```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from scipy.sparse import dok_matrix, coo_matrix
from scipy.sparse.linalg import lsqr
```

1 Constructing the matrix X

First, here is a loop-based function constructing the matrix in **D**ictionary **O**f **K**eys format (`scipy.sparse.dok_matrix`). While it is slightly more clear what the code does, this function takes long to compute and can use up a lot of memory as well.

```
def construct_X_slow(M, alphas, Np = None):
    # define sensor size
    if Np is None:
        Np = int(np.ceil(np.sqrt(2) * M))
        if Np % 2 == 0: Np += 1

    # create sparse matrix X [illegible]
    X = dok_matrix((M*M, len(alphas)*Np), dtype = np.float32)

    # go through all measurement angles
    for i, alpha in enumerate(alphas):
        # convert angle and prepare projection vector
        alph_rad = np.radians(alpha)
        [illegible]

        # go through all coordinates (x,y) in the image
        for x in range(M):
            for y in range(M):
                # center the coordinates ...
                [illegible]
                # ... and rotate them to align them with the sensor array for this measurement
                [illegible]
                # where does the coordinate fall in the sensor array?
                [illegible]

                # find the two neighboring sensor bins
                bin0 = int(np.floor(pos))
                bin1 = int(np.ceil(pos))

                # [illegible]
                if bin0 == bin1:
                    X[y*M + x, i*Np + bin0] = 1.0
                else:
                    # interpolation coefficients
                    [illegible]

                    # [illegible]
                    [illegible]
    return X.T # maybe switch dimensions in the first place?
```

To construct $\mathbf{X}$ more efficiently, we can vectorize all the operations that use the same angle α, and project all image pixels onto the sensor array at once.

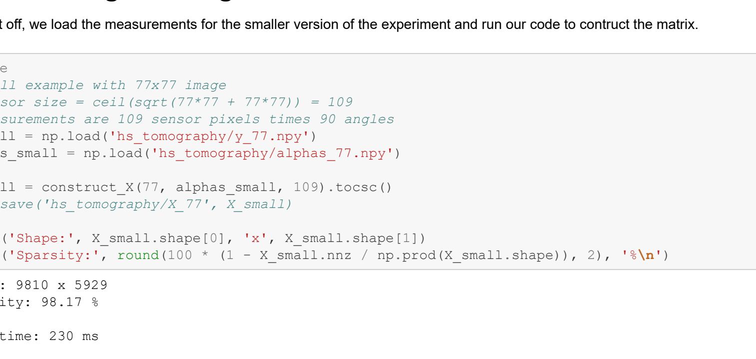
The values we want to put into the sparse matrix, as well as their indices i and j, are collected in three corresponding lists. This way we can use the more efficient **COO**rdinate format (`scipy.sparse.coo_matrix`) to assemble the matrix in the end.

```
In [ ]: def construct_X(M, alphas, Np = None):
    D = M*M
    # define sensor size
    if Np is None:
        Np = int(np.ceil(np.sqrt(2) * M))
        if Np % 2 == 0: Np += 1
    # number of angles
    No = len(alphas)

    # flattened output coordinates
    j = np.mgrid[0:D].astype(np.int32)
    # coordinate matrix for the output pixels
    M2 = (M-1) / 2
    grid = np.mgrid[-M2:M-M2,-M2:M-M2].swapaxes(1,2).reshape(2,D)

    # collect indices and corresponding values for all iterations
    i_indices = []
    j_indices = []
    weights = []

    for k, alpha in enumerate(alphas):
        # convert angle and prepare projection vector
        alph_rad = np.radians(alpha)
        proj_vec = np.array([np.cos(alph_rad), -np.sin(alph_rad)])
        # project coordinates
        proj = np.dot(proj_vec, grid) + Np // 2
        # compute sensor indices and weights below the projected points
        i = np.floor(proj)
        w = (i+1) - proj

        # make sure rays falling outside the sensor are not counted
        clip = np.logical_and(0 <= i, i < Np-1)

        i_indices.append((i + k*Np)[clip])
        j_indices.append(j[clip])
        weights.append(w[clip])
        # compute sensor indices and weights above the projected points
        w = proj - i
        i_indices.append((i+1 + k*Np)[clip])
        j_indices.append(j[clip])
        weights.append(w[clip])

    # construct matrix X
    i = np.concatenate(i_indices).astype(np.int32)
    j = np.concatenate(j_indices).astype(np.int32)
    w = np.concatenate(weights)
    X = coo_matrix((w, (i,j)), shape = (No*Np, D), dtype = np.float32)

    return X


# check correctness
a = [-90,-45,0,45]
X = construct_X(10, a)
XS = construct_X_slow(10, a)
print('Loop-based and vectorized matrix give', 'same' if (X != XS).size == 0 else 'different', 'results!')
```

```
Loop-based and vectorized matrix give same results!
```

We visualize the $\mathbf{X}$ for $M = 10$, $N_p = 15$ and three projections at angles $(-33^\circ, 1^\circ, 42^\circ)$, as shown on the exercise sheet:

```
In [ ]: X = construct_X(10, [-33, 1, 42]).todense()
# np.save('X_example', X)

fig = plt.figure(figsize = (10, 4.5))
plt.imshow(X, interpolation = 'nearest')
plt.gray(); plt.axis('off'); fig.tight_layout(); plt.show()
```



2 Recovering the image

To start off, we load the measurements for the smaller version of the experiment and run our code to contruct the matrix.

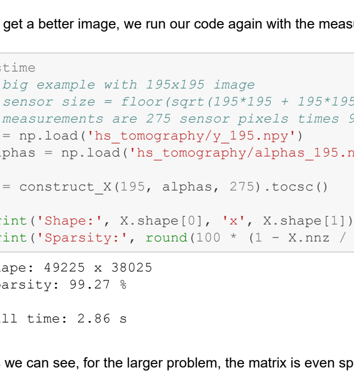
```
In [ ]: %%time
# small example with 77x77 image
# sensor size = ceil(sqrt(77*77 + 77*77)) = 109
# measurements are 109 sensor pixels times 90 angles
y_small = np.load('hs_tomography/y_77.npy')
alphas_small = np.load('hs_tomography/alphas_77.npy')

X_small = construct_X(77, alphas_small, 109).tocsc()
# np.save('hs_tomography/X_77', X_small)

print('Shape:', X_small.shape[0], 'x', X_small.shape[1])
print('Sparsity:', round(100 * (1 - X_small.nnz / np.prod(X_small.shape)), 2), '%\n')
```

```
Shape: 9810 x 5929
Sparsity: 98.17 %

Wall time: 230 ms
```

The *sparsity* denotes what ratio of entries of the matrix is zero — in this case, it's about $98.17\,\%$. Note that the opposite, the ratio of non-zero entries, is called the *density* of the matrix.

Plugging the matrix and the measurements into `scipy.sparse`'s least squares solver, we obtain a (relatively low resolution) tomography image:

```
In [ ]: %%time
beta_small = lsqr(X_small, y_small, atol = 1e-5, btol = 1e-5)[0].reshape(77,77)

fig = plt.figure(figsize = (4,4))
plt.imshow(beta_small, vmin = 0, vmax = 255, interpolation = 'nearest')
plt.gray(); plt.axis('off'); fig.tight_layout(); plt.show()
```



```
Wall time: 970 ms
```

To get a better image, we run our code again with the measurements from the larger version of the experiment.

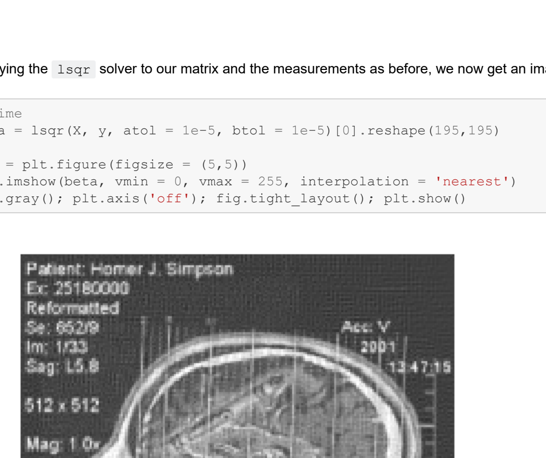
```
In [ ]: %%time
# big example with 195x195 image
# sensor size = floor(sqrt(195*195 + 195*195)) = 275
# measurements are 275 sensor pixels times 90 angles
y = np.load('hs_tomography/y_195.npy')
alphas = np.load('hs_tomography/alphas_195.npy')

X = construct_X(195, alphas, 275).tocsc()

print('Shape:', X.shape[0], 'x', X.shape[1])
print('Sparsity:', round(100 * (1 - X.nnz / np.prod(X.shape)), 2), '%\n')
```

```
Shape: 49225 x 38025
Sparsity: 99.27 %

Wall time: 2.86 s
```

As we can see, for the larger problem, the matrix is even sparser.

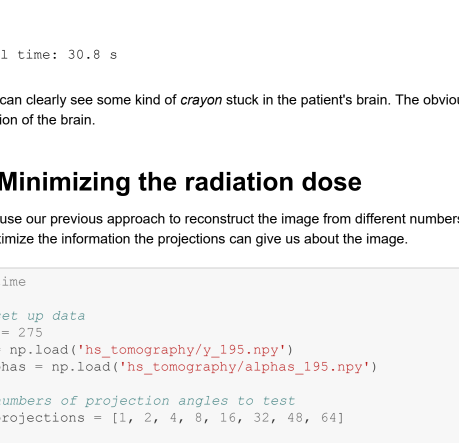
At this point, we can also visualize $\mathbf{y}$ in the shape of a sinogram, as shown on the exercise sheet:

```
In [ ]: fig = plt.figure(figsize = (6,5))
plt.imshow(np.load('hs_tomography/y_195.npy').reshape(179,275).T, interpolation = 'nearest', aspect = 'auto')
plt.gray(); plt.axis('off'); fig.tight_layout(); plt.show()
```



Applying the `lsqr` solver to our matrix and the measurements as before, we now get an image of much higher quality:

```
In [ ]: %%time
beta = lsqr(X, y, atol = 1e-5, btol = 1e-5)[0].reshape(195,195)

fig = plt.figure(figsize = (5,5))
plt.imshow(beta, vmin = 0, vmax = 255, interpolation = 'nearest')
plt.gray(); plt.axis('off'); fig.tight_layout(); plt.show()
```



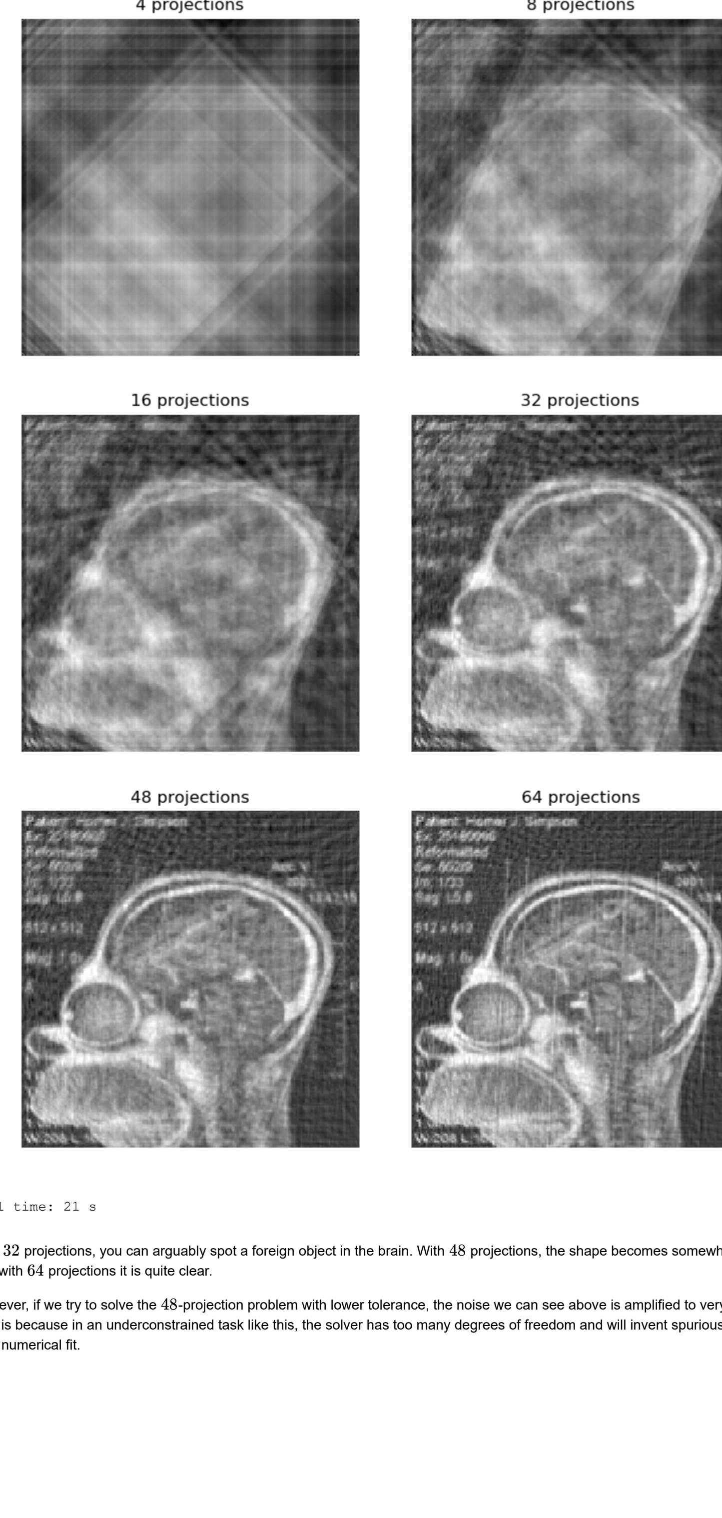

```
Wall time: 30.8 s
```

We can clearly see some kind of crayon stuck in the patient's brain. The obvious treatment would be to surgically remove the damaged portion of the brain.

3 Minimizing the radiation dose

We use our previous approach to reconstruct the image from different numbers of projections. The angles are as spread out as possible to maximize the information the projections can give us about the image.

```
In [ ]: %%time

# set up data
Np = 275
y = np.load('hs_tomography/y_195.npy')
alphas = np.load('hs_tomography/alphas_195.npy')

# numbers of projection angles to test
n_projections = [1, 2, 4, 8, 16, 32, 48, 64]

fig, axes = plt.subplots(4, 2, figsize = (6,16))

for n in range(len(n_projections)):
    # pick specified number of angles
    index = [int(np.ceil(len(alphas) * p/n_projections[n])) for p in range(n_projections[n])]
    alphas_sub = alphas[index]

    # collect corresponding measurements from y
    y_sub = []
    for i in index:
        y_sub.extend(y[i*Np : (i+1)*Np])

    # construct matrix and reconstruct image
    X = construct_X(195, alphas_sub, Np).tocsc()
    beta = lsqr(X, np.array(y_sub), atol = 1e-5, btol = 1e-5)[0].reshape(195,195)

    # plot image
    axes.flat[n].imshow(beta, vmin = 0, vmax = 255, interpolation = 'nearest')
    axes.flat[n].set_title('{} projections'.format(n_projections[n])); axes.flat[n].axis('off')

fig.tight_layout()
plt.show()
```



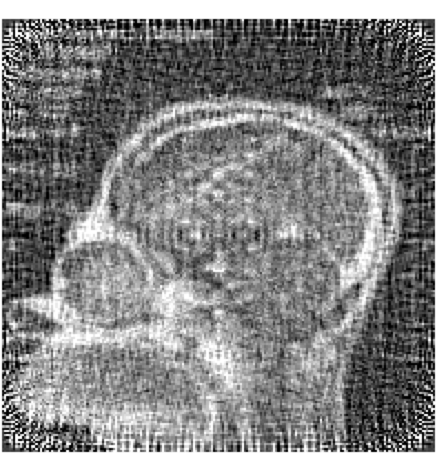

```
Wall time: 21 s
```

With 32 projections, you can arguably spot a foreign object in the brain. With 48 projections, the shape becomes somewhat recognizable, and with 64 projections it is quite clear.

However, if we try to solve the 48-projection problem with lower tolerance, the noise we can see above is amplified to very heavy artifacts. This is because in an underconstrained task like this, the solver has too many degrees of freedom and will invent spurious data to reach the best numerical fit.

In [ ]:

```
%%time

# set up data
Np = 275
y = np.load('hs_tomography/y_195.npy')
alphas = np.load('hs_tomography/alphas_195.npy')

# pick the 48 angles
index = [int(np.ceil(len(alphas) * p/48)) for p in range(48)]
alphas_sub = alphas[index]

# collect corresponding measurements from y
y_sub = []
for i in index:
    y_sub.extend(y[i*Np : (i+1)*Np])

# construct matrix and reconstruct image exactly
X = construct_X(195, alphas_sub, Np).tocsc()
beta = lsqr(X, np.array(y_sub), atol = 1e-7, btol = 1e-7)[0].reshape(195,195)

# plot image
fig = plt.figure(figsize = (5,5))
plt.imshow(beta, vmin = 0, vmax = 255, interpolation = 'nearest')
plt.gray(); plt.axis('off'); fig.tight_layout(); plt.show()
```



```
Wall time: 46min 1s
```