University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

University College Cork

AM6020

Conjugate Gradient Method

Vigneshwar Govindarajalu Dinakaran
Barry O'Donnell
Kai Zhang

Supervisor: Dr Andrew Keane

Submitted: November 2021

Contents

0.1 Author contributions

	Sections	Coding
Vignesh	Introduction & Conclusion	Contour Plots
Kai	Steepest / Gradient Descent	Steepest Descent
Barry	Conjugate Gradient Method	Conjugate Gradient Descent

Chapter 1

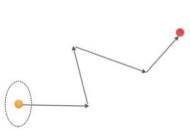
Introduction

1.1 Importance

To solve the unconstrained optimization problem, we usually take the derivatives of the target function iteratively. Gradient descent is one such optimization method where we minimize the target function by iteratively moving in the direction of the steepest descent, which is the negative of the gradient.

Gradient Descent and Steepest Descent are quite similar with the only difference being with the learning rate. Learning rate is the parameter that defines how far a step one should take along the negative gradient direction. While the descent direction is determined from the gradient of the loss function, the learning rate determines how big a step is taken in that direction. A high learning rate might make the learning jump over minima but a low learning rate will either take too long to converge or get stuck in an undesirable local minimum. The solution is an adaptive learning rate in steepest descent instead of setting upfront as in gradient descent methodology.

But the shortcoming with Steepest descent is that the searching route may look zigzaggy. This means the line search might move along the same direction during any of the further iterations. It will be so much more effective if we can find the best direction at each step. A very intuitive idea here is that we want to decompose our direction into a set of orthogonal search directions.So the work done along one direction is not reversed by further directions. The linear conjugate gradient method was proposed by Hestenes and Stiefel [1]



(a) Gradient Descent

(b) Conjugate Gradient Descent

1.2 Algorithm Design

Let's start with our traditional linear equation

$$Ax = b \tag{1.1}$$

We will be solving for the vector x here, while A is the symmetric, positive definitive coefficient matrix and b is a known vector.

To solve this equation for x is equivalent to a minimization problem of a convex function $f(x)$ below. This is called the quadratic form, where c is just a scalar constant.

$$min\ f(x) = 1/2 * x^T Ax - b^T x + c \tag{1.2}$$

This gives us

$$f'(x) = Ax - b \tag{1.3}$$

Thus minimum of $f(x)$ is a solution to $Ax = b$

We first choose the initial point of x_0 and update our next direction:

$$d_0 = \nabla f(x_0) \tag{1.4}$$

If the above function equals 0, then we reach the minimum. If it does not equal 0, we update it with:

$$x_{i+1} = x_i + \alpha_i d_i \tag{1.5}$$

Where α will be

$$\alpha_k = -\frac{d_i^T \nabla f(x_i)}{d_i^T A d_i} \tag{1.6}$$

And the new direction will be

$$d_i = -\nabla f(x_i) + \gamma_{i-1} d_{i-1} \tag{1.7}$$

The new search directions are orthogonal to the previous ones validating the equation

$$d_i^T A d_{i+1} = 0 \tag{1.8}$$

And after n iterations on n orthogonal directions, we will be able to find the solution.

Chapter 2

The Method of Steepest Descent / Gradient descent

2.1 Simple explanation

Objective: To find the optimal solution of objective function: $f(x)$.

That is, get the minimum of this function.

Gradient descent only uses information about the `first derivative` of the objective function – as the name gradient suggests.

And it's supposed to take the `direction of the "fastest drop"` in the value of the target function as the **search direction**, which is where the name "fastest drop" comes from

The idea of gradient descent method is to select an appropriate initial value x_0, keep iteratively updating the value of x, minimize the objective function, and finally converge.

Search direction and ***step size*** are the two most important things to update x.

Let's look at an example of a function.

$$f(x_1, x_2) = (x_1^2 - 2) + (x_1 - 2x_2)^2$$

We want to know the approximate optimal solution to this equation

Let us focus on search direction first.

So what's the `fastest way` to decrease the function?

2.2 Search direction

The direction that the value of the function decrease fast.

We use d for direction and g for gradient.

