Image reconstruction in X-ray tomography

Henrique Miyamoto and Ilyas Hanine

1. Introduction

In the context of X-ray tomography, in order to reconstruct dense volume objects, a set of measurements $y \in \mathrm{R}^M$ are taken, which are related to the sought abosrption image $x \in \mathrm{R}^N$ by the relation

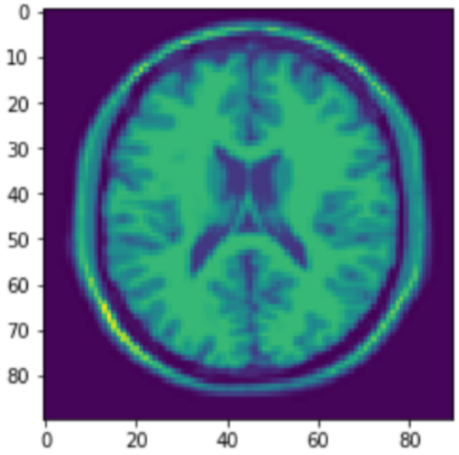
$$y = Hx + w,$$

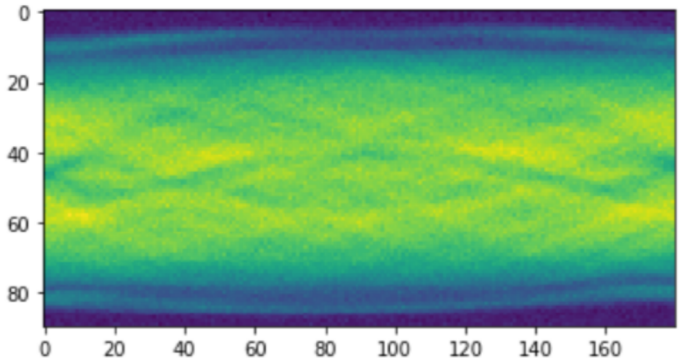
where $w \in \mathrm{R}^M$ is an i.i.d. Gaussian noise with variance σ^2 and $H \in \mathrm{R}^{M \times N}$ is the tomography matrix, which models parallel projections of a 2D object x.

In [1]:
```
# Import data from MATLAB files

import scipy.io

H = scipy.io.loadmat('H.mat')['H']
G = scipy.io.loadmat('G.mat')['G']
x = scipy.io.loadmat('x.mat')['x']
```

In [3]:

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

# Build signal and display it

sigma = 1
y = H @ x + sigma * np.random.randn(16200,1)

N = len(x)
M = len(y)

xresh = np.reshape(x, (90,90), order='F')
yresh = np.reshape(y, (90,180), order='F')

# For some reason, images only appear the second time the c
ell is run
plt.imshow(xresh);
plt.show();

plt.imshow(yresh);
plt.show();
```





2. Optimisation problem

The reconstruction problem may be addressed as an optimisation problem with penalised least-squares criterion:

$$\hat{x} = \min_{x \in \mathrm{R}^N} f(x) = \min_{x \in \mathrm{R}^N} \frac{1}{2} \|Hx - y\|^2 + \lambda r(x),$$

where the regularisation function $r(x)$ is set as

$$r(x) = \sum_{n=1}^{2N} \psi([Gx]^{(n)}),$$

where $G \in \mathrm{R}^{2N \times N}$ is a sparse matrix and ψ is a pontential function defined as

$$\psi(u) = \sqrt{1 + \frac{u^2}{\delta^2}},$$

for some $\delta > 0$ that guarantees the differentiability of r. Let us set $\lambda = 0.13$ and $\delta = 0.02$.

The gradient of $f(x)$ is

$$\begin{aligned}
\nabla f(x) &= H^T Hx + H^T y + \lambda \nabla r(x) \\
&= H^T Hx + H^T y + \lambda \left(\frac{\partial}{\partial x^{(i)}} \sum_{n=1}^{2N} \psi([Gx]^{(n)}) \right)_{1 \le i \le N} \\
&= H^T Hx + H^T y + \lambda G^T \psi'(Gx) \\
&= H^T Hx + H^T y + \lambda G^T \left(\frac{[Gx]^{(n)}}{\delta^2 \sqrt{1 + ([Gx]^{(n)}/\delta)^2}} \right)_{1 \le n \le 2N}
\end{aligned}$$

The following code implements the calculation of this gradient.

In [4]:
```
Ht = np.transpose(H)
Gt = np.transpose(G)

lmb = 0.13
delta = 0.02


def gradf(u):
    u = np.reshape(u, (N,1))
    dpsi = lambda u : u/((delta**2) * np.sqrt(1+(u/delta)**
2))
    grad_r = lmb * Gt.dot(dpsi(G.dot(u)))
    return Ht.dot(H.dot(u)-y) + grad_r
```

The function $\nabla f(x)$ is Lipschitz-continuous with Lipdchitz constant $L = \|H\|^2 + (\lambda/\delta^2)\|G\|^2$, since:

$$
\begin{aligned}
&\|\nabla f(x_1) - \nabla f(x_2)\| \\
&= \|H^T H x_1 + H^T y + \lambda G^T \psi'(Gx_1) - H^T H x_2 - H^T y - \lambda G^T \psi'(Gx_2)\| \\
&= \|H^T H(x_1 - x_2) + \lambda G^T(\psi'(Gx_1) - \psi'(Gx_2))\| \\
&\leq \|H\|^2 \|x_1 - x_2\| + \| \frac{\lambda G^T}{\delta^2} \left(\frac{[Gx_1]^{(n)}}{\sqrt{1 + ([Gx_1]^{(n)}/\delta)^2}} - \frac{[Gx_2]^{(n)}}{\sqrt{1 + ([Gx_2]^{(n)}/\delta)^2}} \right)_{1 \leq n \leq 2N} \| \\
&\leq \|H\|^2 \|x_1 - x_2\| + \left| \frac{\lambda}{\delta^2} \right| \|G^T G\| \|x_1 - x_2\| \\
&= \underbrace{\left(\|H\|^2 + \frac{\lambda}{\delta^2} \|G\|^2 \right)}_{L} \|x_1 - x_2\|
\end{aligned}
$$

The function below computes the value of the constant for $\lambda = 0.13$ and $\delta = 0.02$.

In [6]:
```
# Computes the value of Lipschitz constant L

normH  = scipy.sparse.linalg.svds(H, k=1, which='LM', retur
n_singular_vectors=False)[0]
normG = scipy.sparse.linalg.svds(G, k=1, which='LM', return
_singular_vectors=False)[0]
L = normH**2 + (lmb/delta**2) * normG**2
print(L)
```

```
18092.7732769
```

3. Optimisation algorithms

We consider different approaches to solve the optimisation problem.

To compare their performance, we evaluate the behaviour of the value of $f(x)$ along the iterations of each algorithm, until the following stopping criterion is satisfied:

$$\|\nabla f(x_n)\| \leq \sqrt{N} \times 10^{-4}.$$

In addition, we compute the signal-to-noise ratio (SNR), defned as

$$SNR = 10\log_{10}\left(\frac{\|x\|^2}{\|x - \hat{x}\|^2}\right).$$

In [7]:
```
def cost(u):
    return (0.5)*np.linalg.norm(H.dot(u)-y,2) + lmb*sum(np.
sqrt(1+(G.dot(u)/delta)**2))

def SNR(u):
    return 10*(np.log(np.linalg.norm(x)**2/np.linalg.norm
(u-x)**2)/np.log(10))
```

Gradient descent

We first consider a gradient descent (fixed step) algorithm, which is defined by the following recursive relation:

$$x_{n+1} = x_n - \gamma \nabla f(x_n),$$

where $\gamma \in]0, 2/L[$.

In [8]:
```
# Gradient descent algorithm

import time

def gradDesc(show = False):
    itmax = 10000
    converged = False
    it = 0
    xn = np.zeros((N,1))
    gamma = 0.99*(2/L)
    tol = 1e-4
    start = time.time()

    #Storage of the results:
    timing = [0]
    value = [cost(xn)]
    sol = [xn]

    while ((not converged) and (it < itmax)):
        it += 1
        gradfn = gradf(xn)
        xnn = xn - gamma * gradfn

        timing.append(-start+time.time())
        value.append(cost(xnn))
        sol.append(xnn)

        if np.linalg.norm(gradfn) < tol*np.sqrt(N):
            converged = True
        xn = xnn

    if(show):
        print("Converged: " + str(converged))
        print("Number of iterations: " + str(it))

    return [sol, value, timing]
```

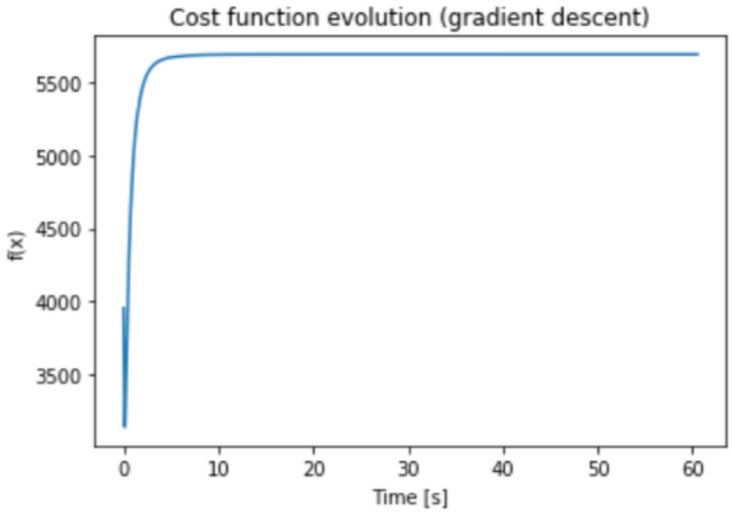
In [9]:
```
# Run reconstruction with gradient descent algorithm
[x_gd,v_gd,t_gd] = gradDesc(True)
```

```
Converged: True
Number of iterations: 2377
```

In [10]:
```
# Evaulate SNR of gradient descent
print(SNR(x_gd[-1]))
```

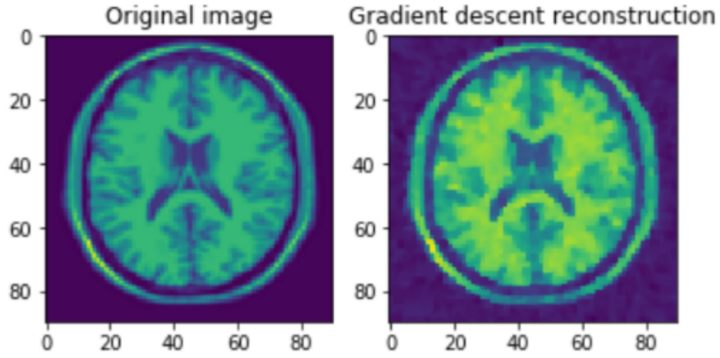
```
18.696564572
```

In [17]:
```
# Plot f(x_n) as function of timme
plt.plot(t_gd, v_gd);
plt.title('Cost function evolution (gradient descent)');
plt.ylabel('f(x)');
plt.xlabel('Time [s]');
plt.show();
```



In [12]:
```
# Show reconstruction with gradient descent

fig, ax = plt.subplots(1,2)
ax[0].set_title('Original image')
ax[0].imshow(xresh)
ax[1].set_title('Gradient descent reconstruction')
ax[1].imshow(x_gd[-1].reshape(90,90,order='F'))
```

Out[12]:
```
<matplotlib.image.AxesImage at 0x7fc0dd2bb898>
```



MM algorithm

We can consider the majorisation-minimisation (MM) quadratic algorithm defined by the equation

$$x_{n+1} = x_n - \theta_n A(X_n)^{-1} \nabla f(x_n), \quad \theta_n \in]0, 2[,$$

with $A(y)$ a strongly positive self-adjoint operator such that the quadratic function

$$h(x, y) = f(y) + \langle \nabla f(y) \,|\, x - y \rangle + \frac{1}{2} \|x - y\|^2_{A(y)}$$

is a majorant function of f at y for all $x \in \mathrm{R}^N$.

To construct such operator A, we divide it into two parts

$$A(x) = A_h(x) + A_r(x),$$

corresponding to the two terms of the objective function $f(x)$.

For the first part, it sufficed to consider the Hessian of the first term

$$A_h(x) = H^T H.$$

For the second part, consider the following:

1. $\psi(\sqrt{\cdot})$ is a concave function ;
2. ψ is positive ;
3. $\lim_{x \to 0^+} \frac{\psi(x)}{x} < +\infty$.

Then, we can consider the operator

$$\Delta(x) = \mathrm{diag}\left(\frac{\psi'(x_i)}{x_i} \right)_{1 \le i \le N}$$

such that

$$A_r(x) = \lambda G^T \Delta(Gx) G$$

has the desired majorisation properties.

Thus, the final form is

$$A(x) = H^T H + \lambda G^T \Delta(Gx) G.$$

In [13]:

```
# MM quadratic algorithm

from scipy.sparse.linalg import LinearOperator
import scipy.sparse
import time

def curv(u):
    u = np.reshape(u, (N,1))
    Gu = G.dot(u)
    Diag_psi = scipy.sparse.diags((1/np.sqrt(1+(Gu/delta)**
2))[:,0]).tocsc()
    Operator = lambda v: Ht.dot(H.dot(v)) + (lmb/delta**2)*
Gt.dot(Diag_psi.dot(G.dot(v)))
    A = LinearOperator((N,N), matvec = Operator, rmatvec =
Operator)
    return A

def MMquad(show = False):
    it = 0
    itmax = 10000
    converged = False
    xn = np.zeros((N,1))
    start = time.time()

    #Storage of the results:
    timing = [0]
    value = [cost(xn)]
    sol = [xn]

    while ((not converged) and (it < itmax)):
        it += 1
        xnn = xn - np.reshape(scipy.sparse.linalg.bicg(curv
(xn),gradf(xn))[0],(N,1))

        timing.append(-start+time.time())
        value.append(cost(xnn))
        sol.append(xnn)

        # gradf(xn) converges strongly to 0
        tol = np.linalg.norm(gradf(xn))
        if tol < np.sqrt(N)*1e-4:
            converged = True

        xn = xnn

    if(show):
        print("Converged: " + str(converged))
        print("Number of iterations: " + str(it))

    return [sol, value, timing]
```

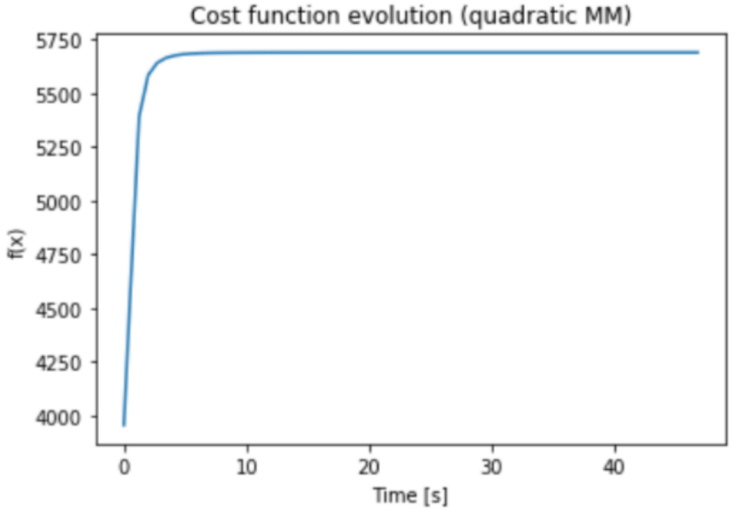
In [14]:
```
# Run reconstruction with quadratic MM
[x_mm,f_mm,t_mm] = MMquad(True)
```

```
Converged: True
Number of iterations: 64
```

In [15]:
```
# Evaulate SNR of quadratic MM
print(SNR(x_mm[-1]))
```

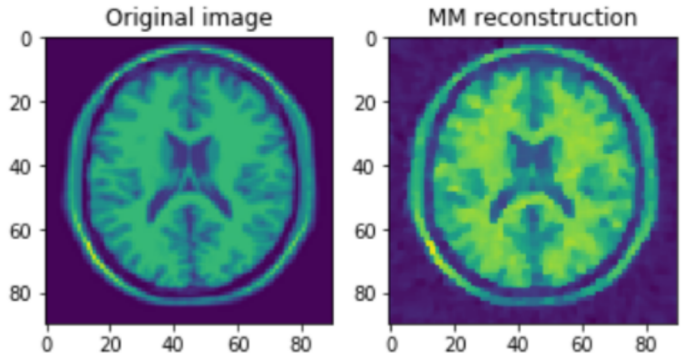
```
18.6965426606
```

In [19]:
```
# Plot f(x_n) as function of timme
plt.plot(t_mm, f_mm)
plt.title('Cost function evolution (quadratic MM)');
plt.ylabel('f(x)');
plt.xlabel('Time [s]');
plt.show();
```



In [18]:
```
# Show reconstruction with MM
fig, ax = plt.subplots(1,2)
ax[0].set_title('Original image')
ax[0].imshow(xresh)
ax[1].set_title('MM reconstruction')
ax[1].imshow(x_mm[-1].reshape(90,90,order='F'))
```

Out[18]: <matplotlib.image.AxesImage at 0x7fc0dd135e10>



Conclusion

We notice that both methods (gradient descent and quadratic MM) are able to reasonably reconstruct the original image, as can been infered from the reconstruction results. The final results have signal-to-noise ration of approximately 18.69.

The differente between the algorithms relies on the execution time. While gradient descent is a simplle algorithm that only requires computing the gradient of the cost function, it took 2377 iterations and about 60 seconds to converge. Quadratic MM, on the other hand, requires the construction of a majorant function as well, but only takes 64 iterations and less than 50 seconds to convege.

The test were run on an Intel Core i5-7200U CPU @ 2.50GHz machine.