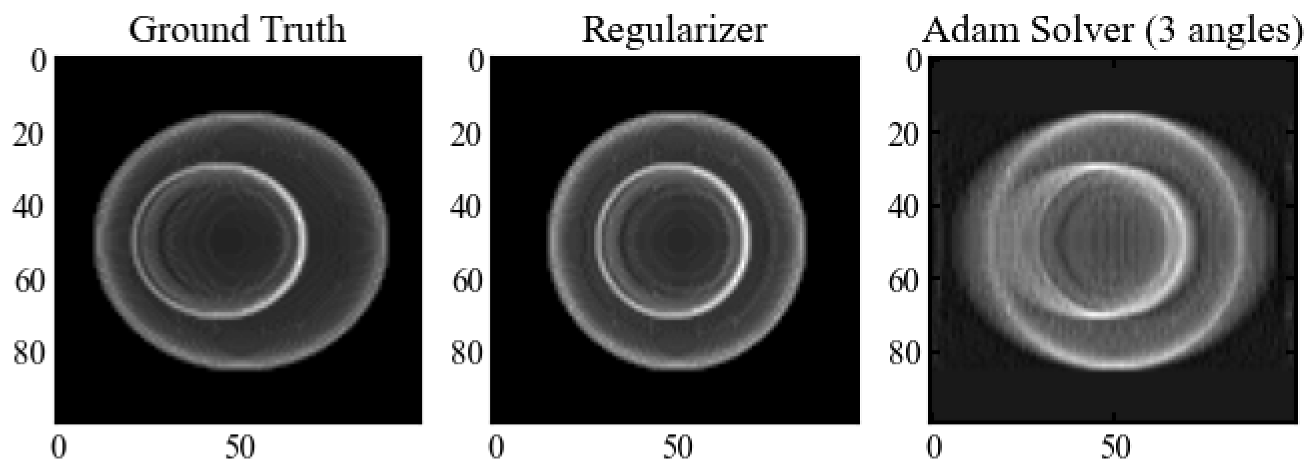
In [448]:
```
fig, axes = plt.subplots(1, 2, figsize=(8, 4.5))

layger_to_probe = 50

axes[0].imshow(image_cube_grad[layger_to_probe, ...], cmap = 'gray')
axes[0].set_title('Ground Truth')

# axes[1].imshow(subtracted, cmap = 'gray')
axes[1].imshow(image_cube_adam[layger_to_probe, ...], cmap ='gray')
axes[1].set_title(f"Adam Solver ({len(image_projs['angle_list'])} angles)")
```

Out[448]:
```
Text(0.5, 1.0, 'Adam Solver (3 angles)')
```



In [463]:
```
fig, axes = plt.subplots(1, 3, figsize=(8, 4.5))
theta_to_probe = 85

axes[0].imshow(radon3D_torch(image_cube_grad_torch, theta = theta_to_probe), cmap = 'gray')
axes[0].set_title('Ground Truth')

axes[1].imshow(radon3D_torch(image_cube_grad_sym_torch, theta = theta_to_probe), cmap = 'gray')
axes[1].set_title('Regularizer')

# axes[1].imshow(subtracted, cmap = 'gray')
axes[2].imshow(radon3D_torch(torch.from_numpy(image_cube_adam), theta = theta_to_probe), cmap ='gray')
axes[2].set_title(f"Adam Solver ({len(image_projs['angle_list'])} angles)")
```

Out[463]:
```
Text(0.5, 1.0, 'Adam Solver (3 angles)')
```

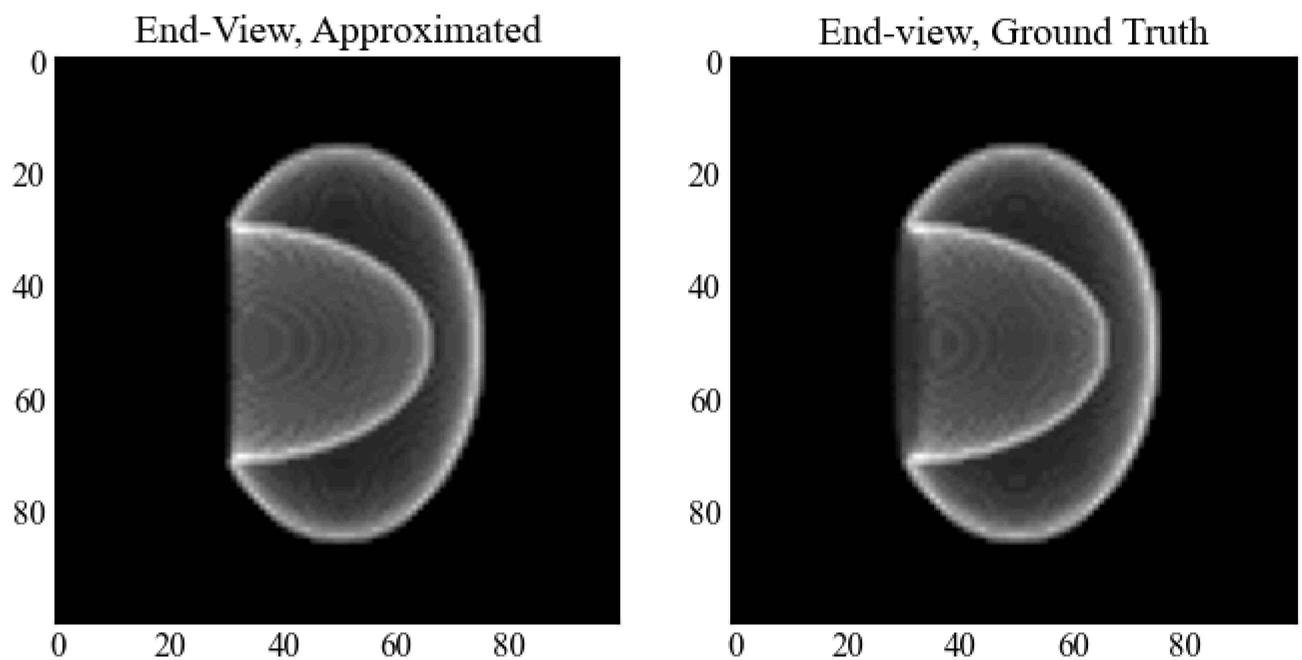


Adam Solver + Morphing-based view interpolation

In [1116]:
```
from skimage.registration import optical_flow_tvl1, optical_flow_ilk
from skimage.transform import ProjectiveTransform
from skimage.transform import warp
```

In [1117]:
```
end_view = radon3D(image_cube_grad, theta = np.array([90]))
end_view_torch = radon3D_torch(image_cube_grad_torch, theta = 0)

end_view_sym = radon3D(image_cube_grad_sym, theta = np.array([90]))
end_view_sym_torch = radon3D_torch(image_cube_grad_sym_torch, theta = 0)

fig, axes = plt.subplots(1, 2, figsize=(8, 4.5))

axes[0].imshow(end_view_sym_torch, cmap = 'gray')
axes[0].set_title('End-View, Approximated')

axes[1].imshow(end_view_torch, cmap = 'gray')
axes[1].set_title('End-view, Ground Truth')
```

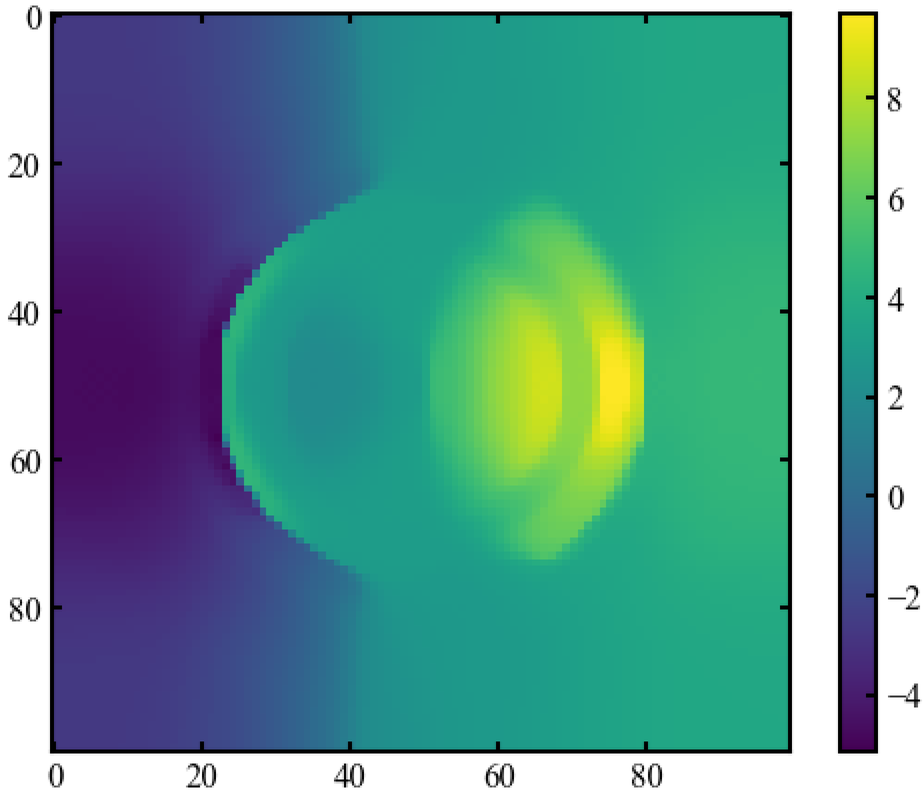
Out[1117]:
```
Text(0.5, 1.0, 'End-view, Ground Truth')
```



In [1118]:
```
u, v =optical_flow_tvl1(gaussian(end_view_sym, 4), gaussian(end_view, 4))
```

In [1119]:
```
# plt.imshow(np.sqrt(u**2 + v**2))
plt.imshow((v))
plt.colorbar()
```

Out[1119]: <matplotlib.colorbar.Colorbar at 0x219c3250e80>



In [1120]:
```
skip = 10

src = np.array([0, 0])
dst = np.array([0, 0])
for i in np.arange(skip, end_view_sym.shape[0]+skip, skip):
    for j in np.arange(skip, end_view_sym.shape[1]+skip, skip):
        src = np.vstack((src, np.array([j-1, i-1])))
        dst = np.vstack((dst, np.array([j-1 + v[i-1, j-1]*1.5, i-1 + u[i-1, j-1]])))
```

In [1121]:
```
skip = 10

src_torch = np.array([0, 0])
dst_torch = np.array([0, 0])
for i in np.arange(skip, end_view_sym.shape[0]+skip, skip):
    for j in np.arange(skip, end_view_sym.shape[1]+skip, skip):
        src_torch = np.vstack((src_torch, np.array([j-1, i-1])))
        dst_torch = np.vstack((dst_torch, np.array([j-1 + np.flip(v)[i-1, j-1]*-1.5, i-1 + u[i-1, j-1]])))
```

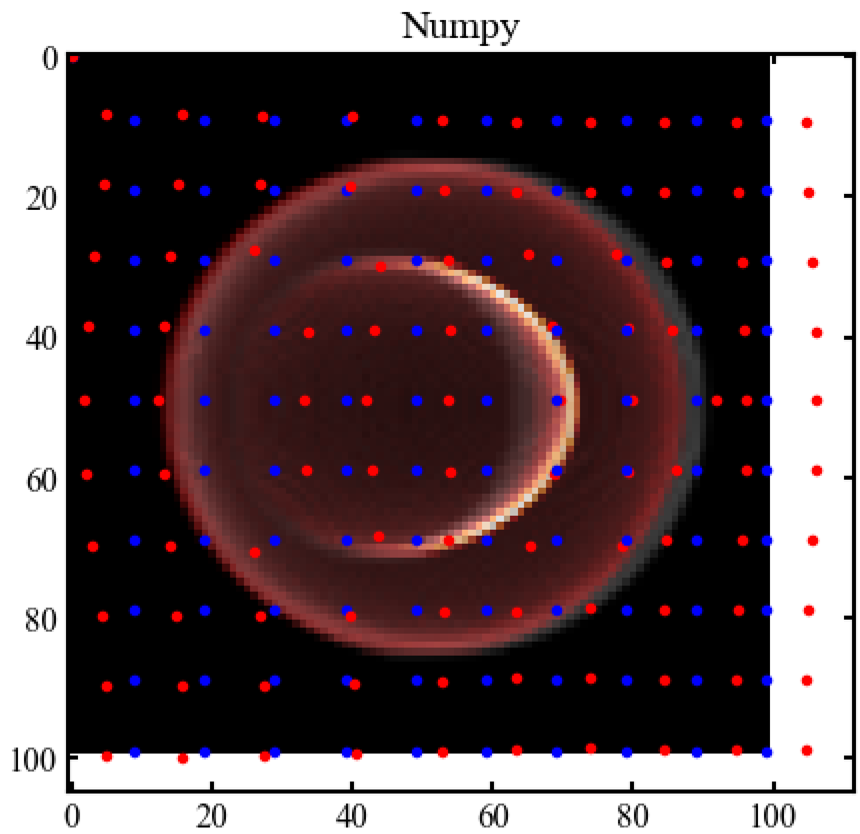
In [807]:
```
tformEW = ProjectiveTransform()
tformEW.estimate(dst, src)
warped = warp(end_view_sym, tformEW, output_shape=end_view_sym.shape)
```

In [808]:
```
tformEW_torch = ProjectiveTransform()
tformEW_torch.estimate(dst_torch, src_torch)
warped_torch = warp(end_view_sym_torch, tformEW_torch, output_shape=end_view_sym.shape)
```

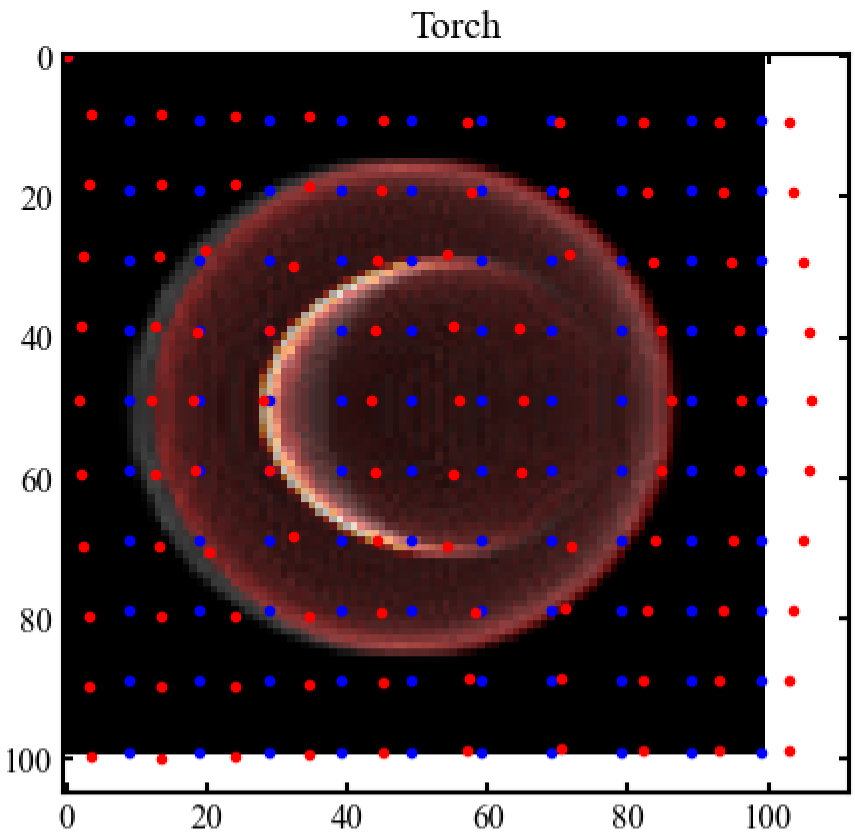
In [809]:
```
test_gt = radon3D(image_cube_grad, theta = np.array([60]))
test_sym = radon3D(image_cube_grad_sym, theta = np.array([60]))

test_interp = warp(test_sym, tformEW)#, output_shape=test_sym.shape)
```