The objective function $f(x)$ is Taylor expanded at the point x_k

$$f(x) = f(x_k) + ag_k d_k^T + \mathcal{O}(a)$$

$\mathcal{O}(a)$ can be ignore, when $g_k d_k^T < 0$, $f(x) < f(x_k)$, the function is decreasing dk is a downward direction at this time

When dk is equal to exactly what, can the objective function value drop the fastest we can use `Cauchy - Schwartz inequality`

$$(a_1b_1 + a_2b_2 + \cdots + a_nb_n) \leq \sqrt{(a_1^2 + a_2^2 + \cdots + a_n^2)}\sqrt{(b_1^2 + b_2^2 + \cdots + b_n^2)}$$

It is equal if and only if

$$\frac{a_1}{b_1} = \frac{a_2}{b_2} = \cdots = \frac{a_n}{b_n}$$

So we can get $\left|\left|d_k^T g_k\right|\right| \leq |d_k|\,|g_k|$

when $d_k = -g_k$,The minimum $d_k^T g_k < 0$, and then $f(x)$ drops the most.

Therefore, $-g_k$ is the fastest descending direction. Direction of negative gradient.

We can set the convergence accuracy ϵ, if $|d_k| < \epsilon$, we can stop.

2.3 Step size

The step size is solved using `Armijo-Goldstein` [2] criterion

Core ideas:

1. The value of the objective function should be reduced enough;
2. The step size α of one-dimensional search should not be too small.

Make α not too large or too small.

Goldstein-Armijo line-search

When computing step length α of $f(x_k+\alpha d_k)$, the new point should sufficiently decrease f and ensure that α is away from 0. Thus, we use following bound is used

$$0 < -\alpha_k \mu_1 \nabla f(x_k)^T d_k \leq f(x_k) - f(x_{k+1}) \leq -\alpha_k \mu_2 \nabla f(x_k)^T d_k$$

where $0 < \mu_1 \leq \mu_2 < 1,\ \alpha_k > 0$ and $\nabla f(x_k)^T d_k < 0$. The upper an lower bounds in the above principle ensure α_k is a good choice.

Algorithm: Choose parameters μ_1, μ_2, ρ_1, ρ_2, α_0 (for example $\mu_1 = 0.2$, $\mu_2 = 0.8$, $\rho_1 = 1/2$, $\rho_2 = 1.5$, $\alpha_0 = 1$).

Step 1: i=0

Step 2: `if` $-\alpha_i \mu_1 \nabla f(x_k)^T d_k > f(x_k) - f(x_k + \alpha_i d_k)$
$\Longrightarrow \alpha_{i+1} = \rho_1 \alpha_i$, $i = i+1$, `goto` **Step 2**
`if` $f(x_k) - f(x_k + \alpha_i d_k) > -\alpha_i \mu_2 \nabla f(x_k)^T d_k$
$\Longrightarrow \alpha_{i+1} = \rho_2 \alpha_i$, $i = i+1$, `goto` **Step 2**

Step 3: $x_{k+1} = x_k + \alpha_i d_k$

Chapter 3

The Method of Conjugate Directions

3.1 Conjugacy

In steepest descent, we search for the immediate steepest decline along some orthogonal direction and choose the steepest route. Perhaps this method could be optimized by altering the view on the plane such that instead of zig-zagging our way down the elliptical bowl, we instead make a more calculated move that will result in a smoother descent with less changing of directions.

The shape of a surface is dictated by the matrix **A** from the quadratic formula. Consider a transformation of this surface, wherein the elliptical bowl now resembles a nice, concentric, circular bowl. Orthogonal vectors on this transformed plane are A-orthogonal, and will be the focus of this method. It is important to note that though these vectors are A-orthogonal, they may not retain this orthogonal characteristic on the original plane.

3.2 Completion in n steps

Define the step process as

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)}, \tag{3.1}$$

where $x_{(i)}$ is the starting position, $x_{(i+1)}$ is the end position, $\alpha_{(i)}$ is some scaling vector that determines the length of our step, and $d_{(i)}$ is the the direction we move in. To determine $\alpha_{(i)}$, we must consider the error $e_{(i)}$ of the move. If we move in orthogonal steps in the original plane, then the error of the current position is orthogonal to the direction we previously moved in, or

$$d_{(i)}^T e_{(i+1)} = 0 \tag{3.2}$$

$$d_{(i)}^T \left(e_{(i)} + \alpha_{(i)} d_{(i)} \right) = 0 \tag{3.3}$$

$$\alpha_{(i)} = -\frac{d_{(i)}^T e_{(i)}}{d_{(i)}^T d_{(i)}}. \tag{3.4}$$

This is great! We now have an expression for $\alpha_{(i)}$ in terms of the previous direction $d_{(i)}$, but with an additional $e_{(i)}$. If we knew $e_{(i)}$, then the problem is solved (we would

immediately know where the local minimum is), so perhaps we should instead try our method of transforming the plane we move in.

