Molecular Distance Geometry Problem and training of neural networks.

Exercize 1: Molecular Distance Geometry Problem

The *Molecular Distance Geometry Problem*, or MDGP, consists in reconstructing the 3-D structure of a molecule from information on the distances between its atoms. Let us assume that the distances between all pairs of atoms are known (with infinite precision). Let $D \in \mathcal{R}^{m\times m}$ the symmetrix matrix of the Euclidean distances d_{ij} for each pair $\{i, j\}$.[1]

Let $\underline{x}^1, \dots, \underline{x}^m \in \mathbb{R}^3$ the unknown positions of the atoms. We can assume w.l.o.g. that the m-th point has coordinates $\underline{x}^m = (0, 0, 0)$. The problem consists of determining the coordinates $\underline{x}^1, \dots, \underline{x}^{m-1} \in \mathbb{R}^3$ of the other $m-1$ atoms so to satisfy the distances d_{ij} between the pairs of atoms i and j. The corresponding constraints are:

$$||\underline{x}^i - \underline{x}^j||_2 = d_{ij}, i, j = 0, 1, \dots, m.$$

a) Give a nonlinear unconstrained optimization formulation for the problem.

b) Due to the non-convexity of the problem, implement a multistart method based on the methods implemented in the previous lab. Let:

- $\varepsilon > 0$ the tolerance of the multistart algorithm
- $\bar{f}$ the known optimal value of the objective function
- M the maximum number of multistart iterations

The multistart algorithm is as follows:

(a) Let $\underline{x}$ be a point in $\mathbb{R}^{3(m-1)}$. Let $\underline{x}^* \leftarrow \underline{x}$.
(b) If $f(\underline{x}^*) < \bar{f} + \varepsilon$ or if more than M iterations, the algorithm stops; otherwise, go to step c).
(c) Find local minimum $\underline{x}' = (x^1, \dots, x^{m-1})$ from initial point $\underline{x}$, with a nonlinear optimization method (with a tolerance $\varepsilon' > 0$ and a maximum number of iterations M').
(d) If $f(\underline{x}') < f(\underline{x}^*)$ update $\underline{x}^* \leftarrow \underline{x}'$.
(e) Find new initial point $\underline{x} \in \mathbb{R}^{3(m-1)}$, randomly.
(f) Go to step b).

c) Apply the algorithm to the following instance with $m = 4$ atoms ($d_{ii} = 0$ for any i):

d_{ij}	2	3	4
1	1.526	2.491389536	3.837572036
2	0	1.526	2.491389535
3	-	0	1.526

[1] A reference is: J.M. Yoon, Y. Gad, Z. Wu, *Mathematical modeling of protein structure using distance geometry*, Technical report TR00-24, DCAM, Rice University, Houston, 2000, available from: `http://www.caam.rice.edu/caam/trs/tr00.html#TR00-24`.

To build the initial random solution, use `rnd.m`, that generates a vector of n elements between `-bound` and `bound` according to a uniform distribution.

```
function xrnd = rnd(n, bound)
  xrnd = zeros(n,1);
  for i = 1:size(xrnd,1)
    xrnd(i) = (rand()-0.5)*2*bound;
  end
end
```

d) Apply the Gauss-Newton Method for the solution of nonlinear least square problems. We give `jac.m`, that computes the Jacobian at `x` of a vector function `r` with `m` components. The second parameter represents the number of components.

```
% Jacobian of a vector function at a point
function J = jac(r, m, x)
  n = length(x);
  J = zeros(m,n);
  h = 0.0001;

  for i = 1:m
   for j = 1:n
     delta = zeros(n, 1); delta(j) = h;
     rd=r(x+delta);
     rx=r(x);
     J(i,j) = (rd(i)  -  rx(i)) / h;
   end
  end
end %end of function
```

Use Matlab `pinv`, that, given matrix A, computes its pseudoinverse $\left(A^TA\right)^{-1}A^T$.

Gauss-Newton method It is a variant of Newton method for nonlinear least squares problem. Consider:

$$f(\underline{x}) = \sum_{i=1}^{m}(r_i(\underline{x}))^2, \tag{1}$$

where $\underline{x} \in R^n$ and $r(\underline{x}) = (r_1(\underline{x}), \ldots, r_m(\underline{x}))^T$ is the vector of residuals. Assume $r_i(\underline{x})$ are nonlinear.