In [810]:
```
test_gt_torch = radon3D_torch(image_cube_grad_torch, theta = 120).numpy()
test_sym_torch = radon3D_torch(image_cube_grad_sym_torch, theta = 120).numpy()

test_interp_torch = warp(test_sym_torch, tformEW_torch)#, output_shape=test_sym.shape)
```

In [811]:
```
plt.imshow(test_gt, cmap = 'gist_heat')
# plt.imshow(test_sym, alpha = 0.5, cmap = 'gray')
plt.imshow(test_interp, alpha = 0.5, cmap = 'gray')
plt.title('Numpy')

for si in range(len(src)):
    plt.plot(src[si, 0], src[si, 1], 'b.')
    plt.plot(dst[si, 0], dst[si, 1], 'r.')
```

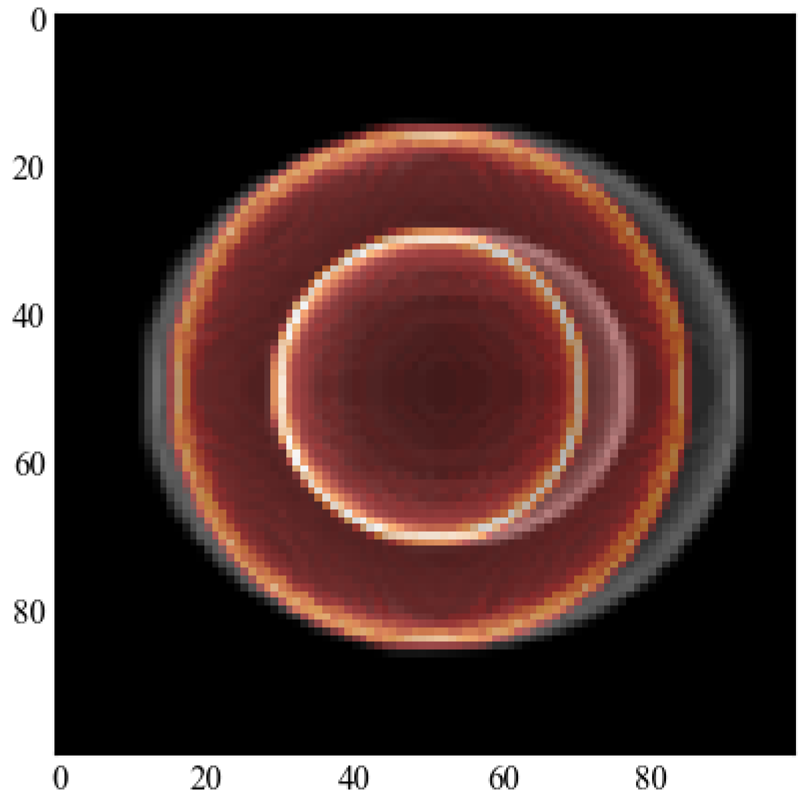


In [812]:
```
plt.imshow(test_gt_torch, cmap = 'gist_heat')
# plt.imshow(test_sym, alpha = 0.5, cmap = 'gray')
plt.imshow(test_interp_torch, alpha = 0.5, cmap = 'gray')
plt.title('Torch')

for si in range(len(src_torch)):
    plt.plot(src_torch[si, 0], src_torch[si, 1], 'b.')
    plt.plot(dst_torch[si, 0], dst_torch[si, 1], 'r.')
```

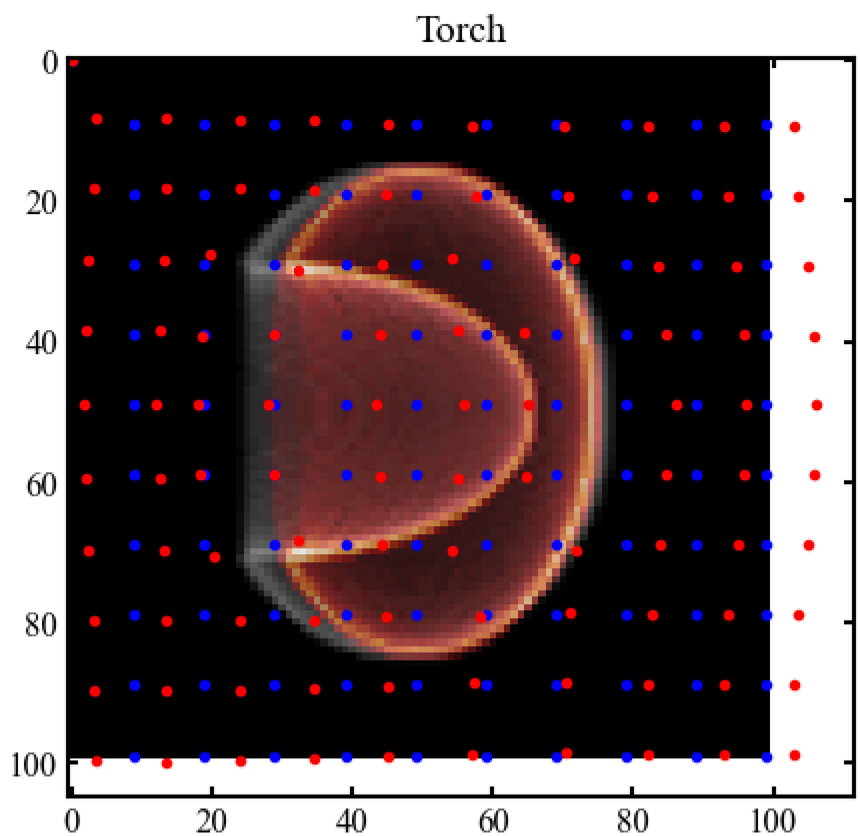


In [1122]:
```
plt.imshow(end_view, cmap = 'gist_heat')
plt.imshow(end_view_sym, cmap = 'gist_heat')

plt.imshow(warped, alpha = 0.5, cmap = 'gray')
# plt.title('Numpy')
# for si in range(len(src)):
#     plt.plot(src[si, 0], src[si, 1], 'b.')
#     plt.plot(dst[si, 0], dst[si, 1], 'r.')
```

Out[1122]: <matplotlib.image.AxesImage at 0x219c34acfd0>

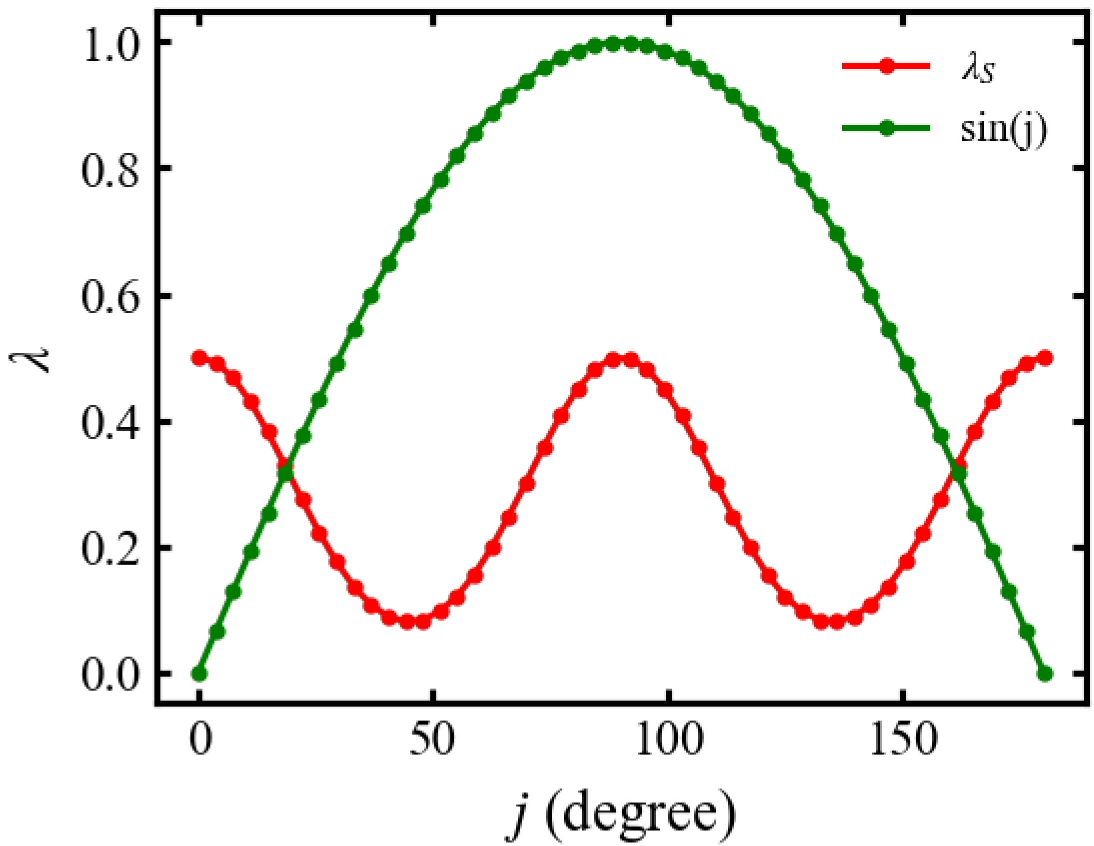


In [814]:
```
plt.imshow(end_view_torch, cmap = 'gist_heat')
plt.imshow(warped_torch, alpha = 0.5, cmap = 'gray')
plt.title('Torch')
for si in range(len(src_torch)):
    plt.plot(src_torch[si, 0], src_torch[si, 1], 'b.')
    plt.plot(dst_torch[si, 0], dst_torch[si, 1], 'r.')
```



In [1124]:
```
reg_angles = np.linspace(0, 180, 50)
lam_gaussian = gaussian_fun(reg_angles, 0, 20) + \
               gaussian_fun(reg_angles, 90, 20) + \
               gaussian_fun(reg_angles, 180, 20)
lam_gaussian = 0.5*lam_gaussian/lam_gaussian.max()

# warp_gaussian = gaussian_fun(reg_angles, 90, 30)
# warp_gaussian = warp_gaussian/warp_gaussian.max()

warp_sine = np.sin(reg_angles/180*np.pi)
# warp_sine = warp_gaussian/warp_gaussian.max()


fig, ax = plt.subplots(dpi = 150, figsize=(4, 3))

ax.plot(reg_angles, lam_gaussian, 'r.-', label = r'$\lambda_{S}$')
# ax.plot(reg_angles, warp_gaussian, 'g.-', label = 'End View')
ax.plot(reg_angles, warp_sine, 'g.-', label = 'sin(j)')

# ax.plot(reg_angles_SW, lam_gaussian_SW, 'r.-', label = 'Side View')
ax.set_xlabel(r'$j$ (degree)')
ax.set_ylabel(r'$\lambda$')
ax.legend(frameon = False, fontsize = 10)
```

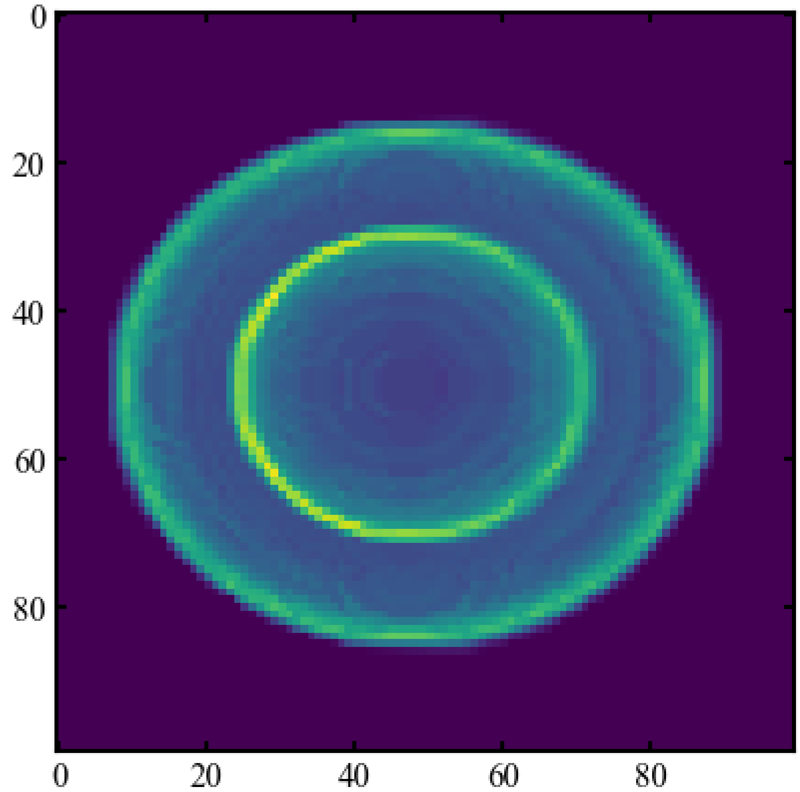
Out[1124]:
```
<matplotlib.legend.Legend at 0x219c350a140>
```


1.0
0.8
0.6
0.4
0.2
0.0
λ
0
50
100
150
j (degree)
λ_S
sin(j)

In [942]:
```
# check if GPU is available, otherwise use CPU
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# create regularizer images

reg_list = tuple()
for ri, reg_angle in enumerate(reg_angles):

    skip = 10

    src = np.array([0, 0])
    dst = np.array([0, 0])
    for i in np.arange(skip, end_view_sym.shape[0]+skip, skip):
        for j in np.arange(skip, end_view_sym.shape[1]+skip, skip):
            src = np.vstack((src, np.array([j-1, i-1])))
            dst = np.vstack((dst, np.array([j-1 + np.flip(v)[i-1, j-1]*-1.5*warp_sine[ri],
                                                i-1 + u[i-1, j-1]*warp_sine[ri]])))
#     print(warp_sine[ri])
    tformEW_torch = ProjectiveTransform()
    tformEW_torch.estimate(dst, src)

    reg = radon3D_torch(image_cube_grad_sym_torch, theta = reg_angle)
    reg = warp(reg.numpy(), tformEW_torch, output_shape=reg_list_SW[-1].shape)
    reg_list += (torch.from_numpy(reg), )
```

