APPM 5360, Spring 2023 - Written Homework 3 and 4

Eappen Nelluvelil

February 17, 2023

1. The matrix-free implementation of the blur function is given below:

Listing 1: Implementation of the blur function

```
# Circular convolution function
def circ_conv(x, h):
        N = np.size(x)

        h_hat = np.zeros((N, ))
        h_hat[0:h.size] = h
        h_hat = np.fft.fft(h_hat)

        val = np.fft.ifft(np.multiply(h_hat, np.fft.fft(x)))
        return np.real_if_close(val)

# Blur function
def blur(x):
        # Our fixed filter
        h = np.exp((-np.power(np.arange(-2, 3), 2))/2)
        return circ_conv(x, h)
```

The function `implicit2explicit`, given below, takes a generic linear function and returns its explicit matrix representation. The function to test the `implicit2explicit` is also given below.

Listing 2: Implementation of the `implicit2explicit` function and the function to test it

```
# Implicit to explicit function
def implicit2explicit(linear_func, N):
        # Pre-allocate explicit matrix for the linear function
        B = np.zeros((N, N))
        e_i = np.zeros((N, ))
        for i in np.arange(0, N):
                e_i[i] = 1
                B[:, i] = linear_func(e_i)
                e_i[i] = 0
        return B

# Function to test the implicit2explicit function
def test_implicit2explicit(linear_func, matrix, x):
        if np.allclose(linear_func(x), matrix @ x):
                return True
```

```
        else:
                False
```

2. We solve the model given in (1) per the homework description using `cvxpy` as follows:

Listing 3: Code that solves the model given in (1) via `cvxpy`

```
def main():
    # Create our input signal
    N = 100
    x = np.zeros((N,))
    non_zero_inds = np.array([9, 12, 49, 69])
    non_zero_vals = np.array([1, -1, 0.3, -0.2])
    x[non_zero_inds] = non_zero_vals
    # print(x)

    # Create stochastic noise vector
    mu = 0
    sigma = 0.02
    rng = default_rng()
    z = rng.normal(mu, sigma, size=N)
    # print(z)

    # Create explicit operator representation of the blur function
    B = implicit2explicit(blur, N)

    # Test that the implicit2explicit operator works as intended
    print("Does the implicit2explicit operator work as intended? ", \
          test_implicit2explicit(blur, B, x))

    # Create blurred and noisy signal
    y = B @ x + z

    # Solve the given optimization problem in the homework
    x_hat_2 = cvx.Variable((N, ))
    obj    = cvx.Minimize(cvx.norm(x_hat_2, 1))

    eps = sigma*np.sqrt(N)
    constraints = [cvx.norm(B @ x_hat_2 - y, 2)**2 <= eps**2]

    prob = cvx.Problem(obj, constraints)
    prob.solve(verbose=False)

    print("Problem 2 status: ", prob.status)
    print("Problem 2 optimal value: ", prob.value)

    # Make plot containing original signal,
    # the blurred and noisy signal,
    # and the recovered signal
    plt.figure()

    xvals = np.arange(1, N+1)
```

```
    plt.plot(xvals, x, "-o", label="Original signal")
    plt.plot(xvals, y, "-*", label="Blurred and noisy signal")
    plt.plot(xvals, x_hat_2.value, "-x", label="Deblurred signal (prob 2)")
```

The resulting plot is given below:

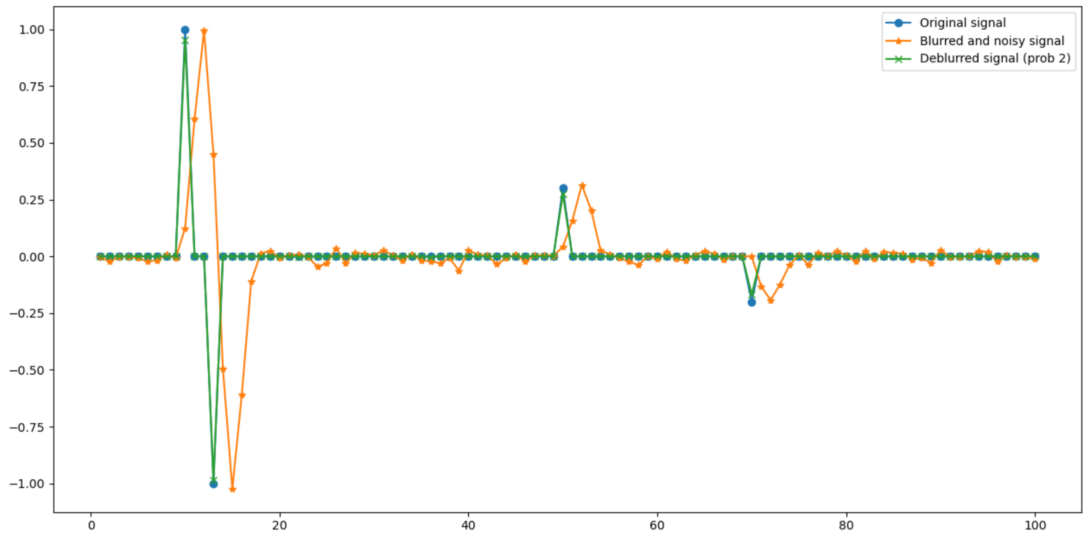

Figure 1: Plot of original signal x, the blurred and noisy signal y, and the estimate $\hat{x}$

3. We solve the equivalent problem using the dual variable corresponding to the single constraint given in (1) using the following code:

Listing 4: Implementation that solves model (2) using the dual variable obtained from solving (1)

```
    # Problem 3: Solve the same problem as above using a first
# order method, and use the dual variable of the solved
# problem from earlier
lambda_val = constraints[0].dual_value

x_hat_3 = cvx.Variable((N, ))
obj_3   = cvx.Minimize(cvx.norm(x_hat_3, 1) +
                       lambda_val*cvx.norm(B @ x_hat_3 - y, 2)**2)
prob3   = cvx.Problem(obj_3)
prob3.solve(verbose=False)

print("Problem 3 status: ", prob3.status)
print("Problem 3 value:  ", prob3.value)

# Verify that the Euclidean norm of the difference of the two
# solutions are close
print("2-norm of difference between recovered signals for problems 2 and 3: ",
```

```
        np.linalg.norm(x_hat_2.value - x_hat_3.value, 2))

plt.plot(xvals, x_hat_3.value, "-x", label="Deblurred signal (prob 3)")

plt.legend()
plt.show()
```

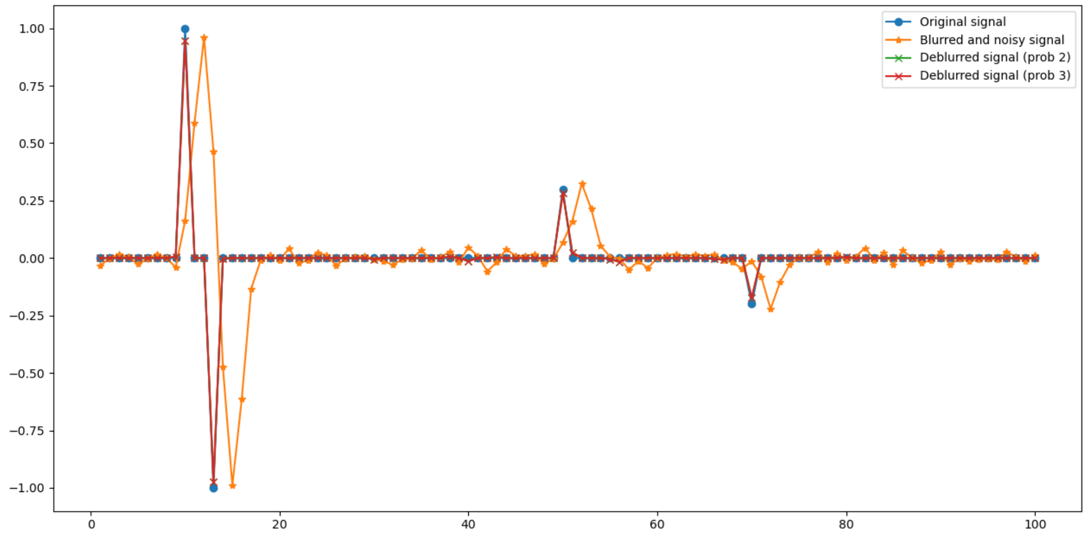

Figure 2: Plot of estimate from problem 3 $\hat{x}$

4. The below code computes $\mathcal{B}$ and $\mathcal{B}^*$ efficiently using `numpy`'s FFT and IFFT implementations, as well as verify that the functions compute $\mathcal{B}^*$:

Listing 5: Code that implements the blur filter and its adjoint efficiently, as well as test to make sure the implementations are adjoints of one another

```
# Function to compute the h_hat vector
# that is used in computing the action of
# H and the adjoint of H in the blur function
def compute_h_hat(N):
    # Our fixed filter
    h = np.exp((-np.power(np.arange(-2, 3), 2))/2)

    # Zero-pad the above filter
    h_hat = np.zeros((N, ))
    h_hat[0:h.size] = h
    h_hat = np.fft.fft(h_hat)

    return h_hat
```

```python
    # Function to compute the action of H on a vector
    def H(z):
    N = np.size(z)
    val = np.multiply(compute_h_hat(N), z)
    return val

    # Function to compute action of the adjoint of H
    # on a vector
    def H_adj(z):
    N = np.size(z)
    val = np.multiply(np.conjugate(compute_h_hat(N)), z)
    return val

    # Function to compute the blur function using the
    # FFT and IFFT
    def compute_B(x):
    val = np.fft.ifft(H(np.fft.fft(x)))
    return np.real_if_close(val)

    # Function to compute the adjoint of the blur function
    # using the FFT and IFFT
    def compute_B_adj(x):
    val = np.fft.ifft(H_adj(np.fft.fft(x)))
    return np.real_if_close(val)

    # Function to test if compute_B and compute_B_adj
    # performs the action of B and the adjoint of B
    # on vectors
    def test_adjoint(N):
    passed_test = True
    max_iters = 100
    for i in np.arange(max_iters):
        x = np.random.rand(N)
        y = np.random.rand(N)

        first_ip = np.vdot(compute_B(x), y)
        second_ip = np.vdot(x, compute_B_adj(y))

        if not np.allclose(first_ip, second_ip):
            passed_test = False
            break

    return passed_test
```

5. We solve model (3) using the `lassoSolver` given in the `firstOrderMethods` module. The code is given below:

Listing 6: Code that solves model (3) using the implicit blur and implicit adjoint blur implementations from problem (4)

```python
# Problem 4: Run test_adjoint function
    print("Was the adjoint of the fast forward blur constructed correctly?", test_adjo
```

```
# Problem 5: Solve the reformulated first-order problem
tau = 1/(2*lambda_val)
x_hat_5, data = FOM.lassoSolver(compute_B, y, tau, At=compute_B_adj, x=None)

# Verify that the Euclidean norm of the difference of the two
# solutions are close
print("2-norm of difference between recovered signals for problems 2 and 5: ",
      np.linalg.norm(x_hat_2.value - x_hat_5, 2))

print("2-norm of difference between recovered signals for problems 3 and 5: ",
      np.linalg.norm(x_hat_3.value - x_hat_5, 2))

plt.plot(xvals, x_hat_5, "-+", label="Deblurred signal (prob 5)")

plt.legend()
plt.show()
```

The below plot gives the estimate of the signal:

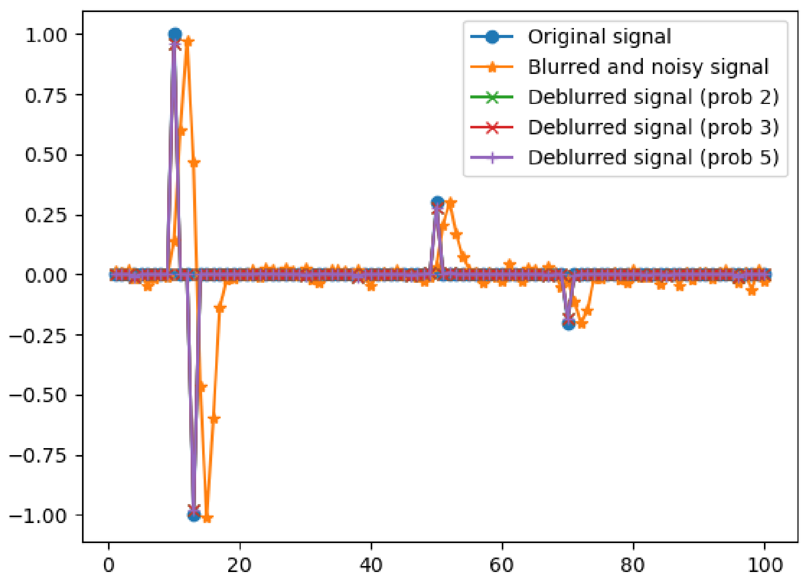

Figure 3: Plot of estimate $\hat{x}$ from problem 5

6. Below is the testing output

```
Does the implicit2explicit operator work as intended?  True
Problem 2 status:  optimal
Problem 2 optimal value:  2.466364999583184
Problem 3 status:  optimal
Problem 3 value:   3.0164780215169644
```

```
Norm of difference for recovered signals for problems 2 and 3:  5.36854056514202e-07
Was the adjoint of the fast forward blur constructed correctly? True
Iter.  Objective Stepsize
-----  --------- --------
    0  2.00e+00  1.43e-01
   59  1.10e-01  1.43e-01
Iter 59 Quitting due to stagnating objective value
2-norm of difference between recovered signals for problems 2 and 5:  0.0018527301565
2-norm of difference between recovered signals for problems 3 and 5:  0.0018527513873
```

Listing 7: Overall code listing

```python
import numpy as np
from numpy.random import default_rng
import cvxpy as cvx
import scipy
from scipy import ndimage
import matplotlib.pyplot as plt
import firstOrderMethods as FOM

# Circular convolution function
def circ_conv(x, h):
        N = np.size(x)

        h_hat = np.zeros((N, ))
        h_hat[0:h.size] = h
        h_hat = np.fft.fft(h_hat)

        val = np.fft.ifft(np.multiply(h_hat, np.fft.fft(x)))
        return np.real_if_close(val)

# Blur function
def blur(x):
        # Our fixed filter
        h = np.exp((-np.power(np.arange(-2, 3), 2))/2)
        return circ_conv(x, h)

# Implicit to explicit function
def implicit2explicit(linear_func, N):
        # Pre-allocate explicit matrix for the linear function
        B = np.zeros((N, N))
        e_i = np.zeros((N, ))
        for i in np.arange(0, N):
                e_i[i] = 1
                B[:, i] = linear_func(e_i)
                e_i[i] = 0
        return B

# Function to compute the h_hat vector
# that is used in computing the action of
# H and the adjoint of H in the blur function
```

```python
def compute_h_hat(N):
        # Our fixed filter
        h = np.exp((-np.power(np.arange(-2, 3), 2))/2)

        # Zero-pad the above filter
        h_hat = np.zeros((N, ))
        h_hat[0:h.size] = h
        h_hat = np.fft.fft(h_hat)

        return h_hat

# Function to compute the action of H on a vector
def H(z):
        N = np.size(z)
        val = np.multiply(compute_h_hat(N), z)
        return val

# Function to compute action of the adjoint of H
# on a vector
def H_adj(z):
        N = np.size(z)
        val = np.multiply(np.conjugate(compute_h_hat(N)), z)
        return val

# Function to compute the blur function using the
# FFT and IFFT
def compute_B(x):
        val = np.fft.ifft(H(np.fft.fft(x)))
        return np.real_if_close(val)

# Function to compute the adjoint of the blur function
# using the FFT and IFFT
def compute_B_adj(x):
        val = np.fft.ifft(H_adj(np.fft.fft(x)))
        return np.real_if_close(val)

# Function to test if compute_B and compute_B_adj
# performs the action of B and the adjoint of B
# on vectors
def test_adjoint(N):
        passed_test = True
        max_iters = 100
        for i in np.arange(max_iters):
                x = np.random.rand(N)
                y = np.random.rand(N)

                first_ip = np.vdot(compute_B(x), y)
                second_ip = np.vdot(x, compute_B_adj(y))

                if not np.allclose(first_ip, second_ip):
                        passed_test = False
```

```
                        break

        return passed_test

def main():
        # Create our input signal
        N = 100
        x = np.zeros((N,))
        non_zero_inds = np.array([9, 12, 49, 69])
        non_zero_vals = np.array([1, -1, 0.3, -0.2])
        x[non_zero_inds] = non_zero_vals
        # print(x)

        # Create stochastic noise vector
        mu = 0
        sigma = 0.02
        rng = default_rng()
        z = rng.normal(mu, sigma, size=N)
        # print(z)

        # Create explicit operator representation of the blur function
        B = implicit2explicit(blur, N)

        # Create blurred and noisy signal
        y = B @ x + z

        # Solve the given optimization problem in the homework
        x_hat_2 = cvx.Variable((N, ))
        obj   = cvx.Minimize(cvx.norm(x_hat_2, 1))

        eps = sigma*np.sqrt(N)
        constraints = [cvx.norm(B @ x_hat_2 - y, 2)**2 <= eps**2]

        prob = cvx.Problem(obj, constraints)
        prob.solve(verbose=False)

        print("Problem 2 status: ", prob.status)
        print("Problem 2 optimal value: ", prob.value)

        # Make plot containing original signal,
        # the blurred and noisy signal,
        # and the recovered signal
        plt.figure()

        xvals = np.arange(1, N+1)
        plt.plot(xvals, x, "-o", label="Original signal")
        plt.plot(xvals, y, "-*", label="Blurred and noisy signal")
        plt.plot(xvals, x_hat_2.value, "-x", label="Deblurred signal (prob 2)")

        # Problem 3: Solve the same problem as above using a first
        # order method, and use the dual variable of the solved
```

```
        # problem from earlier
        lambda_val = constraints[0].dual_value

        x_hat_3 = cvx.Variable((N, ))
        obj_3   = cvx.Minimize(cvx.norm(x_hat_3, 1) +
                                lambda_val*cvx.norm(B @ x_hat_3 - y, 2)**2)
        prob3   = cvx.Problem(obj_3)
        prob3.solve(verbose=False)

        print("Problem 3 status: ", prob3.status)
        print("Problem 3 value:  ", prob3.value)

        # Verify that the Euclidean norm of the difference of the two
        # solutions are close
        print("2-norm of difference between recovered signals for problems 2 and 3: ",
              np.linalg.norm(x_hat_2.value - x_hat_3.value, 2))

        plt.plot(xvals, x_hat_3.value, "-x", label="Deblurred signal (prob 3)")


        # Problem 4: Run test_adjoint function
        print("Was the adjoint of the fast forward blur constructed correctly?", test_adjo

        # Problem 5: Solve the reformulated first-order problem
        tau = 1/(2*lambda_val)
        x_hat_5, data = FOM.lassoSolver(compute_B, y, tau, At=compute_B_adj, x=None)

        # Verify that the Euclidean norm of the difference of the two
        # solutions are close
        print("2-norm of difference between recovered signals for problems 2 and 5: ",
              np.linalg.norm(x_hat_2.value - x_hat_5, 2))

        plt.plot(xvals, x_hat_5, "-+", label="Deblurred signal (prob 5)")

        plt.legend()
        plt.show()

main()
```