Consider instead that our directions $d_{(i)}$ and $d_{(j)}$ are A-orthogonal, or conjugate, therefore,

$$d_{(i)}^T A d_{(j)} = 0.$$

Using this transformation of plane, 3.4 becomes:

$$\alpha_{(i)} = -\frac{d_{(i)}^T A e_{(i)}}{d_{(i)}^T d_{(i)}} \tag{3.5}$$

$$= \frac{d_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}}, \tag{3.6}$$

where $r_{(i)}$ is the residuals of the current move. Finally we have an expression for $\alpha_{(i)}$ in terms we already have.

As per the title of this section, the reader may wonder how we know that the path can be computed in n steps. Consider the error term $e_{(0)}$. It is the 'error' of your starting position with respect to the local minimum. To know what this term is would imply that you know where the minimum is. It might be easiest to imagine as "once you've reached the end of your journey, measure how far you've come!" Which is exactly what we'll do. This error term can be thought as the summation of all your steps towards the local minimum, or

$$e_{(0)} = \sum_{j=0}^{n-1} \delta_j d_{(j)}, \tag{3.7}$$

where δ_j is the length of each component for the direction we travelled in. As we have done before, we utilise the A-orthogonal nature of $d_{(i)}$. Start by multiplying both sides by $d_{(k)}^T A$ to get

$$d_{(k)}^T A e_{(0)} = \sum_{j=0}^{n-1} \delta_j d_{(k)}^T A d_{(j)} \tag{3.8}$$

$$d_{(k)}^T A e_{(0)} = \delta_k d_{(k)}^T A d_{(k)} \quad (d_{(k)}^T A d_{(j)} = 0 \text{ for } k \neq j) \tag{3.9}$$

$$\delta_{(k)} = \frac{d_{(k)} A e_{(0)}}{d_{(k)}^T A d_{(k)}} \tag{3.10}$$

$$= \frac{d_{(k)} A \left(e_{(0)} + \sum_{i=0}^{k-1} \alpha_{(i)} d_{(i)} \right)}{d_{(k)}^T A d_{(k)}} \quad (A\text{-orthogonality of } d \text{ vectors}) \tag{3.11}$$

$$= \frac{d_{(k)} A e_{(k)}}{d_{(k)}^T A d_{(k)}}. \quad (\text{from } 3.1) \tag{3.12}$$

This term looks strikingly similar to 3.6. It turns out that $\alpha_{(i)} = -\delta_{(i)}$. Lets further explore the error term.

$$e_{(i)} = e_{(0)} + \sum_{j=0}^{i-1} \alpha_{(j)} d_{(j)} \tag{3.13}$$

$$= \sum_{j=0}^{n-1} \delta_{(j)} d_{(j)} - \sum_{j=0}^{i-1} \delta_{(j)} d_{(j)} \tag{3.14}$$

$$= \sum_{j=1}^{n-1} \delta_{(j)} d_{(j)}. \tag{3.15}$$

How neat is that! After n steps, each component is cut away, meaning that $e_{(n)} = 0$ and we land precisely on the minimum. ■

3.3 Gram-Schmidt Conjugation

We are satisfied with moving to an A space as it appears that we complete our algorithm in n steps, but now we need a way to generate directions to move in with each iteration. The mathematics behind this method of generation is quite complex and beyond the scope of this paper. From [3] we obtain a method of generating Gram-Schmidt constants,

$$\beta_{(i+1)} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}}. \tag{3.16}$$

In short, Gram-Schmidt makes use of "remembering" previous directions we have travelled in to construct the next direction. Using A-orthogonality, this construction can be milled down until we need only memory of the previous residuals.

3.4 Conjugate Gradient Method

Start by choosing some starting point $x_{(0)}$ and initialise your direction of travel and your residuals vector.

$$d_{(0)} = r_{(0)} = b - Ax_{(0)}.$$

If your residuals $r_{(0)}$ are low enough and within some accuracy you are happy with, then congratulations, you have found the local minimum! If you are not satisfied then follow this algorithm.

Step 1: Construct your next step length as,

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}}, \tag{3.17}$$

and move to your next location,

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)}. \tag{3.18}$$

Step 2: Calculate the residuals from this move,

$$r_{(i+1)} = r_{(i)} - \alpha_{(i)} d_{(i)}. \tag{3.19}$$

Step 3: Calculate the Gram-Schmidt constant,

$$\beta_{(i+1)} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}}. \tag{3.20}$$

Step 4: Finally, adjust your direction of motion for the next step iteration,

$$d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)}. \tag{3.21}$$

Step 5: Repeat steps 1 to 4 until you have reduced $r_{(i)}$ to a value you are satisfied with.

