Constrained Optimization demystified, with implementation in Python.

Designing a new robust constrained optimization algorithm.

Aakash Agrawal Feb 1, 2020 · 10 min read ★

src

Nonlinear constrained optimization problems are an important class of problems with a broad range of engineering, and scientific applications. In this article, we will see how the refashioning of simple unconstrained Optimization techniques leads to a hybrid algorithm for constrained optimization problems. Later, we will observe the robustness of the algorithm through a detailed analysis of a problem set and monitor the performance of optima by comparing the results with some of the inbuilt functions in python.

Keywords— Constrained-Optimization, multi-variable optimization, single variable optimization.

Many engineering design and decision making problems have an objective of optimizing a function and simultaneously have a requirement for satisfying some constraints arising due to space, strength, or stability considerations. So, **Constrained optimizatio**

refers to the process of optimizing an objective function with re to some variables in the presence of constraint of those vari

A constrained optimization problem with N variables is given by:

$$\text{Minimize:} \quad f(x)$$

$$\text{Subject to:} \quad g_j(x) \geq 0, \;\; j = 1, ..., J;$$

$$h_k(x) = 0, \;\; k = 1, ..., K;$$

$$x_i^{(L)} \leq x_i < x_i^{(U)}, \;\; i = 1, ..., N;$$

-where ***gⱼ(x)*** are the J inequality constraints, ***hₖ(x)*** are the K equality constraints, ***f(x)*** is the objective function to be optimized. Let us understand some of the frequently used terminologies in optimization.

THEORY

In the parlance of mathematical optimization, there are two routes by which one can find the optimum(Numerically):

1. Using Direct Search method Here, we only use the function values at a given point to find the optimum. It works by comparing the function values in a neighborhood of a point and subsequently moving in the direction that leads to a decrease in the function value (for minimization problems). Direct Search methods are typically used when the function is **discontinuous** and hence the derivate is not available at that point.

2. Using gradient-based method: Here, we use the first and second-order derivatives to locate the optima. These methods take the gradient information into account and thus have the advantage of converging **faster** to the optima.

How to find the derivative at a particular point numerically? We use the central difference method, which is mathematically given as {limit h ->0}:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Our proposed algorithm for constraint optimization hires two single variable optimization methods and one multi-variable optimization method. Our main intention is to convert this multivariable constraint optimization problem into an unconstraint multi-variable optimization problem, and this unconstraint problem then can be solved using the single variable optimization methods.

Single Variable Optimization

Again, there are two routes for finding the optimum of a linear or non-linear function of a single variable, one is using direct search methods, and the other is through the gradient-based techniques. One can find the optima using solely either of the approaches. Our algorithm for constraint optimization uses both approaches. Using the direct search method, we will bracket the optima, and once we have a particular bound for the optima, we can find the exact optima using the gradient-based method (for single variable function).

There are many direct search and gradient-based methods for obtaining the optimum of a single variable function. Our method uses the Bounding Phase Method and the Secant method.

Note: all the optimization methods described are iterative. We converge to the optimum value gradually after a series of iterations.

Bounding Phase MethodA direct search method to find lower and upper bounds for the minima in a single variable unconstrained optimization. The Algorithm is given as (*f'* refers to the 1st order derivative at a point):

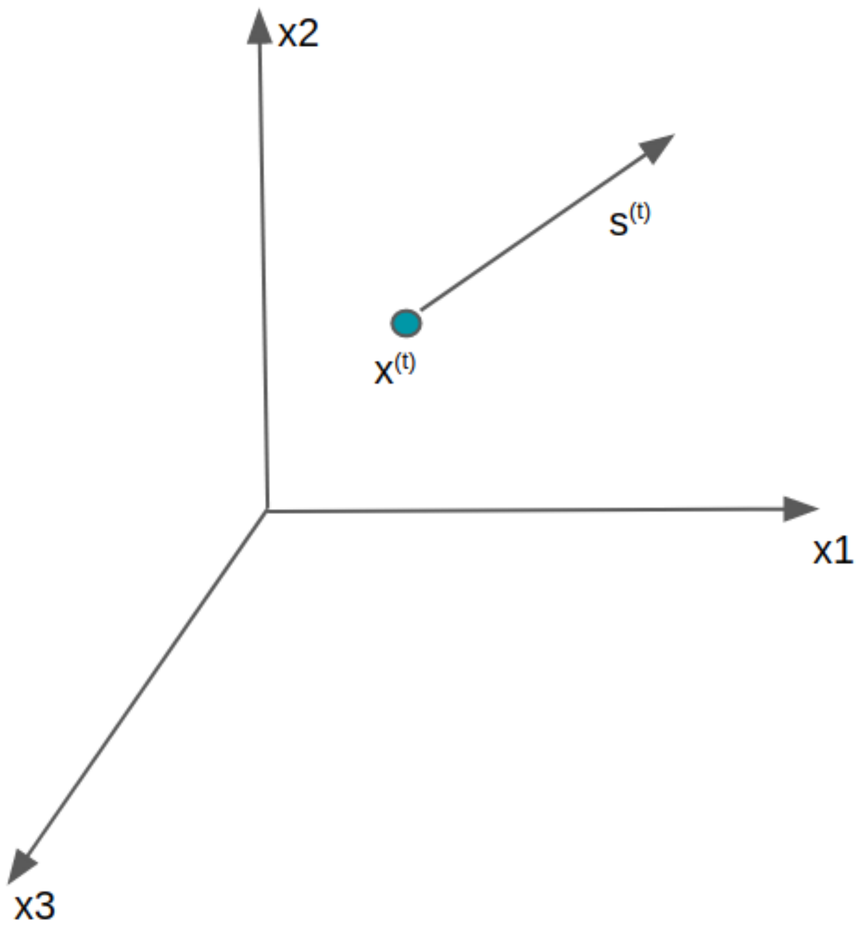
1. Choose initial guess **$x^{(0)}$,** an increment **$\Delta(\sim 0$**, set iteration counter **$k=0$**
2. If **$f(x^{(0)} - \Delta) > f(x^{(0)} + \Delta$**, then Δ is **positive** Else if, **$f(x^{(0)} - \Delta) <$ $f(x^{(0)} + \Delta$**, then, Δ is **negative** Else go to **step_1**
3. Set **$x^{(k+1)} = x^{(k)} + 2^k\Delta$.** (exponential perturbation).
4. If **$f(x^{(k+1)}) < f(x^{(k}$**, set **$k = k+1$** and go to **step_3** Else, the minima lies in **$(x^{(k-1)}, x^{(k+1)})$** and **terminat**

Secant MethodA very popular gradient-based method for a single variable optimization. The termination condition is when the gradient of a function is very small (~0) at a point. The method is as follows:

1. Choose two points **a, b** such that **f'(a)** and **f'(b)** have opposite signs. In other words, **$f'(a).f'(b) < 0$** Choose **ε{epsilon}** a small number, aka the termination factor. Set **$x_1 = a$** and **$x_2 = b$**
2. Compute a new point **$z = x_2 - (f'(x_2)*(x_2-x_1))/(f'(x_2)-f'($** and find **f'(z)**.
3. If **|f'(z)|** ≤, **Terminate.**
   Else if **$f'(z) < 0$** Set **$x_1 = z$** and go to **step_2**,
   Else if **$f'(z) \geq 0$**, Set **$x_2 = z$** and go to **step_2**

Other popular gradient-based methods for single variable optimization are the Bisection method, Newton-Rapson method, etc.

Unidirectional Search: Here, the goal is to find where the function value will be minimum in a particular direction. Mathematically, we need to find scalar **α**(alpha) such that, **$f(\alpha) = f(x^{(t)}+\alpha.s^{($** is minimized, which is achieved using the single variable optimization methods. {$s^{(t)}$ = search direction}.



Note: Many multi-variable optimization techniques are nothing but successive unidirectional search, to find the minimum point along a particular direction.

