# **Duality and Linear Programming.**

In CHAPTER 7 we have introduced the operation of *optimization* as the task of finding the *minimum* (or *maximum*)of a function *F(x1,x2,...,xn)* (the *objective function*) of one or more variables *xi* (the *solution* or *design variables*) in the presence of *contraints* on those variables.

A special case of optimization occurs when the *constraints* are represented by a matrix equation , , or , in which there are more unknowns than equations(*n* > *m*), and for which the only acceptable solutions are those in which all the elements of are *nonnegative* (*xj*  *0*) or, more generally, larger and/or smaller than a *lower* and/or *upper bound*.

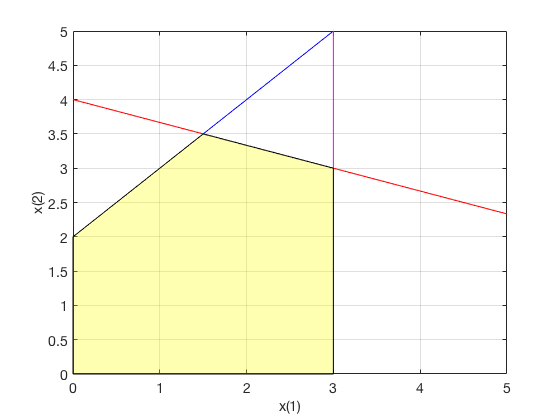
Since the system is *underdetermined,* typically there are infinite solutions to under the constraint *xj*  *0*. The task of *linear programming* is to find the nonnegative (or bounded) solution to that minimizes the *cost* (this is the *objective function*):

Thus, a *linear programming* problem starts with a matrix ***A*** and two vectors, ***b*** and ***c***. Linear programming is the most important application of mathematics to management in different types of industry (e.g., chemical, biotech, financial). Let's consider some very simple examples:

1. We want to *minimize* the objective function:

s.t.

We start by plotting all the constraints in order to find the *feasible region*: this is the region containing all the values of that fulfill the constraint .



LP1 = figure;

constr1 = @(x) -(1/3)\*x + 4;

fplot(constr1,[0 5],'-r')

ylim([0 5])

xlabel('x(1)');ylabel('x(2)')

hold on

constr2 = @(x) x + 2;

fplot(constr2,[0 5],'-b')

constr3 = @(x) 3;

[x2 x1] = fplot(constr3,[0 5])

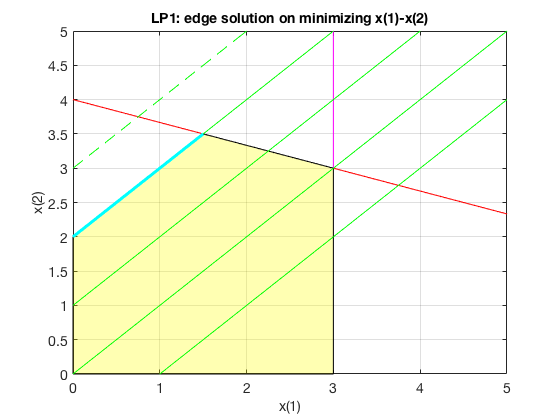
plot(x1,x2,'-m')

% Feasible region

points = [0 3 3 1.5 0; 0 0 3 3.5 2]

fill(points(1,:),points(2,:),'y',’FaceAlpha’,0.3)

box on, grid on

Then we plot the lines corresponding to different values of the objective function:

OF1 = @(x) x-1 % x(1)-x(2) = 1

fplot(OF1,[0 5],'g')

OF2 = @(x) x % x(1)-x(2) = 0

fplot(OF2,[0 5],'g')

OF3 = @(x) x+1 % x(1)-x(2) = -1

fplot(OF3,[0 5],'g')

OF4 = @(x) x+2 % x(1)-x(2) = -2

fplot(OF4,[0 5],'g')

OF5 = @(x) x+3 % x(1)-x(2) = -3

fplot(OF5,[0 5],'--g')

line([0 1.5],[2 3.5],'LineStyle','-',…

'LineWidth',3,'Color','c')

title('LP1: edge solution on minimizing x(1)-x(2)')

hold off

In this case it is clear that there are infinite solutions with lying on the line connecting to : this is an *edge* of the feasible region. This edge is defined by two of the constraints (*active or binding constraints*). The third constraint is *non-binding*.

1. We want to *maximize* the objective function:

We introduce three *slack variables* to convert the inequalities into equalities.

LP2 = figure;

constr1 = @(x) -(1/2)\*x + 1; % x(1) + 2x(2) >= 2

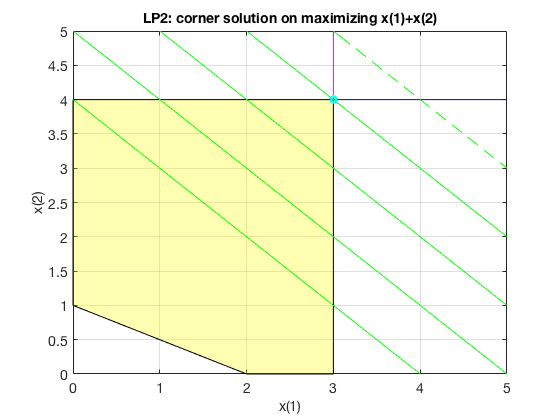
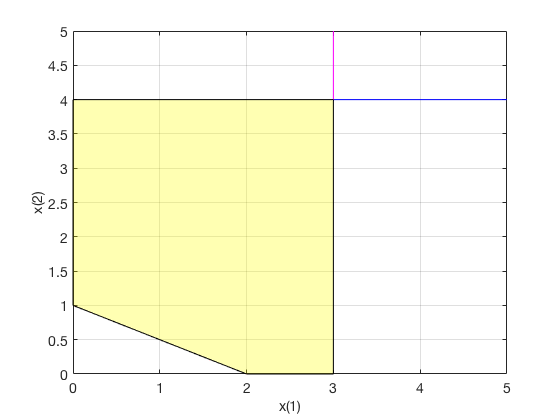
fplot(constr1,[0 5],'-r')

ylim([0 5])

hold on

constr2 = @(x) 4; % x(2) <= 4

fplot(constr2,[0 5],'-b')



constr3 = @(x) 3; % x(1) <= 3

[x1 x2] = fplot(constr3,[0 5])

plot(x2,x1,'-m')

% Feasible region

points = [2 3 3 0 0; 0 0 4 4 1]

fill(points(1,:),points(2,:),'y')

alpha(0.3)

box on, grid on

xlabel('x(1)'),ylabel('x(2)')

Then, we plot the lines corresponding to different values of the objective function:

OF1 = @(x) -x+4 % x(1) + x(2) = 4

fplot(OF1,[0 5],'g')

OF2 = @(x) -x+5 % x(1) + x(2) = 5

fplot(OF2,[0 5],'g')

OF3 = @(x) -x+6 % x(1) + x(2) = 6

fplot(OF3,[0 5],'g')

OF4 = @(x) -x+7 % x(1) + x(2) = 7

fplot(OF4,[0 5],'g')

OF5 = @(x) -x+8 % x(1) + x(2) = 8

fplot(OF5,[0 5],'--g')

plot(3,4,'Marker','o','LineWidth',3,…

'Color','c')

title('LP2: corner solution on maximizing x(1)+x(2)')

hold off

In this case we find that there is a single optimal solution. The point ***x*** = [3 4] is the feasible solution that optimizes the objective function, therefore it represents the *optimal solution*. This solution lies on two of the constraints (*active or binding constraints*). The third constraint is *non-binding*.

We have introduced 3 *slack* variables to convert the inequalities into equalities. We can derive the values of the slack variables at the optimal solution by modifying the original constraint matrix to account for the values that are now known:

A = [-1 0 0;0 1 0;0 0 1]

x = [3 4]

b = [2 - 1\*x(1) - 2\*x(2);4 - 1\*x(2);3 - 1\*x(1)]

s = A\b

At the optimal solution, the *non active* constraint would have value =, and the slack variable *s1* for this constraint (often defined as the *slack* of the constraint) would have value 11-2 = 9. In fact: . The slack of the *active* constraints at the optimal solution is 0.

1. We want to *minimize* the objective function:

Once more we plot all the constraints to find the feasible region:

LP3 = figure;

% x(1) + x(2) >= 20

constr1 = @(x) -x + 20;

fplot(constr1,[0 40],'-r')

ylim([0 40])

hold on

% -2x(1) + 5x(2) <= 150

constr2 = @(x) (2/5)\*x + 30;

fplot(constr2,[0 40],'-b')

constr3 = @(x) 5; % x(1) >= 5

[y x] = fplot(constr3,[0 40])

plot(x,y,'-m')

points = [20 40 40 25 5 5; 0 0 40 40 constr2(5) constr1(5)]

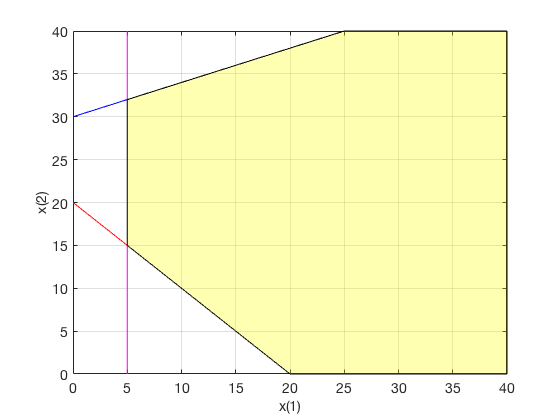
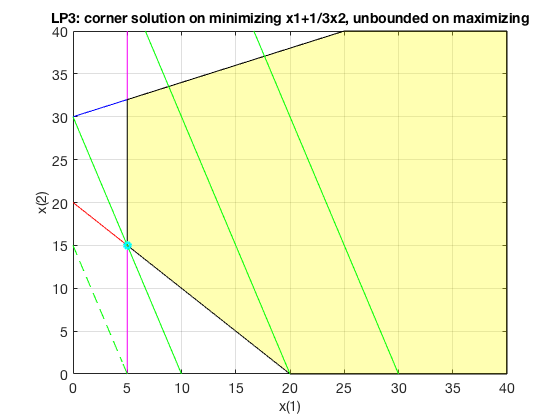
fill(points(1,:),points(2,:),'y')

alpha(0.3)

box on, grid on

xlabel('x'),ylabel('y')

Notice how the feasible region goes on forever off to the right. This region is called *unbounded* because we can go in one direction without limits.



Next, we plot the objective function for different objective values:

OF1 = @(x) -3\*x+90 % x+(1/3)y = 30

fplot(OF1,[0 40],'g')

OF2 = @(x) -3\*x+60 % x+(1/3)y = 20

fplot(OF2,[0 40],'g')

OF3 = @(x) -3\*x+30 % x+(1/3)y = 10

fplot(OF3,[0 40],'g')

OF4 = @(x) -3\*x+15 % x+(1/3)y = 5

fplot(OF4,[0 40],'--g')

plot(5,15,'Marker','o','LineWidth',3,…

'Color','c')

title('LP3: corner solution on minimizing x+1/3y, unbounded on maximizing')

hold off

If we move the objective function line any further we will leave the feasible region. Therefore, we have found our optimal solution at ***x*** = [5 15] with an objective value of 10.

Also in this case we had introduced 3 *slack* variables to convert the inequalities into equalities. We again derive the values of the slack variables at the solution by modifying the original constraint matrix to account for the values that are now known:

A = [-1 0 0;0 1 0;0 0 -1]

x = [5 15]

b = [20 - 1\*x(1) – 1/3\*x(2);150 +2\*x(1)-5\*x(2);5 - 1\*x(1)]

s = A\b

At the optimal solution, the slack of the two *active* constraints is *s1* = -10 and *s2* = 85. In fact:

What would have happened if we had wanted to maximize it? We could keep pushing out the objective function forever. It would never leave the feasible region because the feasible region is unbounded in that direction. This is a case of *unbounded LP* and there is no optimal solution.

**Conclusion**: all the optimal solutions lie on a *corner* (or an *edge*) of the feasible region. This is a fundamental property of linear programming. *If a single optimal solution exists, it is always at a corner of the feasible region*. These corner points are also called extreme points or *basic solutions*.

We may also wonder what would happen if we changed the RHS of the constraint. For example, we could decrease only the 1st element, bringing it down progressively from 20 to 0:

s.t.

Reduce RHS of the 1st constraint to 15.

constr1\_mod1 = @(x) -x + 15; % x(1) + x(2) >= 15

fplot(constr1\_mod1,[0 40],'-r')

Extra area of feasible region due to a change in the RHS of the 1st contraint.

points = [15 20 5 5; 0 0 15 10]

fill(points(1,:),points(2,:),'m');alpha(0.2)

Plot the Objective Function x(1)+1/3\*x(2) for different Objective Values

OF5 = @(x) -3\*x+25 % x(1)+(1/3)x(2) = 8.3333

fplot(OF5,[0 40],'-c')

plot(5,10,'Marker','o','LineWidth',3,'Color','c')

Further reduce RHS of 1st constraint to 10

constr1\_mod2 = @(x) -x + 10; % x(1) + x(2) >= 10

fplot(constr1\_mod2,[0 40],'-r')

points = [10 15 5 5; 0 0 10 5]

fill(points(1,:),points(2,:),'g');alpha(0.2)

OF6 = @(x) -3\*x+20 % x(1)+(1/3)x(2) = 6.6667

fplot(OF6,[0 40],'-c')

plot(5,5,'Marker','o','LineWidth',3,'Color','c')

Further reduce RHS of 1st constraint to 5

constr1\_mod3 = @(x) -x + 5; % x(1) + x(2) >= 5

fplot(constr1\_mod3,[0 40],'-r');ylim([-5 40])

points = [5 10 5 ; 0 0 5]

fill(points(1,:),points(2,:),'k');alpha(0.1)

OF7 = @(x) -3\*x+15 % x(1)+(1/3)x(2) = 5

fplot(OF7,[0 40],'-c')

plot(5,0,'Marker','o','LineWidth',3,'Color','c')

Further reduce RHS of 1st constraint to 0

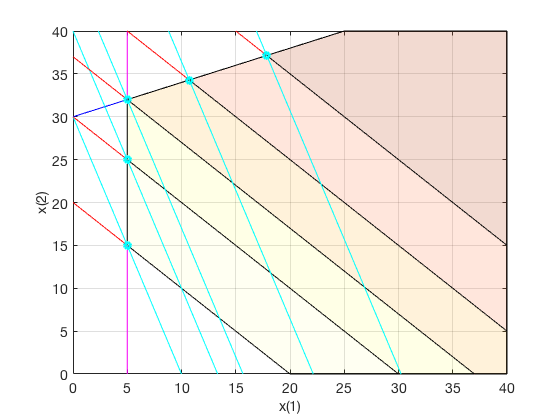
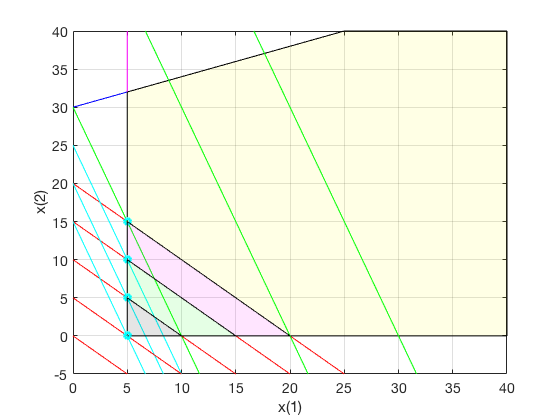
constr1\_mod4 = @(x) -x + 0; % x(1) + x(2) >= 0

fplot(constr1\_mod4,[0 40],'-r');ylim([-5 40])

OF8 = @(x) -3\*x+0 % x(1)+(1/3)x(2) = 0

fplot(OF8,[0 40],'-c')

plot(5,0,'Marker','o','LineWidth',1,'Color','r')



Any time we move a constraint, our feasible region changes.  In this case, our feasible region enlarges to include the magenta, green, and gray area. Now that our feasible region is larger, we can push our objective function (the cyan line) even further than we could before. This gives us a new optimal solution with a new objective value. Notice that when we *decrease* the RHS of this constraint, we find a solution with a *lower* objective value. This is because as a decrease in the RHS makes this constraint less restrictive, and the size of our feasible region larger, we can expect that we might be able to find a "better" solution (= lower minimum).

As we reduce the RHS of the constraint even further to 0, the size of the feasible region does not change, and we are left with the same optimal solution as before. Clearly, if instead of reducing we *increase* the RHS of the constraint we can expect the feasible region to shrink and the optimum of the objective function to increase in value. However, when the RHS of the 1st constraint becomes larger than 37 (see point at x1 = 37, x2 = 0 in the figure above), the feasible region does not intersect any longer the lower bound on x1, and the value of the objective function minimum start changing at a different rate.

We can see clearly this effect if we plot the value of the objective function minimum against the change in the RHS of the constraint:

for i = 1:56

rhs(i) = i -1;

constr1\_mod = @(x) -x + rhs(i);

x\_cross(i) = …

double(solve(-xvar+rhs(i)-(2/5)\*xvar== …

30,xvar));

if x\_cross(i) <= 5

x\_cross\_mod(i) = 5;

else

x\_cross\_mod = x\_cross;

end

y\_cross(i) = …

constr1\_mod(x\_cross\_mod(i));

if y\_cross(i) < 0

y\_cross\_mod(i) = 0;

else

y\_cross\_mod = y\_cross;

end

offset(i) = y\_cross\_mod(i) + 3\*x\_cross\_mod(i);

y\_mod = @(x) -3\*x+offset(i);

OF = @(x,y) x + y/3;

ofv(i) = OF(x\_cross\_mod(i),y\_cross\_mod(i));

end

Shadow\_price = figure;

plot(rhs,ofv,'-b',rhs,ofv,'or','MarkerFaceColor','y')

xlim([-5 60]);ylim([0 35]);xlabel('RHS of constraint');ylabel('Objective Function Value')

grid on;vline(5,{':g','LineWidth',2});vline(37,{':g','LineWidth',2})

title('Shadow price for constraint: x(1) + x(2) >= RHS')

sp = gradient(ofv,1)

annotation('textbox',[0.4 0.05 0.2 0.2],'String','Slope = 0.333...',...

'EdgeColor','r','Color','b','BackgroundColor','y','FaceAlpha',0.3,'FitBoxToText','on');

annotation('textbox',[0.7 0.2 0.2 0.2],'String','Slope = 0.809...',...

'EdgeColor','r','Color','b','BackgroundColor','y','FaceAlpha',0.3,'FitBoxToText','on');

The value of the slope:

the change in the minimum of the objective function value per unit increase in the RHS of the constraint, given all other data remain the same, is called the *SHADOW PRICE* of this constraint.

If the shadow price is *positive*, then the objective value will *increase* by the amount of the shadow price for each unit *increase* in the RHS, and it will *decrease* by the same amount for each unit *decrease* in the RHS. Similarly, if the shadow price is *negative*, then the objective value will *decrease* by the amount of the shadow price for each unit *increase* in the RHS, and it will *increase* by the same amount for each unit *decrease* in the RHS.

Associated with each shadow price is a *range* over which this shadow price holds (dotted green lines in the figure above), typically determined by the intersection of the constraint with other constraints. Furthermore, by looking at the leftmost region in the figure above we also notice that:

*the shadow price of a constraint is always 0 in the range where the constraint is non-binding*

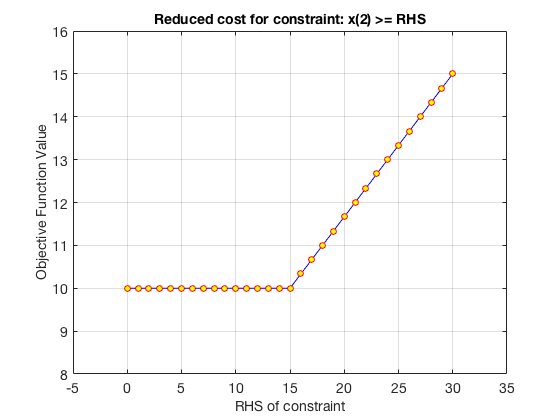
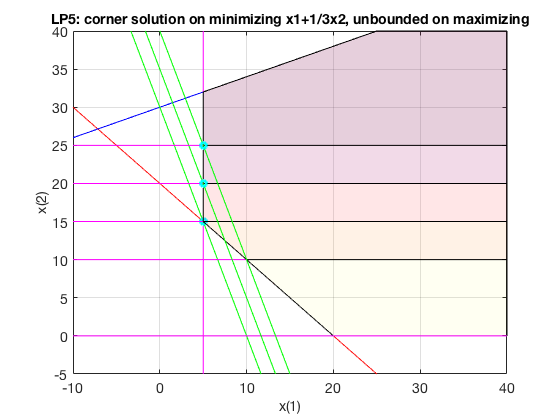
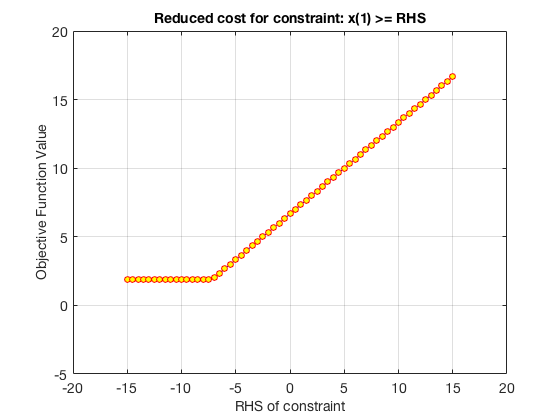
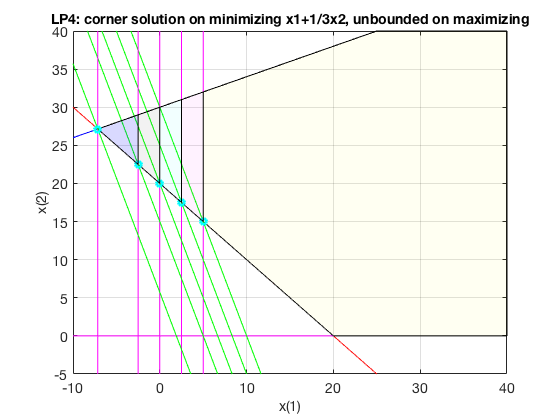
Now, consider again our starting feasible region with the added explicit constraint :

s.t.

Let’s see what happens if we let the *lower bound constraints* on and move. We start by making :

In this case we see that the feasible region becomes *progressively larger* and a *lower minimum* of the objective function is found. If we plot the change in O.F. optimum value against the change in *lower bound* constraint, we notice that once the lower bound on becomes smaller than the intersection between the 1st and 2nd constraint, the O.F. optimum value remains unchanged.

Next we can analyze what happens if we let the *lower bound constraints* on move. We start by making :



In this case we see that the initial increases in *lower bound* *do not change* the O.F. minimum. However, once the *lower bound* equals the value associated with the O.F. minimum, further increases lead to an *increase* in the O.F. value. Also here, the value of the slope:

the change in the objective function value per unit increase in the RHS of the constraint, given all other data remain the same, is the *shadow price* of this constraint. However, it is common practice to refer to the shadow price of the *lower or upper bound constraint* of a variable as the *REDUCED COST* of the constraint:

*Any value of reduced cost of a constraint different from 0 at the optimal solution signifies that the constraint is binding (active) at that solution*

1. We are now ready to examine a real-world case where Linear Programming (LP) finds direct application.

Imagine we work for a company that brews specialty beers. We are starting a new brew that requires a special yeast strain, and in order to produce enough beer for our customers we need to grow 40000 g (wet weight) of this yeast strain: we put this total amount (gr) in a vector, ***b***. We have three options to grow the yeast:

1. option SHAKER (**S**): culture flasks in a large shaker yield = 200 gr/hr
2. option FERM1 (**F1**): a simple fermentor yield = 300 gr/hr
3. option FERM2 (**F2**): a fermentor with oxygen control yield = 500 gr/hr

We put these yields (gr/hr) into a matrix, ***A***. We have all three pieces of equipment, but the cost of running them, including materials and labor, is quite different:

SHAKER cost = $ 2/hr FERM1 cost = $ 5/hr FERM2 cost = $ 8/hr

We put these hourly costs ($/hr) in a *cost vector* .

A = [200 300 500];

b = 40000;

c = [2 5 8]';

In this case ***A*** is *m x n =* 1 *x* 3, and the matrix equation is very simple:

in which the 3 *unknowns* (*x1, x2, x3*) are the *number of hours* that each system must work in order to produce a total of 40000 g of cells. In principle, we could split the task, growing some cells in one system and some in another. For example, we could grow all the needed cells running SHAKER for 20 hours, FERM1 for 20 hours and FERM2 for 60 hours.

The cost of this prep would be:

x\_trial = [20 20 60]'; F = c'\*x\_trial

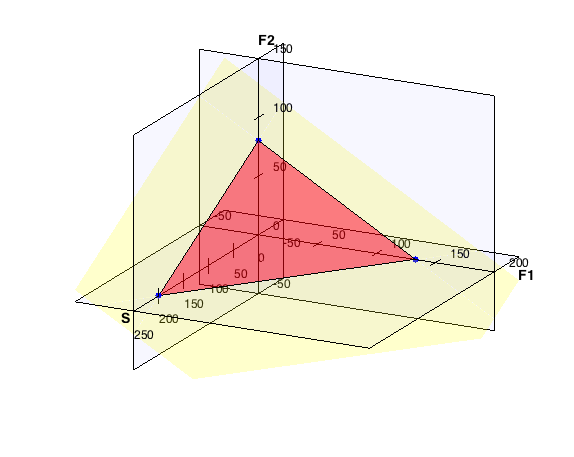
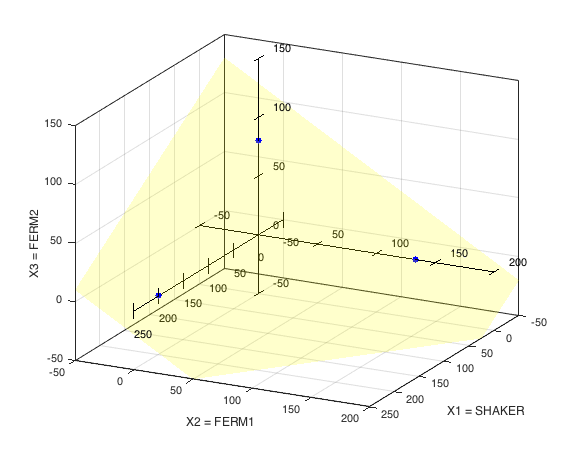
We wonder if we could do better, in terms of reducing our cost, by splitting the jobs differently. We need to minimize:

s.t.

Let's first look at the 1st constraint. The equation :

has infinite solutions. In fact, it is the representation of a *plane* in *n* = 3 dimensional space, with all the possible solutions, providing the coordinates of all the points on the plane.

We recall here that the equation of a plane is where *a, b, c* are the coordinates of the vector normal to the plane and *d* is a coefficient that determines how far the plane is from the origin of the coordinate system. In this case the coefficients in ***A*** are the coordinates of the vector normal to the plane, and ***b*** determines how distant the plane is from the origin. All the points on the plane (shown here as a yellow surface intersecting the axes at the position of three blue circles) are possible solutions to the equation. Since we also have the constraint , the solution plane is itself cut by three orthogonal planes that define the *positive orthant*; what is left of the surface is a triangle (shown here in red) inside which all values of are *nonnegative*. These points are the *feasible* vectors , *and the triangular surface* is the *feasible region* (or *feasible set*) inside which we have to search for the best solution that minimizes the cost . On the ***edges*** of the triangle one component of is 0, at every ***corner*** two components are 0. For example, if the job was carried out using only one growth method (= one corner of the triangle) it would take the following number of hours:



SHAKER (**S**) *x1* = 40000g/200g/h = 200 h

*F* = = 2 $/h x 200 h = $ 400 *y1* = *F/b*= Cost/g = $400/40000g = 0.0100 $/g

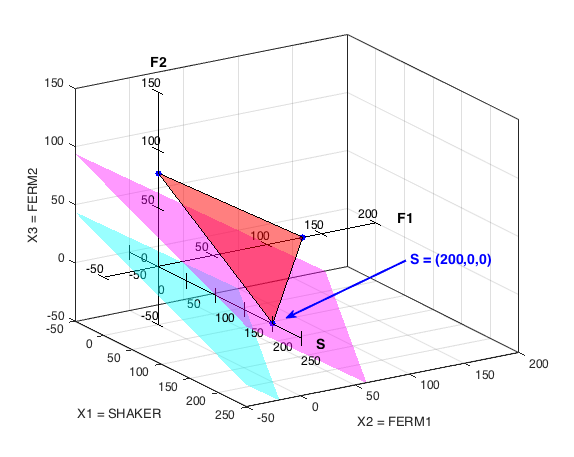
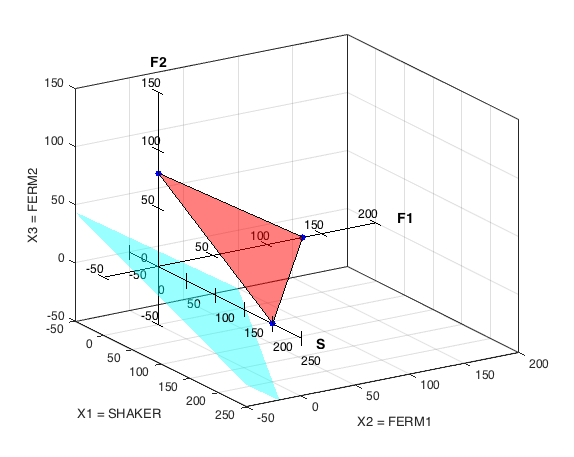
FERM1 (**F1**) *x2* = 40000g/300g/h = 133.3 h

*F* = = 5 $/h x 133.3 h = $ 667 *y2* = *F/b*= Cost/g = $667/40000g = 0.0167 $/g

FERM2 (**F2**) *x3* = 40000g/500g/h = 80 h

*F* = = 8 $/h x 80 h = $ 640 *y3* = *F/b*= Cost/g = $640/40000g = 0.0160 $/g

Notice the introduction of the new variable, *y*, cost/g. The vectors that have 0 cost all lie on the plane: . This plane (shown here as a *cyan* surface) is clearly distinct from, and also has a different *tilt* with respect to the solution plane defined by . In fact, it is the *null space* of , and therefore it goes through the origin. We cannot achieve 0 cost, but as the cost increases new planes *parallel* to move toward the triangle defined by . The first plane to touch the triangle (shown here as a *magenta* surface) has the minimum cost . The point where it touches is the best solution that minimizes the cost. Since the cost plane is tilted with respect to the triangle it is *intuitively* evident that the *minimum cost* plane will first touch the *feasible set* ***triangle*** at one of its ***corners***.



In this simple case, it is quite clear that the minimum cost is accomplished by growing all the yeast using only SHAKER (corner **S**, ), even if it takes longer than using a fermentor (time is not a constraint in this case). In the more general case the *minimum cost* plane might touch the *feasible set* ***polyhedron*** at an *edge*, in which case there would be an infinite number of solutions.

Since in this example, there is only 1 *constraint* equation (***A*** is *m = 1* x *n = 3)*, every corner of the triangle has only 1 component ≠ 0.

More generally, when has *m* equations in *n* variables with *n > m*, and there aren’t any *upper bounds* on the design variable:

1. every corner is a possible ***basic solution*** with *n* coordinates, of which:

*m* coordinates ≠ 0 ***basic variables***

*n-m* coordinates  0 ***non-basic variables***

1. the ***basis*** of the solution are the *m* columns of ***A*** that correspond to the *basic variables*: they are the basis vectors of the *m-dimensional* solution *sub-space*.
2. The number of possible ***basic solutions*** is : the number of ways to choose *m* components out of *n* (function ***n\_choose\_k***:).

It's easy to see how with more equations and unknowns the number of corners can become huge. In our case the *polyhedron* is just a *triangle*, but if we had chosen an example with *m =* 8 equations and *n* = 20 unknowns, the *feasible region* in *n-*dimensional space would be a *polyhedron* with ~126,000 corners (each with *m =* 8 coordinates ≠ 0 and *n-m =* 12 coordinates  0), among which to choose the right solution.

nchoosek(3,1); nchoosek(20,8)

Although in our simple example it is quite obvious that the minimum *cost F* = $ 400, at a Cost/g = *y* = $400/40000g = 0.01 $/g, is achieved by running SHAKER, we could in principle start by calculating the cost of growing all the cells with our best equipment, FERM2, which would be $ 640. In this case *y* = $640/40000g = 0.0160 $/g would be the cost/g of yeast cells.

We introduce some elements of *duality* here:

***Primal problem:***

***x*** *variables vector* (hours) *n* x *1* vector of *design variables*

***c*** *cost vector* (cost/h) *n* x *1* vector of *coefficients* for the design variables

***A*** *constraint coefficients* (g/h) *m* x *n* matrix of *yields/h* with each equipment

***Dual problem:***

***y*** *dual variables vector* (cost/g) *m* x *1* vector of *variables* derived from *F****/b***

***ATy*** *target cost vector* (cost/h) *n* x *1* vector of *target equipment costs* (g/h x cost/g = cost/h) if we wanted to have the same total cost *F =* $640using only one type of equipment at a time

y = 640/b % y is the cost/g

A'\*y % cost/h for each method if we want to have the same total cost using only % each one in turn

In this case would be $ 3.2/h, $ 4.8/h, and $ 8/h. This means that for the growth with SHAKER or FERM1 to cost the same as the growth with FERM2, running SHAKER would have to cost $ 3.2/h instead of $ 2/h, and running FERM1 would have to cost $ 4.8/h instead of $ 5/h. It is clear that running FERM1 would not help, because its cost/h is already higher than $ 4.8/h. We can see this by introducing a new quantity, the *reduced cost vector* ***s***:

s = c - A'\*y

We can't expect to reduce our costs with respect to FERM2 by increasing the use of any equipment corresponding to the positive elements of ***s***; in contrast, we will clearly reduce our costs by increasing the use of equipment corresponding to a negative element of ***s*** (in this case, SHAKER).

The *reduced costs* vector is also called the vector of *dual slack variables* because it representsthe *slack*of the*dual*variables in the inequality . We have seen that a *slack*variable is a variable that is added to an *inequality constraint* to transform it to an *equality*. For example, if our goal had been to produce *at least* 40000 g of yeast cells (that is ) we would modify our problem by introducing a new *slack* variable (actually called a *surplus* variable in this case) so that we would have the same objective function:

but s.t. the *constraints*:

*Slack* or *surplus* variables introduced to represent *inequality constraints* are only used to allow the evaluation of the matrix equation , but do not affect the calculation of the cost function.

We can also look at the process of finding the best growth option from the point of view of *maximization* instead of *minimization*:

Find the *maximum cost* (total grams x cost/g) for the entire yeast prep, provided there is no avenue left to reduce further this cost.

*Maximization* of is the ***dual problem***.The constraint of this dual problem is that the *target cost* for each type of equipment would have to be less or equal than the *actual* equipment cost in the ***primal problem***:

thereby the definition of as a vector of *slack* variables for the constraint of the dual problem.

We may ask the question: why do we want to find the *maximum cost* ? We can understand this by noticing that while it would be nice to have a very small cost/g (e.g. *y* = $0.001/g) for the entire prep, this low cost may not be achievable with the equipment we have. Thus, we need to *increase* little by little our expectation for *y* (cost/g), while at the same time calculating the value of , which givesan estimate of how much running each type of equipment would have to cost per hour if we wanted to make the entire prep with just that type. At some point some element of will *exceed* the corresponding element of ***c*** , indicating we went too far in increasing *y*, and there is now a way to bring it back by increasing the use of the equipment *i*whose *actual* running cost is less than the *anticipated* one for that value of *y*.

We can fully recognize now the *dual nature* of our task of growing the required amount of yeast cells:

The ***primal problem*** is:

minimize (cost/h x hours = total cost)s.t. and

The ***dual problem*** is:

maximize (grams x cost/g = total cost)s.t. and

If either problem has a best solution or , so does the other. In that case we have:

minimum cost (cost/h x hours) = maximum cost (total grams x cost/g)

The following are important features of the *paired problems*:

1. *Minimization* in the primal problem becomes *maximization* in the dual problem.
2. The number of *dual variables* is the same as the number *m* of *primal constraints*.
3. The number of *dual constraints* is the same as the number *n* of *primal variables*.
4. The transpose of the coefficient matrix in the primal problem is the coefficient matrix in the dual problem.
5. The *cost coefficients* in the primal problem become the *right-hand side* of the constraints in the dual problem, and the *right-hand side* of the constraints in the primal problem become the *cost coefficients* in the dual problem.
6. The primal and dual variables both satisfy a *nonnegativity* condition.

*Weak duality* requires that the *maximum cost* does not exceed the *minimum cost* . We can express this relationship also as the value of the dot product . In fact:

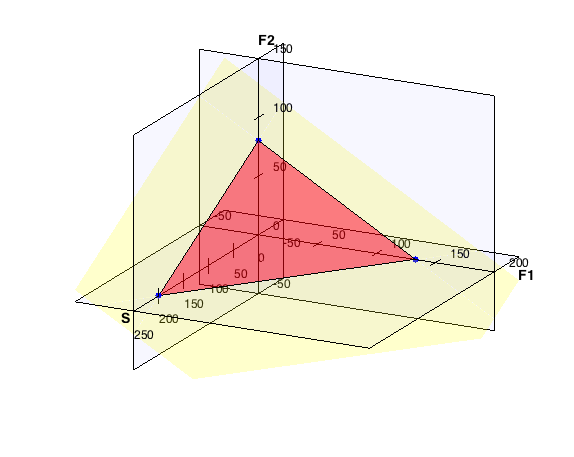
which is achieved only if , since neither nor can be negative at the solution.

*Full duality* occurs when the maximum cost matches the minimum cost . This full dualityrequires (*optimality condition*). In fact, the dot product:

gives us a direct measure of the *duality gap* (often represented with the symbol ).

When full duality is met all the elements of the *reduced cost* vector are indicating there is nowhere to go to achieve a reduced cost for the brewery.

A useful interpretation of the *reduced cost* vector is that it represents the derivative of the objective function *F*with respect to each of the design variables *xi* *at any point along the search of a solution*. For example, consider again the case in which we calculated the total cost of the yeast growth using only FERM2:



We know now that the optimal solution is achieved growing all the yeast with SHAKER in 200 hours. This means we would have to increase our use of SHAKER from *x1* = 0 to*x1\** = 200: this is the same as moving along the edge from **F2** to **S** on the triangle of feasible solutions. Notice that:

Thus, *s1* tells us how much our cost (the objective function *F*) will be reduced per unit increase of *x1*, or:

Alternatively, if at an optimal solution, a design variable has a value of 0 (that is, is not used), the reduced cost for that variable can also be interpreted as the amount by which the corresponding objective function coefficient would have to decrease for that variable to have a positive value (that is, to be used) in an optimal solution.

For example, given an optimal solution:

we have:

which suggests that if *c2* was lowered by 2 from 5 to 3 or *c3* was lowered by 3 from 8 to 5, we could have alternative optimal solutions using only FERM1 or FERM2:

Finally, notice that once we have found the optimal solution the derivative of this solution with respect to the *right hand side* (RHS) ***b*** of the constraint tells us how much our cost will change per unit increase (or decrease) of the amount of yeast we have to make. We have already encountered this derivative as the *dual variable vector*:

When the *optimality condition* is met, the vector is also defined as the ***shadow price*** of the constraint . For each constraint (a row in ***A***) there is a corresponding *shadow price*, and thus has *m* components*.*

In this context, notice that the ***reduced cost*** can be viewed as the *shadow price* of the *lower bound* (nonnegativity) or the *upper bound* constraint (or more generally the *inequality constraints*) for each design variable (i.e., the corresponding change in the objective function value per unit increase in the lower or upper bound of the variable, that is:

We notice at this point that based on their definitions as:

and represent the *Lagrangian multipliers* (up to a sign) of the constraints of the *primal problem*. Therefore, the *dual problem*:

maximize s.t.

can be viewed as the maximization of a function of two Lagrange multipliers:

maximize

It can be shown that this function is always represented by a *convex space*, and thus there always exists a unique global solution to the maximization problem. However, since in the more general case is only the *lower bound* of it is not adviceable to solve the LP problem only from the maximization point of view. As to the practical problem of how to compute and , two classes of algorithms are in strong competition for this purpose, ***simplex methods*** and ***interior point methods***.

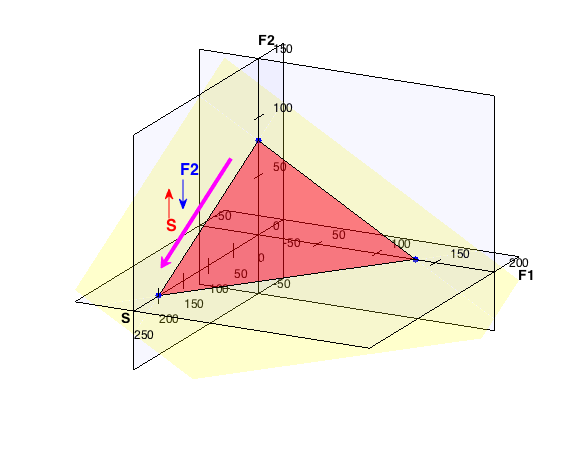
The idea of the ***simplex method*** is simple: move from corner to corner, decreasing the cost at each step until all the edges going out from a corner increase the cost.

We have seen how a corner is a vectorwith *m* nonzero elements and *n-m* 0's. In moving from one corner to the next, the simplex method must decide which component enters the ***basis*** by becoming positive and which component leaves it by becoming 0. The exchange is chosen in order to reduce to cost.Thus**,** each simplex step travels along an edge from corner to corner . One 0 component of becomes positive in (this is the entering variable *xin*) and one nonzero component of (this is the leaving variable *xout*) becomes 0 in .

Here is the plan as applied to our simple example of growing yeast:

**1.** Choose a corner with some initial value of .

**2.** Compute the total cost , the dual cost variable , and the *reduced cost* vector. The entering variable *xin* is the one that gives the *most negative reduced cost* (the greatest cost reduction).

**3.** As we add more of the entering variable *xin* to lower the cost, the other positive components are adjusted to keep . This result can be visualized by noting that as we move from one corner to another along an edge of the triangle the value of the entering variable increases while that of the leaving variable decreases and finally becomes zero at the new corner. Thus, the first positive element of to reach 0 as a consequence of the adjustment is the leaving variable *xout*. This means a new corner has been reached, and the process can start again.

**4.** When all reduced costs are positive, no zero component can become positive without increasing and the process stops with the current being the best solution .

In practice, we use a submatrix (for ***basis***) of with *m* columns corresponding to the nonzeros in . The matrix and the cost vector are split in two parts matching the positive and zero components of at the current .

The reduced cost vector component is the key. If , all edges from increase the cost. In that case ***s***  0; then is the optimal and is the optimal .

We can see this more clearly by solving our brewery problem with the MATLAB code of a very simple ***Simplex*** function (slightly modified from the original at:

<http://math.mit.edu/~gs/cse/codes/simplexcode.m> ):

% function [x,y,cost] = simplex(A,b,c,basis)

A = [200 300 500];

b = 40000;

c = [2 5 8]';

Here we choose the ***basis*** from which we start: these are the columns of the matrix corresponding to the *basic* variables . IMPORTANT: if there are *m* equations we need to choose *m* elements as basis index corresponding to the *m* elements of the solution.

basis = [3];

Here we set the initial arrays.

x = zeros(size(c)); v = zeros(size(c));

[m,n] = size(A);

We update the basis at the start.

B=A(:,basis);

Here we solve for the initial basis feasible solution (bfs).

x(basis) = B\b;

Here we check if initial bfs is <0.

if any (x(basis) < 0)

error('Starting bfs has a component < 0');

end

Here we calculate the cost at the starting corner.

cost = c(basis)'\*x(basis);

fprintf('Basis elements are:\n');

fprintf('%d \n',B);

fprintf('Initial bfs is the vector:\n');

fprintf('%f\n',x);

fprintf('Initial bfs cost is:\n');

fprintf('%f\n',cost);

Next, we iterate to find the best corner: the total number of iterations possible does not need to exceed n\_choose\_m:

niter = nchoosek(n,m);

for iter = 1:niter

y = B'\c(basis); % y is the cost/gr

Here we calculate the reduced cost/hr and the index in of x\_in (the entering variable).

[rmin,in] = min(c - A'\*y);

fprintf('The smallest reduced cost element is: \n');

fprintf('%f \n',rmin);

fprintf('The entering index is:\n');

fprintf('%i \n',in);

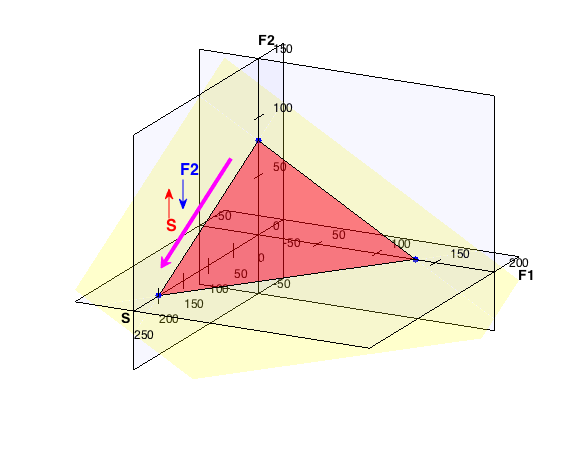
% optimality of x and y is reached if r >= 0

if rmin >= 0

fprintf('The smallest reduced cost is > 0: optimality reached! \n');

break;

end

****

Now that we know which corner enters the basis, we need to identify which corner leaves the basis. ***α*** is the coordinate along the edge from the old to the new corner. For example, by looking at the triangle we can see that the edge from FERM2(x3) to SHAKER(x1) has coordinates F1 = x2 = 0 and S = x1 and F2 = x3 such that:

x3\*500 + x1\*200 = 40000

x3 = 40000/500 - x1\*(200/500) = 80 - x1\*v

where ***v*** is the ratio between 200 () and 500 (). This ratio represents the fractional change in yield from 1 unit of x\_in is:

v(basis) = B\A(:,in);

When the corner is reached x3 = 0, therefore:

0 = 80 - x1\*v

x1 = 80/v

Thus, in this case ***α*** = 80/.4 = 200. *out* is index of the first *x* in (basis) to reach 0:

[alpha,out] = min( x(basis) ./ max(v(basis),0) );

out\_ind = basis(out);

fprintf('The leaving index is:\n');

fprintf('%i \n',out\_ind);

Here we lower the cost at end of each step.

cost = cost + alpha\*rmin;

fprintf('The cost at new corner is:\n');

fprintf('%f \n',cost);

Here we update the old x.

x(basis) = x(basis) - alpha\*v(basis);

x(in) = alpha;

Here we replace the old index with the new one in the basis

basis(out) = in;

Here we update the basis.

B=A(:,basis);

fprintf('Current basis index is:\n');

fprintf('%d \n',basis);

fprintf('Current bfs is:\n');

fprintf('%f \n',x);

end

We report the optimal basis, bfs, cost, and duality gap:

fprintf('The optimal basis is:\n');

fprintf('%d \n',B);

fprintf('The optimal bfs is:\n');

fprintf('%f \n',x);

fprintf('The total cost is:\n');

fprintf('%f \n',cost);

fprintf('x''c = %f \n',x'\*c);

fprintf('b''y = %f \n',b'\*y);

fprintf('The duality gap is %f \n',x'\*c-b'\*y);

fprintf('Number of iterations for convergence = %i \n',iter);

While the ***simplex*** method moves along the edges of the feasible set, the ***interior point*** method (**IPM**) moves inside the feasible set where every element of. A simple way to prevent from reaching negative values is to raise a barrier near the 0 boundaries. This is easily achieved by adding an extra cost in the form of a logarithm of that goes to infinity as any element of approaches 0. It's worth looking at the effect of the barrier with a specific example.

Consider the task of minimizing the objective function:

.

The *augmented* objective function is:

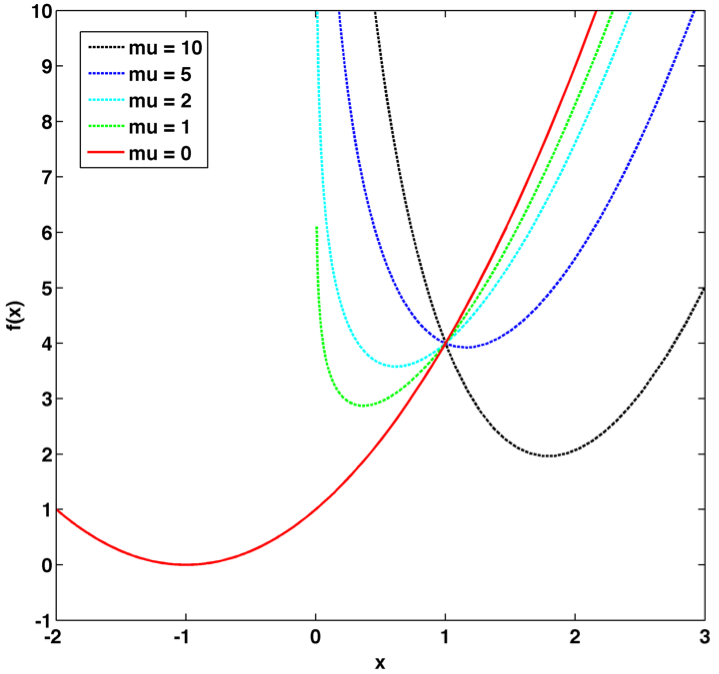
.

where is a small number that is progressively decreased toward 0. Notice how the constraint of nonnegativity is now incorporated inside . More generally, if instead of a nonnegativity constraint, we had both a *lower* and an *upper* bound constraint (often called a *box* constraint):

.

The *augmented* objective function can again be expressed with non-negativity constraints:

.

Using the simpler function with only nonnegativity constraint as an example, we can look at the effect of different values ranging from 0 to 10:

mu = 10;

fn = @(x) (x+1)^2 -mu\*(log(x));

fplot(fn,[0 3],'--k');hold on

mu = 5;

fn = @(x) (x+1)^2 -mu\*(log(x));

fplot(fn,[0 3],'--b');

mu = 2;

fn = @(x) (x+1)^2 -mu\*(log(x));

fplot(fn,[0 3],'--c');

mu = 1;

fn = @(x) (x+1)^2 -mu\*(log(x));

fplot(fn,[0 3],'--g');

mu = 0;

fn = @(x) (x+1)^2 -mu\*(log(x));

fplot(fn,[-2 3],'-r');

ylim([0 10])

The minimum of the unconstrained function is at -1, but the *log barrier* keeps the minimum of the *augmented function* in the positive range. Progressively smaller values produce an augmented function that, within the constraint of nonnegativity, is as similar as possible to the original function.

Typically, **IPM** uses the so-called *perturbed* KKT *conditions*, which take advantage of the matrices of both 1st (*Jacobian*) and 2nd (*Hessian*) derivatives of the objective and constraint functions. In describing this application we use the following definitions:

so that the **IPM** barrier problem looks like:

with *Lagrangian*:

upon defining:

where is used here to represent the *reduced cost vector*.We derive the following KKT optimality conditions from the partial derivatives of with respect to and :

to which we add a third condition:

where again is the average value of the product. Therefore, while is a column vector containing the actual products, is a column vector containing *n* identical elements equal to the average (or *centered*) value of the product. These conditions lead to the ***Newton-Raphson*** iteration step for the three unknown:

***J Δx = -f(x)***

where:

While this matrix (called a *perturbed* KKTmatrix) looks very different from the usual KKT, we can use the last equation to eliminate (and with it the last row and column of the matrix) and obtain a symmetric system. In fact:

Since the first term of the right-hand side of the *Newton-Raphson* system contains a component now we need to remove thatcomponent:

leading to:

the component can be removed from the RHS if we add a component of opposite sign, leading to the *reduced* *KKT* matrix:

Then, at each iteration after solving for we solve directly for . At the end of each iteration the steps are added to the values of from the previous iteration.

% f = 2x1+5x2+8x3 s.t. 200x1 +300x2+500x3 -40000 = 0

variables (x1 x2 x3)

n = 3;

objective function

f = @(x) 2\*x(1) + 5\*x(2) + 8\*x(3);

constraint

c = @(x) 200\*x(1) + 300\*x(2) + 500\*x(3) -40000;

gradient of objective function with respect to x1 x2 x3

fx = [2; 5; 8];

gradient of constraint with respect to x1 x2 x3 - Jacobian

cx = [200; 300; 500];

second gradient of objective function - Hessian

fxx = [0 0 0; 0 0 0; 0 0 0];

second gradient of constraint - Hessian

cxx = [0 0 0; 0 0 0; 0 0 0];

Initial guess values

lam = 1;

x = [40 40 40]';

Solver tuning

mu = 10;

Dual variables

z = mu ./ x';

X = diag(x);

Z = diag(z);

e = ones(n,1);

I = eye(n);

Hessian of the Lagrangian

W = fxx + lam \* cxx;

Iterations

for i = 1:200

Check if the inverse of X is numerically accurate

if rcond(X) < 2e-16

break

end

*Perturbed* KKT: solve A\*d = b for d (search direction)

% A = [W cx -eye(n);

% cx' 0 zeros(1,n);

% Z zeros(n,1) X];

% b = -[fx + cx\*lam - z';c(x);X\*Z\*e-mu\*e];

Equivalent symmetric *reduced* KKT matrix.

invX = X\I;

A = [W+invX\*Z cx ;

cx' 0 ];

b = -[fx + cx\*lam - mu\*invX\*e;

c(x)];

QR factorization

[Q,R] = qr(A,0);

d = R\Q'\*b;

Run the following line together with the *reduced* KKT matrix.

d(5:7) = mu\*invX\*e - Z\*e - invX\*Z\*d(1:3);

update values

x(1) = x(1) + d(1);

x(2) = x(2) + d(2);

x(3) = x(3) + d(3);

lam = lam + d(4);

z(1) = z(1) + d(5);

z(2) = z(2) + d(6);

z(3) = z(2) + d(7);

print summary

disp(f(x)); disp(x); disp(lam)

update x, X, Z

X = diag(x);

Z = diag(z);

lower mu

mu = mu / 10;

end

Notice that since in the brewery example the Hessian matrix is 0, given the objective function in the form (where ***c*** is now the cost vector), and the constraint in the form , we have:

and the *reduced* KKTmatrix:

becomes:

***J Δx*** *=* ***-f(x)***

which can be recognized as the *Newton-Raphson* step that solves the *Lagrangian*:

As tends to 0, the minimum is reached, which means that the *optimality condition* also tends to0**.** This result shows that thevector of ***dual variables*** is the ***negative*** of the vector of *Lagrange multipliers* in the ***interior point*** method.

In practice, we will not write our own programs to solve problems of linear programming, but use some of the tested programs available inside the MATLAB Optimization Toolbox or other commercial Linear Programming toolboxes for the MATLAB environment (e.g., Gurobi, OPTI, CVX, which are free for educational institutions). For example, the following is how we would solve the brewery problem with the MATLAB function *linprog*:

Here we set up the objective function and the constraints:

A = [200 300 500];

b = 40000;

c = [2 5 8]';

lb = zeros(3,1); % Lower bound

ub = [1000 1000 1000]; % Upper bound

x0 = [40 40 40]; % Initial solution guess

Next, we call a linear programming routine:

options = optimoptions(@linprog,'Algorithm','dual-simplex');

[x,fval,exitflag,output,lambda] = linprog(c,[],[],A,b,lb,ub,x0,options);

or using IPM:

options = optimoptions(@linprog,'Algorithm','interior-point');

[x,fval,exitflag,output,lambda] = linprog(c,[],[],A,b,lb,ub,x0,options);

We also derive the *shadow prices* and the *reduced costs*. When the optimal solution is found (*optimality condition* ) the *shadow price* (*dual variable vector*) of the constraint is the derivative of the objective function with respect to the *right hand side* (RHS) ***b*** of this constraint, or in other words, the *sensitivity* of the solution to a change in the constraint. We have seen how this derivative is the *negative* of the Lagrange multiplier itself:

opt\_x = x

objective\_function = fval

lambda = lambda.eqlin

shadow\_prices = -lambda\_eqlin

It's easy to check this by running a loop over small changes in ***b***:

b\_delta = 1e-4

b\_vec = [-b\_delta b\_delta];

nb = length(b);

npoints = length(b\_vec);

fval\_vec = zeros(npoints,1);

for j = 1:nb

for i = 1:npoints

b\_mod = b;

b\_mod(j) = b(j) + b\_vec(i);

[x,fval\_vec(i),exitflag,output,lambda] = ...

linprog(c,[],[],A,b\_mod,lb,ub,x0,options);

end

shadow\_prices\_direct(j) = (fval\_vec(2)-fval\_vec(1))/(2\*b\_delta);

end

shadow\_prices\_direct

The *reduced cost* vector ***s*** is the derivative of the objective function *F*with respect to each of the design variables at the solution. In our case:

Since is the negative of the actual slope, we take the negative of to calculate the reduced costs:

s = c-A'\*-lambda.eqlin

Therefore, any negative element in the *reduced cost* vector would indicate that an *increase* (if allowed by the constraints) in the corresponding design variables would tend to decrease the objective function ***cTx***.

Notice that the *reduced costs* are the Lagrange multipliers on the lower (nonnegativity) and upper bound constraints at the solution. Due to *linprog* sign conventions we have the following:

lower bounds = *reduced costs* due to non-negativity constraints

upper bounds = -*reduced costs* due to upper bounds

For example, we can introduce an upper bound of 100 hours on SHAKER, FERM1, FERM2:

ub = [100 100 100]; % Upper bound

[x,fval,exitflag,output,lambda] = linprog(c,[],[],A,b,lb,ub,[],options);

opt\_x = x

objective\_function = fval

lambda\_eq = lambda.eqlin

lambda\_upper = lambda.upper

lambda\_lower = lambda.lower

shadow\_price = -lambda.eqlin

reduced\_costs = c-A'\*-lambda.eqlin

Finally, we can also introduce lower bounds on the minimum use of all three types of equipment, and a time constraint indicating we want the job to take less than 120 hours of combined time (that is summing up the time used by all three types of growth):

A\_eq = [200 300 500];

b\_eq = 40000;

A\_ineq = [1 1 1];

b\_ineq = 120;

But we would used the following to limit the job to a maximum of 120 hours regardless of which single piece of equipment or combination of equipments is used:

c = [2 5 8]';

lb = ones(3,1)\*10; % Lower bound

ub = [120 120 120]; % Upper bound

options = optimoptions(@linprog,'Algorithm','interior-point');

[x,fval,exitflag,output,lambda] = linprog(c,A\_ineq,b\_ineq,A\_eq,b\_eq,lb,ub,[],options);

x; fval

lambda\_eq = lambda.eqlin

lambda\_ineq = lambda.ineqlin

lambda\_upper = lambda.upper

lambda\_lower = lambda.lower

shadow\_prices = -[lambda.eqlin;lambda.ineqlin]

reduced\_costs = c -[A\_eq;A\_ineq]'\* -[lambda.eqlin;lambda.ineqlin]

Notice that nonzero elements in the vectors always indicate active constraints at the solution (i.e., the solution is on their constraint boundaries).

We can check the values of the shadow prices and of the reduced costs derived from , by calculating them directly looping over the r.h.s. of all the constraints:

First, we calculate the *shadow prices* on the equality and inequality constraints :

On the equalities constraints

b\_delta = 1e-4;

b\_vec = [-b\_delta b\_delta];

nb = length(b\_eq);

npoints = length(b\_vec);

shadow\_prices\_eq = zeros(nb,1);

fval\_vec = zeros(npoints,1);

for j = 1:nb

for i = 1:npoints

b\_mod = b\_eq;

b\_mod(j) = b\_eq(j) + b\_vec(i);

[x,fval\_vec(i),exitflag,output,lambda] = ...

linprog(c,A\_ineq,b\_ineq,A\_eq,b\_mod,lb,ub,[],options);

end

shadow\_prices\_eq(j) = (fval\_vec(2)-fval\_vec(1))/(2\*b\_delta);

end

On the inequalities constraints

b\_delta = 1e-4;

b\_vec = [-b\_delta b\_delta];

nb = length(b\_ineq);

npoints = length(b\_vec);

shadow\_prices\_ineq = zeros(nb,1);

fval\_vec = zeros(npoints,1);

for j = 1:nb

for i = 1:npoints

b\_mod = b\_ineq;

b\_mod(j) = b\_ineq(j) + b\_vec(i);

[x,fval\_vec(i),exitflag,output,lambda] = ...

linprog(c,A\_ineq,b\_mod,A\_eq,b\_eq,lb,ub,[],options);

end

shadow\_prices\_ineq(j) = (fval\_vec(2)-fval\_vec(1))/(2\*b\_delta);

end

shadow\_prices\_direct = [shadow\_prices\_eq;shadow\_prices\_ineq]

Next, we calculate the first component of the *reduced costs* as the *shadow prices* on the lower bounds: we change only one element at a time in the lower bounds.

lb\_delta = 1e-4;

lb\_vec = [-lb\_delta lb\_delta];

nlbs = length(lb);

npoints = length(lb\_vec);

reduced\_costs\_direct\_low = zeros(nlbs,1);

fval\_vec = zeros(npoints,1);

for j = 1:nlbs

for i = 1:npoints

lb\_mod = lb;

lb\_mod(j) = lb(j) + lb\_vec(i);

[x,fval\_vec(i),exitflag,output,lambda] = ...

linprog(c,A\_ineq,b\_ineq,A\_eq,b\_eq,lb\_mod,ub,[],options);

end

reduced\_costs\_direct\_low(j) = (fval\_vec(2)-fval\_vec(1))/(2\*lb\_delta);

end

reduced\_costs\_direct\_low

Next, we calculate the 2nd component of the *reduced costs* as the *shadow prices* on the upper bounds: to this end we change only one element at a time in the upper bounds.

ub\_delta = 1e-5;

ub\_vec = [-ub\_delta ub\_delta];

nubs = length(ub);

npoints = length(ub\_vec);

reduced\_costs\_direct\_up = zeros(nubs,1);

fval\_vec = zeros(npoints,1);

for j = 1:nubs

for i = 1:npoints

ub\_mod = ub;

ub\_mod(j) = ub(j) + ub\_vec(i);

[x,fval\_vec(i),exitflag,output,lambda] = ...

linprog(c,A\_ineq,b\_ineq,A\_eq,b\_eq,lb,ub\_mod,[],options);

end

reduced\_costs\_direct\_up(j) = (fval\_vec(2)-fval\_vec(1))/(2\*ub\_delta);

end

reduced\_costs\_direct\_up

Finally, we sum up the *shadow prices* on both the lower and upper bounds: this sum represents the total *reduced costs*.

reduced\_costs\_direct = reduced\_costs\_direct\_low + reduced\_costs\_direct\_up

**SPECIAL TOPICS: Interior Point Method and Quadratic programming (QP).**

We have seen how the minimizazion of a function gives origin to two types of problems depending on whether the function is:

The general quadratic problem can be stated as:

with both *m* x *n*, and *Lagrangian* :

where is a vector of Lagrange multipliers for the *equality constraints*, and is a vector of Lagrange multipliers for the *inequality constraints*, such that:

where is the *ith* row in and is the *ith* element in . Taking the partial derivative with respect to the three variables, this Lagrangian gives origin to the following KKT optimality conditions:

to which we had:

Introducing the slack vector we can simplify the notation to , which can be shown to be the expression for the *duality gap* in the QP problem with inequalities (with similar to the corresponding of the LP problem). The new optimality conditions become:

For simplicity, we will consider here further the algebraic application of this result for the QP case containing only inequalities (no matrix, vector, vector), which is the most common case. Upon defining:

We introduce the *residual* function:

which we solve in a ***Newton-Raphson*** step for the three unknown:

***J Δx = -f(x)***

Given an initial iterate (at each iteration *k* we solve the system:

to determine the *search direction* . This direction is also called the *affine scaling direction*.  It regularly happens that only a small step can be taken along the affine search direction before the constraints are violated and the step leads into a non-feasible region. A less aggressive approach is to take only a fraction of the full step in the search direction. One additional problem of using Newton methods is that the remaining nonnegativity constraints cannot be taken easily into account. There are various flavors of IPM for *quadratic programming*.(e.g. Path following algorithm, Affine scaling algorithm, Mehrotra Predictor-corrector algorithm, …) that address this task in slightly different ways. In practice, *Mehrotra Predictor-corrector algorithm* is used most of the time.

As indicated by the name there is both a *predictor* and a *corrector* step involved in this algorithm. In order to explain the purpose of each of these steps it is necessary to explain a few concepts first, including that of the *central path*. The central path is a *new function* defined as:

,

This is a *curve* of *strictly feasible points*, , parameterized by a scalar , and for which and are both **0**.  The idea is to have the iterates () progress along this central path and to have decrease with each step. As approaches zero, approximates the optimality conditions . This means that the central path follows a path to the solution such that . At the same time as the optimality condition is approached the pairwise products  vanish to zero. The purpose of using this *central path* concept is that we expect to obtain the fastest rate of convergence by having the iterates follow (approximately) the central path. We also see that the points on the central path satisfy a slightly *perturbed* version of the optimality conditions, with the only difference, compared to the optimality conditions, being the term on the RHS.

In addition to the central path we also define the *complementarity measure*:

and a *centering parameter* .  The complementarity measure tells us something about the average value of the pairwise products . It tells us something about how useful a computed search direction is by determining if a computed step has or has not reduced  significantly. If is reduced significantly little centering is needed and a value of that reflects this is chosen ( in this case). If on the other hand  is not reduced then the computed search direction is not that useful and a larger value of is chosen.

In practice, we start by solving the system obtaining the *affine scaling direction* and then determine a step length  for this step. While we could use a traditional *line search* for this purpose, a close form solution for this value is available (see function *QP\_IPM*, below). The computation of the affine scaling direction and step is also referred to as the *predictor step*.

Next the complementarity measure for the current iterate is computed along with a predicted complementarity measure   for the computed Newton step:

By comparing the values of and we determine if the computed affine scaling direction is a good search direction. If for example  we have a significant reduction in the complementarity measure and the search direction is good, so little centering is needed. In practice the centering parameter  is computed using a formula that experience has proven to work well:

where   is the duality gap at the current point, and is the duality gap that would be achieved if we were to follow the largest feasible step along the affine scaling direction.  On the other hand, if we set up a *centering step*, that is a Newton step toward the point where :

It is important to recognize the structural similarity between the term of this centering step and the term of the standard *Newton-Raphson* step of the *LP* solution, seen previously:

Both terms represent the *KKT* implementation of the *logarithmic barrier* the preserves the solution non-negativity.

We notice at this point that the steps and are also affected by a *linearization error* as we are using only the 1st derivative of . To correct for this error Mehrotra has recommended adding a term to the residual . This leads to the final formula for the *corrector step*, which includes both a 2nd order correction and a centering:

In practice, we will not solve the full system, but instead split the system into three separate equations such that we solve one of those equations and use back substitution to solve the remaining two equations (see function *QP\_IPM*, below). Further algebraic manipulation leads to a modified step equation for the *predictor step* that is numerically more stable:

followed by a derivation for :

and likewise, for the *corrector step* with the difference that we use a modified term:

Furthermore, at each iteration, instead of taking the full step derived from these calculations a scaling (dampening) factor is applied to calculate the actual final step as:

The value of  is essential for the algorithm converging or not converging. If the value is too large we will have non-convergence, a value that is too small will lead to slow convergence so one has to determine a value such that the tradeoff is acceptable. In most cases seems to be a good choice. Larger values lead to non-convergence.

Finally, to stop the sequence of iteration a number of criteria are used jointly. These are:

1. the number of iterations, with a maximum usually kept at 200.

2. a lower bound tolerance on the norm of the residual:

3. a lower bound on the absolute value of the complementarity measure to ensure that the algorithm stops if the complementarity products are reduced to zero or a value close to zero.

,

Let’s see how this method works with a simple quadratic function:

We can use the following simple rules to write this equation in matrix notation:

If the vector is x = (X, Y, Z) T, and you want the 3 x 3 matrix A so your polynomial is xTAx, then:

(1) Put the coefficients of X², Y², Z² (in that order.)

(2) Take the coefficients of each mixed product and divide by 2.

(3) Put each half of these divided coefficients on the off diagonal elements---e.g. if the coefficient of XY is T, put ½T in the {1,2} position and ½T in the {2,1} position.

This will give a symmetric matrix with the desired property.

Example: Y² + 3Z² - 4XZ + 8YZ

The coefficients of X², Y², and Z² are 0, 1, 3 on the main diagonal.

The coefficient of XY = 0, so there are zeroes in the {1,2} and {2, 1} positions.

The coefficient of XZ is -4. So, -2 goes in the {1, 3} and {3, 1} places.

The coefficient of YZ is 8. So, 4 goes in the {2,3} and {3,2} places.

Based on these simple rules the equation above can be written as:

We can plot the function and the constraints in a useful range:

C1 = @(x) 2-x;

C2 = @(x) 1+1/2\*x;

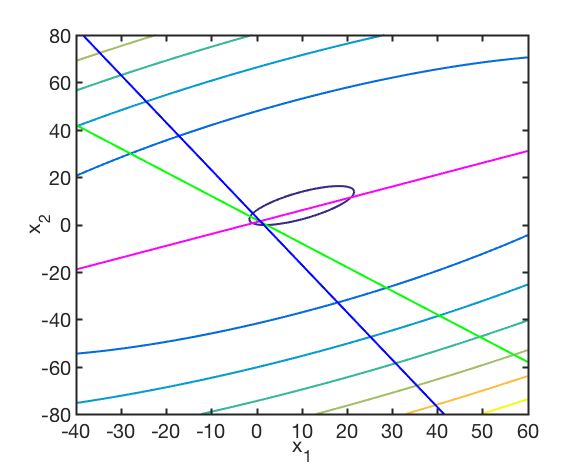
C3 = @(x) 3-2\*x;

G = [1 -1;-1 2];

g = [-2 -6]';

x1 = [-40:1:60];

x2 = [-80:1:80];



nx1 = length(x1);

nx2 = length(x2);

F = zeros(nx2,nx1);

for i = 1:nx1

    for j = 1:nx2

        x = [x1(i) x2(j)]';

        F(j,i) = 1/2\*x'\*G\*x + g'\*x;

    end

end

contour(x1,x2,F)

hold on

fplot(C1,[-40 60],'g')

fplot(C2,[-40 60],'m')

fplot(C3,[-40 60],'b')

ylim([-80 80])

We can see clearly how the function minimum is not in the feasible region contained below the blue and magenta constraints lines.

We can solve this constrained minimization problem using MATLAB *quadprog* function:

G = [1 -1;-1 2];

g = [-2 -6]';

C = [1 1;-1 2;2 1];

d = [2;2;3];

lb = zeros(2,1);

options = optimoptions('quadprog','Algorithm','interior-point-convex','Display','off');

[x,fval,exitflag,output,lambda] = quadprog(H,c,A,b,[],[],lb,[],[],options);

plot(x(1),x(2),'or','LineWidth',6)

or a simple implementation of the Mehrotra *predictor-corrector* algorithm for QP-IPM:

function [ x\_stop,z\_stop,s\_stop,k ] = QP\_IPM( x,z,s,G,g,C,d )

dmp = 0.95;

% Residuals

[~,nC] = size(C);

e = ones(nC,1);

rL = G\*x + g -C\*z;

rC = -C'\*x + s;

rsz = s .\* z;

mu = sum(z .\* s) /nC;

k = 0 ;

maxk = 200;

eps\_L = 1e-10; eps\_C = 1e-10; eps\_mu = 1e-10;

while (k <= maxk && norm(rL) >= eps\_L && norm(rC)>=eps\_C ...

        && abs(mu) >= eps\_mu)

lhs = [G -C;-C' sparse(-diag(s./z))];

rhs = [-rL;-rC + rsz./z];

[L,D,P] = ldl(lhs);

dxyz\_a = P\*(L'\(D\(L\(P'\*rhs))));

dz\_a = dxyz\_a(length(x)+1:length(x)+length(z));

ds\_a = -((rsz + s.\*dz\_a) ./ z);

% Compute alpha\_aff

alpha\_a = 1;

idx\_z = find(dz\_a < 0);

if (isempty(idx\_z) == 0)

    alpha\_a = min(alpha\_a, min(-z(idx\_z)./dz\_a(idx\_z)));

end

idx\_s = find(ds\_a < 0);

if (isempty(idx\_s) == 0)

    alpha\_a = min(alpha\_a, min(-s(idx\_s)./ds\_a(idx\_s)));

end

% Compute the affine duality gap

mu\_a = ((z + alpha\_a\*dz\_a)'\*(s + alpha\_a\*ds\_a))/nC;

% Compute the centering parameter

sigma = (mu\_a/mu)^3;

% Solve the system

rsz = rsz + ds\_a.\*dz\_a - sigma\*mu\*e;

rhs = [-rL;-rC + rsz./z];

[L,D,P] = ldl(lhs);

dxyz = P\*(L'\(D\(L\(P'\*rhs))));

dx = dxyz(1:length(x));

dz = dxyz(length(x)+1:length(x)+length(z));

ds = -((rsz + s.\*dz) ./ z);

% Compute alpha

alpha = 1;

idx\_z = find(dz < 0);

if (isempty(idx\_z) == 0)

    alpha = min(alpha, min(-z(idx\_z)./dz(idx\_z)));

end

idx\_s = find(ds < 0);

if (isempty(idx\_s) == 0)

    alpha = min(alpha, min(-s(idx\_s)./ds(idx\_s)));

end

% Update x,z,s

x = x + dmp\*alpha\*dx;

z = z + dmp\*alpha\*dz;

s = s + dmp\*alpha\*ds;

k = k + 1;

% Update the rhs

rL = G\*x + g - C\*z;

rC = -C'\*x + s + d;

rsz = s.\*z;

mu = sum(z.\*s)/nC;

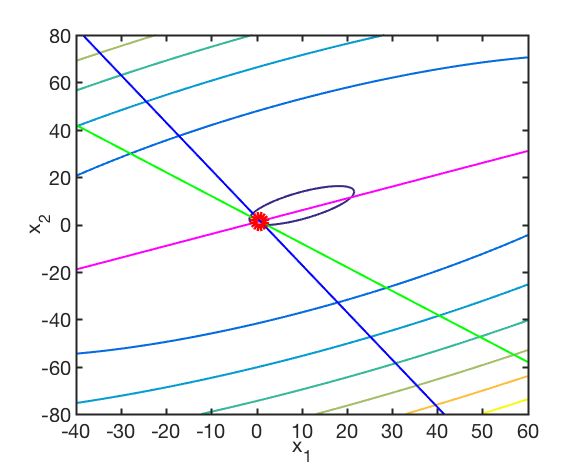
end

% Output

x\_stop = x; z\_stop = z; s\_stop = s;

end

We set up the system as inequalities as requested by the function:



G = [1 -1;-1 2];

g = [-2 -6]';

Ct = [-1 -1;1 -2;-2 -1];

C = Ct';

d = [-2;-2;-3];

x = [1 1]';

z = ones(3,1);

s = ones(3,1);

[ x\_stop,z\_stop,s\_stop,k ] = … QP\_IPM( x,z,s,G,g,C,d )

As expected, both MATLAB *quadprog* and the local *QP\_IPM* function identify the same minimum in the feasible region at x = [0.6667 1.3333].