hw_8_notebook_latex

March 21, 2023

[ ]:
```
import numpy as np
import scipy
import scipy.io as sio
import matplotlib.pyplot as plt
import pickle
import cvxpy as cvx
```

```
(CVXPY) Mar 17 03:08:14 PM: Encountered unexpected exception importing solver
OSQP:
ImportError('DLL load failed while importing qdldl: The specified module could
not be found.')
```

[ ]:
```
def gradientDescentLS(f, grad_f, x0, max_iters=100, stepsize=1, tol=1e-7):
        num_iters  = 0
        func_evals = []
        errors     = []
        xn         = np.zeros(x0.size)

        # Typical line search parameters
        c = 1e-4
        rho = 0.9
        t = stepsize

        while (num_iters < max_iters):
                num_iters += 1

                # Perform a line search to refine the stepsize
                # Our descent direction will be the negative gradient at the
 ↪current iterate
                armijo_cond_iters = 0
                pn = -grad_f(x0)
                while (f(x0 + t*pn) > f(x0) - c*t*np.vdot(grad_f(x0), pn)):
                        t *= rho
                        armijo_cond_iters += 1

                xn = x0 - t*grad_f(x0)
                func_evals.append(f(xn))
```

```python
            if (np.linalg.norm(xn - x0, 2) < tol):
                break

            x0 = xn

            # Increase the stepsize by a factor of two
            # if we decrease the stepsize only once
            if armijo_cond_iters == 1:
                t *= 2

        func_evals = np.array(func_evals)
        return func_evals, xn

def NAG(f, grad_f, x0, max_iters=100, stepsize=1, tol=1e-7):
        func_evals = []
        # lambda_prev = 0
        x_prev = x0
        y_prev = x0
        y_curr = np.zeros(x0.shape)
        x_curr = np.zeros(x0.shape)

        num_iters = 0
        while (num_iters < max_iters):
                x_curr = y_curr - stepsize*grad_f(y_curr)
                y_curr = x_curr + (num_iters/(num_iters + 3))*(x_curr - x_prev)
                num_iters += 1

                func_evals.append(f(x_curr))

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

                x_prev = x_curr
                y_prev = y_curr

        func_evals = np.array(func_evals)
        return func_evals, x_curr

def l1_norm_prox(l, t, y):
        return np.sign(y)*np.maximum(np.abs(y) - t*l, np.zeros(y.shape))

def prox_GD(f, grad_f, l, 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))
                        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
# Load the spam data
spamData = sio.loadmat("spamData")

# Pre-process and training and testing data
Xtrain = np.log(spamData["Xtrain"] + 0.1)
Xtest  = np.log(spamData["Xtest"] + 0.1)

# Load the training and testing labels, and change any 0's to 1's
ytrain = np.array(np.reshape(spamData["ytrain"], newshape=(spamData["ytrain"].
↪size, )), dtype=np.int32)
ytrain[ytrain == 0] = -1

ytest  = np.array(np.reshape(spamData["ytest"],  newshape=(spamData["ytest"].
↪size,  )), dtype=np.int32)
ytest[ytest == 0] = -1

Xtrain_norm = np.linalg.norm(Xtrain, 2)
stepsize = 4/(Xtrain_norm**2)
```

```python
# Define the logistic loss function and its gradient
def lr(w):
    return np.sum(np.log(1 + np.exp(-ytrain * np.matmul(Xtrain, w))))

def grad_lr(w):
    mu = 1/(1 + np.exp(-ytrain * np.matmul(Xtrain, w)))
    return (np.matmul(-Xtrain.T, ytrain*(1 - mu)))

sigmoid = lambda a: np.exp(a)/(1 + np.exp(a))
```