And just like that, we have found the local minimum of a surface!

Let's calculate the computational expense of such an operation. Matrix multiplication is an $\mathcal{O}(n^2)$ operation, while dot products is $\mathcal{O}(n)$. Therefore for large n, conjugate gradient method is an $\mathcal{O}(n^2)$ operation.

Chapter 4

Conclusion

If a system of equations comply with the initial conditions and the matrix A is a sparse matrix then conjugate gradient descent will be the most efficient method of finding the solution. If it is a dense matrix the matrix vector product will be costly and the more efficient way would be to use Newton Method. CGD always converges to the solution in at most n iterations, where n is the number of dimensions in the system of equations. Also with dense matrices there is the problem of round off error [4].

Appendix A

Supplementary Code

A.1 Steepest Descent

[6]:
```
import numpy as np
import matplotlib.pyplot as plt

def function1(x):
    return (x[0] - 2) ** 4 + (x[0] - 2 * x[1]) ** 2


def dfunction1(x):
    return np.array([4 * (x[0] - 2) ** 3 + 2 * (x[0] - 2*x[1]), -4 *␣
 ↪(x[0] - 2*x[1])])
```

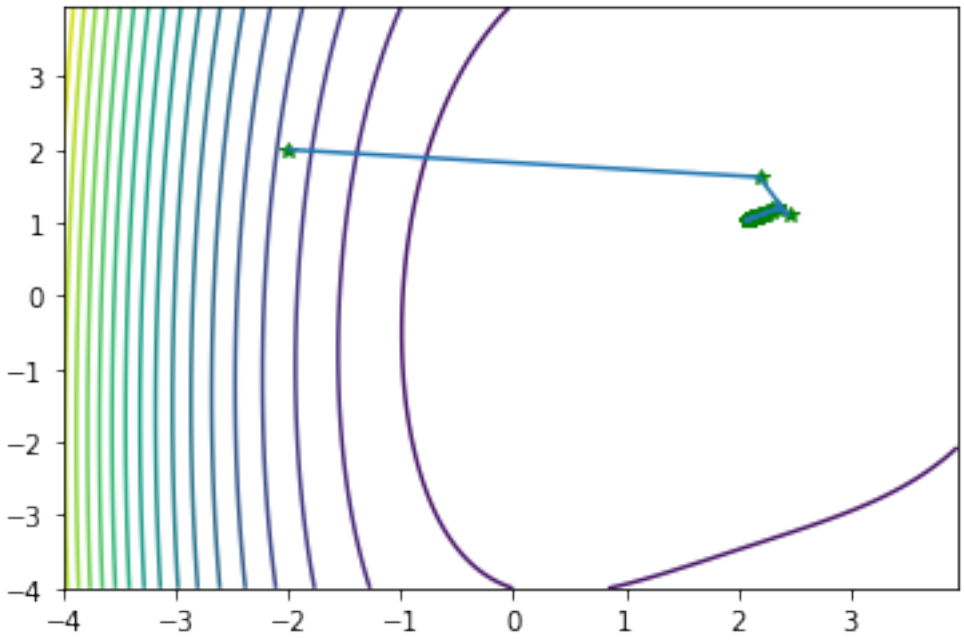
[7]:
```
import numpy as np


def goldsteinsearch(fun,dfun,d,x,alph,rho1,rho2,mu1,mu2):# stepsize␣
 ↪with Armijo-Goldstein,The principle is in the figure above
    flag =0

    fk = fun(x)
    gk = dfun(x)

    fff = fk
    dfff = np.dot(gk.T , d)

    while(flag == 0):
        fff1 = fun((x + alph * d))
        if(fff - fff1 <= (mu2 * (-alph) * dfff)):              # alphem␣
 ↪must not too big or too small
            if(fff - fff1 > (mu1 * (-alph) * dfff)):
```

```
                flag = 1                                                # if alpha=em is suitable,flag==1,Jump out of the loop
            else:
                alph = alph * rho1
        else:
            alph = alph * rho2
    return alph
```

[8]:
```
X1=np.arange(-4,4,0.05)
X2=np.arange(-4,4,0.05)
[x1,x2]=np.meshgrid(X1,X2)                                              # Generate grid point coordinate matrix, easy to draw
f=(x1 - 2) ** 4 + (x1 - 2 * x2) ** 2;                                   # Ready to draw a plot
plt.contour(x1,x2,f,20)                                                 # Draw the 20 contours of the function




def mainstep(x0,pre):
    imax = 100
    M=np.zeros((2,imax))
    M[:,0] = x0
    x = x0
    for i in range(1,imax):
        gard = dfunction1(x)
        p = - gard                                                      # fastest descending direction, negative Gradient
        p1 = np.linalg.norm(gard)
        if p1 < pre:                                                    # If the brane of the vector is smaller than the set value,
            break                                                       # we approximately assume that the minimum value has been reached.
        alph = goldsteinsearch(function1,dfunction1,p,x,1,0.5,2,0.2,0.8)
        x = x + alph * p                                                # New x
        M[:,i] = x                                                      # put each x in matrix M

    print("The number of iterations is:",i)
    print("The approximate optimal solution is:")
    print(x,'\n')
    return M

x0 = np.array([-2,2])
```

```
W=mainstep(x0,0.001)

plt.plot(W[0,:],W[1,:],'g*',W[0,:],W[1,:])          # Add the trajectory␣
 ↪of the algorithm to the graph
plt.show()
```

```
The number of iterations is: 99
The approximate optimal solution is:
[2.06995406 1.0353755 ]
```



We can see that the first few steps of iteration are fast, but the convergence rate is slower as we get closer to the optimal point.

The search direction of the two adjacent iterations of the fastest descent method is orthogonal(three-dimensional space)

The figure above is two-dimensional so it's not very accurate.

A.2 Conjugate Gradient Descent

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
def alph(r, d, A):
    """ Calculates learning rate """

    top = r.T @ r
    bottom = d.T @ A @ d

    return top/bottom

def bet(rk, rkp1):
    """ Constructs Gram-Schmidt constant
    from current and previous residuals """

    top = rkp1.T @ rkp1
    bottom = rk.T @ rk

    return top/bottom
```

[3]:
```
def CGM(A, b, x0, maxiter=1000, accuracy=0.00001):
    """ Utilises the Conjugate Gradient Method to obtain the minimum␣
↪point
    on a 2d-surface

    IN:  A   := numpy array, positive-definite 2 x 2 array
         b   := numpy array, scalar vector 2 x 1 array
         x0 := numpy array, initial guess vector 2 x 1 array
         accuracy := float, to what degree of accuracy should the␣
↪result be

    OUT: xpk1 := numpy array, position of minimum of 2d surface, 2 x␣
↪1 array
         path := numpy array, the path of theconjugate gradient method
         n    := int, amount of iterations to complete method

    """

    ## Checks that A is: symmetric, its eigenvalues are positive and␣
↪not complex
    if ~(np.all(np.linalg.eigvals(A) > 0) and ~np.all(np.iscomplex(np.
↪linalg.eigvals(A))) and np.all(A == A.T)):
```

```
        raise ValueError("A is not a positive definite symmetric matrix with real eigenvalues.")

    ## Initialises values for CGM algorithm
    rkp1 = b - A @ x0 # = r0
    dkp1 = rkp1 # = d0
    xkp1 = x0

    path = x0

    numiter = 0

    ## Begins CGM
    while np.abs(np.linalg.norm(rkp1)) > accuracy and numiter < maxiter:

        numiter += 1
        alphak = alph(rkp1,dkp1,A)

        xkp1 = xkp1 + alphak * dkp1
        rk = rkp1
        rkp1 = rkp1 - alphak * A @ dkp1

        betak = bet(rk,rkp1)

        dkp1 = rkp1 + betak * dkp1

        path = np.concatenate((path, xkp1), axis=1)


    return xkp1, path, numiter #x,y)
```

[4]:
```
def fx(A,b,x1i,x2i):
    """ Quadratic form of a surface equation: f(x) = 1/2 * x^T * A * x - b^T * x

        IN: A := numpy array, Coefficient matrix
            b := numpy array, known vector
            x1/x2 := int, boundaries of the plane

        OUT: z := int, the output of the quadratic equation.
    """

    x = np.array([[x1i],[x2i]])
```

```
    part_one= x.T @ A @ x
    part_two = b.T @ x

    z = 0.5 * part_one - part_two

    return z[0][0]
```

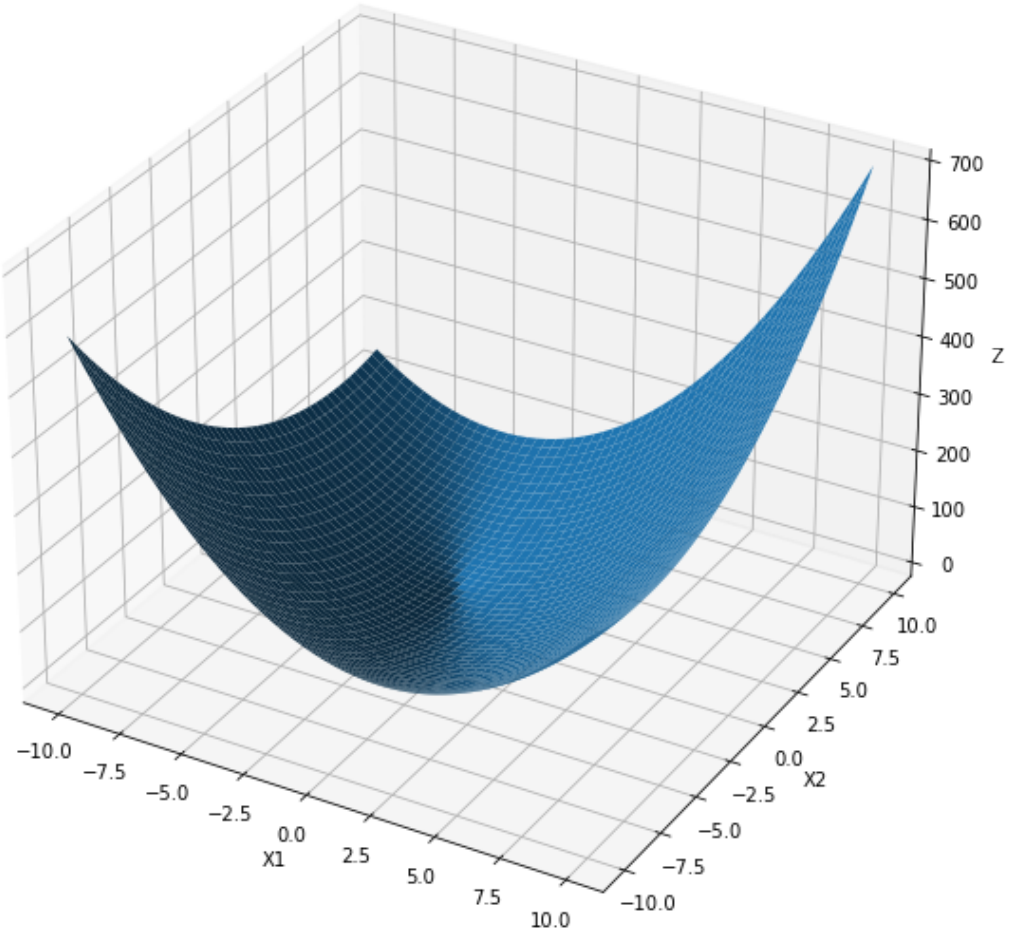
[5]:
```
def CGMconplot(A, b, x0, lims, spaces=0.05, accuracy=0.0001):
    """ Constructs contour plot from quadratic equation and
    plots path we move along for CGM.

        IN: A := numpy array, Coefficient matrix
            b := numpy array, known vector
            x0 := numpy array, initial position
            lims := int, limits of the contour plot
            spaces := float, the step size along the axis for the␣
 ↪contour plot
            accuracy := float, the accuracy the user would like for␣
 ↪the CGM

        OUT: Contour plot with path for user
    """

    x1 = np.arange(-lims, lims, spaces)
    X1, X2 = np.meshgrid(x1, x1)

    ## Calculates f(x) for each point on the surface
    Z = a = np.zeros(shape=(x1.size,x1.size))
    for i in range(x1.size):
        for j in range(x1.size):
            val = fx(A,b,x1[i],x1[j])
            Z[i][j] = val

    zs = Z.reshape(X1.shape)

    try:
        xkp1, path, n = CGM(A, b, x0)
    except ValueError as err:
        return err

    plt.contour(X2, X1, zs)
    plt.plot(path[0], path[1], 'k*-', label='Path')
    plt.plot(path[0][0], path[1][0], 'bd', markersize=10, label='Start')
    plt.plot(xkp1[0], xkp1[1], 'rs', markersize=7, label='End')
    plt.legend()
```

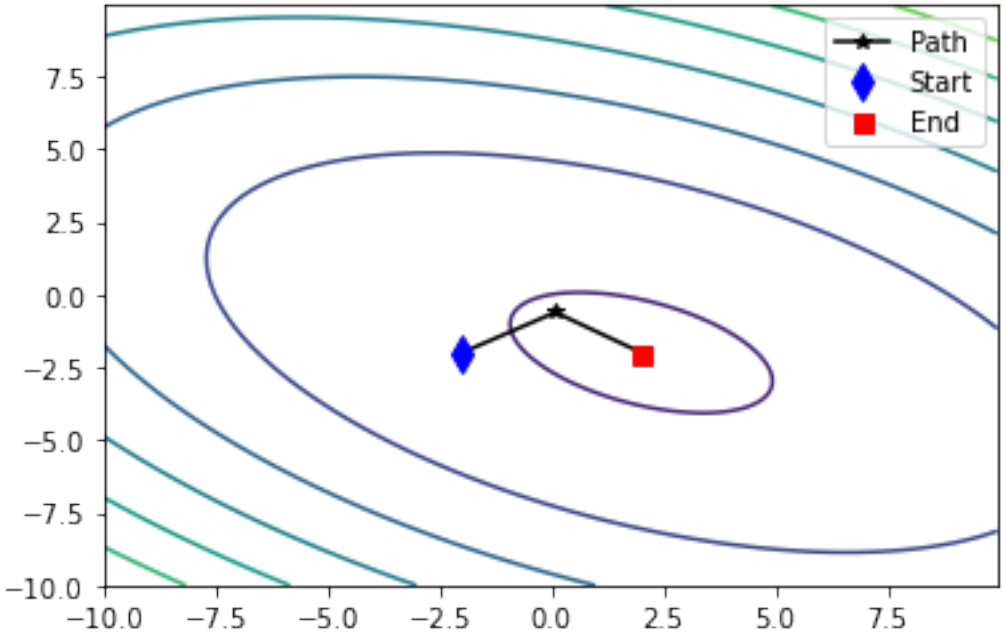
```
    plt.show()

    print("""Number of iterations: {0}
    Path:   x := {1}
            y := {2}
    Minimum point: ({3}, {4})""".format(n, path[0], path[1],␣
 ↪xkp1[0][0], xkp1[1][0]))
```

[6]:
```
def SurfacePlot(A, b, lims=10, spaces=0.05):
    """ Constructs surface plot from quadratic equation.

        IN: A := numpy array, Coefficient matrix
            b := numpy array, known vector
            lims := int, limits of the contour plot
            spaces := float, the step size along the axis for the␣
 ↪contour plot

        OUT: Surface plot for user.
    """
    x1 = x2 =  np.arange(-lims, lims, spaces)
    X1, X2 = np.meshgrid(x1, x2)

    Z = a = np.zeros(shape=(x1.size,x2.size))

    ## Calculates f(x) for each point on the surface

    for i in range(x1.size):
        for j in range(x2.size):
            val = fx(A,b,x1[i],x2[j])
            Z[i][j] = val

    zs = Z.reshape(X1.shape)

    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X1, X2, zs)
    ax.set_xlabel('X1')
    ax.set_ylabel('X2')
    ax.set_zlabel('Z')
    ax.view_init(30)
    plt.show()
```

[7]:
```
## Using values from 'https://www.cs.cmu.edu/~quake-papers/
 ↪painless-conjugate-gradient.pdf'
##  Expect minimum to be at (2, -2)
```

```
A = np.array([[3,2],[2,6]])
b = np.array([[2],[-8]])
x0 = np.array([[-9],[5]])

SurfacePlot(A, b, lims=10, spaces=0.05)
CGMconplot(A, b, x0, 10, spaces=0.05)
```






```
Number of iterations: 2
    Path:    x := [-9.          -1.63423332  2.        ]
             y := [ 5.          -2.75343861 -2.        ]
    Minimum point: (2.0, -2.0)
```

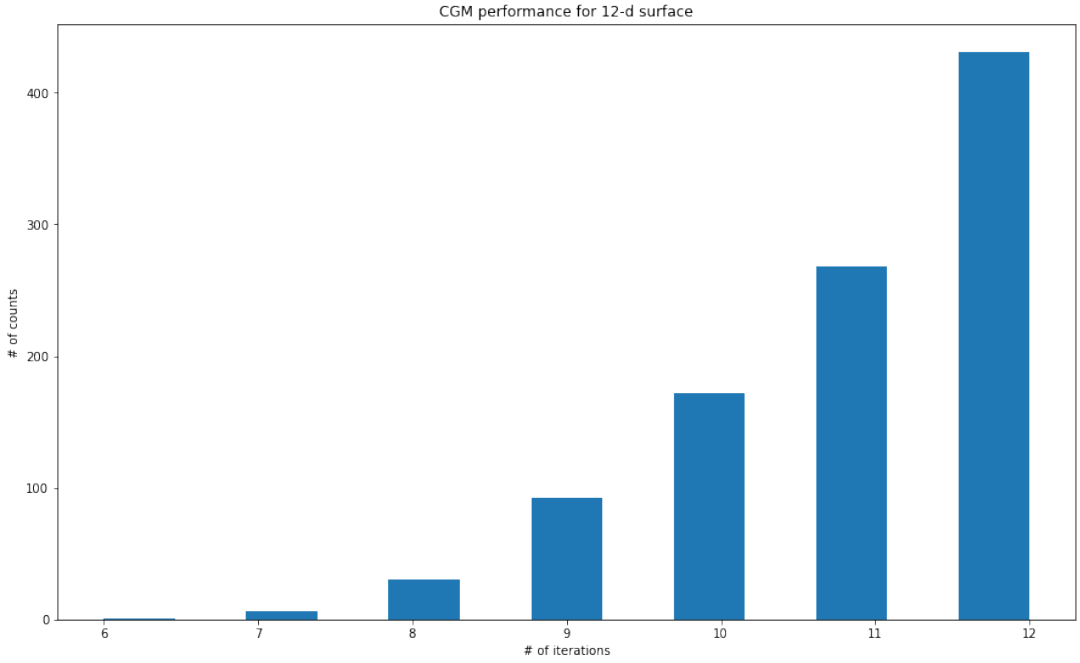
Let's have a closer look at higher dimensions. Modifying our above function, we can obtain;

[8]:
```
def CGM_nd(A, b, x0, maxiter=1000, accuracy=0.00001):
    """ Utilises the Conjugate Gradient Method to obtain the minimum␣
↪point
    on an n-d surface

    IN:  A  := numpy array, positive-definite n x n array
         b  := numpy array, scalar vector n x 1 array
         x0 := numpy array, initial guess vector n x 1 array
         accuracy := float, to what degree of accuracy should the␣
↪result be

    OUT: xpk1 := numpy array, position of minimum of n-d surface, n x␣
↪1 array
         n := int, number of iterations

    """
    if ~(np.all(np.linalg.eigvals(A) > 0) and ~np.all(np.iscomplex(np.
↪linalg.eigvals(A))) and np.all(A == A.T)):
```

```
        raise ValueError("A is not a positive definite symmetric␣
↪matrix with real eigenvalues.")

    ## Initialises values for CGM algorithm
    rkp1 = b - A @ x0 # = r0
    dkp1 = rkp1 # = d0
    xkp1 = x0

    numiter = 0
    while np.abs(np.linalg.norm(rkp1)) > accuracy and numiter <␣
↪maxiter:

        numiter += 1
        alphak = alph(rkp1,dkp1,A)

        xkp1 = xkp1 + alphak * dkp1
        rk = rkp1
        rkp1 = rkp1 - alphak * A @ dkp1

        betak = bet(rk,rkp1)

        dkp1 = rkp1 + betak * dkp1



    return xkp1, numiter
```

[9]:
```
## Check performance of CGM on a 12-d surface.
m = 12
N = 1000
freq = np.zeros(N)
for i in range(N):
    A = np.diag(np.random.rand(m))
    b = np.random.rand(m)
    x0 = np.random.rand(m)
    x, n = CGM_nd(A, b, x0)

    freq[i] = n

plt.figure(1, figsize=(15,9))
plt.hist(freq, bins=m+1)
plt.title("CGM performance for {0}-d surface".format(m))
plt.ylabel("# of counts")
plt.xlabel("# of iterations")
```

```
plt.show()
```



CGM needs at most m amount of steps to find the minimum point on an m-d surface.

Bibliography

[1] M. R. Hestenes, E. Stiefel, *et al.*, *Methods of conjugate gradients for solving linear systems*, 1. NBS Washington, DC, 1952, vol. 49.

[2] T. Terlaky, *SW/CS 4/6TE3 — Continuous Optimization Algorithms*. [Online]. Available: http://www.cas.mcmaster.ca/%7B~%7Dcs4te3/notes/ (visited on 11/28/2021).

[3] J. R. Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain", 1994.

[4] Stanford, *Conjugate Gradient Descent*. [Online]. Available: https://stanford.edu/class/ee364b/lectures/conj_grad_slides.pdf (visited on 11/28/2021).