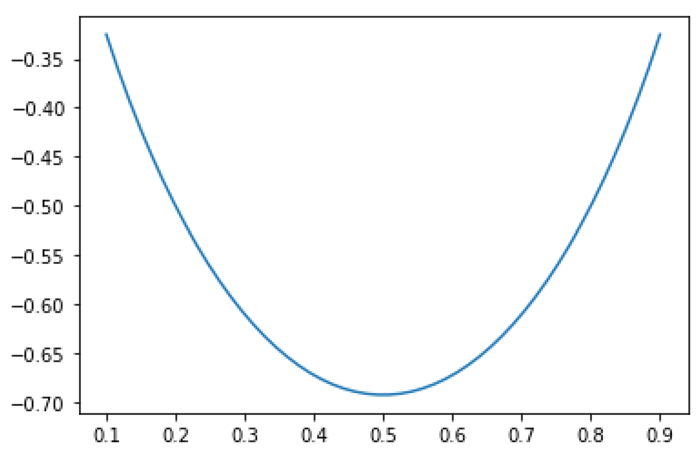
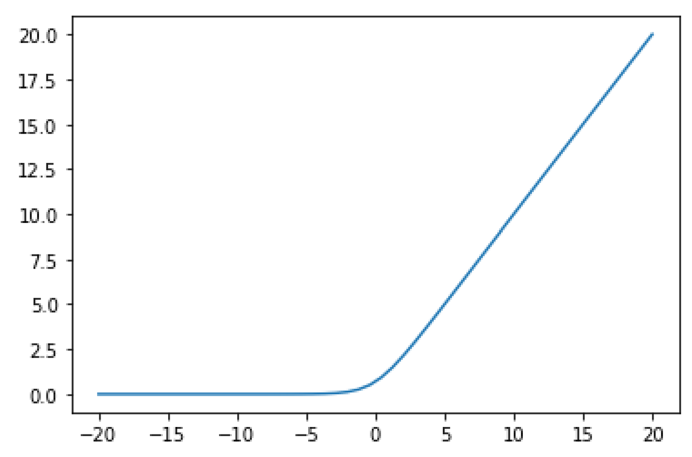
Problem 1: First-order condition for optimality (30 points)

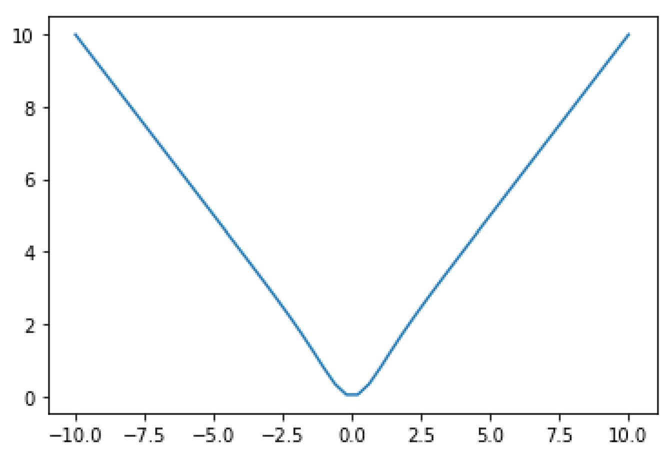
a) Setting the derivative of $g(w)$ to zero gives $g'(w) = \log(w) - \log(1-w) = 0$. Using the property of log, that $\log(a) - \log(b) = \log\left(\frac{a}{b}\right)$ this can be written equivalently as $\log\left(\frac{w}{1-w}\right) = 0$, and exponentiating each side this is equivalently $\frac{w}{1-w} = e^0 = 1$. Rearranging this is $w = 1 - w$, or equivalently $w = \frac{1}{2}$.



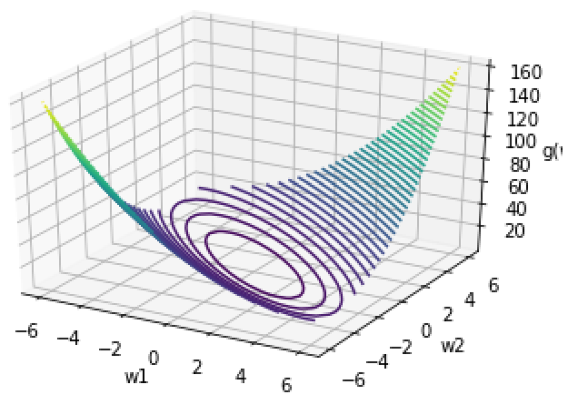
b) The first order system $\frac{\partial}{\partial w} g(w) = \frac{e^w}{1+e^w} = 0$. This fraction is zero only when $w = -\infty$



c) The first order system is $\frac{\partial}{\partial w} g(w) = \tanh(w) + w(1 - \tanh^2(w)) = 0$. The only point satisfying this equality is $w = 0$ (this can be seen by e.g., plotting the function itself).



