Comparison of numerical performance between SeDuMi and MOSEK algorithms

ME5414: Optimization Techniques for Dynamical Systems

<u>Project Report (First Part)</u>

FULL NAME	MATRICULATION NUMBER
Chung Jun Yu Chester	A0201459J

1. Description of SeDuMi (Self-Dual Minimization) algorithm

SeDuMi is a solver for convex optimization problems, based on the primal-dual interior-point method. The algorithm is commonly used due to its conservative memory usage, capability to handle large-scale optimization problems, and ease of use. In this section, a description of the SeDuMi algorithm, beginning from the problem formulation and ending with the convergence testing and termination criteria, will be discussed in detail.

1.1 Problem formulation

The SeDuMi algorithm solves optimization problems of the form:

$$minimize\ c^T x$$

$$\begin{aligned} subject\ to\ Ax &= b \\ Gx &\leq h \end{aligned}$$

where *c* refers to a vector of coefficients for the linear objective function, *x* refers to a vector of decision variables, *A* refers to a matrix of coefficients for the equality constraints of the linear programming optimization problem, *b* refers to the right-hand side vector of constraint values for the equality constraints, *G* refers to the matrix of constraint coefficients for inequality constraints, and *h* is the right-hand side vector of constraint values for inequality constraints.

1.2 Primal-dual interior-point method

The SeDuMi algorithm is mainly dependent on the primal-dual interior-point method, which necessitates solving a series of barrier problems that estimate the original underlying optimization problem. The barrier problem is derived by including a logarithmic barrier term in the inequality constraints, which penalizes solutions which violate the constraints. The barrier term is controlled by a parameter known as the barrier parameter, which is gradually reduced to zero as the algorithm proceeds.

The barrier function for inequality constraints can be expressed as follows:

$$\varphi(x) = -\sum \log(h - Gx)$$

where *φ(x)* refers to the barrier function, and *h-Gx* refers to the vector of violations of the inequality constraints. The barrier function is multiplied by a positive coefficient μ, known as the barrier parameter, which regulates the trade-off between the accuracy of the solution and the computational cost. The barrier parameter adjusts the width of the barrier around the feasible region and is generally selected to be a small positive number at the beginning of the algorithm. As the algorithm is iterated through, the barrier parameter is gradually decreased, to enhance the solution accuracy. The modified objective function for the barrier problem is formulated as follows:

$$f(x) = c^T x + \mu * \varphi(x)$$

The solution of the barrier problem is obtained by solving a sequence of primal-dual interior-point problems, which involve solving a sequence of linear systems of equations that correspond to the Karush-Kuhn-Tucker (KKT) conditions of the barrier problem, which involve the dual variables associated with the inequality constraints, the gradient of the objective function, and the matrix of constraint coefficients.

The algorithm iteratively solves a chain of barrier problems with increasing values of the barrier parameter. At each iteration, the primal and dual variables, together with the barrier parameter are updated, until the duality gap between the primal and dual objective values is below a specified tolerance level. The algorithm also consists of a predictor-corrector step which enhances the stability and convergence rate of the algorithm.

1.3 Initialization

The SeDuMi algorithm starts by initializing the primal and dual variables, together with the barrier parameter. The primal and dual variables are selected to satisfy the primal and dual feasibility constraints, while the barrier parameter is set to be a small positive number. The initial values of the primal and dual variables are derived by solving a scaled version of the underlying optimization problem, which aids in improving the accuracy and numerical stability of the algorithm.

1.4 Barrier iteration

After initialization, the algorithm begins to iteratively solve a series of barrier problems, whereby the barrier parameter is gradually increasing. Each barrier problem is a modified version of the original optimization problem, which consists of an additional barrier term to penalize solutions that fail to satisfy the constraints. The algorithm solves the barrier problem based on a primal-dual interior-point method, which requires solving a series of linear systems of equations.

1.5 Predictor step

At every iteration, the algorithm takes a predictor step to approximate the general direction of the solution. This includes solving a linear system of equations that approximates the constraints using the current values of the primal and dual variables. The predictor step utilizes a preconditioner that aids in improving the efficiency and numerical stability of the algorithm.

1.6 Corrector step

After the predictor step, the SeDuMi algorithm uses a corrector step to adjust the solution and improve the convergence rate. The corrector step involves solving another linear system of equations which introduces correction for the errors brought about by the predictor step. Furthermore, the corrector step updates the barrier parameter, which regulates the width of the barrier around the feasible region.

1.7 Convergence testing and termination criteria

The SeDuMi algorithm terminates when the duality gap between the primal and dual objective values is sufficiently small, below a specified tolerance level. The duality gap is a measure of the optimality of the solution and highlights the difference between the dual objective value and the primal objective value. In addition, the algorithm examines the primal and dual feasibility, and checks for the accuracy of the linear systems of equations solved at each iteration.

In the case whereby the algorithm discovers infeasibility or numerical instability, it may refine the regularization or scaling parameters, or terminate with an error message. Otherwise, if the algorithm successfully converges, it returns the optimal primal and dual variables, together with the corresponding objective values and duality gap.

2. Description of MOSEK algorithm

MOSEK is a solver for linear, conic, quadratic, and mixed-integer optimization problems. The algorithm is based on interior-point methods, which are a class of algorithms that iteratively solve a series of barrier problems that estimate the original underlying optimization problem. Like the SeDuMi algorithm, MOSEK utilizes a primal-dual interior-point method, which simultaneously updates primal and dual variables, so that the solution satisfies the optimality conditions. The primal-dual interior-point method employed by MOSEK can be summarized in four main steps: Initialization, Centering, Homogenization, and Termination.

2.1 Initialization

The MOSEK algorithm begins by initializing a feasible point and an initial value of the barrier parameter. The initial feasible point can be derived by a variety of methods, such as a warm start from a previous solution. The initial value of the barrier parameter is selected such that it is large enough to make sure that the barrier function is significantly greater (more dominant) than the objective function.

2.2 Centering

The MOSEK algorithm updates the primal and dual variables by solving a sequence of centering problems, which include solving a series of linear systems of equations that correspond to the KKT conditions of the barrier problem, as discussed in the description of the SeDuMi algorithm. The centering problems focus on directing the iterates towards the central path, which represents a curve in the primal-dual space that bridges the optimal solution of the barrier problem with the optimal solution of the original optimization problem.

2.3 Homogenization

The MOSEK algorithm refines the step size and the barrier parameter by solving a homogenization problem, which consists of solving a linear system of equations that correspond to the KKT conditions of the barrier problem with a modified right-hand side. This ensures that the iterates remain close to the central path and that the duality gap decreases across the iterations.

2.4 Termination

The MOSEK algorithm terminates when a maximum number of iterations is reached, or when the duality gap is sufficiently small, below a specified tolerance level. The duality gap is the difference between the primal and dual objective values, and it measures the optimality of the solution. The tolerance level is a parameter controlled by the user, which regulates the balance between the solution accuracy and the computational cost of the algorithm.

2.5 Advanced Features

MOSEK also incorporates several advanced features that improve the performance and the robustness of the algorithm, such as presolve, automatic parameter tuning, and mixed-integer optimization.

In the presolve stage, the MOSEK algorithm applies a series of algebraic transformations on the problem, to check for any infeasibilities or unboundedness and to simplify the problem. Therefore, this significantly decreases the size of the problem and causes the solution process to become computationally faster.

With automatic parameter tuning, the MOSEK algorithm refines its parameters based on the problem characteristics and the progress of the algorithm. As a result, this ensures that the algorithm performs generally well on a wide range of optimization problems and removes the need for manual tuning by the user.

Utilizing mixed-integer optimization, the MOSEK algorithm can solve optimization problems which consists of a combination of continuous and integer variables. The algorithm searches the solution space to find the optimal solution, based on branch-and-bound methods.

3. <u>Difference between SeDuMi and MOSEK algorithms</u>

Both SeDuMi and MOSEK utilize primal-dual interior-point methods with logarithmic barrier functions, where they approach the solution through solving a series of barrier subproblems. However, the specific details of the algorithms differ in the way they handle the various components of the optimization problem, like the objective function and the constraints. The main difference between SeDuMi and MOSEK is how they handle sparsity. SeDuMi utilizes a data structure known as a sparse matrix to describe the optimization problem, while MOSEK generally works with dense matrices.

4. <u>Comparison of performance between SeDuMi and MOSEK algorithms</u>

In this report, we investigate the numerical experimental performance of the SeDuMi and MOSEK algorithms and compare them in terms of several metrics. These metrics include computational time, sensitivity to tolerance used for terminating the algorithm, and memory requirement of the algorithms.

For this study, both the SeDuMi and MOSEK algorithms are utilized to solve a Linear Matrix Inequality (LMI) problem. Specifically, both algorithms are used to find the Lyapunov function of a given $A \epsilon R^{n \times n}$ matrix with increasing values of *n*, with some constraints of dimension *m*. In this study, the problem formulation is as follows:

We want to find a symmetric and semidefinite matrix $P \epsilon R^{n \times n}$ that satisfies the Lyapunov inequality as an LMI constraint,

$$A^T P + PA \leq -I \tag{1}$$

For simplicity of the study, the MATLAB in-built function *randi* is used to generate random integers between the range of [-3, 3] for the *A* matrix. For instance, a possible $A \epsilon R^{3 \times 3}$ matrix could be:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & -3 & -3 \end{bmatrix}$$

By randomizing the system matrix *A*, there will be instances whereby the problem becomes infeasible. Intuitively, these infeasible problems will be excluded in the discussion and findings of the results.

The objective function of the optimization problem is to minimize the trace of the solution matrix *P*. The constraints are set as inequality constraints of dimension *m*, whereby the summation of the first *m* diagonal values of the solution matrix *P* are constrained to be equal to or smaller than a chosen positive value *k*.

$$P = \begin{bmatrix} P_{11} & P_{12} & \cdots & P_{1n} \\ \vdots & P_{22} & \ldots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ P_{n1} & \ldots & \cdots & P_{nn} \end{bmatrix}$$

The *m*-dimensional constraint of the optimization problem is as follows:

$$\sum_{i=1}^{m} P_{ii} \leq k \, , where \; k > 0 \tag{2}$$

The SeDuMi and MOSEK solvers are called using the CVX interface, which is implemented as a MATLAB toolbox, and can be utilized within the MATLAB environment for solving a wide range of convex optimization problems, such as linear programming, semidefinite programming, and quadratic programming. The CVX package is designed for convex optimization, which handles SDPs and LMIs in a convenient and efficient way.

4.1 Comparison of computational time for increasing *n* with *m* fixed

In this section, we investigate the difference in computational time for increasing *n* (dimension of system matrix) with *m* (dimension of constraints) fixed, between the SeDuMi and MOSEK solvers. By fixing the dimension of constraints *m* = 2, we derive the computational time for varying dimensions of the system matrix *n* = 3, 5, 10, 30, 50, 70, 90, 100, 300, 500, 700, 900, 1000. As displayed in Equation (2), the value of *k* for this section is fixed at *k* = 2. The solver, standard and reduced tolerances of both algorithms are fixed at the default values of $\varepsilon^{1/2}$, $\varepsilon^{1/2}$, $\varepsilon^{1/4}$ respectively, where $\varepsilon = 2.22 \times 10^{-16}$ (machine precision).

The computational time is measured by the in-built CVX variable *cvx_cputime*. Since the system matrix *A* is randomized for every iteration, to obtain a more accurate result for the computational time, we take the average of 10 iterations of computational time for each *n* value.

The MATLAB code for the implementation of the SeDuMi and MOSEK algorithms through the CVX interface for varying *n* values and a fixed *m* value is in the appendix of the report. Table 1 below illustrates the average computational time taken by the SeDuMi and MOSEK algorithms, for varying values of *n*.

n	**Average computational time (s)** **(SeDuMi)**	**Average computational time (s)** **(MOSEK)**
3	0.1703125	0.1390625
5	0.1734375	0.14375
8	0.196875	0.140625
10	0.3078125	0.18125
20	0.3296875	0.209375
30	0.625	0.365625
40	1.9203125	0.6578125
50	6.2	1.653125
60	17.3890625	4.984375
70	57.3953125	24.478125
80	164.7546875	59.2390625
90	307.2578125	136.1921875
100	718.553125	271.79375

Table 1: Average computational time for varying n values

From Table 1 above, it is evident that there is a general increase in the average computational time taken for both the SeDuMi and MOSEK algorithms, as the value of *n* increases. This agrees with the theory that as the dimension of the system matrix *n* increases, the optimization problem of the Lyapunov function generally becomes more computationally expensive and takes longer to solve, due to the increased computational complexity of matrix operations as well as other issues such as numerical instability and ill-conditioning.

Furthermore, from Table 1, we also see that for each value of *n*, the average computational time taken by the SeDuMi algorithm is significantly greater than that taken by the MOSEK algorithm. For better visualization of the comparison between both algorithms, Figure 1 below illustrates the plots for the average computational time against increasing values of *n* for both algorithms. From Figure 1, it is clear that there is an exponential increase in the average computational time for both algorithms, as *n* exceeds a value of 60. In addition, while both algorithms perform almost equally in terms of computational time for problems whereby $n < 60$, it is evident that the MOSEK algorithm is computationally much faster than the SeDuMi algorithm for problems whereby $n > 60$.

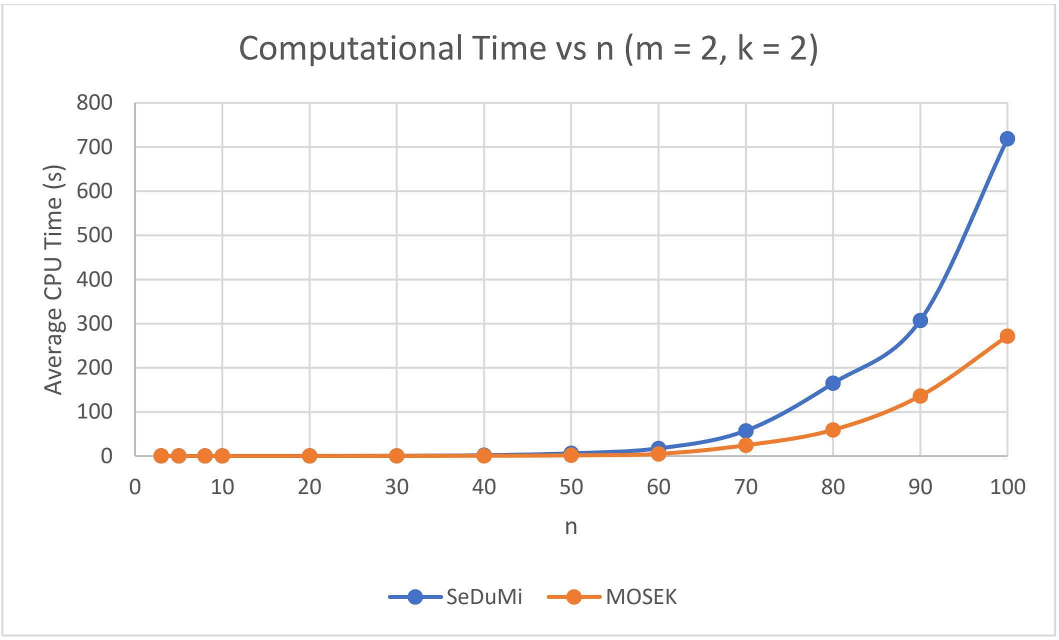

Figure 1: Plots of computational time against n for SeDuMi and MOSEK algorithms

Therefore, the results obtained agree with the theory that MOSEK algorithm is more well-suited and generally faster for solving larger-scale optimization problems, compared to the SeDuMi algorithm.

4.2 Comparison of computational time for increasing *m* with *n* fixed

In this section, we investigate the difference in computational time for increasing m (dimension of constraints) with n (dimension of system matrix) fixed, between the SeDuMi and MOSEK solvers. By fixing the dimension of system matrix $n = 50$, we derive the computational time for varying dimensions of the constraints $m = 2, 4, 6, 8, 10, 15, 20, 25, 30$. As displayed in Equation (2), the value of k for this section is fixed at $k = 3$. The solver, standard and reduced tolerances of both algorithms are fixed at the default values of $\varepsilon^{1/2}$, $\varepsilon^{1/2}$, $\varepsilon^{1/4}$ respectively, where $\varepsilon = 2.22 \times 10^{-16}$ (machine precision). Since the system matrix A is randomized for every iteration, to obtain a more accurate result for the computational time, we take the average of 10 iterations of computational time for each m value.

Table 2 below illustrates the average computational time taken by the SeDuMi and MOSEK algorithms, for varying values of m.

M	**Average computational time (s)** **(SeDuMi)**	**Average computational time (s)** **(MOSEK)**
2	5.7640625	1.5546875
4	5.81875	1.5890625
6	5.859375	1.621875
8	5.9453125	1.6359375
10	6.0890625	1.64375
15	6.1953125	1.6796875
20	6.2375	1.7421875

25	6.2390625	1.765625
30	6.3234375	1.8125

Table 2: Average computational time for varying m values

From Table 2 above, it is evident that there is a slight general increase in the average computational time taken for both the SeDuMi and MOSEK algorithms, as the value of *m* increases. This agrees with the theory that as the dimensions of the constraint increase, the optimization problem becomes more complex, and the time taken to find a solution tends to increase. This is mainly since the optimization algorithm is required to search through a larger space of possible solutions to find the optimal solution to the optimization problem. Furthermore, we also see that for each value of *m*, the average computational time taken by the SeDuMi algorithm is significantly greater than that taken by the MOSEK algorithm.

Figure 2 below illustrates the plots for the average computational time against increasing values of *m* for both algorithms.

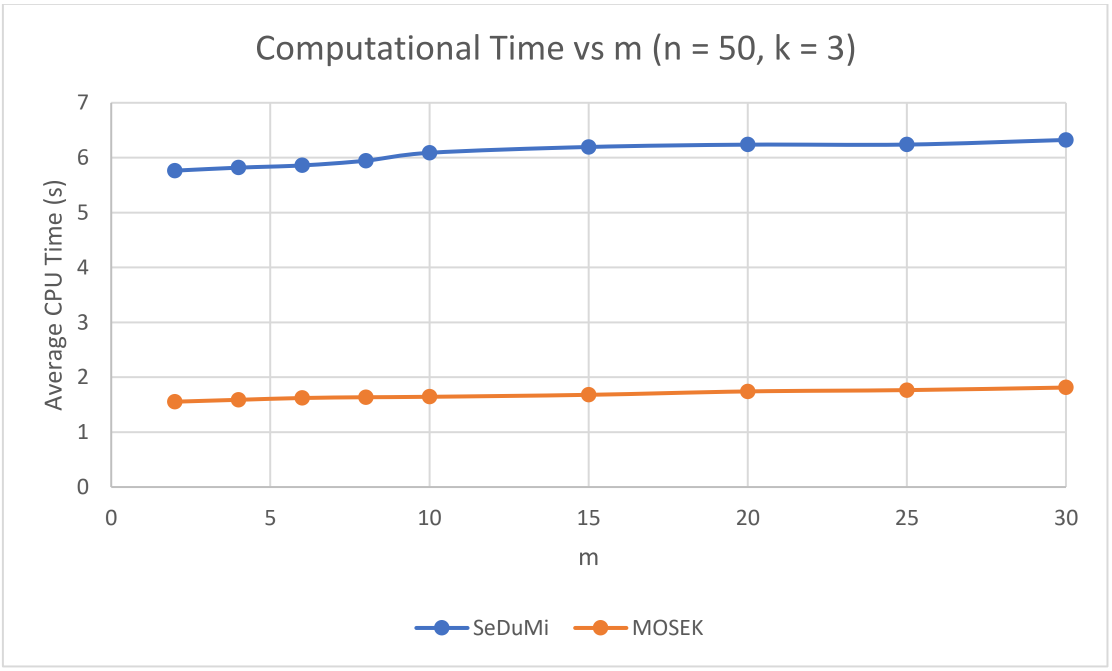

Figure 2: Plots of computational time against m for SeDuMi and MOSEK algorithms

From Figure 2, we see that the average computational time taken by the SeDuMi algorithm for each *m* value is approximately 3-4 times that taken by the MOSEK algorithm. In other words, the MOSEK algorithm is around 3-4 times faster than the SeDuMi algorithm, for the case whereby $n = 50$ and $k = 3$. Therefore, we conclude that for this specific optimization problem, the MOSEK algorithm is a significantly faster solver than the SeDuMi algorithm.

4.3 Comparison of memory usage for increasing *n* with *m* fixed

In this section, we investigate the difference in memory usage for increasing *n* (dimension of system matrix) with *m* (dimension of constraints) fixed, between the SeDuMi and MOSEK solvers. By fixing the dimension of constraints $m = 2$, we derive the computational time for varying dimensions of the system matrix $n = 3, 5, 8, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100$. As displayed in Equation (2), the value of *k* for this section is fixed at $k = 2$. The solver, standard and reduced tolerances of both algorithms are fixed at the default values of $\varepsilon^{1/2}$, $\varepsilon^{1/2}$, $\varepsilon^{1/4}$ respectively, where $\varepsilon = 2.22 \times 10^{-16}$ (machine precision).

Memory usage is measured by the MATLAB *memory* function, using a variable called *MemUsedMATLAB*, which displays information about the amount of memory being used by MATLAB. Furthermore, since the matrices *A* and *Q* are initialized before the start of the optimization process, their sizes (in bytes) are removed from the calculation of memory usage. The sizes of matrices *A* and *Q* are derived using the MATLAB function *whos*. Since the system matrix *A* is randomized for every iteration, to obtain a more accurate result for the memory usage, we take the average of 10 iterations of memory usage for each *n* value.

Table 3 below illustrates the average memory usage of the SeDuMi and MOSEK algorithms, for varying values of *n*.

n	**Average memory usage (Bytes) (SeDuMi)**	**Average memory usage (Bytes) (MOSEK)**
3	3.16E+09	3.30E+09
5	3.16E+09	3.31E+09
8	3.17E+09	3.31E+09
10	3.17E+09	3.31E+09
20	3.19E+09	3.31E+09
30	3.19E+09	3.32E+09
40	3.20E+09	3.32E+09
50	3.23E+09	3.32E+09
60	3.27E+09	3.32E+09
70	3.30E+09	3.32E+09
80	3.55E+09	3.32E+09
90	3.76E+09	3.32E+09
100	3.84E+09	3.33E+09

Table 3: Average memory usage for varying n values

One important point to note is that the method to derive the memory usage of both algorithms might not be the most accurate, because the function *MemUsedMATLAB* measures the total system memory reserved for the MATLAB process rather than the total memory usage of the process. Nevertheless, the method utilized in this section is sufficient for the purpose of comparing the difference in efficiency of the SeDuMi and MOSEK algorithms, in terms of memory usage.

From Table 3 above, it is evident that there is a general increase in the average memory usage of both the SeDuMi and MOSEK algorithms, as the value of *n* increases. This agrees with the theory that as the dimension of the system matrix *n* increases, the optimization problem generally requires more memory due to the increased number of decision variables that need to be stored in memory.

Figure 3 below illustrates the plots for the average memory usage against increasing values of *n* for both algorithms.

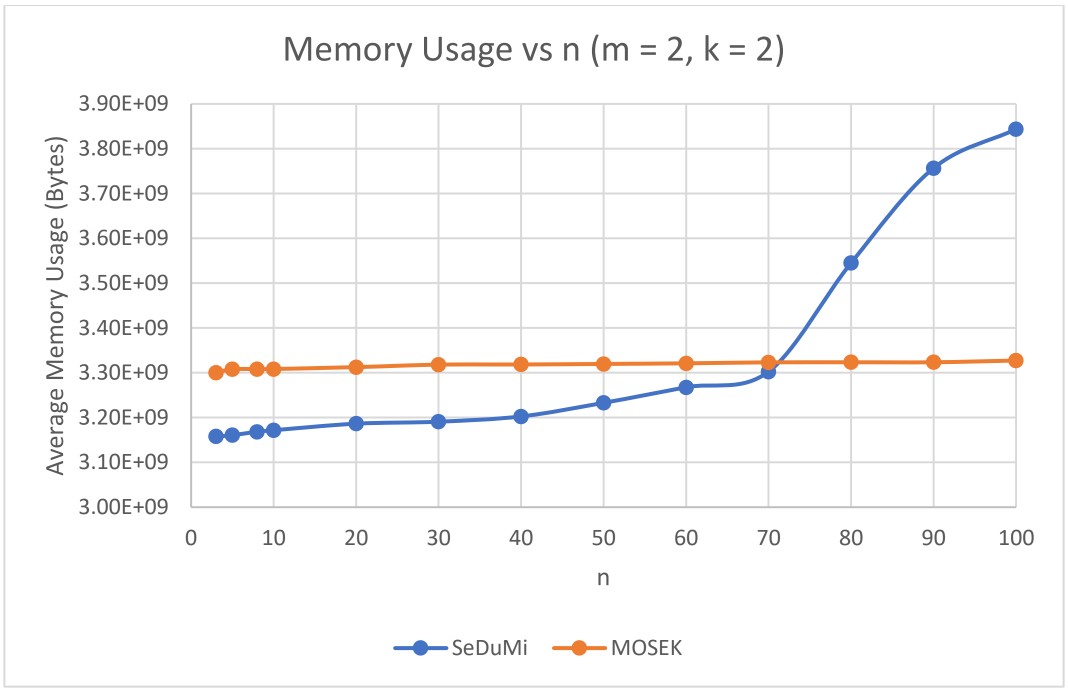

Figure 3: Plots of memory usage against n for SeDuMi and MOSEK algorithms

From Figure 3, we see that there is a significant general increase in the average memory usage of the SeDuMi algorithm as the value of *n* increases, while there is only a slight increase in the average memory usage of the MOSEK algorithm as the value of *n* increases. Furthermore, for smaller-scale optimization problems whereby $n \leq 70$, the SeDuMi algorithm consumes less memory compared to the MOSEK algorithm. However, for larger-scale optimization problems where $n > 70$, the MOSEK algorithm uses significantly less memory compared to the SeDuMi algorithm. This is in tandem with the theory that although the SeDuMi algorithm is generally more memory efficient, the MOSEK algorithm is better designed for handling larger-scale optimization problems. Therefore, we conclude that the SeDuMi algorithm is more memory efficient for small-scale optimization problems, whereas the MOSEK algorithm is more memory efficient for large-scale optimization problems.

4.4 Comparison of sensitivity to tolerance used for stopping the algorithm

In this section, we investigate the difference in sensitivity of the average computational time to the tolerance used for stopping the algorithm, for fixed *n* (dimension of system matrix) and *m* (dimension of constraints), between the SeDuMi and MOSEK solvers. By fixing the dimension of the system matrix $n = 50$, and the dimension of constraints $m = 2$, we derive the computational time for varying tolerance levels as shown in Table 4 below, where $\varepsilon = 2.22 \times 10^{-16}$ (machine precision).

Tolerance Level	**Solver Tolerance**	**Standard Tolerance**	**Reduced Tolerance**
Low	$\varepsilon^{3/8}$	$\varepsilon^{1/4}$	$\varepsilon^{1/4}$
Medium	$\varepsilon^{1/2}$	$\varepsilon^{3/8}$	$\varepsilon^{1/4}$
Default	$\varepsilon^{1/2}$	$\varepsilon^{1/2}$	$\varepsilon^{1/4}$
High	$\varepsilon^{3/4}$	$\varepsilon^{3/4}$	$\varepsilon^{3/8}$
Best	$\varepsilon^{7/8}$	$\varepsilon^{1/2}$	$\varepsilon^{1/4}$

Table 4: Solver, standard and reduced tolerance values for the different tolerance levels

As illustrated in Table 4 above, we see that each tolerance level consists of different values of solver tolerance, standard tolerance and reduced tolerance. The solver tolerance refers to the level requested of the solver. This means that the solver will stop once it achieves this level, or until no further progress is feasible. The standard tolerance refers to the level at which CVX considers the model solved to full precision. Lastly, the reduced tolerance refers to the level at which CVX deems the model to be inaccurately solved.

As displayed in Equation (2), the value of k for this section is fixed at $k = 2$. Since the system matrix A is randomized for every iteration, to obtain a more accurate result for the computational time, we take the average of 10 iterations of computational time for each tolerance value. Table 5 below illustrates the average computational time taken by the SeDuMi and MOSEK algorithms, for varying values of tolerance, for n = 50.

Tolerance	**SeDuMi**		**MOSEK**	
	Average CPU Time (s)	**Percentage Change (%)**	**Average CPU Time (s)**	**Percentage Change (%)**
Low	4.78	-22.90826613	1.14	-30.81285444
Medium	5.93	-4.435483871	1.26	-23.6294896
Default	6.2	0	1.653125	0
High	7.98	28.7046371	2.27	37.24007561
Best	7.90	27.34375	3.10	87.42911153

Table 5: Average computational time and percentage change for varying tolerance values (n = 50)

From Table 5, it is important to note that the percentage change in the average computational time is computed with respect to the default tolerance level. From the table, it is evident that there is a general increase in the average computational time taken for both the SeDuMi and MOSEK algorithms, as the tolerance level becomes higher. This agrees with the theory that as the tolerance level becomes higher, the conditions for stopping the algorithm become stricter, and hence the time taken for the solution to converge tends to increase. Figure 4 below illustrates the plots for the percentage change in computational time against increasing values of tolerance for both algorithms, for $n = 50$.

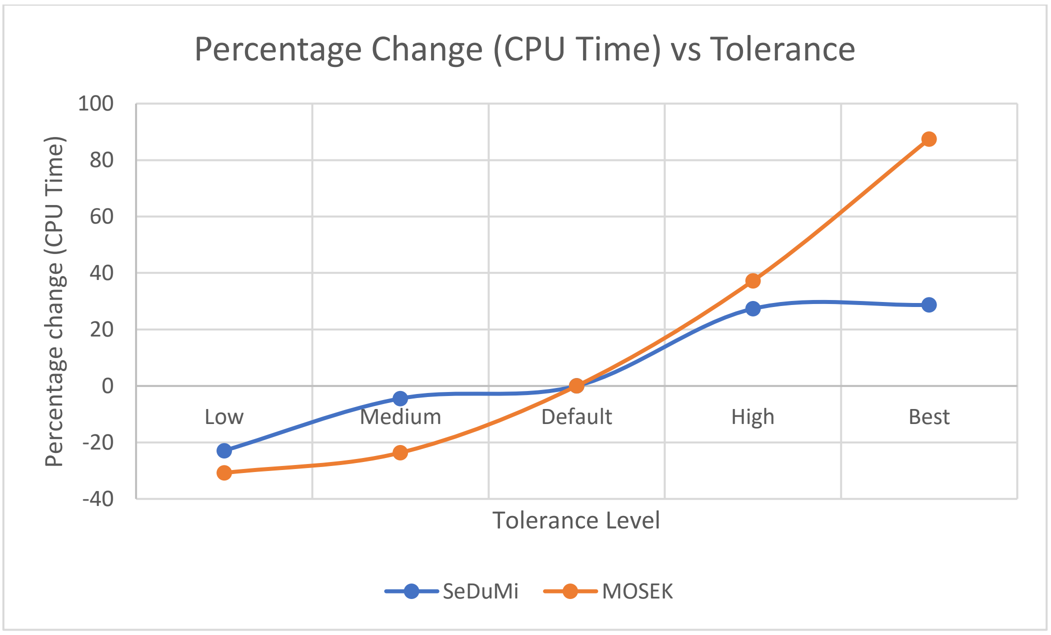

Figure 4: Plots of computational time against tolerance for SeDuMi and MOSEK algorithms

From Figure 4, comparing the blue and orange plots for n = 50, we see that the percentage change in computational time for varying tolerance values of the MOSEK algorithm is greater than that of the SeDuMi algorithm. This indicates that as the tolerance values change, there is a bigger variation in the computational performance of the MOSEK algorithm compared to the SeDuMi algorithm. Therefore, we conclude that the MOSEK algorithm is generally more sensitive to the tolerance applied, compared to the SeDuMi algorithm.

5. <u>Conclusion</u>

In this study, both the SeDuMi and MOSEK algorithms were compared based on several metrics, such as computational time for varying dimensions of system matrix n and varying dimensions of constraints m, memory usage, and sensitivity of computational time to tolerance utilized for terminating the algorithm. From the results, we find that the MOSEK algorithm is a faster algorithm compared to the SeDuMi algorithm, especially for large-scale optimization problems. In terms of memory usage, the SeDuMi algorithm outperforms the MOSEK algorithm for smaller-scale optimization problems, whereas the MOSEK algorithm outperforms the SeDuMi algorithm for larger-scale optimization problems. Lastly, through the results, we find that the MOSEK algorithm is generally more sensitive to the tolerance values used for termination, compared to the SeDuMi algorithm.

Generally, the SeDuMi algorithm has some advantages over other solvers for convex optimization problems, including its memory efficiency and ease of use. The algorithm is also well-suited for problems with several types of constraints, and can handle various types of cones, including the semi-definite cone and the second-order cone. On the other hand, MOSEK is well-known for its high performance and is frequently used for large-scale optimization problems. MOSEK generally consumes more memory than SeDuMi but can solve more complex optimization problems with faster computation times. Therefore, if memory usage is an essential factor and the problem is a convex optimization problem, the SeDuMi algorithm may be the better choice overall. However, for working with large-scale optimization problems that prioritize high performance and can handle higher memory usage, the MOSEK algorithm may be a better choice overall.

Despite investigating the differences between both algorithms, it is still essential to understand that there is no specific solver that is optimal for all optimization problems. Understanding the general performances of different solvers for various optimization problems merely serves as a guideline for users to broadly recognize which solvers they can utilize for their own use case. Experimentation and testing are still required to justify which solver performs better for a particular problem and hardware setup. Besides the SeDuMi and MOSEK solvers discussed in this study, there are other solvers that can be explored to tackle a specific optimization problem, such as DSDP and SDPT3.

DSDP (Dual Scaling algorithm for Discrete Programming) is designed to solve large-scale optimization problems and utilizes a dual-scaling algorithm to do so. SDPT3 (Semi-Definite Programming Solver version 3) is designed to efficiently solve various types of large-scale semi-definite problems, such as standard and non-standard forms, based on an interior-point method.

6. Appendix

The following code illustrates the main MATLAB code for running the various numerical experiments presented in the report. The values of the variables should be changed according to the type of experiment conducted.

6.1 MATLAB Code

```
%% Using SeDuMi/MOSEK solver for Lyapunov Equation. (Matriculation number =
A0201459J)
clear
close all
clc

% Define the problem parameters.
n = 100; % Dimension of the system matrix. (Change for sections 4.1 and 4.3)
A = randi([-3,3],n,n); % System matrix (Random).
Q = eye(n); % Given matrix.

% Define the constraint parameters.
m = 2; % Dimension of the constraints. (Change for section 4.2)
s = 0; % Counter for addition of constraints.
k = 2; % Constraint value (positive).

% Define the tolerance parameters.
e = 2.22e-16; % Machine precision.
tol = [e^0.25; e^0.5; e^(7/8)]; % Reduced tolerance, and Standard & Solver
tolerances. (Change for section 4.4)

% Solve the Lyapunov equation using SeDuMi solver with CVX.
cvx_precision (tol)
cvx_begin
    cvx_solver sedumi % Change the solver type: sedumi OR mosek.
    variable P(n,n) semidefinite symmetric
    minimize(trace(P)) % Minimize trace of solution matrix P.
    subject to
        P*A + A'*P + Q <= 0; % Lyapunov Function (LMI in variable P).
        for i = 1:m
            s = s + P(i,i);
            s <= k; % Inequality constraint.
        end
cvx_end

size_A = whos('A').bytes;
size_Q = whos('Q').bytes;
[user, sys] = memory; % Available memory after optimization.
mem_used = user.MemUsedMATLAB - (size_A + size_Q);

% Display the memory used for optimization process.
disp('Memory used (bytes) = ');
disp(mem_used);

% Display the solution.
disp('Solution P = ');
disp(P);

% Display the solver tolerance.
disp('Solver, Standard, Reduced tolerances: ')
disp(cvx_precision);
```

7. **References**

Controlling precision (no date) *Solvers - CVX Users' Guide*. Available at: http://cvxr.com/cvx/doc/solver.html (Accessed: March 27, 2023).

Labit, Y., Peaucelle, D. and Henrion, D. (no date) “SEDUMI interface 1.02: A tool for solving LMI problems with Sedumi,” *Proceedings. IEEE International Symposium on Computer Aided Control System Design* [Preprint]. Available at: https://doi.org/10.1109/cacsd.2002.1036966.

Wright, S.J. (1997) “Primal-dual interior-point methods.” Available at: https://doi.org/10.1137/1.9781611971453.