Variable Metric Method(Davidon–Fletcher–Powell Method):

The DFP method is a gradient-based **multi-variable optimizati** algorithm. It results in a faster convergence to the optima by not taking into account the **hessian**for creating a search direction, thereby overcoming the limitations of several other multi-variable optimization algorithms. The search direction is given by:

$$s^{(k)} = -A^{(k)}\nabla f(x^{(k)}) \quad ..(1)$$

where the matrix A is given by:

$$A^{(k)} = A^{(k-1)} + \frac{\Delta x^{(k-1)}\Delta x^{(k-1)T}}{\Delta x^{(k-1)T}\Delta e(x^{(k-1)})} - \frac{A^{(k-1)}\Delta e(x^{(k-1)})^T}{\Delta e(x^{(k-1)})^T A^{(k-1)}\Delta e(x^{(k-1)})}$$
$$\Delta x^{(k-1)} = x^{(k)} - x^{(k-1)}$$
$$\Delta e(x^{(k-1)}) = e(x^{(k)}) - e(x^{(k-1)}) \quad ..(2)$$

$e(x^{(k)})$ represents the gradient of the function at a point *$x^{(k)}$*. The uni-direction search involves the secant method and the bounding phase method to find the value of **α** in the search space. The new point obtained after the unidirectional search is :

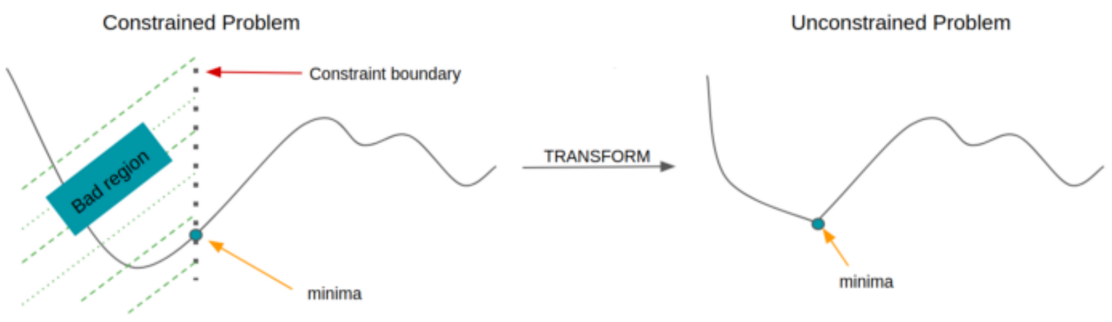
$$x^{(k+1)} = x^{(k)} + \alpha^{(k)}s^{(k)} \quad ..(3)$$

1. Choose initial guess $\mathbf{x^{(0)}}$ and termination parameters $\boldsymbol{\varepsilon_1, \varepsilon_2}$. (Note, here $\mathbf{x^{(0)}}$ is a vector).
2. Find ∇ $\mathbf{f(x^{(0)})}$\{gradient of ***f*** at $\mathbf{x^{(0)}}$\} and set $\mathbf{s^{(0)} = -\nabla f(x^{(0)})}$ \{initial search direction\}.
3. Find **α**(alpha) such that $\mathbf{f(x^{(0)} + \alpha.s^{(0)}}$is minimum with the termination parameter $\boldsymbol{\varepsilon_1}$. Denote it by **α*.** Set $\mathbf{x^{(1)} = x^{(0)} + \alpha^* s^{(}}$ and k=1. Calculate ∇ $\mathbf{f(x^{(1)})}$ \{f($x^{(0)}$ + α.$s^{(0)}$) is a function of **α,** and we will find this **α*** by single variable optimization methods\}.

4. Find $\mathbf{s}^{(k)}$ using the formula in *eq(1) and eq(2).*
5. Find $\lambda^{(k)}$ such that $\mathbf{f(x^{(k)} + \lambda^{(k)}.s^{(k)}}$is minimum with terminating factor ε_1. Set, $\mathbf{x^{(k+1)} = x^{(k)} + \lambda^{(k)}.s}$
6. Check for the termination conditions. Is $||\mathbf{x^{(k+1)} - x^{(k)}}||/||\mathbf{x^{(k)}}|| \leq$ {|| . || denotes the norm of a vector}.
7. If Yes, **TERMINATE;** Else set k = k+1 and go to **step_4.**

Penalty Function Method:

A **penalty**(regularizer) is an additional term we add to our objective function, which helps in controlling the excessive fluctuation of that objective function. By adding these penalty terms, we transform our constrained problem to an unconstrained problem structured such that minimization favors satisfaction of the constraints, as shown in the figure below.



Simply put, the technique is to add a term to the objective function such that it produces a high cost for violation of constraints. This is known as the **Penalty function metho** Mathematically,

$$P(x, R) = f(x) + \Omega(R, g(x), h(x)) \quad ..(4)$$

$$\Omega = R < g(x) >^2 \quad ..(5)$$

where R is a penalty parameter, P(x, R) is the penalty function, and Ω is the penalty term. There are various types of penalty terms depending upon the feasibility and type of constraint. This one is known as the **bracket operator penalt**term. where,

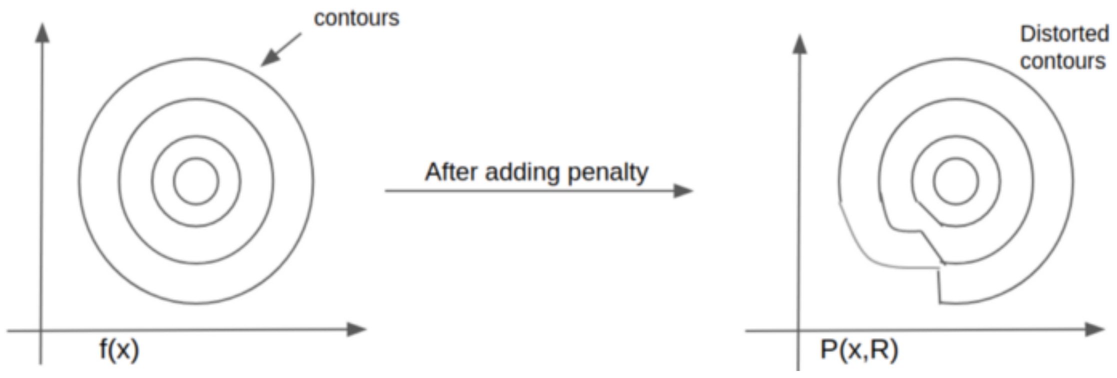
$$< \alpha > = \begin{cases} \alpha & \text{if } \alpha < 0 \\ 0 & \text{if } \alpha >= 0. \end{cases}$$

The method is given as:

1. Choose initial solution $\mathbf{x^{(0)}}$ and termination parameters $\boldsymbol{\varepsilon_1}$, $\boldsymbol{\varepsilon_2}$, penalty parameter $\mathbf{R^{(0)}}$, and an update factor **c**.
2. Form the penalty function $\mathbf{P(x^{(t)},R^{(t)}) = f(x^{(t)}) + \Omega(R^{(t)}, g(x^{(}}$ $\mathbf{h(x^{(t)})}$]
3. Use the **DFP** method to find the minimum of $\mathbf{P(x^{(t)}, R^{(t)})}$.Let the solution be $\mathbf{x^{(t+1)}}$. {This particular step is an unconstraint optimization problem}. For DFP, our initial guess will be $\mathbf{x^{(t)}}$.
4. Check the termination condition:
   Is $\mathbf{|P(x^{(t+1)}, R^{(t)}) - P(x^{(t)}, R^{(t-1)}} \leq \boldsymbol{\varepsilon_2}$? {|.| is the mod function}.
   If yes, Set $\mathbf{x^T = x^{(t+1)}}$and **Terminate**
   Else go to **step_5.**
5. Update $\mathbf{R^{(t+1)} = c{*}R^{(t)}}$. Set, t=t+1 and go to **step_2**

Note: all the optimization methods described are iterative. We converge to the optimum value gradually after a series of iterations. We update this R-value at each iteration.

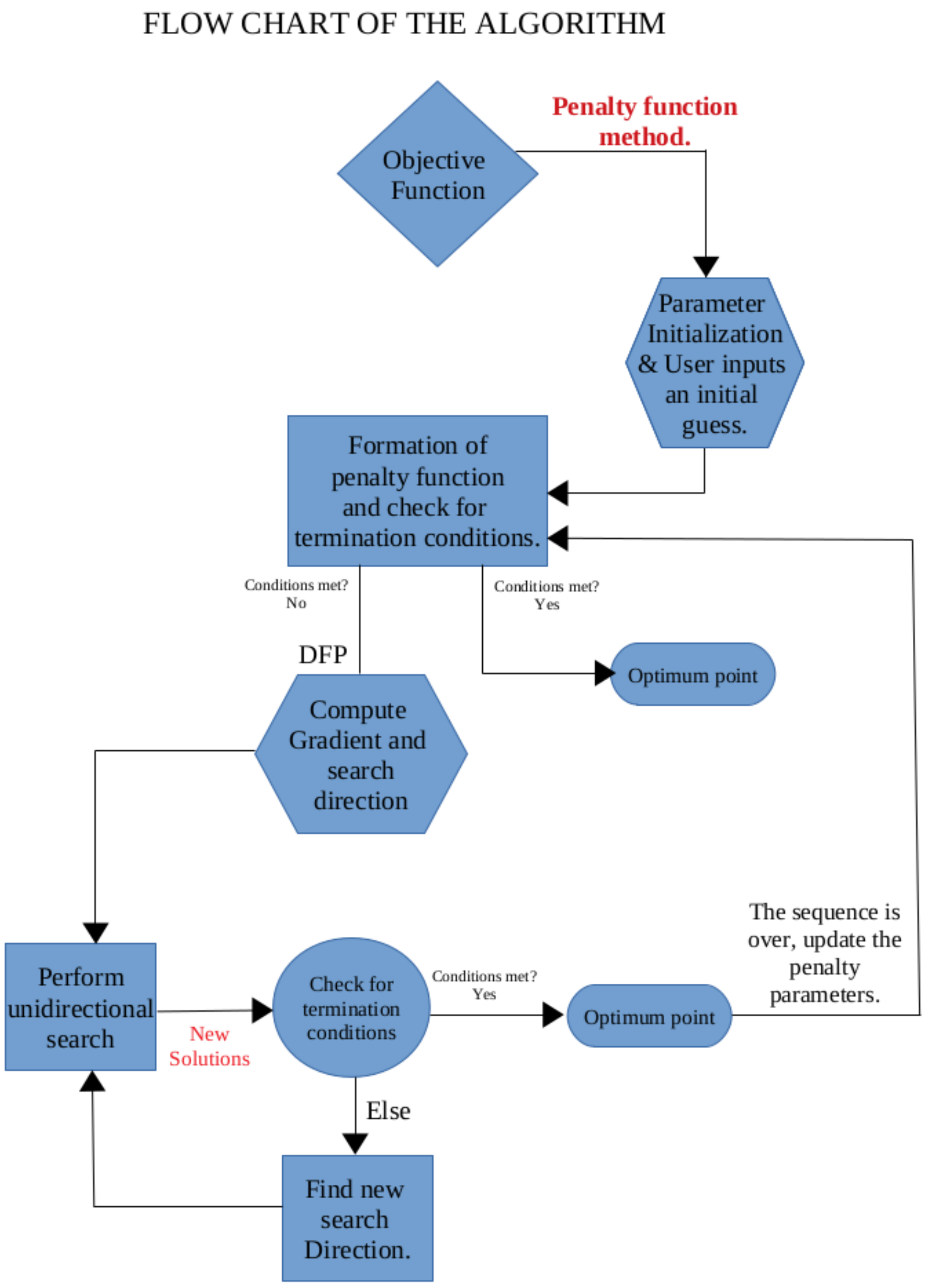
There are several limitations to using the penalty function method. Firstly, It results in a distortion of the contours, due to which the algorithm takes a greater time to converge. Also, this results in the presence of artificial local optimas.



For the purpose of implementation, we will only stick to the penalty function method. There is another method known as the **Method of Multiplier** used to overcome the limitations of distortion. It is basically a slight modification of the Penalty function method. This method does not distort the contours but instead has an effect of shifting the contours towards the constraint optimum point. So, here the presence of artificial local optima will be zero.

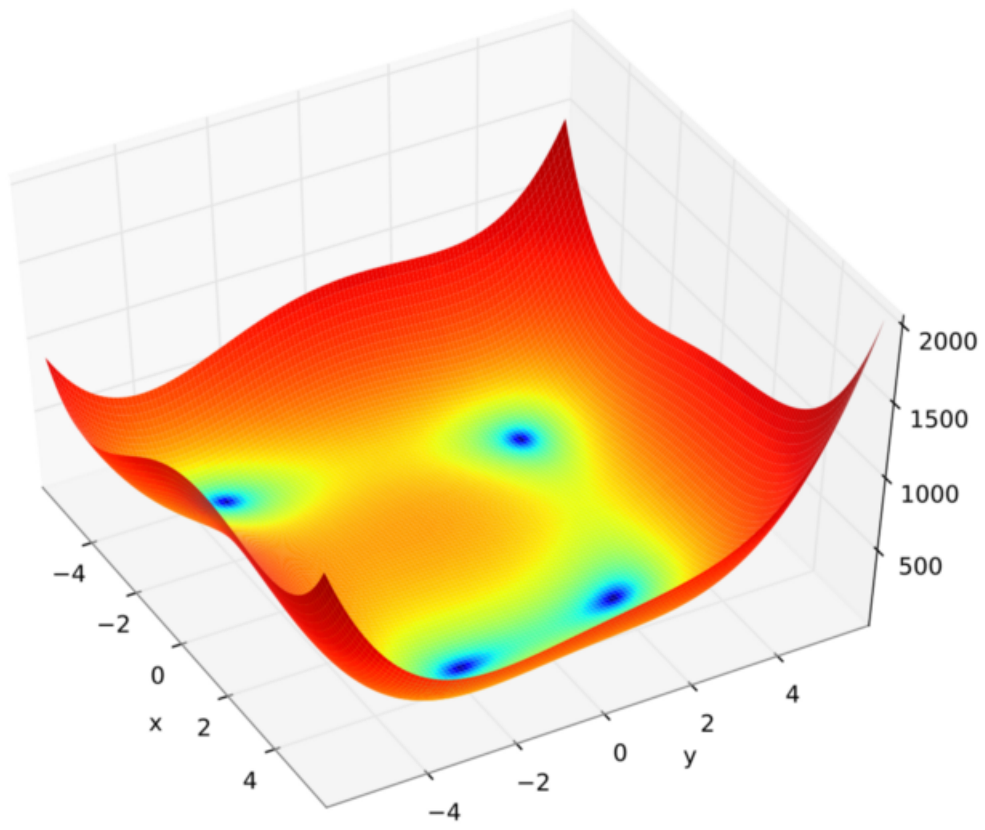
The **method of multipli**and **penalty function meth**both will convert a constrained optimization problem to an unconstrained problem, that further can be solved by any multi-variable optimization method.

Well, that's it!!! If you have come this far, great! Now, let us have a look at the flow chart of our method and then go for the implementation. For the sake of implementation, we will only cover the penalty function method.


FLOW CHART OF THE ALGORITHM
Objective Function
Penalty function method.
Parameter Initialization & User inputs an initial guess.
Formation of penalty function and check for termination conditions.
Conditions met? No
Conditions met? Yes
Optimum point
DFP
Compute Gradient and search direction
Perform unidirectional search
New Solutions
Check for termination conditions
Conditions met? Yes
Optimum point
The sequence is over, update the penalty parameters.
Else
Find new search Direction.