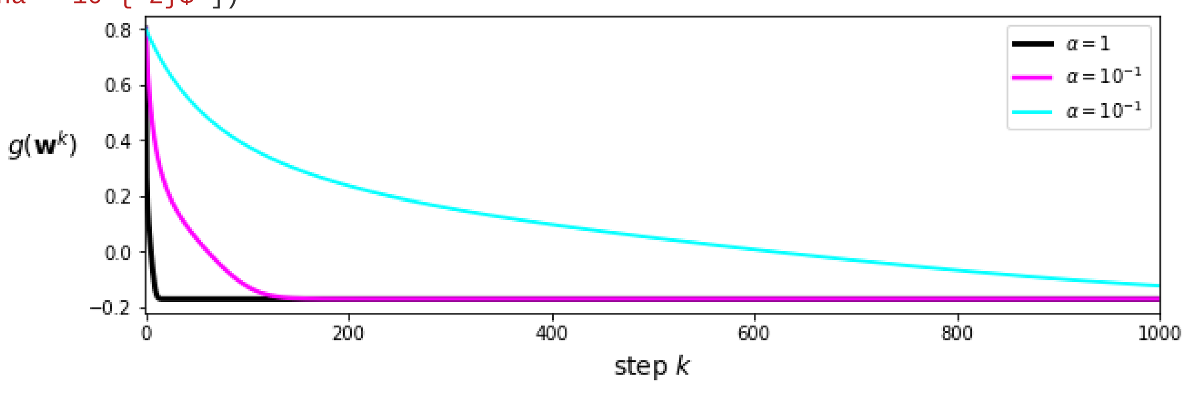
d) Setting up the first order system we have $\mathbf{C}\mathbf{w} = -\mathbf{b}$, and solving this system gives $\mathbf{w} = \begin{bmatrix} -0.4 \\ -0.2 \end{bmatrix}$



Problem 2: Try out gradient descent (30 points)

In [ ]:
```python
import matplotlib.pyplot as plt
# this module can be downloaded from
# https://github.com/jermwatt/machine_learning_refined/blob/gh-pages/mlrefined_libraries/mat
h_optimization_library/static_plotter.py
import static_plotter
plotter = static_plotter.Visualizer();

# gradient descent function - inputs: g (input function), alpha (steplength parameter), max_
its (maximum number of iterations), w (initialization)
# import automatic differentiator to compute gradient module
from autograd import grad
def gradient_descent(alpha,max_its,w):
    # cost for this example
    g = lambda w: 1/50*(w**4 + w**2 + 10*w)

    # the gradient function for this example
#    grad = lambda w: 1/50*(4*w**3 + 2*w + 10)
    gradient = grad(g)

    # run the gradient descent loop
    cost_history = [g(w)]        # container for corresponding cost function history
    for k in range(1,max_its+1):
        # evaluate the gradient, store current weights and cost function value
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # collect final weights
        cost_history.append(g(w))
    return cost_history
```

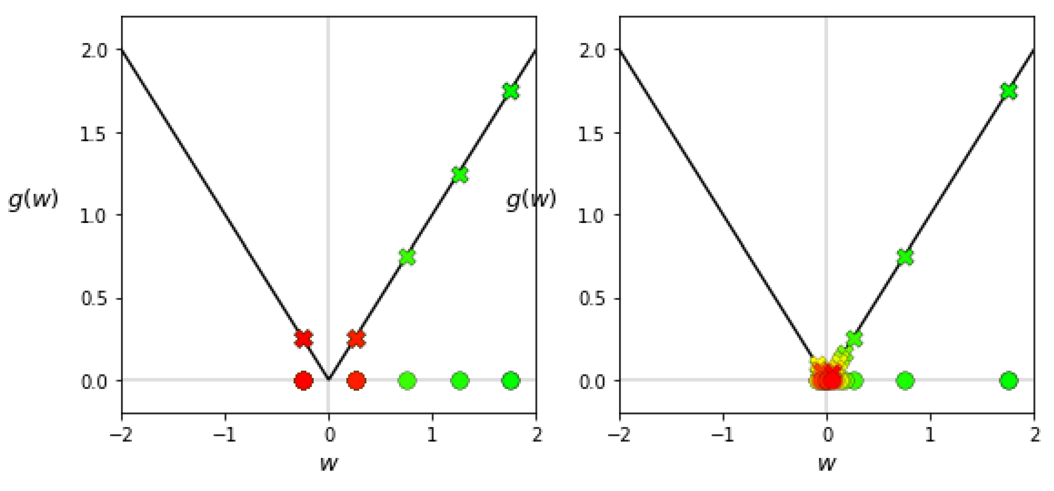
In [ ]:
```python
# initial point
w = 2.0
max_its = 1000

# first run
alpha = 10**(0)
cost_history_1 = gradient_descent(alpha,max_its,w)

alpha = 10**(-1)
cost_history_2 = gradient_descent(alpha,max_its,w)

alpha = 10**(-2)
cost_history_3 = gradient_descent(alpha,max_its,w)

# fig, ax = plt.subplots()
# l1, = ax.plot( cost_history_1)
# l2,l3 = ax.plot( cost_history_2,'--o',cost_history_3,'.')
#l2, l3 = ax.plot(t2, np.sin(2 * np.pi * t2), '--o', t1, np.log(1 + t1), '.')
#l4, = ax.plot(t2, np.exp(-t2) * np.sin(2 * np.pi * t2), 's-.')

# ax.legend((l1, l2,l3), [r'$\alpha = 1$',r'$\alpha = 10^{-1}$',r'$\alpha = 10^{-2}$'], loc
='upper right', shadow=True)
# ax.set_xlabel('step k')
# ax.set_ylabel('$g(W^k)$')
# plt.show()

# plot the cost function history for a given run
plotter.plot_cost_histories([cost_history_1,cost_history_2,cost_history_3],
            start = 0,points = False,labels = [r'$\alpha = 1$',r'$\alpha = 10^{-1}$',r'$\alp
ha = 10^{-2}$'])
```



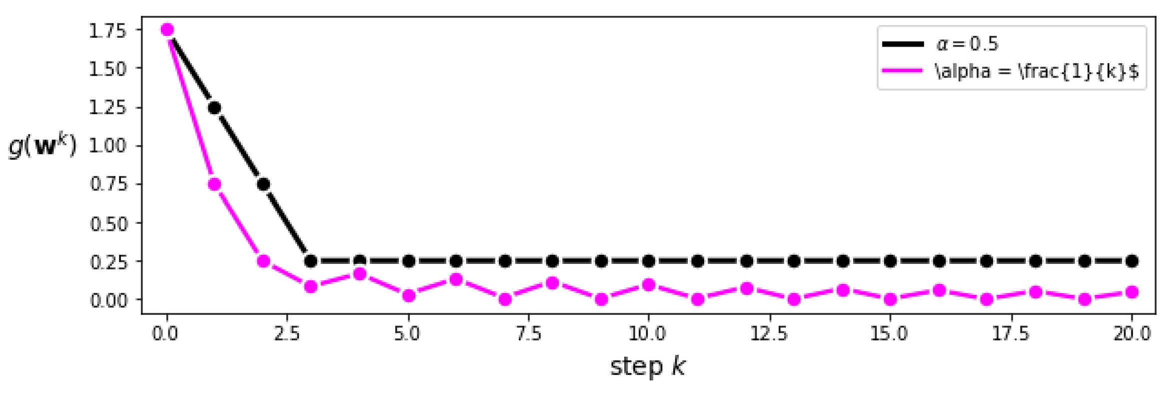
Problem 3: Compare fixed and diminishing steplength values (40 points)

In [ ]:
```python
# import automatic differentiator to compute gradient module
from autograd import grad
# gradient descent function - inputs: g (input function), alpha (steplength parameter),
#max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute gradient module using autograd
    gradient = grad(g)
    # run the gradient descent loop
    weight_history = [w] # container for weight history
    cost_history = [g(w)] # container for corresponding cost function history
    for k in range(max_its):
        # evaluate the gradient, store current weights and cost function value
        if alpha[0] == 'diminishing':
            lr = 1 / (k+1)
        else:
            lr = alpha[0]
        grad_eval = gradient(w)
        # take gradient descent step
        w = w - lr*grad_eval
        # record weight and cost
        weight_history.append(w)
        cost_history.append(g(w))
    return weight_history,cost_history
```

In [ ]:
```python
import autograd.numpy as np

# This code cell will not be shown in the HTML version of this notebook
# what function should we play with? Defined in the next line.
g = lambda w: np.abs(w)
# run gradient descent
alpha_choice = [0.5]; w = 1.75; max_its = 20;
weight_history_1,cost_history_1 = gradient_descent(g,alpha_choice,max_its,w)
alpha_choice = ['diminishing',0.5]; w = 1.75; max_its = 20;
weight_history_2,cost_history_2 = gradient_descent(g,alpha_choice,max_its,w)
# make static plot showcasing each step of this run
plotter.single_input_plot(g,[weight_history_1,weight_history_2],[cost_history_1,cost_history
_2],wmin = -2,wmax = 2,onerun_perplot = True)
```



In [ ]:
```python
# This code cell will not be shown in the HTML version of this notebook
# plot the cost function history for a given run
plotter.plot_cost_histories([cost_history_1,cost_history_2],start = 0,points = True,labels =
[r'$\alpha = 0.5$',r'\alpha = \frac{1}{k}$'])
```



In [ ]: