Optimization in Control and Robotics
Lab 1. Continuous Optimization
Report

Juan Gallostra Acín

October 2016

1 Fitting a line

1.1 Plot these 8 points using the function `plot()`.

```
n=8;                                                 % # points
P=[3 0; 5 -2; 2 1; 0 -4; -3 5; 2 0; 0 3; 8 -5];    % x y coordinates of each point
plot(P(:,1),P(:,2),'ro')
axis([-5,10,-5,5])
```

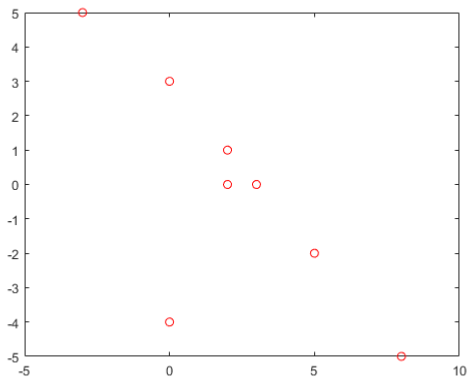


Figure 1.1: Plot of the given points in the cartesian plane.

1.2 Write the MATLAB script that allows to solve, by means of a LP, the problem of finding the line $y = a \cdot x + b$ minimizing $\sum_{i=1}^{n} |ax_i + b - y_i|$. Plot this line in the same figure.

In order to solve the problem using an LP we must define the problem in terms of an LP. This is, the objective function to be minimized and the constraints the variables are subject to. In this particular case the objective function can be expressed as:

$$min \sum_{i=1}^{n} e_i$$

Subject to:

$$a \cdot x_i + b - e_i \leq y_i$$

$$-a \cdot x_i - b - e_i \leq -y_i$$

Where we have 10 variables {a, b, e_1,..., e_8} and 2 inequality constraints for each point, resulting in a total of 16 inequality constraints. This can be defined in MATLAB as:

```
f = [0 0 ones(1,n)];                                      % obj. function
A = [P(:,1),ones(n,1),-eye(n);-P(:,1),-ones(n,1),-eye(n)];   % constraints
b = [P(:,2);-P(:,2)];                                     % b : Ax<=b
```

The values of `f`, `b` and `A` (1x10, 16x1, 10x16 matrices respectively) are:

```
f  =   0    0    1    1    1    1    1    1    1    1

b' =   0   -2    1   -4    5    0    3   -5    0    2   -1    4   -5    0   -3    5
```

```
A =
     3     1    -1     0     0     0     0     0     0     0
     5     1     0    -1     0     0     0     0     0     0
     2     1     0     0    -1     0     0     0     0     0
     0     1     0     0     0    -1     0     0     0     0
    -3     1     0     0     0     0    -1     0     0     0
     2     1     0     0     0     0     0    -1     0     0
     0     1     0     0     0     0     0     0    -1     0
     8     1     0     0     0     0     0     0     0    -1
    -3    -1    -1     0     0     0     0     0     0     0
    -5    -1     0    -1     0     0     0     0     0     0
    -2    -1     0     0    -1     0     0     0     0     0
     0    -1     0     0     0    -1     0     0     0     0
     3    -1     0     0     0     0    -1     0     0     0
    -2    -1     0     0     0     0     0    -1     0     0
     0    -1     0     0     0     0     0     0    -1     0
    -8    -1     0     0     0     0     0     0     0    -1
```

Once the objective function and the constraints have been defined the function `linprog()` should be called to solve the problem. The code to solve the problem and plot the obtained line is as follows:

```
[x,fval,exitflag] = linprog(f,A,b);      % solve linear program
hold on;                                 % plot resulting line in the same figure
a=x(1);b=x(2);
p=-5:0.1:10;plot(p,a*p+b);
```

The output of running the code above is:

```
Optimization terminated.

x' = -0.8750  2.3750  0.2500 -0.0000  0.3750   6.3750  -0.0000  0.6250  0.6250  0.3750

fval = 8.6250

exitflag = 1
```

Where the two first numbers of vector x are the values of a and b and the rest correspond to the absolute values of the error between each point and the computed fitting line. Thus, the equation of the fitted line is $y = -0.875 \cdot x + 2.375$.

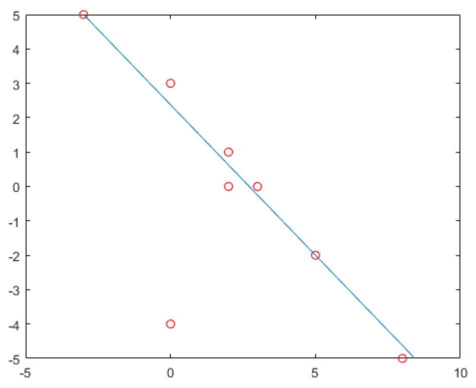


Figure 1.2: Line obtained from the resolution of the linear program

1.3 Find the line obtained with the least squares method. Plot this line in the same figure and compare it with the previous line.

The least squared method consists of minimizing $f(a,b) = \sum_i (ax_i + b - y_i)^2$ instead of $f(a,b) = \sum_{i=1}^{n} |ax_i + b - y_i|$. We can obtain the closed form of a and b by calculating the stationary point of f:

$$f(a,b) = \sum_i (ax_i + b - y_i)^2$$

$$\frac{\partial f}{\partial a} = 2 \cdot \left(a \cdot \sum_i {x_i}^2 + b \cdot \sum_i x_i - \sum_i x_i y_i \right) = 0 \quad \textit{and} \quad \frac{\partial f}{\partial b} = 2 \cdot \left(a \cdot \sum_i x_i + b \cdot n - \sum_i y_i \right) = 0$$

$$\begin{pmatrix} \sum_i {x_i}^2 & \sum_i x_i \\ \sum_i x_i & n \end{pmatrix} \cdot \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_i x_i y_i \\ \sum_i y_i \end{pmatrix} \Rightarrow \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_i {x_i}^2 & \sum_i x_i \\ \sum_i x_i & n \end{pmatrix}^{-1} \cdot \begin{pmatrix} \sum_i x_i y_i \\ \sum_i y_i \end{pmatrix}$$

As it is a convex problem, it is not necessary to check the second order condition. The code which calculates a and b using the formula just derived is:

```matlab
% least squares linear system resolution
X = sum(P(:,1));Y = sum(P(:,2));X2 = sum(P(:,1).^2);XY= sum(P(:,1).*P(:,2));
C=[X2 , X ; X , n]^-1 * [XY ; Y];
a=C(1);b=C(2); plot(p,a*p+b,'g')
```

Furthermore, MATLAB offers the user the function `polyfit()`, which calculates these two coefficients directly from the set of points.

```matlab
% least squares using polyfit() function
C = polyfit(P(:,1), P(:,2),1);
a = C(1);b=C(2);
plot(p,a*p+b,'g')
```

In the next figure we can compare the two lines obtained. The blue line corresponds to the least absolute value and the green line corresponds to the least square. It can be noticed that the first one (absolute value) is less sensitive to the outlier point (0,4). This is due to the fact that in the least squares method the value of each error is squared, which results in prioritizing the minimization of the biggest errors.

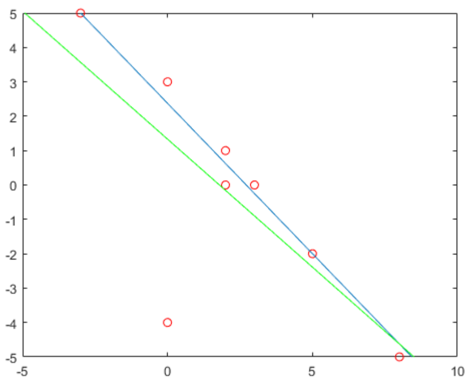


Figure 1.3: Comparison between the two obtained lines.

2 Largest Disk in a Convex Polygon

2.1 Plot these 5 points using function `plot()` and check that these points are consecutive vertices of a convex polygon.

```
P=[0 5; 1 2; 5 3; 4 7;  2 8];                  % x y coordinates of each point
figure
plot(P(:,1),P(:,2),'ko');                      % plot points
axis([-1,10,-1,10]);axis('equal');hold on;
Q=[P;P(1,:)];                                  % check if they are consecutive vertices
quiver(Q(1:5,1),Q(1:5,2),Q(2:6,1)-Q(1:5,1),Q(2:6,2)-Q(1:5,2));
```

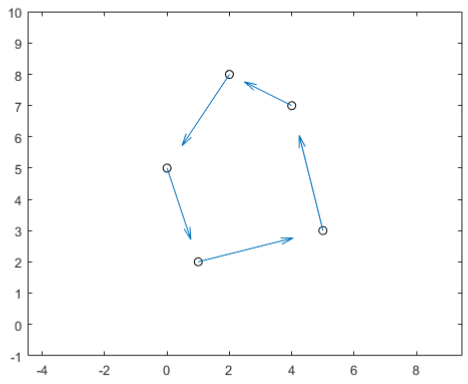


Figure 2.1: Plot of the 5 given points in the cartesian plane and verification that the points are consecutive vertices.

To check if the points are consecutive vertices of a convex polygon an arrow has been drawn from each point to the next one in the list of points. It can then be seen by visual inspection that the points are consecutive vertices and that the polygon they form is convex.

2.2 Write the code to obtain the 5 lines $y = a_i s + b_i$, $i = 1, ..., 5$ that is, obtain A and B such as the line l_i is $y = A(i)x + B(i)$. Draw the five lines in the same figure.

Any side i of the polygon determined by the set of points P is defined by the segment of line l_i that goes from P_i to P_{i+1}. This means that line $l_i : y = a_i s + b_i$ contains both $P_i = (x_i, y_i)$ and $P_{i+1} = (x_{i+1}, y_{i+1})$. This enables us to find A and B by solving the next linear system for $i = 1, ..., 5$, where $a_i = A(i)$ and $b_i = B(i)$:

$$\begin{pmatrix} x_i & 1 \\ x_{i+1} & 1 \end{pmatrix} \cdot \begin{pmatrix} a_i \\ b_i \end{pmatrix} = \begin{pmatrix} y_i \\ y_{i+1} \end{pmatrix} \Rightarrow \begin{pmatrix} a_i \\ b_i \end{pmatrix} = \begin{pmatrix} x_i & 1 \\ x_{i+1} & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} y_i \\ y_{i+1} \end{pmatrix}$$

Where x_i, y_i are the x,y coordinates of the point in P that is located in row i, P_i. When $i = 5$, $i + 1$ is set to 1 in order to compute the parametres of the line connecting the last and first point.

```
figure
plot(P(:,1),P(:,2),'ko');axis([-1,10,-1,10]);axis('equal');hold on;
N=5;P=[P;P(1,1:2)];                            % set P(6,:) to P(1,:)
A=zeros(N,1);B=zeros(N,1);                     % line computation
for i=1:N
   a=[P(i:i+1,1),ones(2,1)];b=P(i:i+1,2);
   x=a\b; A(i)=x(1); B(i)=x(2);
end
p=-2:0.1:10;                                   % draw lines
for i=1:5; plot(p,A(i).*p+B(i),'b'); end
```

The values of A and B are:

```
A' = -3.0000   0.2500  -4.0000  -0.5000   1.5000

B' = 5.0000   1.7500  23.0000   9.0000   5.0000
```

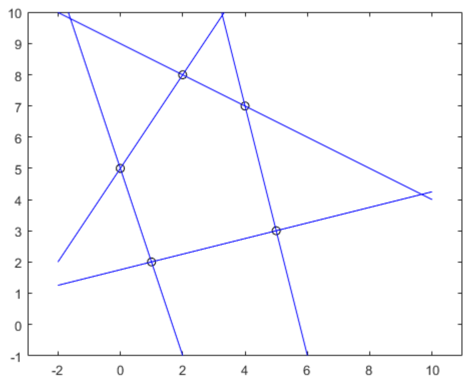


Figure 2.2: Plot of the obtained lines.

2.3 Define the three dimensional vector f associated to the objective function and matrix C and d associated to the inequality constraints $C \cdot x \leq d$ and solve the linear programming problem.

It is clear that the three unknown variables are the coordinates of the center of the disk, s_1 and s_2 and its radius r, and that the objective is to maximize the radius. This results in the objective function of the LP being:

$$max \;\; g(s_1, s_2, r) = 0 \cdot s_1 + 0 \cdot s_2 + 1 \cdot r \Rightarrow min \;\; f(s_1, s_2, r) = 0 \cdot s_1 + 0 \cdot s_2 - 1 \cdot r = -r$$

Subject to the value of the distance from the center s to any of the sides being at least r:

$$a_i \cdot s_1 - s_2 + \sqrt{a_i^2 + 1} \cdot r \leq -b_i \;\; i = 1, 2, ..., k \;\; (for\ lines\ below\ s)$$

$$-a_i \cdot s_1 + s_2 + \sqrt{a_i^2 + 1} \cdot r \leq b_i \;\; i = k+1, k+2, ..., n \;\; (for\ lines\ above\ s)$$

One way to implement and solve this Linear Program in MATLAB is:

```
f=[0 0 -1];                           % objective function

C=zeros(N,3);d=zeros(N,1);            % constraints such that Cx <= d
k=2;
for i=1:k                             % lines below
    C(i,:)=[A(i);-1;sqrt(A(i)^2+1)];
    d(i)=-B(i);
end
for i=k+1:N                           % lines above
    C(i,:)=[-A(i);1;sqrt(A(i)^2+1)];
    d(i)=B(i);
end

[x,fval,exitflag] = linprog(f,C,d)    % solve linear program
```

Where the values of C and d are:

```
C =
   -3.0000   -1.0000    3.1623
    0.2500   -1.0000    1.0308
    4.0000    1.0000    4.1231
    0.5000    1.0000    1.1180
   -1.5000    1.0000    1.8028

d' =  -5.0000   -1.7500   23.0000    9.0000    5.0000
```

And the result of solving the LP is:

```
x' =   2.3675   4.6830   2.1457            % x' = s1 s2 r

fval = -2.1457

exitflag = 1
```

The solution to the stated problem is contained in vector x such that $x(1) = s_1 = x_{center} = 2.3675$, $x(2) = s_2 = y_{center} = 4.6830$ and $x(3) = r = 2.1457$.

2.4 In vector x obtained in the last line of code there are the center and the radius. Plot this circle in the same figure and check that it is the largest circle inscribed in the polygon.

The code to plot the circle obtained as a solution to the LP is:

```
theta=linspace(0,2*pi,300);                       % plot parametrized circle
plot(x(3)*cos(theta)+x(1), x(3)*sin(theta)+x(2),'r')
```

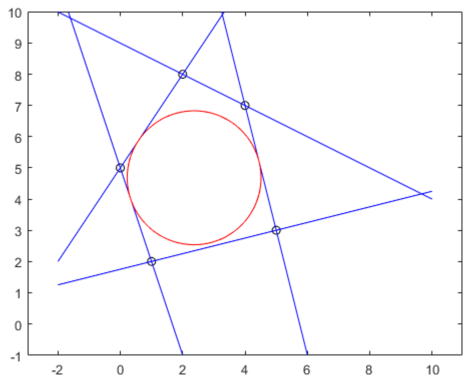


Figure 2.3: Largest circle that can be inscribed in the polygon.

It can be seen that this disk is indeed the largest circle that can be inscribed in the polygon defined by the set of points P.

3 Smallest enclosing balls

3.1 By using the stated theroem, find out the smallest ball enclosing the following 10 points: $p_1 = (0,0)$, $p_2 = (1,0)$, $p_3 = (2,-1)$, $p_4 = (-2,5)$, $p_5 = (-1,-4)$, $p_6 = (2,0)$, $p_7 = (-3,1)$, $p_8 = (-1,2)$, $p_9 = (1,-1)$, $p_{10} = (4,5)$. In order to test the result, plot the points jointly with the smallest circle found.

Theorem 1. *Let $P = p_1,...,p_n \subset R^m$ be a set of n-points from R^m. Let Q be the matrix $m \times n$ whose columns are the points of P, i.e:*

$$Q = \begin{pmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & & \vdots \\ p_{m1} & \cdots & p_{mn} \end{pmatrix}$$

Then:

$f : R^n \to R$ defined by $f(x) = x^T Q^T Q x - \sum_{i=1}^n p_i^T p_i x_i$ is a convex function and, if x^* is the solution of

$$\begin{cases} min\ f(x) \\ s.t.\ \sum_{i=0}^n x_i = 1 \\ x_i \geq 0 \end{cases}$$

the point $c^* = Q \cdot x^*$ is the center of the smallest enclosing ball of P, and the value $-f(x^*)$ is the squared radius of $B(P)$, the smallest enclosing ball containing these n points. □

This theorem enables us to solve the problem of finding the smallest ball enclosing n-points from R^m in the form of a quadratic programming problem, which can be solved in MATLAB with the function `quadprog()`. However, the function `quadprog()` provided by MATLAB to solve quadratic programming problems asks the user to formulate the problem in a specific way so it can undestand it. This formulation is:

$$min\ \frac{1}{2}x^T H x + f^T x \ \ such\ that \begin{cases} A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub \end{cases}$$

A possible implementation for solving the problem for the given points is:

```
n=10;                                  % # of points
Q=[0,0; 1,0; 2,-1; -2,5; -1,4;         % Matrix of n points in columns
   2,0;-3,1; -1,2; 1,-1; 4,5]';
```

Once the points we want to enclose have been defined its time to formulate the objective function and constraints in such a way that MATLAB can understand them. Taking a look at the way the function `quadprog()` solves a quadratic problem (which has been explained above) and the formulas derived from the theorem we can establish the relationships:

$$\mathbf{H} = 2{\cdot}(Q^T Q);\ \mathbf{f} = -\begin{bmatrix} p_1^T \cdot p_1 \\ \vdots \\ p_n^T \cdot p_n \end{bmatrix};\ \mathbf{A} = -\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix};\ \mathbf{b} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix};\ \mathbf{Aeq} = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix};\ \mathbf{beq} = \begin{bmatrix} 1 \end{bmatrix}$$

Which, translated to MATLAB becomes:

```
H=2*((Q')*Q);                                   % objective function
f=diag(Q'*Q);

Aeq = ones(1,n);                                % constraints
beq = ones(1,1);
A = -eye(n);
b = zeros(n,1);

[X,FVAL,EXITFLAG] = quadprog(H,-f,A,b,Aeq,beq);    % Solve quadratic program
```

The code above yields the results:

```
Minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the optimality tolerance,
and constraints are satisfied to within the default value of the constraint tolerance.

X' = 0.0000 0.0000 0.0562 -0.0000 -0.0000 0.0000 0.4671 0.0000 0.0000 0.4766

FVAL = -16.3062

EXITFLAG = 1
```

Following the theorem, once x has been computed we can then calculate the center and radius of the smallest ball enclosing the given points and plot it with the points:

```
C=Q*X;                                  % The center of the smallest ball
R=sqrt(-FVAL);                          % The radius of the smallest ball
plot(Q(1,:),Q(2,:),'rx')                % Plot the points as red crosses
hold on;
theta =linspace(0, 2*pi);               % Draw a circle of center C and radius R
plot(C(1)+R*cos(theta), C(2)+R*sin(theta))
axis equal;
```

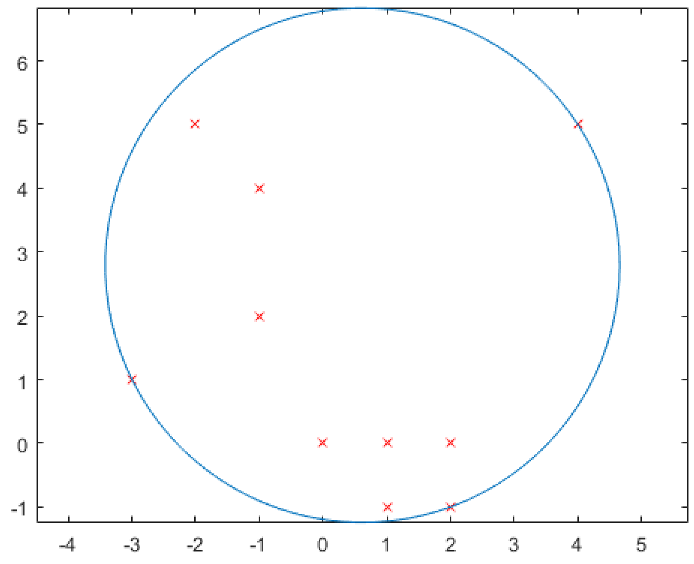


Figure 3.1: Plot of the given points in the cartesian plane.

4 The Thomson Problem

4.1 Define the MATLAB function `f()` associated to the objective function (Coulomb potential).

The problem consists in finding the minimal energy configuration for a given a number n of points from a unit sphere. This means that the objective function in this problem, the one we want to minimize, is the total energy E of a set of n points, P, depending on its spatial distribution.

The total energy is computed with the relationship

$$E(P) = \sum_{i>j} \frac{1}{[dist(p_i, p_j)]^{\varepsilon}}$$

Where $dist(p_i, p_j)$ is the distance betweenp_i and p_j and $\varepsilon \in (0, +\infty)$. In our case of study $\varepsilon = 1$. This case is known as the *Coulomb potential*. Defining this function as a MATLAB function:

```
function z = f(x)                               % x is a 1D vector containing 3D points
   n = numel(x(1,:))/3;                         % # of points
   z = 0;
   for i=0:(n-2)
      for j=(i+1):(n-1)
         z=z+1/(sqrt((x(3*i+1)-x(3*j+1))^2+(x(3*i+2)-x(3*j+2))^2+(x(3*i+3)-x(3*j+3))^2));
      end
   end
end
```

4.2 Define the MATLAB function `confun()` associated to the constraints.

The only constraint present in the problem is that all the points have to be located in the surface of the unit sphere. This means that there are only equality constraints and no inequality constraints. We will assume the centre of the sphere is the point $c = (0, 0, 0)$. This simplifies he problem without any loss of generality. The constraints are then:

$$\begin{cases} x_1^2 + y_1^2 + z_1^2 = 1 \\ x_2^2 + y_2^2 + z_2^2 = 1 \\ \vdots \\ x_n^2 + y_n^2 + z_n^2 = 1 \end{cases} \Rightarrow ceq(x_1, x_2, \cdots, x_{3n}) = \begin{pmatrix} x_1^2 + x_2^2 + x_3^2 - 1 \\ \vdots \\ x_{3n-2}^2 + x_{3n-1}^2 + x_{3n}^2 - 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

In order to solve the problem with the MATLAB function `fmincom()` this constraints have to be the return values of the nonlinear constraints function `confun()`:

```
function [c, ceq ] = confun(x)
   n = numel(x);

   c = [];                                   % Nonlinear inequality constraints

   ceq=[];                                   % Nonlinear equality constraints
   for i=0:n/3-1
      v=[x(i*3+1), x(i*3+2),x(i*3+3)];
      v = v*v'-1;
      ceq(i+1,1) = v;
   end
end
```

4.3 Find a feasible initial value x_0 and solve the optimization problem by means `fmincon()`

A feasible initial value of x_0 can be obtained with $x_i, y_i \in (0, 0.5)$ and $z_i = \sqrt{1 - x_i^2 - y_i^2}$, which ensures that all the initial points lay on the surface of the unit sphere. Making $x_i = a, y_i = b$ and $z_i = c$:

```
N=4;                                        % # of electrons/points
x0=[];                                      % find a feasible initial disposition
for i=0:(N-1)
    a=0.5*rand(1);b=0.5*rand(1);c=sqrt(1-a^2-b^2);
    x0=[x0,a,b,c];
end
```

Taking a look at the `fmincom()` documentation we find that to solve the problem the call we have to do to `fmincom()` needs to be provided with the next input parametres:

`fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)`

Where `fun` is the handle of the objective function, `x0` is the initial point, `nonlcon` is the handle of the nonlinear constraints function and `options` are some options defined by the user. In our case the rest of parametres will be empty matrices, as there are no more constraints that the ones already stated. `A=b=Aeq=beq=lb=ub=[]`. With this in mind we can solve the problem for any number of points N. We will start with $N = 4$:

```
% Solve problem of optimal configuration
options = optimoptions(@fmincon,'Algorithm','sqp');
[x,fval] = fmincon(@f,x0,[],[],[],[],[],[],@confun,options);
```

The output of running the code above is:

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the optimality tolerance,
and constraints are satisfied to within the default value of the constraint tolerance.

x = 0.8298 -0.4544 -0.3238 -0.3057 -0.4301 0.8494 0.1928 0.9670 0.1667 -0.7170 -0.0824 -0.6922

fval = 3.6742
```

4.4 Draw the final position of electrons using function `plot3()`

The code presented below plots the computed location of the points in 3D space:

```
P=[];
for i=0:(N-1); P=[P;[x(3*i+1),x(3*i+2),x(3*i+3)]]; end
figure                              % Plot electron position
for i=1:numel(P(:,1)); plot3(P(i,1),P(i,2),P(i,3),'k.','MarkerSize',10);hold on;end
M=zeros(N);                         % Calculate distances between electrons
for i=1:N
    for j=1:N; M(i,j)=sqrt((P(i,:)-P(j,:))*(P(i,:)-P(j,:))'); end
end
C=connections(M);C = triu(C);    % Get connections between vertices
for i=1:N                           % Plot lines
    for j=1:N
        if C(i,j)==1
            plot3([P(i,1),P(j,1)],[P(i,2),P(j,2)],[P(i,3),P(j,3)],'b','LineWidth',1);hold on;
        end
    end
end
grid on; axis equal;
```

Where the function `connections(M)` is defined as:

```
function F = connections(M)
    Z = round((M*100))/100; nodes = numel(M(:,1));
    Q = zeros(nodes);
    elements = sort(unique(Z(:)))'; elements=elements(2:end);
    if nodes>=3
        edges = 3*nodes-6;
    else edges = 1;
    end
    for i=1:numel(elements)      % Compute connections
        R = Z == elements(i);    % Create adjacency matrix for distance relaxation i
        Q = Q+R;                 % Update adjacency matrix
        C = triu(Q);             % Check number of edges
        if sum(C(:))>=edges
            F = Q;
            break;
        end
    end
end
```

For $N = 4, 5, 6, 8$ the obtained figures are:

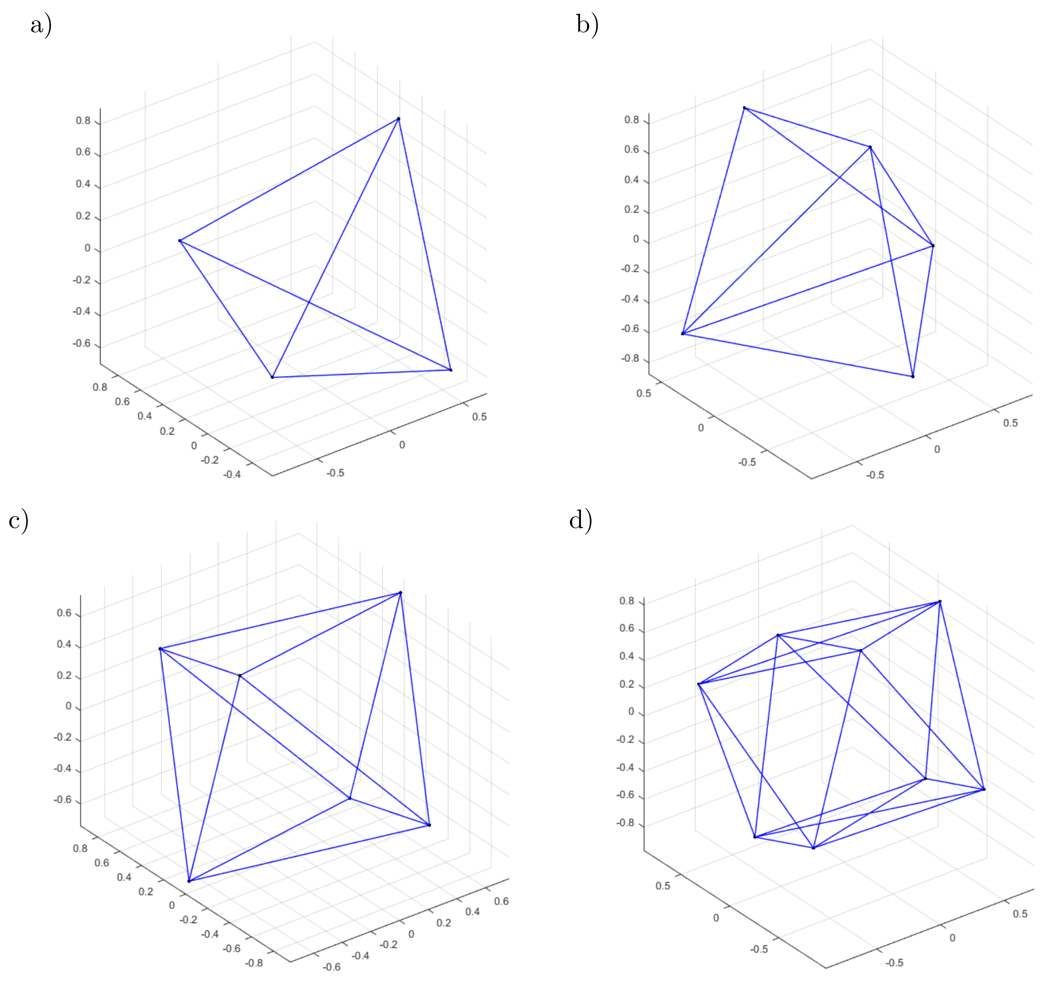


Figure 4.1: Plot of the resulting point distribution for $n = 4$ (a), $n = 5$ (b), $n = 6$ (c), $n = 8$ (d).

4.5 In the cases $n = 4$, and $n = 6$, check that the configurations are a tetrahedron and an octahedron respectively.

The figures of the final configurations when $n = 4$ and $n = 6$ seem to be, respectively, a tetrahedron and an octahedron. This can be seen in Figure 4.1. We will check if this is, indeed, the case.

Suppose we have a cube with inner diagonal length equal to two, the same as the unit sphere diameter, and that both have the center at the same point. To make calculations easier we will assume their center is at point $c = (0, 0, 0)$. It should be clear that the vertices of the cube lay on the surface of the sphere.

Now trace a tetrahedron taking four adjacent vertices of this cube as the four vertices of the tetrahedron, as shown in Figure 4.2. The vertices of this tetrahedron also lay in the surface of the unit sphere and all its edges have the same length, as they are de diagonals of the faces of the cube.

Taking into account that the inner diagonal of the cube the tetrahedron is inscribed into has length two, we can compute the length of the sides of the tetrahedron.

$$2 = \sqrt{l^2 + 2 \cdot l^2} \rightarrow l = \frac{2}{\sqrt{3}} \rightarrow d = \sqrt{\frac{2}{\sqrt{3}}^2 + \frac{2}{\sqrt{3}}^2} = \sqrt{\frac{4}{3} + \frac{4}{3}} = \sqrt{\frac{8}{3}} = 1.6330$$

Where l is the length of the sides of the cube and d the length of the sides of the tetrahedron. This result implies that any four points lying on the surface of the unit sphere that sarisfy that the distante from one point to any of the others is 1.6330 can be rotated to match the position of the tetrahedron we just built.

In our case, when $n = 4$ the distance between points yielded by the optimization is:

```
M =
        0    1.6330    1.6330    1.6330
   1.6330         0    1.6330    1.6330
   1.6330    1.6330         0    1.6330
   1.6330    1.6330    1.6330         0
```

Where $M(i, j) = dist(i, j)$. As this distance is 1.6330 and we know, as it is a constraint, that the four points lay on the surface of the sphere, we can be sure the spatial distribution of these points is a tetrahedron.

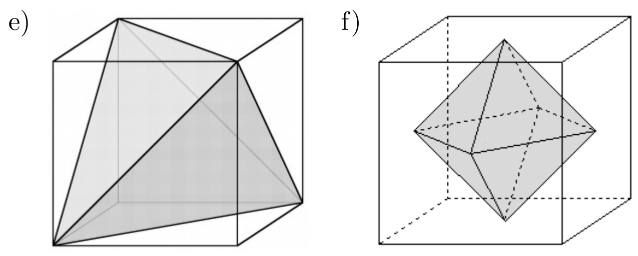


Figure 4.2: A tetrahedron (e) and an octahedron (f) inscribed in a cube.

A similar reasoning can be applied four the case $n = 6$. In this case, we will start with a cube whose side length is equal to the diameter of the unit sphere, this is, a length equal to two. If we center the cube and the unit sphere at the same point, $c = (0, 0, 0)$, it is obvious that the center of any of the faces of the cube is tangent to the sphere. Now we build an octahedron where the six vertices are this six points where the cube and the sphere are tangent, as shown in Figure 4.2. Knowing that the side length of the cube is two we can calculate the length of the sides of the octahedron, which are all equal due to simmetry:

$$d = \sqrt{\frac{l}{2}^2 + \frac{l}{2}^2} = \sqrt{\frac{2 \cdot l^2}{4}} = l \cdot \frac{\sqrt{2}}{2} = 2 \cdot \frac{\sqrt{2}}{2} = \sqrt{2} = 1.4142$$

Where l is the length of the sides of the cube and d the length of the sides of the octahedron. This leads to the result that any given set of six points where for any point the distance to four of the other points is 1.4142 (adjacent vertices) and 2 to the fourth point (opposite vertix) can be rotated to match this octahedron. Furthermore, the six points will lay on the surface of the unit sphere.

In our case, when $n = 6$ the distance between points yielded by the optimization is:

```
M =
         0    1.4142    1.4142    1.4142    2.0000    1.4142
    1.4142         0    1.4142    2.0000    1.4142    1.4142
    1.4142    1.4142         0    1.4142    1.4142    2.0000
    1.4142    2.0000    1.4142         0    1.4142    1.4142
    2.0000    1.4142    1.4142    1.4142         0    1.4142
    1.4142    1.4142    2.0000    1.4142    1.4142         0
```

Where $M(i,j) = dist(i,j)$. As this distance is 1.41420 for adjacent vertices and 2 to the opposite vertex[1] and we know that the six points lay on the surface of the sphere, we can be sure the spatial distribution of these points is an octahedron. As this is the result of our optimization when $n = 6$, we can be sure the shape of this 6 points is an octahedron.

4.6 In the cases $n = 5$, and $n = 8$, investigate the final configurations calculating distances and angles.

To get a closer idea of the spatial distribution of the optimal location of points, apart from the distances between them, we will compute some of the angles that will help us determine the final shape.

For the case $n = 5$ we will compute the angles in degrees between the vectors $\vec{OP_i}$ and $\vec{OP_j}$ for $i = 1, ..., N$ $j = 1, ..., N$, where $O = (0, 0, 0)$. This can be easily implemented in MATLAB with the piece of code:

```
Angles=zeros(N);
for i=1:N
    for j=1:N
        a =P(i,:);b=P(j,:);
        Angles(i,j) = atan2d(norm(cross(a,b)),dot(a,b));
    end
end
```

We have then that the distances between points and angles between vectors obtained from the optimization are:

```
M =
         0    1.7321    1.7321    1.4142    1.4142
    1.7321         0    1.7321    1.4142    1.4142
    1.7321    1.7321         0    1.4142    1.4142
    1.4142    1.4142    1.4142         0    2.0000
    1.4142    1.4142    1.4142    2.0000         0
```

```
Angles =
         0   90.0000   90.0000  120.0000  120.0000
   90.0000         0  180.0000   90.0000   90.0000
   90.0000  180.0000         0   90.0000   90.0000
  120.0000   90.0000   90.0000         0  120.0000
  120.0000   90.0000   90.0000  120.0000         0
```

[1] A vertex can only have one opposite vertex.

Plotting the set of unitary vectors that satisfy this angle matrix we obtain

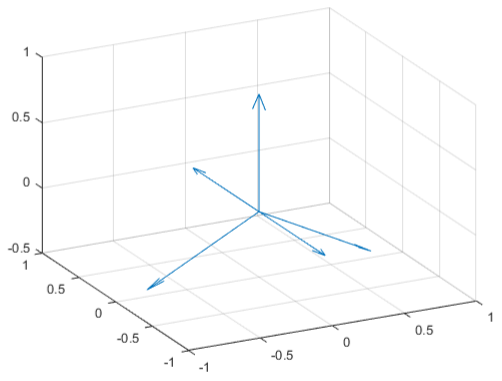


Figure 4.3: Plot of the unitary vectors that satisfy the angle matrix.

From Figure 4.3 it can be deduced that the final distribution in the case $n = 5$ is trigonal bipyramidal.

For the case $n = 8$ we have that the distances between points obtained from the optimization are:

```
M =
         0    1.1712    1.2877    1.2877    1.8969    1.8969    1.6564    1.1712
    1.1712         0    1.2877    1.8969    1.2877    1.8969    1.1712    1.6564
    1.2877    1.2877         0    1.1712    1.1712    1.6564    1.8969    1.8969
    1.2877    1.8969    1.1712         0    1.6564    1.1712    1.8969    1.2877
    1.8969    1.2877    1.1712    1.6564         0    1.1712    1.2877    1.8969
    1.8969    1.8969    1.6564    1.1712    1.1712         0    1.2877    1.2877
    1.6564    1.1712    1.8969    1.8969    1.2877    1.2877         0    1.1712
    1.1712    1.6564    1.8969    1.2877    1.8969    1.2877    1.1712         0
```

From the distances matrix we can deduce that the resulting figure has a high degree of symmetry, as any column or row of the matrix contains the same distances in different order. Furthermore, it can be seen that points the subsets of points $S_1^P : \{P_1, P_2, P_8, P_7\}$ and $S_2^P : \{P_3, P_4, P_6, P_5\}$, in this same order, form two closed loops where the distance from one point to its consecutive point is 1.1712. We can calculate the angles formed by the vectors $\overrightarrow{S_{1,i}^P S_{1,i+1}^P}$ and $\overrightarrow{S_{1,i+1}^P S_{1,i+2}^P}$ and also $\overrightarrow{S_{2,j}^P S_{2,j+1}^P}$ and $\overrightarrow{S_{2,j+1}^P S_{2,j+2}^P}$ with the piece of code:

```
% Computing the angle between the vectors P_3->P_4 and P_4->P_6
ang = atan2d(norm(cross(P(4,:)-P(3,:),P(6,:)-P(4,:))),dot(P(4,:)-P(3,:),P(6,:)-P(4,:)))
```

Which yields the result:

```
ang = 90.0001
```

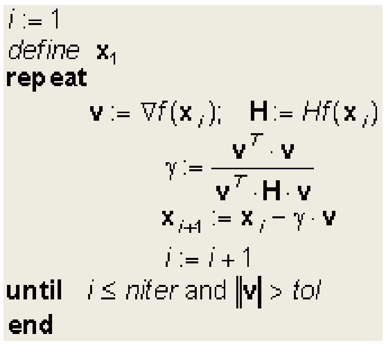
By doing this calculation we find us that the angle is 90^o for any[2] $i = 1, ..., 4$ and $j = 1, ..., 4$. This leads us to the conclusion that the spatial distribution of this two sets of points is a square. By observing the rest of the distances and its symmetry we can also conclude that the two squares are parallel and rotated one with respect to the other around an imaginary axis that is perpendicular to the planes the squares are contained in.

[2]The two sets S_1^P and S_2^P are considered to be circular.As an example, $i + 2$ when $i = 4$ is 2.

5 Numerical methods. Algorithms

5.1 Complete the code for cases 3, 4 and 5 (Steepest descent using second order approximation, Conjugate gradient and Newton method).

The first of this cases to be completed is the steepest descent with second order approximation algorithm:



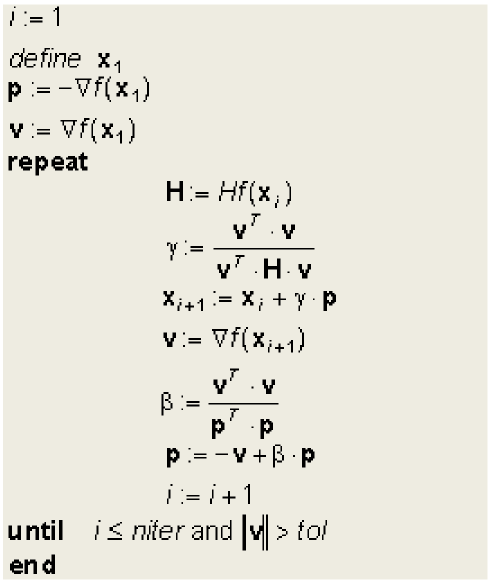
```
$i := 1$
define $\mathbf{x}_1$
repeat
        $\mathbf{v} := \nabla f(\mathbf{x}_i)$;   $\mathbf{H} := Hf(\mathbf{x}_i)$
            $\gamma := \frac{\mathbf{v}^T \cdot \mathbf{v}}{\mathbf{v}^T \cdot \mathbf{H} \cdot \mathbf{v}}$
            $\mathbf{x}_{i+1} := \mathbf{x}_i - \gamma \cdot \mathbf{v}$
            $i := i + 1$
until  $i \le niter$ and $\|\mathbf{v}\| > tol$
end
```

Figure 5.1: Pseudocode for the steepest descent with quadratic approximation algorithm.

This algorithm implemented in MATLAB results in:

```
i=0;
while i<=niter && ngx>tol
   histx=[histx,x];
   gx = gradf(x);hx = hesf(x);
   y=((gx')*gx)/((gx')*(hx*gx));
   x=x-y*gx;
   ngx=norm(gx);
   i=i+1;
end
```

The second case is the conjugate gradient algorithm:



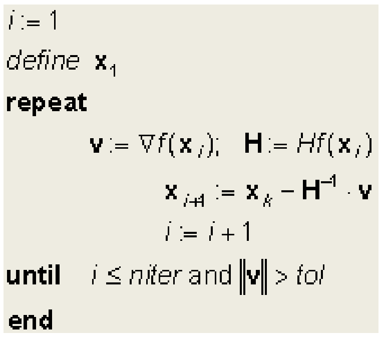
```
$i := 1$
define $\mathbf{x}_1$
$\mathbf{p} := -\nabla f(\mathbf{x}_1)$
$\mathbf{v} := \nabla f(\mathbf{x}_1)$
repeat
        $\mathbf{H} := Hf(\mathbf{x}_i)$
        $\gamma := \frac{\mathbf{v}^T \cdot \mathbf{v}}{\mathbf{v}^T \cdot \mathbf{H} \cdot \mathbf{v}}$
        $\mathbf{x}_{i+1} := \mathbf{x}_i + \gamma \cdot \mathbf{p}$
        $\mathbf{v} := \nabla f(\mathbf{x}_{i+1})$
        $\beta := \frac{\mathbf{v}^T \cdot \mathbf{v}}{\mathbf{p}^T \cdot \mathbf{p}}$
        $\mathbf{p} := -\mathbf{v} + \beta \cdot \mathbf{p}$
        $i := i + 1$
until  $i \le niter$ and $\|\mathbf{v}\| > tol$
end
```

Figure 5.2: Pseudocode for the conjugate gradient algorithm.

This secong algorithm implemented in MATLAB results in:

```
i=1;
px=-gradf(x);
while i<=niter && ngx>tol
   histx=[histx,x];
   gx = gradf(x);hx = hesf(x);
   y=((gx')*gx)/((gx')*(hx*gx));
   x=x+y*px;
   gx = gradf(x);
   b = (gx'*gx)/(px'*px);
   px = -gx+b*px;
   ngx=norm(gx);
   i=i+1;
end
```

The third and last case is the Newton algorithm. The pseucode of the algorithm is shown below:



```
$i := 1$
define $\mathbf{x}_1$
repeat
        $\mathbf{v} := \nabla f(\mathbf{x}_i)$;   $\mathbf{H} := Hf(\mathbf{x}_i)$
                $\mathbf{x}_{i+1} := \mathbf{x}_k - \mathbf{H}^{-1} \cdot \mathbf{v}$
                $i := i + 1$
until   $i \leq niter$ and $\|\mathbf{v}\| > tol$
end
```

Figure 5.3: Pseudocode for the Newton algorithm.

This is the last algorithm we will consider. This last algorithm can be implemented in MATLAB with the following code:

```
i=0;
while i<=niter && ngx>tol
   histx=[histx,x];
   gx = gradf(x);hx = hesf(x);
   x=x-hx\gx;
   ngx=norm(gx);
   i=i+1;
end
```

After coding the algorithms for the cases 3,4 and 5 the full code for the function `minim()` is:

```
function [x,histx,ns,val]=minim(algor,seed,tol,niter,a)
%
%[x,ns,histx,val]=minim(seed,algor,tol,niter,a)
%
% INPUTS
% algor = algorithm to be used
%            1.FixedStep
%            2.SimpleFixedStep
%            3.Steepest using second order approximation
%            4.Conjugate gradient method
%            5.Newton method
% seed =initial value /default [0;0]
% tol  =stop criterium gradf(x)<tol /default 0.001
% niter =iterations number /default 100
% a    =step size for algorithm 1 and 2 /default 0.1
```

```
%
% OUTPUTS
% x     =the minimum
% histx =the historic sequence of x
% ns    =the number of steps
% val   =f(min)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%defaut values
if nargin < 5 a=0.1;                       % default stepsize
   if nargin < 4 niter=500;                % default max # iterations
       if nargin < 3 tol=0.001;            % default tolerance
           if nargin < 2 seed=[0;0];       % default initial point
              if nargin < 1 algor=5;       % default algorithm = Newton
              end
           end
       end
   end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if size(seed,1)==1 seed=seed'; end;        % a column vector is needed

x=seed;                                    % the first point is the seed
histx=[];                                  % sequence of point initialization
gx=gradf(x);                               % calculate the gradient
hx=hesf(x);                                % calculate the hessian
ngx=norm(gx);                              % and the norm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
switch algor
    case {1}             % fixed step gradient descent
       i=1;
       while i<=niter && ngx>tol
           histx=[histx,x];
           x=x-a*gx/ngx;
           gx=gradf(x);
           ngx=norm(gx);
           i=i+1;
       end

    case {2}             % simple gradient descent
       i=1;
       while i<=niter && ngx>tol
           histx=[histx,x];
           x=x-a*gx;
           gx=gradf(x);
           ngx=norm(gx);
           i=i+1;
       end

    case {3}             % steepest gradient descent with second order approx
       i=0;
       while i<=niter && ngx>tol
           histx=[histx,x];
           gx = gradf(x);hx = hesf(x);
           y=((gx')*gx)/((gx')*(hx*gx));
           x=x-y*gx;
           ngx=norm(gx);
           i=i+1;
       end

    case {4}             % conjugate
       i=1;
       px=-gradf(x);
       while i<=niter && ngx>tol
           histx=[histx,x];
```

```
            gx = gradf(x);hx = hesf(x);
            y=((gx')*gx)/((gx')*(hx*gx));
            x=x+y*px;
            gx = gradf(x);
            b = (gx'*gx)/(px'*px);
            px = -gx+b*px;
            ngx=norm(gx);
            i=i+1;
        end

    case {5}          % newton
        i=0;
        while i<=niter && ngx>tol
            histx=[histx,x];
            gx = gradf(x);hx = hesf(x);
            x=x-hx\gx;
            ngx=norm(gx);
            i=i+1;
        end

    otherwise
        disp('minim: Incorrect algorithm')
        return
end
histx=[histx,x];
histx=histx';
ns=i-1;
val=f(x);
end
```

5.2 Use the previous code to obtain the minimum of $f(x,y) = x^4 + 3y^2 + x^2 – 2xy + y$ by using the five algorithms and compare them (for instance, number of steps). Use the proposed default values for variables `seed`, `tol`, `niter` and `a`. You must complete the functions `minim()`, `f()`, `gradf()` and `hesf()`.

In order to minimize the desired function using the five proposed algorithms, the functions `f()` (function to be optimized), `gradf()` (gradient of the function to be optimized) and `hesf()` (hessian of the function to be optimized) have to be defined, as the values they take at a specific point of the function domain will be used to compute the next point iteratively until the minimun is found or the maximum number of steps is reached.

In our case $f(x,y) = x^4 + 3y^2 + x^2 – 2xy + y$. This will be our function `f()` In MATLAB, we can express it as:

```
function z = f(x)
    z = x(1)^4+3*x(2)^2+x(1)^2-2*x(1)*x(2)+x(2);
end
```

Where the input `x` of `f()` is a vector containing the coordinates of the point where the funtion is being evaluated. Knowing the mathematical expression of $f(x,y)$ we can derive the gradient function:

$$(x,y) = x^4 + 3y^2 + x^2 – 2xy + y \Rightarrow \nabla f(x,y) = (4x^3 + 2x – 2y, 6y – 2x + 1)$$

This will become our `gradf()` function:

```
function g = gradf(x)
    g = [4*x(1)^3+2*x(1)-2*x(2) ; 6*x(2)-2*x(1)+1];
end
```

Finally, from the gradient of the function we can derive its Hessian matrix

$$\nabla f(x,y) = (4x^3 + 2x – 2y, 6y – 2x + 1) \Rightarrow H(f(x,y)) = \begin{pmatrix} 12x^2 + 2 & -2 \\ -2 & 6 \end{pmatrix}$$

Which is our `hesf()` function:

```
function h = hesf(x)
    h = [12*(x(1)^2)+2 , -2 ; -2 , 6];
end
```

Once having defined this functions we can now proceed to find the minimum of $f(x,y)$ with each of the five algorithms and compare the results:

```
x=linspace(-1,1,101);
y=linspace(-1,1,101);
[X,Y]=meshgrid(x,y);
Z=zeros(101,101);
for i=1:101
    for j=1:101
        Z(i,j)=f([X(i,j),Y(i,j)]);
    end
end
for algor=1:5
    [x,histx,ns,val]=minim(algor);
    figure
    contour(X,Y,Z,100);
    hold on;
    plot(histx(1:ns+1,1),histx(1:ns+1,2),'-ro')
    title(strcat('Algorithm ', num2str(algor)));
    xlabel(strcat(num2str(ns),' steps'));
end
```

The obtained results after running each of the algorithms are listed below.

	Fixed step gradient descent	Simple gradient descent	Steepest gradient descent with second order approximation	Conjugate	Newton
x^*	(-0.1959,-0.2875)	(-0.2182,-0.2393)	(-0.2186,-0.2395)	(-0.2183,-0.2394)	(-0.2186,-0.2395)
ns	500	37	9	8	3
fval	-0.1123	-0.1221	-0.1221	-0.1221	-0.1221
Algorithm number	1	2	3	4	5

Table 5.1: Results of the optimization obtained with the five different methods.

As it can be seen in the table above, the fixed step gradient descent is the only method that reaches the maximum number of iterations without convergence, as the norm of the gradient after 500 iterations is $|\nabla f(x_{500}, y_{500})| = 0.3664$.This results in this method being the one that yields the highest value of $f(x,y)$, $f(x,y) = -0.1123$. From all the tested algorithms, in this case, this is the less accurate. In contrast, the other four methods converge to a solution. The conjugate method, the Newton method and the steepest gradient descent with second order approximaton method are the faster ones to converge, in 8, 3 and 9 steps respectively. This three methods all yield the same value for $f(x,y)$ at its minimum, $f(x,y) = -0.1221$ and at coordinates that vary in a range of $|(-0.2186) - (-0.2183)| = 0.0003$ for x and $|(-0.2394) - (-0.2395)| = 0.0001$ for y. The simple gradient descent method also yields similar results for the minimum coordinates and the same exact value for the minimum. However, it does so with a higher number of steps, 37. Taking the mean value of the x and y coordinates yielded by the four algorithms that converged we obtain that the location of the minimum is at location $(x, y) = (-0.2184, -0.2394)$.

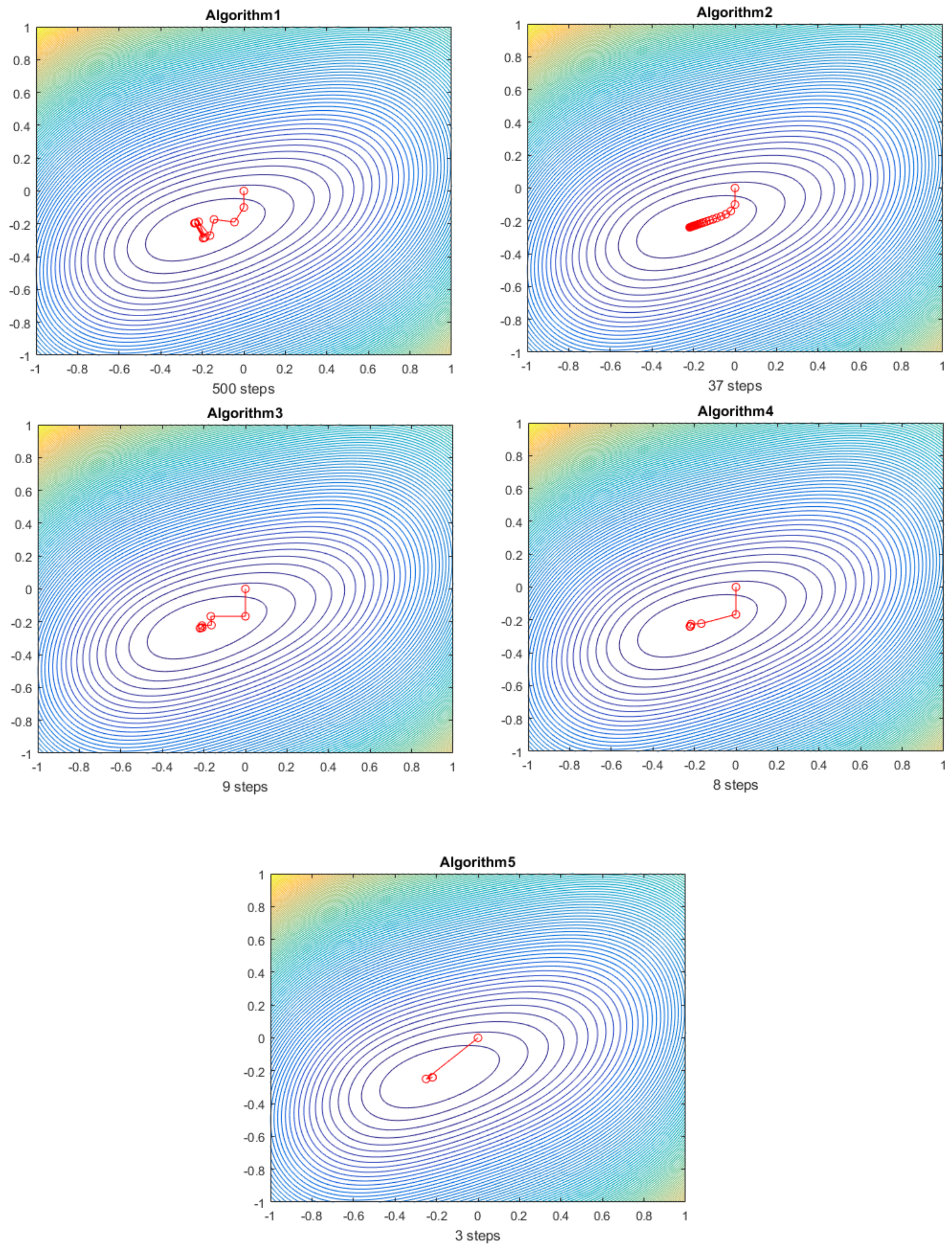


Figure 5.4: Convergence path and number of steps for each of the algorithms.

5.3 Use the MATLAB function `fminunc()` to check the solution. You only have to introduce the instruction and see the answer.

```
[x,fval]=fminunc(@f,[0,0]);
```

```
Local minimum found.

Optimization completed because the size of the gradient is less than
the default value of the optimality tolerance.

<stopping criteria details>


x =  -0.2186  -0.2395

fval = -0.1221
```

As it can be seen, the resolution with the MATLAB function `fminunc()` gives the same result as the one obtained with the previous algorithms, both for the coordinates and function value.