Differentiating (1) we obtain

$$\nabla_{\underline{x}} f(\underline{x}) = \sum_{i=1}^{m} 2r_i(\underline{x})\nabla_{\underline{x}} r_i(\underline{x}).$$

Let $\mathbf{J}_{\underline{x}}\underline{r}(\underline{x}) = \left\{\frac{\partial r_i}{\partial x_k}\right\}_{ik}$ the Jacobian of $\underline{r}$ at $\underline{x}$: we indicate $\mathbf{J}_{\underline{x}}\underline{r}(\underline{x})$ with $\mathbf{J}(\underline{x})$. Observe $\mathbf{J}(\underline{x}) = \begin{pmatrix} \nabla_{\underline{x}}^T r_1(\underline{x}) \\ \vdots \\ \nabla_{\underline{x}}^T r_m(\underline{x}) \end{pmatrix}$. We can write the expression as:

$$\nabla f(\underline{x}) = 2\mathbf{J}_{\underline{x}}\underline{r}(\underline{x})^T \underline{r}(\underline{x}).$$

The Hessian of f at $\underline{x}$, that we indicate with $\mathbf{H}(\underline{x})$, is

$$\mathbf{H}(\underline{x}) = 2\mathbf{J}(\underline{x})^T\mathbf{J}(\underline{x}) + 2\sum_{i=1}^{m}(r_i(\underline{x}))\nabla^2 r_i(\underline{x}). \tag{2}$$

If the residuals are small, we can discard the last term in (2), leading to

$$\mathbf{H}(\underline{x}) \approx 2\mathbf{J}(\underline{x})^T\mathbf{J}(\underline{x}).$$

Observe that, if the residuals are linear in $\underline{x}$, then $\nabla^2 r_i(\underline{x}) = 0$ for any i, so the second-order approximation is exact and the method is the same as the one for linear least squares.

The Netwon method step:

$$\underline{x}_{k+1} = \underline{x}_k - [\mathbf{H}(\underline{x}_k)]^{-1}\nabla f(\underline{x}_k)$$

corresponds, using approximation of the Hessian to the first derivative, to:

$$\underline{x}_{k+1} = \underline{x}_k - \left[\mathbf{J}(\underline{x}_k)^T\mathbf{J}(\underline{x}_k)\right]^{-1}\mathbf{J}(\underline{x}_k)^T\underline{r}(\underline{x}_k).$$

For the convergence of Gauss-Newton, the inital solution must be sufficiently close to a stationary point of f and the discarded terms (2) must be small. Observe that the method does not need to compute the Hessian $\mathbf{H}(\underline{x})$, that would imply computing the Hessian $\nabla^2 r_i$ for each r_i.

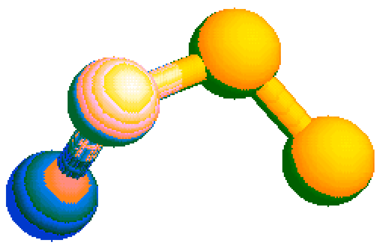


Figure 1: Optimal configuration.

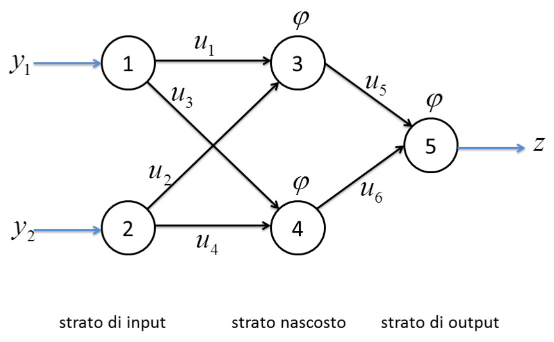
Esercize 2 (extra): neural networks

We have a dataset reported in Table 1, where y_1, y_2 are the input values and z the output of an unknown function: Suppose we want to build a neural network replicating the logical function that generated the data.

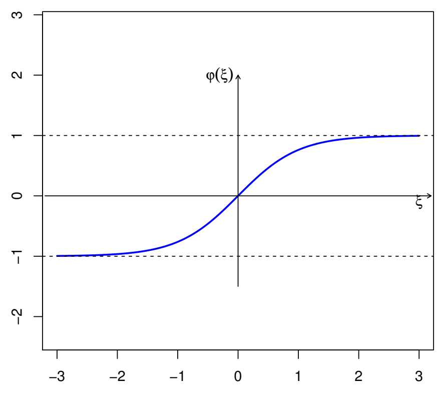
a) Let $\varphi(\xi) = \frac{e^{\xi}-e^{-\xi}}{e^{\xi}+e^{-\xi}}$ the activation function, depicted in Figure 2b. Train the network in Figure 2a with the reported dataset (*training set*), finding a set of weights $\underline{u}$ corresponding to the network arcs so to approximate the data. Use the multistart algorithm to find a good local optimum.

Table 1: Exercise 2.

y_1	y_2	z
0.1	0.1	0.1
0.9	0.9	0.05
0.0	0.95	0.98
0.95	0.1	0.95



(a) Neural network considered in the exercise.



(b) Plot of $\varphi(\xi)$.

Figure 2: Features of the neural network.

Use the file `f_hmap.m`, that represents the function implemented by the neural network in Figure 2a:

```
function f = f_hmap(u, y)
     f = f_act(u(5)* f_act(u(1)*y(1) + u(2)*y(2)) ...
             + u(6)* f_act(u(3)*y(1) + u(4)*y(2)));
end

function f = f_act(xi)
  f = (exp(xi) - exp(-xi)) / (exp(xi) + exp(-xi));
end
```

and the file `f_nnet.m`, that encodes the error function to minimize:

```
function f = f_nnet(u)
  y = [ 0.1  0.1  ;
        0.9  0.9  ;
        0.0  0.95 ;
        0.95 0.1  ];
  z = [ 0.1  0.05  0.98  0.95 ]';
  f = 0;
  for i = 1:length(z)
    f = f + ( z(i) - f_hmap(u, [y(i,1) y(i,2)]') )^2;
  end
end
```

b) Compute the outputs with the inputs (0,0), (0,1), (1,0), (1,1) and establish what kind of circuit we have.

Solution

Exercise 1: molecular geometry with euclidean distances

a) Let us model the problem. Let $\underline{x}^i = (x_1^i, x_2^i, x_3^i)$ be the position of the i-th atom with $i = 1, \ldots, m$. We are looking for a solution which satisfies all the constraints

$$||\underline{x}^i - \underline{x}^j||_2 = d_{ij} \quad \text{with } i = 1, \ldots, m, j = 1, \ldots, m. \tag{3}$$

Recall the expression of the euclidean distance $||\underline{x} - \underline{y}||_2 = \sqrt{\sum_i (x_i - y_i)^2}$. Since, if $||\underline{x}^i - \underline{x}^j||_2 = d_{ij}$, is also valid $||\underline{x}^i - \underline{x}^j||_2^2 = d_{ij}^2$, we can remove the nonlinearity introduced by the square root, by squaring both the terms of the equation. Observe that, since d_{ij} is a given parameter of the problem, no complications are introduced by squaring it.

We can formulate an unconstrained optimization problem in which we minimize a proper function of the errors (residuals) of the constraints, which attains a minimunm in corrispondence of a vector with null errors and which is not-decreasing for increasing errors in norm. We minimize the sum of the square errors $(||\underline{x}^i - \underline{x}^j||_2^2 - d_{ij}^2)^2$. The original problem can be formulated as the following nonlinear unconstrained optimization problem:

$$\min_{\underline{x}_1 \in \mathbb{R}^3, \ldots, \underline{x}_m \in \mathbb{R}^3} f(x) = \sum_{i=1}^{m} \sum_{j=1}^{m} (||\underline{x}^i - \underline{x}^j||^2 - d_{ij}^2)^2. \tag{4}$$

Let us model the function as in the file `f_mdgp.m`.

```
function f = f_mdgp(x)
  f = (-2.32868 + (x(1)-x(4))^2 + (x(2)-x(5))^2 + (x(3)-x(6))^2)^2 + ...
    (-6.20702 + (x(1)-x(7))^2 + (x(2)-x(8))^2 + (x(3)-x(9))^2)^2 + ...
    (-14.727 + (x(1)-0)^2 + (x(2)-0)^2 + (x(3)-0)^2)^2 + ...
    (-2.32868 + (x(4)-x(7))^2 + (x(5)-x(8))^2 + (x(6)-x(9))^2)^2 + ...
    (-6.20702 + (x(4)-0)^2 + (x(5)-0)^2 + (x(6)-0)^2)^2 + ...
    (-2.32868 + (x(7)-0)^2 + (x(8)-0)^2 + (x(9)-0)^2)^2;
end
```

By simple inspection we cannot assess the convexity of the function. Therefore we assume the the whole function is not convex and we apply a global optimization method. Notice also that the expression of the residuals is nonlinear.

b) Since in the optimal solution of problem (4) all the constraints (3) are satisfied (by the hypothesis on the data), the value of the objective function in the optimal solution amounts to $\bar{f} = 0$. Therefore it is possible to stop the algorithm when the error between the current solution and the optimal one falls below a prefixed constant $\varepsilon > 0$.

File `multistart.m`: multistart method. Find an approximate solution $\underline{x}^*$ with value $f(\underline{x}^*)$, given:

- an objective function to be optimized `f`
- the number of components of the solution `n`
- a global sub-optimality tolerance `eps`
- a local sub-optimality tolerance `localeps`
- a limit on the maximum number of multistart iterations `maxit`

- a limit on the maximum number of iterations of the local nonlinear optimization method `maxlocalit`
- the local nonlinear optimization method `myLocalOptimAlg`

```
function [xstar, fstar, counter, error, flocals, fstars, nstars] = ...
    multistart(f, n, eps, localeps, maxit, maxlocalit, myLocalOptimAlg)

    fstars = []; % estimate of the optimal value of the global objective function
    nstars = []; % iteration in which is updated the estimate of the optimal value
    flocals = []; % optimal value of the local objective function

    bound = 5;
    x = rnd(n, bound);
    xstar = x;
    counter = 0;
    termination = 0;

    while termination == 0
        fstar = feval(f, xstar);
        if fstar < eps | counter >= maxit
            termination = 1;
        else
            counter = counter + 1;
            [xlocal, flocal, count, err] = myLocalOptimAlg(f,x,localeps,maxlocalit);
           flocals = [flocals; flocal];
           if flocal < fstar
                xstar = xlocal;
                fstar = flocal;
                error = err;
                fstars = [fstars;fstar];
                nstars = [nstars;counter];
            end
            x = rnd(n, bound);
        end
    end

end
```

c) The script `run_ex1_grad.m` runs the algorithm with tolerance `eps = 0.001`, 15 as maximum number of iterations for every call of the local nonlinear optimization method (in this case the steepest descent has been chosen) and 150 as maximum number of multistart iterations. The number of problem variables is $(4-1) \times 3 = 9$. In this case, also for the tolerance of the local method we set `localeps = 0.001 = eps`, but, in general, the two tolerances can be chosen differently: indeed, `eps` concerns the distance in absolute value of the objective function from the known optimal value, while `localeps` concerns the 2-norm of the gradient vector.

```
clear all
close all
f = @f_mdgp;

myLocalOptimAlg = @steepestdescent;

m=3;
n = 3*m;
localeps = 0.001;
eps = 0.001;

% maximum number of multistart iterations
max_iter = 150;

% maximum number of local algorithm iterations
max_local_iter = 15;

%reset seed (to allow for reproducible results)
rand('twister', 0);

% call the multistart method
[xstar, fstar, counter, error, flocals, fstars, nstars] = ...
    multistart(f,n,eps,localeps,max_iter,max_local_iter,myLocalOptimAlg);
xstar
fstar
figure; plot(flocals,'*');
xlabel('multistart iterations');
ylabel('flocals')
figure;
axis([1,max_iter,0,fstars(1)+0.1])
for i = 1:(size(nstars,1)-1)
    plot([nstars(i),nstars(i+1)],[fstars(i),fstars(i)],'LineWidth',2);
    hold on;
    plot(nstars(i),fstars(i),'o','LineWidth',2);
end
plot([nstars(end),counter],[fstars(end),fstars(end)],'LineWidth',2);
hold on;
plot(nstars(end),fstars(end),'o','LineWidth',2);
xlabel('multistart iterations');
ylabel('fstars')
figure;
plot3([xstar(1),xstar(4),xstar(7),0],[xstar(2),xstar(5),xstar(8),0],...
    [xstar(3),xstar(6),xstar(9),0],'-*');
text(xstar(1)-0.05,xstar(2)-0.05,xstar(3)-0.05,'x^*_1','Color',[0 0 1])
text(xstar(4)-0.05,xstar(5)-0.05,xstar(6)-0.05,'x^*_2','Color',[0 0 1])
text(xstar(7)-0.05,xstar(8)-0.05,xstar(9)-0.05,'x^*_3','Color',[0 0 1])
text(0.1,0.1,0.1,'x^*_4','Color',[0 0 1])
nstars
fstars
counter
```

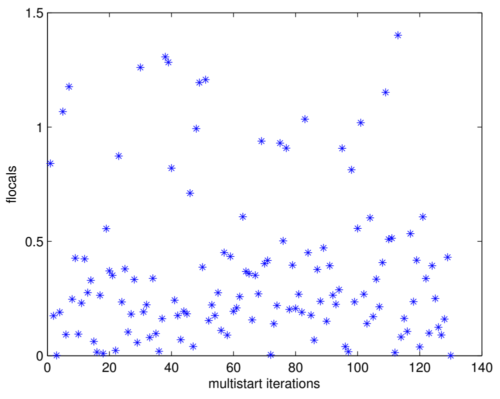
The `nstars` vector contains the values of the multistart iterations whenever the value of tha variable `fstar` (the best objective function value found so far) is updated.

The instruction

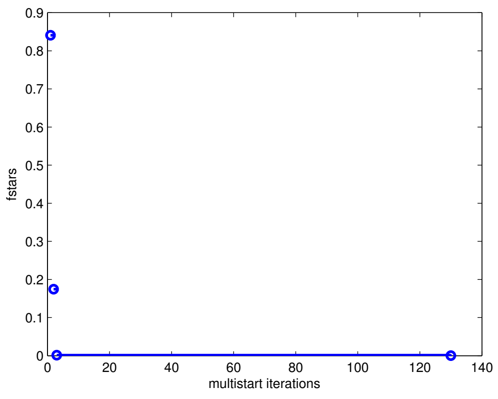
```
rand('twister', 0)
```

is used to fix the seed of the random numbers generator, in order to generate the same sequence of random numbers (and consequently to obtain the same results) at each time that the code is run: it is possible to comment such instruction to observe how the results vary in corrispondence of the variation of the random vector $\underline{x}_0$.

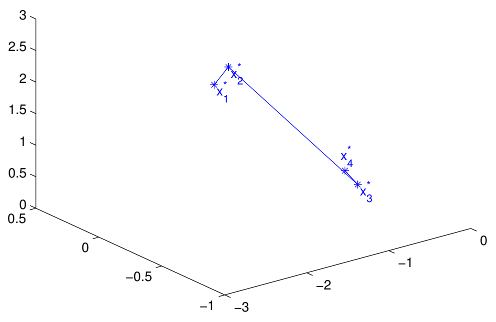
Figure 3 shows the results obtained by applying the multistart algorithm with the steepest descent method as local optimizer.



(a) Best solution found at each iteration.



(b) The evolution of the best objective function value found till the current iteration.



(c) Solution $\underline{x}^*$ found.

Figure 3: Results obtained by applying the multistart algorithm with the steepest descent as local optimizer.

To apply the multistart algorithm with the Newton method (implemented in the file `newton.m`) it is sufficient to replace the instruction

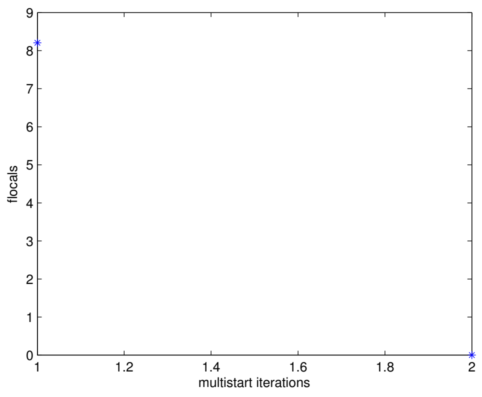
```
myLocalOptimAlg = @steepestdescent
```

with the instruction

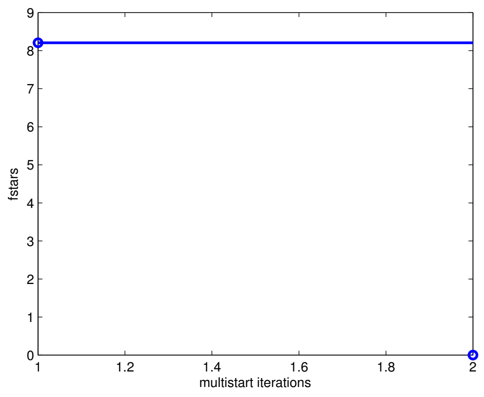
```
myLocalOptimAlg = @newton
```

Figure 4 represents the results obtained by applying the multistart algorithm with the Newton method as local optimizer.

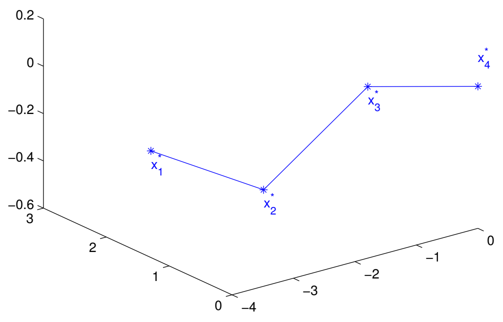
By comparing the results obtained by the two different local methods with the sequence of generated random numbers, it is possible to observe that:



(a) Best solution found at each iteration.



(b) The evolution of the best objective function value found till the current iteration.



(c) Solution $\underline{x}^*$ found.

Figure 4: Results obtained by applying the multistart algorithm with the Newton method as local optimizer.

- with the Newton method the number of multistart iterations is much lower: this result is consistent with the theory, as, if the starting point $\underline{x}_0$ is good enough, the Newton method exhibits a faster convergence rate with respect to the steepest descent;;
- concerning the optimal local solutions obtained at each multistart iteration (see the plot flocals vs multistart iterations), the range of the values provided by the Newton method is about (0,8.5), while the one of the steepest descent is about (0,1.5): also this result is consistent with the theory, as, the steepest descent method is less sensitive than the Newton method with respect to the starting point $\underline{x}_0$.

d) Observe that, to implemtent the Gauss-Newton method, we must provide the residuals vector $\underline{r}$. Let us write the file `r_mdgp.m`:

```
function r = r_mdgp(x)
  r(1) = -2.32868 + (x(1)-x(4))^2 + (x(2)-x(5))^2 + (x(3)-x(6))^2;
  r(2) = -6.20702 + (x(1)-x(7))^2 + (x(2)-x(8))^2 + (x(3)-x(9))^2;
  r(3) = -14.727  + (x(1)-0)^2 + (x(2)-0)^2 + (x(3)-0)^2;
  r(4) = -2.32868 + (x(4)-x(7))^2 + (x(5)-x(8))^2 + (x(6)-x(9))^2;
  r(5) = -6.20702 + (x(4)-0)^2 + (x(5)-0)^2 + (x(6)-0)^2;
  r(6) = -2.32868 + (x(7)-0)^2 + (x(8)-0)^2 + (x(9)-0)^2;
end
```

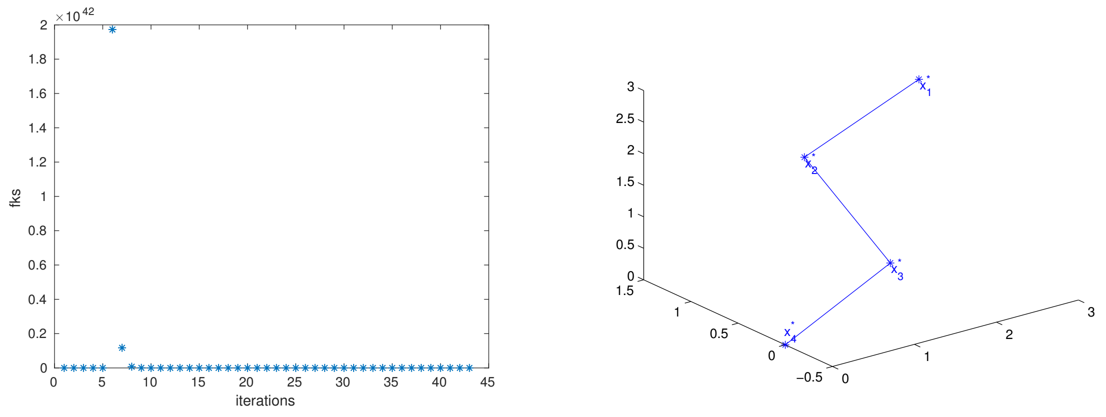
We implement the Gauss-Newton method in the file `gaussnewton.m`

```
% Gauss-Newton method
function [xk, fk, counter, error, xks, fks] = ...
    gaussnewton(r, x0, epsilon, maxiterations)

  xks = [x0'];
  fks = [norm(feval(r,x0))^2];

  xk = x0;
  counter = 0;
  error = 1e300;

  while error > epsilon && counter < maxiterations

    counter = counter + 1;

    J = jac(r, 6, xk);
    d = -pinv(J)*feval(r,xk)';
    alpha = 1;
    xk = xk + alpha * d;

    fk = norm(feval(r,xk))^2;
    error = fk;

  xks = [xks; xk'];
  fks = [fks; fk];
  end

end
```

We choose arbitrarily the starting point $\underline{x}^0 = (\underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0})$.
We execute the script `run_ex1_gaunew.m`.

```
clear all
close all
r = @r_mdgp;

n = 9;
fsol = 0;
eps = 0.001;
max_iter = 100;

x0 = zeros(n,1);

[xstar,fstar,counter,error,xks,fks] = gaussnewton(r, x0, eps, max_iter);

xstar
fstar
plot(fks,'*');
xlabel('iterations');
ylabel('fks');


figure;
plot3([xstar(1),xstar(4),xstar(7),0],[xstar(2),xstar(5),xstar(8),0],...
    [xstar(3),xstar(6),xstar(9),0],'-*');
text(xstar(1)-0.05,xstar(2)-0.05,xstar(3)-0.05,'x^*_1','Color',[0 0 1])
text(xstar(4)-0.05,xstar(5)-0.05,xstar(6)-0.05,'x^*_2','Color',[0 0 1])
text(xstar(7)-0.05,xstar(8)-0.05,xstar(9)-0.05,'x^*_3','Color',[0 0 1])
text(0.1,0.1,0.1,'x^*_4','Color',[0 0 1])
```

Figure 5 represents the results obtained by applying the Gauss-Newton method with 100 as maximum number of iterations and 0.001 as tolerance. Notice that $f\left(\underline{x}^0\right) = \sum_{i,j} d_{ij}^4 = 310.2070$ and $f\left(\underline{x}^1\right) = 6.1586 \cdot 10^{18}$, indicating that the direction determined at each step is not necessary a descent direction. As it is possible to see in the plot, the method obtains a solution $f\left(\underline{x}^k\right)$ within the prefixed tolerance after 42 iterations.



(a) Values of the objective function for the sequence of points generated.

(b) Solution $\underline{x}^*$ found.

Figure 5: Results obtained by applying the Gauss-Newton method.

Esercizio 2: addestramento di reti neurali

Si considerino i valori di output associati a ogni neurone $\underline{x} = (x_1, x_2, x_3, x_4, x_5)$ dove x_i è l'output dell'i-esimo neurone; si considerino inoltre i pesi sugli archi $\underline{u} = (u_1, u_2, u_3, u_4, u_5, u_6)$. Siano $\underline{y} = (y_1, y_2)$ i valori di input ai neuroni 1, 2 nello strato di input, e sia $z = x_5$ l'output della rete.

Gli output di ogni neurone sono legati dalle seguenti equazioni:

$$\begin{aligned} x_5 &= \varphi(u_5x_3 + u_6x_4) \\ x_3 &= \varphi(u_1x_1 + u_2x_2) \\ x_4 &= \varphi(u_3x_1 + u_4x_2) \\ x_1 &= y_1 \\ x_2 &= y_2. \end{aligned}$$

L'uscita della rete è quindi pari alla funzione $h : \mathbb{R}^2 \to \mathbb{R}$, parametrizzata da $\underline{u}$, pari a:

$$h(\underline{u}, \underline{y}) = \varphi\{u_5\varphi\left[u_1y_1 + u_2y_2\right] + u_6\varphi\left[u_3y_1 + u_4y_2\right]\}.$$

Si tratta quindi di risolvere il problema:

$$\min_{\underline{u}} \sum_{i=1}^{4} ||z_i - h(\underline{u}, \underline{y}_i)||^2,$$

dove $(\underline{y}_i, z_i)$ sono le triple date in tabella nel testo del problema.

Il codice per l'algoritmo del gradiente è dato dai file `grad.m` e `steepestdescent.m`, che possono essere trovati nel testo del laboratorio precedente. Il codice per l'algoritmo multistart è in `multistart.m`.

Lo script `run_ex2.m` lancia l'algoritmo con una tolleranza `eps = 0.001` e un limite massimo di 250 iterazioni per ogni chiamata del metodo di ottimizzazione locale (in questo caso è stato scelto il metodo del gradiente) e di 250 multistart. Le variabili del problema sono i pesi degli archi della reurale, pertanto il loro numero è 6.

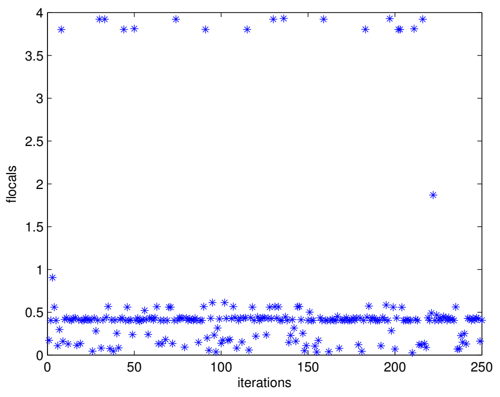
```
clear all
close all
f = @f_nnet;
myLocalOptimAlg = @steepestdescent;

n = 6; %number of arc weights that have to be estimated
eps = 0.001;
localeps = 0.001;
max_iter = 250;
max_local_iter = 250;

%reset seed (to allow for reproducible results)
rand('twister', 0);

% chiamo il metodo multistart
[ustar, fstar, counter, error, flocals, fstars, nstars] = ...
    multistart(f,n,eps,localeps,max_iter,max_local_iter,myLocalOptimAlg);
ustar
fstar
figure; plot(flocals,'*');
xlabel('iterations');
ylabel('flocals')
figure;
axis([1,max_iter,0,fstars(1)+0.2])
for i = 1:(size(nstars,1)-1)
    plot([nstars(i),nstars(i+1)],[fstars(i),fstars(i)]);
    hold on;
    plot(nstars(i),fstars(i),'*');
end
plot([nstars(end),max_iter],[fstars(end),fstars(end)]);
hold on;
plot(nstars(end),fstars(end),'*');
xlabel('iterations');
ylabel('fstars')

%print the output for logical inputs
f_hmap(ustar, [0, 0])
f_hmap(ustar, [0, 1])
f_hmap(ustar, [1, 0])
f_hmap(ustar, [1, 1])
```

La Figura 6 riporta i risultati ottenuti lanciando lo script `run_ex2.m`. Osserviamo che, in alcune iterazioni dell'algoritmo multistart, la stima dell'ottimo locale ha valore nettamente diverso (circa 4) rispetto all'ottimo globale: ciò indica che la convergenza del metodo del gradiente varia sensibilmente in funzione del punto di partenza, il che motiva la scelta di un approccio multistart.

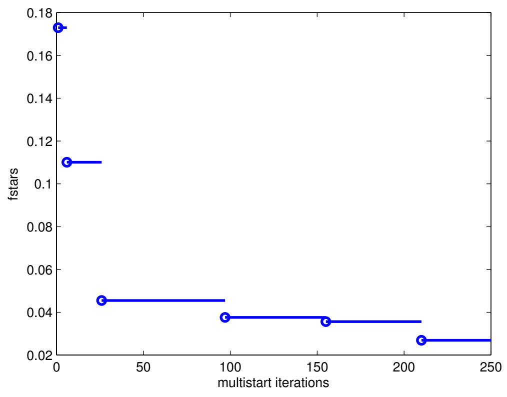
Le ultime righe dello script `run_ex2.m` stampano l'uscita della rete neurale quando i pesi degli archi sono quelli stimati dal nostro algortimo di ottimizzazione (ovvero pari a `ustar`) per i quattro possibili ingressi logici (dato che gli input sono 2, i possibili valori sono (0,0), (0,1), (1,0), (1,1)). Osservandone i valori, si conclude che i chip sono probabilmente delle porte logiche XOR, dove

$$XOR(y_1, y_2) = (y_1 \wedge \neg y_2) \vee (\neg y_1 \wedge y_2)$$

Si noti che la funzione XOR non è lineare, quindi non è sufficiente utilizzare metodi di classificazione lineare per identificarla. Si faccia riferimento al testo classico *Perceptrons* di M.



(a) Miglior soluzione trovata a ogni iterazione.



(b) Andamento del valore della miglior soluzione trovata fino all'iterazione considerata

Figure 6: Risultati ottenuti lanciando lo script `run_ex2.m`.

Minsky e S. Papert (1969) dove gli autori mostrano come una rete di percettroni a singolo strato possa apprendere la classificazione di soli pattern linearmente separabili. Nella Figura 7 si può notare che non esiste un iperpiano che separa i punti "neri" (dove la funzione XOR vale 0) dai punti "bianchi" (dove la funzione XOR vale 1); le linee tratteggiate sono iperpiani separatori che però non riescono a classificare correttamente tutti i punti. Nel nostro caso la rete neurale è però di tipo *feed-forward* multistrato, con un layer nascosto.

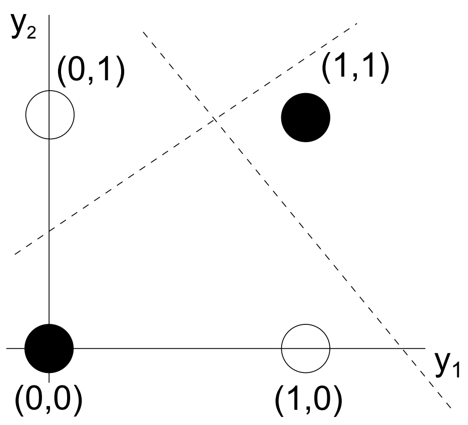
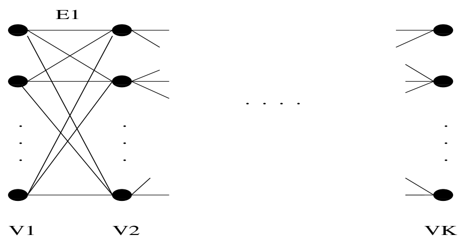


Figure 7: Rappresentazione grafica del dominio della funzione XOR.

Reti Neurali

La fase di addestramento di una rete neurale, tramite l'algoritmo di back-propagation, è in sostanza la soluzione iterativa di un problema di minimi quadrati per mezzo di una versione stocastica del metodo del gradiente. Si consideri una rete multistrato rappresentata da un grafo K-partito $G = (V, E)$ dove V è partizionato in $V_1, \ldots, V_K$, E in $E_1, \ldots, E_{K-1}$ e ogni (V_k, V_{k+1}, E_k) $(k \leq K-1)$ è un grafo bipartito completo. I nodi di questo grafo corrispondono alle unità di elaborazione ("neuroni artificiali") e i lati ai collegamenti tra loro.



A ogni lato del grafo è associato un peso u_k^{sj}, dove $k \leq K$ è lo strato e $\{s, j\}$ indica il lato, con $s \in V_k$ e $j \in V_{k+1}$. A ogni nodo viene associato un valore di output x_k^j, dove k è lo strato e j è il nodo in V_k. Il valore di input del j-esimo nodo nel $(k+1)$-esimo strato è una funzione lineare dei valori di output allo strato precedente: $\sum_{s=1}^{|V_k|} u_k^{sj} x_k^s$, per $j \leq |V_{k+1}|$. L'output del j-esimo nodo nel $(k+1)$-esimo strato è una funzione non lineare φ del suo input, detta funzione di attivazione. Quindi

$$x_{k+1}^j = \varphi\left(\sum_{s=1}^{|V_k|} u_k^{sj} x_k^s\right) \qquad j \leq |V_{k+1}|, 2 \leq k \leq K-1.$$

Di solito si utilizza la funzione di attivazione $\varphi(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}}$, oppure la funzione sigmoidale $\psi(\xi) = \frac{1}{1+e^{-\xi}}$. Si assume che lo strato $k = 1$ sia lo strato di input dell'intera rete neurale: il valore dell'output dei nodi del primo strato viene specificato dall'utente; insieme costtuiscono l'input della rete. I valori di output dell'ultimo strato forniscono la risposta della rete. In pratica, la rete è una mappa h, parametrizzata dal vettore dei pesi sui lati $\underline{u}$, che trasforma un vettore di input $\underline{x}_1 \in \mathbb{R}^{|V_1|}$ in un vettore di output $\underline{x}_K$, in modo che $\underline{x}_K = h(\underline{u}, \underline{x}_1)$.
Supponiamo di avere un insieme di m coppie input/output $T = \{(\underline{y}_i, \underline{z}_i) \mid i \leq m\}$. Durante la fase di addestramento della rete neurale, i pesi $\underline{u}$ vengono modificati in modo da minimizzare una misura complessiva dell'errore su tutte le coppie di T, che dipende dalla differenza tra l'output ottenuto per ogni input $\underline{y}_i$ e l'output desiderato $\underline{z}_i$. Al termine di questa fase, la rete è pronta per l'uso: si spera che a ogni nuovo vettore di input (di cui non si conosce l'output a priori) la rete fornisca un output "sensato" in base alle coppie in T sulle quali è stata addestrata.
L'algoritmo di "back-propagation" utilizzato per l'addestramento non è altro che una variante dell'algoritmo del gradiente applicato strato per strato al problema di minimi quadrati

$$\min_{\underline{u}} \sum_{i=1}^{m} ||\underline{z}_i - h(\underline{u}, \underline{y}_i)||^2.$$

Il fatto che le variabili siano i pesi $\underline{u}$ e che h in generale può essere non lineare e non convessa rende l'intero problema non convesso. Si consideri poi che nei problemi pratici m è di solito un numero molto grande; gli approcci risolutivi devono quindi limitarsi a algoritmi molto semplici (come varianti del metodo del gradiente) e spesso applicati in modo stocastico (a ogni iterazione viene calcolato solo il gradiente rispetto ad un termine della somma dei quadrati).

Figure 8: Neural Network optimization problem.