In [945]:
```
plt.imshow(reg_list[25])
```

Out[945]: <matplotlib.image.AxesImage at 0x219aff45450>



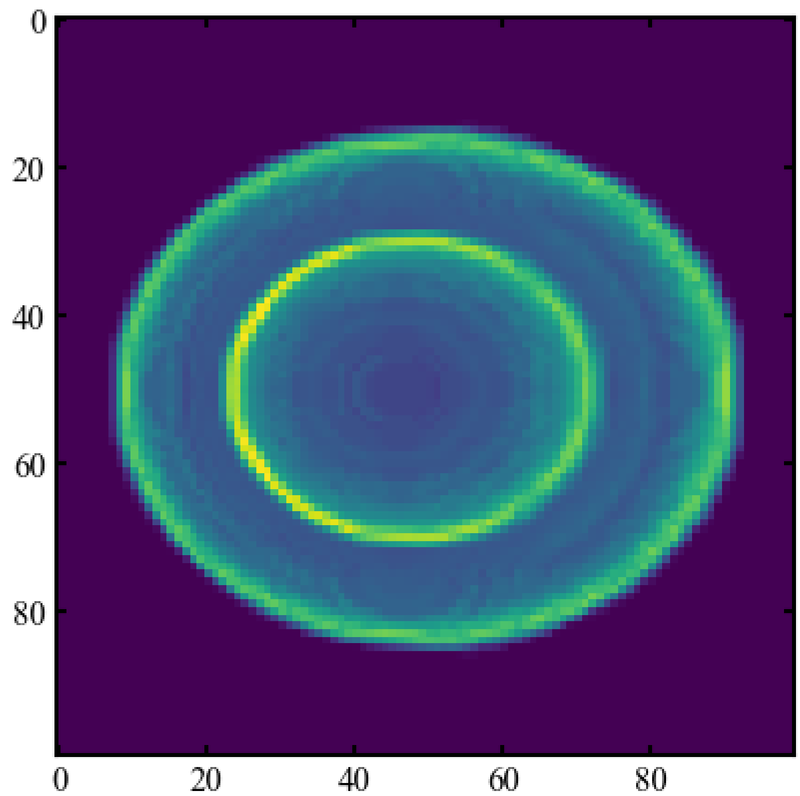
Try using peak finding to get the geometric transformation

In [1060]:
```
from scipy.signal import find_peaks

line_plot = np.arange(0, len(line_sample), 1)

cnt = 0
for which_line in range(end_view_numpy.shape[0]):


    line_sample = end_view_numpy[which_line, :]
    line_sample_sym = end_view_sym_numpy[which_line, :]

    line_features = find_peaks(line_sample, height=4)[0]
    line_features_sym = find_peaks(line_sample_sym, height=4)[0]

    if len(line_features) > 0 and len(line_features) == len(line_features_sym):
        if cnt == 0:
            src_peaks =np.hstack((np.array(line_features).reshape(-1, 1),
                                  which_line*np.ones((len(line_features))).reshape(-1, 1)))

            dst_peaks =np.hstack((np.array(line_features_sym).reshape(-1, 1),
                                  which_line*np.ones((len(line_features))).reshape(-1, 1)))
        else:
            src_peaks = np.vstack((src_peaks, np.hstack((np.array(line_features).reshape(-1, 1),
                                   which_line*np.ones((len(line_features))).reshape(-1, 1)))))

            dst_peaks = np.vstack((dst_peaks, np.hstack((np.array(line_features_sym).reshape(-1, 1),
                                   which_line*np.ones((len(line_features))).reshape(-1, 1)))))

        cnt += 1

u = dst_peaks - src_peaks
```

In [1061]:
```
# check if GPU is available, otherwise use CPU
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# create regularizer images

reg_list = tuple()
for ri, reg_angle in enumerate(reg_angles):

#     print(warp_sine[ri])
    tformEW_torch = ProjectiveTransform()
    tformEW_torch.estimate(src_peaks, src_peaks + u*1.3*warp_sine[ri])

    reg = radon3D_torch(image_cube_grad_sym_torch, theta = reg_angle)
    reg = warp(reg.numpy(), tformEW_torch, output_shape=reg_list_SW[-1].shape)
    reg_list += (torch.from_numpy(reg), )
```

In [1067]:
```
plt.imshow(reg_list[25])
```

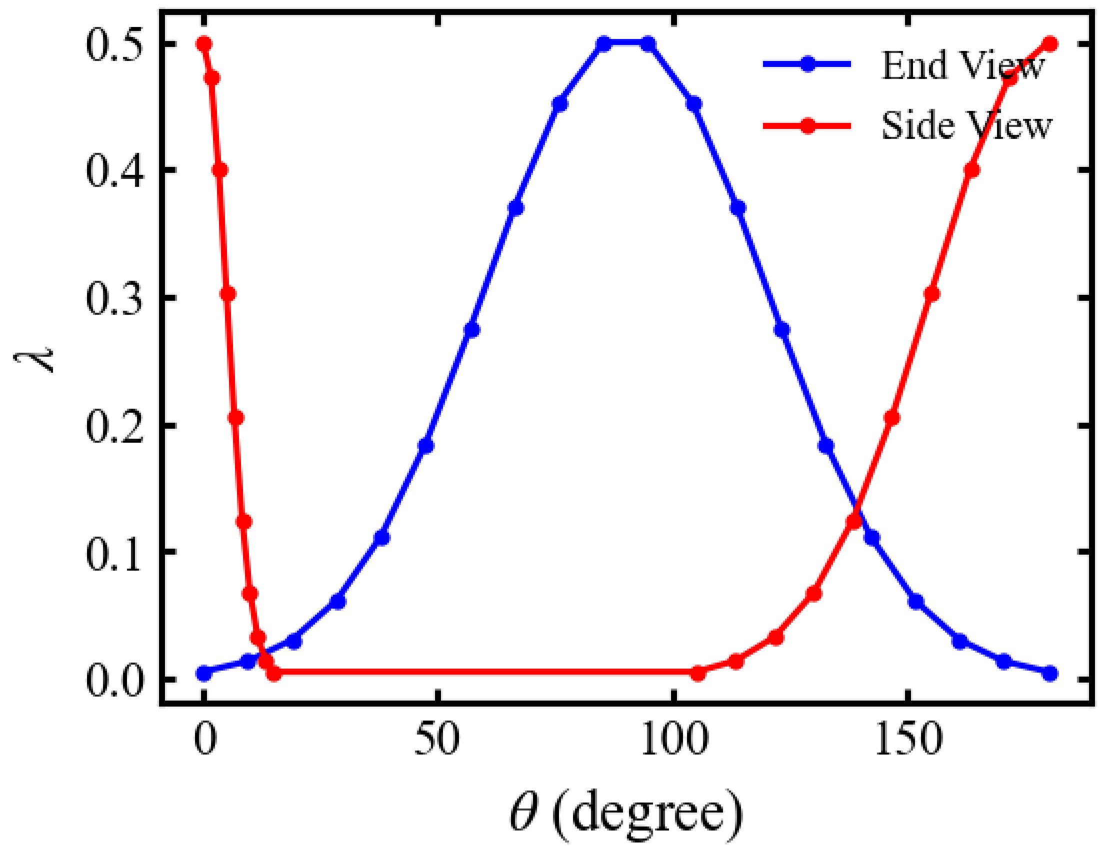
Out[1067]: <matplotlib.image.AxesImage at 0x219bdfc7b80>



Use two separate regularizers

In [852]:
```
reg_angles_EW = np.linspace(90-15*6, 90+15*6, 20)
lam_gaussian_EW = gaussian_fun(reg_angles_EW, 90, 30)
lam_gaussian_EW = 0.5*lam_gaussian_EW/lam_gaussian_EW.max()

reg_angles_SW = np.hstack((np.linspace(0, 5*3, 10), np.linspace(180-25*3, 180, 10)))
lam_gaussian_SW = 0.5*gaussian_fun(reg_angles_SW, 0, 5)/gaussian_fun(reg_angles_SW, 0, 5).max() +\
                  0.5*gaussian_fun(reg_angles_SW, 180, 25)/gaussian_fun(reg_angles_SW, 180, 25).max()
# lam_gaussian_SW = lam_gaussian_SW/lam_gaussian_SW.max()

fig, ax = plt.subplots(dpi = 150, figsize=(4, 3))

ax.plot(reg_angles_EW, lam_gaussian_EW, 'b.-', label = 'End View')
ax.plot(reg_angles_SW, lam_gaussian_SW, 'r.-', label = 'Side View')
ax.set_xlabel(r'$\theta$ (degree)')
ax.set_ylabel(r'$\lambda$')
ax.legend(frameon = False, fontsize = 10)
```

Out[852]:
```
<matplotlib.legend.Legend at 0x219ad26ca30>
```



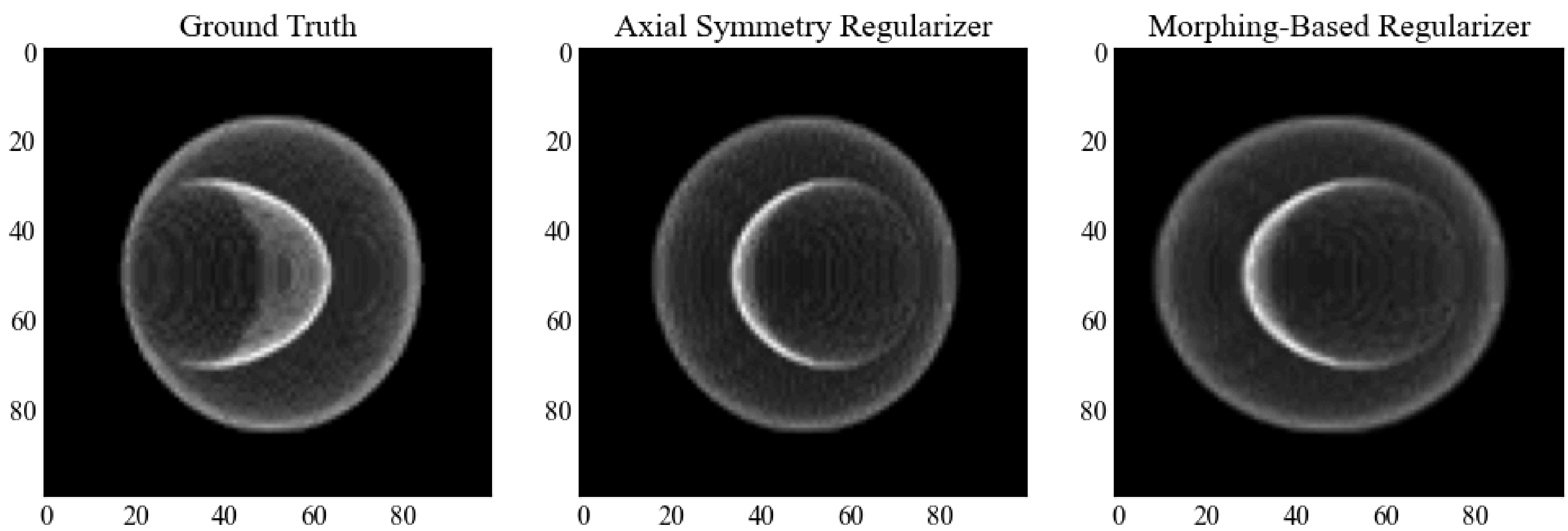
In [831]:
```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# convert the numpy arrays to pytorch tensors
image_cube_grad_torch = torch.from_numpy(image_cube_grad).to(device)
image_cube_grad_sym_torch = torch.from_numpy(image_cube_grad_sym).to(device)
```

In [832]:
```
# check if GPU is available, otherwise use CPU
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# create regularizer images
reg_list_SW = tuple()
for reg_angle in reg_angles_EW:
    reg_list_SW += (radon3D_torch(image_cube_grad_sym_torch, theta = reg_angle), )

reg_list_EW = tuple()
for reg_angle in reg_angles_EW:
    reg_EW = radon3D_torch(image_cube_grad_sym_torch, theta = reg_angle)

    reg_EW = warp(reg_EW.numpy(), tformEW_torch, output_shape=reg_list_SW[-1].shape)
    reg_list_EW += (torch.from_numpy(reg_EW), )
```

In [833]:
```
fig, axes = plt.subplots(1, 3, figsize=(12, 4.5))
index_to_probe = 12

print(reg_angles[index_to_probe])

axes[0].imshow(radon3D_torch(image_cube_grad_torch, theta = reg_angles[index_to_probe]), cmap = 'gray')
axes[0].set_title('Ground Truth')

axes[1].imshow(reg_list_SW[index_to_probe], cmap = 'gray')
axes[1].set_title('Axial Symmetry Regularizer')

axes[2].imshow(reg_list_EW[index_to_probe], cmap = 'gray')
axes[2].set_title('Morphing-Based Regularizer')
```

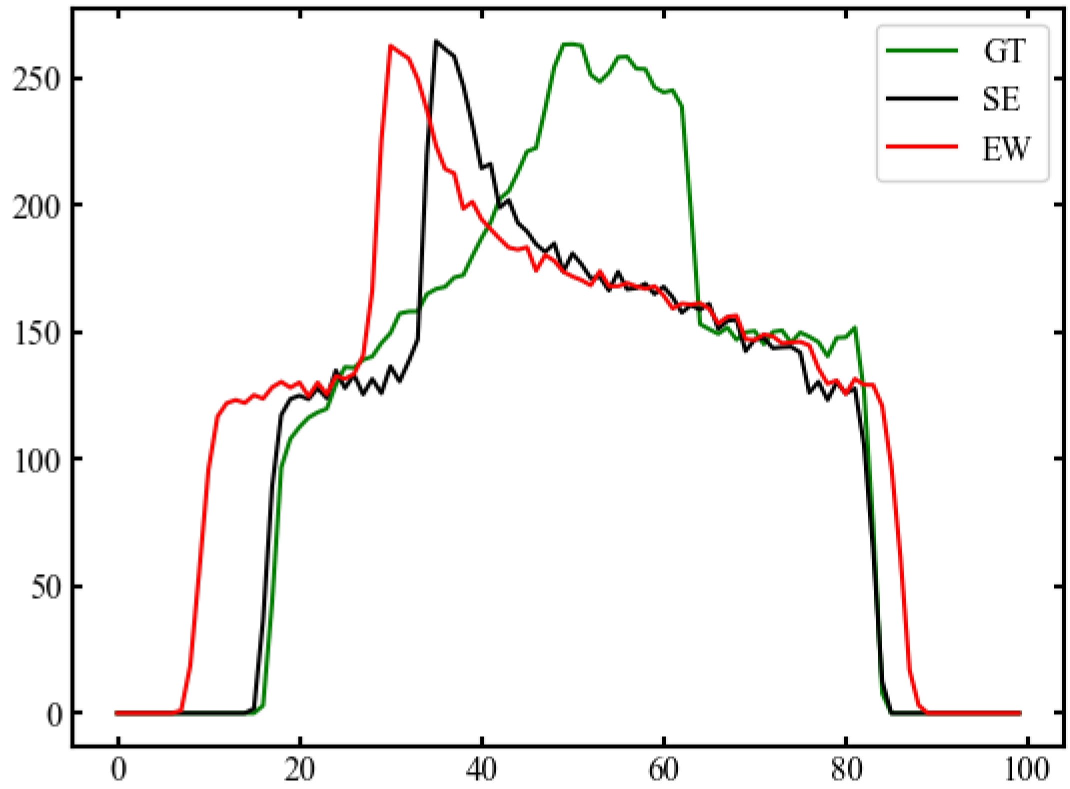
```
44.08163265306123
```

Out[833]:
```
Text(0.5, 1.0, 'Morphing-Based Regularizer')
```



In [834]:
```
plt.plot(np.sum(radon3D_torch(image_cube_grad_torch, theta = reg_angles[index_to_probe]).numpy(), axis = 0), '
plt.plot(np.sum(reg_list_SW[index_to_probe].numpy(), axis = 0), 'k', label = 'SE')
plt.plot(np.sum(reg_list_EW[index_to_probe].numpy(), axis = 0), 'r', label = 'EW')
plt.legend()
```

Out[834]: <matplotlib.legend.Legend at 0x219ad490910>



In [1068]:

```
from tqdm import tqdm
import torch

def iradon_adam_morph(image_projs_torch, image_cube_reg, tformEW = None,
                      reg_anglesSW = np.linspace(0, 180, 15), reg_anglesEW = np.linspace(0, 180, 15),
                      lamSW = np.ones((15, )), lamEW = np.ones((15, )),
                      reg_overwrite = False, reg_list = None, reg_angles = None, lam_gaussian = None,
                      num_iters =75, learning_rate=5e-2):

    # check if GPU is available, otherwise use CPU
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

    # create regularizer images
    if not reg_overwrite:

        reg_list_SW = tuple()
        for reg_angle in reg_anglesSW:
            reg_list_SW += (radon3D_torch(image_cube_reg, theta = reg_angle), )

        reg_list_EW = tuple()
        for reg_angle in reg_anglesEW:
            reg_EW = radon3D_torch(image_cube_reg, theta = reg_angle)
            reg_EW = warp(reg_EW.numpy(), tformEW, output_shape=reg_list_SW[-1].shape)
            reg_list_EW += (torch.from_numpy(reg_EW), )

    # initialize the solution
    x = torch.zeros_like(image_cube_reg,
                         requires_grad=True).to(device) #np.zeros_like(image_cube_reg)

    # initialize Adam optimizer
    optim = torch.optim.Adam(params=[x], lr=learning_rate)

    for it in tqdm(range(num_iters)):

        # set all gradients of the computational graph to 0
        optim.zero_grad()

        # this term computes the data fidelity term of the loss function
        loss_data = 0
        for ai, theta in enumerate(image_projs_torch['angle_list']):

            loss_data += (radon3D_torch(x, theta = theta) -\
                          image_projs_torch['image_list'][ai]).pow(2).sum()
```

```
        # regularizer terms
        if not reg_overwrite:
            loss_regularizer_SW = 0
            for ai, theta in enumerate(reg_anglesSW):
                loss_regularizer_SW += lamSW[ai] *(radon3D_torch(x, theta = theta) - reg_list_SW[ai]).pow(2).s

            loss_regularizer_EW = 0
            for ai, theta in enumerate(reg_anglesEW):
                loss_regularizer_EW += lamEW[ai] *(radon3D_torch(x, theta = theta) - reg_list_EW[ai]).pow(2).s

            # compute weighted sum of data fidelity and regularization term
            loss = loss_data +  loss_regularizer_SW + loss_regularizer_EW
        else:
            loss_regularizer = 0
            for ai, theta in enumerate(reg_angles):
                loss_regularizer += lam_gaussian[ai] *(radon3D_torch(x, theta = theta) - reg_list[ai]).pow(2).

            # compute weighted sum of data fidelity and regularization term
            loss = loss_data +  loss_regularizer


        # compute backwards pass
        loss.backward()

        # take a step with the Adam optimizer
        optim.step()

    # return the result as a numpy array
    return x.detach().cpu().numpy()
```

In [1069]:

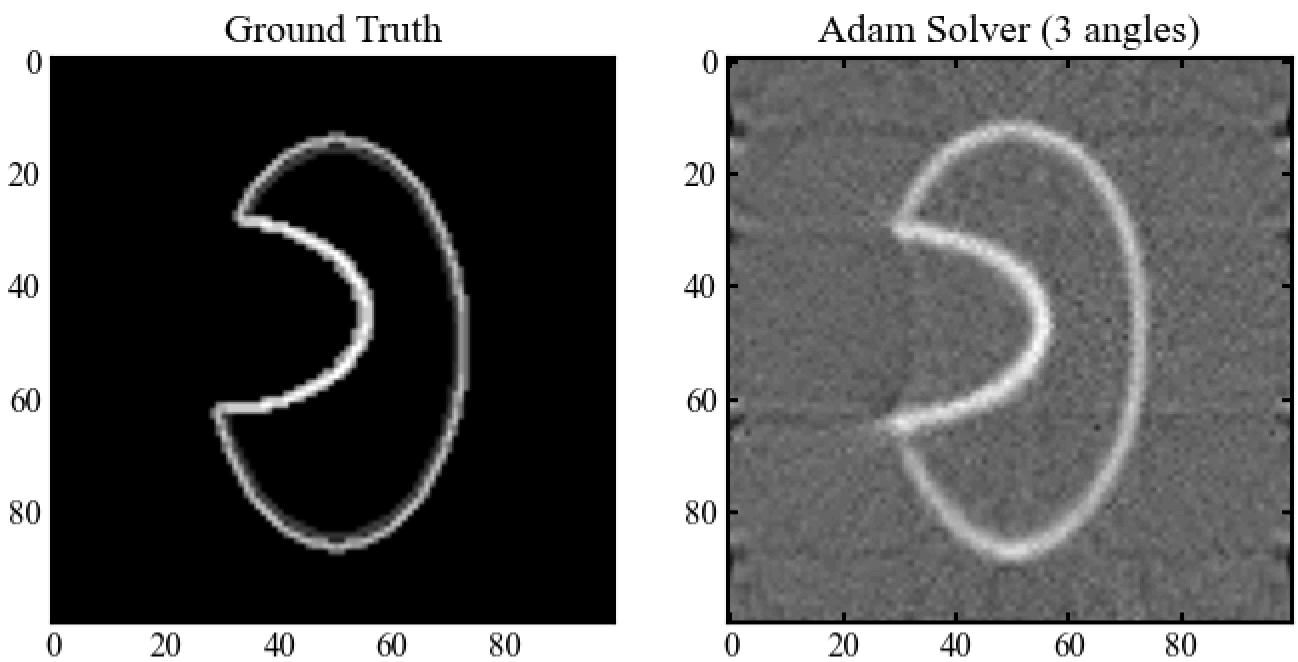
```
# image_cube_adam_morph = iradon_adam_morph(image_projs_torch, image_cube_grad_sym_torch,
#                                           tformEW = tformEW_torch,
#                                           reg_anglesEW = reg_angles_EW, reg_anglesSW = reg_angles_SW,
#                                           lamSW = lam_gaussian_SW, lamEW = lam_gaussian_EW,
#                                           num_iters =75, learning_rate=5e-2)
```

In [1070]:
```
image_cube_adam_morph = iradon_adam_morph(image_projs_torch, image_cube_grad_sym_torch,
                                          tformEW = tformEW_torch,
                                          reg_overwrite = True, reg_list = reg_list, reg_angles = reg_angles,
                                          lam_gaussian = lam_gaussian,
                                          lamSW = lam_gaussian_SW, lamEW = lam_gaussian_EW,
                                          num_iters =75, learning_rate=5e-2)
```

```
100%|██████████| 75/75 [10:37<00:00,  8.50s/it]
```

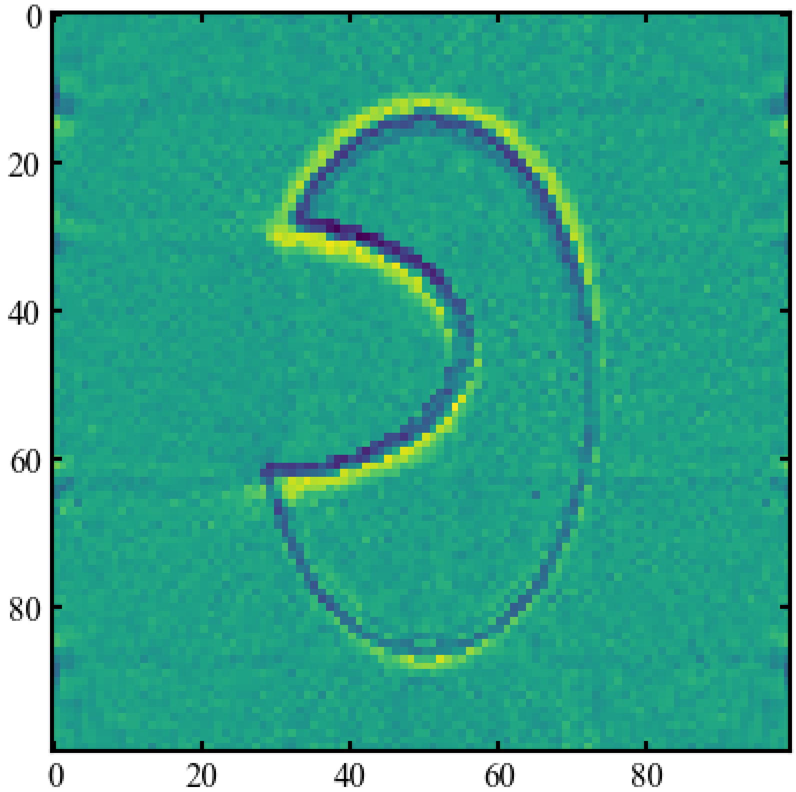
In [1072]:
```
fig, axes = plt.subplots(1, 2, figsize=(8, 4.5))

layger_to_probe = 35

axes[0].imshow(image_cube_grad[layger_to_probe, ...], cmap = 'gray')
axes[0].set_title('Ground Truth')

# axes[1].imshow(subtracted, cmap = 'gray')
# axes[1].imshow(image_cube_grad[layger_to_probe, ...], cmap = 'gray')

axes[1].imshow(image_cube_adam_morph[layger_to_probe, ...], cmap ='gray', alpha = 1)
axes[1].set_title(f"Adam Solver ({len(image_projs['angle_list'])} angles)")
```

Out[1072]:
```
Text(0.5, 1.0, 'Adam Solver (3 angles)')
```



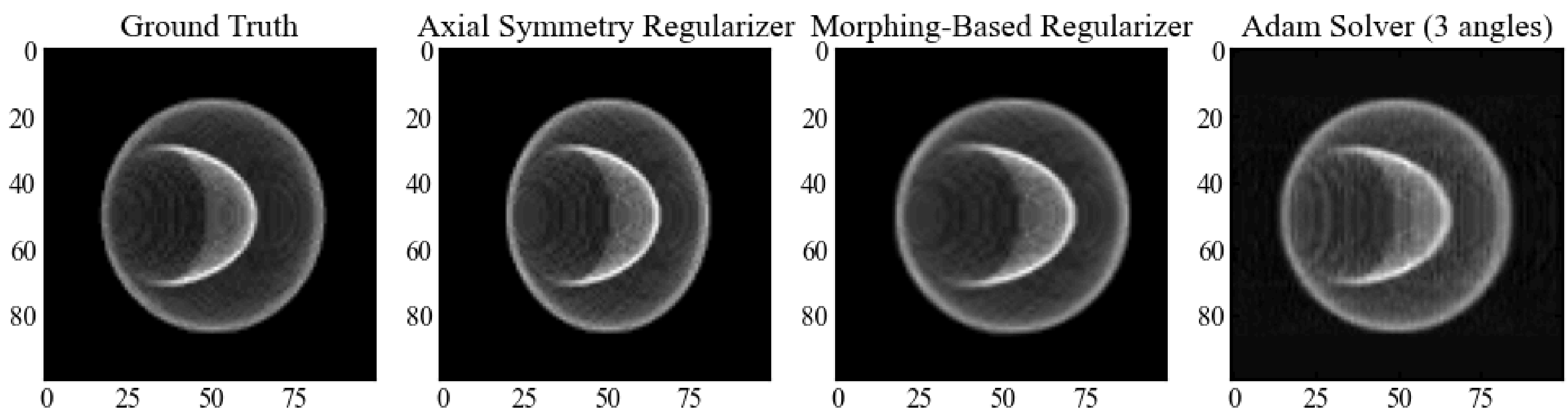
In [1073]:
```
plt.imshow(image_cube_adam_morph[layger_to_probe, ...] - image_cube_grad[layger_to_probe, ...])
```

Out[1073]:
```
<matplotlib.image.AxesImage at 0x219b0707430>
```



In [1075]:
```
fig, axes = plt.subplots(1, 4, figsize=(12, 4.5))
theta_to_probe = 45

axes[0].imshow(radon3D_torch(image_cube_grad_torch, theta = theta_to_probe), cmap = 'gray')
axes[0].set_title('Ground Truth')

axes[1].imshow(radon3D_torch(image_cube_grad_sym_torch, theta = theta_to_probe), cmap = 'gray')
axes[1].set_title('Axial Symmetry Regularizer')

reg_EW = radon3D_torch(image_cube_grad_sym_torch, theta = theta_to_probe)
reg_EW = warp(reg_EW.numpy(), tformEW, output_shape=reg_list_SW[-1].shape)

axes[2].imshow(reg_EW, cmap = 'gray')
axes[2].set_title('Morphing-Based Regularizer')

# axes[1].imshow(subtracted, cmap = 'gray')
axes[3].imshow(radon3D_torch(torch.from_numpy(image_cube_adam_morph), theta = theta_to_probe), cmap ='gray')
axes[3].set_title(f"Adam Solver ({len(image_projs['angle_list'])} angles)")
```

Out[1075]:
```
Text(0.5, 1.0, 'Adam Solver (3 angles)')
```



Morphing-based view interpolation + TV regularizer + ADMM

In [1077]:
```
# gradient calculator for TV regularizaer

def opDx_(x, axis):
    slc = [slice(None)] * len(x.shape)
    slc[axis] = slice(0, 1)
    Dx = np.diff(x, axis=axis, append=x[tuple(slc)])
    return Dx

def opDtx_(diffx, axis):
    Dtx = -diffx + np.roll(diffx, 1, axis=axis)
    return Dtx

def opDx_3D(x):
    Dx = opDx_(x, axis=-1)
    Dy = opDx_(x, axis=-2)
    Dz = opDx_(x, axis=-3)
    if len(Dx.shape) <= 3:
        return np.stack((Dx, Dy, Dz), axis=-4)
    else:
        return np.concatenate((Dx, Dy, Dz), axis=-4)

def opDtx_3D(x):
    Dtx = opDtx_(x[..., 0, :, :, :], axis=-1)
    Dty = opDtx_(x[..., 1, :, :, :], axis=-2)
    Dtz = opDtx_(x[..., 2, :, :, :], axis=-3)
    return Dtx + Dty + Dtz
```

In [1078]:
```
from scipy.ndimage import rotate

def radon_numpy(image, angle):
    rotated_image = rotate(image, angle, reshape=False)
    radon_projection = np.sum(rotated_image, axis=0)
    return radon_projection

def radon_numpy_transpose(radon_projection, angle, image_shape):
    width, height = image_shape
    rotated_projection = np.tile(radon_projection, (width, 1))
    rotated_image = rotate(rotated_projection, -angle, reshape=False)
    return rotated_image

def radon3D_numpy(image_cube, theta = 90):
    '''
    Method to calculate 3D radon transform
    image_cubee: 3D object with dimension nXmXP
    theta: projection angle in the xy plane
    '''
    image_proj = np.zeros((image_cube.shape[0], image_cube.shape[1]))
    for yi in range(image_cube.shape[0]):
        image_slice = image_cube[yi, ...]
        image_proj[yi, :] = radon_numpy(image_slice, angle=theta).reshape((image_cube.shape[1], ))

    return image_proj

def radon3D_numpy_transpose(image_proj, theta = 90, cube_shape = (100, 100, 100)):
    '''
    Method to calculate the transpose of the 3D radon transform
    image_cubee: 3D object with dimension nXmXP
    theta: projection angle in the xy plane
    '''
    image_cube = np.zeros(cube_shape)
    for yi in range(image_proj.shape[0]):
        image_slice = radon_numpy_transpose(image_proj[yi, ...], theta, (cube_shape[1], cube_shape[2]))
        image_cube[yi, ...] = image_slice

    return image_cube
```

In [1079]:
```
import numpy as np
from scipy.sparse.linalg import cg, LinearOperator
from tqdm import tqdm

class iradon_admm:

    def __init__(self, image_projs, image_cube_reg,
                 tformEW = None, reg_anglesSW = np.linspace(0, 180, 15), reg_anglesEW = np.linspace(0, 1
                 lamSW = np.ones((15, )), lamEW = np.ones((15, )), lamTV = 1.0,
                 reg_overwrite = False, reg_list = None, reg_angles = None,
                 lam_gaussian = None,
                 rho = 16, num_iters = 75, anisotropic_tv=True):

        self.image_projs = image_projs
        self.image_cube_reg = image_cube_reg
        self.tformEW = tformEW
        self.reg_anglesSW = reg_anglesSW
        self.reg_anglesEW = reg_anglesEW
        self.lamSW = lamSW
        self.lamEW = lamEW
        self.lamTV = lamTV
        self.reg_overwrite = reg_overwrite
        self.reg_list = reg_list
        self.reg_angles = reg_angles
        self.lam_gaussian = lam_gaussian
        self.rho = rho
        self.num_iters = num_iters
        self.anisotropic_tv = anisotropic_tv

        self.cube_shape = image_cube_reg.shape
        self.vec_length = self.cube_shape[0]*self.cube_shape[1]*self.cube_shape[2]

        if self.reg_overwrite:
            # create regularizer images
            self.reg_list_SW = tuple()
            for reg_angle in self.reg_anglesSW:
                self.reg_list_SW += (radon3D_numpy(self.image_cube_reg, theta = reg_angle), )

            self.reg_list_EW = tuple()
            for reg_angle in self.reg_anglesEW:
                reg_EW = radon3D_numpy(self.image_cube_reg, theta = reg_angle)
                reg_EW = warp(reg_EW, self.tformEW, output_shape=self.reg_list_SW[-1].shape)
                self.reg_list_EW += (reg_EW, )
```

```
    def A_tilt_fun(self, x_vec):

        x = x_vec.reshape(self.cube_shape)

        # data fitting term
        data_term = np.zeros(self.cube_shape)
        for ai, theta in enumerate(self.image_projs['angle_list']):
            data_term += radon3D_numpy_transpose(radon3D_numpy(x, theta = theta),
                                                 theta = theta, cube_shape =self.cube_shape)

        if self.reg_overwrite:
            # side-wall regularizer term
            reg_term = np.zeros(self.cube_shape)
            for ai, theta in enumerate(self.reg_angles):
                reg_term += self.lam_gaussian[ai]*radon3D_numpy_transpose(radon3D_numpy(x, theta = theta),
                                                 theta = theta, cube_shape =self.cube_shape)
        else:

            # side-wall regularizer term
            reg_side_term = np.zeros(self.cube_shape)
            for ai, theta in enumerate(self.reg_anglesSW):
                reg_side_term += self.lamSW[ai]*radon3D_numpy_transpose(radon3D_numpy(x, theta = theta),
                                                 theta = theta, cube_shape =self.cube_shape)

            # end-wall regularizer term
            reg_end_term = np.zeros(self.cube_shape)
            for ai, theta in enumerate(self.reg_anglesEW):
                reg_end_term += self.lamEW[ai]*radon3D_numpy_transpose(radon3D_numpy(x, theta = theta),
                                                 theta = theta, cube_shape =self.cube_shape)

        # TV term
        TV_term = np.zeros(self.cube_shape)
        TV_term = self.rho*opDtx_3D(opDx_3D(x))

        if self.reg_overwrite:
            return data_term.reshape((self.vec_length, )) + \
                    reg_term.reshape((self.vec_length, )) + \
                            TV_term.reshape((self.vec_length, ))
        else:
            return data_term.reshape((self.vec_length, )) + \
                    reg_side_term.reshape((self.vec_length, )) + \
                        reg_end_term.reshape((self.vec_length, )) + \
```

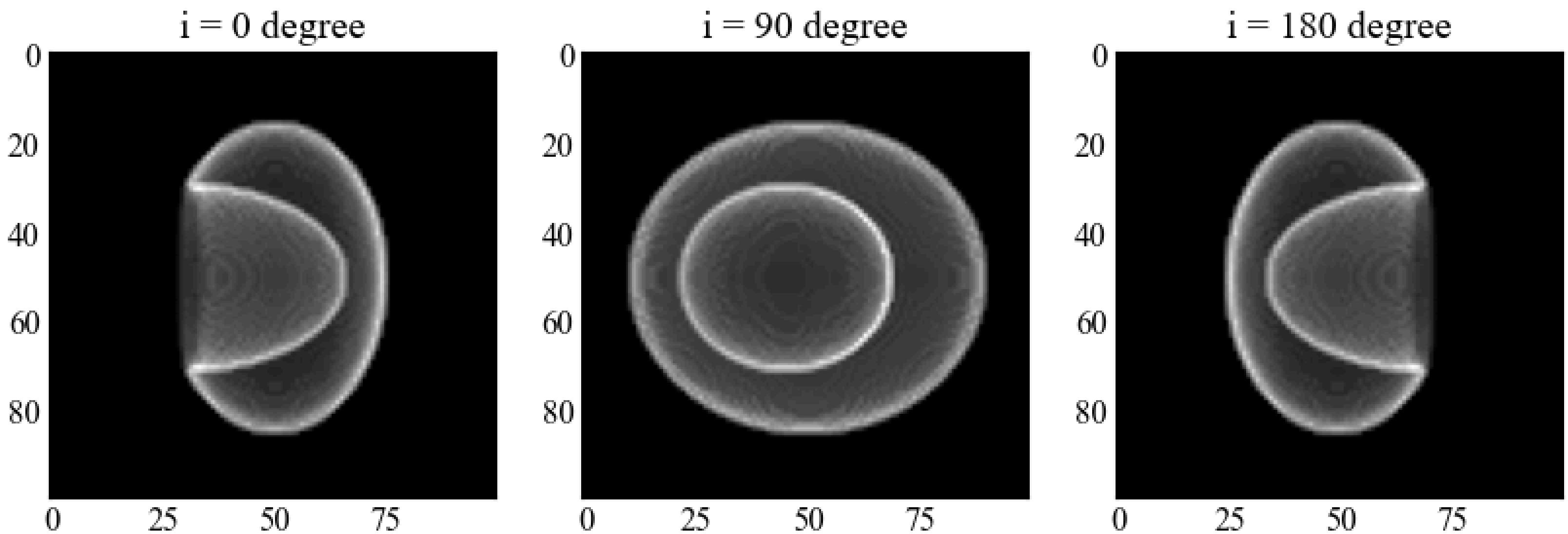
```
                        TV_term.reshape((self.vec_length, ))

    def b_tilt_fun(self, z, u):

        # data fitting term
        data_term = np.zeros(self.cube_shape)
        for ai, theta in enumerate(self.image_projs['angle_list']):
            data_term += radon3D_numpy_transpose(self.image_projs['image_list'][ai],
                                                 theta = theta, cube_shape =self.cube_shape)
        if self.reg_overwrite:
            reg_term = np.zeros(self.cube_shape)
            for ai, theta in enumerate(self.reg_angles):
                reg_term += self.lam_gaussian[ai]*radon3D_numpy_transpose(self.reg_list[ai],
                                                 theta = theta, cube_shape =self.cube_shape)
        else:
            # side-wall regularizer term
            reg_side_term = np.zeros(self.cube_shape)
            for ai, theta in enumerate(self.reg_anglesSW):
                reg_side_term += self.lamSW[ai]*radon3D_numpy_transpose(self.reg_list_SW[ai],
                                                 theta = theta, cube_shape =self.cube_shape)

            # end-wall regularizer term
            reg_end_term = np.zeros(self.cube_shape)
            for ai, theta in enumerate(self.reg_anglesEW):
                reg_end_term += self.lamEW[ai]*radon3D_numpy_transpose(self.reg_list_EW[ai],
                                                 theta = theta, cube_shape =self.cube_shape)

        # TV term
        TV_term = np.zeros(self.cube_shape)
        TV_term = self.rho*opDtx_3D(z-u)

        if self.reg_overwrite:
            return data_term.reshape((self.vec_length, )) + \
                    reg_term.reshape((self.vec_length, )) + \
                        TV_term.reshape((self.vec_length, ))
        else:
            return data_term.reshape((self.vec_length, )) + \
                    reg_side_term.reshape((self.vec_length, )) + \
                      reg_end_term.reshape((self.vec_length, )) + \
                        TV_term.reshape((self.vec_length, ))

    def iradon_admm_tv(self, cg_iters = 25, cg_tolerance = 1e-12):
```

```
        # initialize x,z,u with all zeros
        x = np.zeros(self.cube_shape)
        z = np.zeros((3, *self.image_cube_reg.shape))
        u = np.zeros((3, *self.image_cube_reg.shape))

        A_tilt_op = LinearOperator((self.vec_length, self.vec_length), matvec =self.A_tilt_fun)

        for it in tqdm(range(self.num_iters)):

            # x update using cg solver
            v = z-u

            x = cg(A_tilt_op, self.b_tilt_fun(z, u), tol = cg_tolerance, maxiter = cg_iters)[0]
            x = x.reshape(self.cube_shape)

            # z update - soft shrinkage
            kappa = self.lamTV / self.rho
            v = opDx_3D(x) + u

            # proximal operator of anisotropic TV term
            if self.anisotropic_tv:
                z = np.maximum(1 - kappa/np.abs(v), 0) * v

            # proximal operator of isotropic TV term
            else:
                vnorm = np.sqrt( v[0,...]**2 + v[1,...]**2 + v[2,...]**2)
                z[0,...] = np.maximum(1 - kappa/vnorm,0) * v[0,:,:]
                z[1,...] = np.maximum(1 - kappa/vnorm,0) * v[1,:,:]
                z[2,...] = np.maximum(1 - kappa/vnorm,0) * v[2,:,:]

            # u-update
            u = u + opDx_3D(x) - z

        return x
```

In [1081]:
```
# create regularizer images

reg_list = tuple()
for ri, reg_angle in enumerate(reg_angles):

#     print(warp_sine[ri])
    tformEW_torch = ProjectiveTransform()
    tformEW_torch.estimate(src_peaks, src_peaks + u*1.3*warp_sine[ri])

    reg = radon3D_numpy(image_cube_grad_sym, theta = reg_angle)
    reg = warp(reg, tformEW_torch, output_shape=reg_list_SW[-1].shape)
    reg_list += (reg, )
```

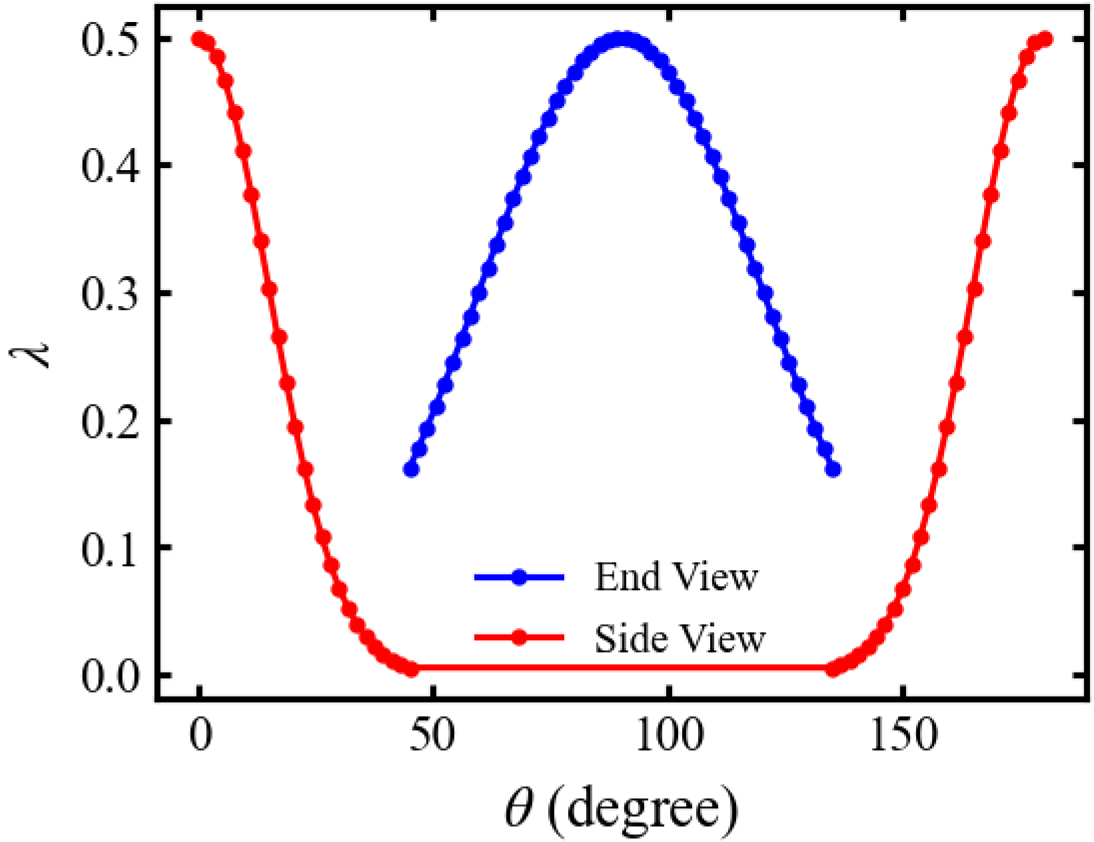
In [1084]:
```
# plt.imshow(reg_list[25])
```

