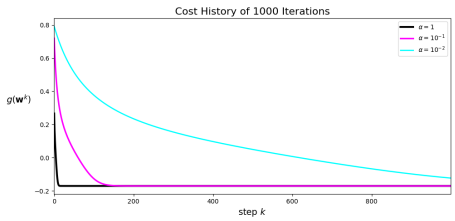
3.5 Try out gradient descent

Run gradient descent to minimize the function

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right) \tag{3.44}$$

with an initial point $w^0 = 2$ and 1000 iterations. Make three separate runs using each of the steplength values $\alpha = 1$, $\alpha = 10^{-1}$, and $\alpha = 10^{-2}$. Compute the derivative of this function by hand, and implement it (as well as the function itself) in `Python` using `NumPy`.

Plot the resulting cost function history plot of each run in a single figure to compare their performance. Which steplength value works best for this particular function and initial point?



$\alpha=1$ works best for this function, as:

① converge faster

② converge without oscillation

● 3-5 Try out gradient descent

```
from mlrefined_libraries.math_optimization_library import static_plotter


def gradient_decent(G, Gradient, study_rate, iteration, w):
    cost = []
    for k in range(1, iteration + 1):
        grad_eval = Gradient(w)
        w -= study_rate * grad_eval
        cost.append(G(w))
    return cost


if __name__ == '__main__':
    iteration = 1000
    w = 2
    G = lambda w: 1 / 50 * (w ** 4 + w ** 2 + 10 * w)
    delta_G = lambda w: 1 / 50 * (4 * w ** 3 + w * 2 + 10)
    study_rate = [1, 0.1, 0.01]
    cost1 = gradient_decent(G, delta_G, study_rate[0], iteration, w)
    cost2 = gradient_decent(G, delta_G, study_rate[1], iteration, w)
    cost3 = gradient_decent(G, delta_G, study_rate[2], iteration, w)
    plotter = static_plotter.Visualizer()
    plotter.plot cost histories(histories=[cost1, cost2, cost3], start=0,
                            labels=[r'$\alpha = 1$', r'$\alpha = 10^{-1}$',
r'$\alpha = 10^{-2}$'],
                            title="Cost History of 1000 Iterations"
                            )
```

3.8 Tune fixed steplength for gradient descent

Take the cost function

$$g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} \tag{3.45}$$

where $\mathbf{w}$ is an $N = 10$ dimensional input vector, and g is convex with a single global minimum at $\mathbf{w} = \mathbf{0}_{N\times 1}$. Code up gradient descent and run it for 100 steps using the initial point $\mathbf{w}^0 = 10 \cdot \mathbf{1}_{N\times 1}$, with three steplength values: $\alpha_1 = 0.001$, $\alpha_2 = 0.1$, and $\alpha_3 = 1$. Produce a cost function history plot to compare the three runs and determine which performs best.

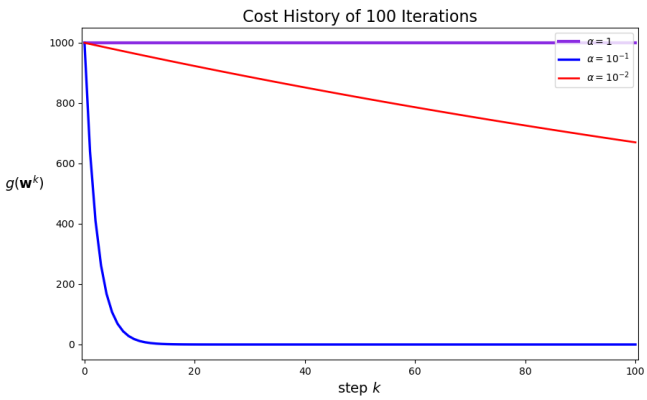
Sol : $\alpha = 0.1$ has the best performance

Since it has the fastest rate of convergence

Compared with:

① $\alpha = 1$ can not converge at all

② $\alpha = 0.1$ converge slower



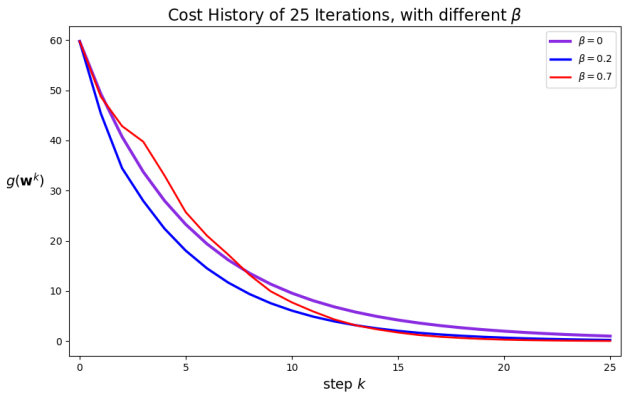
● 3-8 Tune fixed steplength for gradient descent

```
from mlrefined_libraries.math_optimization_library import static_plotter
from autograd import grad
from autograd import numpy as np

plotter = static_plotter.Visualizer()


class fixed_steplength():
    def __init__(self, function, Dim):
        self.fun = function
        self.ini_w = 10 * np.ones((Dim, 1))

    def gradient_descent(self, a):
        w = self.ini_w
        max_its = 100
        gradient = grad(self.fun)
        weight_his = [w]
        cost_his = [self.fun(w)]
        for k in range(max_its):
            grad_eval = gradient(w)
            w = w - a * grad_eval
            weight_his.append(w)
            cost his.append(self.fun(w))
        return weight_his, cost_his


if __name__ == '__main__':
    g = lambda w: np.dot(w.T, w)[0][0]
    FSG = fixed_steplength(g, 10)
    weight1, cost1 = FSG.gradient_descent(a=0.001)
    weight2, cost2 = FSG.gradient_descent(a=0.1)
    weight3, cost3 = FSG.gradient_descent(a=1)

    plotter.plot_cost_histories(histories=[cost3, cost2, cost1], start=0,
                                labels=[r'$\alpha = 1$', r'$\alpha = 10^{-1}$',
r'$\alpha = 10^{-2}$'],
                                title="Cost History of 100 Iterations"
                                )
```

3.9 Code up momentum-accelerated gradient descent

Code up the momentum-accelerated gradient descent scheme described in Section A.2.2 and use it to repeat the experiments detailed in Example A.1 using a cost function history plot to come to the same conclusions drawn by studying the contour plots shown in Figure A.3.

Sol: As illustrated in the history plot below:



① The zig-zagging behavior of gradient decent were ameliorated using the momentum-accelerated gradient decent.

② β=0.2 & 0.7, the cost converge closer to 0 when compared with β=0.

③ β=0.7 converge slower than β=0 & 0.2, that is because when a large β is chosen, each update will use less of each subsequent negative gradient direction.

Example A.1 Accelerating gradient descent on a simple quadratic

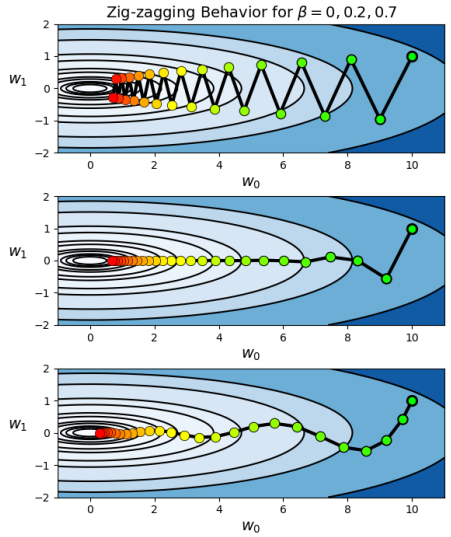
In this example we compare a run of standard gradient descent to the *momentum-accelerated* version using a quadratic function of the form

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w} \tag{A.9}$$

where $a = 0$, $\mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, and $\mathbf{C} = \begin{bmatrix} 0.5 & 0 \\ 0 & 9.75 \end{bmatrix}$.

Here we make three runs of 25 steps: a run of gradient descent and two runs of momentum-accelerated gradient descent using two choices of the parameter $\beta \in \{0.2, 0.7\}$. All three runs are initialized at the same point $\mathbf{w}^0 = [10 \ \ 1]^T$, and use the same steplength $\alpha = 10^{-1}$.

We show the resulting steps taken by the standard gradient descent run in the top panel of Figure A.3 (where significant zig-zagging is present), and the momentum-accelerated versions using $\beta = 0.2$ and $\beta = 0.7$ in the middle and bottom panels of this figure, respectively. Both momentum-accelerated versions clearly outperform the standard scheme, in that they reach a point closer to the true minimum of the quadratic. Also note that the overall path taken by gradient descent is smoother in the bottom panel, due to the larger value of its corresponding β.



● 3-9 Code up momentum-accelerated gradient descent

```
from mlrefined_libraries.math_optimization_library import static_plotter
from autograd import *
from autograd import numpy as np

plotter = static_plotter.Visualizer()


class momentum_gradient_decent:
    def __init__(self, function, w):
        self.fun = function
        self.ini_w = w

    def gradient_descent(self, a):
        w = self.ini_w
        max_its = 100
        gradient = grad(self.fun)
        weight_his = [w]
        cost_his = [self.fun(w)]
        for k in range(max_its):
            grad_eval = gradient(w)
            w = w - a * grad_eval
            weight_his.append(w)
            cost his.append(self.fun(w))
        return weight_his, cost_his

    def momentum(self, beta):
        w = self.ini_w
        max_its = 25
        gradient = value_and_grad(self.fun)
        weight_history = []
        cost_history = []
        a = 0.1
        h = np.zeros(w.shape)
        for k in range(1, max_its + 1):
            cost_eval, grad_eval = gradient(w)
            weight_history.append(w)
            cost_history.append(cost_eval)
            h = beta * h - (1 - beta) * grad_eval
            w = w + a * h
        weight_history.append(w)
        cost_history.append(self.fun(w))
        return weight_history, cost_history


if __name__ == '__main__':
    g = lambda w: (np.dot(0 * np.ones((2, 1)).T, w) + np.dot(np.dot(w.T,
np.array([[0.5, 0], [0, 9.75]])), w))[0]
```

```
    MGD = momentum_gradient_decent(g, np.array([10.0, 1.0]))
    weight1, cost1 = MGD.momentum(beta=0)
    weight2, cost2 = MGD.momentum(beta=0.2)
    weight3, cost3 = MGD.momentum(beta=0.7)
    his = [weight1, weight2, weight3]
    gs = [g, g, g]
    plotter.two_input_contour_vert_plots(title=r"Zig-zagging Behavior for $\beta=0,
0.2, 0.7$", gs=gs, histories=his,
                                         num_contours=30, xmin=-1, xmax=11,
                                         ymin=-2.0,
                                         ymax=2)
    plotter.plot_cost_histories(histories=[cost1, cost2, cost3], start=0,
                                labels=[r'$\beta = 0$', r'$\beta = 0.2$', r'$\beta =
0.7$'],
                                title=r"Cost History of 25 Iterations, with different
$\beta$"
                                )
```

3.10 Slow-crawling behavior of gradient descent

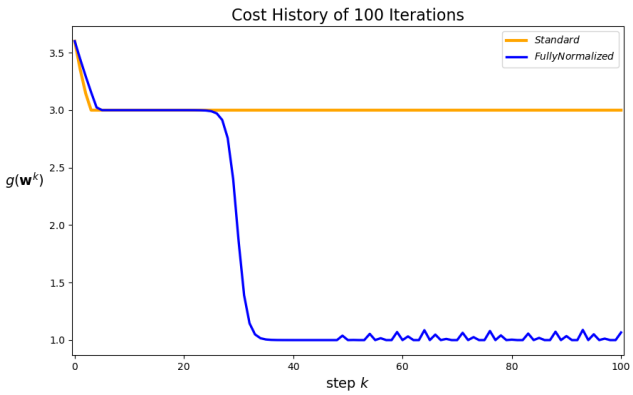
In this exercise you will compare the standard and fully normalized gradient descent schemes in minimizing the function

$$g(w_1, w_2) = \tanh(4\,w_1 + 4\,w_2) + \max(1, 0.4\,w_1^2) + 1. \tag{3.46}$$

Sol: the fully magnitude-normalized gradient decent has steplength $\frac{\alpha}{\|\nabla g(w^{k-1})\|_2}$ that can adjust itself at each step based on the magnitude of the gradient. it can ameliorate slow-crawling near minima and saddle point.

As can be illustrated below, after reaching a relative low cost normalized gradient decent enables the cost converge closer to 0.

w is initialized at [2,2]:



3-10 Slow-crawling behavior of gradient descent

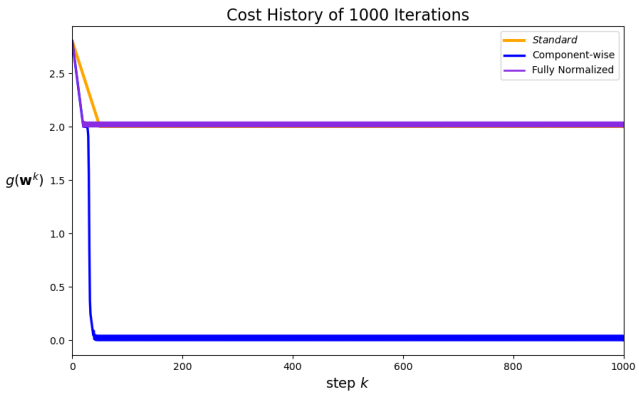
```
from mlrefined_libraries.math_optimization_library import static_plotter
from autograd import *
from autograd import numpy as np

plotter = static_plotter.Visualizer()


class normalized_gradient_decent:
    def __init__(self, function, w):
        self.fun = function
        self.ini_w = w

    def gradient_descent(self, normalized=False):
        gradient = value_and_grad(g)
        w = self.ini_w
        weight_his = []
        cost_his = []
        max_its = 100
        alpha = 0.1
        for k in range(1, max_its + 1):
            cost_eval, grad_eval = gradient(w)
            weight_his.append(w)
            cost his.append(cost eval)
            if normalized:
                grad_norm = np.linalg.norm(grad_eval)
                if grad norm == 0:
                    grad_norm += 10 ** -6 * np.sign(2 * np.random.rand(1) - 1)
                grad_eval /= grad_norm
            else:
                grad_eval = grad_eval
            w = w - alpha * grad_eval
        weight_his.append(w)
        cost_his.append(self.fun(w))
        return weight_his, cost_his

if __name__ == '__main__':
    g = lambda w: np.tanh(4 * w[0] + 4 * w[1]) + max(0.4 * w[0] ** 2, 1) + 1
    w = np.array([2.0, 2.0])
    print(w)
    NGD = normalized_gradient_decent(g, w)
    weight1, cost1 = NGD.gradient_descent()
    weight2, cost2 = NGD.gradient_descent(normalized=True)

    plotter.plot_cost_histories(histories=[cost1, cost2], start=0,
                            labels=[r'$Standard$', r'$Fully Normalized$'],
                            title="Cost History of 100 Iterations"
                            )
```

3.11 **Comparing normalized gradient descent schemes**

Code up the full and component-wise normalized gradient descent schemes and repeat the experiment described in Example A.4 using a cost function history plot to come to the same conclusions drawn by studying the plots shown in Figure A.6.

Sol: The cost-history for 1000 iterations.

Result is illustrated as below:



Our result are completely consistent with Fig A.6:

For "fully normalized", its cost function can not coverge to the minimum, because w_2 is almost 0 everywhere.

Example A.4 **Full versus component-normalized gradient descent**

In this example we use the function

$$g(w_1, w_2) = \max(0, \tanh(4w_1 + 4w_2)) + |0.4w_1| + 1 \tag{A.25}$$

to show the difference between full and component-normalized gradient descent steps on a function that has a very narrow flat region along only a single dimension of its input. Here this function – whose surface and contour plots can be seen in the left and right panels of Figure A.6, respectively – is very flat along the w_2 dimension for any fixed value of w_1, and has a very narrow valley leading towards its minima in the w_2 dimension where $w_1 = 0$. If initialized at a point where $w_2 > 2$ this function cannot be minimized very easily using standard gradient descent *or* the fully normalized version. In the latter case, the magnitude of the partial derivative in w_2 is nearly zero everywhere, and so fully normalizing makes this contribution smaller, and halts progress. In the top row of the figure we show the result of 1000 steps of fully normalized gradient descent starting at the point $\mathbf{w}^0 = [2\ \ 2]^T$, colored green (at the start of the run) to red (at its finale). As can be seen, little progress is made.

In the bottom row of the figure we show the results of using component-normalized gradient descent starting at the same initialization and employing the same steplength. Here we only need 50 steps in order to make significant progress.

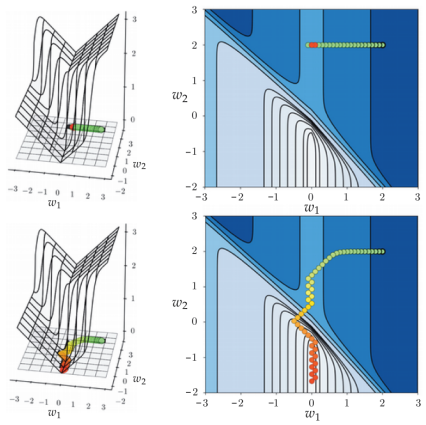


Figure A.6 Figure associated with Example A.4. See text for details.

● 3-11 Comparing normalized gradient descent schemes

```
from mlrefined_libraries.math_optimization_library import static_plotter
from autograd import *
from autograd import numpy as np

plotter = static_plotter.Visualizer()


class normalized_gradient_decent:
    def __init__(self, function, w):
        self.fun = function
        self.ini_w = w

    def gradient_descent(self, normalized=None, max_its=100):
        gradient = value_and_grad(g)
        w = self.ini_w
        weight_his = []
        cost_his = []

        alpha = 0.1
        for k in range(1, max_its + 1):
            cost_eval, grad_eval = gradient(w)
            weight_his.append(w)
            cost his.append(cost eval)
            if normalized == "full":
                grad_norm = np.linalg.norm(grad_eval)
                if grad norm == 0:
                    grad_norm += 10 ** -6 * np.sign(2 * np.random.rand(1) - 1)
                grad_eval /= grad_norm
            elif normalized == "Component-wise":
                component_norm = np.abs(grad_eval) + 10 ** (-8)
                grad_eval /= component_norm
            else:
                grad_eval = grad_eval
            w = w - alpha * grad_eval
        weight_his.append(w)
        cost_his.append(self.fun(w))
        return weight_his, cost_his


if __name__ == '__main__':
    g = lambda w: np.max(np.tanh(4 * w[0] + 4 * w[1]), 0) + np.max(np.abs(0.4 *
w[0]), 0) + 1
    w = np.array([2.0, 2.0])
    max_its = 1000
    print(w)
    NGD = normalized_gradient_decent(g, w)
    weight1, cost1 = NGD.gradient_descent(max_its=max_its)
```

```
    weight2, cost2 = NGD.gradient_descent(normalized="Component-wise",
max_its=max_its)
    weight3, cost3 = NGD.gradient_descent(normalized="full", max_its=max_its)

    plotter.plot_cost_histories(histories=[cost1, cost2, cost3], start=0,
                            labels=[r'$Standard$', r'Component-wise', r'Fully
Normalized'],
                            title="Cost History of {} Iterations".format(max_its)
                            )
```