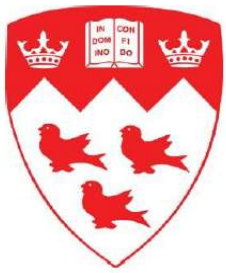
McGill UNIVERSITY

845 Sherbrooke St W, Montreal,

QC H3A 0G4

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

OPTIMIZATION PROJECT



Course - OPTIMIZATION AND OPTIMAL CONTROL

Course code- ECSE-507

Winter 2019

Instructor - Prof. Hannah Michalska

Submitted by-

Surya Kumar Devarajan – 260815492

Date: 26-04-2019

Abstract

Optimization is an area of interest and research in this fast-paced world. Optimization is the act of finding minimum or maximum values of a function. An important distinction is between problems in which there are no constraints on the variables and problems in which there are constraints on the variables. Unconstrained optimization problems arise directly in many practical applications; they also arise in the reformulation of constrained optimization problems in which the constraints are replaced by a penalty term in the objective function. Constrained optimization problems arise from applications in which there are explicit constraints on the variables.The constraints on the variables can vary widely from simple bounds to systems of equalities and in-equalities that model complex relationships among the variables.Today, the results of unconstrained and constrained optimization are applied in different branches of science, as well as generally in practice. Here, we present several unconstrained algorithms which minimise the functions and with line search techniques. Further, in this chapter we consider some constrained optimization methods as well. We try to present these methods but also to present some contemporary results in this area.

Keywords: unconstrained optimization, line search, steepest descent method, Armijo step size rule selection, conjugate gradient, the secant algorithm, finite difference approximations of the gradient, the penalty and barrier function, the augmented Lagrangian, the Lagrange-Newton method

1 Introduction

Optimization is the act of achieving the best possible result under given circumstances. In design, construction, maintenance, engineers have to take decisions. The goal of all such decisions is either to minimize effort or to maximize benefit. The effort or the benefit can be usually expressed as a function of certain design variables. Hence, optimization is the process of finding the conditions that give the maximum or the minimum value of a function [10]. It is obvious that if a point x^* corresponds to the minimum value of a function $f(x)$, the same point corresponds to the maximum value of the function $-f(x)$. Thus, optimization can be taken to be minimization. There is no single method available for solving all optimization problems efficiently. Hence, a number of methods have been developed for solving different types of problems.

Today, there exist many modern optimization methods which are made to solve a variety of optimization problems. Now, they present the necessary tool for solving problems in diverse fields. At the beginning, it is necessary to define an objective function, which, for example, could be a technical expense, profit or purity of materials, time, potential energy, etc. The object function depends on certain characteristics of the system, which are known as variables. The goal is to find the values of those variables, for which the object function reaches its best value, which we call an extremum or an optimum. It can happen that those variables are chosen in such a way that they satisfy certain conditions, i.e., restrictions. The process of identifying the object function, variables, and restrictions for the given problem is called modeling. The first and the most important step in an

optimization process is the construction of the appropriate model, and this step can be the problem by itself. Namely, in the case that the model is too much simplified, it cannot be a faithful reflection of the practical problem. By the other side, if the constructed model is too complicated, then solving the problem is also too complicated. After the construction of the appropriate model, it is necessary to apply the appropriate algorithm to solve the problem. It is no need to emphasize that there does not exist a universal algorithm for solving the set problem [5] [2].

An important distinction is between problems in which there are no constraints on the variables and problems in which there are constraints on the variables. Unconstrained optimization problems arise directly in many practical applications; they also arise in the reformulation of constrained optimization problems in which the constraints are replaced by a penalty term in the objective function. Constrained optimization problems arise from applications in which there are explicit constraints on the variables. The constraints on the variables can vary widely from simple bounds to systems of equalities and inequalities that model complex relationships among the variables. Constrained optimization problems can be furthered classified according to the nature of the constraints (e.g., linear, nonlinear, convex) and the smoothness of the functions (e.g., differentiable or non-differentiable) [3].

After the construction of the appropriate model, it is necessary to apply the appropriate algorithm to solve the problem. It is no need to emphasize that there does not exist a universal algorithm for solving the set problem. We are following steepest descent method, Armijo step size rule selection, conjugate gradient, secant algorithm, the penalty and barrier function, the augmented Lagrangian, the Lagrange-Newton algorithms to solve constrained and unconstrained problems. Matlab is used [6].

2 Statement of Optimization

An optimization, or a mathematical programming problem can be stated as follows[7]. Find

$$x = (x^1, x^2...., x^n)$$

which minimizes

$$f(x)$$

subject to the constraints

$$g_j(x) \leq 0$$

for $j = 1, ..., m$, and

$$l_j(x) = 0$$

for $j = 1, ..., p$.
The variable x is called the design vector, $f(x)$ is the objective function, $g_j(x)$ are the inequality constraints and $l_j(x)$ are the equality constraints. The number of variables n and the number of constraints $p + m$ need not be related. If $p + m = 0$ the problem is called an unconstrained optimization problem [5].

3 Line Search

Now, let us consider the problem

$$\min_{x \in R^n} f(x)$$

where $f : R^n \to R$ is a continuously differentiable function, bounded from below. There exists a great number of methods made in the aim to solve the problem.
The optimization methods based on line search utilize the next iterative scheme:

$$x_{k+1} = x_k + t_k d_k$$

where x_k is the current iterative point, $xk + 1$ is the next iterative point, d_k is the search direction, and t_k is the step size in the direction d_k.
In our project we have implemented the constant value for alpha, step size, armijo line search method and Golden Section method [9].

4 Optimization Algorithms

4.1 Steepest Descent Algorithm

The classical steepest descent method [7] which is designed by Cauchy can be considered as one among the most important procedures for minimization of real-valued function defined on R^n. Steepest descent is one of the simplest minimization methods for unconstrained optimization. Since it uses the negative gradient as its search direction, it is known also as the gradient method. If, instead, one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent.

It has low computational cost and low matrix storage requirement, because it does not need the computations of the second derivatives to be solved to calculate the search direction.
Suppose that $f(x)$ is continuously differentiable in a certain neighborhood of a point x_k and also suppose that $g_k = \nabla f(x_k) \neq 0$.

Using Taylor expansion of the function f near x_k as well as Cauchy-Schwartz inequality, one can easily prove that the greatest fall of f exists if and only if $d_k = -g_k$, i.e.,g_k is the steepest descent direction. The iterative scheme of the SD method is $x_{k+1} = x_k - w_k g_k$. The classical steepest descent method uses the exact line search. Now, we see the algorithm of the steepest descent method,

The algorithm is given below: Choose $x_0 \in R^n$. Set $j = 0$.

Iteration j:

- **Step 1:** Stop if $\nabla V(x_j) = 0$ as then x_j satisfies a necessary condition for a local minimizer. (In practice, use $\|\nabla V(x_j)\| < 10^{-8}$.)
- **Step 2:** Set $s_j = -\nabla V(x_j)$

- **Step 3:** Choose a scalar $w_j > 0$ so $V(x_j + w_j s_j) < V(x_j)$ Often, w_j is chosen to minimize $V(x_j + ws_j)$ with $w \in R$ or at-least approximately.(α updated using Golden section method)[9]
- **Step 4:** Set $x_{j+1} = x_j + w_j s_j$ and $j = j + 1$.
- **Step 5:** Go to step 1

The MATLAB program is given in Appendix 1 and results are shown in Results section. The initial code has gradient calculation by partial differentiation and substitution method whereas the final code has the partial differentiation method.

4.2 Optimization algorithm with Armijo line search

A backtracking line search, a search scheme based on the Armijo–Goldstein condition[10], is a line search method to determine the maximum amount to move along a given search direction. It involves starting with a relatively large estimate of the step size for movement along the search direction, and iteratively shrinking the step size (i.e., "backtracking") until a decrease of the objective function is observed that adequately corresponds to the decrease that is expected, based on the local gradient of the objective function.
For standard quadratic V

$$\hat{w_j} = argmin_{w \in R^n} V(x_j + ws_j)$$

One practical way to estimate $\hat{w_j}$ is the Armigo Algorithm.
Let $a > 0, \sigma \in (0, 1)$ and

$$\gamma \in (0, 1/2)$$

be given and define the set of points

$$A = \alpha \in R : \alpha = a\sigma_j, j = 0, 1, ...$$

Armijo method consists in finding the largest $\alpha \in A$ such that

$$\Phi(\alpha) = f(x_k + \alpha d_k) - f(x_k) \leq \gamma\alpha\nabla f(x_k)'dk = \gamma\alpha\phi(0)$$

Armijo method can be implemented using the following (conceptual) algorithm.

- **Step 1:** $\alpha = a$..
- **Step 2:** If $f(x_k + \alpha d_k) - f(x_k) \leq \gamma\alpha\nabla f(x_k)'dk$. Set $\alpha_k = \alpha$ and STOP. Else go to Step 3.
- **Step 4:** Set $\alpha = \sigma\alpha$, and go to Step 2

The main disadvantage of Armijo method is in the fact that, to find α_k, all points in the set A, starting from the point $\alpha = a$, have to be tested till the condition in Step 2 is fulfilled. There are variations of the method that do not suffer from this disadvantage. We can use Goldstein conditions,a criterion similar to Armijo's, but that allows to find an acceptable α_k in one step.

The MATLAB program is given in Appendix 2 and results are shown in Results section.

4.3 Conjugate-Gradient Algorithm

The conjugate gradient method[7] is an algorithm for finding the nearest local minimum of a function of n variables which presupposes that the gradient of the function can be computed. It uses conjugate directions instead of the local gradient for going downhill. If the vicinity of the minimum has the shape of a long, narrow valley, the minimum is reached in far fewer steps than would be the case using the method of steepest descent.

It is an algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive-definite. The conjugate gradient method is often implemented as an iterative algorithm, applicable to sparse systems that are too large to be handled by a direct implementation or other direct methods such as the Cholesky decomposition. Large sparse systems often arise when numerically solving partial differential equations or optimization problems.

The Algorithm of Conjugate Gradient is as follows: Choose $x_0 \in \mathbf{R}^n$, set $j = 0$

- **Step 1:** Set $s_j = -\nabla V(x_j)$
- **Step 2:** Stop if $\|\nabla V(x_j)\| < \epsilon$, As then $x_j \cong \hat{x}$
- **Step 3:** Choose $W_j \in \arg\min V(x_j + ws_j)$

$$x_{j+1} = x_j + ws_j$$

- **Step 4:** Set

$$\beta_{j+1} = \frac{[\nabla V(x_{j+1} - \nabla V(x_j)]^{\nabla} V(x_{j+1})}{\|\nabla V(x_j)\|^2} \in R$$

$$s_{j+1} = -\nabla V(x_{j+1}) + \beta_{j+1} s_j$$

- **Step 5:** $j = j+1$ and return to step 1. (α updated using Golden section method)[9]

The MATLAB program is given in Appendix 3 and results are shown in Results tab. The initial code has gradient calculation by partial differentiation and substitution method whereas the final code has the partial differentiation method.

4.4 Secant Algorithm

Conjugate gradient methods have proved to be more efficient than the gradient method. However, in general, it is not possible to guarantee superlinear convergence. The main advantage of conjugate gradient methods is in the fact that they do not require to construct and store any matrix, hence can be used in large scale problems.

Consider the Newton method. For various practical problems, the computation of Hessian may be very expensive, or difficult, or Hessian can be unavailable analytically. So, the class of so-called Secant or quasi-Newton methods is formed, such that it uses only the objective function values and the gradients of the objective function and it is close to Newton method. Secant Method[7] is such a class of methods which does not compute Hessian, but it generates a sequence of Hessian approximations and maintains a fast rate of convergence. One of the first quasi-Newton methods has been proposed by Davidon, Fletcher and Powell.
The algorithm used in Secant Method is as follows:
Choose
(i) $x_0 \in R^n$
(ii) symmetric, positive definite $H_0 \in R^{n\tilde{A}[U+0097]n}$. H_0 is an estimate of unknown $C^{\hat{a}[U+0088][U+0092]1}$.
Use $H_0 = I$ if no better guess is available.
Set $j := 0$
Iteration j:

- **Step 1:** Set $s_j := -H_j \nabla V(x_j)$ (pseudo-Newton search direction).
- **Step 2:** Choose $w_j \geq 0$ so $V(x_j + w_j s_j) < V(x_j)$. Ideally w_j is chosen to minimize this.
- **Step 3:** Set $x_{j+1} := x_j + w_j s_j$
- **Step 4:** Stop if $\|\nabla V(x_j)\| < \epsilon =$ small
- **Step 5:** $\Delta x_j := x_{j+1} - x_j$, $\Delta g_j := \nabla V(x_{j+1}) - \nabla V(x_j)$.
- **Step 6:** Choose symmetric, positive definite $H_{j+1} \in R^{n \times n}$ so $H_{j+1} \sim= C^{-1}$ −improved.
- **Step 7:** Set $j := j + 1$ and return to Step 1.

There exist many ways for choosing suitable H_j. The most famous is the DFP.[10] H_{j+1} in the above algorithm is chosen using

$$Hj + 1 = H_j + \frac{\Delta x_j (\Delta x_j)^T}{(\Delta x_j)^T \Delta g_j} - \frac{H_j \Delta g_j (H_j \Delta g_j)^T}{(\Delta g_j)^T H_j \Delta g_j}$$

If approximate minimization is used along each s_j, we have to use other update schemes for finding H_{j+1}, e.g. Symmetric Rank, BFGS (probably the best), Broyden.

The MATLAB program is given in Appendix 4 and results are shown in Results tab. The initial code has gradient calculation by partial differentiation and substitution method whereas the final code has the partial differentiation method.

4.5 Optimization methods involving finite difference approximations of the gradient

We can estimate the derivative of a function at a point by evaluating the function at nearby points. This gives a useful method for checking gradient computations.
The derivative $f'(x)$ of a function $f(x)$ at an arbitrary point x is usually approximated by finite difference method like forward, central and backward differences.
In our project we use forward-difference derivative as they consume less computer time.
First-order derivatives:

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + h_i e_i) - f(x)}{h_i}$$

Second-order derivatives for dense Hessian

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_i h_j}$$

Steepest descent, Conjugate gradient and secant methods are implemented using finite differences whereas the grad computation is done in armijo line search method

4.6 Penalty and Barrier Method

Penalty function methods[1] work in a series of sequences, each time modifying a set of penalty parameters and starting a sequence with the solution obtained in the previous sequence. At any sequence, the following penalty function[4] is minimized:

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

where R is a set of penalty parameters, Ω is the penalty term chosen to favor the selection of feasible points over infeasible points. For equality or inequality constraints, different penalty terms are used.

- **Option 1:** Penalty Method:

$$\Omega = -R \sum_{j \in J} \mid g(x) \mid .$$

This penalty term is used for handling inequality constraints. Here, the term R is a large number and J denotes the set of violated constraints at the current point. Thus, a penalty proportionate to the constraint violation is added to the objective function. In this term, only one sequence with a large value of the penalty parameter is used. Since only infeasible points are penalized, this is also an exterior penalty term[4]

- **Option 2:** Log penalty(Barrier):

$$\Omega = -Rln[g(x)].$$

This penalty term is also used for inequality constraints. For infeasible points, $g(x) < 0$. Thus, this penalty term cannot assign penalty to infeasible points. For feasible points, more penalty is assigned to points close to the constraint boundary or points with very small $g(x)$. Since only feasible points are penalized, this is an interior penalty term pr Barrier method [4].

- **Option 3:** Inverse penalty(Barrier):

$$\Omega = R[\frac{1}{g(x)}].$$

Like the log penalty term, this term is also suitable for inequality constraints. This term penalizes only feasible pointsâ[U+0080][U+0094]the penalty is more for boundary points. This is also an interior penalty term or Barrier [4] and the penalty parameter is assigned a large value in the first sequence.

The algorithm for the above methods is as follows:

- **Step 1:** Choose two termination parameters ϵ_1, ϵ_2 , an initial solution $x^{(0)}$, a penalty term Ω, and an initial penalty parameter $R^{(0)}$. Choose a parameter c to update R such that $0 < c < 1$ is used for interior penalty terms and $c > 1$ is used for exterior penalty terms. Set $t = 0$.
- **Step 2:** Form $P(x^{(t)}, R^{(t)}) = f(x^{(t)}) + \Omega(R^{(t)}, g(x^{(t)}), h(x^{(t)}))$.
- **Step 3:** Starting with a solution $x^{(t)}$, find $x^{(t+1)}$ such that $P(x^{(t+1)}, R^{(t)})$ is minimum for a fixed value of $R^{(t)}$. Use ϵ_1 to terminate the unconstrained search.
- **Step 4:** Is $\mid P(x^{(t+1)}, R^{(t)}) - P(x^{(t)}, R^{(t-1)}) \mid \leq \epsilon_2$ If yes, $x = x^{(t+1)}$ and Terminate. Else go to Step 5.
- **Step 5:** Choose $R^{(t+1)} = cR^{(t)}$. Set $t = t + 1$ and go to Step 2.

The MATLAB program is given in Appendix 5 and results are shown in Results tab. Part A of the code deals with barrier and penalty methods with inequality constraints and Part B of the code deals with Penalty method[9] involving an equality and an inequality constraint.

4.7 Augmented Lagrangian method

The Augmented Lagrangian method [9] [8] is an indirect method for solving a constrained optimization problem. This method is also used to determine the minimum of the constrained problem. It is an algorithm to solve nonlinear optimization problems with equality and inequality constraints. The idea is to add to the objective functions a penalization

for the violation of some of the constraints, and this function will depend of some multipliers(weights). At each iteration a stationary point for the obtained problem is found, and then the multipliers are updated. Under certain conditions, it can be proved that the accumulation points of the generated sequence are feasible and are KKT points for the original problem. The objective is to:

Minimize:

$$f(x_1, x_2, ...x_n)$$

Subject to:

$$h_k(x_1, x_2, ...x_n) = 0, k = 1, 2, ...l$$
$$g_j(x_1, x_2, ...x_n) = 0, j = 1, 2, ...m$$

Side Constraints:

$$x_i^l \leq x_i \leq x_i^u; i = 1, 2, ...n$$

Transformation to Unconstrained Problem:
Minimize:

$$F(X^q, \lambda^q, \beta^q, r_k^q, r_g^q) : F(X) + r_h \sum_{k=1}^{l} h_k(X)^2 + r_g \sum_{j=1}^{m} (max[g_j(X), -\frac{\beta}{2r_g}])^2$$

$$+ \sum_{k=1}^{l} \lambda_k h_k(X) + \sum_{j=1}^{m} \beta_j (max[g_j(X), -\frac{\beta}{2r_g}])$$

Side Constraints:

$$x_i^l \leq x_i \leq x_i^u; i = 1, 2, ...n$$

- **Step 1:** Choose appropriate Parameters like number of iterations, penalty multipliers, tolerance, scaling value for multipliers, maximum value for multipliers and initial multiplier values.
- **Step 2:** Call secant method to minimize $F(X^q, \lambda^q, \beta^q, r_k^q, r_g^q)$ where q is the iteration counter.
- **Step 3:** Check for convergance with the stopping criteria:

$$\Delta F = F_q - F_{q-1}, \Delta X = X_q^* - X_{q-1}^*$$

if

$$\Delta F \Delta F \leq \varepsilon_1 : Stop$$

if

$$\Delta X \Delta X \leq \varepsilon_1 : Stop$$

if

$$q = N_s; Stop$$

- **Step 4:** Update parameters and Go to Step 2.

The MATLAB program is given in Appendix 6 and results are shown in Results tab.

4.8 Newton Lagrange Method

Sequential Quadratic programming are very efficient methods for solving non-linear problems. These methods can be obtained by applying Newton's Method to find a stationary point of Lagrangian function, and therefore called as Newton - Lagrange method.[7]

Consider the problem of minimizing a scalar objective function $I(X, u)$ subject to the constraint that $F(X, u) = 0$, where $X = (X_1, X_2, .., X_n)^T$ and $u = (u_1, u_2, .., u_m)^T$. Typically, $F(X, u)$ is a discretization of a boundary value problem which given u can be solved for X. We assume that this solvability implies that $\partial F_i/\partial X_k$ is invertible for the values of u of interest. Jacobian and Hessian matrices are calculated. Lagrangian multipliers are included. . If Newton's method is applied, we obtain the following equations for the updates u and$X$. Please refer to lecture notes for detailed algorithm.

The MATLAB program is given in Appendix 7 and results are shown in Results tab.

5 Results

Please refer to the detailed Result section provided in Page 12. All the algorithms are tested and results are tabulated along with necessary plots.
1. Steepest Descent: Table 1, Figures 1,2,3,4,5 and Appendix 1.
2. Armijo line search : Figures 6,7,8,9 and Appendix 2.
3. Conjugate Gradient : Table 2, Figures 10, 11, 12, 13, 14 and Appendix 3.
4. Secant Method: Table 3, Figures 15, 16, 17, 18, 19 and Appendix 4.
5. Finite Differences Method: Refer to the results of Steepest descent, Conjugate Gradient and Secant Method and Appendix 1,3 and 4.
6. Penalty and Barrier Methods: Figures 20, 21, 22 and Appendix 5.
7. Augumented Lagrangian: Figure 23 and Appendix 6.
8. Newton Lagrange Method: Verify the results with Appendix 7.

Complete code of all methods can be viewed in Appendix section from Page 29. Few results of algorithms are verified with minimization of function by fmincon of Optimization toolbox and the proposed method yielded results with less iterations.

6 Discussion and Conclusion

More concentration is given to the theoretical studies on methods for solving general unconstrained and constrained minimization problems The components of a constrained optimization problem are its objective function, its decision variable, and its constraint. The objective function represents the quantitative measure that the decision maker aims to minimize/maximize. Decision variables are constituents of the system for which decisions are being taken to improve the value of the objective function. The constraints are the restrictions on decision variables, often pertaining to resources. These restrictions are defined by equalities /inequalities involving functions of decision variables. In addition, parameters are constant values used in objective function and constraints, like the multipliers for the decision variables or bounds in constraints. We investigated all of the

above with the algorithms like Steepest Descent, Conjugate Gradient, Secant, Barrier and Penalty, Augumented Lagrangian and Newton Lagrange methods are dealt and results are reproduced. There was a little difficulty in understanding some complex methods like Newton Lagrange owing to time constraints.

We have seen how the line search methods play an important role in those algorithms. This paves the way for Sequential programming. The theory of trust region methods has several advantages over that of line search methods but both approaches seem to perform equally well in practice. Line search methods are more commonly used because they have been known for many years and because they can be simpler to implement At present, line search and trust region methods coexist and it is difficult to predict if one of these two approaches will become dominant. This will depend on the theoretical and algorithmic advances that the future has in store

7 Statement of Contributions

Surya Kumar Devarajan, Nikhil Jayam, Ameya Bhope, Shanthanil Bhagchi, Santosh Devapati and Anjali Maria joined together in brainstroming ideas, worked on algorithms, implemented the codes and tested the results. We exchanged our ideas about the implementation of various tasks owing to the complexity of the project and time constraints.

References

[1] Berhe, H. W. Barrier function methods using matlab. *Scholarly Journal of Mathematics and Computer Science* (2012).

[2] Boyd S P, V. L. Convex optimization. *Cambridge University Press* (2018).

[3] Deb, K. Multi-objective optimization using evolutionary algorithms. *John Wiley Sons* (2008).

[4] Deb, K. Optimization for engineering design. *PHI Learning Private Limited* (2012).

[5] Fletcher, R. Practical methods of optimization. chichester. *Wiley* (1986).

[6] Matlab optimization toolbox, R2017B.

[7] Michalska, H. *Optimization and Optimal Control Notes*. 2003.

[8] P.Venkataraman. https://people.rit.edu/pnveme/mod6numerical/mod6sec3alm.html. *ALM* (2012).

[9] Venkataraman, P. Applied optimization with matlab programming. *John Wiley and Sons* (2002).

[10] Wikipedia. https://en.wikipedia.org. *Optimization algorithms and methods* (2018).

RESULTS

1. Steepest Descent Method:

	Problem	Iteration	Minimal point	Minimal function

1	a = [9, 1, 7, 5, 4, 7; 1, 11, 4, 2, 7, 5; 7, 4, 13, 5, 0, 7; 5, 2, 5, 17, 1, 9; 4, 7, 0, 1, 21, 15; 7, 5, 7, 9, 15, 27];
	b = [1, 4, 5, 4, 2, 1];
	c = 5;
	retval = transpose([x(1);x(2);x(3);x(4);x(5);x(6)])*a*[x(1);x(2);x(3);x(4);x(5);x(6)]+ b*[x(1);x(2);x(3);x(4);x(5);x(6)]+c;
		3	0.3338
			0.0540
			-0.4274
			-0.1914
			-0.2693
			0.2091
				3.6550

2	retval = (sqrt((x(1)^2 +1)*(2*x(2)^2 + 1)))/(x(1)^2+x(2)^2+0.5);	2	No Minima Maxima is [0 0]	Function is maximum at -2

3	b = [1,2];
	a = [12 ,3; 3, 10];
	retval = 1 + b*[x(1);x(2)] + 0.5* transpose([x(1);x(2)])*a*[x(1);x(2)]+10*log(1+(x(1)^4))*sin(100*x(1))+10*log(1+(x(2)^4))*cos(100*x(2));
		5	-0.1176 -0.6599	0.3209

e1	retval = x(1) - x(2) + 2*x(1)^2 + 2*x(1)*x(2) + x(2)^2;	14	-0.9996 1.4993	-1.2500
e2	retval = x(1)^2+x(2)^2;	2	0.0000 0.0000	0.0000

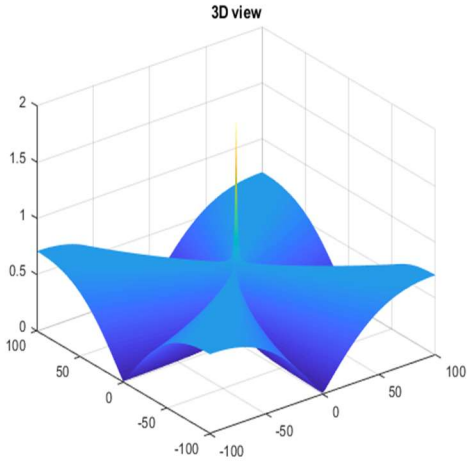
Table 1: List of Part A problems and some examples minimized by Steepest Descent method. Number of iterations and Optimal values are specified.

Part A, 1st problem:

```
a = [9, 1, 7, 5, 4, 7; 1, 11, 4, 2, 7, 5; 7, 4, 13, 5, 0, 7; 5, 2, 5, 17, 1, 9; 4, 7, 0, 1, 21, 15; 7, 5, 7, 9, 15, 27];
b = [1, 4, 5, 4, 2, 1];
c = 5;
sol =
transpose([x(1);x(2);x(3);x(4);x(5);x(6)])*a*[x(1);x(2);x(3);x(4);x(5);x(6)]+ b*[x(1);x(2);x(3);x(4);x(5);x(6)]+c;

SteepestDescent('func',[0 0 0 0 0 0],30, 0.0001, 0,1 ,15)

No. of iterations:      3
 - Optimal point, Value of function at Optimal point

ans =
    0.3338    0.0540   -0.4274   -0.1914   -0.2693    0.2091    3.6550
```

Part A, 2nd problem:

```
sol = -(sqrt((x(1)^2 +1)*(2*x(2)^2 + 1)))/(x(1)^2+x(2)^2+0.5);
```



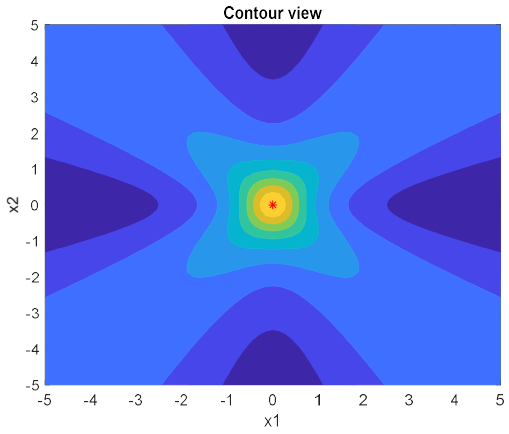


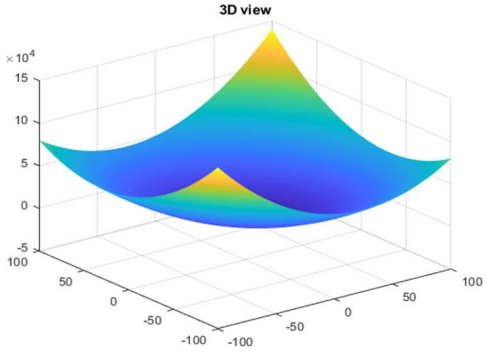
Figure 1: a) 3D view of the function *b) Contour view of the function*

As you can above from 3d view, there is only maxima for this problem. So, we are finding the maxima.

```
SteepestDescent('func',[0 0],30, 0.0001, 0,1 ,15)

No. of iterations:      2

 - Optimal point, Value of function at Optimal point

ans =
            0         0        -2
```

Part A, 3rd problem:

```
b = [1,2];
a = [12 ,3; 3, 10];
sol = 1 + b*[x(1);x(2)] + 0.5*
transpose([x(1);x(2)])*a*[x(1);x(2)]+10*log(1+(x(1)^4))*sin(100*x(1))+10*
log(1+(x(2)^4))*cos(100*x(2));
```



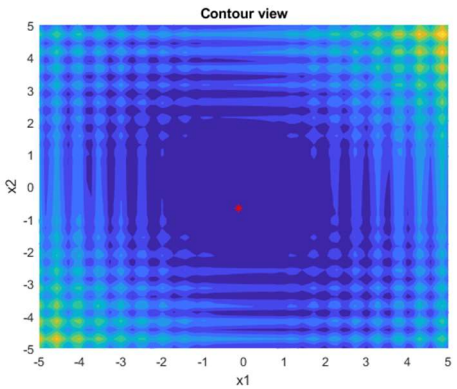


Figure 2: a) 3D view of the function *b) Contour view of the function*

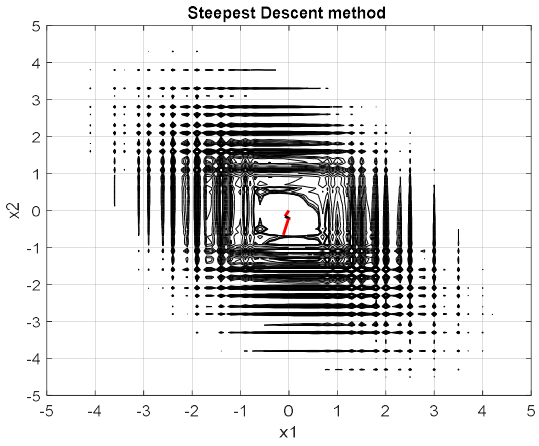
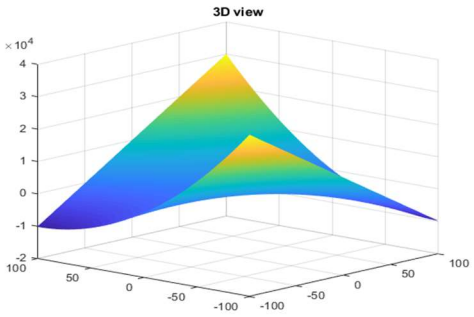


Figure 3: Plot of the minimal point (-0.1176, -0.6599) with the minimal function value at 0.3209

```
SteepestDescent('func',[0 0],30, 0.0001, 0,1 ,15)

No. of iterations:      5

 - Optimal point, Value of function at Optimal point

ans =
            -0.1176   -0.6599    0.3209
```

1st example:

sol = x(1) - x(2) + 2*x(1)^2 + 2*x(1)*x(2) + x(2)^2;



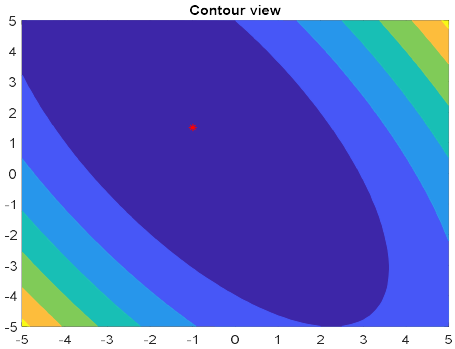


Figure 4: a) 3d view of the function *b) Contour plot of the function*

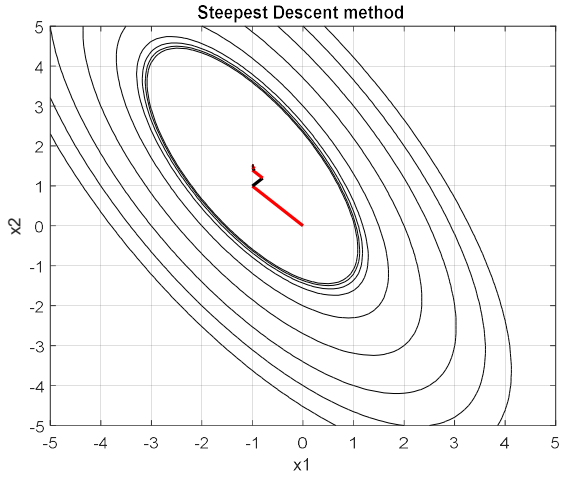


Figure 5: The plot of minimal point (-0.9996 ,1.4993) with the minimal function at -1.2500

```
SteepestDescent('func',[0 0],30, 0.0001, 0,1 ,15)
```

```
No. of iterations:      14

 - Optimal point, Value of function at Optimal point

ans =
                -0.9996    1.4993   -1.2500
```

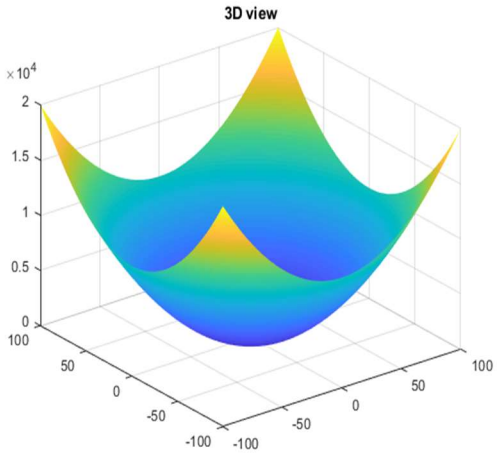
INFERENCE: The above results are checked with Optimization toolbox using fminunc. The unconstrained problem minimized by Part A, problem 2 takes 7 iterations to reach the optimal value where the above model takes only 5 iterations. This concludes the better performance of our algorithm.

---

2. Armijo Line search

1st example:

fun = x(1)^2+x(2)^2;



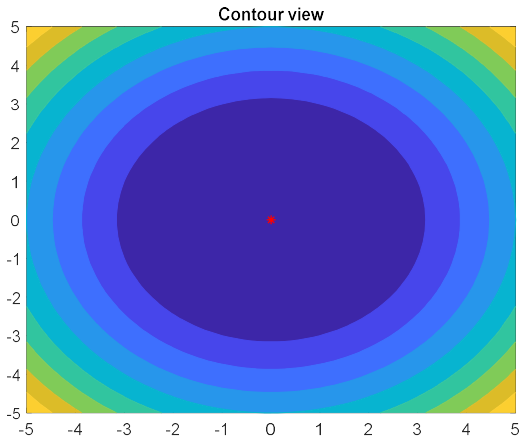


Figure 6: a) 3D plot of the function *b) Contour view of the function*

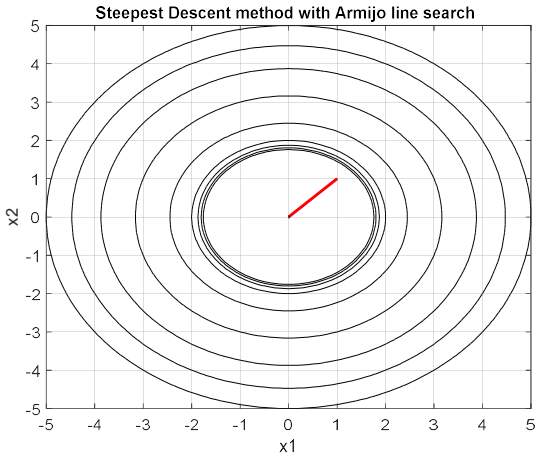


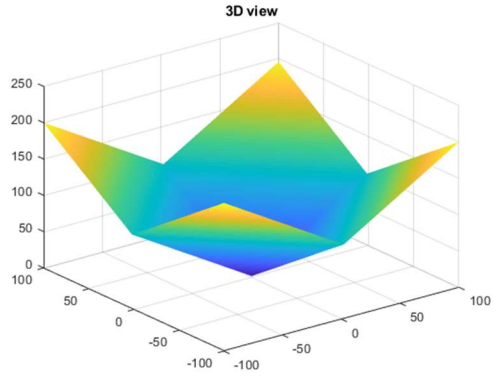
Figure 7: Plot of the minimal point (0,0) for the function

```
>> ArmijoLineSearch(@fun_armijo,[1 1])
Minimum point is obtained
```

```
Number of Iterations for the cost function convergence: 2

Mininimum point:
0
0
```

2nd example:

```
fun = abs(x(1)-2+abs(x(2)-2);
```



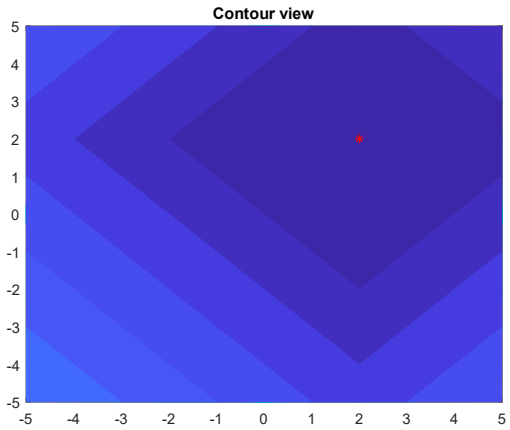


Figure 8: a) 3D view of the plot *b) Contour view of the plot*

>>

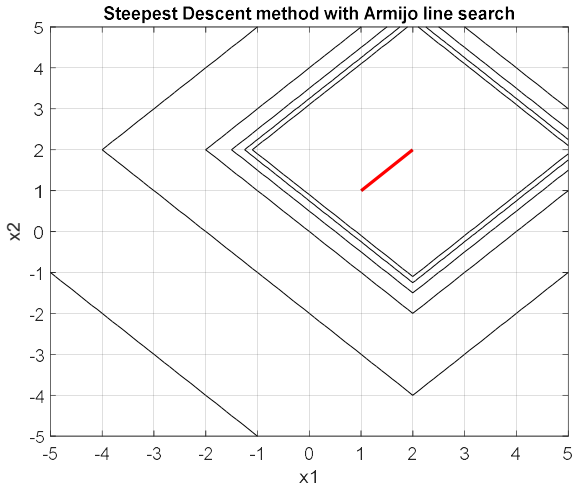


Figure 9: Plot of the minimal point (2,2) of the function

```
>>
ArmijoLineSearch(@fun_armijo,[1 1])

Minimum point is obtained

Number of Iterations for the cost function convergence: 1

Mininimum point:
     2
     2
```

---

3. Conjugate Gradient Method:

	Problem	Iterations	Minimal point	Minimal function

A	a = [9, 1, 7, 5, 4, 7; 1, 11, 4, 2, 7, 5; 7, 4, 13, 5, 0, 7; 5, 2, 5, 17, 1, 9; 4, 7, 0, 1, 21, 15; 7, 5, 7, 9, 15, 27];
	b = [1, 4, 5, 4, 2, 1];
	c = 5;
	retval = transpose([x(1);x(2);x(3);x(4);x(5);x(6)])*a*[x(1);x(2);x(3);x(4);x(5);x(6)]+ b*[x(1);x(2);x(3);x(4);x(5);x(6)]+c;
		2	3.3582e-01
			5.5265e-02
			-4.2907e-01
			-1.9181e-01
			-2.7108e-01
			2.0988e-01
				3.6550e+00

B	retval = (sqrt((x(1)^2 +1)*(2*x(2)^2 + 1)))/(x(1)^2+x(2)^2+0.5);	2	No Minima Maxima is [0 0]	Function is maximum at -2

C	b = [1,2];
	a = [12 ,3; 3, 10];
	retval = 1 + b*[x(1);x(2)] + 0.5* transpose([x(1);x(2)])*a*[x(1);x(2)]+10*log(1+(x(1)^4))*sin(100*x(1))+10*log(1+(x(2)^4))*cos(100*x(2));
		26	-2.6319e-01
			-1.0931e-01
				1.0354e+00

1	retval = x(1) - x(2) + 2*x(1)^2 + 2*x(1)*x(2) + x(2)^2;	3	-9.9991e-01 1.4998e+00	-1.2500e+00
2	retval = x(1)^2+x(2)^2;	2	1.1921e-04 1.1921e-04	2.8421e-08

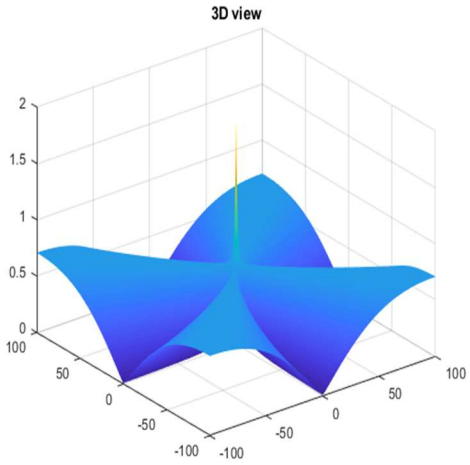
Table 2:List of Part A problems and some examples minimized by Conjugate Gradient method. Number of iterations and Optimal values are specified.

Part A, 1st problem:

```
a = [9, 1, 7, 5, 4, 7; 1, 11, 4, 2, 7, 5; 7, 4, 13, 5, 0, 7; 5, 2, 5, 17, 1, 9; 4, 7, 0, 1, 21, 15; 7, 5, 7, 9, 15, 27];

b = [1, 4, 5, 4, 2, 1];
c = 5;
sol =
transpose([x(1);x(2);x(3);x(4);x(5);x(6)])*a*[x(1);x(2);x(3);x(4);x(5);x(6)]+ b*[x(1);x(2);x(3);x(4);x(5);x(6)]+c;

Conjugategradient('func',[0 0 0 0 0 0],30, 0.0001, 0,1 ,15)
No. of iterations:      2
Optimal point, Value of func at minimal pt

ans =
   3.3582e-01   5.5265e-02  -4.2907e-01  -1.9181e-01  -2.7108e-01
2.0988e-01   3.6550e+00
```

Part A, 2nd problem:

```
sol = -(sqrt((x(1)^2 +1)*(2*x(2)^2 + 1)))/(x(1)^2+x(2)^2+0.5);
```



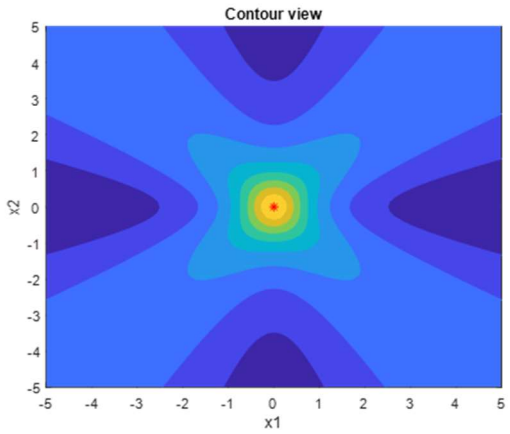


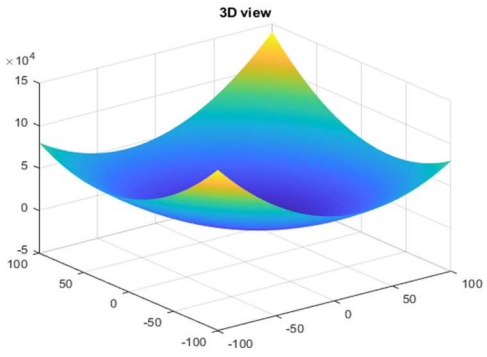
Figure 10: a) 3D view of the function *b) Contour view of function*

As you can above, there is no minima for this problem. So, we are finding the maxima.

```
Conjugategradient('func',[0 0],30, 0.0001, 0,1 ,15)

No. of iterations:      2
 - Optimal point, Value of function at Optimal point

ans =
            0         0        -2
```

Part A, 3rd problem:

```
b = [1,2];
a = [12 ,3; 3, 10];
sol = 1 + b*[x(1);x(2)] + 0.5*
transpose([x(1);x(2)])*a*[x(1);x(2)]+10*log(1+(x(1)^4))*sin(100*x(1))+10*
log(1+(x(2)^4))*cos(100*x(2));
```



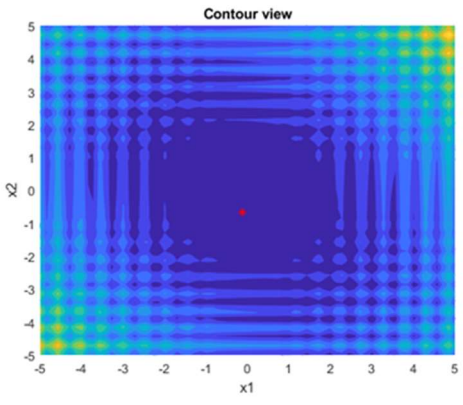


Figure 11: a)3D view of the function *b) Contour plot of the function*

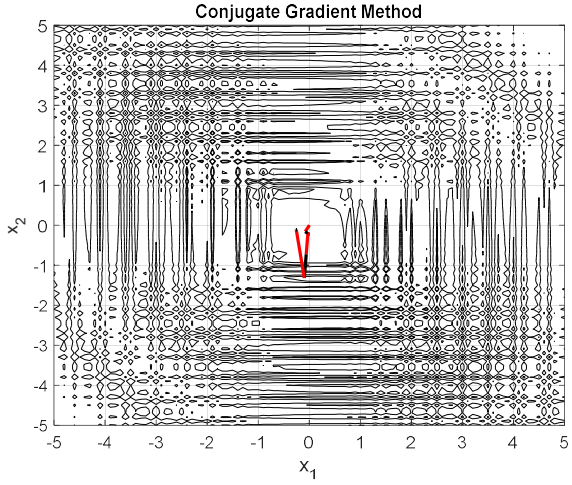
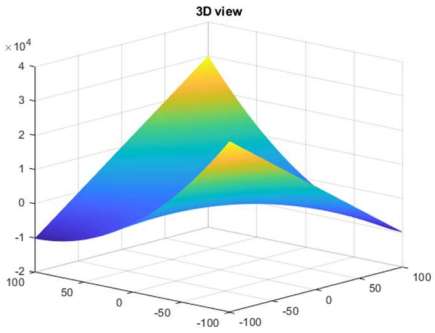


Figure 12: Plot of the minimal point (-2.6e-01, -1.09e-01) with the minimal function at 1.0354

```
Conjugategradient('func',[0 0],30, 0.0001, 0,1 ,15)
No. of iterations:      26

- Optimal point, Value of function at Optimal point
ans =
        -2.6319e-01   -1.0931e-01    1.0354e+00
```

1st example:

```
sol = x(1) - x(2) + 2*x(1)^2 + 2*x(1)*x(2) + x(2)^2;
```



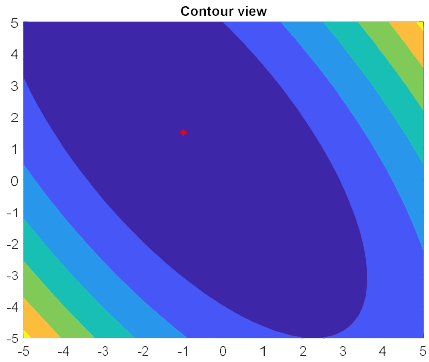


Figure 13: a) 3D view of the function *b) Contour plot of the function*

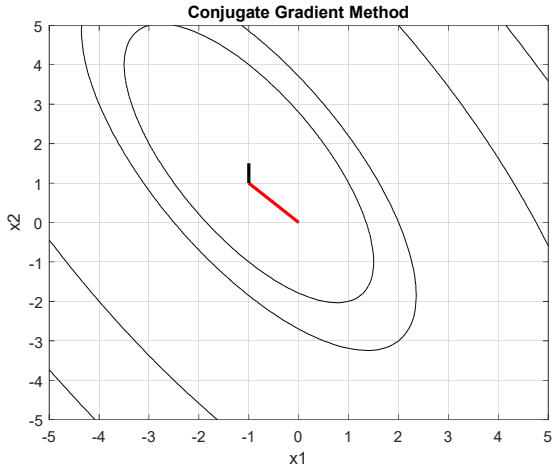


Figure 14: Plot of minimal point (-9.99e-01, 1.49e+00) with the minimal function value at -1.25

```
Conjugategradient('func',[0 0],30, 0.0001, 0,1 ,15)

No. of iterations:       3

- Optimal point, Value of function at Optimal point

ans =
      -9.9991e-01   1.4998e+00   -1.2500e+00
```

INFERENCE: The above results are checked with Optimization toolbox using fminunc. The unconstrained problem minimized by using example 1 takes 5 iterations to reach the optimal value where the above model takes only 3 iterations. This concludes the better performance of our algorithm.

---

4. Secant method:

	Problem	Iterations	Minimal point	Minimal function

1	a = [9, 1, 7, 5, 4, 7; 1, 11, 4, 2, 7, 5; 7, 4, 13, 5, 0, 7; 5, 2, 5, 17, 1, 9; 4, 7, 0, 1, 21, 15; 7, 5, 7, 9, 15, 27];
	b = [1, 4, 5, 4, 2, 1];
	c = 5;
	retval = transpose([x(1);x(2);x(3);x(4);x(5);x(6)])*a*[x(1);x(2);x(3);x(4);x(5);x(6)]+b*[x(1);x(2);x(3);x(4);x(5);x(6)]+c;
		2	3.3658e-01
			5.6085e-02
			-4.3003e-01
			-1.9202e-01
			-2.7142e-01
			2.0999e-01
				3.6550e+00

2	retval = (sqrt((x(1)^2 +1)*(2*x(2)^2 + 1)))/(x(1)^2+x(2)^2+0.5);	2	No Minima Maxima is [0 0]	Function is maximum at -2

3	b = [1,2];
	a = [12 ,3; 3, 10];
	retval = 1 + b*[x(1);x(2)] + 0.5* transpose([x(1);x(2)])*a*[x(1);x(2)]+10*log(1+(x(1)^4))*sin(100*x(1))+10*log(1+(x(2)^4))*cos(100*x(2));
		5	-4.2797e-02
			-1.6293e-01
				7.9014e-01

ex.1	retval = x(1) - x(2) + 2*x(1)^2 + 2*x(1)*x(2) + x(2)^2;	3	-1.0002e+00 1.5002e+00	-1.2500e+00
ex.2	retval = x(1^2+x(2)^2;	2	1.1921e-04 1.1921e-04	2.8421e-08

Table 3: Table 2:List of Part A problems and some examples minimized by Secant method. Number of iterations and Optimal values are specified.

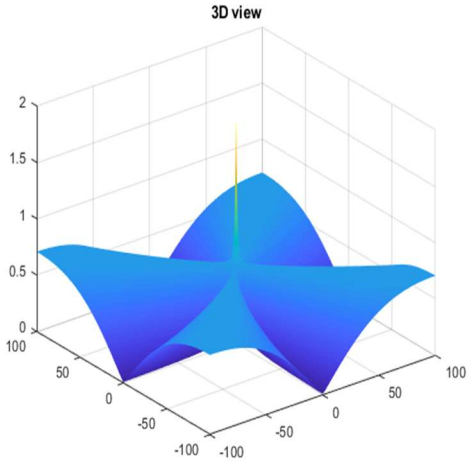
Part A, 1st problem:

```
a = [9, 1, 7, 5, 4, 7; 1, 11, 4, 2, 7, 5; 7, 4, 13, 5, 0, 7; 5, 2, 5, 17,
1, 9; 4, 7, 0, 1, 21, 15; 7, 5, 7, 9, 15, 27];
b = [1, 4, 5, 4, 2, 1];
c = 5;
sol =
transpose([x(1);x(2);x(3);x(4);x(5);x(6)])*a*[x(1);x(2);x(3);x(4);x(5);x(
6)]+ b*[x(1);x(2);x(3);x(4);x(5);x(6)]+c;
```

```
secant_DFP('func',[0 0 0 0 0 0],30, 0.0001, 0,1 ,15)

No. of iterations:      8

-Optimal point, Value of func at minimal pt

ans =

3.3658e-01   5.6085e-02   -4.3003e-01   -1.9202e-01   -2.7142e-01    2.0999e-
01    3.6550e+00
```

Part A, 2nd problem:

```
sol = -(sqrt((x(1)^2 +1)*(2*x(2)^2 + 1)))/(x(1)^2+x(2)^2+0.5);
```



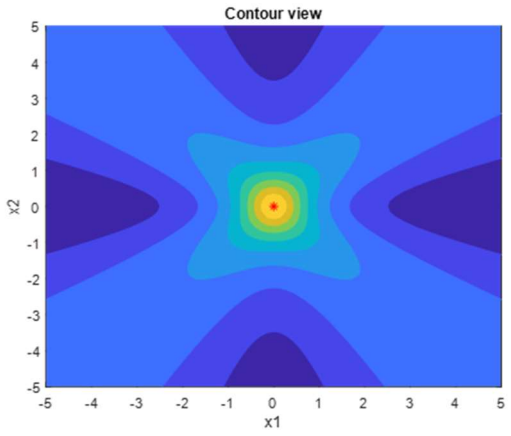


Figure 15: a) 3D plot of the function

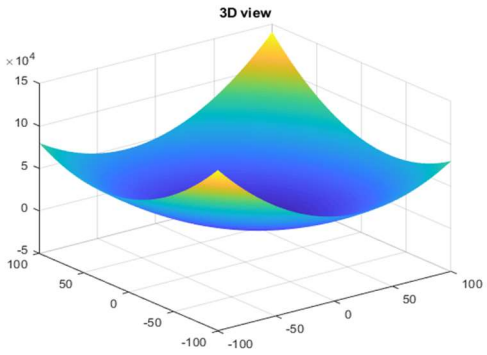
b) Contour view of the function

As you can above, there is no minima for this problem. So, we are finding the maxima.

```
secant_DFP('func',[0.2 0.1],30, 0.0001, 0,1 ,15)
No. of iterations:      2

 - Optimal point, Value of function at Optimal point

ans =
            0         0        -2
```

Part A, 3rd problem:

```
b = [1,2];
a = [12 ,3; 3, 10];
sol = 1 + b*[x(1);x(2)] + 0.5*
transpose([x(1);x(2)])*a*[x(1);x(2)]+10*log(1+(x(1)^4))*sin(100*x(1))+10*
log(1+(x(2)^4))*cos(100*x(2));
```



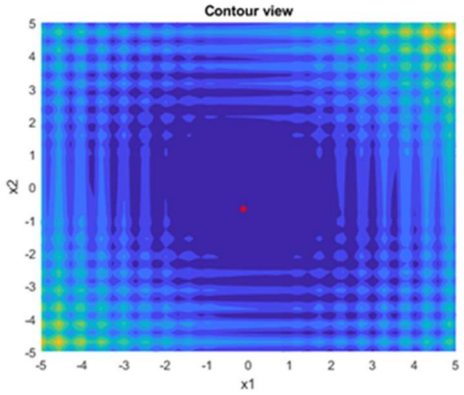


Figure 16: a) 3D view of the function *b) Contour plot of the function*

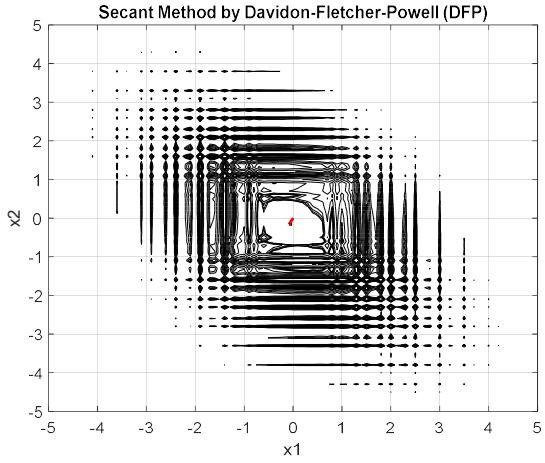
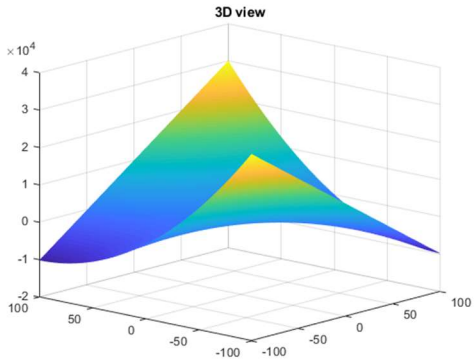


Figure 17: Plot of minimal point (-0.0428, -0.1629) with minimal function at 0.7901

```
secant_DFP('func',[0 0],30, 0.0001, 0,1 ,15)

No. of iterations:      5

 - Optimal point, Value of function at Optimal point

ans =
        -0.0428   -0.1629    0.7901
```

1st example:

```
sol = x(1) - x(2) + 2*x(1)^2 + 2*x(1)*x(2) + x(2)^2;
```



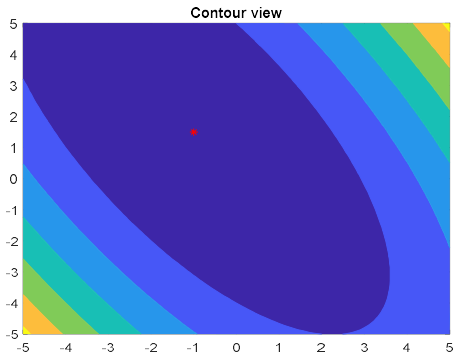


Figure 18: a) 3D plot of the function *b) Contour plot of the function*

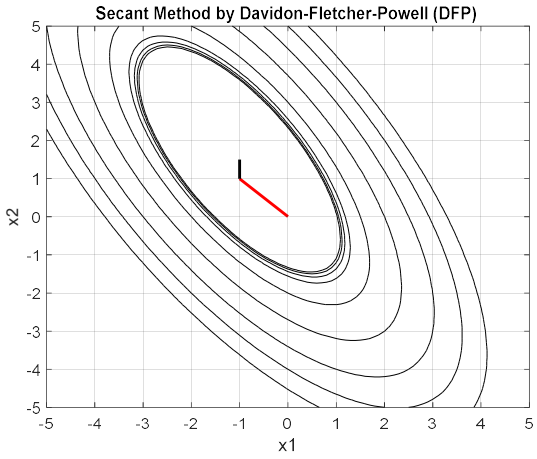


Figure 19: Plot of the minimal point(-1.0002, 1.5002) with its minimal function value at -1.25

```
secant_DFP('func',[0 0],30, 0.0001, 0,1 ,15)

No. of iterations:       3

- Optimal point, Value of function at Optimal point

ans =
      -1.0002    1.5002   -1.2500
```

INFERENCE: The above results are checked with Optimization toolbox using fminunc. The unconstrained problem minimized by using Part A, problem 2 and example 1 takes 7 and 5 iterations respectively to reach the optimal value where the above model takes only 5 and 3 iterations respectively. This concludes the better performance of our algorithm.

---

5. Finite Difference:

All the above methods except Armijo are implemented by Finite Differences. Kindly verify the results of 1, 3, 4, where gradient calculation is done on Armijo Line search.

---

6. Penalty and barrier

f is constrained function with barrier term added.
P is the unconstrained function formed by adding the objective function and the barrier function.

Example 1:

```
f = R.*(((x1-5).^2 +x2.^2 -26).^2);
P = (x1.^2 +x2 -11).^2 + (x1 +x2.^2-7).^2 +f;
```

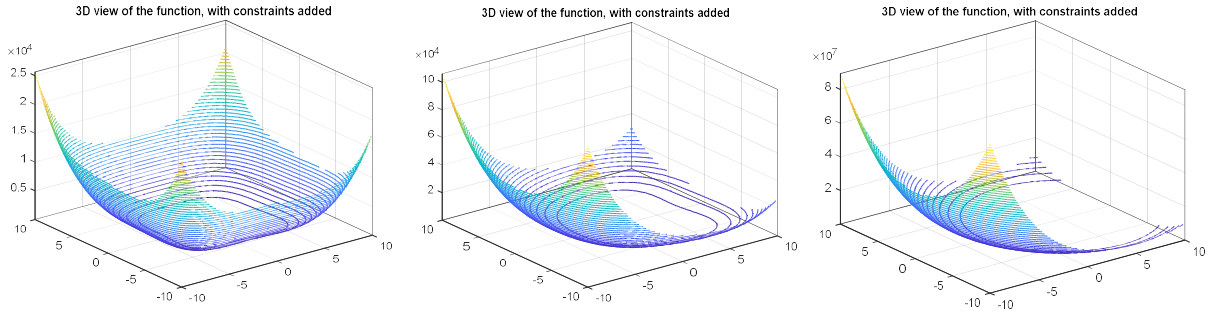


Figure 20: Comparison of the contour plot for different Penalty values. a) R = 0.1. b) R =1. c) R = 100

Output:

```
>> penalty_barrier
Enter 0 for Barrier method, 1 for penalty Method = 0
The minimum point is (2.999704,1.999905) with a function value of 0.0000

>> penalty_barrier
Enter 0 for Barrier method, 1 for penalty Method = 1
The minimum point is (2.999880,2.000282) with a function value of 0.0000
```

Example 2:

```
f = R.*((-x1*x2).^2);
P = (-x1+x2^2+1).^2 + (x1 +x2).^2 +f;
```

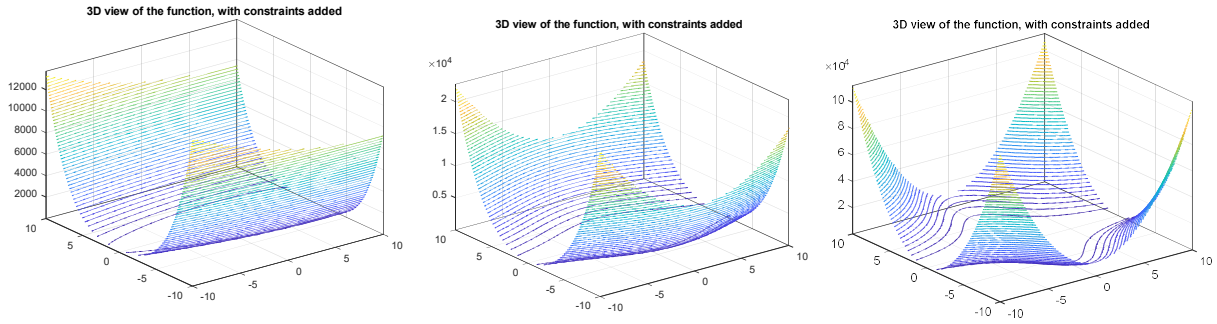


Figure 21: Comparison of the contour plot for different Penalty values. a) R = 0.1. b) R =1. c) R = 100

Output:

```
>> penalty_barrier
Enter 0 for Barrier method, 1 for penalty Method = 0
The minimum point is (0.875261,-0.499876) with a function value of 0.2813

>> penalty_barrier
Enter 0 for Barrier method, 1 for penalty Method = 1
The minimum point is (0.874179,-0.499223) with a function value of 0.2813
```

Example 3:

```
f = R.*((x1-x2*x2).^2);
P = abs(x1-2)+abs(x2-2)+f;
```

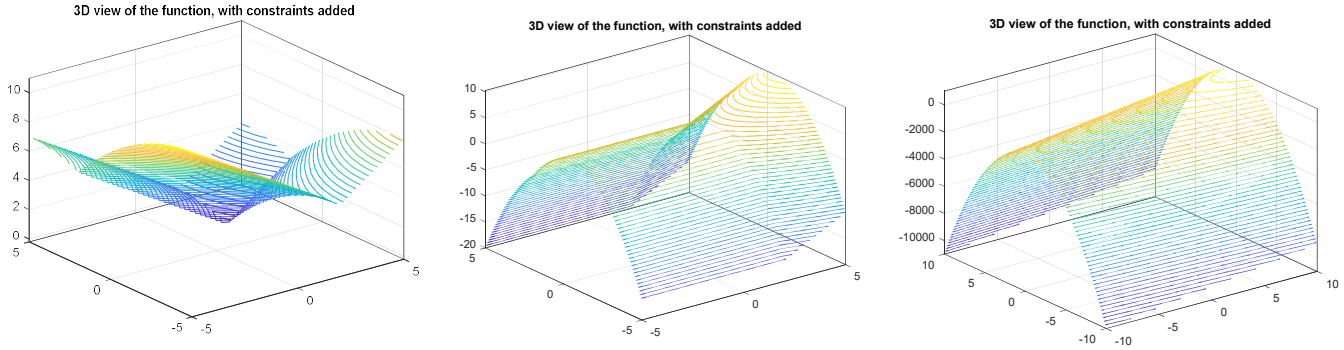


Figure 22: Comparison of the contour plot for different Penalty values. a) R = 0.1. b) R =1. c) R = 100

```
>> penalty_barrier
Enter 0 for Barrier method, 1 for penalty Method = 0
The minimum point is (1.999741,1.999857) with a function value of 0.0004
>> penalty_barrier
Enter 0 for Barrier method, 1 for penalty Method = 1
The minimum point is (1.999952,2.000305) with a function value of 0.0004
```

Penalty method involving one inequality and one equality constraint. We use secant method for minimization.

Part 2 – 1st problem:

```
f = abs(x(1)-2)+abs(x(2)-2);
g = x(1)-x(2)*x(2);
h = x(1)*x(1) + x(2)*x(2) -1;
```

Output:

```
 Penalty iteration:      1
Value of (X) :     7.5916e-01   8.5774e-01

 Penalty iteration:      2
Value of (X) :     6.3601e-01   7.9458e-01

 Penalty iteration:      3
Value of (X) :     6.1987e-01   7.8702e-01
```

```
 Penalty iteration:      4
Value of (X) :     6.1822e-01   7.8624e-01

 Penalty iteration:      5
Value of (X) :     6.1805e-01   7.8616e-01

 Optimal value x :
   6.1805e-01   7.8616e-01
```

Example 1:

```
f = x(1)^4 - 2*x(1)*x(1)*x(2) +x(1)*x(1) + x(1)*x(2)*x(2) -2*x(1) + 4
g = (0.25*x(1)*x(1) + 0.75*x(2)*x(2) -1);
h = sol = [(x(1)*x(1) + x(2)*x(2) -2)];
```

Output:

```
 Penalty iteration:      1
Value of (X) :     9.2194e-01   1.0393e+00

 Penalty iteration:      2
Value of (X) :     9.6040e-01   1.0281e+00

 Penalty iteration:      3
Value of (X) :     9.9316e-01   1.0051e+00

 Penalty iteration:      4
Value of (X) :     9.9922e-01   1.0006e+00

 Optimal value x :
   9.9922e-01   1.0006e+00
```

---

7. Augmented Lagrangian:

Example 1:

```
f = x(1)^4 - 2*x(1)*x(1)*x(2) +x(1)*x(1) + x(1)*x(2)*x(2) -2*x(1) + 4;
g = (0.25*x(1)*x(1) + 0.75*x(2)*x(2) -1);
h = [(x(1)*x(1) + x(2)*x(2) -2)];
```

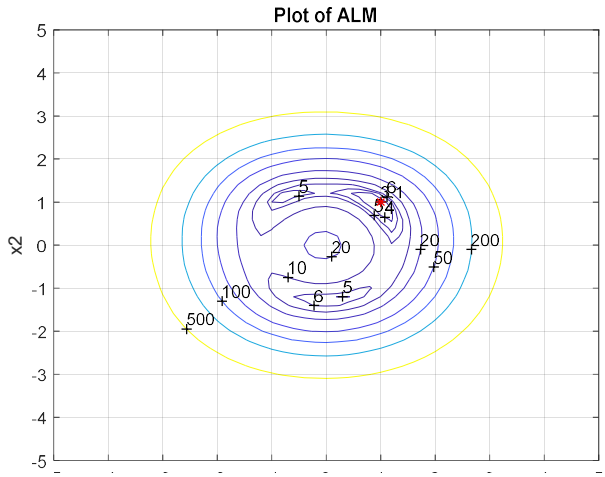


Figure 23: Constrained plot of the function solved by ALM method

```
ALM iteration:      1
Value of (X) :     7.9603e-01   7.1223e-01

ALM iteration:      2
Value of (X) :     9.3198e-01   1.0516e+00

ALM iteration:      3
Value of (X) :     9.9939e-01   9.9802e-01

ALM iteration:      4
Value of (X) :     1.0007e+00   9.9950e-01

ALM iteration:      5
Value of (X) :     1.0006e+00   9.9936e-01

Optimal value x :
   1.0006e+00   9.9936e-01
```

Example 2:

```
f = log(x(1))-x(2);
g = x(1)-1;
h = [(x(1)*x(1) + x(2)*x(2) -4)];

ALM iteration:      1
Value of (X) :     5.1845e-07   7.1427e-01

Function cannot be increased in ntrials
ALM iteration:      2
Value of (X) :    -5.7794e+04  -1.1896e+01

Function cannot be increased in ntrials
ALM iteration:      3
Value of (X) :    -7.2207e+15  -1.4862e+12

Optimal value x :
  -7.2207e+15  -1.4862e+12
```

Example 3:

```
f = abs(x(1)-2)+abs(x(2)-2);
g = x(1)-x(2)*x(2);
h = x(1)*x(1) + x(2)*x(2) -1;

ALM iteration:      1
Value of (X) :     3.1604e-01   7.9528e-01

ALM iteration:      2
Value of (X) :     4.3825e-01   6.3142e-01

Function cannot be increased in ntrials
ALM iteration:      3
```

```
Value of (X) :     5.4890e-01   7.4331e-01

ALM iteration:      4
Value of (X) :     6.1755e-01   7.8558e-01

ALM iteration:      5
Value of (X) :     6.1718e-01   7.8562e-01

Optimal value x :
   6.1718e-01   7.8562e-01
```

---

8. Lagrange-Newton

Example 1:

```
x = sym('x',[2,1]); %(example 1)
a = 1;
b = [1;2];
c = [12,3;3,10];
fx = a + (b'*x)+(0.5*x'*c*x);
cx = -1+x(1)^2+x(2)^2;

>> [x_optimal,Iterations]= lagrange_newton_method(fx,cx,[1;0],1e-4)
x_optimal =
   3.4630e-01
  -9.3812e-01
Iterations =
    50
```

Example 2:

```
x = sym('x',[6,1]); %(example 1)
a = [9, 1, 7, 5, 4, 7; 1, 11, 4, 2, 7, 5; 7, 4, 13, 5, 0, 7; 5, 2, 5, 17,
1, 9; 4, 7, 0, 1, 21, 15; 7, 5, 7, 9, 15, 27];
b = [1, 4, 5, 4, 2, 1];
c = 5;
fx = x'*a*x + b*x+c;
cx = -1+x(1)^2+x(2)^2;
>> [x_optimal,Iterations]= lagrange_newton_method(fx,cx,[1;0; 0; 0; 0;
0],1e-4)
x_optimal =
   9.2884e-01
   3.7049e-01
  -8.4812e-01
  -3.1156e-01
  -5.4743e-01
   2.9993e-01
Iterations =
    50
```

APPENDIX

Appendix 1:

a) Initial code of Steepest Gradient Method:

```
% This code gives the minimization of a function by Steepest Descent Method. In this
%program alpha is updated in every step. The objective function is specified with
%initial values and tolerance. Gradient is calculated by Partial differentiation
%method.We get the Optimal point as the output.

clc
clear all
close all

% Function definition
syms x1 x2;
%f=-(sqrt((x1^2 +1)*(2*x2^2 + 1)))/(x1^2+x2^2+0.5);
%f = abs(x1-2) + abs(x2-2);
f = x1 - x2 + 2*x1^2 + 2*x1*x2 + x2^2;

% Initial values
x(1) = -4;
y(1) = -4;
err = 1e-2;
i = 1;

% Gradient Computation
df_dx = diff(f, x1);
df_dy = diff(f, x2);
grad = [subs(df_dx,[x1,x2], [x(1),y(1)]) subs(df_dy, [x1,x2], [x(1),y(1)])];

while(norm(grad)>err)

    %grad = [subs(df_dx,[f,x1], [x(i),y(i)]) subs(df_dy, [f,x2], [x(i),y(i)])];
    % New Search Direction
    s_j = -(grad);

    % Assigning alpha
    syms alpha; % Step size
    g = subs(f, [x1,x2], [x(i)+s_j(1)*alpha,y(i)+alpha*s_j(2)]);
    dg_dalpha = diff(g,alpha);
    alpha = solve(dg_dalpha, alpha);

    g = subs(f, [x1,x2], [x(i)+s_j(1)*alpha,y(i)+alpha*s_j(2)]);
    h = subs(f, [x1,x2], [x(i),y(i)]);
    if (g<h)
        % finding the next value
        x(i+1) = x(i)+alpha*s_j(1);
        y(i+1) = y(i)+alpha*s_j(2);
        i = i+1;
    end
    % Updating Gradient
    grad = [subs(df_dx,[x1,x2], [x(i),y(i)]) subs(df_dy, [x1,x2], [x(i),y(i)])];
end

% Plots
fcontour(f, 'Fill', 'On');
hold on;
plot(x,y,'*-r');
```

```
% Displaying results
%if (norm(s_j)<err)
    % when x_j satisfies the necessary condition for local minimum
 %   disp("Minimum point is obtained");
%end
fprintf('Number of Iterations for the cost function convergence: %d\n\n', i);
fprintf('Mininimum point: [%d,%d]\n\n', x(i), y(i));
```

---

b) Final code of Steepest Method:

1. SteepestDescent.m

```
% Steepest Descent Method for n variables
% Requires:  minfuncheck.m, linesearch.m, gradfunction.m and function file
% Parameters:  initial value of x, number of iterations, tolerance, the initial value
of stepsize,
% incremental value and number of trials
% eg: SteepestDescent('func',[0 0],30, 0.0001, 0,1 ,15)

function sol = SteepestDescent(functname,x_initial,iter,tol,lb,int,ntrials)
clf
err = 1.0e-08;
n = length(x_initial); % number of variables
if (n == 2)
%  plotting contours
    x1 = -5:0.1:5;
    x2 = -5:0.1:5;
    len_x1 = length(x1);
    len_x2 = length(x2);
    for i = 1:len_x1
    for j = 1:len_x2
        x1_x2 =[x1(i) x2(j)];
        fun(j,i) = feval(functname,x1_x2);
    end
    end
    c1 = contour(x1,x2,fun,[3.1 3.25 3.5 4 6 10 15 20 25],'k');
    grid
    xlabel('x1')
    ylabel('x2')
    title('Steepest Descent method');
end
%  Algorithm starts
x_i(1,:) = x_initial;
x = x_initial;
line_col = 'r';
opt_f(1) = feval(functname,x); % value of function at start
opt_x(1)=0;
s_j = -(gradfunction(functname,x)); % steepest descent

conv(1)=s_j*s_j';
for i = 1:iter-1

   x_o = linesearch(functname,tol,x, s_j,lb,int,ntrials);
   opt_x(i+1) = x_o(1);
   opt_f(i+1) = x_o(2);
   for k = 1:n
      x_i(i+1,k)=x_o(2+k);
      x(k)=x_o(2+k);
   end
   s_j = -(gradfunction(functname,x)); % steepest descent
   conv(i+1)=s_j*s_j';
```

```
   % lines in contour
   if (n == 2)
      line([x_i(i,1) x_i(i+1,1)],[x_i(i,2) x_i(i+1,2)],'LineWidth',2, 'Color',line_col)
   if strcmp(line_col,'r')
         line_col = 'k';
   else
         line_col = 'r';
   end
    pause(1)
   end

   if(conv(i+1)<= err)
       break;
   end

end
len=length(opt_x);
opt_pt=x_i(length(opt_x),:);

% Displaying Results
fprintf('\nNo. of iterations:  '),disp(len)
fprintf('\n - Optimal point , Value of function at Optimal point \nduring the
iterations\n')
disp([x_i opt_f'])
sol = [opt_pt opt_f(len)];
```

2. linesearch.m

```
% This code gives the line search technique using Golden Section Method
function ReturnValue =linesearch(functname,tol,x,s,lower_bound,intr,trials)
format compact;

% finding the upper bound
upper_val = minfuncheck(functname,x,s,lower_bound,intr,trials);
u_b=upper_val(1);

% if upper bound is close to lowbound reverse the direction of the search vector
if (u_b <= 1.0e-06)
   l_b = lower_bound;
   xL = x + l_b*s;
   f_lb =feval(functname,xL);
   ReturnValue =[l_b f_lb x];
   return
end

if (tol == 0)
    tol = 0.0001;
end

eps = tol/(u_b - lower_bound);
tau = 0.38197;
nmax = round(-2.078*log(eps));

l_b = lower_bound;
a1 = (1-tau)*l_b + tau*u_b;
x1 = x + a1*s;
fa1 = feval(functname,x1);
a2 = tau*l_b + (1 - tau)*u_b;
x2 = x + a2*s;
fa2 = feval(functname,x2);

for i = 1:nmax
```

```
    if fa1 >= fa2
    l_b = a1;
    a1 = a2;
    fa1 = fa2;
    a2 = tau*l_b + (1 - tau)*u_b;
    x2 = x + a2*s;
    fa2 = feval(functname,x2);
    else
       u_b = a2;
       a2 = a1;
       fa2 = fa1;
       a1 = (1-tau)*l_b + tau*u_b;
       x1 = x + a1*s;
       fa1 = feval(functname,x1);
    end
end
% returns the value
ReturnValue =[a1 fa1 x1];
```

3. minfuncheck.m

```
% This code gives the minimum of a function
function sol = minfuncheck(functname,x,s,initial_step,inc_step,ns)

format compact
if (ns ~= 0)
    ntrials = ns;
else
    ntrials = 10;
end

if (inc_step ~= 0)
    step = inc_step;
else
    step = 1;
end

% finds a value of function greater than or equal to the previous value
for i = 1:ntrials
   j = 0;
   del_a = j*step;
   a_old = initial_step + del_a;
   dx0 = a_old*s;
   x0 = x + dx0;
   f0 = feval(functname,x0);
   j = j+1;
   del_a = j*step;
   a_new = initial_step + del_a;
   dx1 = a_new*s;
   x1 = x + dx1;
   f1 = feval(functname,x1);
   f1s = f1;
   if f1 < f0
         for j = 2:ntrials
            a_new = initial_step + j*step;
            dx1 = a_new*s;
            x1 = x + dx1;
            f1 = feval(functname,x1);
            f1s = min(f1s,f1);
            if f1 > f1s
                sol = [a_new f1 x1];
               return;
            end
```

```
         end
         fprintf('\nFunction cannot be increased in ntrials')
        sol = [a_new f1 x1];
         return;
   else f1 >= f0;
      step = 0.5*step;
   end
end
sol =[initial_step f0 x0];
```

4. gradfunction.m

```
% This function gives the gradient computation by finite difference method
function Return = gradfunction(functname,x)
hstep = 0.001;
n = length(x);
f = feval(functname,x);

for i = 1:n
   xs = x;
   xs(i) = xs(i) + hstep;
   gradx(i)= (feval(functname,xs) -f)/hstep;
end
Return = gradx;
```

5.func.m

```
function sol = func(x)

% Part 1, 1st problem
% a = [9, 1, 7, 5, 4, 7; 1, 11, 4, 2, 7, 5; 7, 4, 13, 5, 0, 7; 5, 2, 5, 17, 1, 9; 4, 7,
0, 1, 21, 15; 7, 5, 7, 9, 15, 27];
% b = [1, 4, 5, 4, 2, 1];
% c = 5;
% sol = transpose([x(1);x(2);x(3);x(4);x(5);x(6)])*a*[x(1);x(2);x(3);x(4);x(5);x(6)]+
b*[x(1);x(2);x(3);x(4);x(5);x(6)]+c;

% Part 1, 2nd problem
%sol = -(sqrt((x(1)^2 +1)*(2*x(2)^2 + 1)))/(x(1)^2+x(2)^2+0.5);

% Part 1, 3rd problem
% b = [1,2];
% a = [12 ,3; 3, 10];
% sol = 1 + b*[x(1);x(2)] + 0.5*
transpose([x(1);x(2)])*a*[x(1);x(2)]+10*log(1+(x(1)^4))*sin(100*x(1))+10*log(1+(x(2)^4)
)*cos(100*x(2));

% Example 1
%sol =  x(1) - x(2) + 2*x(1)^2 + 2*x(1)*x(2) + x(2)^2;

% Example 2
%sol = x(1)^2+x(2)^2;

% Example 3
sol = abs(x(1)-2)+abs(x(2)-2);
```

Appendix 2:

1. ArmijoLineSearch.m

```
% This code is for Armijo line search which is implemeted with Steepest descent
algorithm.

% This code gives the minimization of a function by Steepest Descent Method. In this %
program alpha is updated by Armijo method. The objective function is specified with %
initial values and tolerance. Gradient is calculated by Partial differentiation %
method.We get the Optimal point as the output

% We need {fun_armijo and grad_armijo}.m files to run this program

function [argmin, iterations] = ArmijoLineSearch(fun_armijo,x0,sigma,beta, eps)
if (nargin == 2)
    sigma = 1e-4;
    beta = 0.5;
    eps = 1e-4;
elseif (beta>=1 || beta<=0 || eps <=0)
    disp('Wrong paramaters used beta should be Beta and sigma in (0,1), eps >0');
end

x = x0;
k = 0;
maxIterations = 1e8;
g = grad_armijo(x);

while norm(g) > eps
    g = grad_armijo(x);
    d = -g;
    l = 0;
    t = 0;
    while(fun_armijo(x+(beta^l)*d)>fun_armijo(x)+(beta^l)*sigma*g'*d)
        l = l+1;
    end
    t = beta^l;
    k = k+1;
    x = x+t*d;
    if k == maxIterations
        break;
    end
end

disp("Minimum point is obtained");
fprintf('Number of Iterations for the cost function convergence: %d\n\n', k);
fprintf('Mininimum point: \n');
disp(x(1))
disp(x(2))
```

2. fun_armijo.m

```
function [fun] = fun_armijo(x)
%fun = 2*x(1)^4+3*x(2)^4+2*x(1)^2+4*x(2)^2+x(1)*x(2)-3*x(1)-2*x(2);
%fun = x(1)^2 + x(2)^2;
fun = abs(x(1)-2)+abs(x(2)-2);
end
```

3. grad_armijo.m

```
function [grad] = grad_armijo(x)
syms x1 x2
%f = 2*x1^4+3*x2^4+2*x1^2+4*x2^2+x1*x2-3*x1-2*x2;
```

```
%f = x1^2 + x2^2;
v =[x1,x2];
df_dx = diff(fun_armijo(v), x1);
df_dy = diff(fun_armijo(v), x2);
grad = [subs(df_dx,[x1,x2], [x(1),x(1)]) subs(df_dy, [x1,x2], [x(1),x(2)])];
grad = grad';
end
```

Appendix 3:

a) Initial code of Conjugate Gradient Method:

```
% This code gives the minimization of a function by Conjugate Gradient Method. In this
% program alpha is updated in every step. The objective function is specified with %
initial values and tolerance. Gradient is calculated by Partial differentiation method.
% We get the Optimal point as the output.


clc
clear all
close all

% Function definition
syms x1 x2;
%f=-(sqrt((x1^2 +1)*(2*x2^2 + 1)))/(x1^2+x2^2+0.5);
f = (x1^2 + x2^2);
% f = abs(x1-2) + abs(x2-2);
%f = x1 - x2 + 2*x1^2 + 2*x1*x2 + x2^2;


% Initial values
x(1) = 4;
y(1) = 4;
err = 1e-2;
i = 1;
alpha = 0.01;


% Gradient Computation
df_dx = diff(f, x1);
df_dy = diff(f, x2);
grad = [subs(df_dx,[x1,x2], [x(1),y(1)]) subs(df_dy, [x1,x2], [x(1),y(1)])];
grad = grad';
s_j = -grad;


while(norm(s_j)>err)
    %s_j = -grad;
    % finding the next value
    x(i+1) = x(i)+alpha*s_j(1);
    y(i+1) = y(i)+alpha*s_j(2);
    grad_new=[subs(df_dx,[x1,x2], [x(i+1),y(i+1)]) subs(df_dy, [x1,x2],
[x(i+1),y(i+1)])];
    grad_new= grad_new';
    beta = ((grad_new- grad)' * grad_new)./ norm(grad)^2;
    s_j = -grad_new + beta * s_j;
    grad = grad_new;
    i = i+1;
end

% Plots
```

```
fcontour(f, 'Fill', 'On');
hold on;
plot(x,y,'*-r');

% Displaying results
fprintf('Number of Iterations for the cost function convergence: %d\n\n', i);
fprintf('Mininimum point: [%d,%d]\n\n', x(i), y(i));
```

b) Final code of Conjugate Gradient Method:

```
% Conjugate gradient for n variables
% Requires:  minfuncheck.m, linsearch.m, gradfunction.m and function file
% Parameters:  initial value of x, number of iterations, tolerance, the initial value
of stepsize,
% incremental value and number of trials
% eg: Conjugategradient('fun',[0 0],30, 0.0001, 0,1 ,15)

function sol = Conjugategradient(functname,x_initial,iter,tol,lb,int,ntrials)
clf
format short e
% convergence
err = 1.0e-04;
n = length(x_initial); %  number of variables
if (n == 2)

% Contour plot
    x1 = -5:0.1:5;
    x2 = -5:0.1:5;
    len_x1 = length(x1);
    len_x2 = length(x2);
    for i = 1:len_x1
        for j = 1:len_x2
            x1x2 =[x1(i) x2(j)];
            fun(j,i) = feval(functname,x1x2);
        end
    end
    c1 = contour(x1,x2,fun,[5 10 50 100 200 500 1000],'k');
     % [3.1 3.25 3.5 4 6 10 15 20 25],'k');
    grid
    xlabel('x1')
    ylabel('x2')
    title('Conjugate Gradient Method')
end

%Algorithm Starts
x_i(1,:) = x_initial;
x = x_initial;
line_col = 'r';
opt_f(1) = feval(functname,x); % value of function at start
opt_x(1)=0;
s_j = -(gradfunction(functname,x));
conv(1)=s_j(1)*s_j(1)';

for i = 1:iter-1
   % determine search direction
   if (i > 1)
      beta = grad*grad'/(s_j*s_j');
      s_j = -grad + beta*s_j;
   end
   x_o = linesearch(functname,tol,x, ...
      s_j,lb,int,ntrials);
   opt_x(i+1) = x_o(1);
```

```
    opt_f(i+1) = x_o(2);
    for k = 1:n
        x_i(i+1,k)=x_o(2+k);
        x(k)=x_o(2+k);
    end
    grad= (gradfunction(functname,x));
    conv(i+1)=grad*grad';

    % Line draw
    if (n == 2)
        line([x_i(i,1) x_i(i+1,1)],[x_i(i,2) x_i(i+1,2)],'LineWidth',2,'Color',line_col)
        if strcmp(line_col,'r')
            line_col = 'k';
        else
            line_col = 'r';
        end
        pause(1)
    end
    if(conv(i+1)<= err)
        break;
    end
end
len=length(opt_x);
opt_value=x_i(length(opt_x),:);

% Displaying Results
fprintf('\nNo. of iterations:  '),disp(len)
fprintf('\nOptimal point, Value of func at minimal pt \nduring the iterations\n')
disp([x_i opt_f'])
sol = [opt_value opt_f(len)];
```

Appendix 4:

a) Initial code of Secant Method:

```
% This code gives the minimization of a function by Secant  Method. In this program %
alpha is updated in every step. The objective function is specified with initial values
% and tolerance. Gradient is calculated by Partial differentiation method.We get the %
Optimal point as the output


clc
clear all
close all

% Function definition
syms x1 x2;
%f=-(sqrt((x1^2 +1)*(2*x2^2 + 1)))/(x1^2+x2^2+0.5);
f = (x1^2 + x2^2);
% f = abs(x1-2) + abs(x2-2);
%f = x1 - x2 + 2*x1^2 + 2*x1*x2 + x2^2;

% Initial values
x_old = 4;
y_old = 4;
err = 1e-4;
i = 1;
alpha = 0.1;

% Gradient Computation
df_dx = diff(f, x1);
```

```
df_dy = diff(f, x2);
grad = [subs(df_dx,[x1,x2], [x_old,y_old]) subs(df_dy, [x1,x2], [x_old,y_old])];
grad = grad';
H=eye(2);
s_j = -H* grad;

%loop
while(norm(s_j)>err)
    s_j = -H* grad;
    g = subs(f, [x1,x2], [x_old+s_j(1)*alpha,y_old+alpha*s_j(2)]);
    h = subs(f, [x1,x2], [x_old,y_old]);
    %if (g<h)
    % finding the next value
    x_new = x_old+alpha*s_j(1);
    y_new = y_old+alpha*s_j(2);
   % end
    del_x = x_new - x_old;
    del_y = y_new - y_old;
    del_xy = [del_x, del_x]';
    %del_xy =  alpha* s_j;
    grad_new=[subs(df_dx,[x1,x2], [x_new,y_new]) subs(df_dy, [x1,x2], [x_new,y_new])];
    grad_new = grad_new';
    del_g = grad_new - grad;
    H = H + ((del_xy* del_xy')./(del_xy' * del_g)) - (((H* del_g) * (H *
del_g)')./(del_g' * H * del_g));
    x_old = x_new;
    y_old= y_new;
    grad = grad_new;
    i=i+1;
end

% Plots
fcontour(f, 'Fill', 'On');
hold on;
plot(x_old,y_old,'*-r');

% Displaying results
%if (norm(s_j)<err)
    % when x_j satisfies the necessary condition for local minimum
 %   disp("Minimum point is obtained");
%end
fprintf('Number of Iterations for the cost function convergence: %d\n\n', i);
fprintf('Mininimum point: [%d,%d]\n\n', x_old, y_old);
```

b) Final code of Secant Method

```
% This is the code for secant algorithm with Davidon Fletcher Powell(DFP) Method
% Requires:  minfuncheck.m, linsearch.m, gradfunction.m and function file
% Parameters:  initial value of x, number of iterations, tolerance, the initial value
of stepsize,
% incremental value and number of trials
% eg: secant_DFP('fun',[0 0],30, 0.0001, 0,1 ,15)

function sol = secant_DFP(functname,x_initial,iter,tol,lb,int,ntrials)

clf
% convergence
err = 1.0e-04;
n = length(x_initial);% number of variables
if (n == 2)
%Contour plot
    x1 = -5:0.1:5;
```

```
    x2 = -5:0.1:5;
    len_x1 = length(x1);
    len_x2 = length(x2);
    for i = 1:len_x1
        for j = 1:len_x2
            x1_x2 =[x1(i) x2(j)];
            fun(j,i) = feval(functname,x1_x2);
        end
    end
    c1 = contour(x1,x2,fun,[3.1 3.25 3.5 4 6 10 15 20 25],'k');
    grid
    xlabel('x1')
    ylabel('x2')
    title('Secant Method by Davidon-Fletcher-Powell (DFP)')
end

% Algorithm
x_i(1,:) = x_initial;
x = x_initial;
line_col = 'r';
opt_f(1) = feval(functname,x);
opt_x(1)=0;
grad = (gradfunction(functname,x));
H = eye(n);  % Initial H is identity matrix function
conv(1)=grad*grad';

for i = 1:iter-1
   % determine search direction
   s = (-H*grad')';
   x_o = linesearch(functname,tol,x,s,lb,int,ntrials);
   opt_x(i+1) = x_o(1);
   opt_f(i+1) = x_o(2);
   for k = 1:n
      x_i(i+1,k)=x_o(2+k);
      x(k)=x_o(2+k);
   end
   grad= (gradfunction(functname,x));
   conv(i+1)=grad*grad';

   % Draw line
   if (n == 2)
      line([x_i(i,1) x_i(i+1,1)],[x_i(i,2) x_i(i+1,2)],'LineWidth',2,'Color',line_col)
      if strcmp(line_col,'r')
         line_col = 'k';
      else
         line_col = 'r';
      end
      pause(1)
   end

if(conv(i+1)<= err)
    break;
end
del_x = (x - x_i(i,:))';
del_g = (grad -gradfunction(functname,x_i(i,:)))';
H = H + ((del_x*del_x')/(del_x'*del_g)) -
(((H*del_g)*(H*del_g)')/(del_g'*H*del_g));%DFP
end

len=length(opt_x);
opt_value=x_i(length(opt_x),:);

%Displaying results
fprintf('\nNo. of iterations:  '),disp(len) % Comment for ALM
```

```
fprintf('\nOptimal point, Value of func at minimal pt \nduring the iterations\n')  %
Comment for ALM
disp([x_i opt_f'])  % Comment for ALM
sol = [opt_value opt_f(len)];
```

Appendix 5:

a) Barrier and penalty

1. Code for both barrier and penalty

```
% This code gives the method for penalty and barrier functions
% Reference text book "Optimization for Engineering Design" by Kalyanboy
% This code needs func_pb.m and runevolution.m files

eps1 = 10e-5;
eps2 = 10e-5;
rg_initial = 0.1;           % initial Penalty parameter
x_initial =[rand*2 rand*2];
% x0 = [0 0];
c = input('Enter 0 for Barrier method, 1 for penalty Method = ');
 if c==0
     c = rand;
 else
     c = rand*1.9;
 end
 t = 1;
 x(t,:)= x_initial;
 rg(t)= rg_initial;
 x1 = x(1);
 x2 = x(2);
 constraint = ((x1-5).^2 +x2.^2 -26); % change the constraint here for every example
 if (constraint < 0)
     for t = 1
         x1 = x(t,1);
         x2 = x(t,2);
         constraint = (x1-x2*x2);
         k = rg.*constraint^2;
         F = func_pb(x(t,:),0);
         P(1,t) = func_pb(x(t,:),rg(1,t));
         G(t,:) = runevolution(x(t,:),rg(1,t));     % to perform unconstrained search
for given R
         x(t+1,:) = G(t,:);
         t = t+1;
         P(1,t) = func_pb(G(t-1,:),rg(1,t-1));
         err = abs(P(1,t));
         rg(1,t) = c*rg(1,t-1);
         x1 = x(t,1);
         x2 = x(t,2);
         constraint = ((x1-5).^2 +x2.^2 -26);

     end
 else
     x1 = x(t,1);
     x2 = x(t,2);
     k = 0;
     F = func_pb(x(t,:),0);
     P(1,t) = func_pb(x(t,:),0);
     G(t,:) = runevolution(x(t,:),0);
     t = t+1;
     P(1,t) = func_pb(G(t-1,:),0);
     err = abs(P(1,t));
```

```
    rg(1,t) = c*rg(1,t-1);
    x(t,:) = G(t-1,:);
 end

 a(:,1) = G(t-1,:)';

 while err>eps2
    G(t,:) = runevolution(x(t,:),rg(1,t));
    x(t+1,:) = G(t,:);
    t = t+1;
    rg(1,t) = c*rg(1,t-1);
    P(1,t) = func_pb(G(t-1,:),rg(1,t-1));
    err = abs(P(1,t)-P(1,t-1));
      x1 = x(t,1);
      x2 = x(t,2);
      constraint = ((x1-5).^2 +x2.^2 -26);

    a(:,1) = G(t-1,:)';
    if abs(constraint) <eps1
        break;
    elseif (abs(x(t,1)-x(t-1,1))<10e-5 && abs(x(t,2)-x(t-1,2))<10e-5)
        break;
    end

 end
fval = func_pb(x(t,:),0);
fprintf('The minimum point is (%4f,%4f) with a function value of %.4f  \n',a,fval)
```

2. runevolution.m

```
% Box's evolutionary optimization is a simple optimization technique developed by G. E.
P. Box1
% in 1957. The algorithm requires (2N +1) points, of which 2N
% are corner points of an N-dimensional hypercube2
% centred on the other point.
% All (2N + 1) function values are compared and the best point is identified.
% In the next iteration, another hypercube is formed around this best point. If
% at any iteration, an improved point is not found, the size of the hypercube is
% reduced. This process continues until the hypercube becomes very small.
 % Referred from optimization for engineering design by kalyanmoy deb

function [xnew] = runevolution(x,R)

del = [2,2];
x_initial =x;
tol = 10^-3;
y = x_initial;
del1 = (sqrt(sum(del.^2)));
j=1;

while del1 > tol

    z1 = x_initial - del./2;
    z2 = x_initial + ([-(del(1)),del(2)])./2;
    z3 = x_initial + ([del(1),-(del(2))])./2;
    z4 = x_initial + del./2;
    x = [x_initial;z1;z2;z3;z4]';
    f = zeros (1,5);
    for i=  1:5
        f(i) = func_pb(x(:,i),R);
    end
    [~ , idx] = min(f(:,:));
```

```
    y = x(:,idx);
    i=1;
    if (y == x_initial')
        del = del./2;
    else
        x_initial = y;
        x_initial =x_initial';
    end
    del1 = (sqrt(sum(del.^2)));
    j = j+1;
end
xnew = y;
```

3. **func_pb.m**

```
% function used in penalty and barrier function method
function P = func_pb(x,R)
     x1 = x(1);
     x2 = x(2);

     %f = R.*(((x1-5).^2 +x2.^2 -26).^2);       % R * constraint function
     %P = (x1.^2 +x2 -11).^2 + (x1 +x2.^2-7).^2 +f;  % Objective function

     %f = R.*(-x1*x2);       % R * constraint function
     %P = (-x1+x2^2+1).^2 + (x1 +x2).^2 +f;  % Objective function

     f =  R.*(x1-x2*x2);
     P =  abs(x1-2)+abs(x2-2)+f;
end
```

b.) Penalty method for equality and inequality functions

1. func_penalty.m

```
% This function calculates the unconstrained fuction for Penalty method
function sol = func_penalty(x)

global m leq
global rh rg
sol = f_penalty(x);

if leq > 0
   hval = hfun_penalty(x);
   sol = sol +  rh*(hval*hval);
end

if m > 0
   gval = gfun_penalty(x);
   sol = sol + rg*(max(0,gval))^2;
end
```

2. f_penalty.m

```
function sol = f_penalty(x)
% objective function
sol = x(1)^4 - 2*x(1)*x(1)*x(2) +x(1)*x(1) + x(1)*x(2)*x(2) -2*x(1) + 4;
%sol = log(x(1))-x(2); %(Part 2, 3rd problem)
%sol = abs(x(1)-2)+abs(x(2)-2); %(Part 2, 1st problem)
```

3. gfun_penalty.m

```
function sol = gfun_penalty(x)

% Inequality constraint

sol = [(0.25*x(1)*x(1) + 0.75*x(2)*x(2) -1)];

%sol = x(1)-1; %(Part 2, 3rd problem)

%sol = x(1)-x(2)*x(2); %(Part 2, 1st problem)
```

4. hfun_penalty.m

```
function sol = hfun_penalty(x)

% the equality constraint vector

sol = [(x(1)*x(1) + x(2)*x(2) -2)];

%sol = [(x(1)*x(1) + x(2)*x(2) -4)]; %(Part 2, 3rd problem)

%sol = x(1)*x(1) + x(2)*x(2) -1; %(Part 2, 1st problem)
```

5. Code for Penalty method

```
% This code is for Penalty method invovling one inequality
% and one equality constraint. We use secant method for minimization
% We need {f_penalty, gfun_penalty, hfun_penalty, func_penalty, secant_DFP, linesearch,
% minfuncheck}.m files to run this program

clear all
clc

% Global values to be used in other functions
global n m leq
global rh rg

% Initialising the values
format compact
format short e
warning off
x_old = [3 2]; % Initial value for the problem
Xmin = [-5 -5]; % less than the initial value (~ 3 times)
Xmax = [5 5]; % more than the  initial value (~ 3 times)
n = length(x_old);
leq = 1; % number of equality constraints
m = 1; % number of inequality constraints
iter_penalty = 20; % limit of iter used in this program
iter_secant = 20; % limit of iter used in secant method

%Following are the variables used in secant method
tol = 1.0e-08;
lb = 0;
int = 1.0;
ntrials = 20;
epsilon = 1.0e-016; % error value

rh = 1;
rg = 1;
ch = 10; % scaling factor for lamda
cg = 10; % scaling factor for gamma
```

```
% calculating and storing the values
f_old = f_penalty(x_old);
g_old = gfun_penalty(x_old);
h_old = hfun_penalty(x_old);

% store values
iter = 1;
x_s(iter,:) = x_old;
rh_s(iter) = rh;
rg_s(iter) = rg;
f_s(iter) = f_old;
uncon_f_s(iter)=func_penalty(x_old);

% method starts here
x_cal = x_old;
count = 0;

for loop = 1:iter_penalty

   count = count + 1;
   x_initial = x_cal ;
   functname = char('func_penalty');
   fresult = secant_DFP(functname,x_initial,iter_secant,tol,lb,int,ntrials);
   for ix = 1:n
      xnew(ix) = fresult(ix);
   end
   Fnew = fresult(n+1);
   fend = f_penalty(xnew);
   iter = iter + 1;
   x_s(iter,:) = xnew;
   f_s(iter) = fend;
   uncon_f_s(iter) = Fnew;
   gVend = gfun_penalty(xnew);
   hVend = hfun_penalty(xnew);
   x_initial = xnew;

   % convergence

   fdiff = Fnew - uncon_f_s(iter - 1);
   % stop if iterations are exceeded
   if loop == iter_penalty
   fprintf('maximumum number of iterations reached : %6i \n',iter_penalty);
   fprintf('\n The values for x  : \n');
   disp(x_s);
  break;
   end

   % convergence in changes in x
   xdiff = (xnew-x_cal)*(xnew-x_cal)';
   if xdiff < epsilon
       fprintf('\n The values for x are : \n');
       disp(x_s);
       fprintf('\n Optimal value x :\n');
       disp(xnew);
  break;
   end

   if ((fdiff)^2 < epsilon)
       fprintf('Convergence in f : % 14.3E  reached in %6i iterations \n', ...
           abs(fdiff), loop);
       fprintf('\n The values for x :\n');
       disp(x_s);
```

```
        fprintf('\n Optimal value x :\n');
        disp(xnew);
       break;
   else
        fprintf('\n Penalty iteration: '),disp(count)
        fprintf('Value of (X) :  '),disp(x_initial);
   end

   x_cal = x_initial;
   rh = rh*ch;
   rg = rg*cg;
   rh_s(iter) = rh;
   rg_s(iter) = rg;
end
```

Appendix 6:

1. Code for Augumented Lagrangian method

```
% This code is for Augumented Lagrangian method invovling one inequality
% and one equality constraint. We use secant method for minimization
% We need {f_alm, gfun_alm, hfun_alm, func_alm, secant_DFP, linesearch,
% minfuncheck}.m files to run this program

clear all
clc
% Global values to be used in other functions
global lamda beta
global n m leq
global rh rg

% Initialising the values
format compact
format short e
warning off
x_old = [3 2]; % Initial value for the problem
Xmin = [-5 -5]; % less than the initial value (~ 3 times)
Xmax = [5 5]; % more than the  initial value (~ 3 times)
n = length(x_old);
leq = 1; % number of equality constraints
lamda = 1; % default value used in equality constraint
m = 1; % number of inequality constraints
beta = 1; % default value used in inequality constraint
iter_alm = 20; % limit of iter used in this program
iter_secant = 20; % limit of iter used in secant method

%Following are the variables used in secant method
tol = 1.0e-08;
lb = 0;
int = 1.0;
ntrials = 20;
epsilon = 1.0e-016; % error value

ch = 10; % scaling factor for lamda
cg = 10; % scaling factor for gamma

% calculating and storing the values
f_old = f_alm(x_old);

g_old = gfun_alm(x_old);
```

```
g_err = g_old*g_old';
gErr(1) = g_err;
rg = f_old/g_err; %Penalty factor

h_old = hfun_alm(x_old);
h_err = h_old*h_old';
hErr(1) = h_err;
rh = f_old/h_err; %Penalty factor

% store values
iter = 1;
x_s(iter,:) = x_old;
lamda_s(iter,:) =lamda;
beta_s(iter,:)= beta;
rh_s(iter) = rh;
rg_s(iter) = rg;
f_s(iter) = f_old;
uncon_f_s(iter)=func_alm(x_old);

% method starts here
x_cal = x_old;
count = 0;

for loop = 1:iter_alm

   count = count + 1;
   x_initial = x_cal ;

   functname = char('func_alm');
   fresult = secant_DFP(functname,x_initial,iter_secant,tol,lb,int,ntrials);

   for ix = 1:n
      xnew(ix) = fresult(ix);
   end
   Fnew = fresult(n+1);

   fend = f_alm(xnew);

   iter = iter + 1;
   x_s(iter,:) = xnew;
   f_s(iter) = fend;
   uncon_f_s(iter) = Fnew;

   gVend = gfun_alm(xnew);
   g_err = gVend*gVend';
   gErr(iter) = g_err;

   hVend = hfun_alm(xnew);
   h_err = hVend*hVend';
   hErr(iter) = h_err;

   x_initial = xnew;

   % convergence

   fdiff = Fnew - uncon_f_s(iter - 1);
   gdiff = gErr(iter) - gErr(iter -1);
   hdiff = hErr(iter) - hErr(iter -1);

   % stop if iterations are exceeded
   if loop == iter_alm
   fprintf('maximumum number of iterations reached : %6i \n',iter_alm);
```

```
    fprintf('\n The values for x  : \n');
    disp(x_s);
   break;
    end

    % convergence in changes in x
    xdiff = (xnew-x_cal)*(xnew-x_cal)';
    if xdiff < epsilon
        fprintf('\n The values for x are : \n');
        disp(x_s);
        fprintf('\n Optimal value x :\n');
        disp(xnew);
   break;
    end

    if abs(fdiff) < epsilon && abs(gdiff) <epsilon && abs(hdiff) < epsilon
        fprintf('Convergence in f : % 14.3E  reached in %6i iterations \n', ...
            abs(fdiff), loop);
        fprintf('\n The values for x :\n');
        disp(x_s);
        fprintf('\n Optimal value x :\n');
        disp(xnew);
       break;
    else
        fprintf('\n ALM iteration: '),disp(count)
        fprintf('Value of (X) :  '),disp(x_initial);
    end

    x_cal = x_initial;
     lamda = lamda + 2*rh*hVend;
    beta = beta + 2*rg*(gVend-beta/(2*rg));
    lamda_s(iter,:) =lamda;
    beta_s(iter,:)= beta;

    rh = rh*ch;
    rg = rg*cg;
    rh_s(iter) = rh;
    rg_s(iter) = rg;

end
```

2. func_alm.m

```
function sol = func_alm(x)
%
% calculates the unconstrained fuction for ALM method
%
global lamda beta
global m leq
global rh rg
sol = f_alm(x);

if leq > 0
   hval = hfun_alm(x);
   sol = sol + lamda*hval'+ rh*(hval*hval');
end

if m > 0
   gval = gfun_alm(x);
   for j = 1:m
      g(j) = max(gval(j),-beta(j)/(2*rg));
   end
   sol = sol + beta*g'+ rg*(g*g');
```

```
end
```

3. f_alm.m

```
function sol = f_alm(x)
% objective function
%sol = x(1)^4 - 2*x(1)*x(1)*x(2) +x(1)*x(1) + x(1)*x(2)*x(2) -2*x(1) + 4;
%sol = log(x(1))-x(2); %(Part 2, 3rd problem)
sol = abs(x(1)-2)+abs(x(2)-2); %(Part 2, 1st problem)
```

4. gfun_alm.m

```
function sol = gfun_alm(x)

% Inequality constraint

%sol = [(0.25*x(1)*x(1) + 0.75*x(2)*x(2) -1)];
%sol = x(1)-1; %(Part 2, 3rd problem)
sol = x(1)-x(2)*x(2); %(Part 2, 1st problem)
```

5. hfun_alm.m

```
function sol = hfun_alm(x)

% the equality constraint vector

%sol = [(x(1)*x(1) + x(2)*x(2) -2)];
%sol = [(x(1)*x(1) + x(2)*x(2) -4)]; %(Part 2, 3rd problem)
sol = x(1)*x(1) + x(2)*x(2) -1; %(Part 2, 1st problem)
```

Appendix 7:

```
% This function uses Lagrange Newton method to calculate the optimal point
% of a function.
% Only Equality constraints are taken into account.

% The method to give the input is as follows.
% Run the following in command window.

% x = sym('x',[2,1]); (example 1)
% a = 1;
% b = [1;2];
% c = [12,3;3,10];
% fx = a + (b'*x)+(0.5*x'*c*x);
% cx = -1+x(1)^2+x(2)^2;
% [xhat1,i1]= Lagrange_Newton_Eq(fx,cx,[1;0;0;0;0;0],1e-4)

% x = sym('x',[6,1]); (example 2)
% a = [9, 1, 7, 5, 4, 7; 1, 11, 4, 2, 7, 5; 7, 4, 13, 5, 0, 7; 5, 2, 5, 17, 1, 9; 4, 7,
0, 1, 21, 15; 7, 5, 7, 9, 15, 27];
% b = [1, 4, 5, 4, 2, 1];
% c = 5;
% fx = x'*a*x + b*x+c;
% cx = -1+x(1)^2+x(2)^2;

% [xhat1,i1]= Lagrange_Newton_Eq(fx,cx,[1;0],1e-4)
% where fx is the function we want to optimize, cx is constraints, initial
```

```
% point and tolerance

function [xold,Iterations] = Lagrange_Newton_Eq(func,constraint,x_initial,eps)

n = length(x_initial);
m = length(constraint);
Iterations = 0; % Initialization
x = sym('x',[n,1]);
A = sym('A',[n,m]);
lambda = ones(m,1);
xold = x_initial;
y = zeros(n,n);
for i = 1:m
    % Calculation of A and lambda
    A(:,i) = gradient(constraint(i),x);
    y = y + lambda(i)*hessian(constraint(i),x);
end
% Calculating Wx
Wx = hessian(func,x) - y;
% It can be viewed to deliver a K-T point
% Expression for L
L = [Wx,-A;-A',zeros(m,m)];
% Y
Y = [-gradient(func,x);constraint];
% Stopping conditions
LxGrad = gradient(func,x) - sum(lambda.*gradient(constraint,x));
LyGrad = constraint;
while (((norm(subs(LxGrad,x,xold)) > eps) || (norm(subs(LyGrad,x,xold)) > eps)) &&
(Iterations<50)) % Start iteration
    % Calculate deltax and lambda
    dxdl = inv(double(subs(L,x,xold)))*double(subs(Y,x,xold));
    % Update the variables
    delta = dxdl(1:n,1);
    lambdanew = dxdl(n+1:length(dxdl),1);
    xnew = xold + delta;
    xold = xnew;
    lambda = lambdanew;
    y = zeros(n,n);
    for i = 1:m
        A(:,i) = gradient(constraint(i),x);
        y = y + lambda(i)*hessian(constraint(i),x);
    end
    Wx = hessian(func,x) - y;
    L = [Wx,-A;-A',zeros(m,m)];
    Y = [-gradient(func,x);constraint];
    LxGrad = gradient(func,x) - sum(lambda.*gradient(constraint,x));
    Iterations = Iterations+1;
end
end
```