In [1109]:
```
image_projs = dict()
image_projs['angle_list'] = [0, 90, 180]
fig, axes = plt.subplots(1, len(image_projs['angle_list']), figsize=(10, 4.5))

image_projs['image_list'] = tuple()
for ai, angle in enumerate(image_projs['angle_list']):
    image_projs['image_list'] += (radon3D_numpy(image_cube_grad, theta = angle), )

    axes[ai].imshow(image_projs['image_list'][-1], cmap = 'gray')
    axes[ai].set_title(f"i = {angle} degree")
```



In [589]:
```
reg_angles_EW = np.linspace(90-15*3, 90+15*3, 50)
lam_gaussian_EW = gaussian_fun(reg_angles_EW, 90, 30)
lam_gaussian_EW = 0.5*lam_gaussian_EW/lam_gaussian_EW.max()

reg_angles_SW = np.hstack((np.linspace(0, 15*3, 25), np.linspace(180-15*3, 180, 25)))
lam_gaussian_SW = gaussian_fun(reg_angles_SW, 0, 15)  + gaussian_fun(reg_angles_SW, 180, 15)
lam_gaussian_SW = 0.5*lam_gaussian_SW/lam_gaussian_SW.max()

fig, ax = plt.subplots(dpi = 150, figsize=(4, 3))

ax.plot(reg_angles_EW, lam_gaussian_EW, 'b.-', label = 'End View')
ax.plot(reg_angles_SW, lam_gaussian_SW, 'r.-', label = 'Side View')
ax.set_xlabel(r'$\theta$ (degree)')
ax.set_ylabel(r'$\lambda$')
ax.legend(frameon = False, fontsize = 10)
```

Out[589]:
```
<matplotlib.legend.Legend at 0x219bfd2b8b0>
```



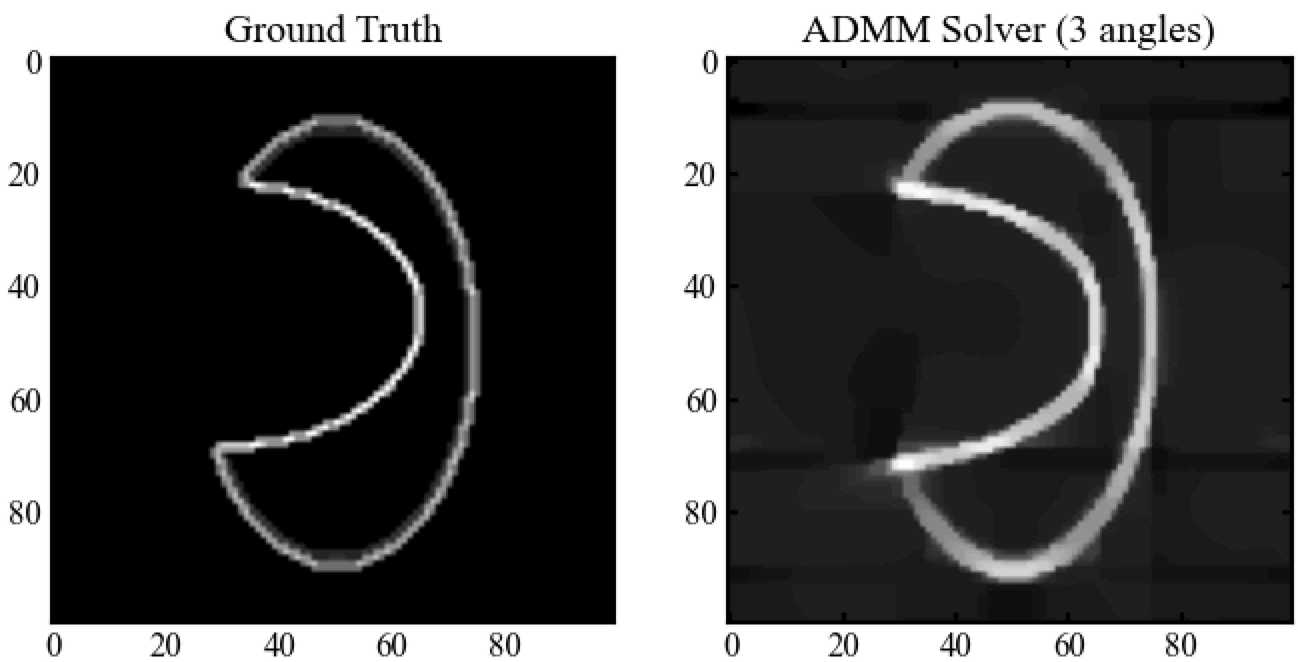
In [1085]:
```
iradmm = iradon_admm(image_projs, image_cube_grad_sym,
                     tformEW = tformEW_torch, reg_anglesSW = reg_angles_SW, reg_anglesEW = reg_angles_EW,
                     lamSW = lam_gaussian_SW, lamEW = lam_gaussian_EW, lamTV = 1.0,
                     reg_overwrite = True, reg_list = reg_list, reg_angles = reg_angles,
                     lam_gaussian = lam_gaussian,
                     rho = 16, num_iters = 75, anisotropic_tv=True)
```

In [1086]:
```
image_cube_admm_tv = iradmm.iradon_admm_tv()
```

```
100%|██████████| 75/75 [3:24:19<00:00, 163.47s/it]
```

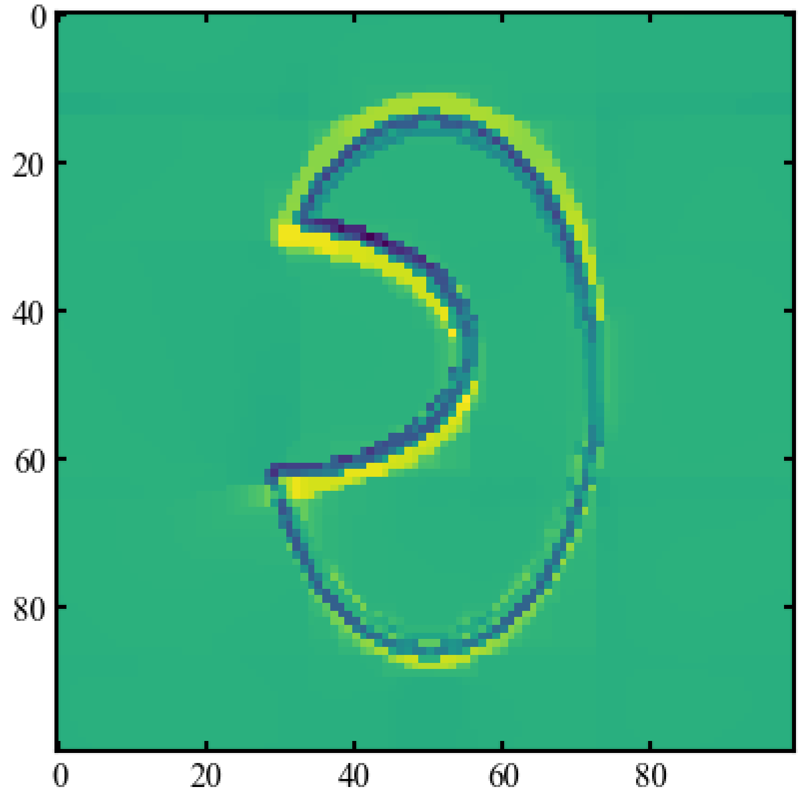
In [1104]:
```
fig, axes = plt.subplots(1, 2, figsize=(8, 4.5))

layger_to_probe = 45

axes[0].imshow(image_cube_grad[layger_to_probe, ...], cmap = 'gray')
axes[0].set_title('Ground Truth')

# axes[1].imshow(subtracted, cmap = 'gray')
axes[1].imshow(image_cube_grad[layger_to_probe, ...], cmap = 'gray')

axes[1].imshow(image_cube_admm_tv[layger_to_probe, ...], cmap ='gray', alpha = 1)
axes[1].set_title(f"ADMM Solver ({len(image_projs['angle_list'])} angles)")
```

Out[1104]:
```
Text(0.5, 1.0, 'ADMM Solver (3 angles)')
```



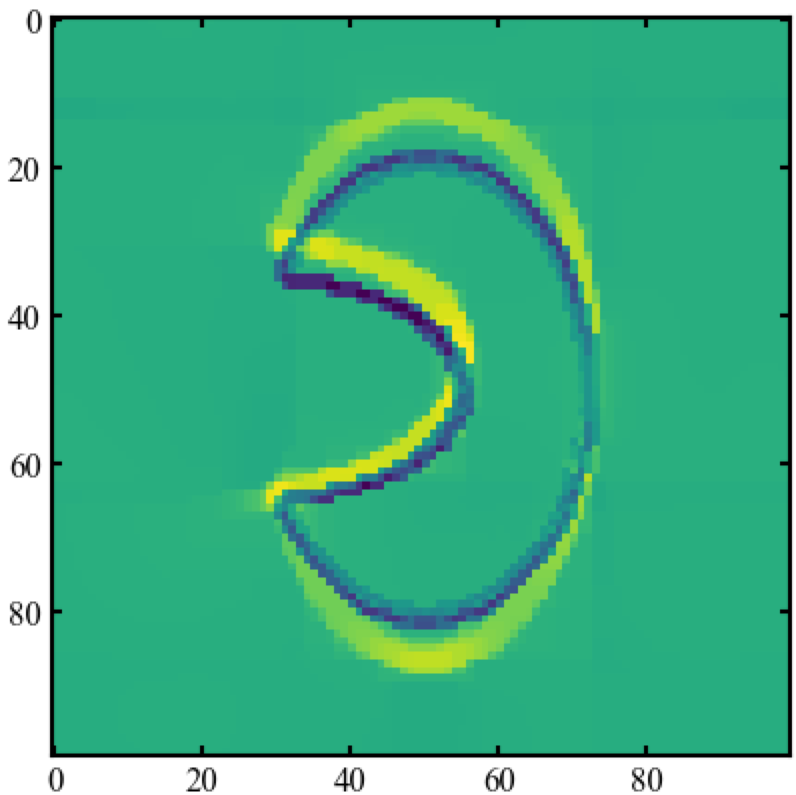
In [1093]:
```
plt.imshow(image_cube_admm_tv[layger_to_probe, ...] - image_cube_grad[layger_to_probe, ...])
```

Out[1093]: <matplotlib.image.AxesImage at 0x219c08df400>



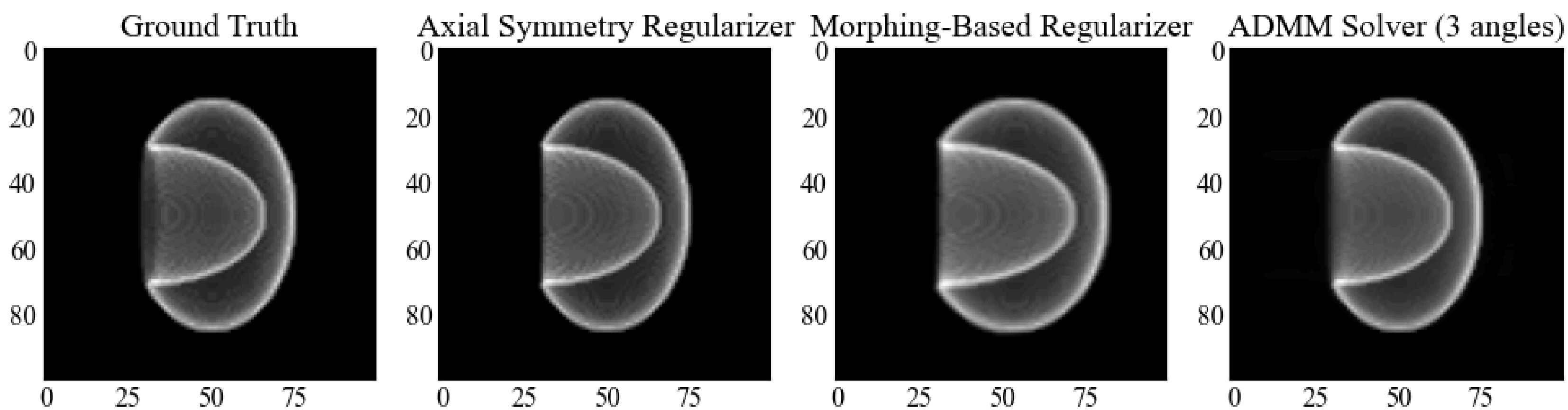
In [1095]:
```
plt.imshow(image_cube_admm_tv[layger_to_probe, ...] - image_cube_grad_sym[layger_to_probe, ...])
```

Out[1095]: <matplotlib.image.AxesImage at 0x219bfaba890>



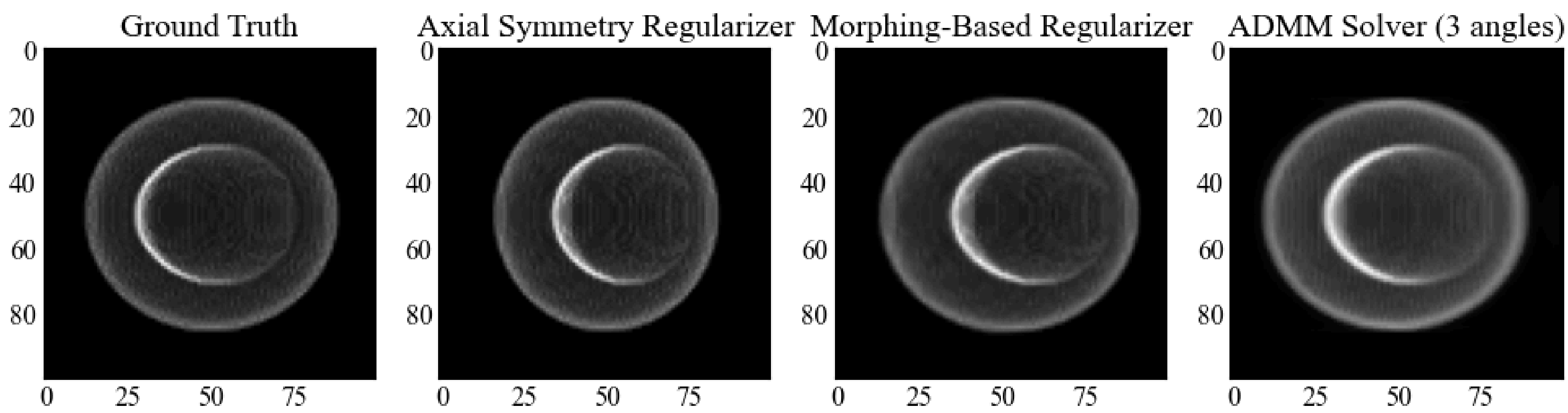
In [1099]:
```
fig, axes = plt.subplots(1, 4, figsize=(12, 4.5))
theta_to_probe = 0

axes[0].imshow(radon3D_torch(image_cube_grad_torch, theta = theta_to_probe), cmap = 'gray')
axes[0].set_title('Ground Truth')

axes[1].imshow(radon3D_torch(image_cube_grad_sym_torch, theta = theta_to_probe), cmap = 'gray')
axes[1].set_title('Axial Symmetry Regularizer')

reg_EW = radon3D_torch(image_cube_grad_sym_torch, theta = theta_to_probe)
reg_EW = warp(reg_EW.numpy(), tformEW, output_shape=reg_list_SW[-1].shape)

axes[2].imshow(reg_EW, cmap = 'gray')
axes[2].set_title('Morphing-Based Regularizer')

# axes[1].imshow(subtracted, cmap = 'gray')
axes[3].imshow(radon3D_torch(torch.from_numpy(image_cube_admm_tv), theta = theta_to_probe), cmap ='gray')
axes[3].set_title(f"ADMM Solver ({len(image_projs['angle_list'])} angles)")
```

Out[1099]:
```
Text(0.5, 1.0, 'ADMM Solver (3 angles)')
```



In [1091]:
```
fig, axes = plt.subplots(1, 4, figsize=(12, 4.5))
theta_to_probe = 115

axes[0].imshow(radon3D_torch(image_cube_grad_torch, theta = theta_to_probe), cmap = 'gray')
axes[0].set_title('Ground Truth')

axes[1].imshow(radon3D_torch(image_cube_grad_sym_torch, theta = theta_to_probe), cmap = 'gray')
axes[1].set_title('Axial Symmetry Regularizer')

reg_EW = radon3D_torch(image_cube_grad_sym_torch, theta = theta_to_probe)
reg_EW = warp(reg_EW.numpy(), tformEW, output_shape=reg_list_SW[-1].shape)

axes[2].imshow(reg_EW, cmap = 'gray')
axes[2].set_title('Morphing-Based Regularizer')

# axes[1].imshow(subtracted, cmap = 'gray')
axes[3].imshow(radon3D_torch(torch.from_numpy(image_cube_admm_tv), theta = theta_to_probe), cmap ='gray')
axes[3].set_title(f"ADMM Solver ({len(image_projs['angle_list'])} angles)")
```

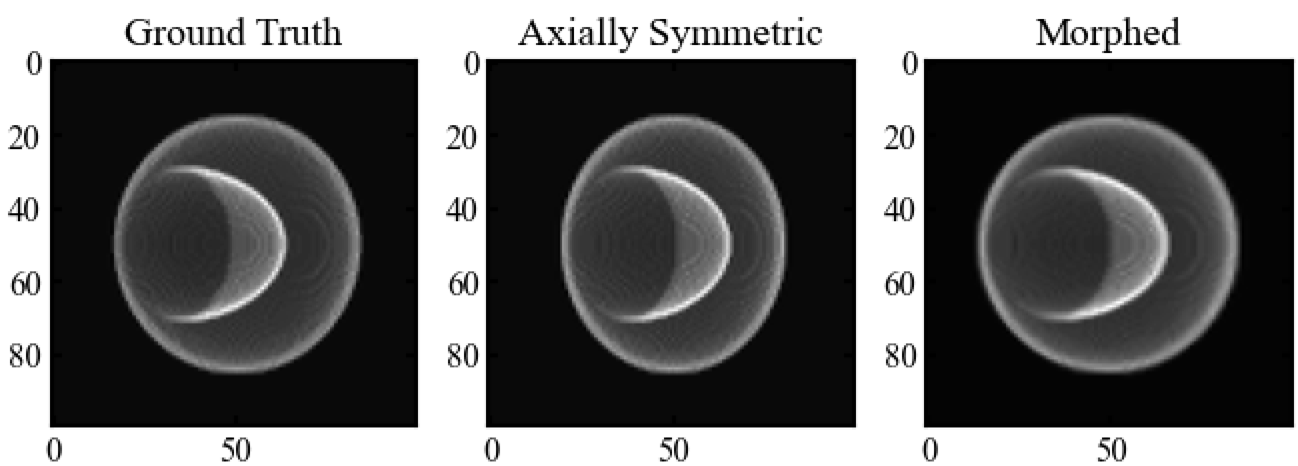
Out[1091]:
```
Text(0.5, 1.0, 'ADMM Solver (3 angles)')
```



Try cross reference using points

In [1113]:
```
end_view = radon3D(image_cube_grad, theta = np.array([45]))
end_view_torch = radon3D_torch(image_cube_grad_torch, theta = 45)
end_view_numpy = radon3D_numpy(image_cube_grad, theta = 45)

end_view_sym = radon3D(image_cube_grad_sym, theta = np.array([45]))
end_view_sym_torch = radon3D_torch(image_cube_grad_sym_torch, theta = 45)
end_view_sym_numpy = radon3D_numpy(image_cube_grad_sym, theta = 45)

fig, axes = plt.subplots(1, 3, figsize=(8, 4.5))

axes[0].imshow(end_view_numpy, cmap = 'gray')
axes[0].set_title('Ground Truth')

axes[1].imshow(end_view_sym_numpy, cmap = 'gray')
axes[1].set_title('Axially Symmetric')

axes[2].imshow(reg_list[13], cmap = 'gray')
axes[2].set_title('Morphed')
```

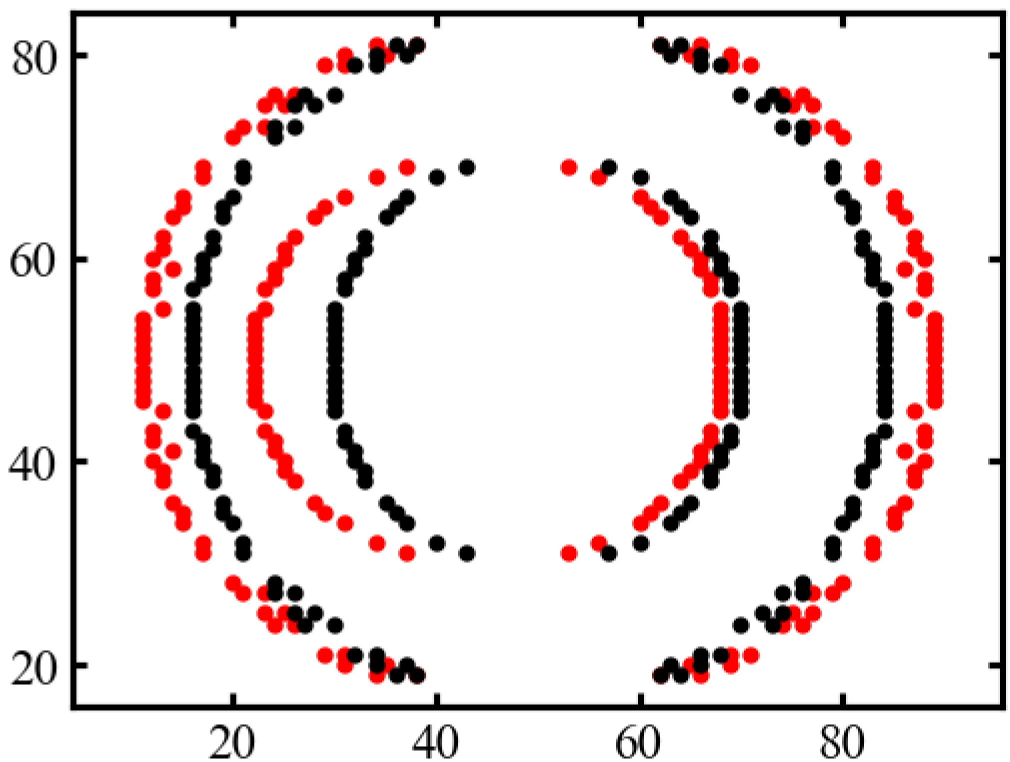
Out[1113]:
```
Text(0.5, 1.0, 'Morphed')
```



In [1045]:
```
from scipy.signal import find_peaks

line_plot = np.arange(0, len(line_sample), 1)

cnt = 0
for which_line in range(end_view_numpy.shape[0]):


    line_sample = end_view_numpy[which_line, :]
    line_sample_sym = end_view_sym_numpy[which_line, :]

    line_features = find_peaks(line_sample, height=4)[0]
    line_features_sym = find_peaks(line_sample_sym, height=4)[0]

    if len(line_features) > 0 and len(line_features) == len(line_features_sym):
        if cnt == 0:
            src_peaks =np.hstack((np.array(line_features).reshape(-1, 1),
                                  which_line*np.ones((len(line_features))).reshape(-1, 1)))

            dst_peaks =np.hstack((np.array(line_features_sym).reshape(-1, 1),
                                  which_line*np.ones((len(line_features))).reshape(-1, 1)))
        else:
            src_peaks = np.vstack((src_peaks, np.hstack((np.array(line_features).reshape(-1, 1),
                                   which_line*np.ones((len(line_features))).reshape(-1, 1)))))

            dst_peaks = np.vstack((dst_peaks, np.hstack((np.array(line_features_sym).reshape(-1, 1),
                                   which_line*np.ones((len(line_features))).reshape(-1, 1)))))

        cnt += 1

u = dst_peaks - src_peaks
```

In [1032]:
```
fig, ax = plt.subplots(dpi = 150, figsize=(4, 3))

#

ax.plot(src_peaks[:, 0], src_peaks[:, 1], 'r.')
ax.plot(dst_peaks[:, 0], dst_peaks[:, 1], 'k.')

# ax.plot(line_plot[line_features], line_sample[line_features], 'r.')
# ax.plot(line_plot, line_sample_sym)
# ax.plot(line_plot[line_features_sym], line_sample_sym[line_features_sym], 'k.')

ax.axis('equal')
```

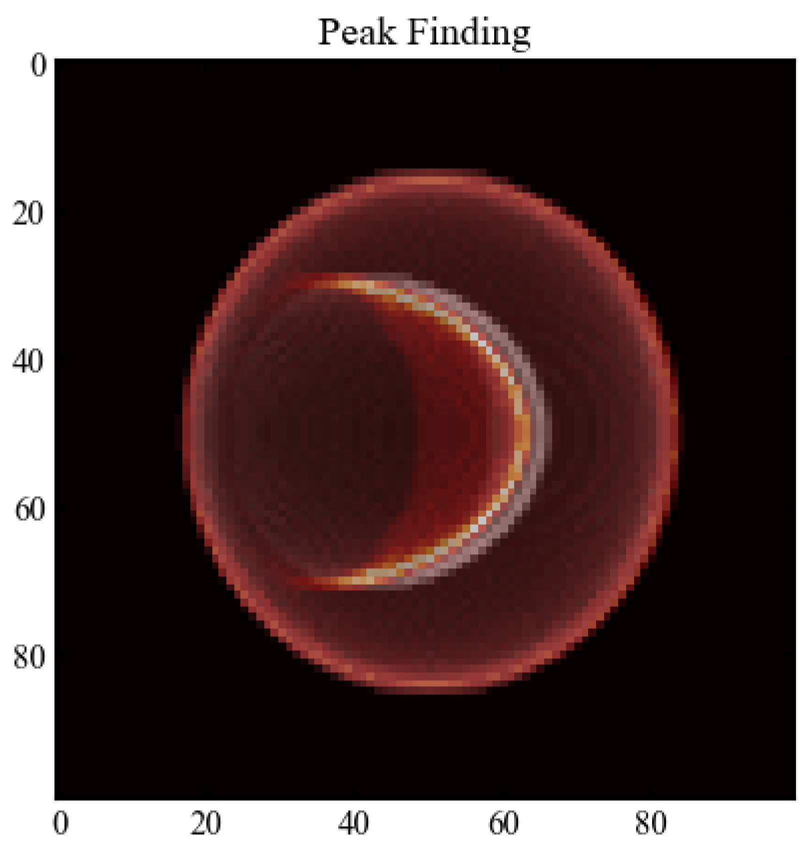
Out[1032]: (7.1, 92.9, 15.9, 84.1)



In [1056]:
```
tformEW_peaks = ProjectiveTransform()
tformEW_peaks.estimate(src_peaks, src_peaks + u*1.3)
warped_peaks = warp(end_view_sym_numpy, tformEW_peaks, output_shape=end_view_sym_numpy.shape, order = 3)
```

In [1114]:
```
plt.imshow(end_view_numpy, cmap = 'gist_heat')
plt.imshow(test_sym, alpha = 0.5, cmap = 'gray')
# plt.imshow(warped_peaks, alpha = 0.5, cmap = 'gray')
plt.title('Peak Finding')

# plt.plot(src_peaks[:, 0], src_peaks[:, 1], 'b.')
# plt.plot(dst_peaks[:, 0], dst_peaks[:, 1], 'r.')
```

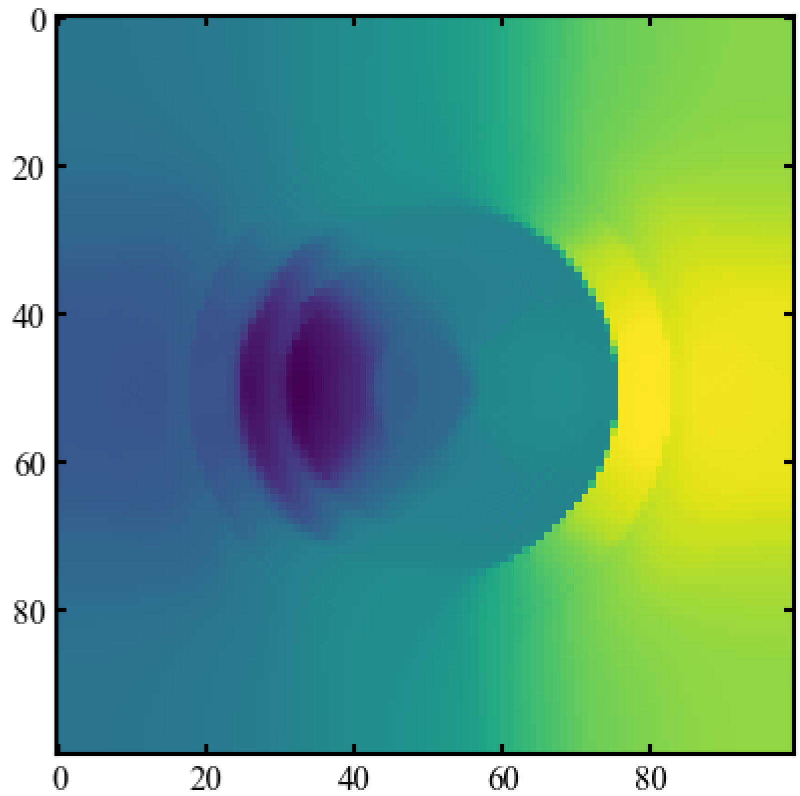
Out[1114]: Text(0.5, 1.0, 'Peak Finding')



In [672]:
```
u_numpy, v_numpy =optical_flow_tvl1(gaussian(end_view_sym_numpy, 2), gaussian(end_view_numpy, 2))
```

In [673]:
```
plt.imshow(v_numpy)
```

Out[673]: <matplotlib.image.AxesImage at 0x219adaf3b20>

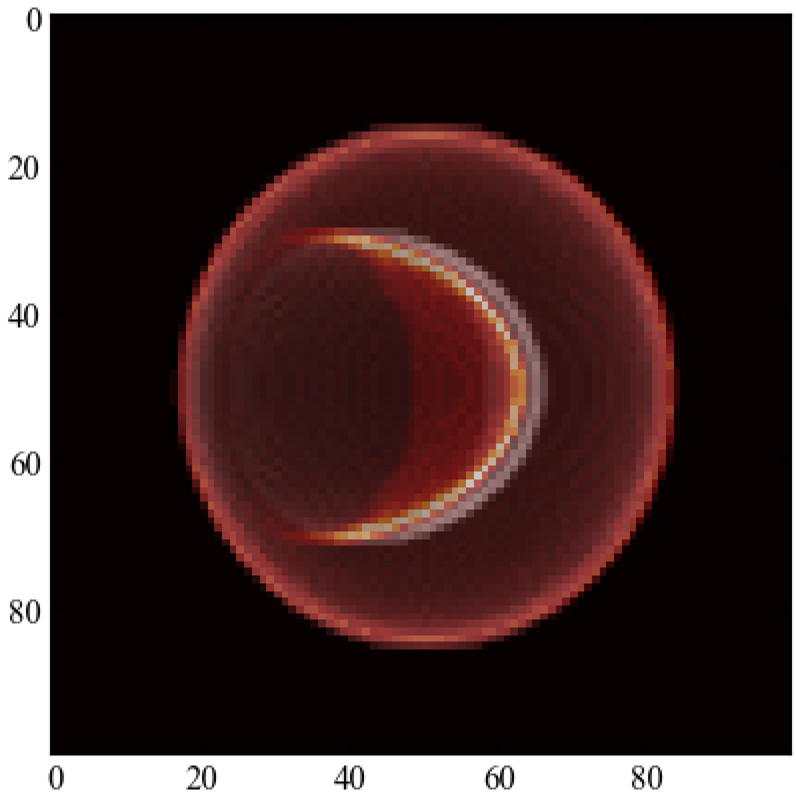


In [798]:
```
skip = 10

src_numpy = np.array([0, 0])
dst_numpy = np.array([0, 0])
for i in np.arange(skip, end_view_sym.shape[0]+skip, skip):
    for j in np.arange(skip, end_view_sym.shape[1]+skip, skip):
        if (i-1-50)**2 + (j -1 - 50)**2 >=10**2:
            src_numpy = np.vstack((src_numpy, np.array([j-1, i-1])))
            dst_numpy = np.vstack((dst_numpy, np.array([j-1 + np.flip(v_numpy)[i-1, j-1]*-1.5, i-1 + u_numpy[i
        else:
            src_numpy = np.vstack((src_numpy, np.array([j-1, i-1])))
            dst_numpy = np.vstack((dst_numpy, np.array([j-1 + np.flip(v_numpy)[i-1, j-1]*-8, i-1 + u_numpy[i-1
```

In [799]:
```
tformEW_numpy = ProjectiveTransform()
tformEW_numpy.estimate(dst_numpy, src_numpy)
warped_numpy = warp(end_view_sym_numpy, tformEW_numpy, output_shape=end_view_sym_numpy.shape, order = 1)
```

In [1115]:
```
plt.imshow(end_view_numpy, cmap = 'gist_heat')
plt.imshow(test_sym, alpha = 0.5, cmap = 'gray')
# plt.imshow(np.flip(warped_numpy), alpha = 0.5, cmap = 'gray')
# plt.title('Numpy')

# for si in range(len(src_numpy)):
#     plt.plot(src_numpy[si, 0], src_numpy[si, 1], 'b.')
#     plt.plot(dst_numpy[si, 0], dst_numpy[si, 1], 'r.')
```

Out[1115]: <matplotlib.image.AxesImage at 0x219c31ca170>

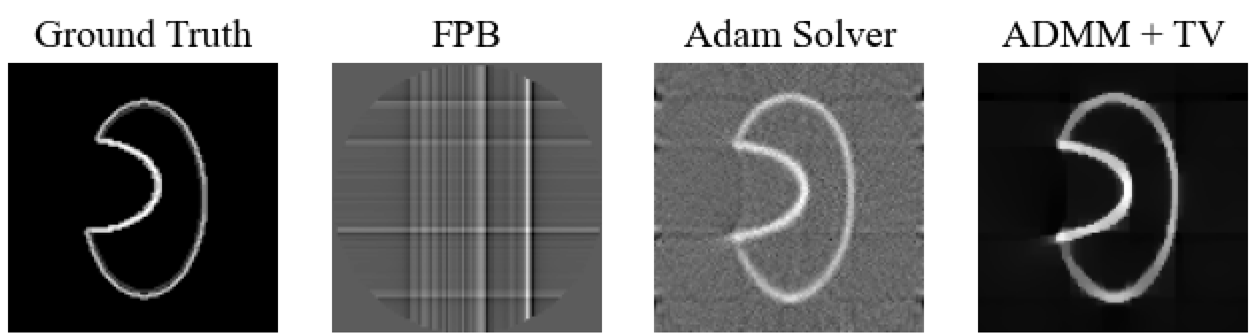


In [1107]:
```
save_dir = r"G:\My Drive\BoxMigration\Jackie Research\3D flame surface reconstruction"

save_name = '\\image_cube_adam.npy'
np.save(save_dir+save_name, image_cube_adam)
```

Compare results from different techniques

In [1147]:
```
fig, axes = plt.subplots(1, 4, figsize=(8, 10))

axes[0].imshow(image_cube_grad[35, ...], cmap = 'gray')
axes[0].set_title('Ground Truth')
axes[0].axis('off')

axes[1].imshow(image_cube_fbp[35, ...], cmap = 'gray')
axes[1].set_title(f"FPB")
axes[1].axis('off')

axes[2].imshow(image_cube_adam_morph[35, ...], cmap = 'gray')
axes[2].set_title(f"Adam Solver")
axes[2].axis('off')

axes[3].imshow(image_cube_admm_tv[35, ...], cmap = 'gray')
axes[3].set_title(f"ADMM + TV")
axes[3].axis('off')
```

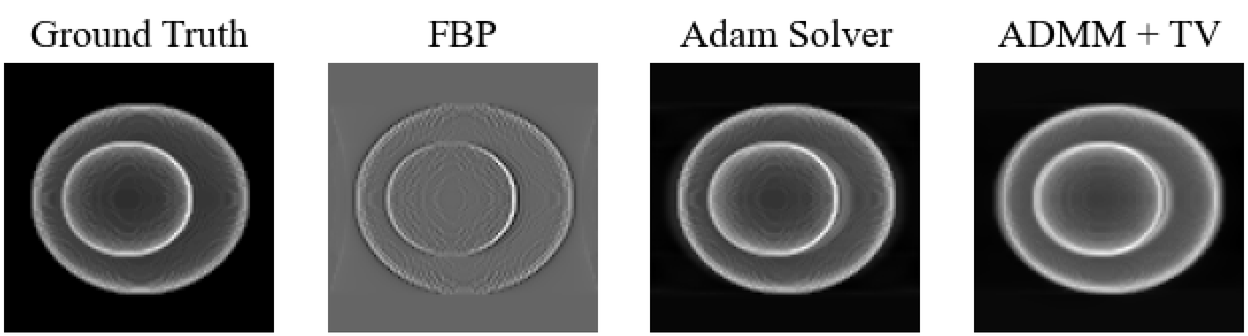
Out[1147]: (-0.5, 99.5, 99.5, -0.5)



In [1244]:

```
fig, axes = plt.subplots(1, 4, figsize=(8, 10))
theta_to_probe = 90

axes[0].imshow(radon3D_torch(image_cube_grad_torch, theta = theta_to_probe), cmap = 'gray')
axes[0].set_title('Ground Truth')
axes[0].axis('off')

axes[1].imshow(radon3D_numpy(image_cube_fbp, theta = theta_to_probe), cmap = 'gray')
axes[1].set_title('FBP')
axes[1].axis('off')

reg_EW = radon3D_numpy(image_cube_adam_morph, theta = theta_to_probe)
# reg_EW = warp(reg_EW.numpy(), tformEW, output_shape=reg_list_SW[-1].shape)

axes[2].imshow(reg_EW, cmap = 'gray')
axes[2].set_title('Adam Solver')
axes[2].axis('off')

# axes[1].imshow(subtracted, cmap = 'gray')
axes[3].imshow(radon3D_numpy(image_cube_admm_tv, theta = theta_to_probe), cmap ='gray')
axes[3].set_title(f"ADMM + TV")
axes[3].axis('off')
```

Out[1244]: (-0.5, 99.5, 99.5, -0.5)



In [1245]:

```
from skimage.metrics import peak_signal_noise_ratio

print(peak_signal_noise_ratio(np.clip(radon3D_torch(image_cube_grad_torch, theta = theta_to_probe).numpy().ast
      np.clip(radon3D_numpy(image_cube_fbp, theta = theta_to_probe), 0, 1)))

print(peak_signal_noise_ratio(np.clip(radon3D_torch(image_cube_grad_torch, theta = theta_to_probe).numpy().ast
      np.clip(radon3D_numpy(image_cube_adam_morph, theta = theta_to_probe), 0, 1)))

print(peak_signal_noise_ratio(np.clip(radon3D_torch(image_cube_grad_torch, theta = theta_to_probe).numpy().ast
      np.clip(radon3D_numpy(image_cube_admm_tv, theta = theta_to_probe), 0, 1)))
```

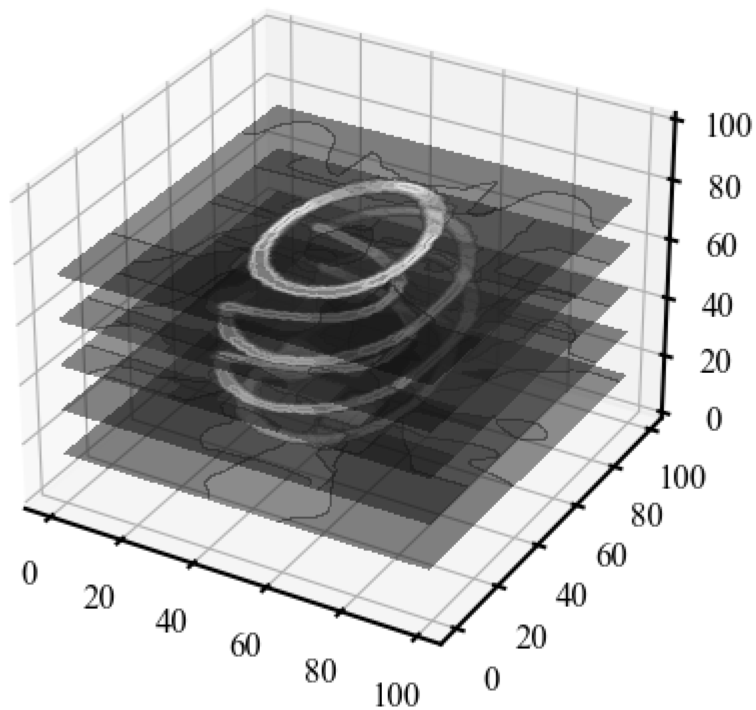
```
5.479495802460672
18.149778689736163
16.81717100523467
```

In [1220]:

```
%matplotlib inline

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from matplotlib import cm

# create a 21 x 21 vertex mesh
xx, yy = np.meshgrid(np.arange(0,image_cube_admm_tv.shape[1],1),
                         np.arange(0,image_cube_admm_tv.shape[2],1))

# create the figure
fig = plt.figure()

# show the reference image
# ax1 = fig.add_subplot(121)
# ax1.imshow(data, cmap=plt.cm.BrBG, interpolation='nearest', origin='lower', extent=[0,1,0,1])

# show the 3D rotated projection
ax2 = fig.add_subplot(111, projection='3d')
for h in np.arange(15, 90, 15):
    cset = ax2.contourf(xx, yy, image_cube_admm_tv[h, ...], 10, zdir='z', offset=h, cmap='gray', alpha = 0.5)

# cset = ax2.contourf(xx, image_cube_admm_tv[:, 50, :], yy, 100, zdir='x', offset=50, cmap='gray', alpha = 0.5

ax2.set_zlim((0.,100))

# plt.colorbar(cset)
# plt.show()
```

Out[1220]:

```
(0.0, 100.0)
```


0
20
40
60
80
100
0
20
40
60
80
100
0
20
40
60
80
100

In [1218]:
```
# import plotly.graph_objects as go
# import numpy as np
# X, Y, Z = np.meshgrid(np.arange(0,image_cube_admm_tv.shape[1],1),
#                            np.arange(0,image_cube_admm_tv.shape[2],1),
#                        np.arange(0,image_cube_admm_tv.shape[0],1))

# fig = go.Figure(data=go.Volume(
#     x=X.flatten(),
#     y=Y.flatten(),
#     z=Z.flatten(),
#     value=image_cube_admm_tv.flatten(),
#     isomin=-0.1,
#     isomax=0.8,
#     opacity=0.1, # needs to be small to see through all surfaces
#     surface_count=21, # needs to be a large number for good volume rendering
#     ))
# fig.show()
```

In [1219]:
```
# import plotly.graph_objects as go
# import numpy as np
# X, Y, Z = np.meshgrid(np.arange(0,image_cube_admm_tv.shape[1],1),
#                            np.arange(0,image_cube_admm_tv.shape[2],1),
#                        np.arange(0,image_cube_admm_tv.shape[0],1))

# fig = go.Figure(data=go.Volume(
#     x=X.flatten(),
#     y=Y.flatten(),
#     z=Z.flatten(),
#     value=image_cube_grad.flatten(),
#     isomin=-0.1,
#     isomax=0.8,
#     opacity=0.1, # needs to be small to see through all surfaces
#     surface_count=21, # needs to be a large number for good volume rendering
#     ))
# fig.show()
```

In [ ]: