hw_10_notebook

April 7, 2023

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.sparse, scipy.fftpack
import cvxpy as cvx
import pickle
```

[ ]:
```
def project_l1(x, tau=1.):
    """
    project_l1(x, tau) -> y
      projects x onto the scaled l1 ball, ||x||_1 <= tau
      If tau is not provided, the default is tau = 1.

    Stephen Becker and Emmanuel Candes, 2009/2010.
    Crucial bug fix: 3/17/2017, SRB
    """
    absx = np.abs(x)
    s = np.sort(absx, axis=None)[::-1] # sort in descending order
    cs = np.cumsum(s)

    if cs[-1] <= tau:
        # x is already feasible, so no thresholding needed
        return x

    # Check some "discrete" levels of shrinkage, e.g. [s(2:end),0]
    # This lets us discover which indices will be nonzero
    n = x.size
    i_tau = np.where(cs -
        np.arange(1,n+1)*np.concatenate((s[1:],0), axis=None) >= tau)[0][0]

    # Now that we know which indices are involved, it's a very simple problem
    thresh = (cs[i_tau]-tau) / (i_tau+1)

    # Shrink x by the amount "thresh"
    return np.sign(x)*np.maximum(absx - thresh, 0)

def forwardShortTimeDCT(y, win=None):
    """
    forwardShortTimeDCT(y, win=None) -> coeff (, win)
```

```
    Applies the Modified DCT to the signal y
    This is a linear function.
    Assumes y is a vector of length N
    This code then uses a lapped (50% overlapping)
    DCT on segments of y of length blockSize.

    Note: this function zero-pads y to be an even multiple of blockSize.

    An example window that we recommend, so that the transpose of this
    function is its pseudo-inverse is

    win = np.sin(np.pi*(np.arange(1,blockSize+1)+0.5)/blockSize)
    (a typical value of blockSize = 1024)

    On input, if win is not specified, we return (coeff, win) so the
    caller has access to the window we used.

    This satisfies the Princen-Bradley conditions, meaning that we can
    guarantee  win**2+np.roll(win, int(blockSize/2))**2 == 1.

    Stephen Becker, 3/18/2017
    See also adjointShortTimeDCT
    """
    # Make a window if not provided by the user
    if win is None:
        blockSize = 1024
        win = np.sin(np.pi*(np.arange(1,blockSize+1)+0.5)/blockSize)
        return_win = True
    else:
        blockSize = win.size
        return_win = False

    # Zero-pad y so it is a multiple of blockSize
    N = y.size
    nBlocks = int(np.ceil(float(N)/blockSize))
    y = np.concatenate((y, np.zeros(nBlocks*blockSize-N)))

    # Apply DCT to aligned blocks
    Win = scipy.sparse.spdiags(win, [0], blockSize, blockSize)
    Y = np.reshape(y, (blockSize, nBlocks), order='f')
    C = scipy.fftpack.dct(Win*Y, axis=0, norm='ortho')

    # Apply DCT to 50% shifted blockSize
    Y = np.reshape(np.roll(y, int(-blockSize/2)), (blockSize, nBlocks),
        order='f')
    C2 = scipy.fftpack.dct(Win*Y, axis=0, norm='ortho')
```

```
    coeff = np.concatenate((C.ravel(order='f'), C2.ravel(order='f')))

    if return_win: return coeff, win
    else: return coeff

def adjointShortTimeDCT(coeff, win, Ntrue=None):
    """
    adjointShortTimeDCT(coeff, win, Ntrue=None) -> y

    Applies the adjoint/transpose Modified DCT to the coefficients coeff.
    This is also the pseudo-inverse of the forward MDCT.

    If Ntrue=N_original, where N_original is the original length of
    the signal y (i.e., before zero-padding in forwardShortTimeDCT),
    we truncate the padded zeros and return the original y.

    See forwardShortTimeDCT for an example of the window win.

    Stephen Becker, 3/18/2017
    See also forwardShortTimeDCT
    """

    if coeff.size % 2:
        raise ValueError("""coeff should have an even number of elements.
            Did you compute coeff with forwardShortTimeDCT?""")
    N = int(coeff.size/2)

    blockSize = win.size
    nBlocks = int(np.ceil(float(N)/blockSize))

    if Ntrue is not None and Ntrue > N:
        raise ValueError("""The specified value of Ntrue ({}) is too big
            for the number of coefficients in coeff ({})""".format(
            Ntrue, N))

    Win = scipy.sparse.spdiags(win, [0], blockSize, blockSize)

    C = np.reshape(coeff[0:N], (blockSize, nBlocks), order='f')
    Y = Win*scipy.fftpack.idct(C, axis=0, norm='ortho')
    y = Y.ravel(order='f')

    C2 = np.reshape(coeff[N:], (blockSize, nBlocks), order='f')
    Y2 = Win*scipy.fftpack.idct(C2, axis=0, norm='ortho')
    y2 = np.roll(Y2.ravel(order='f'), int(blockSize/2))
    y += y2
```

```
    if Ntrue:
        y = y[0:Ntrue]

    return y

def my_upsample(y, sampleSet, n):
    """
    my_upsample(y, sampleSet, n) -> x
    Returns x of length n such that x[sampleSet] = y
    """
    if y.ndim == 1:
        x = np.zeros(n)
        x[sampleSet] = y
    else:
        x = np.zeros((n, y.shape[1]))
        x[sampleSet,:] = y
    return x
```

[ ]:
```
# Proximal gradient descent function
def prox_GD(f, grad_f, prox_g, x0, use_g=False, max_iters=100, stepsize=1,␣
↪tol=1e-7):
        # If g is the zero function, then use Nesterov's accelerated gradient␣
↪descent
        if not use_g:
                _, w_NAG = NAG(f, grad_f, x0, max_iters=max_iters,
                                                stepsize=stepsize, tol=tol)
                return w_NAG
        # If g is not the zero function, then use proximal gradient descent
        else:
                x_prev = x0
                y_prev = x0

                x_curr = np.zeros(x0.shape)
                y_curr = np.zeros(x0.shape)

                num_iters = 0
                while (num_iters < max_iters):
                        # x_curr = prox_g(l, stepsize, y_prev -␣
↪stepsize*grad_f(y_prev))
                        x_curr = prox_g(y_prev - stepsize*grad_f(y_prev))
                        y_curr = x_curr + ((num_iters)/(num_iters + 3))*(x_curr␣
↪- x_prev)

                        num_iters += 1

                        if (np.linalg.norm(x_curr - x_prev, 2) < tol):
                                break
```

```
                    x_prev = x_curr
                    y_prev = y_curr

            return x_curr
```

[ ]:
```
# Problem 1, part a
# Create random A and b
m = 10
n = 20

A = np.random.random((m, n))
b = np.random.random(m)

# Solve the basis pursuit problem with the above random A and b
x_BP = cvx.Variable((n, ))
obj = cvx.Minimize(cvx.norm(x_BP, 1))
constraints = [A @ x_BP == b]
prob = cvx.Problem(obj, constraints)
prob.solve(solver=cvx.ECOS, abstol=1e-14, reltol=1e-14,
           feastol=1e-13, max_iters=500, verbose=True)
```

```
===============================================================================
                                     CVXPY
                                     v1.3.0
===============================================================================
(CVXPY) Apr 06 09:40:17 PM: Your problem has 20 variables, 1 constraints, and 0
parameters.
(CVXPY) Apr 06 09:40:17 PM: It is compliant with the following grammars: DCP,
DQCP
(CVXPY) Apr 06 09:40:17 PM: (If you need to solve this problem multiple times,
but with different data, consider using parameters.)
(CVXPY) Apr 06 09:40:17 PM: CVXPY will first compile your problem; then, it will
invoke a numerical solver to obtain a solution.
-------------------------------------------------------------------------------
                                  Compilation
-------------------------------------------------------------------------------
(CVXPY) Apr 06 09:40:17 PM: Compiling problem (target solver=ECOS).
(CVXPY) Apr 06 09:40:17 PM: Reduction chain: Dcp2Cone -> CvxAttr2Constr ->
ConeMatrixStuffing -> ECOS
(CVXPY) Apr 06 09:40:17 PM: Applying reduction Dcp2Cone
(CVXPY) Apr 06 09:40:17 PM: Applying reduction CvxAttr2Constr
(CVXPY) Apr 06 09:40:17 PM: Applying reduction ConeMatrixStuffing
(CVXPY) Apr 06 09:40:17 PM: Applying reduction ECOS
(CVXPY) Apr 06 09:40:17 PM: Finished problem compilation (took 1.696e-02
seconds).
-------------------------------------------------------------------------------
                                Numerical solver
```

```
-----------------------------------------------------------------------------
(CVXPY) Apr 06 09:40:17 PM: Invoking solver ECOS  to obtain a solution.
-----------------------------------------------------------------------------
                                    Summary
-----------------------------------------------------------------------------
(CVXPY) Apr 06 09:40:18 PM: Problem status: optimal_inaccurate
(CVXPY) Apr 06 09:40:18 PM: Optimal value: 3.596e+00
(CVXPY) Apr 06 09:40:18 PM: Compilation took 1.696e-02 seconds
(CVXPY) Apr 06 09:40:18 PM: Solver (including time spent in interface) took
1.506e-01 seconds
```

[ ]:
```
3.5962206470641997
```

[ ]:
```
# Problem 1, part b
tau = np.linalg.norm(x_BP.value, 1)

# Solve LS_tau, version 2, using a proximal gradient descent algorithm
f = lambda x: (1/2)*np.linalg.norm(A @ x - b)**2
grad_f = lambda x: A.T @ (A @ x - b)
stepsize = 1/(np.linalg.norm(A, ord=2)**2)

# The proximity function for the indicator function is the projection
# operator
indicator_prox = lambda x: project_l1(x, tau)

x0 = np.random.random(x_BP.shape)
x_tau = prox_GD(f, grad_f, indicator_prox, x0, use_g=True, max_iters=15000,
                stepsize=stepsize, tol=1e-10)

print("Infinity-norm of difference between x_tau and X_BP: ",
       np.linalg.norm(x_tau - x_BP.value, ord=np.inf))
```

```
Infinity-norm of difference between x_tau and X_BP:  2.226177914232963e-08
```

1 Problem 1, part (b)

Upon solving the problem (LS_τ), version 2, via proximal gradient descent, we see that the recovered solution x_τ agrees well with the original solution x_{BP}, which was obtained by solving the problem via CVXPY, i.e., $||x_\tau - x_{\mathrm{BP}}||_\infty$ is on the order of 10^{-8}.

[ ]:
```
# Newton solver for tau
def newton_solver(tau_0, A, A_T, b, f, grad_f, x_0, max_iters_newton=100,
                  tol_newton=1e-7, max_iters_prox_GD=100,
                  stepsize_prox_GD=1, tol_prox_GD=1e-7):
    iter = 0

    tau_prev = tau_0
    tau_curr = 0
```

```
    x_prev   = np.array(x_0)
    x_curr   = np.zeros(x_0.shape)

    # Uncomment if A is explicitly passed on
    # sigma = lambda x: np.linalg.norm(A@x - b, ord=2)
    sigma = lambda x: np.linalg.norm(A(x) - b, ord=2)

    while (iter < max_iters_newton):
        iter += 1

        # Solve the primal problem for the given tau to obtain x_tau
        indicator_prox = lambda x: project_l1(x, tau_prev)
        x_curr = prox_GD(f, grad_f, indicator_prox, x_prev,
                         use_g=True, max_iters=max_iters_prox_GD,
                         stepsize=stepsize_prox_GD, tol=tol_prox_GD)


        # Solve for z_tau
        z_tau = (-1)*A(x_curr) + b

        # Solve for nu_tau
        nu_tau = -z_tau/np.linalg.norm(z_tau, ord=2)

        lambda_tau = np.linalg.norm(A_T(nu_tau), ord=np.inf)

        tau_curr = tau_prev - sigma(x_curr)/((-1)*lambda_tau)

        x_prev = np.array(x_curr)

        if (np.fabs(tau_curr - tau_prev) < tol_newton):
            break

        tau_prev = tau_curr

    return (x_curr, tau_curr)
```

[ ]:
```
# Problem 2
tau_0 = (1/4)
max_iters_newton = 5
tol_newton = 1e-10
max_iters_prox_GD = 15000
tol_prox_GD = 1e-10

A_op   = lambda x: A @ x
A_T_op = lambda x: A.T @ x
```

```
(approx_x_tau, approx_tau) = newton_solver(tau_0, A_op, A_T_op, b, f, grad_f,
                            x0, max_iters_newton=max_iters_newton,
                            tol_newton=tol_newton,
                            max_iters_prox_GD=max_iters_prox_GD,
                            stepsize_prox_GD=stepsize,
                            tol_prox_GD=tol_prox_GD)
```

[ ]:
```
print("Infinity-norm of difference between tau and approximate tau: ",
      np.fabs(tau - approx_tau))

print("Infinity-norm of difference between x_BP and approximate x_tau: ",
      np.linalg.norm(x_BP.value - approx_x_tau, ord=np.inf))
```

```
Infinity-norm of difference between tau and approximate tau:
4.502422434882192e-09
Infinity-norm of difference between x_BP and approximate x_tau:
2.7489109094558373e-07
```

2 Problem 2

From the numerical results, we see that the Newton solver works as intended, and we have that that the ∞-norm of the difference between the τ and τ_{approx} is on the order of 10^{-9}. Similarly, the ∞-norm of the difference between x_{BP} and $x_{\tau,2}$ is on the order of 10^{-8}, as was the case with the previous problem.

Furthermore, due to the small size of the problem, the Newton solver took only a few iterations to converge to good approximations to τ and x_{BP}, e.g., between 5 and 10 iterations.

[ ]:
```
# Problem 3
(y, Fs) = pickle.load(open("handel.pkl", "rb"))
Fs = Fs[0][0]
Fs = float(Fs)
y = y.ravel()

blockSize = 1024
win = np.sin(np.pi*(np.arange(1,blockSize+1)+0.5)/blockSize)

# Zero-pad y so that its length is an even multiple of the block size
N_original = blockSize * int(np.ceil(y.size/blockSize))
y_padded = np.zeros((N_original, ))
y_padded[0:y.size] = y
y = np.array(y_padded)

# Randomly sample a fourth of the entries in y
random_indices = np.random.choice(np.arange(y.size), int(np.floor(y.size/4)),
                                  replace=False)

# Create the operator psi that samples the input vector y
```

```
# at the above random indices
psi = lambda y: y[random_indices]

b_handel = psi(y)

# Make the operator A, which is psi*D^(T)
A_handel   = lambda x: psi(adjointShortTimeDCT(x, win, Ntrue=N_original))
A_T_handel = lambda y: forwardShortTimeDCT(my_upsample(y, random_indices,␣
 ↪N_original),
                                           win=win)

tau_0_handel = (1/4)
x_0_handel = np.random.random((y.shape[0]*2, ))

f_handel = lambda x: (1/2)*np.linalg.norm(A_handel(x) - b_handel, ord=2)**2
grad_f_handel = lambda x: A_T_handel(A_handel(x) - b_handel)

# For the stepsize to pass into proximal gradient descent, the operator␣
 ↪A_handel is unitary,
# so the Lipschitz constant of f is 1
(approx_t_tau_handel, approx_tau_handel) = newton_solver(tau_0_handel,␣
 ↪A_handel, A_T_handel,
                                                          b_handel, f_handel,␣
 ↪grad_f_handel,
                          x_0_handel, max_iters_newton=10,
                          tol_newton=tol_newton,
                          max_iters_prox_GD=300,
                          stepsize_prox_GD=1,
                          tol_prox_GD=tol_prox_GD)
```

[ ]:
```
# Check if we formed the adjoint of A_handel correctly
correct_adjoint = True
for i in range(100):
    x_random = np.random.random((y.shape[0]*2, ))
    x_random = x_random/np.linalg.norm(x_random)
    y_random = np.random.random((random_indices.shape[0], ))
    y_random = y_random/np.linalg.norm(y_random)

    inner_prod_1 = np.vdot(A_handel(x_random), y_random)
    inner_prod_2 = np.vdot(x_random, A_T_handel(y_random))

    if (np.fabs(inner_prod_1 - inner_prod_2) > 1e-10):
        correct_adjoint = False
        break

if not correct_adjoint:
    print("A_handel's adjoint was not correctly formed")
```

```
else:
    print("A_handel's adjoint was correctly formed")
```

```
A_handel's adjoint was correctly formed
```

[ ]:
```
# Recover the signal
y_recovered = adjointShortTimeDCT(approx_t_tau_handel, win, Ntrue=N_original)

import numpy as np
from numpy.linalg import norm
import scipy, scipy.io
import pickle
import scipy.signal as sig
import matplotlib.pyplot as plt
from IPython.display import Audio

# Audio(y, rate=Fs)
Audio(y_recovered, rate=Fs)
```

[ ]:
```
<IPython.lib.display.Audio object>
```

3 Problem 3

We check first if we formed the adjoint of the operator A_{handel} correctly, and from the output of the above kernels, we see that we did. We then proceed to invoke the Newton solver that we created earlier to solve the compressed sensing problem, using only a few iterations of Newton's method ($\approx 5 - 10$ iterations) and far fewer iterations of proximal gradient descent for each iteration of Newton's method (≈ 300 iterations).

Using the last iterate from the Newton solver, we obtain the recovered signal $\hat{y} = D^T x$ and use the provided helper code to listen to the recovered signal. Remarkably, the recovered signal sounds like Handel's hallelujah chorus, but grainer. The high-fequency components are recovered fairly well, despite us using only a random fourth of the original signal's entries.