Recontruction of DOSY NMR signals - Part I

Henrique Miyamoto and Ilyas Hanine

1. Introduction

In the context of the Diffusion Order Spectroscopy (DOSY) experiment, a series of Nuclear Magnetic Resonance (NMR) measurements is acquired, for different pulsed field gradient strenghts and the data is analysed in order to separate chemical species according to their diffusion coefficient.

The data $y = (y^{(m)})_{1 \le m \le M} \in \mathrm{R}^M$ gathers the result of M experiments characterised by paramter $t = (t^{(m)})_{1 \le m \le M} \in \mathrm{R}^M$, which is related to field strength and acquisition time. $y^{(m)}$ is the Laplace transform of $\chi(t^{(m)})$, where $\chi(\cdot)$ is the unknown diffusion distribution :

$$y^{(m)} = \mathcal{L}\{\chi(t^{(m)})\} = \int \chi(T)\exp(-t^{(m)}T)\mathrm{d}T.$$

The problem is then to reconstruct $\chi(T)$ from measurements y. To do so, we write data as

$$y = K\bar{x} + w,$$

where $\bar{x} \in \mathrm{R}^N$ is the sought signal, w is an additive noise and

$$K = \left(\exp(-T^{(n)}t^{(m)})\right)_{m,n}.$$

We estimate $\hat{x} \in \mathrm{R}^N$ as the solution of the minimisation problem

$$\hat{x} = \arg\min_{x \in \mathrm{R}^N} \frac{1}{2}\|Kx - y\|^2 + \beta g(x),$$

where $g \in \Gamma_0(\mathrm{R}^N)$ and $\beta \ge 0$ compose the regularisation term.

2. Generation of synthetic data

1. Import the diffusion signal R^N.

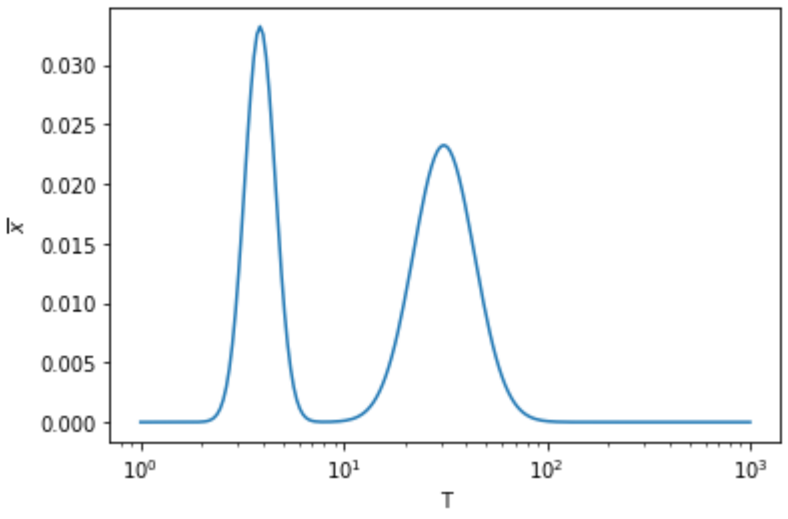
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.genfromtxt("x.txt", usecols=0, dtype=float)
N = len(x)
```

2. Create vector T with exponential sampling.

In [2]:
```
Tmin = 1
Tmax = 1000

T = [] * N
for n in range(1,N+1):
    T.append(Tmin*np.exp(-(n-1)*np.log(Tmin/Tmax)/(N-1)))
```

3. Display the original vector $\bar{x}$ as function of T.

In [3]:
```
plt.plot(T,x)
plt.xscale('log')
plt.xlabel('T');
plt.ylabel('$\overline{x}$');
```



4. Create t vector with regular sampling.

In [4]:
```
M = 50
tmin = 0
tmax = 1.5

t = [] * M
for m in range(1,M+1):
    t.append(tmin+(tmax-tmin)*(m-1)/(M-1))
```

5. Construct matrix $K = \left(\exp(-T^{(n)}t^{(m)})\right)_{m,n}$.

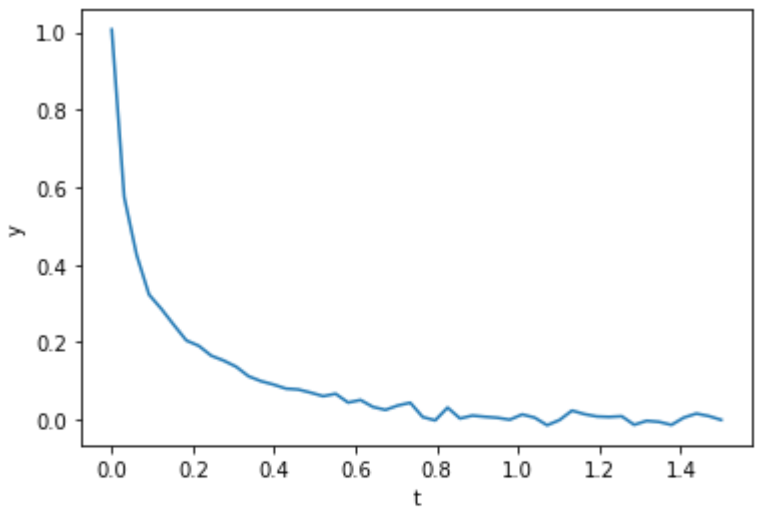
In [5]:
```
K = np.zeros((M, N))

for m in range(M):
    for n in range(N):
        K[m][n] = np.exp(-t[m]*T[n])
```

6. Add AWGN noise, by taking $w \sim \mathcal{N}(0, \sigma^2 I_M)$, with $\sigma = 0.01z^{(1)}$.

In [6]:
```
sigma = 0.01*(K@x)[0]
w = sigma*np.random.randn(M)
y = K @ x + w
```

7. Display noisy data y as function of t.

In [7]:
```
plt.plot(t,y)
plt.xlabel('t');
plt.ylabel('y');
```



3. Comparison of regularisation strategies

3.1. Smoothness prior

By considering the penalisation function

$$g(x) = \frac{1}{2}\|Dx\|^2, \qquad [Dx]^{(n)} = x^{(n)} - x^{(n-1)}, \quad x^{(0)} = x^{(N)},$$

the optimisation problem becomes

$$\hat{x} = \arg\min_{x \in \mathrm{R}^N} f(x) = \arg\min_{x \in \mathrm{R}^N} \frac{1}{2}\|Kx - y\|^2 + \frac{\beta}{2}\|Dx\|^2.$$

1. The set R^N being closed and the cost function $f(x)$ being in $\Gamma_0(\mathrm{R}^N)$ (i.e., convex, lower semi-continuous. and proper) and coercive (as it is the sum of such functions), there exists a solution for the minimisation problem. Furthermore, R^N being convex and $f(x)$ being strictly convex ($\|\cdot\|^2$ is strongly convex), the solution is unique.

2. In this case we can simply compute the solution by annulating the gradient $\nabla f(\hat{x}) = 0$. In matrice form, we have

$$f(x) = \frac{1}{2}x^T K^T K x - 2x^T K^T y - y^T y + \frac{\beta}{2}x^T D^T D x$$

and

$$\nabla f(x) = K^T K x - K^T y - \beta D^T D x.$$

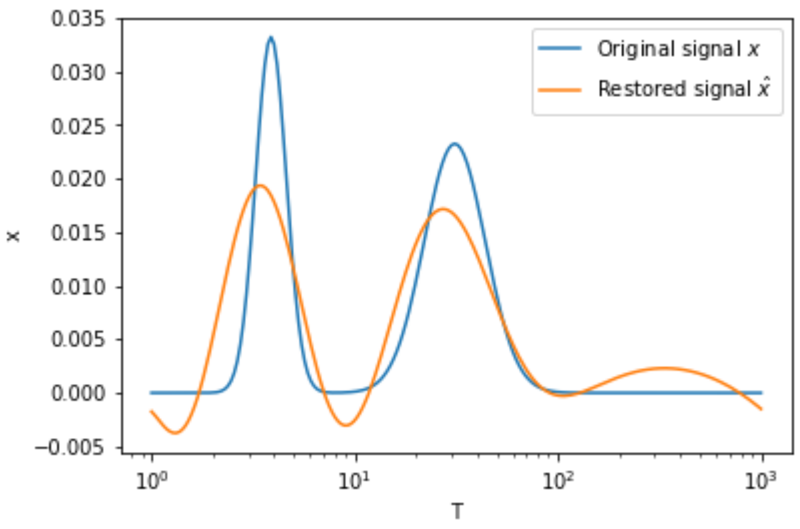
By imposing $\nabla f(\hat{x}) = 0$, we have

$$\hat{x} = (K^T K + \beta D^T D)^{-1} K^T y,$$

if the inverse exists.

3. Implement the solution, with $\beta = 1$.

```
In [8]: # Create matrix D
        D = np.eye(N,N,0) - np.eye(N,N,-1)
        D[0][N-1] = -1

        # Solve minimisation problem via smooth penalisation
        def smooth(beta):
            return np.linalg.inv(np.transpose(K)@K + beta*np.transp
        ose(D)@D)@np.transpose(K)@y

        # Use smooth penalisation with beta = 1
        xhat1 = smooth(1)

        plt.plot(T, x, label='Original signal $x$');
        plt.plot(T, xhat1, label='Restored signal $\hat{x}$');
        plt.xscale('log');
        plt.xlabel('T');
        plt.ylabel('x');
        plt.legend();
        plt.show()
```



4. Evalutate the normalised quadratic error.

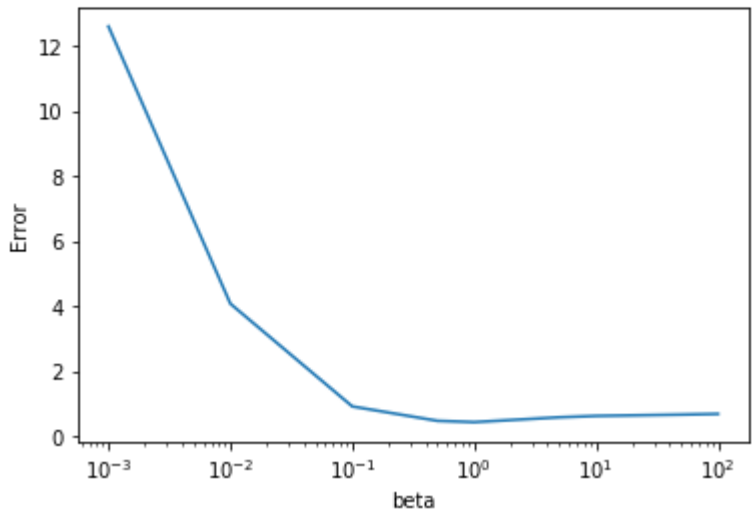
```
In [9]: def error(xbar,xhat):
            return np.linalg.norm(xhat - xbar)/np.linalg.norm(xbar)

        print(error(x,xhat1))
```

```
0.438829623253
```

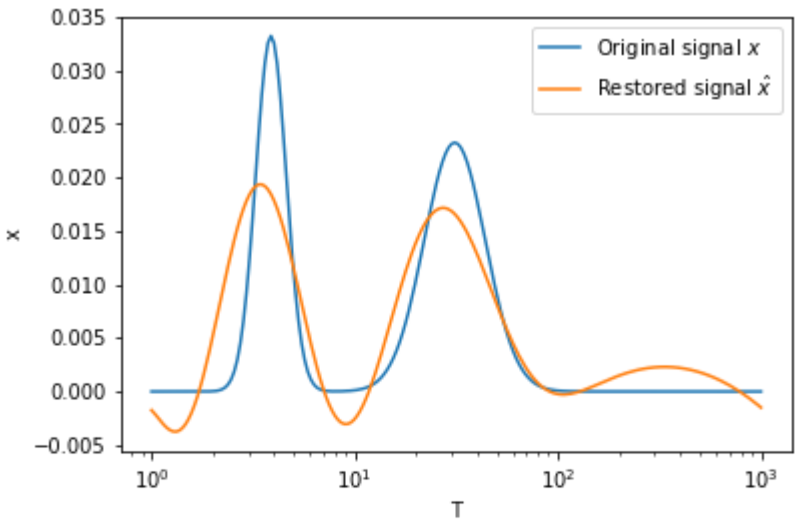
5. Search for the best value for β.

In [10]:
```
e_vector = []
b_vector = [1e-3, 1e-2, 1e-1, 0.5, 1, 5, 10, 50, 100]

# Test error for many values of beta
for beta in b_vector:
    e_vector.append(error(x,smooth(beta)))

# Choose best value of beta
e_min = min(e_vector)
b_min = b_vector[np.argmin(e_vector)]

plt.plot(b_vector, e_vector)
plt.xscale('log');
plt.xlabel('beta');
plt.ylabel('Error');

print("Minimum error = {}, beta = {}".format(e_min,b_min))
```

```
Minimum error = 0.43882962325277086, beta = 1
```



In [11]:
```python
# Use smooth penalisation with best beta
xhat1 = smooth(b_min)

plt.plot(T, x, label='Original signal $x$');
plt.plot(T, xhat1, label='Restored signal $\hat{x}$');
plt.xscale('log');
plt.xlabel('T');
plt.ylabel('x');
plt.legend();
plt.show()
```



3.2. Smoothness prior + constraints

In this case, we consider the penalisation function

$$g(x) = \frac{1}{2}\|Dx\|^2 + \iota_{[x_{\min};x_{\max}]^N}(x),$$

with D as previously defined and $x_{\min}, x_{\max}$ the minimum and maximum values of original singal $x$ , respectively. The optimisation problem becomes

$$\hat{x} = \underset{x\in\mathbb{R}^N}{\arg\min}\ f(x) = \underset{x\in\mathbb{R}^N}{\arg\min}\ \frac{1}{2}\|Kx-y\|^2 + \beta\left(\frac{1}{2}\|Dx\|^2 + \iota_{[x_{\min};x_{\max}]^N}(x)\right).$$

1. The set $\mathbb{R}^N$ being closed and the cost function $f(x)$ being in $\Gamma_0(\mathbb{R}^N)$ and coercive (as it is the sum of such functions), there exists a solution for the minimisation problem. As the characteristic function $\iota_C(x),\ C\subset\mathbb{R}^N$ is not strictly convex, we cannot guarantee uniqueness of the solution.

2. We can use the *projected gradient algorithm*, as

- $C = [x_{\min}; x_{\max}]^N$ is a nonempty closed convex subset of R^N ;
- $f(x) \in \Gamma_0(\mathrm{R}^N)$ is differentiable with ν-Lipschitzian gradient :

$$\|\nabla f(x_1) - \nabla f(x_2)\| = \|(K^T K - \beta D^T D)(x_1 - x_2)\| \leq \max \sigma_i(K^T K - \beta D^T D)\|x_1 - x_2\| \Rightarrow \nu = \max \sigma_i(K^T K -$$

By choosing $\gamma \in]0, 2/\nu[$, $\delta = 2 - \gamma\nu/2 \in]1, 2[$, $\lambda \in [0, \delta]$ and $x_0 \in \mathrm{R}^N$, we can apply

$$\begin{cases} y_n = x_n - \gamma \nabla f(x_n) \\ x_{n+1} = x_n + \lambda(P_C y_n - x_n) \end{cases}$$

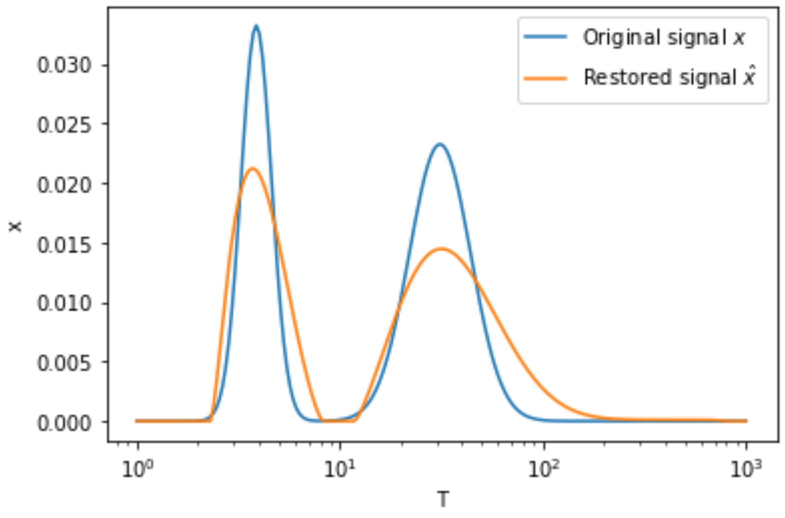
which converges weakly to a minimiser of f over C.

3. Implement projected gradient algorithm.

In [12]:
```
# Projection over C=[x_min;x_max]
def proj(u):
    xmin = min(x)
    xmax = max(x)
    return np.clip(u, xmin, xmax)

# Gradient of f
def grad1(u, beta):
    return np.transpose(K)@K@u - y@K + beta*np.transpose(D)
@D@u

# Solve minimisation problem via smooth penalisation + cons
traints
def projgrad(beta, show=False):
    itmax = 20000
    converged = False
    it = 0
    xn = np.zeros(N)
    tol = 1e-4
    nu = max(np.linalg.eig(np.transpose(K)@K - beta*np.tran
spose(D)@D)[0])
    gamma = 0.99 * (2/nu)
    lmb = 2 - gamma*nu/2

    while ((not converged) and (it < itmax)):
        it += 1
        yn = xn - gamma*grad1(xn, beta)
        xnn = xn + lmb*(proj(yn)-xn)
        if np.linalg.norm(xnn-xn,1) < tol*np.linalg.norm(x
n,1):
            converged = True
        xn = xnn

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

```
In [13]: # Use projected gradient with beta = 1
         xhat2 = projgrad(1, True)

         plt.plot(T, x, label='Original signal $x$');
         plt.plot(T, xhat2, label='Restored signal $\hat{x}$');
         plt.xscale('log');
         plt.xlabel('T');
         plt.ylabel('x');
         plt.legend();
         plt.show()
```

```
Converged: True
Number of iterations: 2501
```



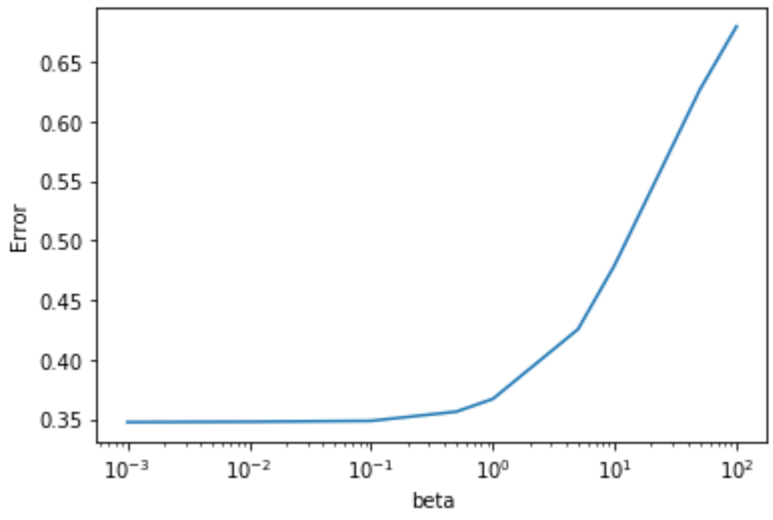
4. Evalutate the normalised quadratic error.

```
In [14]: error(x,xhat2)

Out[14]: 0.36719465352420083
```

5. Search for the best value for β.

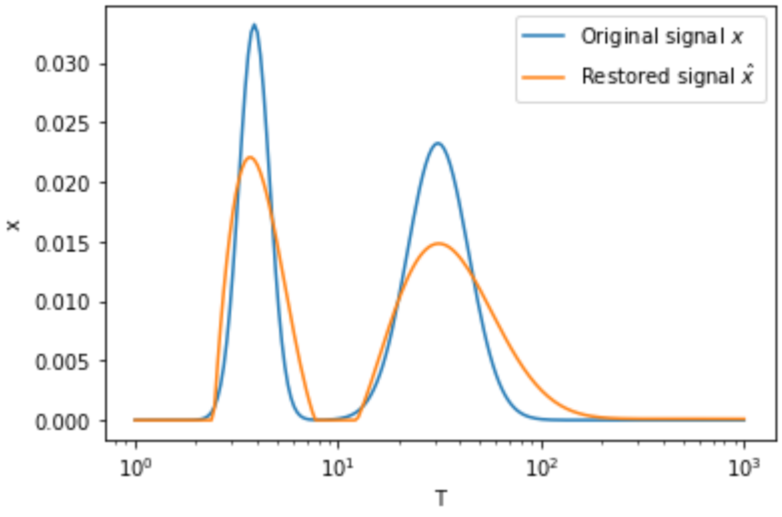
In [15]:
```
e_vector = []
b_vector = [1e-3, 1e-2, 1e-1, 0.5, 1, 5, 10, 50, 100]

# Test error for many values of beta
for beta in b_vector:
    e_vector.append(error(x,projgrad(beta)))

# Choose best value of beta
e_min = min(e_vector)
b_min = b_vector[np.argmin(e_vector)]

plt.plot(b_vector, e_vector)
plt.xscale('log');
plt.xlabel('beta');
plt.ylabel('Error');

print("Minimum error = {}, beta = {}".format(e_min,b_min))
```

```
Minimum error = 0.34770181289331376, beta = 0.001
```



In [16]:
```
# Use projected gradient with best beta
xhat1 = projgrad(b_min, True)

plt.plot(T, x, label='Original signal $x$');
plt.plot(T, xhat1, label='Restored signal $\hat{x}$');
plt.xscale('log');
plt.xlabel('T');
plt.ylabel('x');
plt.legend();
plt.show()
```

```
Converged: True
Number of iterations: 2647
```



3.3. Sparsity prior

Now, consider the penalisation function

$$g(x) = \|x\|_1.$$

The optimisation problem becomes

$$\hat{x} = \arg\min_{x \in \mathbb{R}^N} f(x) + \beta g(x) = \arg\min_{x \in \mathbb{R}^N} \frac{1}{2}\|Kx - y\|^2 + \beta\|x\|_1.$$

1. The set $\mathbb{R}^N$ being closed and the cost function $f(x) + \beta g(x)$ being in $\Gamma_0(\mathbb{R}^N)$ and coercive (as it is the sum of such functions), there exists a solution for the minimisation problem. As the 1-norm $\|\cdot\|_1$ is not strictly convex, we cannot guarantee uniqueness of the solution.

2. We can use the *forward-backward* algorithm, as

- $f(x) = \frac{1}{2}\|Kx - y\|^2 \in \Gamma_0(\mathrm{R}^N)$ is differentiable, with ν-Lipschitzian gradient :

  $$\|\nabla f(x_1) - \nabla f(x_2)\| = \|(K^TK)(x_1 - x_2)\| \leq \max \sigma_i(K^TK)\|x_1 - x_2\| \Rightarrow \nu = \max \sigma_i(K^TK) \in \mathrm{R}.$$

- $\beta g(x) = \beta\|x\|_1 \in \Gamma_0(\mathrm{R}^N)$.

By choosing $\gamma \in]0, 2/\nu[$, $\delta = 2 - \gamma\nu/2 \in]1, 2[$, $\lambda \in [0, \delta]$ and $x_0 \in \mathrm{R}^N$, we can apply

$$\begin{cases} y_n = x_n - \gamma\nabla f(x_n) \\ x_{n+1} = x_n + \lambda(\mathrm{prox}_{\gamma\beta g} y_n - x_n) \end{cases}$$

which converges weakly to a minimiser of $f + \beta g$.

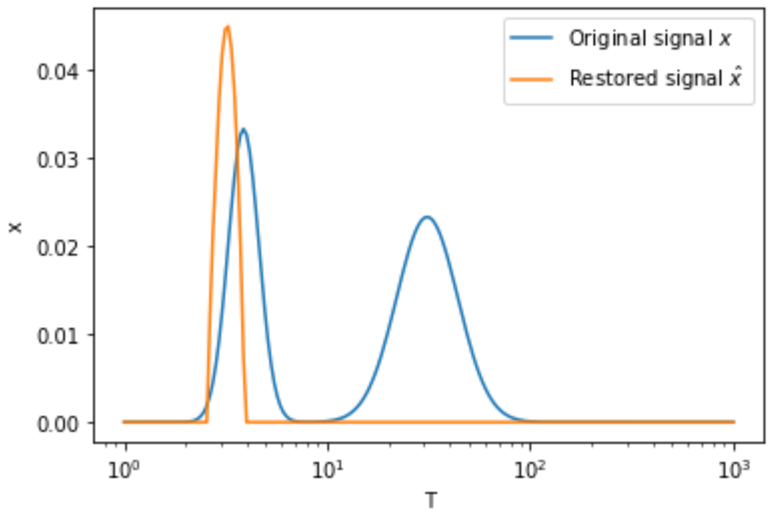
To compute $\mathrm{prox}_{\gamma\beta g}$, note that $\gamma\beta g(x) = \gamma\beta\|x\|_1 = \sum_{i=1}^n \gamma\beta|x^{(i)}|$. We know that $\mathrm{prox}_{\xi\mapsto\gamma\beta|\xi|}(x) = \mathrm{sign}(x)\max\{|x| - \gamma\beta, 0\}$ [1]. Then, we compute coordinate-wise $\mathrm{prox}_{\gamma\beta g}(x) = \left(\mathrm{prox}_{\xi\mapsto\gamma\beta|\xi|}(x^{(i)})\right)_{1\leq i\leq n}$

3. Implement forward-backward algorithm.

In [22]:
```python
# Prox operator of g
def prox(u, gamma, beta):
    out = np.zeros(len(u))
    for i in range(len(u)):
        out[i] = max(np.sign(u[i])*u[i]-gamma*beta,0)
    return out

# Gradient of f
def grad2(u):
    return np.transpose(K)@K@u - y@K

# Solve minimisation problem via sparsity prior
def forback(beta, show=False):
    itmax = 30000
    converged = False
    it = 0
    xn = np.ones(N)
    tol = 1e-4
    nu = np.real(max(np.linalg.eig(np.transpose(K)@K)[0]))
    gamma = 0.99 * (2/nu)
    lmb = 2 - gamma*nu/2

    while ((not converged) and (it < itmax)):
        it += 1
        yn = xn - gamma*grad2(xn)
        xnn = xn + lmb*(prox(yn, gamma, beta)-xn)
        if np.linalg.norm(xnn-xn,1) < tol*np.linalg.norm(x
n,1):
            converged = True
        xn = xnn

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

In [23]:
```
# Use forward-bacward with beta = 1
xhat3 = forback(1, True)

plt.plot(T, x, label='Original signal $x$');
plt.plot(T, xhat3, label='Restored signal $\hat{x}$');
plt.xscale('log');
plt.xlabel('T');
plt.ylabel('x');
plt.legend();
plt.show()
```

```
Converged: True
Number of iterations: 3806
```



4. Evalutate the normalised quadratic error.

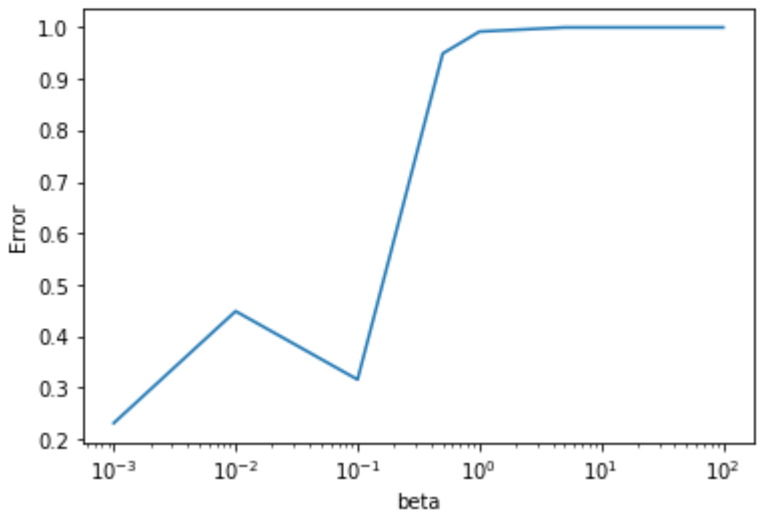
In [24]:
```
error(x,xhat3)
```

Out[24]:
```
0.99192415888317864
```

5. Search for the best value for β.

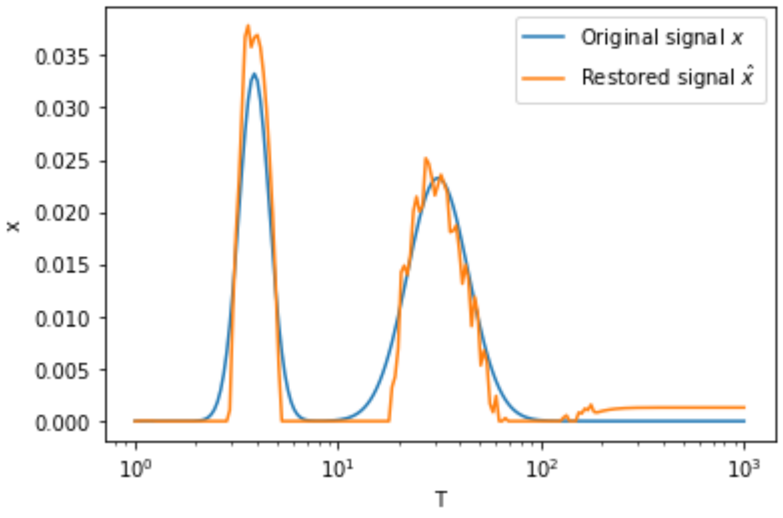
In [25]:
```
e_vector = []
b_vector = [1e-3, 1e-2, 1e-1, 0.5, 1, 5, 10, 50, 100]

# Test error for many values of beta
for beta in b_vector:
    e_vector.append(error(x,forback(beta)))

# Choose best value of beta
e_min = min(e_vector)
b_min = b_vector[np.argmin(e_vector)]

plt.plot(b_vector, e_vector)
plt.xscale('log');
plt.xlabel('beta');
plt.ylabel('Error');

print("Minimum error = {}, beta = {}".format(e_min,b_min))
```

```
Minimum error = 0.23093598756044564, beta = 0.001
```



In [26]:
```python
# Use forward-backward with best beta
xhat3 = forback(b_min, True)

plt.plot(T, x, label='Original signal $x$');
plt.plot(T, xhat3, label='Restored signal $\hat{x}$');
plt.xscale('log');
plt.xlabel('T');
plt.ylabel('x');
plt.legend();
plt.show()
```

```
Converged: False
Number of iterations: 30000
```



3.4. Conclusions

We have implemented three regularisation strategies to recover the original signal from a noisy version. All of them allowed us to recover in some degree the original signal. In the following we summarise the results, highlighting the differences between the methods.

- Smootheness prior is the simplest method. On the one hand, it is fast, as it is based on direct calculations and not on an iterative loop. On the other hand, it has the poorest performance, with normalised error 0.4388, for $\beta = 1$.
- Smoothness prior with constraints, solved with projected gradient achieved normalised error of 0.3477 for $\beta = 0.001$, requiring 2647 iterations.
- Sparsity prior, solved with forward-backward algorithm, has the best performance in terms of normalised error: 0.1774, for $\beta = 0.001$. We can visually see that the restored signal obtained with this algorithm is the one that approaches the most the original signal. The drawback is that the required number of iterations was over 30000.

References

[1] Emilie Chouzenoux and Jean-Christophe Pesquet. Large Scale and Distributed Optimization, Part IV: Proximity operator. Course notes, CentraleSupélec, 2019.

[2] Emilie Chouzenoux and Jean-Christophe Pesquet. Large Scale and Distributed Optimization, Part V: Fixed point algorithms. Course notes, CentraleSupélec, 2019.