The Himmelblau Function:

For the purpose of illustration of our method, we will work with the famous Himmelblau function(see fig.), given as $f(x, y)=(x^2+y-11)^2 +(x+y^2-7)^2$.



src

Let’s solve the following constraint optimization problem using our proposed algorithm.

$$\text{Minimize} \quad (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

subject to

$$(x_1 - 5)^2 + x_2^2 - 26 \geq 0, \qquad x_1, x_2 \geq 0.$$

IMPLEMENTATION

DFP method is used for multi-variable optimization, and a combination of the **bounding phase** and the **secant method** is used to obtain a uni-directional search. The code is implemented in python and hosted in my GitHub repo. We now briefly demonstrate each of the functions used:

multi_f: This function takes an **input vector x** (a point in search space) and returns the function value (penalized function value) at that point.

```
import numpy as np

eps = 0.000001
delta = 0.001
nc = 1 ## number of constraints.
g = np.zeros(nc)

def multi_f(x):
        ## Himmel Blau function!
        sum_ = pow((pow(x[0],2) + x[1] - 11),2) + pow((pow(x[1],2) + x[0] -
        g[0] = -26.0 + pow((x[0]-5.0), 2) + pow(x[1],2);#constraints.

        for i in range(nc):
                if(g[i] < 0.0): ## meaning that the constraint is violatd.
                        sum_ = sum_ + r*g[i]*g[i];
```

grad_multi_f: This function uses the **central difference** method to calculate the gradient vector at a particular point in search space.

```
def grad_multi_f(grad, x_ip):
        d1 = np.zeros(M);
        d2 = np.zeros(M);
        delta_=0.001;

        for i in range(M):
                for j in range(M):
                        d1[j]=x_ip[j]; d2[j]=x_ip[j];

                d1[i] = d1[i] + delta_;
                d2[i] = d2[i] - delta_;

                fdiff = multi_f(d1); bdiff = multi_f(d2);
                grad[i] = (fdiff - bdiff)/(2*delta_);
```

bracketing_: This function implements the bounding phase method used to bracket the **α***(minima obtained by performing the uni-directional search). It takes a vector **x** and vector **s**(search direction) and outputs an interval on the basis of which **α** can be evaluated.

```
## BOUNDING PHASE METHOD,, TO BRACKET THE ALPHASTAR.
def bracketing_(x_0, s_0):
        k=0;

        ## Step__1
        ## "choose an initial guess and increment(delta) for the Bounding ph
        del_=0.0000001; w_0=0.5;#initial guess for bounding phase.

        while(1):
                f0 = uni_search(w_0, x_0, s_0, 0.0, 1.0);
                fp = uni_search(w_0+del_, x_0, s_0, 0.0, 1.0);
                fn = uni_search(w_0-del_, x_0, s_0, 0.0, 1.0);

                if(fn >= f0):
                        if(f0 >= fp):
                                del_ = abs(del_); ## step__2
                                break;

                        else:
                                a=w_0-del_;
                                b=w_0+del_;

                elif((fn <= f0) & (f0 <= fp)):
                        del_ = -1*abs(del_);
                        break;

                else:
                        del_ /= 2.0;

        wn = w_0 - del_;## wn = lower limit and w_1 = upper limit.

        ## Step__3
        w_1=w_0 + del_*pow(2,k); ## << exponential purterbation
        f1=uni_search(w_1, x_0, s_0, 0.0, 1.0);

        ## Step__4
        while(f1 < f0):
                k+=1;
                wn=w_0;
                fn=f0;
                w_0=w_1;
                f0=f1;
                w_1=w_0 + del_*pow(2,k);
                f1=uni_search(w_1, x_0, s_0, 0.0, 1.0);
```

f _dash: This function is used to get the first-order differential for a single variable function using the central difference method. (represents ***f'***).

```
def f_dash(x_0, s_0, xm):
        xd = np.zeros(M);

        # using central difference.
        for i in range(M):
                xd[i] = x_0[i] + (xm+delta)*s_0[i];
                fdif = multi_f(xd);

        for i in range(M):
                xd[i] = x_0[i] + (xm-delta)*s_0[i];
                bdif = multi_f(xd);

        f_ = (fdif - bdif)/(2*delta);
```

secant_minima: This function takes the bounds found from the bounding phase method, a point **x**, and a search direction **s** as input and evaluates the alphastar.

compute_z: This function is used to compute the formula used in the secant method:

$$z = x_2 - f'(x_2)\frac{x_2 - x_1}{f'(x_2) - f'(x_1)}$$

```
## This is a function to compute the "z" used in the formula of SECANT Metho
def compute_z(x_0, s_0, x1, x2):
        z_ = x2 - ((x2-x1)*f_dash(x_0, s_0, x2))/(f_dash(x_0, s_0, x2)-f_das
        return(z_)

def secant_minima(a, b, x_0, s_0):
        ## Step__1
        x1=a;
        x2=b;

        ##Step__2 --> Compute New point.
        z = compute_z(x_0, s_0, x1, x2);

        ##Step__3
        while(abs(f_dash(x_0, s_0, z)) > eps):
                if(f_dash(x_0, s_0, z) >= 0):
                        x2=z; # i.e. eliminate(z, x2)
                else:
                        x1=z; # i.e. elimintae(x1, z)
                z = compute_z(x_0, s_0, x1, x2); #Step__2
```

DFP: It takes only the input vector ***x*** as an argument and returns the solution vector. This function is called inside the *main* function until the termination conditions for the penalty function method are met.

```python
def DFP(x_ip):
        eps1 = 0.0001;
        eps2 = 0.0001;

        grad0=np.zeros(M);
        grad1=np.zeros(M);
        xdiff=np.zeros(M);
        Ae=np.zeros(M);
        graddiff=np.zeros(M);
        x_1=np.zeros(M);
        x_2=np.zeros(M);

        ## step_1
        k=0;
        x_0=x_ip #This is the initial Guess_.

        ## step_2
        s_0=np.zeros(M); ##Search Direction.
        grad0 = grad_multi_f(grad0, x_ip);
        s_0 = -grad0

        ## step_3 --> unidirectional search along s_0 from x_0. To find alph
        ## find alphastar such that f(x + alphastar*s) is minimum.
        a,b = bracketing_(x_0, s_0);
        alphastar = secant_minima(a, b, x_0, s_0);

        for i in range(M):
                x_1[i] = x_0[i] + alphastar*s_0[i];

        grad1 = grad_multi_f(grad1, x_1); # Now computing grad(x(1))..

        ## step_4
        ## Lets initialize the matrix A,
        A = np.eye(M) #Identity Matrix.
        dA = np.zeros((M,M))
        A1 = np.zeros((M,M))

        for j in range(50): # 50 is the total iterations we will perform.
                k+=1;
                #print("Iteration Number -- ", k)
                for i in range(M):
                        xdiff[i]=x_1[i]-x_0[i];
                        graddiff[i]= grad1[i]-grad0[i];

```

RESULTS

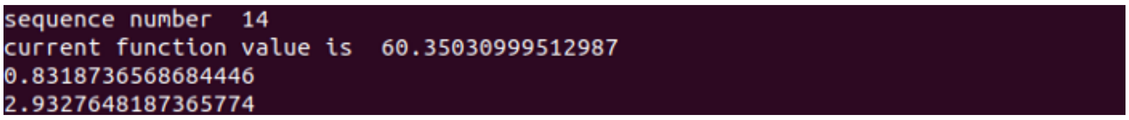
[illegible]

Optimization Data Science Mathematics Programming Machine Learning

generalized and applicable to any number of dimensions one wants to work on.
The parameter setting for our algorithm is:
* **M=2**{specifies the total dimensions we are working with},
* **R=0.1**{panalty parameter} ,
* **c=1.55**{factor for updating R},
* **x_ip** (initial guess)=**(0.11, 0.1)**

About · Write · Help Legal

I suggest the reader to try using different initial guesses and play with these parameter values. So, our algorithm converged after **14 sequences**And we get the optimum solution to the constrained optimization problem.



```
sequence number  14
current function value is  60.35030999512987
0.8318736568684446
2.9327648187365774
```

Let's compare our results with those been found from the ***optimize*** module of the ***scipy*** library in ***python***.{Working with the same initial guess}: