# **Duality, Optimization and Minimum Principles.**

The operation of *optimization* is the task of finding the *minimum* or *maximum* of a function *F(x1,x2,...,xn)* of several variables in the presence of *contraints* on those variables. Constraints can appear as *equalities* of the form ***Ax*** = ***b***, *inequalities* of the form ***Ax*** *≥* ***b***,*xj ≥ 0*, or other more complicated forms.

To understand exactly what we are talking about we can go back to the familiar example of *ordinary* *least squares*. Here we have a matrix ***A*** whose *n* columns are independent, and a vector ***b*** with *m* components. Most often, there will be no solution for ***Au*** *=* ***b*** because the system of equations is *overdetermined*, with more equations than unknowns, which causes ***A*** to have more *rows* than *columns* (*m*>*n*). For example:

***A u b***

If there is an exact solution to ***Au*** *=* ***b*** it means that ***b - Au*** *=***0**. On the other hand, if there is no solution, there is an infinite number of values of ***u*** such that ***b - Au*** *=* ***e*** with ***e*** being a vector different from **0**. Thus, the *best* we can do is to search for a vector such that the *error* ***e*** is as close as possible to **0**. Since the elements of ***e*** can be both positive and negative, to find means to *minimize* the value of the *squared length* ||***b - Au***||2 = ||***e***||2. We have learned that is the projection of ***b*** ontothe *column space* of ***A***, and that is found directly by solving the *normal equation*:



Given the normal equation, the equality solves also:

In fact:

***e*** is perpendicular to all the columns of ***A*** or, in other words, ***e*** is the projection of ***b*** onto the *left null space* of ***A*** (*null space of* ***AT***).

The *least squares* problem can be viewed as two simultaneous problems:

***PRIMAL PROBLEM***  minimize projection of ***b*** onto**N*(AT)***

to find the that solves:

***DUAL PROBLEM*** minimize projection of ***b*** onto**C*(A)***

*subject to* (s.t.) the *constraint*:

The two highlighted matrix equations correspond to two systems of linear equations:

and can be combined into a single *block* matrix equation:

By solving the *dual problem*, and ***e*** are found together as they add up to ***b***.

The dual problem for the first time focuses our attention on the importance of *constraints* and on the strategy to handle them in minimizing a function. For example, let's consider how we would minimize the function of the *dual problem*:

in which both ***e*** *and* ***b*** have two components,and ***A*** has one column.

**Lagrange** suggested a general strategy to achieve this result:

the key is to place the *constraint* (in the form *constraint = 0*) in the function itself, multiplied by a *multiplier* ***λ***

We obtain a new function called a ***Lagrangian***:

As for any other function, in order to find the minimum we set the derivative of ***L*** with respect to each component of ***e*** and to as = 0.

In matrix format:

For example, if , the ***primal problem*** solution is:

A = [1; 3];

b = [4;7];

u = A\b

Au = A\*u

e = b - Au

The solution of the ***dual problem*** is:

**S ulong blong**

S = [1 0 1;0 1 3;1 3 0]

b\_long = [4;7;0]

u\_long = S\b\_long

lambda = u\_long(3)

e = u\_long(1:2)

We can see how the **Lagrange** multiplier is actually the solution of the ***primal problem***. More generally, if the matrix ***A*** is *m x n* then ***e*** has *m*components and has *n*components. Also notice how the Lagrangian solution has the form of the ***dual problem*** solution:

If we were to solve the matrix equation by Gaussian elimination we would subtract from the 2nd row the 1st row multiplied by , which would lead to:

in which the *normal equation*  appears clearly in the 2nd row.

The block matrix ***S*** in this equation:

is called the *saddle point matrix* or Karush-Kuhn-Tucker *KKT* matrix, and represents the prototype for all matrices involved in optimization problems. In most favorable cases this matrix is *invertible*, but is never *positive definite*. In fact, we can see how during elimination the identity matrix ***I*** (*m x* *m*)will give *m* *positive pivots* and (*n x n*), called the *Schur complement*, will give *n**negative pivots*.

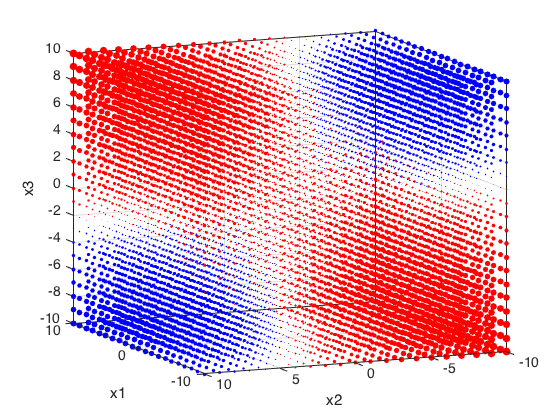
We can check this by carrying out a symbolic **LU** factorization of ***S***:

***S L U***

S = sym(S)

[L,U] = lu(S)

The upper matrix ***U*** shows the result of the Gaussian elimination with the *pivots* on the diagonal, while the lower triangular ***L*** shows the *multipliers* of the 1st and 2nd row that must be subtracted from the 3rd row to obtain ***U***.

Therefore, the KKT matrix is *indefinite.* We recall here that a matrix is *positive definite* if for any vector ***x, xTAx*** >0.For example, in the case of the ***S*** matrix, we can calculate the value of the product ***xTSx*** using a range of values from -10 to 10 for each of the three coordinates of ***x***:

x1 = [-10:10]; x2 = [-10:10]; x3 = [-10:10]; npoints = length(x1)

X = zeros(3,npoints^3); E = zeros(1,npoints^3);

n = 0

for i = 1:npoints

x = x1(i);

for j = 1: npoints

y = x2(j);

for k = 1: npoints

z = x3(k);

n = n+1;

X(:,n) = [x;y;z];

E(n) = X(:,n)'\*S\*X(:,n);

end

end

end

Epos = E > 0; Eneg = E < 0;

figure;

scatter3(X(1,Epos)',X(2,Epos)',X(3,Epos)',E(Epos)'/15,'r','filled'); hold on

scatter3(X(1,Eneg)',X(2,Eneg)',X(3,Eneg)',-E(Eneg)'/15,'b','filled')

box on; grid on; xlabel('x1');ylabel('x2');zlabel('x3');

The scatter plot of the ***xTSx*** product with respect to the three coordinates of ***x*** shows clearly that in some directions the value of the product is > 0 (red dots), while in other directions it is < 0 (blue dots). Thus, instead of having a *global minimum* there is a *saddle point*, thereby the name for ***S*** as '*saddlepoint matrix'*.

In order to derive a more general form of the KKT matrix it is worth looking at a slightly more complex example of *linear least squares* in which *weights* (as a square matrix) are assigned to ***b***. We have seen that in this case, we replace with **,**  with , and becomes ***C***.

which can be rearranged as:

The ***primal problem*** is to minimize:

to find the that solves , and thus the *unchanged* equality:

where and are different from the corresponding and that we would have in the non-weighted case, but still add up to .

The ***dual problem*** is to minimize:

*subject to* (s.t.):

We have again two systems of linear equations:

which can be combined into a single *block* matrix equation introducing and :

Notice the similarity between the ***dual problem*** in the absence of weights:

and the ***dual problem*** in the presence of weights:

A = [1; 3];

b = [4;7];

sigma\_b = [.3;.5]

W = diag(1./sigma\_b)

C = W'\*W

invC = inv(C)

S = [invC(1,1) invC(1,2) A(1);invC(2,1) invC(2,2) A(2);A(1) A(2) 0]

b\_long = [b;0]

u\_long = S\b\_long

w = u\_long(1:2)

u = u\_long(3)

e = invC\*w

Au = A\*u

and add up to ***b***, but they are no longer orthogonal to each other (*oblique projections*).

Au+e

Au'\*e

and are orthogonal to each other, but they do not add up to ***b***.

Au+w

Au'\*w

is now ***C orthogonal*** to the columns of ***A***, as expected from

A'\*C\*e

Gaussian elimination on the new KKT matrix is carried out by subtracting times the 1st row from the 2nd row:

in which the *normal equation*  for *weighted* *least squares* appears in the 2nd row with the *Schur complement* .

An understanding of the function that is being minimized in the *weighted* *least squares* and the constraints imposed on the minimization is obtained by looking at the *Lagrangian* of the dual problem:

We set the derivative of ***L*** with respect to each component of and to as = 0. These are the *Kuhn-Tucker* *optimality* equations and the *Lagrange multiplier* equation, respectively:

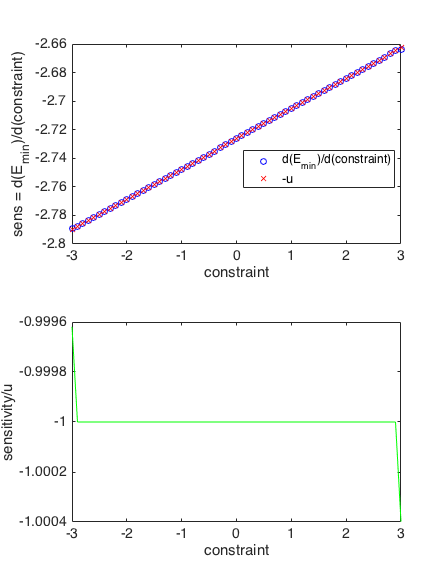
In matrix format:

Thus, the *objective function* minimized is **,** and the *constraint* is . The Lagrange multiplier corresponds to .

Since we have derived a KKT matrix representing the process of optimization with constraints, and we know that this process is associated with calculating the partial derivatives of a *Lagrangian*, we are now ready to ask the question:

what is the meaning of the *Lagrange multipliers*?

We gain an insight into this question by looking at our example of *weighted* *least squares* using the KKT matrix to compute , , and from these the *objective function* minimum:

for a range of values on the *right hand side* (RHS)of the constraint:

We notice here that the value on the RHS of the constraint (**0** in this case) appears at the last position of the extended vector in the KKT matrix. Thus, it is this position that is modified when we test different values of the constraint.

Finally, we calculate the derivative of with respect to this constraints. This calculation is often referred to as a *sensitivity analysis*:

A = [1; 3];b = [4;7];b3\_vec = [-3:0.1:3];

nfs = length(b3\_vec)

E\_min = zeros(nfs,1);u\_vec = zeros(nfs,1);

for i = 1:nfs

b3 = b3\_vec(i);

b\_long = [b;b3]

u\_long = S\b\_long

w = u\_long(1:2);

e = invC\*w

u\_vec(i) = u\_long(3);

E\_min(i) = 0.5\*(b-e)'\*C\*(b-e);

end

sens = gradient(E\_min,0.1)

figure;

plot(b3\_vec,sens,'ob'); hold on

Here we take the negative of u\_vec:

plot(b3\_vec,-u\_vec,'xr')

xlabel('constraint')

ylabel('sens = d(E\_m\_i\_n)/d(constraint)')

legend('d(E\_m\_i\_n)/d(constraint)','-u','Location','best')

The plot shows that is the *negative* of the derivative of with respect to the *right hand side* of the constraint.

In fact, in general if we want to to minimize:

with Lagrangian:

Thus, in the *least square* process the *Lagrange multiplier* reflects the *sensitivity* of the *objective function*  to a change in this constraint.

We can easily derive the form that the Lagrangian takes in a constrained optimization in the general cases of *weighted least squares*:

with the two equations (the derivatives of ***L*** set to 0):

leading to the KKT matrix:

**SPECIAL TOPIC**

**Taylor series.**

*Taylor series for a function of a Single Variable.*The *Taylor series* expansion of a function about the point is the infiniteseries:

A Taylor series is meaningful only if all the derivatives of exist at and the series converges. In general, convergence occurs only if *x* is sufficiently close to *a*, that is if:

where is the *radius of convergence* of the series.

Another useful form of the Taylor series is the expansion about an arbitrary value of *x*:

Since it is not possible to evaluate all the terms of an infinite series, the effect of truncating the series is of great practical importance. Keeping the first *n*+1 terms, we have:

where is the *truncation error* (sum of the truncated terms starting with the term including ). An information conveyed by this equation is that:

which is a concise way of saying that the truncation error is *of the order of* , or behaves as . If is within the radius of convergence, then

which means that the error is always reduced if a term is added to the truncated series.

*Taylor series for a function of Several Variable.*If *f* is a function of the *m* variables , then its Taylor series expansion about the point is:

which can be written in matrix form as:

where the vector is the *gradient* of and is the *Hessian* matrix of .

**Newton-Raphson.** A common problem in many applications is to determine the value of ***x*** for which a given non-linear function *f*(***x***) takes the value of 0. The solutions are known as the *roots* of the equation *f*(***x***) = 0, or the *zeros* of the function *f*(***x***). In most cases, the problem is not confined to a single equation, but is multi-dimensional, namely:

or using a scalar notation:

The most effective way of computing ***x*** is the *Newton-Raphson* method. In order to derive this method we start with the *Taylor series expansion* of about the point ***x***:

Dropping the term of order we obtain the *linear approximation* of the function ***f*** in the vicinity of point ***x***:

where is the *n* x *n* *Jacobian matrix* of partial derivatives:

If we assume that ***x*** is our current approximation (or just a guess) of the solution of , then we are justified in thinking that a small change from , will give us a better solution. We can obtain this correction by setting the value in the linear approximation of ***f*** in the vicinity of ***x***:

This new equation allows us to calculate the step we have to take in order to obtain an improved solution. Now we are ready to outline the complete *Newton-Raphson* algorithm:

Start with a solution vector

**⇓**

Evaluate

**⇓**

Calculate the Jacobian matrix

**⇓**

Solve for

**⇓**

The iteration continues until **,** whereis the *error tolerance*. Typically the partial derivatives that go in the Jacobian are calculated by one of the *finite difference approximations* as:

where *h* is a small increment applied to and represents a unit vector in the direction of . Typically, in differentiating a vector keeping a constant order *n* of accuracy (pronounced ‘big O *n*’), we will use the 2nd forward difference for the first point, the 1st central difference for the interior points, and the 2nd backward difference for the last point.

Notice that the equation is a standard *linear* matrix equation of the type, and is solved the usual way by Gaussian elimination, or linear least squares if *i>j*. In the following example we use a simple implementation of the *Newton-Raphson* algorithm as a function (modified from the original written by Jaan Kiusalaas, 'Numerical Methods in Engineering with MATLAB', Cambridge Press) to solve the system of non-linear equations:

First we set up an anonymous function to represent a system of three non-linear equations (notice the delimitation of each equation with parentheses to form a 3 x 1 vector of equations):

func = @(x) [(sin(x(1)) + x(2)^2 + log(x(3)) - 7); ...

(3\*x(1) + 2^(x(2)) - x(3)^3 + 1);

(x(1) + x(2) + x(3) - 5)];

Here we prepare a trial vector as the starting solution:

x = [1 1 1]

Here we find the roots using different types of finite differences to calculate the Jacobian:

x = newton\_raphson(func,[1 1 1],'for1')

x = newton\_raphson(func,[1 1 1],'for2')

x = newton\_raphson(func,[1 1 1],'cen')

**PRACTICE**

1. Solve the weighted least squares Problem 2 of the CHAPTER 6 PRACTICE section using a KKT matrix and QR factorization.
2. Solve a generalized least squares problem of the same type described in CHAPTER 6 using a KKT matrix and QR factorization. Use the following small script to generate the data ***b***:

npoints = 7;

xvec = [1:npoints]';

int = 5;

slope = 8;

yvec = int + slope\*xvec;

KKT\_lsq\_plot = figure

h(1) = plot(xvec,yvec,'+r');

hold on

set(gca,'Xlim',[0 8],'Ylim',[0 80]);

sigma\_y = 0.2\*yvec;

nsamples = 1000;

yvec\_m = zeros(npoints,nsamples);

for i = 1:nsamples

noise\_0 = normrnd(0,mean(sigma\_y));

corr\_noise = 0.5\*noise\_0 + normrnd(zeros(7,1),sigma\_y);

yvec\_m(:,i) = yvec+corr\_noise;

end

A = [ones(npoints,1) (1:npoints)'];

nvars = size(A,2);

B = (yvec\_m(:,randi(nsamples,10,1)))';

b = mean(B)'

plot(xvec,b,'om');hold on

V = cov(B,1);

C = inv(V);

1. The Newton-Raphson algorithm finds an important application in the *constrained optimization* of functions. For example, consider the problem of finding the *x* and *y* dimensions of a page that will maximize the area of text, *T*, inside the margins, under the constraints that the total area of the page should be *P* = 160 cm2, the side margins should be equal to *a* = 4 cm, and the top and bottom margins should be equal to *b* = 3 cm. The problem can be stated as finding:

There is a solution to this problem by conventional calculus:

*First Order Optimality Condition (FOC) for minimum or maximum*:

However we could as well solve the problem using a *general strategy* that involves *Lagrange multipliers* and bringing the *constraints* inside a *Lagrangian*. In our case the *Lagrangian* is:

We have now a *homogeneous system* of 3 *non-linear equations* (*g1,g2,g3*) in 3 unknown (*x,y,λ*) of which we seek the *zeros*. We proceed with the Newton-Raphson method by initializing a guess vector ***u*** of unknown (*u1=x, u2=y, u3=λ*) and by calculating the values of the 3 functions for the guess vector:

Start with a solution vector

**⇓**

Evaluate

**⇓**

Calculate the *Jacobian* matrix:

**⇓**

Solve for

**⇓**

Write a MATLAB script that uses the Newton-Raphson method to find the *x* and *y* dimensions of the page that will maximize the area of text under the aforementioned constraints.

The same solution can also be derived using the MATLAB function *fmincon* for constrained optimization (Optimization Toolbox):

global Pa

a = 4;b = 3;Pa = 160;

x0 = [10,10]

T = @(x) -(x(1)-2\*a)\*(x(2)-2\*b)

nlincon = @p\_area

Aineq = [];bineq = [];Aeq = [];beq = [];

[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(T,x0,[],[],[],[],[0,0],[],nlincon)

T(x);Pcheck = x(1)\*x(2)

where the *p\_area* function is:

function [ c,ceq ] = p\_area( x)

global Pa

c = [];ceq = x(1)\*x(2)-Pa

end

In this case the *Lagrange multiplier* is provided as the output variable *lambda.eqnonlin* contained inside the output structure *lambda*.