532 Activity 16 DEVIN BRESSER

1. a.)

$$f(\boldsymbol{w}) = (\boldsymbol{w} - \boldsymbol{w}_{LS})^T \boldsymbol{X}^T \boldsymbol{X} (\boldsymbol{w} - \boldsymbol{w}_{LS}) + c$$

where $\boldsymbol{w}_{LS} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$ and $c = \boldsymbol{y}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$. This assumes the n-by-p ($p < n$) matrix $\boldsymbol{X}$ is full rank. $f(\boldsymbol{w})$ is called a "quadratic form" in $\boldsymbol{w}$ since it is a quadratic function of $\boldsymbol{w}$.

$f(\underline{w})$ includes $(\underline{w} - \underline{w}_{LS})^T \cdot \; \cdots \; \cdot (\underline{w} - \underline{w}_{LS})$

Which is the dot product of $(\underline{w} - \underline{w}_{LS})$ with itself.

When $\underline{w} = \underline{w}_{LS}$, this term goes to 0 and only c remains.

But, when $|\underline{w} - \underline{w}_{LS}| > 0$ (order doesn't matter because the term is squared), the $(\underline{w} - \underline{w}_{LS})^T(\underline{w} - \underline{w}_{LS})$ term is positive and $f(\underline{w}) > c$.

b.)

b) Suppose $\boldsymbol{y} = \begin{bmatrix} 1 \\ 1/2 \\ 1 \\ 0 \end{bmatrix}$ and the 4-by-2 $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ has singular value decomposition $\boldsymbol{U} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$, $\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 1/2 \end{bmatrix}$, and $\boldsymbol{V} = \boldsymbol{I}$. Sketch a contour plot of $f(\boldsymbol{w})$ in the w_1-w_2 plane.

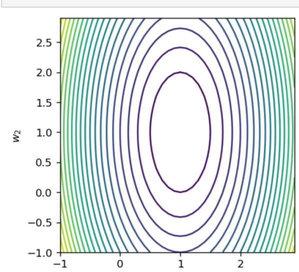
Question 1b)

In [3]:
```python
U = np.array([[1, 0], [0, 1], [0, 0], [0, 0]])
S = np.array([[1, 0], [0, 0.5]])
Sinv = np.linalg.inv(S)
V = np.eye(2)
X = U @ S @ V.T
y = np.array([[1], [0.5], [1], [0]])

### Find Least Squares Solution
w_ls = V @ Sinv @ U.T @ y
c = y.T @ y - y.T @ X @ w_ls

### Find values of f(w), the contour plot surface for
w1 = np.arange(-1,3,.1)
w2 = np.arange(-1,3,.1)
fw = np.zeros((len(w1), len(w2)))
for i in range(len(w2)):
    for j in range(len(w1)):
        w = np.array([ [w1[j]], [w2[i]] ])
        fw[i,j] = (w-w_ls).T @ X.T @ X @ (w-w_ls) + c

### Plot the countours
plt.figure(num=None, figsize=(4, 4), dpi=120)
plt.contour(w1,w2,fw,20)
plt.xlim([-1,3])
plt.ylim([-1,3])
plt.xlabel('$w_1$')
plt.ylabel('$w_2$')
plt.axis('square');
```



c.)

Comment: making S_2 smaller made the contours wider with respect to the range of w_2.

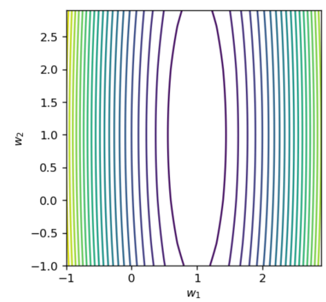
Question 1c)

In [4]:
```python
U = np.array([[1, 0], [0, 1], [0, 0], [0, 0]])
S = np.array([[1, 0], [0, 0.2]])
Sinv = np.linalg.inv(S)
V = np.eye(2)
X = U @ S @ V.T
y = np.array([[1], [0.2], [1], [0]])

### Find Least Squares Solution
w_ls = V @ Sinv @ U.T @ y
c = y.T @ y - y.T @ X @ w_ls

### Find values of f(w), the contour plot surface for
w1 = np.arange(-1,3,.1)
w2 = np.arange(-1,3,.1)
fw = np.zeros((len(w1), len(w2)))
for i in range(len(w2)):
    for j in range(len(w1)):
        w = np.array([ [w1[j]], [w2[i]] ])
        fw[i,j] = (w-w_ls).T @ X.T @ X @ (w-w_ls) + c

### Plot the countours
plt.figure(num=None, figsize=(4, 4), dpi=120)
plt.contour(w1,w2,fw,20)
plt.xlim([-1,3])
plt.ylim([-1,3])
plt.xlabel('$w_1$')
plt.ylabel('$w_2$')
plt.axis('square');
```



d.) Comment:

Adding asymmetric right singular vectors causes the contour lines to rotate about 45°.

Question 1d)

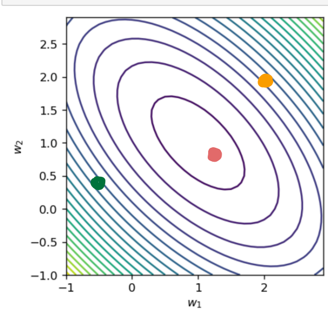
In [9]:
```python
U = np.array([[1, 0], [0, 1], [0, 0], [0, 0]])
S = np.array([[1, 0], [0, 0.5]])
Sinv = np.linalg.inv(S)
V = 1/np.sqrt(2) * np.array([[1,1],[1,-1]])
X = U @ S @ V.T
y = np.array([[np.sqrt(2)], [0], [1], [0]])

### Find Least Squares Solution
w_ls = V @ Sinv @ U.T @ y
c = y.T @ y - y.T @ X @ w_ls

### Find values of f(w), the contour plot surface for
w1 = np.arange(-1,3,.1)
w2 = np.arange(-1,3,.1)
fw = np.zeros((len(w1), len(w2)))
for i in range(len(w2)):
    for j in range(len(w1)):
        w = np.array([ [w1[j]], [w2[i]] ])
        fw[i,j] = (w-w_ls).T @ X.T @ X @ (w-w_ls) + c

### Plot the countours
plt.figure(num=None, figsize=(4, 4), dpi=120)
plt.contour(w1,w2,fw,20)
plt.xlim([-1,3])
plt.ylim([-1,3])
plt.xlabel('$w_1$')
plt.ylabel('$w_2$')
plt.axis('square');
```

1e.)



$w_0 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \to \begin{bmatrix} 2 \\ 2 \end{bmatrix}$

$w_0 = \begin{bmatrix} 0 \\ 2 \end{bmatrix} \to \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}$

$w_0 = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \to \begin{bmatrix} 1.25 \\ 0.75 \end{bmatrix}$

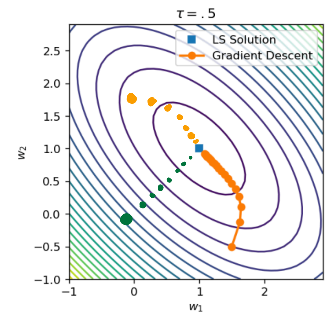
e.) $\nabla_{\underline{w}} f(\underline{w}) = 2\underline{X}^T(\underline{X}\,\underline{w} - \underline{y})$

2a.) $\tau < 2/\|X\|_{op}^2 = 2/1^2 = 2$.

2b.) Gradient descent will always move in the direction away from the gradient at w.
The direction of the gradient points in the direction of the steepest increase of the loss function, so we move in the other direction by subtracting.

In [47]:
```
w_init = np.array([[1.5],[-0.5]]) # complete this line with a 2x1 numpy array for the values
it = 20
tau = 0.5
W = graddescent(X,y,tau,w_init,it);


### Create plot
plt.figure(num=None, figsize=(4, 4), dpi=120)
plt.contour(w1,w2,fw,20)
plt.plot(w_ls[0],w_ls[1],"s", label="LS Solution")
plt.plot(W[0,:],W[1,:],'o-',linewidth=2, label="Gradient Descent")
plt.legend()
plt.xlim([-1,3])
plt.xlabel('$w_1$')
plt.ylim([-1,3])
plt.ylabel('$w_2$')
plt.title(r'$\tau = .5$');
plt.axis('square');
```



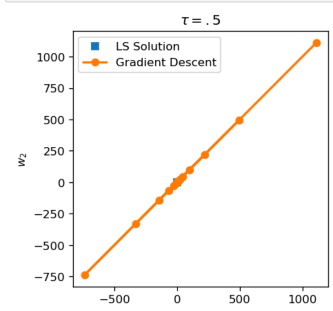
$w_{start} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

$w_{start} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$

2c.) The algorithm does not converge and instead quickly grows much too large.

In [49]:
```
w_init = np.array([[1.5],[-0.5]]) # complete this line with a 2x1 numpy array for the values
it = 20
tau = 2.5
W = graddescent(X,y,tau,w_init,it);


### Create plot
plt.figure(num=None, figsize=(4, 4), dpi=120)
plt.contour(w1,w2,fw,20)
plt.plot(w_ls[0],w_ls[1],"s", label="LS Solution")
plt.plot(W[0,:],W[1,:],'o-',linewidth=2, label="Gradient Descent")
plt.legend()
plt.xlim([-1,3])
plt.xlabel('$w_1$')
plt.ylim([-1,3])
plt.ylabel('$w_2$')
plt.title(r'$\tau = .5$');
plt.axis('square');
```



Question 2c)

2d.)

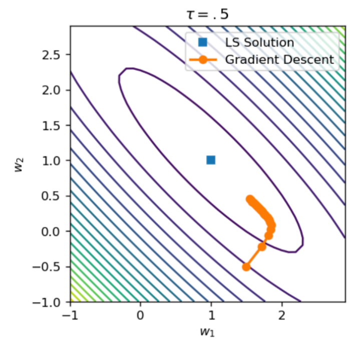
The algorithm does not converge within 20 iterations.

So more iterations are needed.

The larger the condition # of $\underline{X}$, the more iterations are needed to converge.

In [51]:
```
w_init = np.array([[1.5],[-0.5]]) # complete this line with a 2x1 numpy array for the values
it = 20
tau = 0.5
W = graddescent(X,y,tau,w_init,it);


### Create plot
plt.figure(num=None, figsize=(4, 4), dpi=120)
plt.contour(w1,w2,fw,20)
plt.plot(w_ls[0],w_ls[1],"s", label="LS Solution")
plt.plot(W[0,:],W[1,:],'o-',linewidth=2, label="Gradient Descent")
plt.legend()
plt.xlim([-1,3])
plt.xlabel('$w_1$')
plt.ylim([-1,3])
plt.ylabel('$w_2$')
plt.title(r'$\tau = .5$');
plt.axis('square');
```



2e.) As condition # ↑, # of iterations to convergence ↑.

In terms of the cost function, a higher condition # elongates the geometry of the cost function and so, if you start along the longer part, more iterations are needed to reach the bottom.