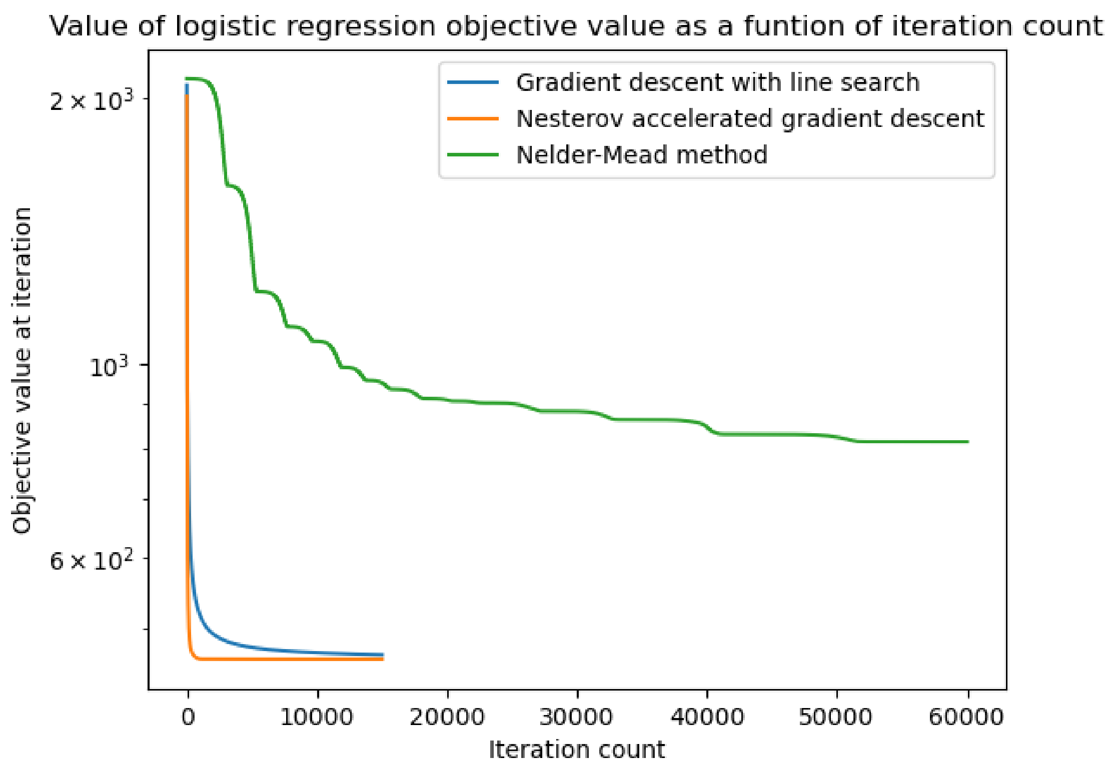
[ ]:
```python
w0 = np.random.rand(Xtrain.shape[1])/1000

# Solve the logistic regression problem using gradient descent
# with line search
func_evals_GDLS, w_GDLS = gradientDescentLS(lr, grad_lr, w0, max_iters=15000,
                                            stepsize=1, tol=1e-10)

# Solve the logistic regression problem using a variant of
# Nesterov accelerated gradient descent
func_evals_NAG, w_NAG = NAG(lr, grad_lr, w0, max_iters=15000,
                            stepsize=stepsize, tol=1e-10)

# Solve the logistic regression problem using the Nelder-Mead method
minimum_NM = scipy.optimize.fmin(lr, w0, xtol=1e-7, maxiter=60000,␣
 ↪full_output=1,
                                 retall=1)
```

```
C:\Users\eappe\AppData\Local\Temp\ipykernel_20908\2480348587.py:21:
RuntimeWarning: overflow encountered in exp
  return np.sum(np.log(1 + np.exp(-ytrain * np.matmul(Xtrain, w))))
C:\Users\eappe\AppData\Local\Temp\ipykernel_20908\758742428.py:14:
RuntimeWarning: Maximum number of iterations has been exceeded.
  minimum_NM = scipy.optimize.fmin(lr, w0, xtol=1e-7, maxiter=60000,
full_output=1,
```

[ ]:
```python
# Plot the objective values on a semilogy plot
plt.figure(1)
plt.semilogy(np.arange(0, func_evals_GDLS.size), func_evals_GDLS,
                       label="Gradient descent with line search")
plt.semilogy(np.arange(0, func_evals_NAG.size), func_evals_NAG,
                       label="Nesterov accelerated gradient descent")

NM_iters = np.array(minimum_NM[5])
func_evals_NM_iters = np.array([lr(NM_iters[i, :]) for i in np.arange(NM_iters.
 ↪shape[0])])
plt.semilogy(np.arange(0, func_evals_NM_iters.size), func_evals_NM_iters,
                       label="Nelder-Mead method")
```

```
plt.title("Value of logistic regression objective value as a funtion of␣
 ↪iteration count")
plt.xlabel("Iteration count")
plt.ylabel("Objective value at iteration")
plt.legend()
```

[ ]: <matplotlib.legend.Legend at 0x1e792e27e80>



1 Problem 1

In the above plot, we see that the Nesterov accelerated gradient descent takes fewer iterations than the gradient descent with line search to reach a minimizer of the objective function. However, the Nelder-Mead method is not optimal compared to either descent method since it does not use any derivative information. We see that with even sixty thousand iterations, Nelder-Mead does not minimize the objective function.

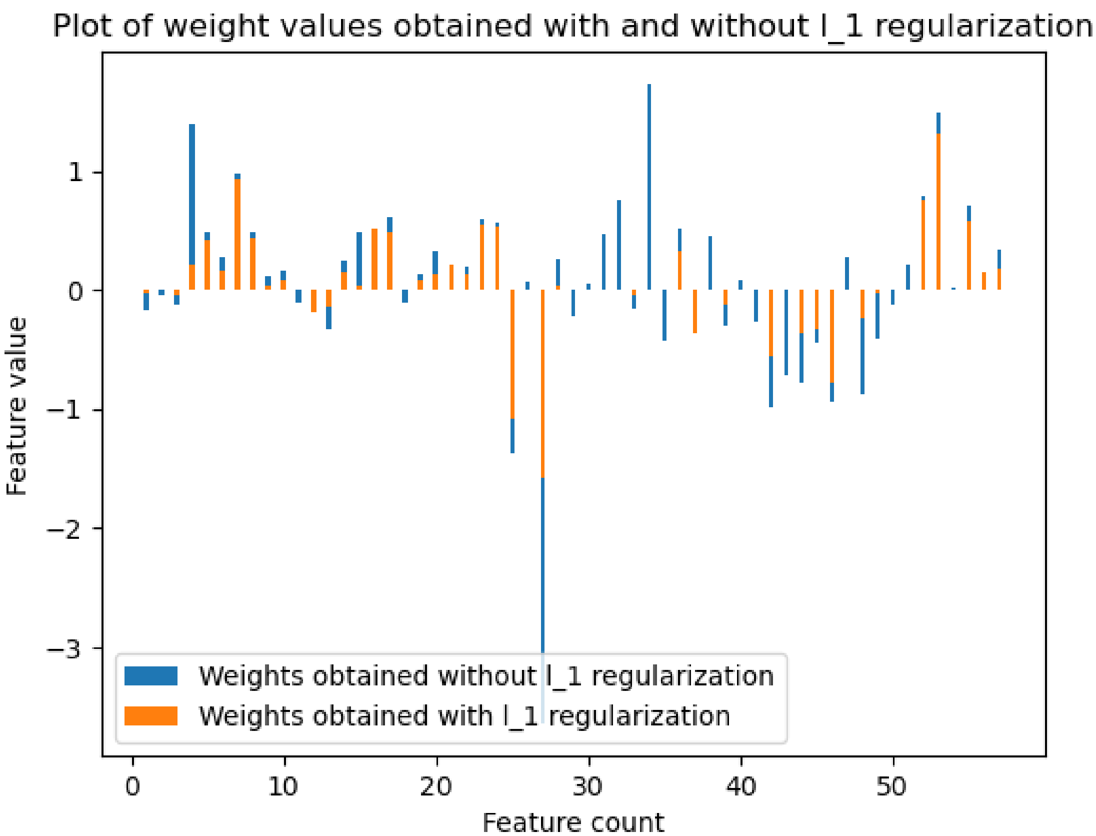
[ ]:
```
# Problem 2
# Re-run logistic regression on the spam data, but this time,
# add a l1-penalty term
l = 5

w_PGD = prox_GD(lr, grad_lr, l, l1_norm_prox, w0, use_g=True, max_iters=15000,
```

```
                 stepsize=stepsize, tol=1e-10)

# Print out the training and testing data classication accuracies corresponding
# to the non-regularized and regularized weights

# Perform classification on the training and testing data with
# non-regularized weights
training_computed_labels_nr = sigmoid(Xtrain @ w_NAG)
training_computed_labels_nr[training_computed_labels_nr > 0.5] = 1
training_computed_labels_nr[training_computed_labels_nr <= 0.5] = -1

testing_computed_labels_nr  = sigmoid(Xtest @ w_NAG)
testing_computed_labels_nr[testing_computed_labels_nr > 0.5] = 1
testing_computed_labels_nr[testing_computed_labels_nr <= 0.5] = -1

# Perform classification on the training and testing data with
# regularized weights
training_computed_labels_r = sigmoid(Xtrain @ w_PGD)
training_computed_labels_r[training_computed_labels_r > 0.5] = 1
training_computed_labels_r[training_computed_labels_r <= 0.5] = -1

testing_computed_labels_r   = sigmoid(Xtest @ w_PGD)
testing_computed_labels_r[testing_computed_labels_r > 0.5] = 1
testing_computed_labels_r[testing_computed_labels_r <= 0.5] = -1

print("Training data misclassification rate (non-regularized): ",
        (1 - (np.sum(ytrain == training_computed_labels_nr)/ ytrain.size)))
print("Testing data misclassification rate (non-regularized):  ",
        (1 - (np.sum(ytest  == testing_computed_labels_nr) / ytest.size)))
print()
print("Training data misclassification rate (regularized):     ",
        (1 - (np.sum(ytrain == training_computed_labels_r)/ ytrain.size)))
print("Testing data misclassification rate (regularized):      ",
        (1 - (np.sum(ytest  == testing_computed_labels_r) / ytest.size)))
```

```
Training data misclassification rate (non-regularized):  0.05220228384991843
Testing data misclassification rate (non-regularized):   0.05924479166666663

Training data misclassification rate (regularized):      0.052528548123980445
Testing data misclassification rate (regularized):       0.05794270833333337
```

[ ]:
```
# Make a barplot of the non-regularized and regularized weights
width = 0.3
labels = np.arange(1, (w_NAG.size) + 1)

plt.figure(2)
plt.bar(labels, w_NAG, width,
```

```
         label="Weights obtained without l_1 regularization")
plt.bar(labels, w_PGD, width,
         label="Weights obtained with l_1 regularization")
plt.xlabel("Feature count")
plt.ylabel("Feature value")
plt.title("Plot of weight values obtained with and without l_1 regularization")
plt.legend()
plt.show()
```



2 Problem 2

We see in the above barplot that adding the ℓ_1 regularization term to the logistic regression objective results in the regularized weights having smaller entry values in magnitude compared to the non-regularized weights. This is expected, as the purpose of regularization is to prevent overfitting of weight values to the training data. When we look at the misclassification rates on the training and testing data corresponding to the non-regularized and regularized weights, we see that the training data misclassification rate corresponding to the non-regularized weights is lower compared to the rate computed via the regularized weights. However, there the testing data misclassification rate corresponding to the regularized weights is lower compared to the rate computed via the non-regularized weights.

```
[ ]: # Problem 3
Y = pickle.load(open("SheppLogan_150x150.pkl", "rb"))

# Select roughly 10% of the pixel values in the Shepp Logan phantom to add
# random noise to
num_random_rows_cols = int(Y.size * 0.10)
random_rows_Y = np.random.choice(Y.shape[0], num_random_rows_cols, replace=True)
random_cols_Y = np.random.choice(Y.shape[1], num_random_rows_cols, replace=True)

Y_noisy = np.array(Y)
Y_noisy[random_rows_Y, random_cols_Y] += np.random.uniform(low=0.0, high=1.0,
                                                                              ␣
 ↪size=(num_random_rows_cols, ))

# Form the discrete gradient operator
n1 = Y.shape[0]
n2 = Y.shape[1]

D_n1 = scipy.sparse.spdiags((-1)*np.ones(n1), 0, m=n1, n=n1) + \
        scipy.sparse.spdiags(np.ones(n1 - 1), 1, m=n1, n=n1)
D_n2 = scipy.sparse.spdiags((-1)*np.ones(n2), 0, m=n2, n=n2) + \
        scipy.sparse.spdiags(np.ones(n2 - 1), 1, m=n2, n=n2)

I_n1 = scipy.sparse.spdiags(np.ones(n1), 0, m=n1, n=n1)
I_n2 = scipy.sparse.spdiags(np.ones(n2), 0, m=n2, n=n2)

L_h_tilde = scipy.sparse.kron(D_n2, I_n1)
L_v_tilde = scipy.sparse.kron(I_n2, D_n1)

def L(X):
    L_X = np.column_stack((L_h_tilde @ np.ndarray.flatten(X, "F"),
                           L_v_tilde @ np.ndarray.flatten(X, "F")))
    return L_X
```

```
[ ]: # Define the phi, g, and TV functions to compute the parameter tau
# used in the optimization problem
def phi(y_1, y_2):
    return np.sqrt(y_1**2 + y_2**2)

def g(y):
    result = 0
    for i in range(0, len(y)):
        result += phi(y[i, 0], y[i, 1])
    return result

def TV(X):
    return g(L(X))
```

```python
# Define analogous functions as above that are compatible with
# CVXPY
def L_cvx(X):
    L_X = cvx.vstack([cvx.vec(D_n1 @ X),
                      cvx.vec(D_n2 @ X.T)]).T
    return L_X

def TV_cvx(X):
    return cvx.mixed_norm(L_cvx(X), 2, 1)
```

[ ]:
```python
# Solve the TV de-noising problem in CVX
# using the discrete gradient operator
tau = (1/4)*TV(Y_noisy)
print(tau)

X = cvx.Variable((n1, n2))
obj = cvx.Minimize((1/2)*cvx.norm(X - Y_noisy, "fro"))
constraints = [TV_cvx(X) <= tau,
               X >= 0,
               X <= 1]
prob = cvx.Problem(obj, constraints)
prob.solve(verbose=True)
```

```
1025.6859808359723
===============================================================================
                                     CVXPY
                                     v1.3.0
===============================================================================
(CVXPY) Mar 17 05:50:08 PM: Your problem has 22500 variables, 3 constraints, and
0 parameters.
(CVXPY) Mar 17 05:50:08 PM: It is compliant with the following grammars: DCP,
DQCP
(CVXPY) Mar 17 05:50:08 PM: (If you need to solve this problem multiple times,
but with different data, consider using parameters.)
(CVXPY) Mar 17 05:50:08 PM: CVXPY will first compile your problem; then, it will
invoke a numerical solver to obtain a solution.
-------------------------------------------------------------------------------
                                  Compilation
-------------------------------------------------------------------------------
(CVXPY) Mar 17 05:50:08 PM: Compiling problem (target solver=ECOS).
(CVXPY) Mar 17 05:50:08 PM: Reduction chain: Dcp2Cone -> CvxAttr2Constr ->
ConeMatrixStuffing -> ECOS
(CVXPY) Mar 17 05:50:08 PM: Applying reduction Dcp2Cone
(CVXPY) Mar 17 05:50:08 PM: Applying reduction CvxAttr2Constr
(CVXPY) Mar 17 05:50:08 PM: Applying reduction ConeMatrixStuffing
(CVXPY) Mar 17 05:50:08 PM: Applying reduction ECOS
(CVXPY) Mar 17 05:50:08 PM: Finished problem compilation (took 3.443e-01
```

```
seconds).
-------------------------------------------------------------------------------
                                Numerical solver
-------------------------------------------------------------------------------
(CVXPY) Mar 17 05:50:08 PM: Invoking solver ECOS  to obtain a solution.
-------------------------------------------------------------------------------
                                    Summary
-------------------------------------------------------------------------------
(CVXPY) Mar 17 05:50:23 PM: Problem status: optimal
(CVXPY) Mar 17 05:50:23 PM: Optimal value: 1.162e+01
(CVXPY) Mar 17 05:50:23 PM: Compilation took 3.443e-01 seconds
(CVXPY) Mar 17 05:50:23 PM: Solver (including time spent in interface) took
1.450e+01 seconds
```

[ ]: 11.623631311361269

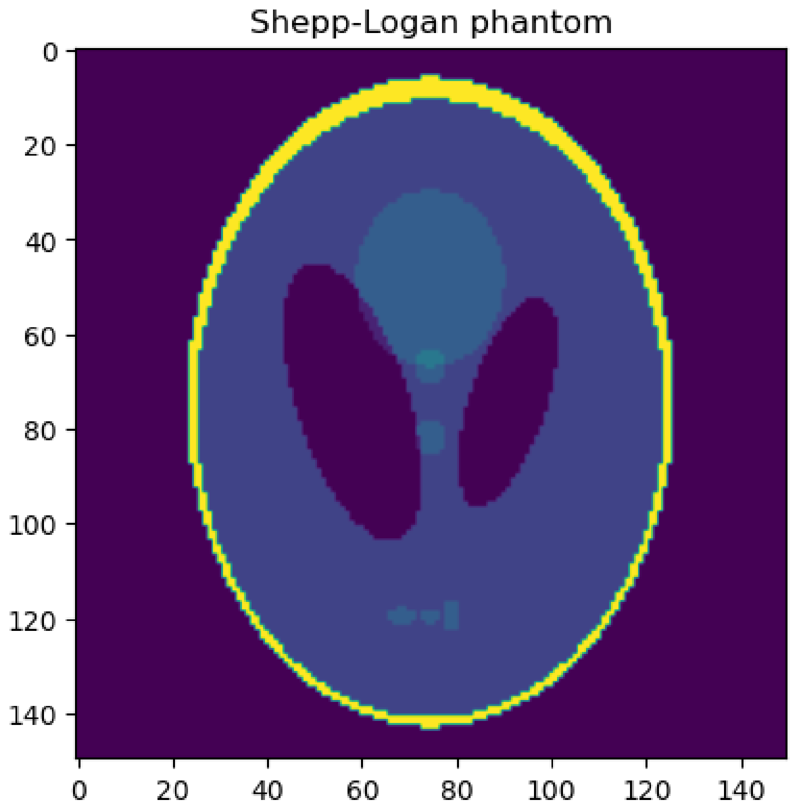
[ ]:
```python
# Solve the TV de-noising problem in CVX
# using the CVXPY's built-in TV function
X_2 = cvx.Variable((n1, n2))
obj_2 = cvx.Minimize((1/2)*cvx.norm(X_2 - Y_noisy, "fro"))
constraints_2 = [cvx.tv(X_2) <= tau,
                 X_2 >= 0,
                 X_2 <= 1]
prob_2 = cvx.Problem(obj_2, constraints_2)
prob_2.solve(verbose=True)
```

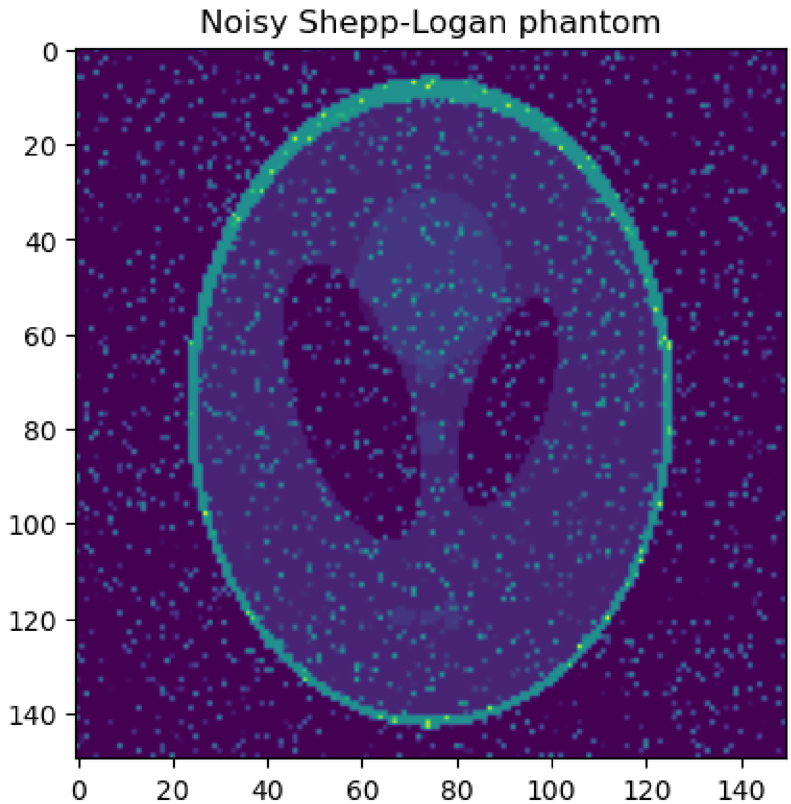
```
===============================================================================
                                     CVXPY
                                     v1.3.0
===============================================================================
(CVXPY) Mar 17 05:50:25 PM: Your problem has 22500 variables, 3 constraints, and
0 parameters.
(CVXPY) Mar 17 05:50:25 PM: It is compliant with the following grammars: DCP,
DQCP
(CVXPY) Mar 17 05:50:25 PM: (If you need to solve this problem multiple times,
but with different data, consider using parameters.)
(CVXPY) Mar 17 05:50:25 PM: CVXPY will first compile your problem; then, it will
invoke a numerical solver to obtain a solution.
-------------------------------------------------------------------------------
                                  Compilation
-------------------------------------------------------------------------------
(CVXPY) Mar 17 05:50:25 PM: Compiling problem (target solver=ECOS).
(CVXPY) Mar 17 05:50:25 PM: Reduction chain: Dcp2Cone -> CvxAttr2Constr ->
ConeMatrixStuffing -> ECOS
(CVXPY) Mar 17 05:50:25 PM: Applying reduction Dcp2Cone
(CVXPY) Mar 17 05:50:25 PM: Applying reduction CvxAttr2Constr
(CVXPY) Mar 17 05:50:25 PM: Applying reduction ConeMatrixStuffing
(CVXPY) Mar 17 05:50:25 PM: Applying reduction ECOS
```

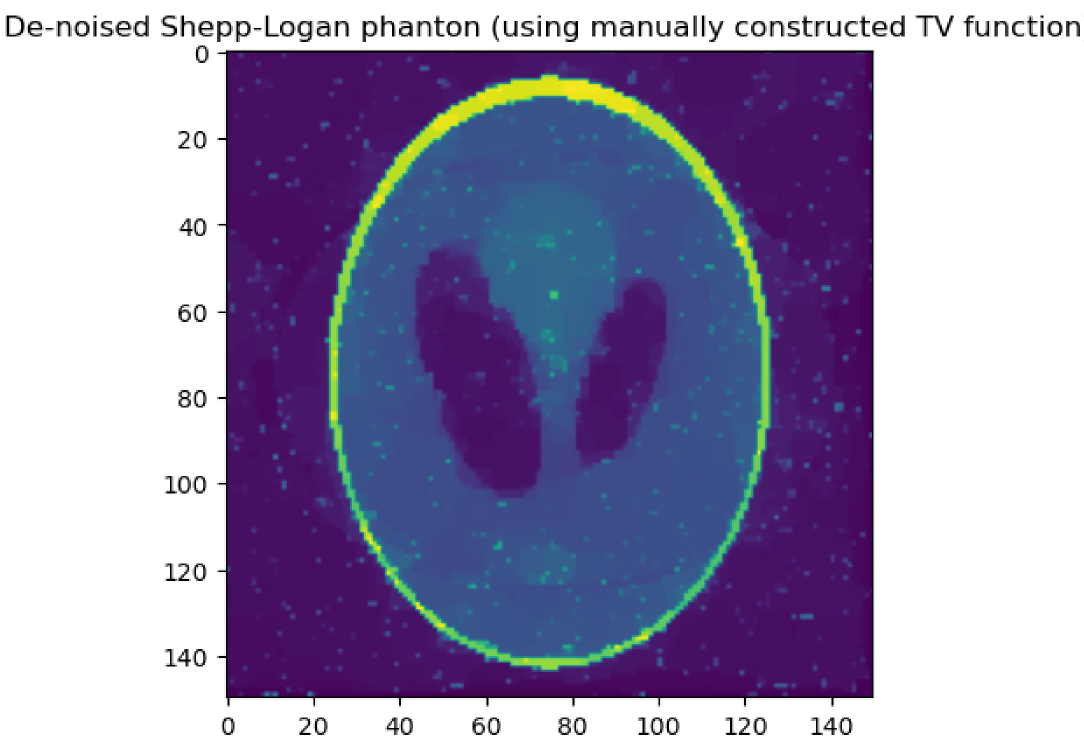
```
(CVXPY) Mar 17 05:50:25 PM: Finished problem compilation (took 2.563e-01
seconds).
-------------------------------------------------------------------------------
                                Numerical solver
-------------------------------------------------------------------------------
(CVXPY) Mar 17 05:50:25 PM: Invoking solver ECOS  to obtain a solution.
-------------------------------------------------------------------------------
                                    Summary
-------------------------------------------------------------------------------
(CVXPY) Mar 17 05:50:35 PM: Problem status: optimal
(CVXPY) Mar 17 05:50:35 PM: Optimal value: 1.117e+01
(CVXPY) Mar 17 05:50:35 PM: Compilation took 2.563e-01 seconds
(CVXPY) Mar 17 05:50:35 PM: Solver (including time spent in interface) took
9.846e+00 seconds
```

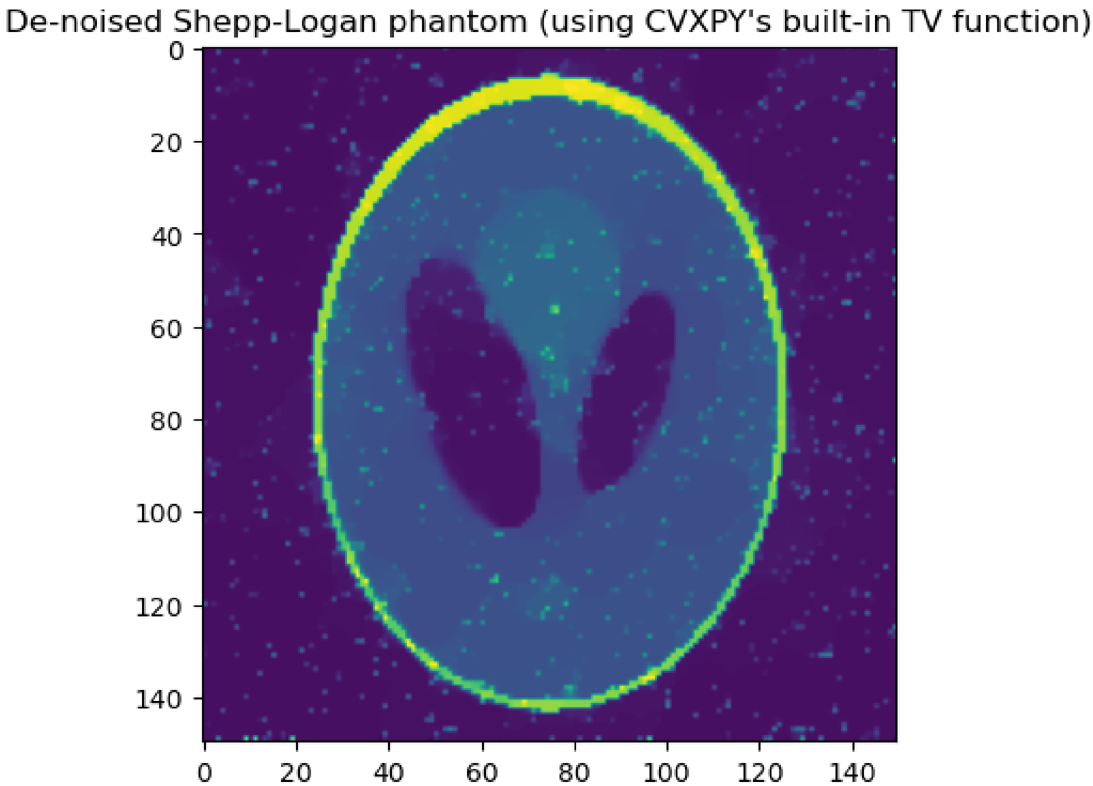
[ ]: 11.174569441547455

[ ]:
```
plt.figure()
plt.imshow(Y)
plt.title("Shepp-Logan phantom")

plt.figure()
plt.imshow(Y_noisy)
plt.title("Noisy Shepp-Logan phantom")

plt.figure()
plt.imshow(X.value)
plt.title("De-noised Shepp-Logan phanton (using manually constructed TV␣
 ↪function)")

plt.figure()
plt.imshow(X_2.value)
plt.title("De-noised Shepp-Logan phantom (using CVXPY's built-in TV function)")

plt.show()
```


Shepp-Logan phantom
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


Noisy Shepp-Logan phantom
0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140


De-noised Shepp-Logan phanton (using manually constructed TV function)
0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140



3 Problem 3

We solve the TV-denoising problem using our constructed TV function via CVXPY, and to verify that we correctly solved the problem, we also solve the same problem using CVXPY's built-in TV function. After solving both problems and plotting the de-noised images, we see that the two recovered images look more-or-less identical, with some differences probably arising from CVXPY's built-in TV function having different boundary conditions compared to our constructed TV function.