Technical University of Denmark

02612 Constrained Optimization 2017

Assignment 1

Authors:
Miguel Suau de Castro (s161333)
Michal Baumgartner (s161636)
Ramiro Mata (s161601)

March 28, 2017

Contents

1 Quadratic Optimization

1.1 Quadratic programming with equality constraints

We are given the following constrained optimization problem:

$$\begin{aligned} \min_x f(x) = 3x_1^2 + 2x_1x_2 + x_1x_3 + 2.5x_2^2 + 2x_2x_3 + 2x_3^2 - 8x_1 - 3x_2 - 3x_3 \\ s.t. \quad x_1 + x_3 = 3 \\ x_2 + x_3 = 0 \end{aligned} \tag{1}$$

This is a quadratic problem and thus it can be written in the form:

$$\min_x f(x) = \frac{1}{2} x^T H x + g^T x \tag{2}$$

Where in this case:

$$g = \begin{bmatrix} -8 \\ -3 \\ -3 \end{bmatrix} \quad H = \begin{bmatrix} 6 & 2 & 1 \\ 5 & 2 & 2 \\ 4 & 2 & 1 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 3 \\ 0 \end{bmatrix} \tag{3}$$

The Lagrangian function associated to the problem is:

$$\mathcal{L}(x, \lambda) = \frac{1}{2} x^T H x + g^T x - \lambda'(A'x - b) \tag{4}$$

Any feasible solution of 1 will satisfy the following first order KKT conditions:

$$\begin{aligned} \nabla_x \mathcal{L}(x, \lambda) = Hx + g - \lambda A = 0 \\ c(x) = Ax - b = 0 \end{aligned} \tag{5}$$

One could then find a local minimum by setting to zero the gradient of the lagrangian function. Moreover, since $\nabla^2_{xx}\mathcal{L} = H$ and H is positive definite, we know that the problem is convex and that the solution is unique and corresponds to a global minimum.

We have then implemented the function EqualityQPSolver in Matlab which can solve any equality constrained convex quadratic program. The function takes as input H, g, A and b and returns both the solution and the Lagrangian multipliers by solving the linear system below:

$$\begin{bmatrix} H & -A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -g \\ b \end{bmatrix} \tag{6}$$

It is easy to see that the matrix in the left hand side of equation 6, called KKT matrix, is symmetric. Considering that the efficiency of the algorithm will mainly depend on how fast this matrix is inverted, we should make use of the symmetry properties of the KKT matrix to solve the system. Consequently, applying Cholesky decomposition, the KKT matrix can be expressed as the product of a lower triangular matrix and its transpose and the system can then be easily solved in two steps by forward and backward substitution.

According to the documentation, the backslash operator in Matlab will analyze the matrix and come up with the most efficient approach. However, we may save some computations if we factorize first and then solve using the operator.

We have tested the code by solving one hundred random convex quadratic programs. The problems are generated by giving random values to the matrices H and A, and the solution x. We then compute the vectors g and b, call the function, and compare the solution with the initialized value. In order to ensure that the problem is convex, the code initializes the H matrix a few times in every iteration until it is positive definite.

The code for the random quadratic program generator and the solver are shown in the appendix.

1.2 Sensitivity analysis

In most practical situations apart from giving an accurate solution it may also be interesting to know how much this solution could be affected by a small change either in the objective function or in the constraints. With this in mind, we shall perform a sensitivity analysis to investigate the influence of the parameters g and b on our results.

The values in these two vectors will be now substituted by a new set of variables, named p , so that by taking the derivatives with respect to those variables we can study how they relate to the solution.

$$g = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} \qquad b = \begin{bmatrix} p_4 \\ p_5 \end{bmatrix} \tag{7}$$

According to the lecture slides, the parameter sensitivities can be calculated applying the equation below:

$$\begin{bmatrix} \nabla x(p) & \nabla \lambda(p) \end{bmatrix} = - \begin{bmatrix} W_{xp} & -\nabla_p c(x^*, p) \end{bmatrix} \begin{bmatrix} W_{xx} & -\nabla_x c(x^*, p) \\ -\nabla_x c(x^*, p)^T & 0 \end{bmatrix}^{-1} \tag{8}$$

Where:

$$\begin{aligned} W_{xp} &= \nabla^2_{xp} f(x, p) - \sum_{i \in \varepsilon} \lambda_i \nabla^2_{xp} c_i(x, p) \\ W_{xx} &= \nabla^2_{xx} f(x, p) - \sum_{i \in \varepsilon} \lambda_i \nabla^2_{xx} c_i(x, p) \end{aligned} \tag{9}$$

And in our case:

$$W_{xp} = \begin{bmatrix} I \\ 0 \end{bmatrix} \quad \nabla_p c(x^*, p) = \begin{bmatrix} 0 \\ -I \end{bmatrix} \quad W_{xx} = H \quad \nabla_x c(x^*, p) = A \tag{10}$$

Therefore, the sensitivities can be computed as:

$$\begin{bmatrix} \nabla x(p) & \nabla \lambda(p) \end{bmatrix} = - \begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix}^{-1} \tag{11}$$

We have written a function in Matlab, that takes as input W_{xp}, $\nabla_p c(x^*, p)$, W_{xx} and $\nabla_x c(x^*, p)$ and returns $\nabla x(p)$ and $\nabla \lambda(p)$.

Another way to approximate the parameter sensitivities, would be to apply the finite difference method. The idea is to introduce small perturbations in our system and compare the solution obtained with the true values. We will run a loop that will add a small quantity, say $\varepsilon = 10^{-4}$, to every value in the vectors g and b and call the quadratic solver to get the new results. Thus, the values of $\nabla x(p)$ and $\nabla\lambda(p)$ can be approximated as follows:

$$\begin{aligned} \nabla x_i(p) &\approx \frac{(x_t^* - x^*)}{\varepsilon} \\ \nabla \lambda_i(p) &\approx \frac{(\lambda_t^* - \lambda^*)}{\varepsilon} \end{aligned} \tag{12}$$

Where x_t^* and λ_t^* are the solutions obtained after adding ε to an specific element in the vectors g or b. We can then check whether or not, the values obtained are similar to those returned by the function.

The code for the function and the testing script can be found in the appendix.

1.3 Dual program

A convex quadratic program like the one given by equation 1 can also be expressed in the Wolfe dual form, and its solution will be equivalent to the original or primal problem, as:

$$\begin{aligned} \max_{x,\lambda} \quad \mathcal{L}(x,\lambda) &= \frac{1}{2}x^T Hx + gx - \lambda^T(Ax - b) \\ s.t. \quad \nabla_x \mathcal{L}(x,\lambda) &= Hx + g - A'\lambda = 0 \end{aligned} \tag{13}$$

The objective function can be then simplified by using the constraint:

$$\begin{aligned} \max_{x,\lambda} \quad -\frac{1}{2}x^T Hx + (Hx + g - \lambda A)^T x + \lambda^T b &= -\frac{1}{2}x^T Hx + \lambda^T b \\ s.t. \quad Hx + g - A'\lambda &= 0 \end{aligned} \tag{14}$$

We see now that two variables x and λ are being optimized, instead of just x as in the primal problem. Moreover, solving for x in the equation of the constraint

$$x = H^{-1}(A^T\lambda - g) \tag{15}$$

we can reformulate the expression as an unconstrained optimization problem:

$$\max_{\lambda} \quad q(\lambda) = -\frac{1}{2}(A^T\lambda - g)^T H^{-1}(A^T\lambda - g) + \lambda^T b \tag{16}$$

and since this is a concave function, that is, $\nabla^2_{\lambda\lambda} q(\lambda) = -A^T H^{-1} A$ is negative definite, the solution will be unique and the global maximum can be found by setting the gradient to 0. This single condition is both necessary and sufficient, in contrast to the primal problem, where also the constraints needed to be satisfied.

$$\nabla_{\lambda} q(\lambda) = -A^T H^{-1} A\lambda - A^T H^{-1} g + b = 0 \tag{17}$$

One can easily obtain the Lagrange multipliers by solving the linear system. Finally, the solution to both the dual and the primal program can be found plugging in the value of λ in equation 15.

The function EqualityQPdual, which can be found in the appendix, finds the minimum of the convex constrained quadratic problem by solving the dual program using the equations above.

All in all, we see that after transforming the dual program into an unconstrained optimization problem, the matrices needed to be inverted, while still symmetric, they are smaller than the KKT matrix which means that for large problems the algorithm becomes computationally more efficient.

2 Equality constrained quadratic solvers

2.1 Test problem

In this section, we are going test the efficiency of a set of algorithms, in solving equality constraint quadratic programs for different problem sizes. To do so, we will solve the following n+1-dimension convex problem:

$$
\begin{aligned}
\min_u f(x) &= \frac{1}{2}\sum_{i=1}^{n+1}(u_i - \bar{u})^2 \\
s.t. \quad u_i + u_n &= -d_0 \\
-u_i + u_{i+1} &= 0 \\
u_{n-1} - u_n - u_{n+1} &= 0
\end{aligned}
\tag{18}
$$

Where $n \geq 3$, $\bar{u} = 0.2$ and $d_0 = 1$.

Te problem can be formulated in matrix form as:

$$
\begin{aligned}
\min_x f(x) &= \frac{1}{2}x^T Hx + g^T x \\
s.t. \quad A^T x &= b
\end{aligned}
\tag{19}
$$

Which for $n = 10$

$$
g = -0.2e \qquad H = I \tag{20}
$$

$$
A = \begin{bmatrix}
-1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}
\qquad
b = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\tag{21}
$$

The Lagrangian function can be then written as:

$$
\mathcal{L}(x, \lambda) = \frac{1}{2}x^T Hx + g^T x - \lambda^T (Ax - b) \tag{22}
$$

We know that the objective function is convex and that $\nabla^2_{xx}\mathcal{L} = H$. Therefore, for any value of n there is just one solution to this problem and the first order KKT conditions are both necessary and sufficient.

$$\begin{aligned} \nabla_x \mathcal{L}(x,\lambda) = Hx + g - \lambda A = 0 \\ c(x) = Ax - b = 0 \end{aligned} \tag{23}$$

2.2 Dense solvers

We mentioned in section 1 that the KKT matrix for convex quadratic programs is square and symmetric, we shall then use these property to decompose the matrix so that we can solve the linear system more efficiently.

In our first approach we are going to implement an algorithm that performs LU decomposition. The KKT matrix is expressed as the product of a lower and an upper triangular matrix. This is very convenient because then the system can be solved in two steps by forward and backward substitution.

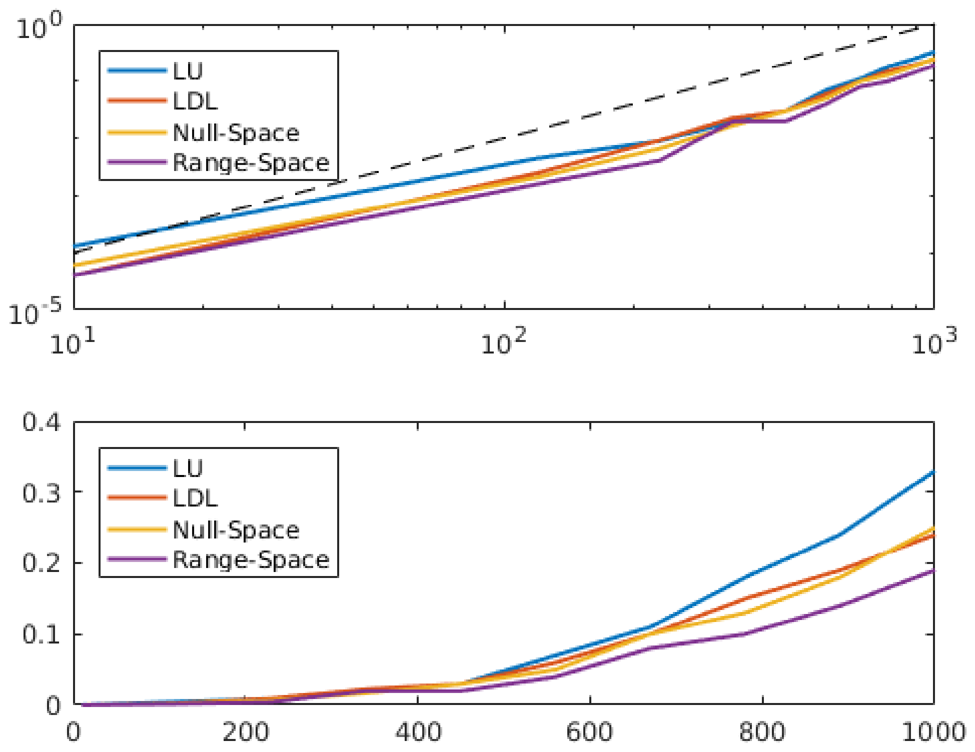


Figure 1: CPU-time for all dense solvers and different problem sizes, the top graph is a double logarithmic plot

A better method could be obtained by applying Cholesky factorization to decompose the matrix. This uses the symmetry properties of the KKT matrix to write it as the product of a lower triangular matrix and its transpose. Then we solve the system again by forward and backward substitution. This clearly holds better memory usage, since we are continuously accessing the same space in memory.

These two methods have been implemented in Matlab, along with two more advanced algorithms for solving linear systems the Null-Space and the Range-Space method. All the code can be found in the appendix. Once the matrices have been decomposed the solution is computed using the backslash operator so as to allow Matlab to use the most appropriate solver.

We have then measured the CPU-time for all four implementations and different problem sizes. To be more precise, for the matrices with the lower dimensions we allowed the code to be executed several times and then took the mean of the different run times.

The results are shown in figure 1, where the top graph is a double logarithmic plot and the black dashed line corresponds to $O(h^2)$. All algorithms seem to scale with the square of the matrix size. Besides, it is easy to see, especially in the bottom graph, that for large problems the Range-Space method is by far the best of the four methods. On the other hand, we see that the LDL is much faster than the LU solver for large matrices, which matches with our previous explanation.

2.3 Sparse solvers

Another nice property of the KKT-matrix is that there are not many non-zero elements. This is very advantageous if one thinks about storage. A matrix with a lot of zero elements is usually known as sparse matrix in computer science, and even though it is stored differently, it allows some programs to run much faster than with dense matrices. Figure 2 shows the sparsity pattern of the KKT matrix.

The Matlab function sparse can convert a dense matrix into the sparse form. The difference in memory usage between storing the KKT matrix as a dense or a sparse matrix of double precision floating point numbers is shown in the table below.

Name	Size	Bytes	Class	Attributes
K	2001×2001	32032008	double	
Ks	2001×2001	96032	double	sparse

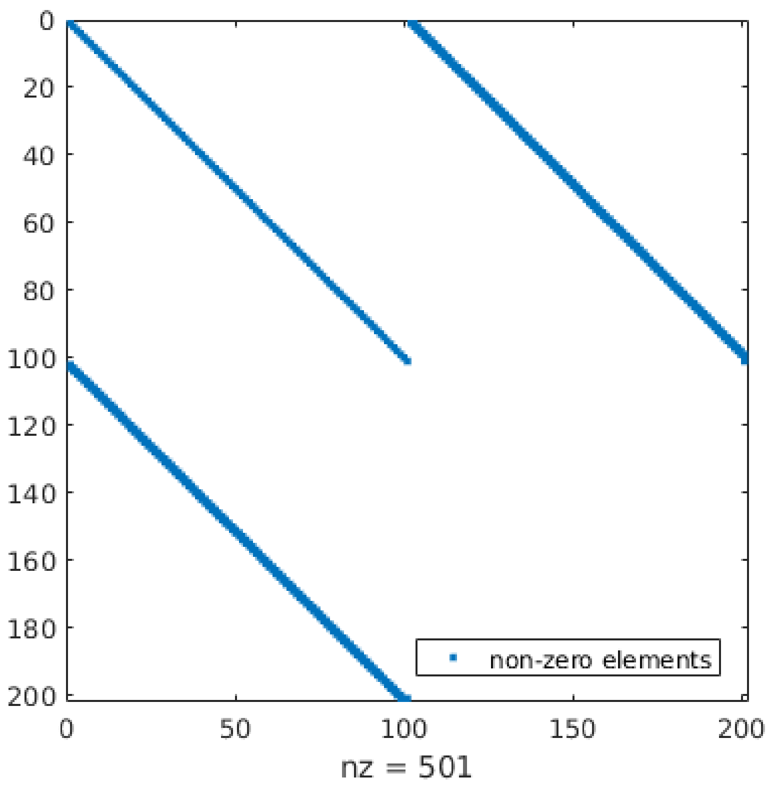
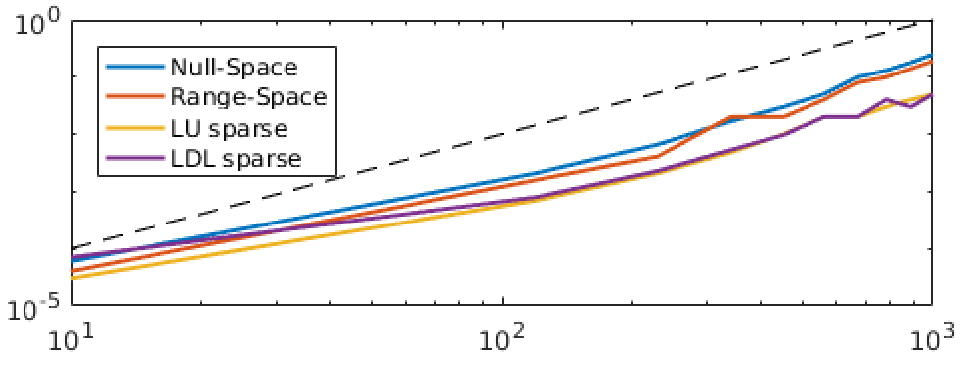


Figure 2: Sparsity pattern of the KKT matrix

Both the lu and the ldl functions return the matrices L and U in sparse form when the input is sparse. Besides, according to Matlab documentation, the backslash operator can also handle sparse matrices. Therefore why modified the code in these to solver so before decomposing the KKT-matrix we convert the matrix to sparse form.

The measures of CPU-time are shown in figure 3 where we now plot in logarithmic scale the sparse solvers against the Null-Space and the Range-Space methods for dense matrices. The bottom graph shows the performance of all six implementations, it is obvious that handling the information in sparse form gives a clear advantage. However, the slope of the curves in the top graph reveal that the complexity scale remains quadratic where again the black dashed line corresponds to $O(h^2)$



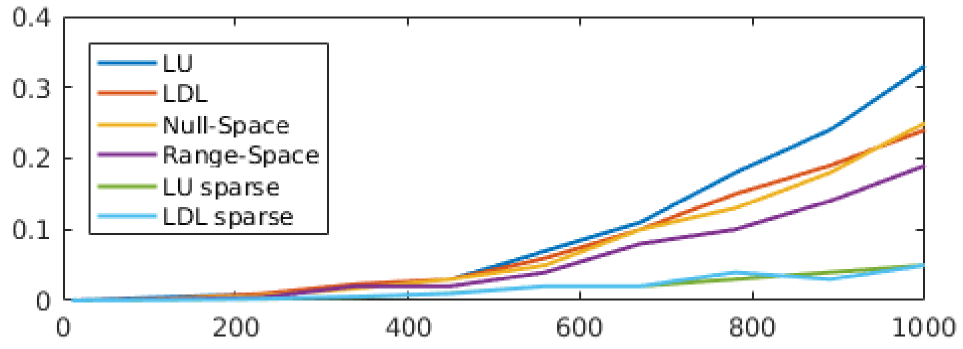


Figure 3: Top graph: CPU-time in a double logarithmic plot for sparse solvers, Null-Space and Range-Space methods. Bottom graph CPU-time for all dense and sparse solvers.

3 Inequality Constrained Quadratic Programming

3.1 Test problem

In order to study different approaches for solving inequality constraint convex quadratic programs, we are given the problem below:

$$
\begin{aligned}
\min_x q(x) = (x_1-1)^2 + (x_2-2.5)^2 \\
s.t. \quad x_1 - 2x_2 + 2 \geq 0 \\
-x_1 - 2x_2 + 6 \geq 0 \\
-x_1 + 2x_2 + 2 \geq 0 \\
x_1 \geq 0 \\
x_2 \geq 0
\end{aligned}
\tag{24}
$$

This is a quadratic problem and thus it can be written in the form:

$$\min_x f(x) = \frac{1}{2}x^T H x + g^T x \tag{25}$$

Where in this case:

$$g = \begin{bmatrix} -2 \\ -5 \end{bmatrix} \quad H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad A = \begin{bmatrix} 1 & -2 \\ -1 & -2 \\ -1 & 2 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \quad b = \begin{bmatrix} -2 \\ -6 \\ -2 \\ 0 \\ 0 \end{bmatrix} \quad \lambda = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \end{bmatrix} \tag{26}$$

The Lagrangian function associated to the problem is:

$$\mathcal{L}(x,\lambda) = \frac{1}{2}x^T H x + g^T x - \lambda'(A'x - b) \tag{27}$$

Any feasible solution of 24 will satisfy the following first order KKT conditions:

$$
\begin{aligned}
\nabla_x \mathcal{L}(x,\lambda) = Hx + g - \lambda A = 0 \\
c(x) = Ax - b \geq 0 \\
\lambda \geq 0 \\
\lambda c(x)_i = 0
\end{aligned}
\tag{28}
$$

The local minimum (x^*) can be found by letting the gradient of the lagrangian function equal to zero and solving. In regards to the second order KKT conditions, the problem satisfies both the necessary and sufficient optimality conditions. This is because $\nabla^2_{xx}\mathcal{L} = H$ and H is positive definite, so this explicitly satisfies the

sufficient second order optimality conditions. From this, we have confirmed that the problem is **strictly** convex and thus has a unique solution (global minimum) given the inequality constraints.

3.2 Conceptual active set algorithm

We shall make use of the function implemented in the first section, named `EqualityQPSolver`, for solving the test problem. As we mentioned before the function can solve equality constrained convex quadratic programs. Since we would like to apply it to find the minimum for different working sets, we modified the function so that it can also solve problems with no constraints. Even though the functions implemented in section 2 may run faster for large matrices, considering that the size and the sparsity of our problem will depend on the number of active constrains, we thought that using the backslash operator was more convenient in this case.

Figure 16.3 in Nocedal and Wright shows $(2,0)$ as the initial point and since it lies on the intersection of constraint 3 ($-x_1 + 2x_2 + 2 \geq 0$) and 5 ($x_2 \geq 0$), therefore the working set W_0 is chosen as $\{3,5\}$.

Following the algorithm, Lagrange multipliers are computed according to 16.42 in N&W as $\lambda_3 = -2$ and $\lambda_5 = -1$. All the multipliers are negative and the third constraint has the lowest λ so it can be removed from the working set.

Afterwards the step-length formula (16.41) is used to compute $\alpha_1 = 1$ and the new iterate $x^2 = (2,0) + 1 \cdot (-1,0) = (1,0)$. There are no blocking constraints as $\alpha_1 = 1$ so W remains unchanged. Once again the Lagrange multiplier has to be computed as $p^2 = \mathbf{0}$ resulting in $\lambda_5 = -5$, which is negative and therefore it is removed resulting in W being an empty set (unconstrained).

Stepping further, α_3 is 0.6, which is less than 1 so the step was blocked by some constraint that is not included in W yet. Note that x_4 is updated and is now $(1,0) + 0.6 \cdot (0,2.5) = (1,1.5)$.

It can be seen that the first constraint ($x_1 - 2x_2 + 2 \geq 0$) is added and then the step-length is found to be $\alpha_4 = 1$ so there are no blocking constraints and the next iterate of $x_5 = (1,1.5) + 1 \cdot (0.4,0.2) = (1.4,1.7)$. Since $p^5 = \mathbf{0}$, the only Lagrange multiplier λ_1 is computed and is equal to 0.8, which is non-negative so all KKT conditions are satisfied (p. 469 in N&W) and the algorithm can be terminated with result of $x^* = (1.4,1.7)$.

k	W_k	x^k	λ_k
0	$\{3,5\}$	$[2 \quad 0]^T$	$(\lambda_3, \lambda_5) = (-2, -1)$
1	$\{5\}$	$[2 \quad 0]^T$	
2	$\{5\}$	$[1 \quad 0]^T$	$\lambda_5 = -5$
3	$\{\}$	$[1 \quad 0]^T$	
4	$\{1\}$	$[1 \quad 1.5]^T$	
5	$\{1\}$	$[1.4 \quad 1.7]^T$	$\lambda_1 = 0.8$

Table 1: Active set - iterations

Table 1 shows concise representation of each iteration. The contour plot with the iteration sequence is shown in 4.

The section about "Comment on the Lagrange multipliers at each iteration" was also merged with this one as the elements (constraints) and updating strategy of working set W_k depends on the values of Lagrange multipliers in each step. If all multipliers are non-negative the algorithm can be terminated as the fourth KKT-condition (16.37d in N&W) and all previous KKT conditions are met (as $p = 0$) therefore the solution is found.

3.3 Active set method

In contrast to the simplex algorithm, were the iteration sequence can just move around the feasible region boundaries, we are going to describe and implement an algorithm where every step in the path, is based on the solution of a secondary quadratic program. Moreover, the decision on the working set is not just based on the value of the Lagrangian multipliers, but also, on the proximity of the current iterate to the constraints.

We now refer our description to Algorithm 16.3 in N.& W. The method needs a starting point within the feasible region before the first iteration. The choice of the initial working set is then based on the active constraints at that point. We then solve the quadratic program given by equation 16.39 using our equality constraint solver.

This function will give us the direction of the next step in the iteration sequence. In case the vector returned has norm 0 the algorithm has found a point that minimizes the function for the current working set constraints, and therefore we should check the Lagrange multipliers to see whether or not this point is the

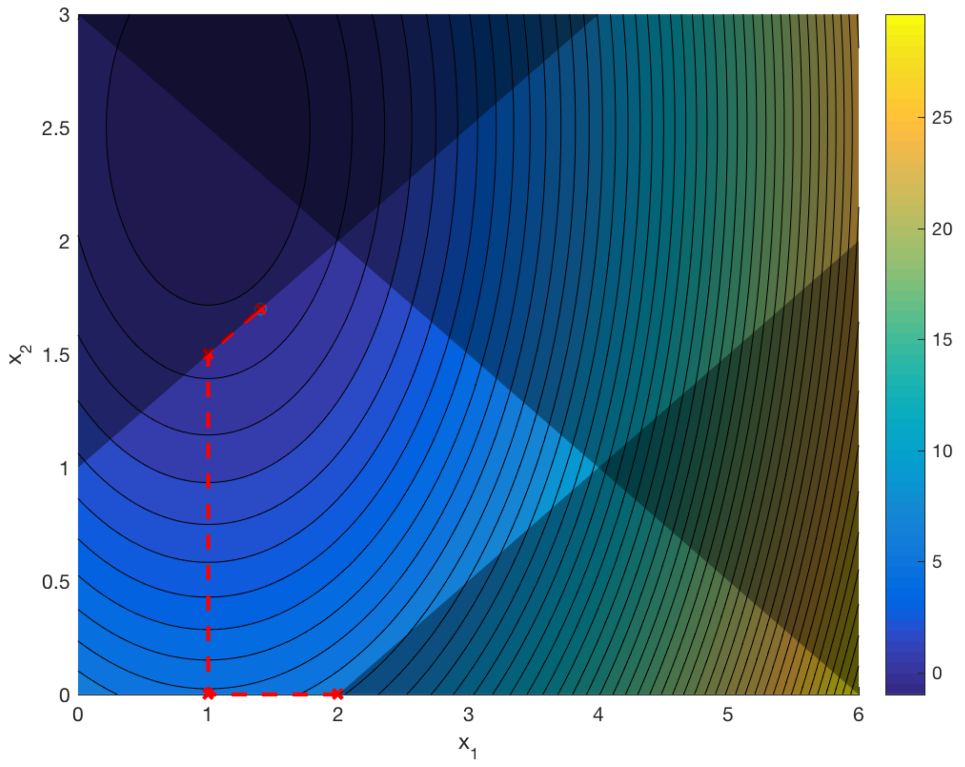


Figure 4: Contour plot of $q(x)$ with iteration sequence of the conceptual active set algorithm depicted in a red dashed line

global optimum. If these are all positive, then we have found a solution and the algorithm stops. If on the other hand, one or more of these coefficients are negative we shall remove the constraint with the smaller coefficient from the working set and start again.

When the norm of the direction vector is not 0, the algorithm tries to find the largest step length within $[0, 1]$ that satisfies all constraints. If the iteration sequence finds a constraint blocking the search direction, the step length will be set to be smaller than 1, so that the next iterate lies within the feasible region, and the specific constraint blocking the path will be added to the working set.

The algorithm implemented in Matlab can be found in the appendix. This takes H, g, A, b and an initial feasible point as input, and returns the solution, the Lagrange multipliers and some information regarding the number of iterations, the iteration sequence and the working set in every iteration.

The iteration sequence is shown in figure 5 from different starting points.

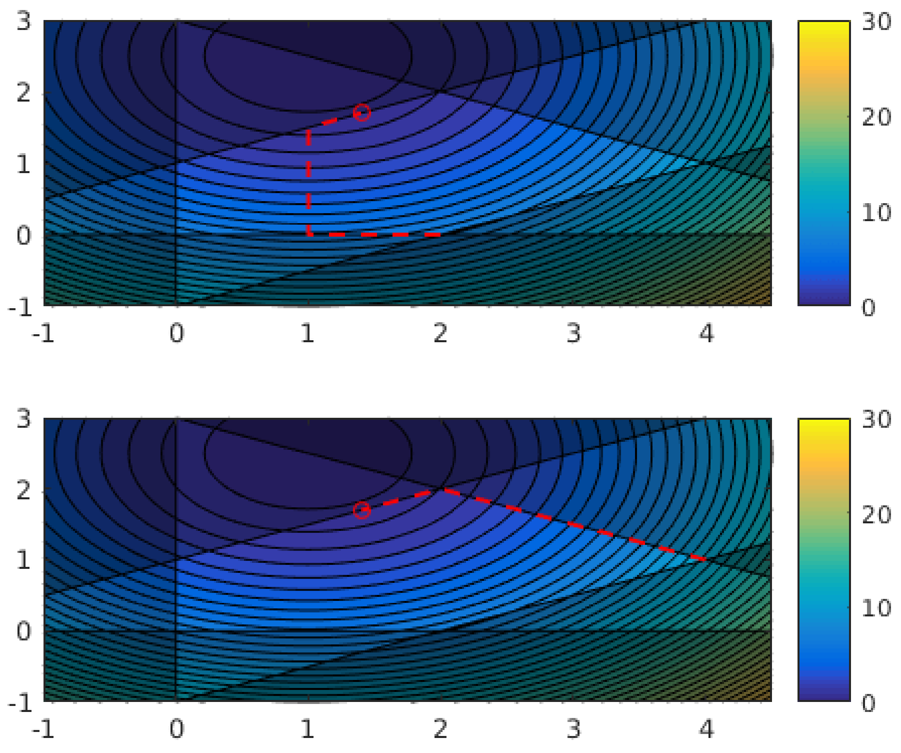


Figure 5: Contour plot of $q(x)$ with iteration sequence of the active set algorithm depicted in a red dashed line from two different starting points [2,0] and [4,1]

3.4 Feasible point

As we mentioned before, the algorithm needs a initial point within the feasible region to start running. In our case, it would be easy to determine it just by looking at the contour plot. However, this is not always possible, and thus we are going to use a method that allows to compute this initial point.

The method given in page 473 in N.& W. is described as a variation of the ”Phase I” method for linear programming, and it requires an initial guess that does not necessary need to be within the feasible region.

One could then build the vectors needed and use the Matlab function linprog to solve the linear program.

We have implemented and run the code shown in the appendix. Setting the initial value of x to $[-1,-1]$ (out of the feasible region), linprog returns $x_0 = [1.9876, 0.9471]$ (within the feasible region).

Finally figure 6 shows the iteration sequence returned by the active set algorithm when running with the initial guess given by linprog.

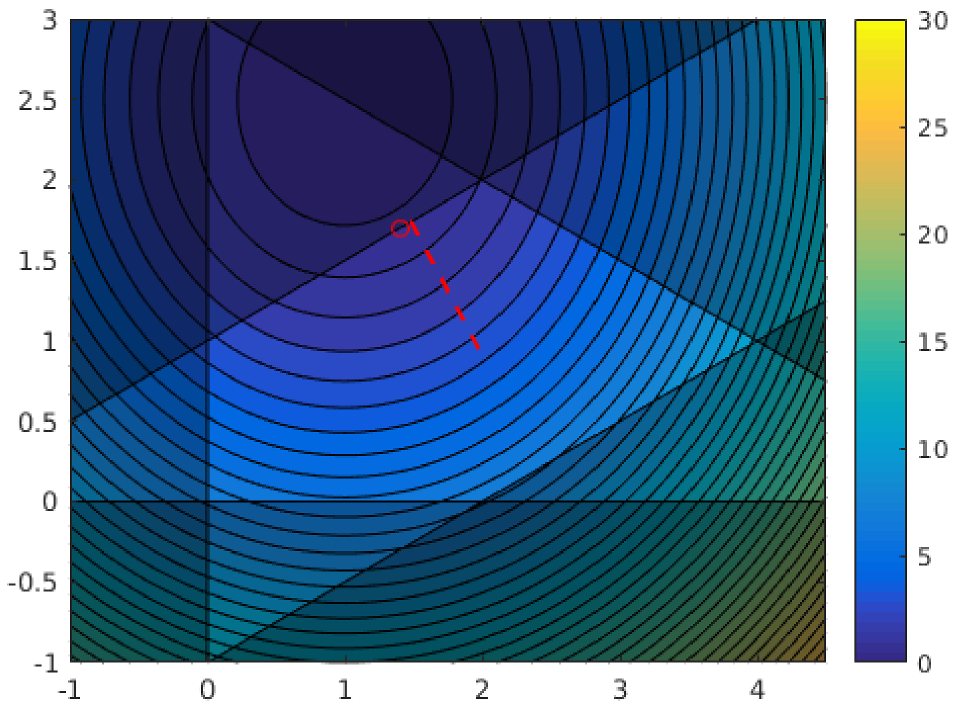


Figure 6: Contour plot of $q(x)$ with iteration sequence of the active set algorithm depicted in a red dashed line using the feasible point returned by linprog

We have also collected all the information returned by the function in the table below.

Iteration	Sequence	Working Set
1	[1.9786,0.9571]	
2	[1.4792,1.7396]	1
3	[1.4000,1.7000]	1

3.5 Penalty method

Section 16.47 and 16.48 in N&W book outlines ”big M” method of using a L-inf and the L-1 norm penalty. Where the η and **M** are hyper-parameters introduced in the L-inf implementation, and **v** and $\mathbf{e}^T$ are introduced in the L-1 implementation. These 2 ”Big M” methods applied on inequality constrained optimization problems act as a substitute of ”Phase I”, which is used to initialize a feasible starting point for **x**. Using these ”big M” methods, on the other hand, allow us to start an initial guess at any point (it does not have to be feasible). In the case of the L-inf implementation for instance, during the active set iterative process, the algorithm will converge to the global minimum if η becomes zero. If η is a non-zero positive number, this means we have to increase **M** until η converges to zero, which will mean that the algorithm has converged to the unique solution.

We outline our implementation of the L-inf penalty for the inequality constrained optimization problem as follows:

$$\begin{aligned} \min_{x,\eta} q(x) = \frac{1}{2}x^T H x + g^T x + M\eta \\ s.t. \quad b_i - a_i^T x \leq \eta \\ 0 \leq \eta \end{aligned} \tag{29}$$

And for the L-1 penalty as follows:

$$\begin{aligned} \min_{x,v} q(x) = \frac{1}{2}x^T H x + g^T x + M e^T v \\ s.t. \quad a_i^T x - b + v_i \geq 0 \\ v \geq 0 \end{aligned} \tag{30}$$

We implement the L-1 method with our ActiveSetMethod.m algorithm with the following formulation:

$$g = \begin{bmatrix} -2 \\ -5 \\ M \\ M \\ M \\ M \\ M \end{bmatrix} \quad H = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{31}$$

$$A = \begin{bmatrix} 1 & -2 & 1 & 1 & 1 & 1 & 1 \\ -1 & -2 & 1 & 1 & 1 & 1 & 1 \\ -1 & 2 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad b = \begin{bmatrix} -2 \\ -6 \\ -2 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} \tag{32}$$

We implement the L-inf method with our ActiveSetMethod.m algorithm with the following formulation. Notice that the last element in H is made arbitrarily small for stability (otherwise we would end up with a singular matrix):

$$g = \begin{bmatrix} -2 \\ -5 \\ M \end{bmatrix} \quad x = \begin{bmatrix} -1 \\ -1 \\ \eta \end{bmatrix} \quad H = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1e-3 \end{bmatrix} \tag{33}$$

$$A = \begin{bmatrix} 1 & -2 & 1 \\ -1 & -2 & 1 \\ -1 & 2 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad b = \begin{bmatrix} -2 \\ -6 \\ -2 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{34}$$

The last element in the H matrix should be zero. But it has been set to a small number in order to prevent division by 0. Matlab's function quadprog can handle that but our function cant.

With this method, we minimize the problem w.r.t. **x** and v. v is a vector containing a scalar variable for each individual inequality constraint. It is a result of multiplying with the e^T vector, which is a vector of 1's the size of however many inequality constraints we have - in this case 5. Notice that now our matrix implementations above now accommodates for these new v's. The H matrix contains the old H and a diagonal of 1's each accounting for the v's and e^T vector. Similarly, the A matrix comprises of the old A matrix, and columns of ones to accommodate for the v's and e^T vector. We now change our solution vector, $\mathbf{x}$, to include the v's so that they are minimized as well.

With this implementation, our ActiveSetMethod.m function allows us to pick any initial points (feasible or infeasible) and still converge to the global minimum.

Similarly, we implement the L-inf "Big M" method but in this case we use an artificial scalar (η). By controlling the η and **M** hyper-parameters we can manage to converge even if we start from the infeasible region (see Figure 7). This so long as η is chosen big enough to satisfy the inequality constraints. Our algorithm, formulation and implementation is found in the corresponding Matlab code attached, and as can be seen in Figure 7, it achieves convergence to the global minimum. We double-checked with our implementation using quadprog() and obtained convergence.

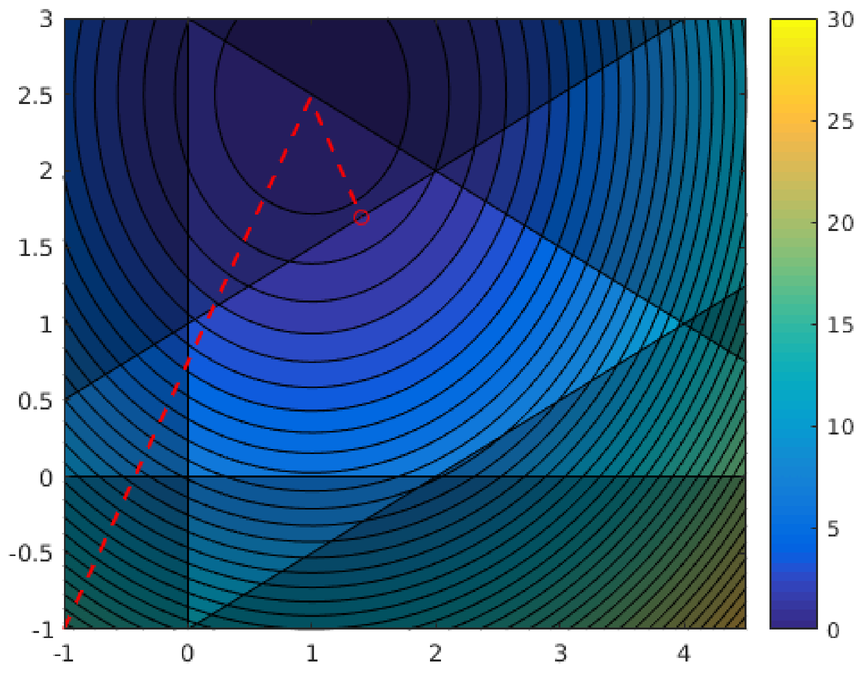


Figure 7: "Big M" Method (L-inf Penalty): As can be seen above, this method allows us to pick any initial **x** estimate regardless whether it's a feasible or infeasible point. We started with $\mathbf{x} = [-1, -1]$, which is clearly outside the inequality constraints and therefore in the infeasible region (shaded), yet it still converges the the global minimum.

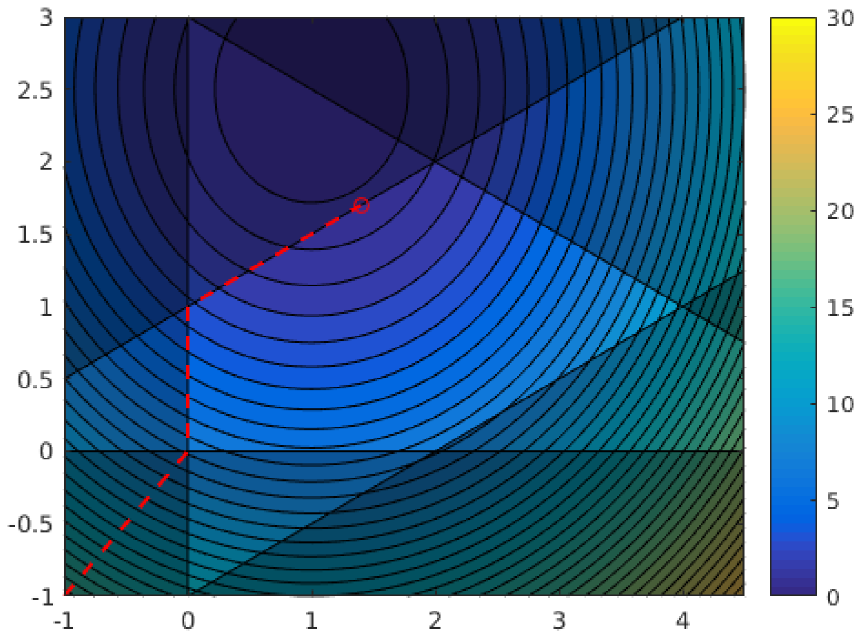


Figure 8: ”Big M” Method (L-1 Penalty): As can be seen above, this method allows us to pick any initial **x** estimate regardless whether it’s a feasible or infeasible point. We started with $\mathbf{x} = [-1, -1]$, same as the previous plot, yet the L-1 method shows a different path. Nonetheless, they both converge to the global minimum.

4 Markowitz Portfolio Optimization

4.1 Markowitz' Portfolio optimization as a quadratic program.

We formulate Markowitz Portfolio Optimization problem as a QP as follows:

$$\min_{x} \frac{1}{2}\mathbf{x}^T \Sigma \mathbf{x} \tag{35}$$

subject to:

$$\mu^T \mathbf{x} \geq R \tag{36}$$

$$A_{eq}\mathbf{x} = b_{eq} \tag{37}$$

$$C\mathbf{x} \geq d \tag{38}$$

Basically, our aim is to minimize the variance subject to the portfolio returns ($\mu_t x$) being equal or greater than our target return (R). The other two constraints specify that the weights equal to 1 (i.e. , we invest all of our capital), and that each weight cannot be negative (i.e., no short-selling is allowed).

4.1.1 Notation

x is a vector of weights x_1, x_2, x_3, ... , x_n, each corresponding to one of the n number of assets in our portfolio. In this case, we have 5 assets and thus $n = 5$. In essence, the vector **x** is our portfolio weights given the assets we're given. Σ corresponds to the variance-covariance matrix of the returns of our assets. Similarly, μ is a vector of of the expected returns of our assets (i.e. $\mu = \mu_1, \mu_2, ..., \mu_5$). Finally, R is the target return value of our choice. Now that we have defined our notation, $\frac{1}{2}\mathbf{x}^T\Sigma\mathbf{x}$ is the portfolio variance, and $\mu^T\mathbf{x}$ is the portfolio expected return.

Therefore, the first constraint simply says that our portfolio expected return should be equal to or greater than our target return (R). In Matlab, we make it an equality constraint to get the exact portfolio for a specific return (e.g. 10% return). The second constraint can be used to specify whether we want our stocks to be invested in certain combinations or proportions. For instance, we could design the A_{eq} matrix such that we invest in assets 1 and 2 always in the same proportion. However, for the first part of this assignment we will use it to specify that the sum of the weights should equal to 1 (i.e., we invest 100% of our capital) since for this section a risk free rate is not available. Finally, our third constraint can be used to specify whether we are allowed to short stocks or not. In this assignment, we do not want to short stocks, then C would be the identity matrix and d would be a vector of zeros.

4.2 Minimal and maximal possible return

This depends on whether we have access to a risk-free asset (i.e., allowed to borrow or lend money at a fixed rate) and whether we are allowed to short assets.

Assuming risk free asset, but no shorting, then the minimum return possible would correspond to losing all of our initial money plus all the money we borrowed plus the interest from the loan. The maximum return we can expect is directly proportional to (1) the market run, and (2) how much we are leveraged by the risk free asset. By leverage, in financial terms this means how much money we borrowed at the risk free rate to expand the volume of our portfolio.

In the case of no shorting and also no risk-free asset, the minimum return we can achieve is -100%, i.e. losing all the money we started with. Similarly to the first case, the maximum return we can achieve is directly proportional to the market run given our portfolio's CAPM beta coefficient. The latter coefficient simply measures a portfolio's exposure to the market. That is, for a beta coefficient of 0.5, if the market rises by 6% then our portfolio rises 3%. Similarly, if it drops 6%, then our portfolio only loses 3%. So the maximum we could achieve in any given market would be 50% if our portfolio's CAPM beta coefficient was 0.5, for instance.

4.3 Optimal portfolio for $R = 10$ and its risk

With the above formulation, we can use quadprog to find optimal portfolios. For an $R = 10$ we obtain the following:

$$OptimalPortfolio = \begin{bmatrix} 0 & 0.2816 & 0 & 0.7184 & 0 \end{bmatrix} \tag{39}$$

$$Variance = 2.0923 \tag{40}$$

4.4 Efficient frontier and optimal portfolio (without risk free asset)

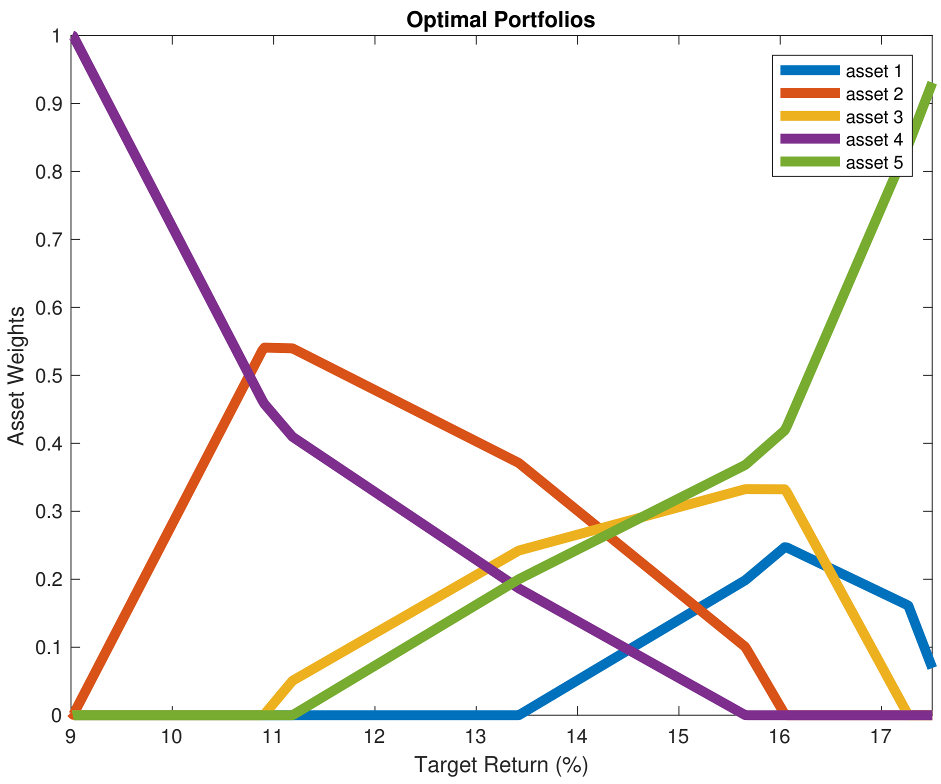


Figure 9: Optimal Portfolios (No Risk Free Asset): the asset weights of the portfolio as a function of return. Notice that for a target return of 10%, the plot tells us to allocate asset 2 and 4 to the portfolio and the rest to zero, coinciding with the weights found previously by quadprog.

4.5 Add $r_f = 2$: new covariance matrix and return vector

Now that we have access to a risk free security, we simply add this at the end of our return vector. Similarly, we add its variance information to the variance-covariance matrix. Note that since it's a risk free asset, by definition its risk is 0. Thus, we modify the variance-covariance matrix by adding an extra row and an extra column of zeros. We obtain the following:

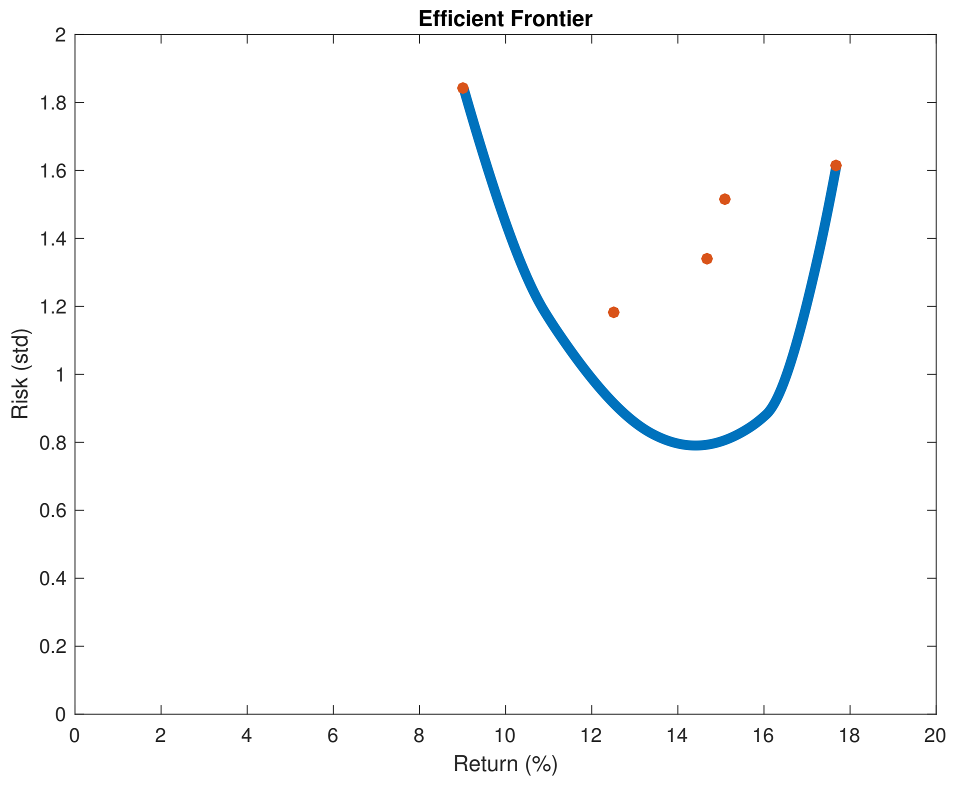


Figure 10: Efficient Frontier (No Risk Free Asset): Each circle corresponds to investing 100% of our portfolio in one of those 5 assets. One of the main findings in Markowitz mean-variance portfolio theory tells us of the benefits of diversification. All optimal portfolios (denoted by the efficient frontier) optimize on the return-risk space and are therefore more efficient than those inside the curve (the inefficient portfolios). Markowitz theory tells us that optimization in the risk-variance space can be achieved by portfolio diversification.

$$\Sigma = \begin{bmatrix} 2.3 & 0.93 & 0.62 & 0.74 & -0.23 & 0 \\ 0.93 & 1.4 & 0.22 & 0.56 & 0.26 & 0 \\ 0.62 & 0.22 & 1.8 & 0.78 & -0.27 & 0 \\ 0.74 & 0.56 & 0.78 & 3.4 & -0.56 & 0 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{41}$$

$$Returns = \begin{bmatrix} 15.1 & 12.5 & 14.7 & 9.02 & 17.68 & 2 \end{bmatrix} \tag{42}$$

4.6 Efficient frontier(with risk free asset)

4.7 Comment

The effect of the risk free asset is that now our efficient frontier is a line starting at the risk free asset to the highest-slope portfolio (i.e., the highest Sharpe ratio) of the previous efficient frontier without risk free asset. The Sharpe ratio is the slope on figure 12. Notice that now all of our efficient portfolios i the efficient frontier have the same Sharpe ratio. This allows us to use the risk free asset as a dial to match clients with their specific risk or return personal preferences for a fixed Sharpe ratio. In addition, if we were allowed to borrow with that free risk asset, we could leverage the portfolio and unlock return-risk coordinates further up in the upper-right positive quadrant.

4.8 Minimal risk and optimal portfolio for R = 15.00.

$$OptimalPortfolio = \begin{bmatrix} 0.1655 & 0.1365 & 0.3115 & 0.0266 & 0.3352 & 0.0247 \end{bmatrix} \tag{43}$$

$$Variance = 0.6383 \tag{44}$$

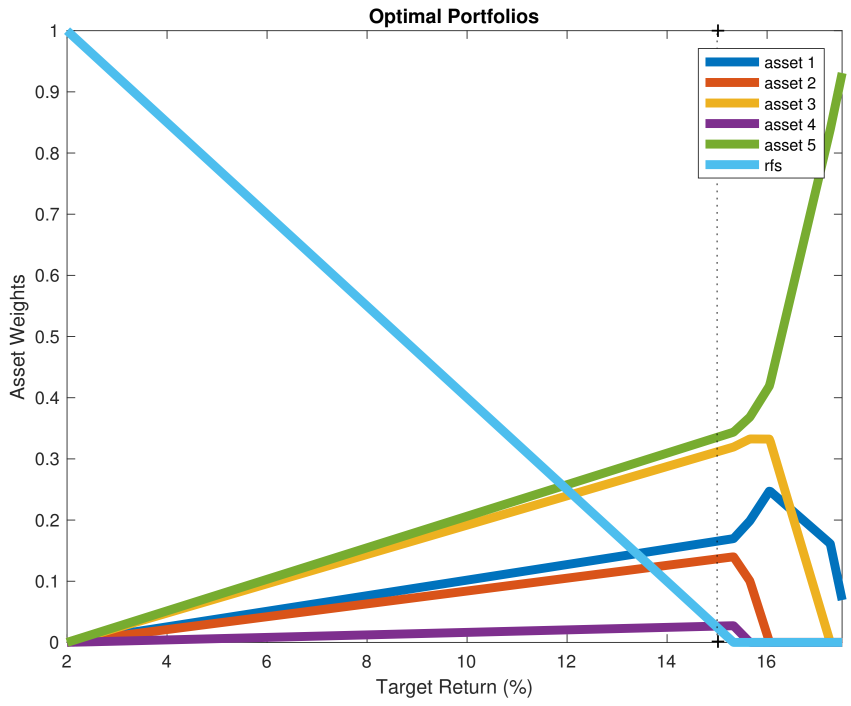


Figure 11: Asset Weights: The blue line corresponds to the risk free asset. Notice that the plot tells us to invest 100% in the risk free asset for a target return of 2%. This is in line with our model. For greater return preferences, we would shift from investing solely in the risk free asset to investing more in the other assets. For a target return of 15% (denoted by the dashed-line), notice that we would invest almost nothing in our risk free asset, and invest in the others somewhat more evenly. For returns greater than 16%, however, we would invest mostly in asset 5.

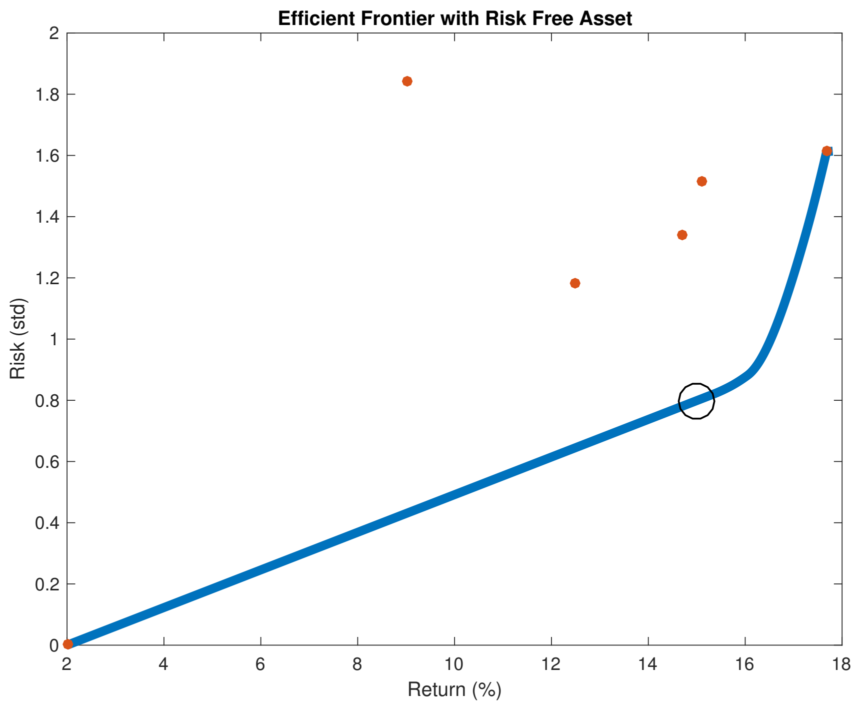


Figure 12: Efficient Frontier (with Risk Free Asset): The red circles denote the individual securities. Notice the risk free asset near the origin. The efficient frontier (blue line) corresponds to the line from the risk free asset to the black-cross (which is the highest-slope portfolio of the previous efficient frontier without risk free rate).

5 Interior-Point Algorithm for Convex Quadratic Programming

For quadratic program that is convex and defined as

$$\min_{x} \quad \frac{1}{2}x^T H x + g^T x \tag{45a}$$
$$\text{s.t.} \quad A^T x = b, \qquad i \in \mathcal{E}, \tag{45b}$$
$$C^T x \geq d, \qquad i \in \mathcal{I} \tag{45c}$$

one may rewrite the KKT-conditions by introducing a so called *slack* variables $s \triangleq C^T x - d \geq 0$ resulting in:

$$\begin{aligned}
&r_L = \nabla_x L(x, \lambda) = Hx + g - Ay - Cz = 0 && && \\
&r_A = A^T x - b = 0 && && \\
&C^T x - d \geq 0 && \rightarrow && r_C = C^T x - s - d = 0 \\
&(C^T x - d)_i z_i = 0 && \rightarrow && s_i y_i = 0 \\
&z \geq 0 && \rightarrow && (z, s) \geq 0
\end{aligned} \tag{46}$$

The convex quadratic program can then be solved by solving the system defined in equation 46 if H is assumed to be positive semidefinite (so the KKT-conditions are not only necessary but also sufficient - see N&W theorem 16.4). In this section S and Z will denote the diagonal matrix obtained from s and z respectively, e will represent a column vector of ones and the complementarity conditions $s_i z_i = 0$, $i = 1, \ldots, m_c$ (where m_c is the number of inequality constraints) can be rewritten as $r_{SZ} = SZe = 0$.

One could then solve the system $[r_L, r_A, r_C, r_{SZ}]^T = \mathbf{0}$ with a Newton-like method to obtain a step. However it can be improved by modifying the Newton procedure so that the search direction is skewed towards the interior of the space subset given by $(z, s) \geq 0$ (i.e. quadrant in 2d, octant in 3d, orthant for higher dimensions) because it allows to move further along the search direction before one of the components of (z, s) becomes negative. Another advantage is that it prevents these components of getting too close to the boundary of the aforementioned orthant. This modification is known as the *primal-dual interior point method*.

In practice the most used interior point method is based on Mehrotra's extension and will be referred to as the *predictor-corrector algorithm*. The psedocode given in slides from week 6 will be used a reference for the Matlab implementation.

The algorithm itself consists of several parts parts which will be described and later on tied together as a pseudocode explanation.

Finding the initial point

A heuristic for finding the initial point can be achieved either by a user supplied initial guess or by a more sophisticated way that consists of multiple parts described in subsections below. The main gist is computing the residuals and the affine search direction by solving a system using LDL factorization which will be talked about later. In the end the initial point is updated with the addition of max$\{1, |\{z, s\} + \Delta\{z, s\}^{\text{aff}}|\}$.

Residuals $\mathbf{r}_{\{L,A,C,SZ\}}$ and μ

The residuals denoted in the KKT-conditions 46 with $\mathbf{r}_{\cdot}$ and the dual-gap μ are computed as following:

$$\begin{aligned} r_L &= Hx + g - Ay - Cz \\ r_A &= A^T x - b \\ r_C &= C^T x - s - d \\ r_{SZ} &= SZe \\ \mu &= (z^T s)/m_c \end{aligned} \tag{47}$$

The dual-gap measures how different are the primal and dual problems, which can be looked at as the distance from 'optimality'. From the definition one can easily see that the best result would be achieved either if z or s was zero. This is where the predictor correction comes in and enforces that z or s cannot deviate much from each other (using the example from the lecture, one can imagine a quadrant given by the positive z and s and staying close to $z = s$ as it was shown that the algorithm can get stuck towards one axis when the deviation is high and produce small steps which isn't ideal).

Affine direction

In practice the calculation of the affine direction is a slight variation of the Newton direction

$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} \tag{48}$$

given by an augmented system (slides 24-26 in week 6) that is defined as

$$\begin{bmatrix} \bar{H} & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta y^{\text{aff}} \end{bmatrix} = - \begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix} \tag{49}$$

where

$$\begin{aligned} \bar{H} &= H + C(S^{-1}Z)C^T \\ \bar{r}_L &= r_L - C(S^{-1}Z)(r_C - Z^{-1}r_{sz}) \end{aligned} \tag{50}$$

and the affine step sizes

$$\begin{aligned} \Delta z^{\text{aff}} &= -(S^{-1}Z)C^T \Delta x^{\text{aff}} + (S^{-1}Z)(r_C - Z^{-1}r_{sz}) \\ \Delta s^{\text{aff}} &= -Z^{-1}r_{sz} - Z^{-1}S\Delta z^{\text{aff}} \end{aligned} \tag{51}$$

then the largest $\Delta\alpha^{\text{aff}}$ needs to be found such that

$$\begin{aligned} z + \Delta\alpha^{\text{aff}} \Delta z^{\text{aff}} \geq 0 \\ s + \Delta\alpha^{\text{aff}} \Delta s^{\text{aff}} \geq 0 \end{aligned} \tag{52}$$

Note that finding the solution to the augmented system using LDL factorization is the most computationally heavy step and since it will be reused in the affine centering correction direction the result will be saved to a variable to cut down time of the second factorization. However before moving to the affine centering correction direction step it is needed to compute the affine duality gap and the centering parameter σ.

Duality gap and the centering parameter

The motivation is given by Mehrotra's modification to find if the affine step is good or bad. When it is good the centering parameter σ shall be small, potentially close to 0, however if the affine step is bad, σ will be set to 0. Following the definition of μ:

$$\begin{aligned} \mu^{\text{aff}} &= \frac{1}{m_c}(z + \Delta\alpha^{\text{aff}}\Delta z^{\text{aff}})^T(s + \Delta\alpha^{\text{aff}}\Delta s^{\text{aff}}) \\ \sigma &= \left(\frac{\mu^{\text{aff}}}{\mu}\right)^3 \end{aligned} \tag{53}$$

Affine-centering-correction direction

Having all the necessary information to correct the step of z and s in order to be close to $z = s$ as described in the previous sections, the precomputed LDL-factorization will be used to solve a similar system, but with updated RHS. The solution of this system will be used to calculate the desired steps for z and s as shown:

$$\begin{bmatrix} \bar{H} & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta y^{\text{aff}} \end{bmatrix} = - \begin{bmatrix} \bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}\hat{r}_{sz}) \\ r_A \end{bmatrix} \tag{54}$$

where $\bar{r}_{sz} = r_{sz} + \Delta S^{\text{aff}}\Delta Z^{\text{aff}}e - \sigma\mu e$ and finally

$$\begin{aligned} \Delta z &= -(S^{-1}Z)C^T\Delta x + (S^{-1}Z)(r_C - Z^{-1}\bar{r}_{sz}) \\ \Delta s &= -Z^{-1}\bar{r}_{sz} - Z^{-1}S\Delta z \end{aligned} \tag{55}$$

then the largest $\Delta\alpha$ needs to be found such that

$$\begin{aligned} z + \Delta\alpha\Delta z \geq 0 \\ s + \Delta\alpha\Delta s \geq 0 \end{aligned} \tag{56}$$

The whole process of prediction correction has now ended and the algorithm is closer to the optimum than in the past iteration or at the optimum if it converged (more on convergence later).

Updating the iteration

What is left is to updated the iteration using $\bar{\alpha} = \eta\alpha$ where η is chosen to be 0.995 (note that value can be anything from 0 to 1) as the following:

$$\begin{aligned} x &= x + \bar{\alpha}\Delta x \\ y &= y + \bar{\alpha}\Delta y \\ z &= x + \bar{\alpha}\Delta s \\ s &= x + \bar{\alpha}\Delta s \end{aligned} \tag{57}$$

using the iterates the residuals as well as the dual-gap need to be recalculated as mentioned in the previous subsection.

In the end convergence conditions are check before moving on to the next iteration.

Checking for convergence and the stopping criteria

The implementation provided includes two types of stopping criteria. One, often referred to as simple stopping criterion can be expressed as the norm of the residuals has to be less than or equal to a specified tolerance (ϵ) for A, L and C, also the dual-gap μ is incorporated as in a way that the absolute value $\leq \epsilon_\mu$. The other criterion, which is used in practice, checks whether the infinity norm of the residuals in smaller than a user supplied tolerance multiplied by maximum of 1 or the infinity norm of concatenated matrices used to compute the respective residuals. μ is then checked against the same tolerance times $10^{-2}\mu^0$. Full definition can be found in the slide 21 of week 6.

The general outline of the implemented method is shown in form of pseudocode that references the previous subsections:

```
Require: $H, g, A, b, C, d$ and $x, y, z, s$
Ensure: $(z, s) > 0$
 1: Optional: Calculate the initial point
 2: Compute the residuals $\mathbf{r}_{(\cdot)}$ and the dual-gap $\mu$
 3: Stopped $\leftarrow$ Perform convergence condition check
 4: while $\neg$ Stopped do
 5:     Compute $\hat{H}$
 6:     Factorize using LDL(on the LHS matrix of augmented system from 49)
 7:     Affine direction
 8:     Duality gap $\mu$ and the centering parameter $\sigma$
 9:     Affine-centering-correction direction
10:     Update the iteration
11:     Stopped $\leftarrow$ Perform convergence condition check
12: end while
13: return $x^*$
```

The Matlab code is included in the appendix.

5.1 What is H, g, A, C, b, and d for the Markowitz Portfolio Optimization Problem with R = 15 and the presence of a risk-free security?

$$H = \begin{bmatrix} 2.3 & 0.93 & 0.62 & 0.74 & -0.23 & 0 \\ 0.93 & 1.4 & 0.22 & 0.56 & 0.26 & 0 \\ 0.62 & 0.22 & 1.8 & 0.78 & -0.27 & 0 \\ 0.74 & 0.56 & 0.78 & 3.4 & -0.56 & 0 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad g = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{58}$$

$$A' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 15.1 & 12.5 & 14.7 & 9.02 & 17.68 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ R = 15 \end{bmatrix} \tag{59}$$

$$C = I_6, \quad d = \mathbf{0} \tag{60}$$

The H matrix corresponds to the covariance matrix Σ and additional row and column of zeros is added because by definition there is no risk present.

5.2 Testing on the Markowitz Portfolio Optimization Problem

The benchmarks ran from `Ex5/markowitz_test.m` suggest that the implementation of the interior point algorithm is slower that Matlab's `quadprog` on a fresh run. Note that the option to use IPA was passed to the aforementioned method so that the results are comparable. The absolute difference between the solutions is around 10^{-8} component-wise.

Implementation of the IPA performs 8 iterations before finding the minimum, while Matlab's `quadprog` reports 10 iterations with the default options for tolerance (10^{-8} for step, optimality and convergence tolerance).

Looking at results obtain for the risk-free security from section 4 (using `quadprog`) one can say that the implemented interior point algorithm matches the results as can be seen on figure 13.

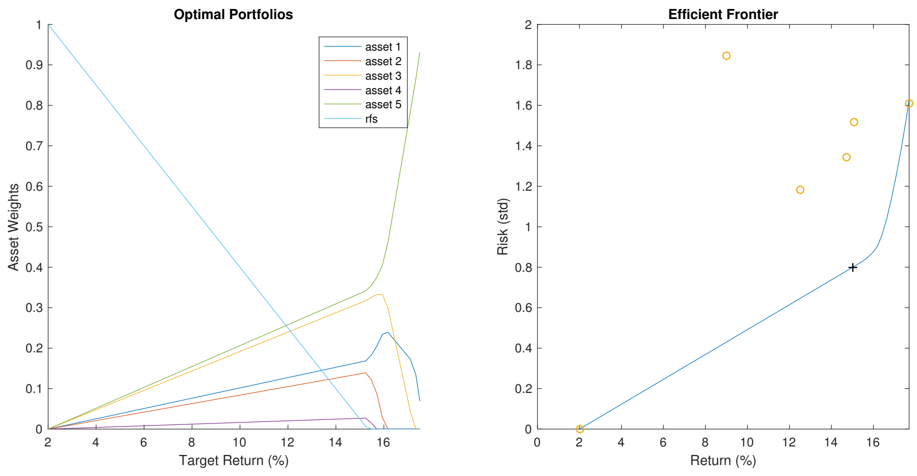


Figure 13: Efficient frontier and optimal portfolio for the situation with a risk-free security using the implementation of the interior point algorithm

5.3 Apply the algorithm to the quadratic program in Problem 3. Plot the iteration sequence in the contour plot.

The quadratic program with inequality constraints and no equality constraints can be written as $\min_x q = \frac{1}{2}x^T Hx + g^T x + \gamma$, where

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, \quad g = \begin{bmatrix} -2 \\ -5 \end{bmatrix}, \quad \gamma = \begin{bmatrix} 1 \\ 6.25 \end{bmatrix} \tag{61}$$

$$A' = \begin{bmatrix} 1 & -1 & -1 & 1 & 0 \\ -2 & -2 & 2 & 0 & 1 \end{bmatrix}, \quad b' = \begin{bmatrix} -2 & -6 & -2 & 0 & 0 \end{bmatrix} \tag{62}$$

or as

$$\min_x q(x) = (x_1 - 1)^2 + (x_2 - 2.5)^2 \tag{63}$$

$$\text{s.t.} \quad x_1 - 2x_2 + 2 \geq 0 \tag{64}$$

$$-x_1 - 2x_2 + 6 \geq 0 \tag{65}$$

$$-x_1 + 2x_2 + 2 \geq 0 \tag{66}$$

$$x_1 \geq 0 \tag{67}$$

$$x_2 \geq 0 \tag{68}$$

The matrix form is preferred in this case as it can be used with our implementation of the interior point algorithm and with slight modifications with Matlab's

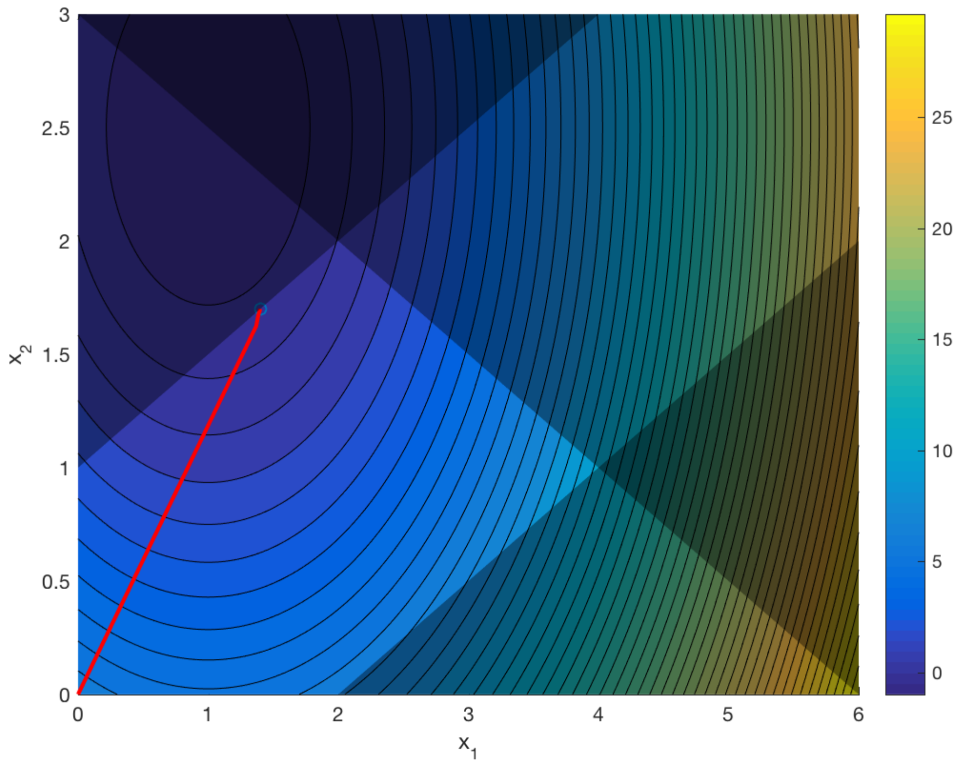


Figure 14: Contour plot for problem 3 with iteration sequence of the interior point algorithm depicted as a red line starting at $(0, 0)$ and ending at the optimum $(1.4, 1.7)$

quadprog. The benchmarks will be run wiht quadprog's options set to use interior point algorithm. A fresh run of both methods results in 0.0266 seconds elapsed for quadprog and 0.0135 for our implementation. Difference between the two solutions obtained by aforementioned methods is roughly 10^{-11}. The iteration sequence seemingly reaches the solution in 2 steps, however upon inspection of the steps taken, the implementation of the interior point algorithm hovers extremely closely around the minimum for up to 3 iterations because $||r_L||$ can't get below the desired tolerance of 10^-8 and afterwards singularities can arise.

The code used to generate the contour plot and the step sequence in figure 14 as well as the benchmark code is included in the appendix (`Ex5/qp_test.m`).

Quadratic optimization

```
clear all
%% Solve Quadratic Problem
H = [6 2 1; 2 5 2; 1 2 4];
g = [-8; -3; -3];

A = [1 0 1; 0 1 1]';
b = [3; 0];

[x,lambda] = EqualityQPSolver(H,g,A,b);

%% Random convex quadratic programs
for j = 1:100
p = 1;
while p ~= 0
H = rand(3);
[~,p] = chol(H);
end
xrand = rand(3,1);
A = rand(3,2);

b = A'*xrand;
g = -H*xrand;


[xt,lambdat] = EqualityQPSolver(H,g,A,b);
if norm(xt-xrand) > 10e-4
    disp('The solution is not correct')
end
end
%% Sensivity
H = [6 2 1; 2 5 2; 1 2 4];
g = [-8; -3; -3];

A = [1 0 1; 0 1 1]';
b = [3; 0];

cx = A;
Wxx = H;
Wxp = [eye(3); zeros(2,3)];
cp = [zeros(3,2); -eye(2)];
[Spx,Splambda] = EqualityQPSensitivity(Wxp,cp,Wxx,cx);
epsilon = 10e-4;

% Sensitivity testing
for i = 1:3
    gt = g;
    bt = b;

    gt(i) = g(i) + epsilon;

    [xt,lambdat] = EqualityQPSolver(H,gt,A,bt);
    Spxt(i,:) = (xt - x)./epsilon;
    Splambdat(i,:) = (lambdat - lambda)./epsilon;
end


for i = 1:2
    gt = g;
    bt = b;

    bt(i) = b(i) + epsilon;

    [xt,lambdat] = EqualityQPSolver(H,gt,A,bt);
    Spxt(i+3,:) = (xt - x)./epsilon;
    Splambdat(i+3,:) = (lambdat - lambda)./epsilon;
end

if norm(Spxt - Spx) > 10e-4 || norm(Splambdat - Splambda) > 10e-4
    disp('The solution is not correct')
end

%% Dual program
t = cputime;
[x,lamdba] = EqualityQPdual(H,g,A,b);
```

```
function [x,lambda] = EqualityQPSolver(H,g,A,b)

if isempty(A) && isempty(b)
    x = H\-g;
    lambda = [];
else
m = size(H,1);
z = [H -A; A' zeros(length(b))]\[-g; b];
x = z(1:m);
lambda = z(m+1:end);
end

end
```

```
function [x,lambda] = EqualityQPdual(H,g,A,b)
AtH = A'/H;
lambda = (AtH*A)\(AtH*g+b);
x = H\(A*lambda-g);

end
```

```
function [Spx,Splambda] = EqualityQPSensitivity(Wxp,cp,Wxx,cx)

[m,n] = size(cx);
Sp = -[Wxp -cp]/[Wxx -cx; -cx' zeros(n,n)];
Spx = Sp(:,1:m);
Splambda = Sp(:,m+1:end);

end
```

Equality constrained quadratic solvers

```
close all
clear
% This script will be used for testing the computing performance of 6
% different QP solvers

u = 0.2;
d0 = 1;

%% DENSE SOLVERS

%% LU solver

n = linspace(10,1000,10);
e1 = zeros(10,1);

t = cputime;
for j = 1:1000
[x,lambda] = LUsolver(n(1),u,d0);
end
e1(1) = (cputime-t)/1000;

for i = 2:4
t = cputime;
for j = 1:100
[x,lambda] = LUsolver(n(i),u,d0);
end
e1(i) = (cputime -t)/100;
end

for i = 5:10
t = cputime;
[x,lambda] = LUsolver(n(i),u,d0);
e1(i) = (cputime -t);
end
%% LDL solver
e2 = zeros(10,1);
```

```

t = cputime;
for j = 1:1000
[x,lambda] = LDLsolver(n(1),u,d0);
end
e2(1) = (cputime-t)/1000;

for i = 2:4
t = cputime;
for j = 1:100
[x,lambda1] = LDLsolver(n(i),u,d0);
end
e2(i) = (cputime-t)/100;
end

for i = 5:10
t = cputime;
[x,lambda1] = LDLsolver(n(i),u,d0);
e2(i) = (cputime -t);
end

%% Null-Space solver
e3 = zeros(10,1);

t = cputime;
for j = 1:1000
[x,lambda] = NullSpaceSolver(n(1),u,d0);
end
e3(1) = (cputime-t)/1000;

for i = 2:4
t = cputime;
for j = 1:100
[x,lambda] = NullSpaceSolver(n(i),u,d0);
end
e3(i) = (cputime-t)/100;
end

for i = 5:10
t = cputime;
[x,lambda] = NullSpaceSolver(n(i),u,d0);
e3(i) = (cputime-t);
end
%% Range-Space solver
e4 = zeros(10,1);

t = cputime;
for j = 1:1000
[x,lambda] = RangeSpaceSolver(n(1),u,d0);
```

```
end
e4(1) = (cputime-t)/1000;

for i = 2:4
t = cputime;
for j = 1:100
[x,lambda] = RangeSpaceSolver(n(i),u,d0);
end
e4(i) = (cputime-t)/100;
end

for i = 4:10
t = cputime;
[x,lambda] = RangeSpaceSolver(n(i),u,d0);
e4(i) = (cputime-t);
end
%% CPUtime plot
subplot(2,1,1)
loglog(n,e1,n,e2,n,e3,n,e4,'linewidth',1.5)
hold on
plot(n,n.^2./10^6,'k--')
legend('LU','LDL','Null-Space','Range-Space','location','northwest')
subplot(2,1,2)
plot(n,e1,n,e2,n,e3,n,e4,'linewidth',1.5)
legend('LU','LDL','Null-Space','Range-Space','location','northwest')
%% SPARSE SOLVERS

%% Sparsity pattern
K = KKTmatrix(100,0.2,1);
figure
spy(K);
legend('non-zero elements','location','southeast')

%% LU sparse solver

e5 = zeros(10,1);

t = cputime;
for j = 1:1000
[x,lambda] = LUsparseSolver(n(1),u,d0);
end
e5(1) = (cputime-t)/1000;

for i = 2:4
t = cputime;
for j = 1:100
[x,lambda] = LUsparseSolver(n(i),u,d0);
end
e5(i) = (cputime -t)/100;
```

```
end

for i = 5:10
t = cputime;
[x,lambda] = LDLsparseSolver(n(i),u,d0);
e5(i) = (cputime -t);
end

%% LDL sparse solver

e6 = zeros(10,1);

t = cputime;
for j = 1:1000
[x,lambda] = LDLsparseSolver(n(1),u,d0);
end
e6(1) = (cputime-t)/1000;

for i = 2:5
t = cputime;
for j = 1:100
[x,lambda] = LDLsparseSolver(n(i),u,d0);
end
e6(i) = (cputime -t)/100;
end

for i = 6:10
t = cputime;
[x,lambda] = LDLsparseSolver(n(i),u,d0);
e6(i) = (cputime -t);
end

%% CPUtime plot
figure
subplot(2,1,1)
loglog(n,e3,n,e4,n,e5,n,e6,'linewidth',1.5)
legend('Null-Space','Range-Space','LU sparse','LDL sparse','location'
    ,'northwest')
hold on
plot(n,n.^2./10^6,'k--')
subplot(2,1,2)
plot(n,e1,n,e2,n,e3,n,e4,n,e5,n,e6,'linewidth',1.5)
legend('LU','LDL','Null-Space','Range-Space','LU sparse','LDL sparse'
    ,'location','northwest')
```

```
function [K,d] = KKTmatrix(n,u,d0)
% This function constructs the KKT matrix for solving the quadratic
% program in problem 2

```

```
[H,g,A,b] = matrixForm(n,u,d0);
K = [H -A; A' zeros(length(b))];
d = [-g; b];

end
```

```
function [H,g,A,b] = matrixForm(n,u,d0)
% This function constructs H,g,A and b to express problem 2 as a
    quadratic
% program of the form 1/2x'Hx + g'x s.t. A'x = b

H = eye(n+1);
g = -u*ones(n+1,1);
At = [diag(-ones(n,1)) + diag(ones(n-1,1),-1) zeros(n,1)];
At(n,n+1) = -1;
At(1,n) = 1;
A = At';
b = zeros(n,1);
b(1) = -d0;

end
```

```
function [x,lambda] = LDLsolver(n,u,d0)
% This function uses LDL factorization for solving the quadratic
    program in
% problem 2

[K,d] = KKTmatrix(n,u,d0);

z = zeros(size(K,1),1);
[L,D,p] = ldl(K,'lower','vector');
z(p) = L'\(D\(L\d(p)));
x = z(1:n+1);
lambda = z(n+2:end);
end
```

```
function [x,lambda] = LDLsparseSolver(n,u,d0)
% This function uses LDL factorization for solving the quadratic
    program in
% problem 2

[K,d] = KKTmatrix(n,u,d0);
K = sparse(K);

z = zeros(size(K,1),1);
[L,D,p] = ldl(K,'lower','vector');
z(p) = L'\(D\(L\d(p)));
x = z(1:n+1);
lambda = z(n+2:end);
end
```

```matlab
function [x,lambda] = LUsolver(n,u,d0)
% This problem uses LU factorization for solving the quadratic
    program in
% problem 2
[K,d] = KKTmatrix(n,u,d0);

[L,U,p] = lu(K,'vector');
z = U\(L\d(p));

x = z(1:n+1);
lambda = z(n+2:end);
end
```

```matlab
function [x,lambda] = LUsparseSolver(n,u,d0)
% This problem uses LU factorization for solving the quadratic
    program in
% problem 2
[K,d] = KKTmatrix(n,u,d0);
K = sparse(K);
[L,U,p] = lu(K,'vector');
z = U\(L\d(p));

x = z(1:n+1);
lambda = z(n+2:end);
end
```

```matlab
function [x,lambda] = NullSpaceSolver(n,u,d0)
% This problem uses the null-space method for solving the quadratic
    program in
% problem 2
[H,g,A,b] = matrixForm(n,u,d0);

[Q,Rbar] = qr(A);
m1 = size(Rbar,2);
Q1 = Q(:,1:m1);
Q2 = Q(:,m1+1:end);
R = Rbar(1:m1,1:m1);
xy = R'\b;
xz = (Q2'*H*Q2)\(-Q2'*(H*Q1*xy+g));
x = Q1*xy + Q2*xz;
lambda = R\Q1'*(H*x+g);
end
```

```matlab
function [x,lambda] = RangeSpaceSolver(n,u,d0)
% This problem uses the range-space method for solving the quadratic
    program in
% problem 2
[H,g,A,b] = matrixForm(n,u,d0);
L = chol(H,'lower');
```

```
v = L'\(L\g);
La = chol(A'*(L'\(L\A)),'lower');
lambda = La'\(La\(b+A'*v));
x = L'\(L\(A*lambda-g));
end
```

Inequality constrained quadratic programming

```
function [x,lambda,info] = ActiveSetMethod(H,g,A,b,xk)
% This function solves strictly convex quadratic programs with
    inequality
% constraints.
% The function returns:
% x: solution to the optimization problem.
% lambda: lagrange multipliers at the solution
% info.iter: number of iterations
% info.xs: iteration sequence
% info.ws: working set in each iteration

m = size(A,1);
lambda = zeros(m,1);

% Initial working set
w = (A*xk == 0);

stop = false;
k = 1;
while ~ stop
    ws(:,k) = w;
    xs(:,k) = xk;
    Ak = A(w,:);
    bk = b(w);
    gk = H*xk+g;
    if isempty(Ak) && isempty(bk)
        pk = EqualityQPSolver(H,gk,[],[]);
    else
        pk = EqualityQPSolver(H,gk,Ak',zeros(size(Ak,1),1));
    end
    if norm(pk) < 10e-4
        lambdak = Ak'\(H*xk +g);
        if sum(lambdak < 0) == 0
            x = xk;
            lambda(w) = lambdak;
            stop = true;
        else
           [~,idx] = min(lambdak);
           wk = ones(length(lambdak),1);
```

```
            wk(idx) = 0;
            w(w == 1) = wk;
        end
    else
        Ai = A(~w,:);
        bi = b(~w,:);
        i = Ai*pk < 0;
        alpha = ones(length(bi),1);
        alpha(i) = (bi(i)-Ai(i,:)*xk)./(Ai(i,:)*pk);
        [alphak,idx] = min(alpha);
        if alphak < 1
            xk = xk + alphak*pk;
            wk = zeros(length(alpha),1);
            wk(idx) = 1;
            w(w == 0) = wk;
        else
            xk = xk + pk;
        end
    end
    k = k+1;
end
info.iter = k;
info.xs = xs;
info.ws = ws;
end
```

```
clear;
close all;

H = [2 0; 0 2];
g = [-2 ; -5];
A = [1 -1 -1 1 0;-2 -2 2 0 1]';
b = [-2; -6; -2; 0; 0];
%% 3.1 : Contour plot with constraints
x=-2:0.05:5;
y=-2:0.05:5;
[X Y] = meshgrid(x,y);

q=(X-1).^2+(Y-2.5).^2;
contourf(X,Y,q,linspace(0,30,50))
axis([-1 4.5 -1 3])
colorbar;

yc1=x/2+1;
yc2=-x/2+3;
yc3=x/2-1;
xc4=zeros(size(x));
yc5=zeros(size(y));
hold on
```

```
%      plot(x,yc1);
%      plot(x,yc2);
%      plot(x,yc3);
%      plot(xc4,y);
%      plot(x,yc5);
    fill([x x(end) x(1)],[yc1 y(end) y(end)],[0.7 0.7 0.7],'facealpha
       ',0.3, 'facecolor', 'black')
    fill([x x(end) x(1)],[yc2 y(end) y(end)],[0.7 0.7 0.7],'facealpha
       ',0.3, 'facecolor', 'black')
    fill([x x(end) x(1)],[yc3 y(1) y(1)],[0.7 0.7 0.7],'facealpha'
       ,0.3, 'facecolor', 'black')
    fill([xc4 x(1) x(1)],[y y(end) y(1)],[0.7 0.7 0.7],'facealpha'
       ,0.3, 'facecolor', 'black')
    fill([x x(end) x(1)],[yc5 y(1) y(1)],[0.7 0.7 0.7],'facealpha'
       ,0.3, 'facecolor', 'black')

X = quadprog(H,g,-A,-b);
hold on
plot(X(1),X(2),'ro', 'MarkerSize', 6)

%% 3.8 : Find a feasible point using linprog

[m,n] = size(A);
xe = [-1; -1];
ganma = sign(A*xe - b);
Af = [A diag(ganma); zeros(m,n) eye(m)];
bf = [b; zeros(m,1)];
f = [zeros(n,1); ones(m,1)];
z = linprog(f,-Af,-bf);
xk = z(1:n);
[x,lambda,info] = ActiveSetMethod(H,g,A,b,xk);

plot(info.xs(1,:),info.xs(2,:),'r--','LineWidth',1.5)

%% 3.10 %%  (16.47) Big M Penalty Method: L-inf norm


Hm = [2 0 0;...
      0 2 0;...
      0 0 1e-3];
gm = [g ; big_M];
xm = [-1 ; -1 ; v];
Aineq_m =   [A ones(size(A,1),1); 0 0 1;];
bineq_m = [b ; 0];

% Implementation:
[x,lambda,info] = ActiveSetMethod(Hm,gm,Aineq_m,bineq_m,xm);

% Summary
```

```
plot(info.xs(1,:),info.xs(2,:),'r--','LineWidth',1.5)



%% 3.10 %%  (16.47) Big M Penalty Method: L-1 norm

% NOTE: The objective function should be minimized wrt x AND eta
%        - eta should equal 0 if global solution is found
%        - set eta large enough such that all constraints are
    satisfied
%        - if during solving eta converges to positive number,
%          increase big_M until eta converges to 0
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Setting up Hyperparameters
eta = 200 ;
big_M = 1;

% Setting up Variables
Hm = [2 0 0 0 0 0 0;...
      0 2 0 0 0 0 0;...
      0 0 1 0 0 0 0;...
      0 0 0 1 0 0 0;...
      0 0 0 0 1 0 0;...
      0 0 0 0 0 1 0;...
      0 0 0 0 0 0 1];
gm = [g ; big_M; big_M; big_M; big_M; big_M;];
xm = [-1 ; -1 ; eta; eta; eta; eta; eta;];
one = ones(5,1);
unos = ones(6,1);
Aineq_m = [A one; 0 0 1;];
Aineq_m = [Aineq_m unos unos unos unos];
bineq_m = [b ; 0; 0; 0; 0; 0;];

% Implementation:
[x,lambda,info] = ActiveSetMethod(Hm,gm,Aineq_m,bineq_m,xm);

% Summary
x
info.iter
info.ws
info.xs

plot(info.xs(1,:),info.xs(2,:),'r--','LineWidth',1.5)
% Plot with iterations
% RM: If anyone gets a chance, that would be great! : )
%  but maybe they're not needed
%  Right Here!
%
%
%% (16.48) Missing

%  Setting up Hyperparameters
v = 200 ;
big_M = 0.005;

%  Setting up Variables
e =  [1 1 1 1 1];  % <-- FIGURE OUT HOW e GOES BELOW IN G AND H!
                   %     Hm, gm and xm might look different than
                       below
Hm = [2 0 0 0 0 0 0;...
      0 2 0 0 0 0 0;...
      0 0 0 0 0 0 0;...
      0 0 0 0 0 0 0;...
      0 0 0 0 0 0 0;...
      0 0 0 0 0 0 0;...
      0 0 0 0 0 0 0];
gm = [g ; big_M; big_M; big_M; big_M; big_M;];
xm = [0 ; 0 ; v; v; v; v; v;];
one = ones(5,1);
unos = ones(6,1);
Aineq_m =   [A one; 0 0 1;];
Aineq_m = [Aineq_m unos unos unos unos];
bineq_m = [b ; 0; 0; 0; 0; 0;];

% Implementation:
[x,lambda,info] = ActiveSetMethod(Hm,gm,Aineq_m,bineq_m,xm);

% Summary
x
info.iter
info.ws
info.xs

% Plot with iterations
% RM: If anyone gets a chance, that would be great! : )
%     but maybe they're not needed
%  Right Here!
```

Markowitz portfolio optimization

```
clear all;
clc;

```

```matlab
% RUN SECTION BY SECTION : variable names might repeat
%% Given Variables: Asset returns, target portfolio return, assets
    covariance
returns = [15.1, 12.5, 14.7, 9.02, 17.68];
R = 10;
covariance = [2.3 .93 .62 .74 -.23;
              0.93 1.4 0.22 0.56 0.26;
              .62  .22 1.8  .78  -0.27;
              .74 .56 .78 3.4 -0.56;
              -0.23 0.26 -0.27 -0.56 2.6];
asset_var = diag(covariance);    % variance of individual asssets
asset_std = sqrt(asset_var);     % std. dev of individual asssets

%% Setting up variables according to quadprog() documentation
H = covariance;
f = []; %f is not used, hence [] the empty brace

% This constraint allows us to specify our target return
A1 = returns;
b1 = R;

% This constraints makes portfolio weights (x) equal to 1
A2 = [1,1,1,1,1];
b2 = 1;

% Combining the above 2 equality constraints into 1; to make quadprog
    happy
Aeq = [A1; A2];
beq = [b1; b2];

% Inequality constraint to disallow short-selling
Aineq = -eye(5);
bineq = zeros(5,1);

% Lower and Upper Bounds on x
LB = zeros(5,1);
UB = [1; 1; 1; 1; 1];
%% Solving
%optimal port. weights found by quadprog
x = quadprog( H, f, Aineq, bineq, Aeq, beq, LB, UB );

%Calculating portfolio variance (i.e., inherent risk of portfolio)
port_var = x'*covariance*x; % --> 2.0923
port_std = sqrt(port_var);  % --> 1.4465

%Calculating porfolios expected return (should be 10%)
port_expected_return = returns*x;

%% Efficient Frontier (without Risk Free Security)
```

```matlab
% We minimize the variance for a sequence of target returns
b1 = linspace(2,25,1000); %seq of target returns

% We iterate on the seq of target returns
xset = zeros(5,1000);
for i=1:1000
    beq = [b1(i); b2];
    xset(: ,i) = quadprog( H, f, Aineq, bineq, Aeq, beq, LB, UB);
end
%% Plot
plot(b1, xset, 'LineWidth',5); hold on;
title('Optimal Portfolios');
legend('asset 1', 'asset 2', 'asset 3', 'asset 4', 'asset 5');
xlabel('Target Return (%)');
ylabel('Asset Weights');
axis([9 17.5 0 1]); hold off;

%% Obtaining Variance of each optimal porfolio in xset to plot eff
    frontier

%Calculating variance of all optimal porfolios (found in xset)
opt_port_var = zeros(1000,1);
opt_port_std = zeros(1000,1);
opt_port_return= zeros(1000,1); %should be the same as b1
for i=1:length(xset)
    opt_port_var(i) = xset(:,i)'*covariance*xset(:,i);
    opt_port_std(i) = sqrt(opt_port_var(i));

    %Calculating returns to verify with b1 that they're correct
    opt_port_return(i) = returns*xset(:,i);
end

%% Plot
plot(opt_port_return(305:682), opt_port_std(305:682),  'LineWidth',5)
title('Efficient Frontier');hold on;
xlabel('Return (%)');
ylabel('Risk (std)');
scatter(returns, asset_std, 'filled');
axis([0 20 0 2]); hold off;

%% Adding Risk Free Security

returns = [15.1, 12.5, 14.7, 9.02, 17.68, 2];
R = 15;
covariance = [2.3 .93 .62 .74 -.23, 0;
              0.93 1.4 0.22 0.56 0.26, 0;
              .62  .22 1.8  .78  -0.27, 0;
              .74 .56 .78 3.4 -0.56, 0;
              -0.23 0.26 -0.27 -0.56 2.6, 0;
```

```
                0    0   0   0    0   0;];
asset_var2 = diag(covariance);
asset_std2 = sqrt(asset_var2);


%% Setting up variables according to quadprog() documentation
H = covariance;
f = []; %f is not used, hence [] the empty brace

% This constraint allows us to specify our target return
A1 = returns;
b1 = R;

% This constraints makes portfolio weights (x) equal to 1
A2 = [1,1,1,1,1,1];
b2 = 1;

% Combining the above 2 equality constraints into 1; to make quadprog
    happy
Aeq = [A1; A2];
beq = [b1; b2];

% Inequality constraint to disallow short-selling
Aineq = -eye(6);
bineq = zeros(6,1);

% Lower and Upper Bounds on x
LB = zeros(6,1);
UB = [1; 1; 1; 1; 1; 1];
%% Solving
%optimal port. weights found by quadprog
x = quadprog( H, f, Aineq, bineq, Aeq, beq, LB, UB );

%Calculating portfolio variance (i.e., inherent risk of portfolio)
port_var = x'*covariance*x;
port_std = sqrt(port_var);

%Calculating porfolios expected return (should be 10%)
port_expected_return = returns*x;

%% Efficient Frontier (with Risk Free Security)
% We minimize the variance for a sequence of target returns
b1 = linspace(2,25,1000); %seq of target returns

% We iterate on the seq of target returns
xset = zeros(6,1000);
for i=1:1000
    beq = [b1(i); b2];
    xset(: ,i) = quadprog( H, f, Aineq, bineq, Aeq, beq, LB, UB);
```

```
end

%% Plot
plot(b1, xset, 'LineWidth',5); hold on;
title('Optimal Portfolios');
legend('asset 1', 'asset 2', 'asset 3', 'asset 4', 'asset 5', 'rfs');
xlabel('Target Return (%)');
ylabel('Asset Weights');
axis([2 17.5 0 1]);
vline(15, 'k+: '); hold off;

%% Obtaining Variance of each optimal porfolio in xset to plot eff
    frontier

%Calculating variance of all optimal porfolios (found in xset)
opt_port_var = zeros(1000,1);
opt_port_std = zeros(1000,1);
opt_port_return= zeros(1000,1); %should be the same as b1
for i=1:length(xset)
    opt_port_var(i) = xset(:,i)'*covariance*xset(:,i);
    opt_port_std(i) = sqrt(opt_port_var(i));

    %Calculating returns to verify with b1 that they're correct
    opt_port_return(i) = returns*xset(:,i);
end

%% Plot
plot(opt_port_return(1:685), opt_port_std(1:685), 'LineWidth',5)
title('Efficient Frontier with Risk Free Asset');hold on;
scatter(returns, asset_std2, 'filled')
xlabel('Return (%)');
ylabel('Risk (std)');
plot(15, port_std, 'k o' , 'MarkerSize',20);
hold off;
```

Interior-point algorithm for convex quadratic programming

```
function [ x_r, xiter ] = pcipa( H, g, A, b, C, d, x, y, z, s )
% Predictor-Corrector Interior-Point Algorithm

mc = length(s);
S = diag(s);
Z = diag(z);
e = ones(mc, 1);
% a = 0.999;
% ahat = 1;


%Initial step
rL = H*x+g-A*y-C*z;
rA = b-A'*x;
rC = s+d-C'*x;
rsz = s.*z;
Hhat = H+C*(inv(S)*Z)*C';
HA = [Hhat -A; -A' zeros(size(A,2))];
rhatL = rL - C*((inv(S)*Z))*(rC-inv(Z)*rsz);
[L,D,p] = ldl(HA,'vector');
bl = [-rhatL; -rA];
xx(p) = L'\(D\(L\bl(p)));
dxaff = xx(1:end-size(A,2));
dyaff = xx(end-size(A,2)+1:end);
dzaff = -((inv(S)*Z))*C'*dxaff'+((inv(S)*Z))*(rC-inv(Z)*rsz);
dsaff = -inv(Z)*rsz-(inv(Z)*S)*dzaff;
z = max(1, abs(z+dzaff));
s = max(1, abs(s+dsaff));


% Compute residuals
rL = H*x+g-A*y-C*z;
rA = b-A'*x;
rC = s+d-C'*x;
rsz = s.*z;
mu = (z'*s)/mc;


iter = 0; % current iteration number
maxiter = 200;
AdvancedConvCheck = true;
xiter = x;
% epsilon = 1e-8;
% mu0 = 1e-8;
eta = 0.995; % or 0.95 not sure if mistake or on purpose (slides say
    0.995)
epsL=1e-8; epsA=1e-8; epsC=1e-8; epsmu=1e-8;
if size(A,2) == 0
    epsL = 1e-2;
end
Stop = check_convergence();

while ~Stop && (iter <= maxiter)
    iter = iter + 1;
    % Precompute
    S=diag(s); Z=diag(z);
    invStimesZ = diag(z./s);
    % Compute Hhat
    Hhat = H+C*(invStimesZ)*C';
    % Compute LDL-factorization of [Hhat -A; -A' 0]
    HA = [Hhat -A; -A' zeros(size(A,2))];

    % ## Affine Direction
    rhatL = rL - C*((z./s).*(rC-rsz./z));% rL - C*(invStimesZ)*(rC-inv(Z)*rsz);
    % Solve HA*[dxaff; dyaff] = [-rhatL; -rA] using LDL (Ax=b)
    [L,D,p] = ldl(HA,'vector');
    bl = [-rhatL; -rA];
    xx(p) = L'\(D\(L\bl(p)));
    dxaff = xx(1:end-size(A,2));
    dyaff = xx(end-size(A,2)+1:end);
    % Compute dzaff
    dzaff = -(invStimesZ)*C'*dxaff' + (z./s).*(rC-rsz./z);%-(invStimesZ)*C'*dxaff' + (invStimesZ)*(rC-inv(Z)*rsz);%%(C'*(-(z./s).*dxaff'))
    % Compute dsaff
    dsaff = -((rsz+s.*dzaff)./z);%-inv(Z)*rsz - inv(Z)*S*dzaff;%%

    % ### Compute the larget aaff
    %     s.t. z+aaff*dzaff >= 0 and s+aaff*dsaff >= 0
    aaff = 1.0;
    dzs = find(dzaff < 0.0);
    dss = find(dsaff < 0.0);
    if (~isempty(dzs))
        aaff = min(1, min(-z(dzs)./dzaff(dzs)));
    end
    if (~isempty(dss))
        aaff = min(aaff, min(-s(dss)./dsaff(dss)));
    end
    %disp(sprintf('Alpha affine is %f', aaff));

    % ## Duality gap and centering parameter
    % affine duality gap
    muaff = ((z+aaff*dzaff)'*(s+aaff*dsaff))/mc;
    % centering param
    sigma = (muaff/mu)^3;

    % ## Affine-centering-correction direction
    rhatsz = rsz + dsaff.*dzaff - sigma*mu*e;
    rhatL = rL - C*((z./s).*(rC-rhatsz./z));%rhatL = rL - C*(invStimesZ)*(rC-inv(Z)*rhatsz);
    % Solve the system again
    bl = [-rhatL; -rA];
    xx(p) = L'\(D\(L\bl(p)));
    dx = xx(1:end-size(A,2));
    dy = xx(end-size(A,2)+1:end);
    % Compute dzaff
    dz = -(invStimesZ)*C'*dxaff' + (z./s).*(rC-rhatsz./z);%-(
        invStimesZ)*C'*dxaff' + (invStimesZ)*(rC-inv(Z)*rhatsz);%%(C
        '*(-(z./s).*dx'))
    % Compute dsaff
    ds = -((rhatsz+s.*dz)./z);%-inv(Z)*rsz - inv(Z)*S*dzaff;%%

    % ### Compute the largest alpha
    %     s.t. z+alpha*dz >= 0 and s+alpha*ds >= 0
    alpha = 1;
    dzs = find(dz < 0.0);
    dss = find(ds < 0.0);
    if (~isempty(dzs))
        alpha = min(1, min(-z(dzs)./dz(dzs)));
    end
    if (~isempty(dss))
        alpha = min(alpha, min(-s(dss)./ds(dss)));
    end
    %disp(sprintf('Alpha is %f', alpha));

    % ## Update iteration
    ahat = eta*(alpha);
    x = x+ahat*dx';
    xiter = [xiter x];
    y = y+ahat*dy';
    z = z+ahat*dz;
    s = s+ahat*ds;
    S = diag(s);
    Z = diag(z);
    % residuals
    rL = H*x+g-A*y-C*z;
    rA = b-A'*x;
    rC = s+d-C'*x;
    rsz = s.*z;
    mu = (z'*s)/mc;

    % ## Check for convergence conditions
    Stop = check_convergence();

end % end while

x_r = x; y_r = y; s_r = s; z_r = z;

%%% Helper functions %%%
    function [converges] = check_convergence()
        AdvancedConvCheck = true;

        %for markowitz when close to singularity for the last
```

```
           iterations
        if (isnan(max(H*x+g-A*y-C*z))) || (max(rL) > 1000000)
            converges = true;
            return
        end

        if AdvancedConvCheck
            arL = (max(H*x+g-A*y-C*z) <= 1e-2*max(1,norm([H g A C], 'inf')));
            if size(rA,1) == 0
                arA = true;
            else
                arA = (max(b-A'*x) <= 1e-8*max(1,norm([A' b], 'inf')));
            end
            arC = (max(d+s-C'*x) <= 1e-8*max(1,norm([eye(size(C,2)) d C'], 'inf')));
            amu = (mu <= 1e-8*10^(-2)*1e-4);
            %disp(sprintf('\nChecking covergence %d (rL=%f, rA=%f, rC=%f, mu=%f)\n', iter, max(H*x+g-A*y-C*z), max(b-A'*x), max(d+s-C'*x), (mu)));

            converges = arL && arA && arC && amu;
        else
            %disp(sprintf('\nChecking covergence %d (rL=%f, rA=%f, rC=%f, mu=%f)\n', iter, norm(rL), norm(rA), norm(rC), abs(mu)));
            if size(rA,1) == 0
                convrA = true;
            else
                convrA = norm(rA)<=epsA;
            end

            converges = (norm(rL)<=epsL) && (convrA) && (norm(rC)<=epsC) && (abs(mu)<=epsmu);
        end
    end

end
```

```
G = [   2.3     .93     .62     .74     -.23    0;
        0.93    1.4     0.22    0.56    0.26    0;
        .62     .22     1.8     .78     -0.27   0;
        .74     .56     .78     3.4     -0.56   0;
        -0.23   0.26    -0.27   -0.56   2.6     0;
        0       0       0       0       0       0;
];

r_f = 2;
```

```
returns = [15.1, 12.5, 14.7, 9.02, 17.68, r_f];
R = 15;

% Quadprog form
H = G; % covariance matrix
g = zeros(6,1);
A = [ones(1,6); returns]';
b = [1; R]; % target R
C = eye(6);
d = zeros(6,1);

% just testing
x = 0*ones(6,1);
y = ones(2,1);
z = ones(6,1);
s = ones(6,1);

tic
[x_ours, xsteps] = pcipa( H, g, A, b, C, d, x, y, z, s );
toc
% x_qp = 0.1655 0.1365 0.3115 0.0266 0.3352 0.0247


%% Setting up variables according to quadprog() documentation
% This constraint allows us to specify our target return
A1 = returns;
b1 = R;

% This constraints makes portfolio weights (x) equal to 1
A2 = [1,1,1,1,1,1];
b2 = 1;

% Combining the above 2 equality constraints into 1; to make quadprog
    happy
Aeq = [A1; A2];
beq = [b1; b2];

% Inequality constraint to disallow short-selling
Aineq = -eye(6);
bineq = zeros(6,1);

% Lower and Upper Bounds on x
LB = zeros(6,1);
UB = [1; 1; 1; 1; 1; 1];

%optimal port. weights found by quadprog
options = optimoptions(@quadprog, 'Algorithm', 'interior-point-convex
    ', 'MaxIterations', 200, 'Display', 'none');
tic
```

```
[x_qp, fval, exitflag, output, lambda] = quadprog( H, [], Aineq,
    bineq, Aeq, beq, LB, UB, ones(6,1) ,options );
toc

disp((x_qp - x_ours)')
disp(xsteps)

%% Plotting Markowitz
G = [   2.3     .93     .62     .74     -.23    0;
        0.93    1.4     0.22    0.56    0.26    0;
        .62     .22     1.8     .78     -0.27   0;
        .74     .56     .78     3.4     -0.56   0;
        -0.23   0.26    -0.27   -0.56   2.6     0;
        0       0       0       0       0       0;
];
asset_var2 = diag(G);
asset_std2 = sqrt(asset_var2);
r_f = 2;
R = 15;
returns = [15.1, 12.5, 14.7, 9.02, 17.68, r_f];

% Quadprog form
H = G; % covariance matrix
g = zeros(6,1);
A = [ones(1,6); returns]';
b = [1; R]; % target R
C = eye(6);
d = zeros(6,1);
% We iterate on the seq of target returns
xset = zeros(6,100);
b1 = linspace(2,25,100); %seq of target returns
for i=1:100
    [x1,x1steps] = pcipa( H, g, A, [1; b1(i)], C, d, x, y, z, s );
    xset(: ,i) = x1;
end
figure;
subplot(1,2,1)
hold on;
xsum = sum(xset,1);
plot(b1, xset);
title('Optimal Portfolios');
legend('asset 1', 'asset 2', 'asset 3', 'asset 4', 'asset 5', 'rfs');
xlabel('Target Return (%)');
ylabel('Asset Weights'); hold off;
axis([2 17.5 0 1]);

% Obtaining Variance of each optimal porfolio in xset to plot eff
    frontier

```

```
%Calculating variance of all optimal porfolios (found in xset)
opt_port_var = zeros(100);
opt_port_std = zeros(100);
opt_port_return= zeros(100); %should be the same as b1
for i=1:length(xset)
    opt_port_var(i) = xset(:,i)'*G*xset(:,i);
    opt_port_std(i) = sqrt(opt_port_var(i));

    %Calculating returns to verify with b1 that they're correct
    opt_port_return(i) = returns*xset(:,i);
end

subplot(1,2,2)
plot(opt_port_return, opt_port_std)
title('Efficient Frontier');hold on;
scatter(returns, asset_std2)
plot(15, port_std, 'k +' );
xlabel('Return (%)');
ylabel('Risk (std)');  hold off;
xlim([0 17.7])
```

```
close all

%% Rewrite the problem in the quadratic form
H = [ 2 0;
      0 2 ];
g = [-2; -5];
% inequality constraints
C = [-1 2; 1 2; 1 -2];
d = [2; 6; 2];

%% Calculate the minimum using the implemented algorith
x = [0;0];
y = 0.5*ones(0,1);
z = 0.5*ones(3,1);
s = 0.5*ones(3,1);
A = zeros(2,0);
b = zeros(0,1);

%% Check with Matlab's `quadprog`
options = optimoptions('quadprog', 'Algorithm', 'interior-point-
    convex', 'MaxIterations', 200);
tic
xqp = quadprog( H, g, C, d, [], [], zeros(2,1), [], [], options );
toc

%% Our implementation of interior point algorithm
%  (arguments need to be a bit modified compared to Matlab's quadprog
%   to match the definition's on the slides)
```

```matlab
tic
[xopt, xsteps] = pcipa(H,g,A,b,-1*C',-1*d,x,y,z,s);
toc
%xsteps = xsteps(:,all(~isnan(xsteps))); % remove NaN

%% Plot the function and the constraints
[X1,X2] = meshgrid(0:0.01:6, 0:0.01:3);
Q = X1.^2 - 2.*X1 + X2.^2 - 5.*X2 + 6.25;

figure
hold on
contourf(X1,X2,Q,50), colorbar
%contour(X1,X2,Q,20,'ShowText','on','LineWidth',2), colorbar

scatter(xopt(1), xopt(2))
plot(xsteps(1,:), xsteps(2,:),'-','LineWidth',2,'Color','red')

patch('Faces',[1 2 3], 'Vertices', [2 0; 6 2; 6 0], 'FaceColor', '
    black', 'FaceAlpha', 0.4, 'EdgeColor', 'none')
patch('Faces',[1 2 3], 'Vertices', [0 1; 6 4; 0 4], 'FaceColor', '
    black', 'FaceAlpha', 0.4, 'EdgeColor', 'none')
patch('Faces',[1 2 3], 'Vertices', [0 3; 6 0; 6 3], 'FaceColor', '
    black', 'FaceAlpha', 0.4, 'EdgeColor', 'none')

xlim([0 6]); ylim([0 3]);
xlabel('x_1'); ylabel('x_2');

%% Check the results
format long
abs(xopt - xqp)'
format short
```