University of Cagliari

FACULTY OF ENGINEERING
Department of Electronic Engineering

Master Thesis

ALGORITHMS FOR THE STABILIZATION OF A LEADER - FOLLOWER FORMATION DESCRIBED VIA COMPLEX LAPLACIAN

Thesis advisors:
Prof. Alessandro Giua
Prof. Zhiyun Lin

Candidate:
Fabrizio Serpi

Thesis submitted in 2012

Contents

List of Figures

List of Tables

List of Algorithms

List of Matlab Files

List of Symbols

ϵ_M machine precision

$\iota = \sqrt{-1}$ imaginary unit

$|S|$ The cardinality of the set S

$\mathbb{C}^{n\times n}$ complex space of $n \times n$-dimensional matrices

$\mathbb{F}^{n\times p}$ generic field of dimension $n \times p$

$\mathbb{R}^n$ real n-dimensional vector space

$\mathcal{B}_i$ Bounding function for the i-th step in a backtracking algorithm

$\mathcal{D} = (\mathcal{V}(\mathcal{D}), \mathcal{A}(\mathcal{D}), w)$ weighted directed pseudograph

$\mathcal{D} = (\mathcal{V}, \mathcal{A})$ directed graph

$\mathcal{G} = (\mathcal{V}, \mathcal{E})$ undirected graph

$\mathcal{G} \cong \mathcal{H}$ isomorphic relation between graph $\mathcal{G}$ and $\mathcal{H}$

$\mu_D(v_i, v_j)$ number of arcs from a vertex v_i to a vertex v_j in a directed pseudograph

$\sigma(A)$ entire spectrum of a given square matrix A

$\theta(f, g)$ isomorphism of one graph onto another

$\xi \in \mathbb{C}^n$ A planar formation basis for n agents

A^* conjugate and transpose of the matrix $A \in \mathbb{C}^{(m\times n)}$

$A_i \oplus A_j$ direct sum of matrix A_i and A_j

$d_D^+(v_i)$ out-degree of a vertex v_i in a directed multigraph

$d_D^-(v_i)$ in-degree of a vertex v_i in a directed multigraph

E elementary matrix

F_ξ A formation of n agents with four degree of freedom

I_n identity matrix of order n

L_{ff} Follower-follower sub-matrix of the Laplacian matrix L

L_{lf} Leader-follower sub-matrix of a Laplacian matrix L

$N(v_i)$,N_i neighborhood of vertex v_i in a graph

$N_D^+(v_i)$ out-neighbor set of v_i in a directed pseudograph

$N_D^-(v_i)$ in-neighbor set of v_i in a directed pseudograph

$T(x_1, x_2, \ldots, x_i)$ Set of all possible values for the variable x_{i+1} such that, at the $(i+1)$th step of the backtracking algorithm, $(x_1, x_2, \ldots, x_{i+1})$ is also a path to a problem state

Abstract

A multi-agent system (MAS) or coupled cell system is a collection of individual, but **interacting**, dynamical systems (called cells or agents). With coupling, the state of certain individual systems affects the time-evolution of other agents [28]. Multi-agent systems can be used to solve problems which are difficult or impossible for an individual agent or monolithic system to solve. One way to study the interaction among coupled systems is by mean of robots playing the role of agents. In this scenario, one problem which arises is the formation control of the agents. The state of the art literature describes three different approaches to formation control. Recently, in [29] a fourth approach has been presented and fully discussed both for the single-integrator kinematics and the double-integrator dynamics case. This new approach involves complex weighted directed graphs and its Laplacian matrix on which the control laws are based. In this work the formation control via complex Laplacian is presented. In chapter 3 the full extent of this theory is presented, while in chapters 4 and 5 algorithms to implement the control laws work are discussed. Finally, in chapter 6 experiments are presented in order to show the formation control working in practice. The simulations have been written in **MATLAB®** code and they can be found in appendix.

Chapter 1

Introduction

> *"A multi-agent system (MAS) is a system composed of multiple* ***interacting*** *intelligent agents. Multi-agent systems can be used to solve problems which are difficult or impossible for an individual agent or monolithic system to solve."[2]*
>
> *"A coupled cell system is a collection of individual, but* ***interacting****, dynamical systems. With coupling, the state of certain individual systems (called cells or agents) affects the time-evolution of other agents."[28].*

The definitions of a Multi-Agent System given above, span from the most general, that is the former, to the most specific control engineering-oriented definition, that is the latter. In both cases, what stands is that the entities forming the system are related to each other in some ways; there is a sense of "together". That sense of union could be seen in different ways, depending on which discipline multi-agent modelling is applied on.

The interaction among agents play a key role to reach the common objective the overall system has. From a control engineering point of view, it means that the overall system has to evolve from a stable state to another stable state in pursuing his objective. One way to study the interaction among coupled systems is by mean of robots playing the role of agents. The common objectives of the MAS become simply the movements in the plane or in a 3-D space. In this scenario, and in general in multi-robot systems [28] there are different research problems some of which are:

- Rendezvous, consensus;
- Formation control;

- Coverage (static, dynamic);
- Cooperative target enclosing;
- Coordinated path following;
- Distributed target assignment.

In this work we are mainly interested in *formation control problems*, that is the study of the stability of a MAS in pursuing is objective, and in particular in a complex Laplacian-based approach recently introduced in [29].

Formation control (see [29]) is an important category of networked multi-agent systems due to their civil and military applications. Three main approaches to formation control have been discussed in recent literature:

the first approach describes a formation in terms of inter-agent distance measures and uses gradient control laws resulted from distance-based artificial potentials;

the second approach describes a formation in terms of inter-agent bearing measures and uses angle only control laws;

the third approach describes a formation in terms of inter-agent relative positions and uses consensus-based control laws with input bias, which are related real-valued Laplacians.

In table 1.1 a list of the main aspects of the mentioned approaches to formation control, including the complex Laplacian one, are given.

For the first approach [29], the majority of the algorithms consider representation of formations in terms of *inter-agent distance* measures. This results to be more successful when agents formation is represented by an undirected graph and together with the concept of *graph rigidity*, in which two neighboring agents work together to reach the specified distance between them. The directed case needs a further concept called *persistence* to characterize a planar formation. Still, it is challenging to synthesize a control law and analyse the stability property for a group of agents modelled by a digraph and most works are then limited to directed acyclic graphs. Angle-based control for formations in terms of inter-agents bearing measuring, that is approach number two, is relatively new and it has not been fully explored. Nonetheless, for a group of three agents global asymptotic convergence results are established to reach a triangular formation with angle-only constraints. In the third approach, formations are considered in terms of relative positions. Compared with formations

Formation Control Approaches		
	Approach	**Features**
First	Distance specified formation and gradient descent control	- non-linear; - local stability; - simple in analysis for undirected formation but challenging for directed formation; - require more relative position measurement; - do not need global information; - 3 degree of freedom (translation and rotation).
Second	Angle specified formation and angle-based control	- non-linear; - global stability but limited to special cases; - challenging in analysis for both undirected and directed formation; - do not require relative position measurement but angles; - do not need global information; - 4 degrees of freedom (translation, rotation, and scaling).
Third	Relative position specified formation and consensus - based control	- affine; - global stability; - simple in analysis for both undirected and directed formation; - require less relative position measurement; - need global information: a common sense of direction; - 1 degrees of freedom (translation only).
Fourth	Relative position specified formation and complex Laplacian based control	- linear; - global stability; - simple in analysis for both undirected and directed formation; - require intermediate relative position measurement; - do not need global information; - 4 degrees of freedom (translation, rotation, and scaling).

Table 1.1: Summary of the main formation control approaches in MAS.

described in terms of inter-agent distance constraints and inter-agent angle constraints, the third approach requires less links and it is easier to extend from undirected to directed graphs. The consensus-based control laws with input bias are affine and thus could lead to global stability results. Nonetheless, the approach has the drawback that all the agents should have a common sense of direction since input bias is defined in a common reference frame.

A fourth approach to formation control has been introduced in [29]. It is based on complex Laplacians and it exploits complex weighted 2-reachable digraphs with inter-agent distance measure together with a leader-follower organization to agents formation. The paper aimed to study the formation control problem in the plane. For a network of n interacting agents modelled as a weighted digraph, they represent a planar formation as an n-dimensional complex vector called *formation basis* and introduce a complex Laplacian of the directed graph to characterize the planar formation. The result is that the formation basis is another linearly independent eigenvector of the complex Laplacian associated with zero eigenvalues in addition to the eigenvector of ones. In this way, a planar formation is subjected to translation, rotation and scaling. In order to uniquely determine the location, orientation, and size of the group formation, they consider a leader-follower formation with two co-leaders. The result of describing the formation, that is the sensing graph of the networked agents, with a complex Laplacian, leads to a simple distributed control law. One of the advantages is that the control law is locally

implementable without requiring a common reference frame. For example, for single integrator kinematics, the velocity control of each follower agent is the complex combination of the relative positions of its neighbors using the complex weights on the incoming edges. A complex weight multiplying the relative position of a neighbor actually means that the agent moves along the line of sight rotated by an offset angle with certain speed gain (magnitude of the complex weight). This complex Laplacian based control law has also been generalized to double integrator dynamics, which has been investigated in [29] as well. The approach however, has the drawback that a few eigenvalues of the complex Laplacian might be in the left half complex plane which would lead to instability of the overall system, unlike the real Laplacian that do not manifest such problem. To tackle this technical issue, they shown that there is a way to stabilize the possibly unstable system by updating the complex weights, which is related to a traditional problem called Multiplicative Inverse Eigenvalue Problem (MIEP)(i.e., see [14]).They presented sufficient conditions for the existence of a stabilizing matrix and also provided algorithms to find it. The aim of the present work is to implement the algorithms given in [29] and to simulate multi-agent system movements like translation, rotation and scaling in the plane verifying its stability. Before presenting possible solutions to the implementation of the algorithms, we present in chapter 3 the results obtained in [29]. In chapter 2 is presented a summary of the main graph theory concepts needed to understand results in chapter 3. Chapters 4 and 5 present algorithms possible implementation and finally, in chapter 6 a complete simulation for a group of agents is shown.

Chapter 2

Background Theory

Multi-Agent Systems are studied through mathematical models like graphs and mathematical tools from linear algebra and control theory. In this chapter the necessary background to tackle the problem of MAS formation control is presented. In section 2.1 a brief introduction to permutation matrices is given. It is necessary in order to understand graphs isomorphism, especially the particular case presented in chapter 3 and 4. A glance at graph theory is fundamental and it is given in section 2.2. In 2.2.4 is presented one of the matrix used to algebraically describe a graph, the Laplacian matrix and in 2.2.5 the graph isomorphism problem is defined. The last section 2.3 is devoted to Gauss elimination process and LU factorization. This is not strictly related to graphs but we will need it in chapter 4 in order to look at the isomorphism problem on a different perspective.

2.1 Permutation Matrix

In order to define a permutation matrix, we will present identity matrices and elementary matrices first.

Identity Matrix

An *identity matrix* I_n of order n ([26],[35]), is a square $n \times n$ matrix that has exactly one non-zero entry in each row and column. In addition, these non-zero entries are the diagonal elements and each of them is valued exactly

1:

$$I = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ \vdots & & \ddots & & 0 \\ 0 & \cdots & & & 1 \end{bmatrix}. \tag{2.1}$$

There are several properties an identity matrix has, but we are mainly interested in the following ones:

1. $\det(I) = 1$,

2. given a square matrix $A \in \mathbb{C}^{n\times n}$ the left and the right product of I by A don't change matrix A, that means:

$$IA = AI = A.$$

The vector e_i defined as a column vector with only one non-zero element in the i-th position,

$$e_i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

is the i-th vector of the orthonormal basis for the real n-dimensional vector space $\mathbb{R}^n$. Noting that rows and columns of an identity matrix are the e_is vectors (transposed in the first case), we can write I in terms of them in the following ways:

$$I = \begin{bmatrix} e_1^T \\ e_2^T \\ e_3^T \\ \vdots \\ e_n^T \end{bmatrix}, \tag{2.2a}$$

$$I = \begin{bmatrix} e_1 & e_2 & e_3 & \cdots & e_n \end{bmatrix}. \tag{2.2b}$$

Elementary Matrix

An *elementary matrix* E of order n, is a square $n \times n$ matrix obtained by doing one elementary row operation to an identity matrix. Thus, there are three types, one for each different row operation:

1. a multiple of one row of I has been added to a different row. For example, the following matrix

$$E^{(v,s)}(m) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & m & & \\ & & & \ddots & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix}, \tag{2.3}$$

   is an elementary matrix where row s of I has been multiplied by m and added to row v;

2. two different rows of I have been exchanged. For example, the following matrix

$$E^{(v,s)} = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & & & 1 & & \\ & & & \ddots & & & \\ & & 1 & & & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix}, \tag{2.4}$$

   is an elementary matrix where row s of I has been exchanged with row v;

3. one row of I has been multiplied by a non-zero scalar. For example, the

following matrix

$$E^v(m) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & m & & & & \\ & & & \ddots & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix}, \tag{2.5}$$

is an elementary matrix where row v of I has been multiplied by the scalar m.

Let E be an elementary matrix obtained by doing one elementary row operation to I. The following statements are true.

1. E is invertible and its inverse is another elementary matrix of the same type.

2. If the same row operation we did on E is done to an $n \times p$ matrix $A \in \mathbb{F}^{n \times p}$, the result equals EA.

We are mainly interested in the second type of elementary matrices, the one where two rows have been exchanged. In fact, these kind of matrices are used in Gauss elimination process. Let us have an elementary matrix $E^{(v,s)}$ of the form 2.4, the following statements are always true.

1. The inverse of an elementary matrix is itself, ${E^{(v,s)}}^{-1} = E^{(v,s)}$.

2. Since the determinant of an identity matrix is unity, then $\det(E^{(v,s)}) = -1$.

3. Let us have a square matrix $A \in \mathbb{F}^{n \times n}$, the left and right product between $E^{v,s}$ and A have different results:

   (a) the left multiplication of $E^{(v,s)}$ by A, EA, exchanges row v and s in matrix A. It follows that $\det(EA) = -\det(A)$,

   (b) the right multiplication of $E^{(v,s)}$ by A, AE, exchanges column v and s in matrix A. This operation does not change the determinant, so $\det(EA) = \det(A)$.

Properties (3a) and (3b) are of great importance in Gauss elimination algorithm.

Example: let us have the elementary matrix $E^{(1,2)}$ and the real matrix A as follows:

$$E^{(1,2)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 5 & 6 & 7 & 8 \\ 8 & 7 & 6 & 5 \end{bmatrix}.$$

Thus, the left and right multiplication of $E^{(1,2)}$ by A, will result in the following matrices:

left multiplication

$$C_1 = E^{(1,2)}A = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 8 & 7 & 6 & 5 \end{bmatrix},$$

right multiplication

$$D_1 = AE^{(1,2)} = \begin{bmatrix} 2 & 1 & 3 & 4 \\ 3 & 4 & 2 & 1 \\ 6 & 8 & 7 & 8 \\ 7 & 5 & 6 & 5 \end{bmatrix},$$

where C_1 is matrix A with the first two rows exchanged, while D_1 is matrix A with the first two columns exchanged. Elementary matrices are also important because they help to define permutation matrices.

Permutation Matrix

A *permutation matrix* P can be defined as a product of elementary matrices of the form (2.4):

$$P = E^{(v_n,s_n)}E^{(v_{n-1},s_{n-1})}\dots E^{(v_1,s_1)}, \tag{2.6}$$

where the generic elementary matrix E is a square matrix of order n. A permutation matrix has only one non-null element in each row and column, valued 1. The following statements are true:

1. P is an orthogonal matrix, consequently $P^{-1} = P^T$ and $PP^T = P^TP = I$;

2. $\det(P) = (-1)^{\#(P)}$, where $\#(P)$ is the number of exchanges between rows of I to obtain P.

What a permutation matrix does, is to apply at the same time row or column exchanges which would have been performed by the consecutive multiplication of the elementary matrices it is made up of.

Example: Let us have the same matrices A and $E^{(1,2)}$ of the last example and let us take a second elementary matrix, say $E^{(3,4)}$:

$$E^{(3,4)} = \begin{bmatrix} e_1^T \\ e_2^T \\ e_4^T \\ e_3^T \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

For what we said earlier, the permutation matrix that corresponds to the sequence of left applications of the elementary matrices $E^{(1,2)}$ and $E^{(3,4)}$ is

$$P_l = E^{(3,4)}E^{(1,2)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

and the left multiplication of P_l by A is

$$P_lA = E^{(3,4)}E^{(1,2)}A = E^{(3,4)}C_1 = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 \\ 8 & 7 & 6 & 5 \\ 5 & 6 & 7 & 8 \end{bmatrix}.$$

The right application of $E^{(1,2)}$ and $E^{(3,4)}$ to a matrix, gives the permutation matrix

$$P_r = E^{(1,2)}E^{(3,4)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

and the right multiplication of P_r by A is

$$AP_r = AE^{(1,2)}E^{(3,4)} = D_1E^{(3,4)} = \begin{bmatrix} 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 6 & 8 & 8 & 7 \\ 7 & 5 & 5 & 6 \end{bmatrix}.$$

Note that, if the sequence of elementary matrices is the same in left and right multiplication, then we have:

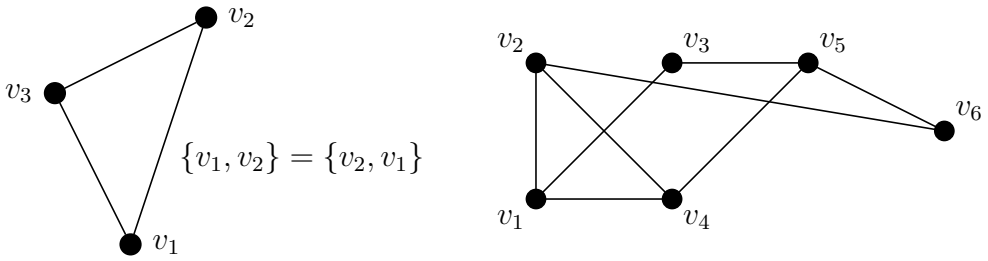
$$P_l = P_r^T \text{or} P_r = P_l^T. \tag{2.7}$$

The definition of permutation matrix given earlier, is one of the most intuitive. Moreover, it helps to understand how a permutation matrix works when multiplied by another matrix. Nonetheless, a more formal definition can be given. Denote by $\Sigma := \text{Sym}(n)$ the set of permutations of $1, 2, \ldots, n$. For $\sigma \in \Sigma$, the $(n \times n)$-*permutation matrix* P_σ is the matrix whose entries are

$$p_{ij} = \begin{cases} 1, & \text{if } j = \sigma(i), \\ 0 & \text{if } j \neq \sigma(i). \end{cases} \tag{2.8}$$

We can notice that, given an $n \times n$ matrix A, $P_\sigma A$ is obtained from A by permuting its rows in such a way that the elements $a_{\sigma(j),j}$ are on its diagonal.

In summary, a permutation matrix P of order n can be obtained from row exchanges over an identity matrix I_n. A single row exchange gives an elementary matrix while more than one give a permutation matrix. Note that the possible row combinations of an identity matrix of order n is $n!$. Hence, we will have $n!$ possible permutation matrices of the same order.



(a) (b)

Figure 2.1: Example of undirected graphs. The edges have no orientation.

2.2 Graph Theory

Graph theory (see [9], [28], [6], [7]) is one of the fundamental tools in the study of Multi-Agent Systems. In fact, graphs are models which provide a suitable representation for the interaction between agents. For this reason, in this section we will give a brief introduction to the subject, giving some of the main definitions.

2.2.1 Undirected Graphs

A *graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (*simple graph*) consists of a finite non-empty set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ of elements called *vertices* and a prescribed set $\mathcal{E}$ of *unordered* pairs of *distinct* vertices of $\mathcal{V}$ called *edges*. An edge can be written as

$$\alpha = \{v_i, v_j\} = \{v_j, v_i\},$$

where v_i and v_j are the *endpoints* of α. In fig.(2.1) two simple undirected graphs are shown. Graph (2.1a) has sets $\mathcal{V} = \{v_1, v_2, v_3\}$ and $\mathcal{E} = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}\}$; graph (2.1b) has sets $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $\mathcal{E} = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_4\}, \{v_2, v_4\}, \{v_2, v_6\}, \{v_3, v_5\}, \{v_4, v_5\}, \{v_5, v_6\}\}$. The edges have no orientation, so you can move back and forth between two connected nodes in any direction. This is highlighted in fig. (2.1a). Two vertices on the same edge or two distinct edges with a common vertex are said to be *adjacent*. Also, an edge and a vertex are *incident* with one another if the vertex is contained in the edge. A vertex is said to be *isolated* if it is incident with no edge, like v_7 in graph (2.2a). Two adjacent vertices are also said to be *neighbors*. The set of neighbors of a vertex v_i is its *neighborhood* and is denoted by $N(v_i)$ or N_i. The *degree* (*valency*) of a vertex in a graph $\mathcal{G}$ is the number of edges incident with the vertex. Since each edge of $\mathcal{G}$ has two distinct endpoints, the sum of the degrees of the vertices of $\mathcal{G}$ is twice the number of its edges. The graph $\mathcal{G}$

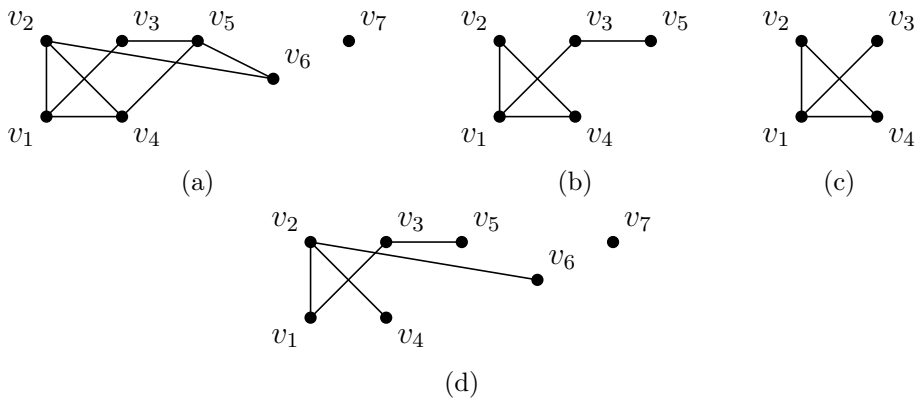


Figure 2.2: Example of subgraphs. In (2.2b), (2.2c) and (2.2d) are shown a subgraph, an induced subgraph and a spanning subgraph of graph (2.2a).

is said to be *regular* if all vertices have the same degree. If there are precisely k edges incident with each vertex of a graph, then the graph is said to be *regular of degree* k. A regular graph of degree 3 is called *cubic*. For instance, graph (2.1a) is regular of degree 2 since each vertex has degree 2. The maximum degree in a graph is often denoted by Δ.

A *complete graph* is a graph in which all possible pairs of vertices are edges. Let $\mathcal{G}$ be a graph and let $\mathcal{K}$ be the complete graph with the same vertex set $\mathcal{V}$. Then the *complement* $\bar{\mathcal{G}}$ of $\mathcal{G}$ is the graph with vertex set $\mathcal{V}$ and with edge set equal to the set of edges of $\mathcal{K}$ minus those of $\mathcal{G}$. In fig. (2.3) a self-explanatory example is shown.

A *subgraph* of a graph $\mathcal{G}$ consists of a subset $\mathcal{V}'$ of $\mathcal{V}$ and a subset $\mathcal{E}'$ of $\mathcal{E}$ that themselves form a graph. This is the most general definition. In fact, there are two main variations depending on which constraints sets $\mathcal{V}'$ and $\mathcal{E}'$ are subjected to.

- An *induced subgraph* of $\mathcal{G}$, is a subgraph $\mathcal{G}' = \mathcal{G}(\mathcal{V}')$ where set $\mathcal{E}'$ contains all edges of $\mathcal{G}$ both of whose endpoints belong to $\mathcal{V}'$.

- A *spanning subgraph* of $\mathcal{G}$, is a subgraph $\mathcal{G}'$ which has the same vertices of $\mathcal{G}$, that is, $\mathcal{V}' = \mathcal{V}$.

In fig. (2.2) those different cases are shown. Graphs (2.2b), (2.2c) and (2.2d) are a subgraph, an induced subgraph and a spanning subgraph of graph (2.2a).

The definition of a simple graph given earlier can be modified adding constraints, relaxing existing ones or both. In this way, other graphs can be defined. For example, the adjective *simple* in the earlier definition, means that two vertices can be connected by at most one edge. Relaxing this constraint,

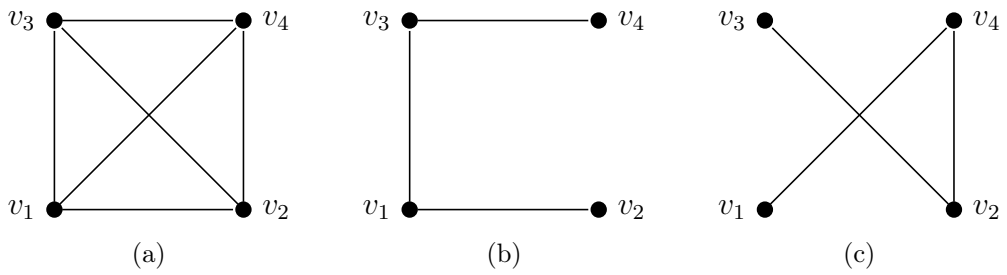


Figure 2.3: Example of a graph, its complete form and its complement. Graph (2.3b) has its complete graph in (2.3a) and its complement graph in (2.3c).

that is, allowing two vertex to be connected by more than one edge gives us the definition of a *multigraph*. We can give a brief list of further generalizations of a simple graph.

A multigraph is a simple graph where a pair of vertices is allowed to form more than one distinct edge. The edges are called *multiedges* (*multilines*) and the number of distinct edges of the form $\{v_i, v_j\}$ is called the multiplicity $m\{v_i, v_j\}$ of the edge $\{v_i, v_j\}$.

A general graph is a simple graph where multiedges and *loops* are allowed. Loops are edges of the form $\{v_i, v_i\}$ which make vertices adjacent to themselves.

A weighted graph is a simple graph where at each edge is assigned a numerical value (real or complex) called *weight*.

A directed graph (or *digraph*) is a simple graph where edges have orientation. That is, edges are an ordered pair of vertices. For example, the two edges (v_i, v_j) and (v_j, v_i) are different even though they have the same endpoints.

An infinite graph is a simple graph where the vertex set $\mathcal{V}$ is allowed to be infinite.

Let $\mathcal{G}$ be a general graph. A *walk* of *length* m is a sequence of m successively adjacent edges, and can be denoted by three different notations:

1. $\{v_1, v_2\}$,$\{v_2, v_3\}$,...,$\{v_{m-1}, v_m\}$, and $m > 0$;
2. $v_1 \to v_2 \to v_3 \to \ldots \to v_{m-1} \to v_m$;
3. v_1,v_2,v_3,...,v_{m-1},v_m.

The vertices v_1 and v_m are the *endpoints* of the walk. For example, in the graph of fig. (2.2a), $v_1, v_2, v_4, v_5, v_3, v_1, v_4$ is a walk. This is the most general definition. In fact there are several variations:

- an *open walk* is a walk with different endpoints, that is $v_1 \neq v_m$;
- a *closed walk* is a walk with equal endpoints, that is $v_1 = v_m$;
- a *trail* is a walk with distinct edges;
- a *chain* is a walk with distinct edges and distinct vertices (except, possibly, for the endpoints). If the endpoints match, it is called a *closed chain*;
- a *cycle* is a closed chain.

Notice that in a graph a cycle must contain at least 3 edges. But in a general graph a loop or a pair of multiple edges form a cycle.

A general graph $\mathcal{G}$ is *connected* if every pair of vertices v_i and v_j is joined by a walk with v_i and v_j as endpoints. Otherwise, the general graph is said to be *disconnected*. Note that a vertex is regarded as trivially connected to itself. Connectivity between vertices is reflexive, symmetric, and transitive. Hence, connectivity defines an equivalence relation on the vertices of $\mathcal{G}$ and produces a partition

$$\mathcal{V}_1 \cup \mathcal{V}_2 \cup \cdots \cup \mathcal{V}_n,$$

of the vertices of $\mathcal{G}$. The induced subgraphs $\mathcal{G}(\mathcal{V}_1), \mathcal{G}(\mathcal{V}_2), \ldots, \mathcal{G}(\mathcal{V}_n)$ of $\mathcal{G}$ formed by taking the vertices in $\mathcal{V}_i$ and the edges incident to them are called the *connected components* of $\mathcal{G}$.

Let $\mathcal{G}$ be a connected general graph. The *distance* $d(v_i, v_j)$ between v_i and v_j in $\mathcal{G}$ is the length of the shortest walk between the two vertices. A vertex is regarded as distance 0 from itself. The *diameter* of $\mathcal{G}$ is the maximum value of the distance function over all pairs of vertices. A connected general graph of diameter d has at least d+1 distinct eigenvalues in its spectrum.

A *tree* is a connected graph that contains no cycle. Let $\mathcal{T}$ be a graph of order n. Then the following statements are equivalent:

1. $\mathcal{T}$ is a tree;
2. $\mathcal{T}$ contains no cycles and has exactly $n - 1$ edges;
3. $\mathcal{T}$ is connected and has exactly $n - 1$ edges;

4. each pair of distinct vertices of $\mathcal{T}$ is joined by exactly one chain.

Any graph without a cycle is a *forest*. Note that each component of a forest is a tree.

We are mainly concerned in weighted digraphs which will be introduced later in this section. In fact, as we will see in chapter 3, they have been used in the representation of multi-agent formations and in the development of a new control theory for them.

2.2.2 Directed Graphs

Digraphs are directed analogues of graphs. Thus, as we will see later, they have many similarities. In fact, many definitions we have previously seen apply to digraphs as well, with little or no change. Nonetheless, important differences will be outlined.

A *directed graph* (or *digraph*) $\mathcal{D}$ consists of a non-empty finite set $\mathcal{V}$ of elements called *vertices* and a finite set $\mathcal{A}$ of *ordered* pairs of *distinct* vertices called *arcs* (*directed edges, directed lines*). A directed graph is often denoted by $\mathcal{D} = (\mathcal{V}, \mathcal{A})$, which means that $\mathcal{V}$ and $\mathcal{A}$ are the *vertex set* and the *arc set* of $\mathcal{D}$. The *order* (*size*) of $\mathcal{D}$ is the number of vertices (arcs) in $\mathcal{D}$ and it is often denoted by $|\mathcal{D}| = n$ ($|\mathcal{D}| = m$ respectively). We can see two examples of digraphs in fig. (2.4). In fig. (2.4a) $\mathcal{D}$ is a digraph with vertex set $\mathcal{V} = \{v_1, v_2\}$ and arc set $A = \{(v_1, v_2), (v_2, v_1)\}$. Digraph (2.4a) is quite simple but shows the main difference between undirected and directed graphs. In fact, the two arcs (v_1, v_2) and (v_2, v_1) are not the same arc as it would be if the graph were undirected. For an arc (v_i, v_j) the first vertex v_i is its *tail* and the second vertex v_j is its *head*. It can be also said that the arc (v_i, v_j) *leaves* v_i and *enters* v_j. The head and the tail of an arc are said to be its *end-vertices* (or *endpoints* as in the undirected case). The end-vertices of an arc are said to be *adjacent* vertices, i.e. v_i is adjacent to v_j and v_j is adjacent to v_i. If (v_i, v_j) is an arc, then v_i *dominates* v_j (or v_j *is dominated by* v_i) denoting it by $v_i \leftarrow v_j$.

We say that a vertex v_i is *incident* to an arc $\alpha = (v_i, v_j)$ if v_i is the head or tail of α. An arc can be denote by (v_i, v_j) or simply by $v_i v_j$. For a pair X, Y of vertex sets of a digraph $\mathcal{D}$, we define

$$(X, Y)_D = \{xy \in \mathcal{A} : x \in X, y \in Y\}, \tag{2.9}$$

i.e. $(X, Y)_D$ is the set of arcs with tail in X and head in Y. For example, for the

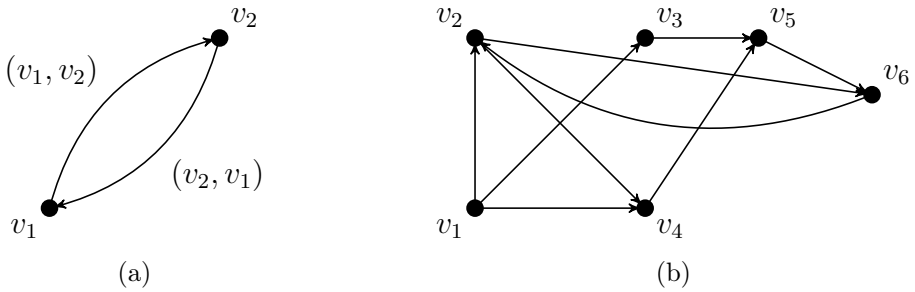


Figure 2.4: Example of digraphs. Digraph (2.4a) shows the difference between arcs with the same vertex but different orientation. Digraph (2.4b) shows a more complex digraph than (2.4a), where paths and other characteristics can be found.

digraph $\mathcal{D}$ in fig. (2.5a), choosing $(X, Y)_D$ we have that $(\{v_1, v_2\}, \{v_3, v_4\}) = \{v_1v_2, v_2v_3, v_3v_4\}$. A different choice for sets X and Y would have led to a different result. In fact, if $X = \{v_2, v_4\}$ and $Y = \{v_1, v_3\}$, then $(X, Y)_D = \{v_2v_3, v_4v_1\}$. For disjoint subsets X and Y of $\mathcal{V}$, we can use the following notation:

1. $X \to Y$, and it means that every vertex of X dominates every vertex of Y;

2. $X \Rightarrow Y$, and it means that $(Y, X) = \emptyset$;

3. $X \mapsto Y$, and it means that both $X \to Y$ and $X \Rightarrow Y$ hold.

The above definition of digraph implies that a digraph is allowed to have arcs with the same end-vertices like arcs v_iv_j and v_jv_i, but it is not allowed to contain *parallel* or *multiple* arcs, that is, pairs of arcs with the same tail and the same head, or *loops* that are arcs for which the end-vertex coincide. We can modify the definition of a digraph, as we have already done earlier for a simple graph, obtaining further generalizations. Thus, a digraph for which multiple arcs are allowed is a *directed multigraph*, while a digraph for which multiple arcs and loops are allowed is a *directed pseudograph*. In fig. (2.5) we can see an example of them. The directed multigraph (2.5b) has been obtained from the digraph (2.5a) adding multiple arcs between vertices v_1 and v_2, while the directed pseudograph (2.5c) has been obtained adding loops to multigraph (2.5b). For directed pseudographs $\mathcal{D}$, A and $(X, Y)_D$ are multisets because multiple arcs provide repeated elements. In order to denote the number of arcs from a vertex v_i to a vertex v_j in a directed pseudograph $\mathcal{D}$, the symbol $\mu_D(v_i, v_j)$ is used. In particular, $\mu_D(v_i, v_j) = 0$ means that there is no arc from v_i to v_j.

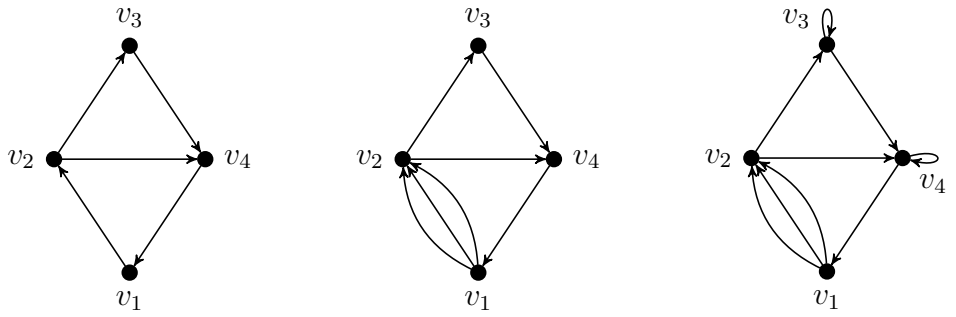


Figure 2.5: Directed multigraph and pseudograph example. In (b) we can see a directed multigraph obtained from digraph (a) adding multiple arcs. In (c), after adding loops to multigraph (b), a pseudograph has been obtained.

Definitions will be henceforth given, are of general validity. Hence, if not specified, $\mathcal{D} = (\mathcal{V}, \mathcal{A})$ will be considered a directed pseudograph. Restrictions to digraphs will be named. For a vertex v_i in $\mathcal{D}$, we can divide its adjacent vertices in different sets:

- the *out-neighborhood* (or *out-neighbor* set) $N_D^+(v_i)$ of v_i is the set of vertices that are heads of arcs whose tail is v_i,

$$N_D^+(v_i) = \{v_j \in \mathcal{V} - v_i : v_i v_j \in \mathcal{A}\}. \tag{2.10}$$

  Vertices in $N_D^+(v_i)$ are called the *out-neighbors* of v_i;

- the *in-neighborhood* (or *in-neighbor* set) $N_D^-(v_i)$ of v_i is the set of vertices that are tails of arcs whose head is v_i,

$$N_D^-(v_i) = \{v_j \in \mathcal{V} - v_i : v_j v_i \in \mathcal{A}\}. \tag{2.11}$$

  Vertices in $N_D^-(v_i)$ are called the *in-neighbors* of v_i;

- the *neighborhood* (or *neighbor* set) $N_D(v_i)$ of v_i is the set of in-neighbors and out-neighbors of v_i, that is,

$$N_D(v_i) = N_D^+(v_i) \cup N_D^-(v_i). \tag{2.12}$$

  Vertices in $N_D^-(v_i)$ are called the *neighbors* of v_i.

For example, let us take vertex v_2 of the digraph $\mathcal{D}$ in fig. (2.5a). Then, the in-neighborhood, the out-neighborhood and the neighborhood of v_2 are

$N_D^-(v_2) = \{v_1\}$, $N_D^+(v_2) = \{v_3, v_4\}$ and $N_D(v_2) = \{v_1, v_3, v_4\}$. For a set $\mathcal{V}' \subseteq \mathcal{V}$ we can define:

- the *out-degree* $d_D^+(\mathcal{V}')$ of $\mathcal{V}'$ as the number of arcs in $\mathcal{D}$ whose tails are in $\mathcal{V}'$ and heads are in $\mathcal{V} - \mathcal{V}'$,

$$d_D^+(\mathcal{V}') = |(\mathcal{V}', \mathcal{V} - \mathcal{V}')_D|; \tag{2.13}$$

- the *in-degree* $d_D^-(\mathcal{V}')$ of $\mathcal{V}'$ as the number of arcs in $\mathcal{D}$ whose heads are in $\mathcal{V}'$ and tails are in $\mathcal{V} - \mathcal{V}'$,

$$d_D^-(\mathcal{V}') = |(\mathcal{V} - \mathcal{V}', \mathcal{V}')_D|; \tag{2.14}$$

- the *semi-degree* of a set $\mathcal{V}'$ as its in-degree and out-degree;

- the *degree* $d_D(\mathcal{V}')$ of $\mathcal{V}'$ as the sum of its in-degree and out-degree,

$$d_D(\mathcal{V}') = d_D^+(\mathcal{V}') + d_D^-(\mathcal{V}'). \tag{2.15}$$

For a vertex v_i the in-degree is the number of arcs, except for loops, with head v_i, while the out-degree is the number of arcs, except for loops, with tail v_i. If $\mathcal{D}$ is a digraph (that is, it has no loops or multiple arcs), then the in-degree and the out-degree of a vertex equal the number of in-neighbors and out-neighbors of this vertex. If we want to count loops in the semi-degrees, the definitions above need to be slightly modified:

- the *in-pseudodegree* of a vertex v_i of a directed pseudograph $\mathcal{D}$ is the number of all arcs with head v_i;

- the *out-pseudodegree* of a vertex v_i of a directed pseudograph $\mathcal{D}$ is the number of all arcs with tail v_i.

The *minimum in-degree* of $\mathcal{D}$ is defined as

$$\delta^-(\mathcal{D}) = \min\{d_D^-(v_i) : v_i \in \mathcal{V}(\mathcal{D})\}, \tag{2.16}$$

while the *minimum out-degree* of $\mathcal{D}$ is defined as

$$\delta^+(\mathcal{D}) = \min\{d_D^+(v_i) : v_i \in \mathcal{V}(\mathcal{D})\}. \tag{2.17}$$

The *minimum semi-degree* of $\mathcal{D}$ is then

$$\delta^0(\mathcal{D}) = \min\{\delta^+(\mathcal{D}), \delta^-(\mathcal{D})\}. \tag{2.18}$$

Similarly, the *maximum in-degree* of $\mathcal{D}$ is defined as

$$\Delta^-(\mathcal{D}) = \max\{d_D^-(v_i) : v_i \in \mathcal{V}(\mathcal{D})\}, \tag{2.19}$$

while the *maximum out-degree* $\mathcal{D}$ is defined as

$$\Delta^+(\mathcal{D}) = \max\{d_D^+(v_i) : v_i \in \mathcal{V}(\mathcal{D})\}. \tag{2.20}$$

The *maximum semi-degree* of $\mathcal{D}$ is then

$$\Delta^0(\mathcal{D}) = \max\{\Delta^+(\mathcal{D}), \Delta^-(\mathcal{D})\}. \tag{2.21}$$

We say that $\mathcal{D}$ is *regular* if $\delta^0(\mathcal{D}) = \Delta^0(\mathcal{D})$. In this case, $\mathcal{D}$ is also called $\delta^0(\mathcal{D})$*-regular*. Since the number of arcs in a directed multigraph equals the number of their tails (or their heads), the following basic result is obtained.

Proposition 2.1. *For every directed multigraph $\mathcal{D}$,*

$$\sum_{v_i \in \mathcal{V}} d^-(v_i) = \sum_{v_i \in \mathcal{V}} d^+(v_i) = |\mathcal{A}|.$$

This proposition is also valid for directed pseudographs if in-degrees and out-degrees are replaced by in-pseudodegrees and out-pseudodegrees. Let us have digraphs $\mathcal{D}$ and $\mathcal{H}$. We can say that:

- $\mathcal{H}$ is a *subdigraph* of $\mathcal{D}$ if $\mathcal{V}(\mathcal{H}) \subseteq \mathcal{V}(\mathcal{D})$, $\mathcal{A}(\mathcal{H}) \subseteq \mathcal{A}(\mathcal{D})$ and every arc in $\mathcal{A}(\mathcal{H})$ has both end-vertices in $\mathcal{V}(\mathcal{H})$;
- $\mathcal{H}$ is a *spanning subdigraph* (or a *factor*) of $\mathcal{D}$ if $\mathcal{V}(\mathcal{H}) = \mathcal{V}(\mathcal{D})$, $\mathcal{A}(\mathcal{H}) \subseteq \mathcal{A}(\mathcal{D})$ and every arc in $\mathcal{A}(\mathcal{H})$ has both end-vertices in $\mathcal{V}(\mathcal{H})$;
- $\mathcal{H}$ is an *induced subdigraph* of $\mathcal{D}$ if every arc of $\mathcal{A}(\mathcal{D})$ with both end-vertices in $\mathcal{V}(\mathcal{H})$ is in $\mathcal{A}(\mathcal{H})$. It can be said that $\mathcal{H}$ is induced by $X = \mathcal{V}(\mathcal{H})$ and we can write $\mathcal{H} = \mathcal{D}\langle X\rangle$. If $\mathcal{H}$ is a non-induced subdigraph of $\mathcal{D}$, then there is an arc v_iv_j such that $v_i, v_j \in \mathcal{V}(\mathcal{H})$ and $v_iv_j \in \mathcal{A}(\mathcal{D}) - \mathcal{A}(\mathcal{H})$. Such an arc v_iv_j is called a *chord* of $\mathcal{H}$ in $\mathcal{D}$;
- $\mathcal{D}$ is a *superdigraph* of $\mathcal{H}$ if $\mathcal{H}$ is a subdigraph of $\mathcal{D}$.

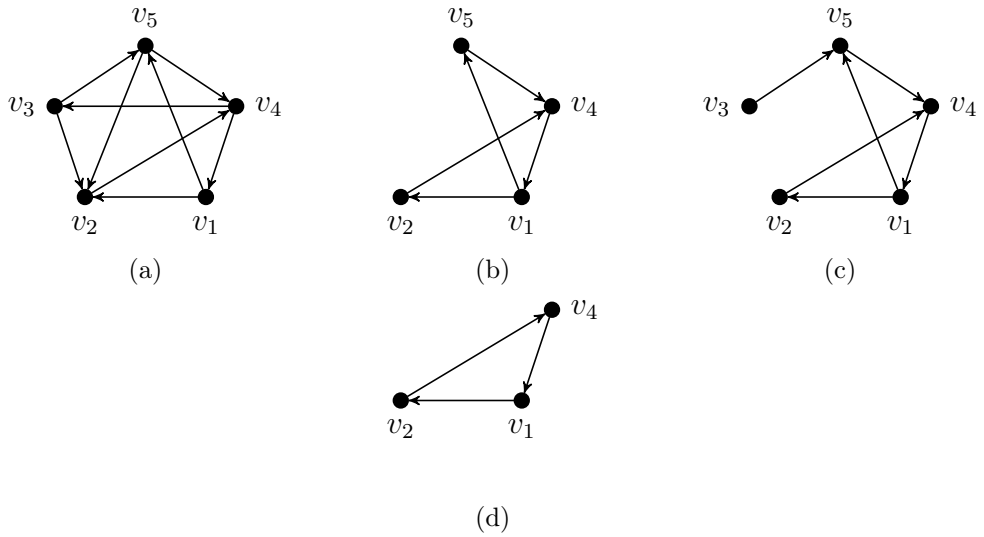


Figure 2.6: Digraph and relative subdigraphs example. In (a) is shown a digraph $\mathcal{D}$ with no multiple arcs neither loops. In (b) is shown a subdigraph of $\mathcal{D}$ and in (c) one of its spanning subdigraph. In (d) is shown an induced subdigraph of $\mathcal{D}$. Note that subdigraph (b) is also a non-induced subdigraph of $\mathcal{D}$ and arc v_5v_1 is its chord.

In fig. (2.6) we can see an example of subdigraph (2.6b), spanning subdigraph (2.6c) and induced subdigraph (2.6d). It is trivial to extend the above definitions of subdigraphs to directed pseudodigraphs. To avoid lengthy terminology, the 'parts' of directed pseudodigraph can be called just *subdigraphs*, instead of, say, directed subpseudographs.

Walks, Trails, Paths, Cycles

Let us consider a directed pseudograph $\mathcal{D}$. A *walk* (or *directed walk*) in $\mathcal{D}$ is an alternating sequence $W = v_1a_1v_2a_2v_3a_3 \dots v_{n-1}a_{n-1}v_n$ of vertices v_i and arcs a_j from $\mathcal{D}$ such that the tail of a_i is v_i and the head of a_i is v_{i+1} for every $i = 1, 2, \dots, k-1$. The set of vertices $\{v_1, v_2, \dots, v_k\}$ is denoted by $\mathcal{V}(W)$ and the set of arcs $\{a_1, a_2, \dots, a_n\}$ is denoted by $\mathcal{A}(\mathcal{W})$. We say that W is a walk *from* v_1 *to* v_k or an (v_1, v_k)*-walk*. The *length* of a walk is the number of its arcs. Hence, the aforementioned walk W has length $k-1$. A walk is *even* if its length is even, while it is *odd* if its length is odd. In general, if the arcs of W are defined from the context or simply unimportant, W will be denoted by $v_1v_2 \dots v_k$. A walk is the most general way 'to traverse a graph '. In fact, as in the case of undirected graphs, we can define variations of it:

- a *closed walk* W is a walk whose first and last vertex coincide, that is, $v_1 = v_n$;

- an *open walk* W is a walk whose first and last vertex are different, that is, $v_1 \neq v_n$. The first vertex of the walk, say v_1, is said to be the *initial* vertex of W while the last vertex, say v_n, is said to be the *terminal* vertex of W. Vertices v_1 and v_n are also said to be the *end-vertices* of W;

- a *trail* is a walk W in which all arcs are distinct. Sometimes, a trail is identified with the *directed pseudograph* $(\mathcal{V}(W), \mathcal{A}(W))$, which is a *subdigraph* of D;

- a *path* (or *directed chain*) is a walk W in which all arcs and all vertices are distinct. A path P is an $[v_i, v_j]$-*path* if P is a path between v_i and v_j, e.g. P is either a (v_i, v_j)-path or a (v_j, v_i)-path. A *longest* path in $\mathcal{D}$ is a path of maximal length in $\mathcal{D}$;

- a *cycle* (or *circuit*) is a path W whose end-vertices are equal, that is, $v_1 = v_k$. A *longest* cycle in $\mathcal{D}$ is a cycle of maximal length in $\mathcal{D}$. When W is a cycle and v_i is a vertex of W, we say that W is a cycle *through* v_i. A loop is also considered a cycle of length one. A k-*cycle* is a cycle of length k. The minimum integer k for which $\mathcal{D}$ has a k-cycle is the *girth* of $\mathcal{D}$, denoted by $g(\mathcal{D})$. If $\mathcal{D}$ does not have a cycle, we define $g(\mathcal{D}) = \infty$. If $g(\mathcal{D})$ is finite then we call a cycle of length $g(\mathcal{D})$ a *shortest cycle* in $\mathcal{D}$.

For subsets X, Y of $\mathcal{V}(\mathcal{D})$, a (v_x, v_y)-path P is a (X, Y)-path if $v_x \in X$, $v_y \in Y$ and $\mathcal{V}(P) \cap (X \cup Y) = \{v_x, v_y\}$. Note that, if $X \cap Y \neq \emptyset$ then a vertex $v_x \in X \cap Y$ forms an (X, Y)-path by itself. Subsets X and Y would even be set-vertices of subdigraphs of $\mathcal{D}$, for example $\mathcal{H}$ and $\mathcal{H}'$. Thus, an (X, Y)-path will be denoted by $(\mathcal{V}(\mathcal{H}), \mathcal{V}(\mathcal{H}'))$-path or simply by $(\mathcal{H}, \mathcal{H}')$-path. A (v_1, v_n)-path $P = v_1 v_2 \dots v_n$ is *minimal* if, for every (v_1, v_n)-path Q, either $\mathcal{V}(P) = \mathcal{V}(Q)$ or Q has a vertex not in $\mathcal{V}(P)$. Note that paths and cycles can be considered as digraphs themselves. Let $\vec{P}_n$ ($\vec{C}_n$) denote a path (a cycle) with n vertices, i.e. $\vec{P}_n = (\{1, 2, \dots, n\}, \{(1,2), (2,3), \dots, (n-1, n)\})$ and $\vec{C}_n = \vec{P}_n + (n, 1)$. A walk (path, cycle) W is a *Hamilton* or *hamiltonian* walk (path, cycle) if $\mathcal{V}(W) = \mathcal{V}(\mathcal{D})$. A digraph $\mathcal{D}$ is *hamiltonian* if $\mathcal{D}$ contains a Hamilton cycle; $\mathcal{D}$ is *traceable* if $\mathcal{D}$ possesses a Hamilton path. A trail $W = v_1 v_2 \dots v_n$ is an *Euler* or *eulerian* trail if $\mathcal{A}(W) = \mathcal{A}(\mathcal{D})$, $\mathcal{V}(W) = \mathcal{V}(\mathcal{D})$ and $v_1 = v_k$; a directed multigraph $\mathcal{D}$ is *eulerian* if it has an Euler trail.

To illustrate some definitions given so far, consider the digraph $\mathcal{D}$ in fig. (2.7). For example, a walk in $\mathcal{D}$ is $W_w = v_1 v_3 v_5 v_3 v_4 v_2 v_4 v_6 v_2 v_4$; a trail in $\mathcal{D}$ is $W_t = v_3 v_5 v_6 v_1 v_3 v_4$, also called a (v_3, v_4)-trail; a path in $\mathcal{D}$ is $P = v_3 v_5 v_6 v_2 v_1$,

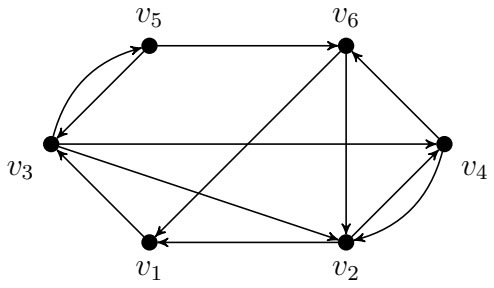


Figure 2.7: Digraph and walks example. In digraph $\mathcal{D}$ it is easy to find walks, trails, paths and cycles.

also called a (v_3, v_1)-path; a cycle in $\mathcal{D}$ is $W_c = v_3v_5v_3$, also called 2-cycle because its length is 2. Note that the girth of $\mathcal{D}$ is $g(\mathcal{D}) = 2$ too. A hamiltonian walk for $\mathcal{D}$ is $W_h = v_1v_3v_5v_6v_2v_4$, but there are neither hamiltonian path nor hamiltonian cycles. Hence, $\mathcal{D}$ is neither hamiltonian nor traceable. Moreover, $\mathcal{D}$ is not an eulerian digraph as well, because no eulerian trails can be found. Let $W = v_1v_2 \dots v_k$ and $Q = q_1q_2 \dots q_t$ be a pair of walks in a digraph $\mathcal{D}$. The walks W and Q are *disjoint* if $\mathcal{V}(W) \cap \mathcal{V}(Q) = \emptyset$ and *arc-disjoint* if $\mathcal{A}(W) \cap \mathcal{A}(Q) = \emptyset$. If W and Q are open walks, they are called *internally disjoint* if $\{v_2, v_3, \dots, v_{k-1}\} \cap \mathcal{V}(Q) = \emptyset$ and $\mathcal{V}(W) \cap \{q_2, q_3, \dots, q_{t-1}\} = \emptyset$. A path or a cycle can also be denoted by

$$W[v_i, v_j] = v_iv_{i+1}v_{i+2} \dots v_j.$$

It is easy to see that $W[v_i, v_j]$ is a path for $v_i \neq v_j$. If $1 < i < k$ then the *predecessor* of v_i on W is the vertex v_{i-1} and is also denoted by v_i^-. If $1 < i < k$, then the *successor* of v_i on W is the vertex v_{i+1} and is also denoted by v_i^+. Similarly, one can define $v_i^{++} = (v_i^+)^+$ and $v_i^{--} = (v_i^-)^-$, when these exist (which they always do if W is a cycle). Let $\mathcal{D}$ be a digraph and let $v_1, v_2, \dots, v_n$ be an ordering of its vertices. $\mathcal{D}$ is said to be *acyclic* if it has no cycle. The ordering is called an *acyclic ordering* if, for every arc v_iv_j in $\mathcal{D}$, we have $i < j$. Clearly, an acyclic ordering of $\mathcal{D}$ induces an acyclic ordering of every subdigraph $\mathcal{H}$ of $\mathcal{D}$. Since no cycle has an acyclic ordering, no digraph with a cycle has an acyclic ordering. In the other hand, every acyclic digraph has an acyclic ordering of its vertices. An *oriented* graph is a digraph with no cycle of length two. A *tournament* is an oriented graph where every pair of distinct vertices are adjacent. In other words, a digraph $\mathcal{T}$ with vertex set $\{v_1, v_2, \dots, v_n\}$ is a tournament if exactly one of the arcs v_iv_j and v_jv_i is in $\mathcal{T}$ for every $i \neq j \in \{1, 2, \dots, n\}$. In fig. (2.8) we can see an example of it.

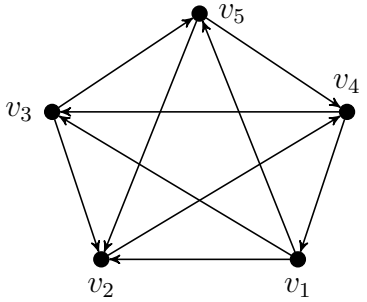


Figure 2.8: Tournament example.

Connectivity

In a digraph $\mathcal{D}$ a vertex v_j is *reachable* from a vertex v_i if $\mathcal{D}$ has an (v_i, v_j)-walk. In particular, a vertex is reachable from itself. The following proposition gives rise to a stronger condition.

Proposition 2.2. *Let $\mathcal{D}$ be a digraph and let v_i, v_j be a pair of distinct vertices in $\mathcal{D}$. If $\mathcal{D}$ has a (v_i, v_j)-walk W, then $\mathcal{D}$ contains a (v_i, v_j)-path P such that $\mathcal{A}(P) \subseteq \mathcal{A}(W)$. If $\mathcal{D}$ has a closed (v_i, v_i)-walk W, then $\mathcal{D}$ contains a cycle C through v_i such that $\mathcal{A}(C) \subseteq \mathcal{A}(W)$.*

By proposition 2.2 we can say that v_j is reachable from v_i if and only if $\mathcal{D}$ contains a (v_i, v_j)-path. Moreover, if there is still a path from one vertex v_i to one vertex v_j with the removal of any other single vertex, then vertex v_j is said to be *2-reachable* from vertex v_i. A digraph $\mathcal{D}$ is *strongly connected* (or *strong*) if, for every pair v_i, v_j of distinct vertices in $\mathcal{D}$, there exists a (v_i, v_j)-walk and a (v_j, v_i)-walk. In other words, $\mathcal{D}$ is strong if every vertex of $\mathcal{D}$ is reachable from every other vertex of $\mathcal{D}$. A digraph with one vertex is defined to be strongly connected. It is easy to see that $\mathcal{D}$ is strong if and only if it has a closed Hamilton walk. In fact, as $\vec{C}_n$ is strong, every hamiltonian digraph is strong. A digraph $\mathcal{D}$ is complete if, for every pair v_i, v_j of distinct vertices of $\mathcal{D}$, both v_iv_j and v_jv_i are in $\mathcal{D}$. For a strong digraph $\mathcal{D} = (\mathcal{V}, \mathcal{A})$, a set $\mathcal{S} \subset \mathcal{V}$ is a *separator* (or a *separating set*) if $\mathcal{D} - \mathcal{S}$ is not strong. A digraph $\mathcal{D}$ is *k-strongly connected* (or *k-strong*) if $|\mathcal{V}| \geq k+1$ and $\mathcal{D}$ has no separator with less than k vertices. It follows from the definition of strong connectivity that a complete digraph with n vertices is $(n-1)$-strong, but is not n-strong. The largest integer k such that $\mathcal{D}$ is k-strongly connected is the *vertex-strong connectivity* of $\mathcal{D}$, denoted by $k(\mathcal{D})$. If a digraph $\mathcal{D}$ is not strong, then $k(\mathcal{D}) = 0$. For a pair v_i, v_j of distinct vertices of a digraph $\mathcal{D}$, a set $\mathcal{S} \subseteq \mathcal{V}(\mathcal{D}) - \{v_i, v_j\}$ is a *(v_i, v_j)-separator* if $\mathcal{D} - \mathcal{S}$ has no (v_i, v_j)-paths. For a strong digraph

$\mathcal{D} = (\mathcal{V}, \mathcal{A})$, a set of arcs $\mathcal{W} \subseteq \mathcal{A}$ is a *cut* (or a *cut set*) if $\mathcal{D} - \mathcal{A}$ is not strong. A digraph $\mathcal{D}$ is *k-arc-strong* (or *k-arc-strongly connected*) if $\mathcal{D}$ has no cut with less than k arcs. The largest integer k such that $\mathcal{D}$ is k-arc-strongly connected is the *arc-strong connectivity* of $\mathcal{D}$, denoted by $\lambda(\mathcal{D})$. If $\mathcal{D}$ is not strong, then $\lambda(\mathcal{D} = 0)$. Note that $\lambda(\mathcal{D}) \geq k$ if and only if $d^+(\mathcal{X}), d^-(\mathcal{X}) \geq k$ for all proper subsets $\mathcal{X} \subset \mathcal{V}$. A *strong component* of a digraph $\mathcal{D}$ is a maximal induced subdigraph of $\mathcal{D}$ which is strong. If $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_t$ are the strong components of $\mathcal{D}$, then clearly $\mathcal{V}(\mathcal{D}_1) \cup \mathcal{V}(\mathcal{D}_2) \cup \cdots \cup \mathcal{V}(\mathcal{D}_t) = \mathcal{V}(\mathcal{D})$. Moreover, we must have $\mathcal{V}(\mathcal{D}_i) \cap \mathcal{V}(\mathcal{D}_j) = \emptyset$ for every $i \neq j$ as otherwise all the vertices $\mathcal{V}(\mathcal{D}_i) \cup \mathcal{V}(\mathcal{D}_j)$ are reachable from each other, implying that the vertices of $\mathcal{V}(\mathcal{D}_i) \cup \mathcal{V}(\mathcal{D}_j)$ belong to the same strong component of $\mathcal{D}$. We call $\mathcal{V}(\mathcal{D}_1) \cup \mathcal{V}(\mathcal{D}_2) \cup \cdots \cup \mathcal{V}(\mathcal{D}_t)$ the *strong decomposition* of $\mathcal{D}$. The *strong component digraph* $SC(\mathcal{D})$ of $\mathcal{D}$ is obtained by contracting strong components of $\mathcal{D}$ and deleting any parallel arc obtained in this process. In other words, if $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_t$ are the strong components of $\mathcal{D}$, then $\mathcal{V}(SC(\mathcal{D})) = \{v_1, v_2, \ldots, v_t\}$ and $\mathcal{A}(SC(\mathcal{D})) = \{v_i v_j : (\mathcal{V}(\mathcal{D}_i), \mathcal{V}(\mathcal{D}_j))_D \neq \emptyset\}$. The subdigraph of $\mathcal{D}$ induced by the vertices of a cycle in $\mathcal{D}$ is strong, i.e. is contained in a strong component of $\mathcal{D}$. Thus, $SC(\mathcal{D})$ is acyclic. The following preposition characterizes acyclic digraphs.

Proposition 2.3. *Every acyclic digraph has an acyclic ordering of its vertices.*

By preposition 2.3 the vertices of $SC(\mathcal{D})$ have an acyclic ordering. This implies that the strong components of $\mathcal{D}$ can be labelled $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_t$ such that there is no arc from $\mathcal{D}_j$ to $\mathcal{D}_i$ unless $j < i$. An ordering such that is called *acyclic ordering* of the strong components of $\mathcal{D}$. The strong components of $\mathcal{D}$ corresponding to the vertices of $SC(\mathcal{D})$ of in-degree (out-degree) zero are the *initial (terminal) strong components* of $\mathcal{D}$. The remaining strong components of $\mathcal{D}$ are called *intermediate strong components* of $\mathcal{D}$. A digraph $\mathcal{D}$ is *unilateral* if, for every pair v_i, v_j of vertices of $\mathcal{D}$, either v_i is reachable from v_j or v_j is reachable from v_i (or both). Every strong digraph, result to be unilateral. The following proposition is a characterization of unilateral digraphs.

Proposition 2.4. *A digraph $\mathcal{D}$ is unilateral if and only if there is a unique acyclic ordering $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_t$ of the strong components of $\mathcal{D}$ and $(\mathcal{V}(\mathcal{D}_i), \mathcal{V}(\mathcal{D}_{i+1})) \neq \emptyset$ for every $i = 1, 2, \ldots, t-1$.*

In fig. 2.9 is shown a digraph $\mathcal{D}$ (2.9a) that is neither a strongly connected digraph nor a complete one. Nonetheless, $\mathcal{D}$ has strong components $\mathcal{D}_1$ with $\mathcal{V}(\mathcal{D}_1) = \{v_4, v_{10}\}$, $\mathcal{D}_2$ with $\mathcal{V}(\mathcal{D}_2) = \{v_1, v_2, v_3, v_9\}$ and $\mathcal{D}_3$ with $\mathcal{V}(\mathcal{D}_3) =$

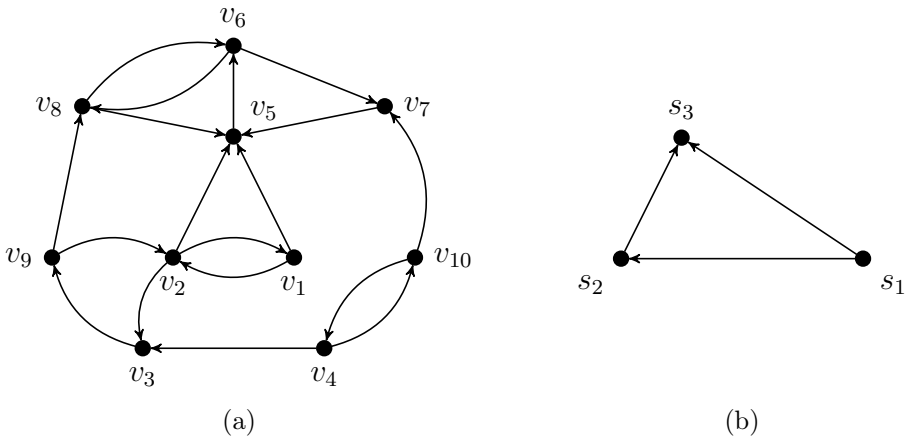


Figure 2.9: A digraph and its strong component digraph.

$\{v_5, v_6, v_7, v_8\}$. The strong component digraph $SC(\mathcal{D})$ of $\mathcal{D}$ is depicted in fig. 2.9b.

2.2.3 Weighted Directed Graphs

A *weighted directed pseudograph* is a directed pseudograph $\mathcal{D}$ along with one of the following mappings:

$$w : \mathcal{A}(\mathcal{D}) \to \mathbb{R} - \{0\}, \tag{2.22a}$$

$$w : \mathcal{A}(\mathcal{D}) \to \mathbb{C} - \{0\}. \tag{2.22b}$$

where mapping (2.22a) associates to each arc of $\mathcal{D}$ a real number while mapping (2.22b) associates to each arc of $\mathcal{D}$ a complex number. in the general case, a complex mapping will be considered. A weighted directed pseudograph can then be represented by a triple $\mathcal{D} = (\mathcal{V}(\mathcal{D}), \mathcal{A}(\mathcal{D}), w)$, and a weight associated to an arc (v_i, v_j) will be denoted by w_{ij}. Weights can be associated to vertices as well. In this case $\mathcal{D}$ is called a *vertex-weighted directed pseudograph*, i.e. a directed pseudograph $\mathcal{D}$ along with a mapping $w : \mathcal{V}(\mathcal{D}) \to \mathbb{C}$. If a is an *element* (i.e. a vertex or an arc) of a weighted directed pseudograph $\mathcal{D} = (\mathcal{V}(\mathcal{D}), \mathcal{A}(\mathcal{D}), w)$, then $w(a)$ is called the *weight* or the *cost* of a. Note that an unweighted directed pseudograph can be viewed as a vertex-weighted directed pseudograph whose elements are all of weight one. For a set $\mathcal{B}$ of arcs of a weighted directed pseudograph $\mathcal{D} = (\mathcal{V}, \mathcal{A}, w)$, we define the weight of $\mathcal{B}$ as follows:

$$w(\mathcal{B}) = \sum_{a \in \mathcal{B}} w(a). \tag{2.23}$$

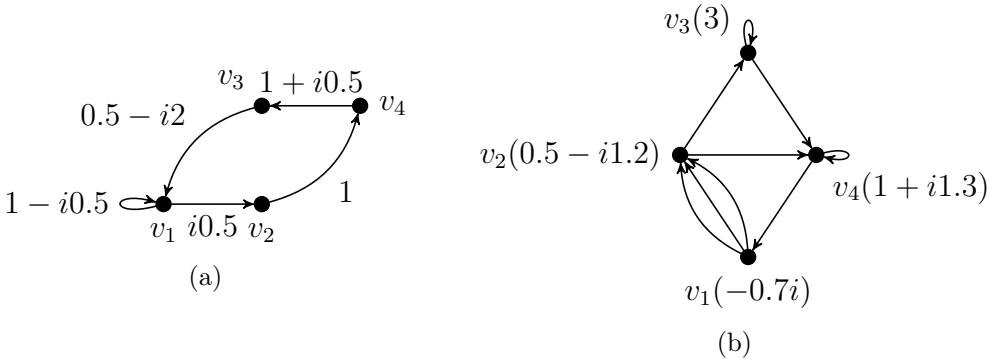


Figure 2.10: Weighted directed pseudographs. Digraph (a) is a arc-weighted directed pseudograph while digraph (b) is a vertex-weighted directed pseudodigraph. Note that the vertex weights are given in brackets.

Similarly, one can define the weight of a set of vertices in a vertex-weighted directed pseudograph. The *weight of a subdigraph* $\mathcal{H}$ of a weighted (vertex weighted) directed pseudograph $\mathcal{D}$ is the sum of the weights of the arcs (vertices) in $\mathcal{H}$.

In fig. (2.10) are shown both a arc-weighted and a vertex-weighted directed pseudographs. In digraph (2.10a) the set of arcs $\{v_1v_2, v_2v_4, v_4v_3\}$ has weight $2 + \imath 1$. In digraph (2.10b) the subdigraph $\mathcal{H} = (\{v_1, v_4, v_3\}, \{v_3v_4, v_4v_1\})$ has weight $4 + \imath 0.5$.

2.2.4 Matrices and Graphs. The Laplacian Matrix of a Graph.

Graphs can be represented by some different matrices. They are the Adjacency, the Incidence and the Laplacian matrix. Moreover, these matrices describe graphs in different ways, highlighting different properties and strictly relating them to algebra. In this section matrices related to graphs are presented for undirected, directed and weighted directed graphs. As it will be seen, matrices definition are almost the same for each case. They will only be slightly modified for each different graph.

Undirected Graphs

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a general graph of order n with vertex set

$$\mathcal{V} = \{v_1, v_2, \ldots, v_n\}.$$

Let $m(v_i, v_j)$ be the multiplicity of the edges of the form (v_i, v_j), as defined in subsec. 2.2.1 and let $a_{ij} = m(v_i, v_j)$. This means, of course, that $a_{ij} = 0$ if there are no edges of the form (v_i, v_j). Also, $m(v_i, v_i)$ equals the number of loops at vertex v_i. The *adjacency matrix* of $\mathcal{G}$ is the resulting matrix of order n such that

$$A = [a_{ij} = m(v_i, v_j)], \quad (i, j = 1, 2, \ldots, n). \tag{2.24}$$

If $\mathcal{G}$ is a *simple graph*, that is, no loops and no multiple edges are allowed, then the *adjacency matrix* is defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } v_i v_j \in \mathcal{E}, \\ 0 & \text{if } v_i v_j \notin \mathcal{E}. \end{cases} \tag{2.25}$$

Adjacency matrix A has some interesting properties which are related to the nature of the graph it describes:

- if $\mathcal{G}$ is a general graph , then
  - A is a symmetric matrix with non-negative integral elements;
  - the trace of A denotes the number of loops;
- if $\mathcal{G}$ is a multigraph, then
  - the trace of A is zero;
  - the sum of line i of A equals the degree of vertex v_i;
- if $\mathcal{G}$ is a simple graph, then
  - A is a symmetric $(0, 1)$-matrix;
  - the trace of A is zero.

The power of an adjacency matrix has interesting properties as well. Let us form

$$A^2 = \left[\sum_{t=1}^{n} a_{it} a_{tj} \right], \quad (i, j = 1, 2, \ldots, n). \tag{2.26}$$

Then eq. (2.26) implies that the element in the (i, j) position of A^2 equals the number of walks of length 2 with v_i and v_j as endpoints. In general, the element in the (i, j) position of A^k equals the number of walks of length k with v_i and v_j as endpoints. The number for closed walks appear on the main diagonal of A^k.

Let $\mathcal{G}$ be a general graph and A its adjacency matrix. The polynomial

$$f(\lambda) = \det(\lambda I - A) \tag{2.27}$$

is called the *characteristic polynomial* of $\mathcal{G}$. The collection of the n eigenvalues of A is called the *spectrum* of $\mathcal{G}$. Since A is symmetric the spectrum of $\mathcal{G}$ consists of n real numbers.

A general graph $\mathcal{G}$ is *connected* (see subsec. 2.2.1) provided that every pair of vertices v_i and v_j is joined by a walk with v_i and v_j as endpoints. Connectivity defines an equivalence relation on the vertices of $\mathcal{G}$ and yields a partition

$$\mathcal{V}_1 \cup \mathcal{V}_2 \cup \cdots \cup \mathcal{V}_n,$$

of the vertices of $\mathcal{G}$. The induced subgraphs $\mathcal{G}(\mathcal{V}_1), \mathcal{G}(\mathcal{V}_2), \ldots, \mathcal{G}(\mathcal{V}_n)$ of $\mathcal{G}$ formed by taking the vertices in $\mathcal{V}_\rangle$ and the edges incident to them are called the *connected components* of $\mathcal{G}$. Connectivity has a direct interpretation in terms of the adjacency matrix A of $\mathcal{G}$. In fact, we may simultaneously permute the lines of A so that A is transformed into a direct sum of the form

$$A_1 \oplus A_2 \oplus \cdots \oplus A_t,$$

where A_i is the adjacency matrix of the connected component $\mathcal{G}(\mathcal{V}_i)$, $(i = 1, 2, \ldots, t)$.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a general graph of order n with vertex set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ and edge set $\mathcal{E} = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$. The *incidence matrix* $B^{n\times m} = [b_{ij}]$ of G is defined by

$$b_{ij} = \begin{cases} 1 & \text{if } v_j \in \alpha_i, \\ 0 & \text{if } v_j \notin \alpha_i. \end{cases} \tag{2.28}$$

In other words, we set $b_{ij} = 1$ if vertex b_j is on edge α_i and we set $b_{ij} = 0$ otherwise. As it can be seen, B is a $(0, 1)$-matrix of size $m \times n$. This is the definition of the conventional incidence matrix in which the edges are regarded as subsets of vertices. There are some properties to highlight:

- each row of B contains at least one 1 and not more than two 1's;
- rows with a single 1 in B correspond to the edges in $\mathcal{G}$ that are loops;
- identical rows in B correspond to multiple edges in $\mathcal{G}$.

The incidence matrix and the adjacency matrix of a multigraph are related in the following way:

Theorem 2.1. *Let $\mathcal{G}$ be a multigraph of order n. Let B be the incidence matrix of $\mathcal{G}$ and let A be the adjacency matrix of $\mathcal{G}$. Then*

$$B^T B = D + A,$$

where D is a diagonal matrix of order n whose diagonal entry d_i is the degree of the vertex v_i of $\mathcal{G}$, $(i = 1, 2, \ldots, n)$.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a general graph of order n with vertex set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ and edge set $\mathcal{E} = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ as seen before..We can assign to each of the edges of $\mathcal{G}$ one of the two possible orientations and thereby transform $\mathcal{G}$ into a graph in which each of the edges of $\mathcal{G}$ is assigned a direction. Thus, we can define the *oriented incidence matrix* $B = [b_{ij}]$ of $\mathcal{G}$ as follows:

$$b_{ij} = \begin{cases} 1 & \text{if } v_j \text{ is the } \textit{initial} \text{ vertex of } \alpha_i, \\ -1 & \text{if } v_j \text{ is the } \textit{terminal} \text{ vertex of } \alpha_i, \\ 0 & \text{if } v_j \notin \alpha_i. \end{cases} \tag{2.29}$$

The oriented incidence matrix of $\mathcal{G}$ is a $(0, 1, -1)$-matrix of size $(m \times n)$. The relation between B and the adjacency matrix A is the same as in theorem 2.1. Hence, the oriented incidence matrix satisfies at the following relation:

$$B^T B = D - A. \tag{2.30}$$

The oriented incidence matrix is used to determine the number of connected components of $\mathcal{G}$ as stated from the following theorem:

Theorem 2.2. *Let $\mathcal{G}$ be a graph of order n and let t denote the number of connected components of $\mathcal{G}$. Then the oriented incidence matrix B of $\mathcal{G}$ has rank $n - t$. In fact, each matrix obtained from B by deleting t columns, one corresponding to a vertex of each component, has rank $n - t$. A submatrix B' of B of order $n - 1$ has rank $n - t$ if and only if the spanning subgraph G' of G whose edges are those corresponding to the rows of B' has t connected components.*

A consequence of theorem 2.2 is that a connected graph has an oriented incidence matrix of rank $n - 1$.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph of order n with vertex set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ and edge set $\mathcal{E} = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$. Let B be the $(m \times n)$ oriented incidence matrix of $\mathcal{G}$, and let A be the adjacency matrix of $\mathcal{G}$. The *(combinatorial) Laplacian matrix* L of $\mathcal{G}$ is the matrix of order n such that

$$L = B^T B = D - A, \tag{2.31}$$

where D is the diagonal matrix of order n whose diagonal entry d_i is the degree of the vertex v_i of $\mathcal{G}$ $(i = 1, 2, \ldots, n)$. The Laplacian matrix has some important features:

- L is a singular matrix and has rank at most equal to $n - 1$. In fact, by theorem 2.2 the matrix B has rank at most equal to $n - 1$, and hence the Laplacian matrix L as well;

- L is a positive semidefinite symmetric matrix. In fact, taking a real n-vector $x = (x_1, x_2, \ldots, x_n)^T$, we have that

$$x^T L x = x^T B^T B x = \sum_{\alpha_t = \{v_i, v_j\}} (x_i - x_j)^2 > 0, \tag{2.32}$$

  where the summation is over all m edges $\alpha_t = \{v_i, v_j\}$ of $\mathcal{G}$;

- 0 is an eigenvalue of L with corresponding eigenvector $x = (1, 1, \ldots, 1)^T$.

Directed Graphs

Let $\mathcal{D} = (\mathcal{V}, \mathcal{A})$ be a directed pseudograph of order n with vertex set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ and arc set $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. We let a_{ij} equal the multiplicity $m(a_i, a_j)$ of the arcs of the form (a_i, a_j). Then, the *adjacency matrix* A of $\mathcal{D}$ is the following matrix of order n

$$A = [a_{ij}], \quad (i, j = 1, 2, \ldots, n). \tag{2.33}$$

The entries of A are non-negative integers. But A needs no longer be symmetric. In the event that A is symmetric, then $\mathcal{D}$ is said to be a *symmetric directed pseudograph*. For a directed graph $\mathcal{D}$, without loops and multiple arcs then, since the multiplicity of an arc is 1 if the arc exist and 0 otherwise, the

adjacency matrix can be define as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in \mathcal{A}, \\ 0 & \text{if } (v_i, v_j) \notin \mathcal{A}. \end{cases} \tag{2.34}$$

It follows that, without loops, diagonal elements a_{ii} of A are null. If loops are allowed, then diagonal elements would be non null. Some interesting features are:

- the sum of row i of the adjacency matrix A is the outdegree of vertex v_i;
- the sum of column j of A is the indegree of vertex v_j;
- the assertion that $\mathcal{D}$ is regular if degree k is equivalent to the assertion that A has all of its line sums equal to k.

The adjacency matrix A can be related to many properties of a digraph. For example it is related to the connectedness of $\mathcal{D}$ by its structure. In fact, we can say that a digraph is *disconnected* if and only if its vertices can be ordered such that its adjacency matrix A can be expressed as the direct sum of two square submatrices A_1 and A_2 as follows:

$$A = A_1 \oplus A_2, \tag{2.35a}$$

$$A = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}. \tag{2.35b}$$

Such a partitioning is possible if and only if the vertices in the submatrix A_1 have no arcs going to or coming from the vertex set in A_2. Similarly, a digraph is weakly connected if and only if its vertices can be ordered such that its adjacency matrix A can be expressed in one of the following forms:

$$A = \begin{bmatrix} A_1 & A_{12} \\ 0 & A_2 \end{bmatrix}, \tag{2.36a}$$

$$A = \begin{bmatrix} A_1 & 0 \\ A_{21} & A_2 \end{bmatrix}, \tag{2.36b}$$

where A_1 and A_2 are square submatrices. Form (2.36a) represents the case when there is no directed arc going from the digraph corresponding to A_2 to the subdigraph corresponding to A_1, while form (2.36b) represents the case when there is no directed arc going from the subdigraph corresponding to A_1

to the one corresponding to A_2. If the adjacency matrix of $\mathcal{D}$ can not be represented in one of the aforementioned forms, then the digraph is said to be *strongly connected*. Let $\mathcal{D}$ be a digraph and let A be its adjacency matrix. As we have already seen for the undirected case, the power of the adjacency matrix A^t gives information about directed arc sequences between two vertices. In fact, we have the following result:

Theorem 2.3. *Let $\mathcal{D}$ be a directed graph and let A be its adjacency matrix. The entry a_{ij} in A^t equals the number of different, directed arc sequences of t arcs from vertex v_i to vertex v_j.*

These arc sequences fall in three different categories:

1. directed walks from v_i to v_j;

2. *trails* from v_i to v_j;

3. *paths* from v_i to v_j.

Unfortunately, there is no easy way of separating these different sequences from one another.

Let $\mathcal{D} = (\mathcal{V}, \mathcal{A})$ be a directed multigraph (without loops then) of order n with vertex set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ and arc set $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. The *incidence matrix* M of $\mathcal{D}$ is defined as in the case of the oriented incidence matrix of a general graph. That is, M is the matrix whose generic element is:

$$m_{ij} = \begin{cases} 1 & \text{if } v_j \text{ is the } \textit{initial} \text{ vertex of } \alpha_i, \\ -1 & \text{if } v_j \text{ is the } \textit{terminal} \text{ vertex of } \alpha_i, \\ 0 & \text{if } v_j \notin \alpha_i. \end{cases} \tag{2.37}$$

Then, the incidence matrix of $\mathcal{D}$ is a $(0, 1, -1)$-matrix of size $(n \times n)$. Note that, if we disregard the orientations of the arcs and correspondingly change -1 to 1 in M, then we obtain the incidence matrix of an undirected graph. As in the case of undirected graphs, since the sum of each column in M is zero, the rank of the incidence matrix of a digraph of n vertices is less than n. In fact, for a directed graph the following theorem holds:

Theorem 2.4. *Let $\mathcal{D}$ be a connected digraph and let M be its incidence matrix. Then, the rank of M is $n - 1$.*

Let $\mathcal{D} = (\mathcal{V}, \mathcal{A})$ be a directed graph of order n with vertex set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ and arc set $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. Let D be a diagonal matrix of order n whose diagonal entry d_i is the in-degree of vertex v_i. The *Laplacian matrix* of digraph $\mathcal{D}$ is defined as follows:

$$L = D - A, \tag{2.38}$$

where A is the adjacency matrix of $\mathcal{D}$. Thus, matrix L has entries:

$$l_{ij} = \begin{cases} d^-(v_i) & \text{if } i = j, \\ -a_{ij} & \text{if } i \neq j. \end{cases} \tag{2.39}$$

Since the sum of the entries in each column of L is equal to zero, then the n rows are linearly dependent and $\det(L) = 0$. As done for the undirected case, we can represent the Laplacian matrix of a digraph by its incidence matrix as well. Thus, we can also define L as follows:

$$L = M^T M. \tag{2.40}$$

Weighted Directed Graphs

Definitions we have already seen for the directed graphs can be extended for the weighted digraphs with some changes. In order to define the Laplacian matrix for the weighted case, the adjacency matrix and the degree matrix will be presented first.

Let $\mathcal{D} = (\mathcal{V}, \mathcal{A}, w)$ be a weighted directed graph with vertex set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, arc set $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ and a complex mapping $w : \mathcal{A}(\mathcal{D}) \to \mathbb{C} - \{0\}$; the complex mapping associates a complex number w_{ij} to each arc (j, i) of the graph. The *adjacency matrix* A of the weighted digraph $\mathcal{D}$ is defined as follows:

$$a_{ij} = \begin{cases} w_{ij} & \text{if } (j,i) \in \mathcal{A}, \\ 0 & \text{if } (j,i) \notin \mathcal{A}. \end{cases} \tag{2.41}$$

The *degree matrix* D of a weighted digraph is defined as the diagonal matrix whose diagonal elements d_{ii} are the sum of the weights belonging to the arcs which have vertex v_i as their head. We can simply write:

$$d_{ii} = \sum_{j \in N_i^-} w_{ij}. \tag{2.42}$$

The *Laplacian matrix* of $\mathcal{D}$ can finally be defined as the difference between the degree matrix and the adjacency matrix as defined above:

$$L = D - A. \tag{2.43}$$

We can write the Laplacian matrix in a more suitable form, defining its elements as follows:

$$l_{ij} = \begin{cases} -w_{ij} & \text{if } i \neq j \text{ and } j \in N_i^-, \\ 0 & \text{if } i \neq j \text{ and } j \notin N_i^-, \\ \sum_{j \in N_i^-} w_{ij} & \text{if } i = j. \end{cases} \tag{2.44}$$

The Laplacian matrix of a weighted digraph will be of particular interest in chapter 3 where a new formation control approach will be surveyed.

Example

Let us consider the graphs in fig. (2.11). We want to write the matrices associated to each one.

In fig. (2.11a) is shown a simple (undirected) graph $\mathcal{G}$ with vertex set $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$ and edge set $\mathcal{E} = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_4, v_1\}\}$. Graph $\mathcal{G}$ has associated the following matrices:

1. adjacency matrix,
$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix};$$

2. *incidence matrix*, using the edge sequence $\{\{v_1, v_2\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_4, v_1\}\}$ for columns from 1 to 5,
$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix};$$

3. *oriented incidence matrix*, using arcs $\{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4),$

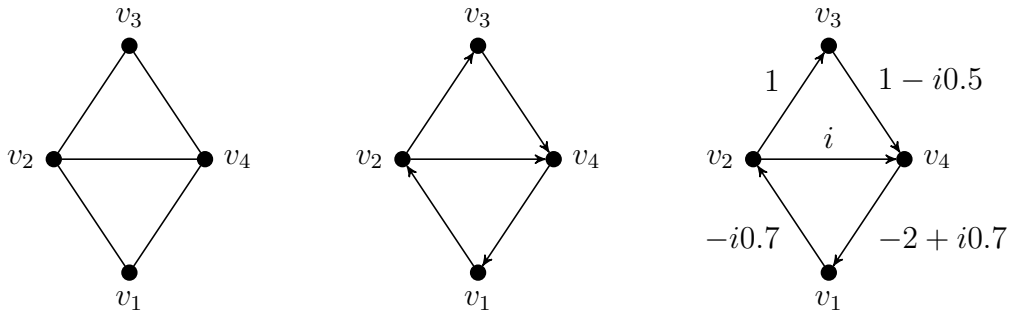


Figure 2.11: In figure are shown different versions of the same graph. In (a) vertices and edges forms an undirected graph. In (b) the same vertices in (a) and an oriented version of the edges give a digraph. In (c) it is shown a weighted digraph obtained adding a complex mapping to the arc set of digraph (b).

$(v_4, v_1)\}$ for columns from 1 to 5,

$$B_o = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ -1 & 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & -1 & 1 \end{bmatrix};$$

4. *degree matrix*,

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix};$$

5. *Laplacian matrix*,

$$L = B_o^T B_o = D - A = \begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 3 & -1 & -1 \\ 0 & -1 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}.$$

In fig. (2.11b) is shown a digraph $\mathcal{D}$ with vertex set $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$ and arc set $\mathcal{A} = \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_1)\}$. Digraph $\mathcal{D}$ is nothing more than the oriented version of graph $\mathcal{G}$. The following matrices belongs to digraph $\mathcal{D}$:

1. *adjacency matrix*,

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

where it can be seen that A no longer needs to be symmetric;

2. *incidence matrix*, using arcs $\{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_1)\}$ for columns from 1 to 5,

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ -1 & 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & -1 & 1 \end{bmatrix}.$$

Note that M equals the oriented incidence matrix of the undirected case since the same arcs orientation have been used;

3. *degree matrix*, where in the directed case the diagonal elements are the in-degree of the vertices

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix};$$

4. *Laplacian matrix*,

$$L = D - A = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 2 \end{bmatrix}.$$

Finally, in fig.(2.11c) is shown a weighted digraph $\mathcal{H}$ that is digraph $\mathcal{D}$ along with a complex mapping over its arc set. Weighted digraph $\mathcal{H}$ has associated the following matrices:

1. *adjacency matrix*,

$$A = \begin{bmatrix} 0 & 0 & 0 & -2+i0.7 \\ -i0.7 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & i & 1-i0.5 & 0 \end{bmatrix};$$

2. *degree matrix*,

$$D = \begin{bmatrix} -2+i0.7 & 0 & 0 & 0 \\ 0 & -i0.7 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1+i0.5 \end{bmatrix};$$

3. *Laplacian matrix*,

$$L = D - A = \begin{bmatrix} -2+i0.7 & 0 & 0 & 2-i0.7 \\ i0.7 & -i0.7 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -i & -1+i0.5 & 1+i0.5 \end{bmatrix}.$$

2.2.5 Graphs Isomorphism

Let $\mathcal{G}$ and $\mathcal{H}$ be graphs. An *isomorphism* from $\mathcal{G}$ to $\mathcal{H}$ is a pair of *bijections* $f : \mathcal{V}(\mathcal{G}) \to \mathcal{V}(\mathcal{H})$ and $g : \mathcal{E}(\mathcal{G}) \to \mathcal{E}(\mathcal{H})$ such that each edge $\{v_i, v_j\} \in \mathcal{E}(\mathcal{G})$ is mapped to an edge $\{f(v_i), f(v_j)\} \in \mathcal{E}(\mathcal{H})$. If there is an isomorphic relation from $\mathcal{G}$ to $\mathcal{H}$ it is said that $\mathcal{G}$ is *isomorphic to* $\mathcal{H}$ and it is denoted by

$$\mathcal{G} \cong \mathcal{H}, \tag{2.45}$$

and the pair of ordered mappings can be denoted by

$$\theta(f, g). \tag{2.46}$$

Clearly, if the two graphs $\mathcal{G}$ and $\mathcal{H}$ are isomorphic we have that

$$|\mathcal{V}(\mathcal{G})| = |\mathcal{V}(\mathcal{H})| \text{and} |\mathcal{E}(\mathcal{G})| = |\mathcal{E}(\mathcal{H})|. \tag{2.47}$$

In the case of simple graphs, the definition of isomorphism can be stated more concisely, because if $\theta(f, g)$ is an isomorphism between two simple graphs, say

$\mathcal{G}$ and $\mathcal{H}$, the mapping g is completely determined by f. Thus, we can say that an *isomorphism* from $\mathcal{G}$ to $\mathcal{H}$ is a *bijection* $f : \mathcal{V}(\mathcal{G}) \to \mathcal{V}(\mathcal{H})$ which preserves adjacency. In general, we can say that, an isomorphism preserves:

- *adjacency between vertices*, that is, vertices v_i and v_j are adjacent in $\mathcal{G}$ if and only if $f(v_i)$ and $f(v_j)$ are adjacent in $\mathcal{H}$;
- *incidence between vertices and edges*, that is, vertex v is incident with edge α in $\mathcal{G}$ if and only if vertex $f(v)$ is incident with edge $g(\alpha)$ in $\mathcal{H}$.

Note that from what stated so far, if two graphs are isomorphic, then they are either identical or differ just in the names of their vertices and edges. Thus they have the same structure (diagram). Let $\mathcal{G}$ and $\mathcal{H}$ be (simple) graphs. The *isomorphism relation*[1] consisting of the set of ordered pairs $(\mathcal{G}, \mathcal{H})$ such that $\mathcal{G}$ is isomorphic to $\mathcal{H}$, has the reflexive, symmetrical and transitive property.

The definition of isomorphism its the same both for undirected and directed case. Nonetheless, definition for the directed case will be given for the sake of completeness. Let $\mathcal{D}$ and $\mathcal{H}$ be (unweighted) directed pseudographs. It can be said that $\mathcal{D}$ and $\mathcal{H}$ are isomorphic if there exists a bijection $f : \mathcal{V}(\mathcal{D}) \to \mathcal{V}(\mathcal{H})$ such that $\mu_D(v_i, v_j) = \mu_H(f(v_i), f(v_j))$ for every *ordered pair* v_i, v_j of vertices in $\mathcal{D}$. As we can see, preservation of adjacency is the key point of isomorphism, even though in the directed case arcs orientation too has to be preserved. Note that, in case we do want to distinguish between isomorphic graphs (digraphs), we speak of *labelled (digraphs) graphs*. In this case, a pair of (digraphs) graphs $\mathcal{Q}$ and $\mathcal{H}$ is indistinguishable if and only if they completely coincide, that is, $\mathcal{V}(\mathcal{Q}) = \mathcal{V}(\mathcal{H})$ and $\mathcal{E}(\mathcal{Q}) = \mathcal{E}(\mathcal{Q})$ $(\mathcal{A}(\mathcal{Q}) = \mathcal{A}(\mathcal{Q}))$.

An *automorphism* between to graphs $\mathcal{Q}$ and $\mathcal{H}$ is an isomorphism of $\mathcal{Q}$ onto itself, that is, $\mathcal{Q}$ and $\mathcal{H}$ are the same graph or *identical*. An automorphism is then nothing less than a pair of identity relations $I_v : \mathcal{V}(\mathcal{Q}) \to \mathcal{V}(\mathcal{Q})$ and $I_e : \mathcal{E}(\mathcal{Q}) \to \mathcal{E}(\mathcal{Q})$ such that for each vertex v and each edge α in $\mathcal{Q}$ we have $I(v) = v$ and $I_e(\alpha) = \alpha$. The same definition applies to directed graphs as well.

Let $\mathcal{Q}$ and $\mathcal{H}$ be simple graphs or digraphs of order n. Let $Sigma$ be the set of all permutations of the first n positive integers and let $\sigma \in \Sigma$. One way to find if there is an isomorphism relation between $\mathcal{Q}$ and $\mathcal{H}$, is to find an order of the vertices of one of the graphs such that its adjacency matrix equals the adjacency matrix of the other. In other words, it has to be found a

[1]Note that a *relation* on a set W is a collection of ordered pairs from W. An *equivalence relation* is a relation that is reflexive, symmetric, and transitive.

permutation σ such that P_σ verifies the following relation:

$$A_{\mathcal{H}} = P_\sigma A_{\mathcal{Q}} P_\sigma^T. \tag{2.48}$$

Equation (2.48) holds even if the adjacency matrix is substituted with the Laplacian matrix:

$$L_{\mathcal{H}} = P_\sigma L_{\mathcal{Q}} P_\sigma^T. \tag{2.49}$$

Consequently, in order to find an automorphism of a graph or digraph, say $\mathcal{H}$, it has to be found a permutation σ of the vertices such that the adjacency matrix (the Laplacian matrix) will be transformed onto itself. In other words, it has to be found a permutation σ such that P_σ verifies the following relations:

$$A_{\mathcal{H}} = P_\sigma A_{\mathcal{H}} P_\sigma^T. \tag{2.50}$$

$$L_{\mathcal{H}} = P_\sigma L_{\mathcal{H}} P_\sigma^T. \tag{2.51}$$

Note that the transpose of a permutation matrix equals its inverse. Hence, isomorphism (automorphism) relations holds if we substitute P^T with P^{-1}. For example, given Q and H it can be written:

$$A_{\mathcal{H}} = P_\sigma A_{\mathcal{Q}} P_\sigma^T. \tag{2.52}$$

Equation (2.52) is the well known similarity relation between matrices which tells us, in this case, that isomorphic graphs have similar adjacency (Laplacian) matrix. Consequently, an isomorphism relation preserves the spectrum of a graph.

Example

Let $\mathcal{G}$ and $\mathcal{H}$ be the digraphs in fig. (2.12a) and (2.12b) respectively. We want to know if they are isomorphic. Then, we need to find a permutation σ of the vertices such that, adjacency matrices

$$A_{\mathcal{G}} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix},$$

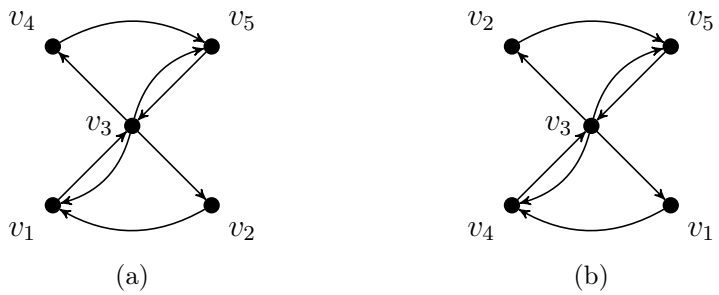


Figure 2.12: Graphs isomorphism example. In figure are depicted two isomorphic digraphs. They are drawn the same way in order to highlight the nodes relabelling operation due to the search for isomorphism.

and

$$A_{\mathcal{H}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix},$$

transform one into another. Note that digraphs are drawn in a similar way to make clear how nodes relabelling works when searching for an isomorphism relation. In fact, looking carefully at the picture we can see that, a permutation of nodes $\sigma = (4, 1, 3, 2, 5)$ in $\mathcal{G}$ will lead to digraph $\mathcal{H}$. In fact, as we have graphically verified, the permutation matrix P_σ verifies the isomorphism relation $A_{\mathcal{H}} = P_\sigma A_{\mathcal{G}} P_\sigma^T$.

2.3 Gauss Elimination and the LU Factorization

Gauss Elimination (see [35],[26],[22] and [8]) is a technique widely used to solve systems of linear equations of the form

$$Ax = b. \tag{2.53}$$

In fact, the Gauss method consists in reducing matrix A in a row echelon form, that is an upper triangular matrix, and solve the system with a back substitution of the unknowns. This is possible because a matrix is equivalent to its row echelon forms that can be used in the linear system instead of the original one. In fact, a row echelon form of A can be obtained by a sequence of elementary row operations that transform A in its REF and that ensure the equivalence between the two matrices. To solve a linear system by Gauss

elimination not only is far easier than solve it with classical methods, but also supplies an algorithm that can be exploited in a software environment. Let us see what is a row echelon form of a matrix first.

Let us have a matrix $A \in \mathbb{F}^{(m\times n)}$. The matrix A is said to be in *row echelon form* (REF) when the following two conditions are met (see [26]):

1. Any zero rows are below all non-zero rows.

2. For each non-zero row i, $i \leq m-1$, either row $i+1$ is zero or the leading entry[2] of row $i+1$ is in a column to the right of the column of the leading entry in row i.

Consequently, the echelon form of a matrix has the following general structure,

$$A = \begin{bmatrix} (*) & * & * & * & * & * & * & * \\ 0 & 0 & (*) & * & * & * & * & * \\ 0 & 0 & 0 & (*) & * & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & (*) & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{2.54}$$

where the non-null elements in brackets are called *pivot elements*. The matrix A is in *reduced row echelon form* (RREF) if it is in row echelon form and the following third condition is also met:

3. If a_{ik} is the leading entry in row i, then $a_{ik} = 1$, and every entry of column k other than a_{ik} is zero.

If A is in row echelon form, then

- the *pivot positions* are the positions of the leading entries in its non-zero rows;

- the *pivots* are the leading entries in its non-zero rows;

- the *pivot column (pivot row)* is a column (row) that contains a pivot position.

As said before, in order to reduce a matrix in its row echelon form we need to do some elementary row operations on it. *Elementary row operations* on a matrix are operations of the following types:

[2]When a row of A is not zero, its first non-zero entry is the *leading entry* of the row.

1. Add a multiple of one row to a different row.

2. Exchange two different rows.

3. Multiply one row by a non-zero scalar α.

Moreover, they are the basis for the Gaussian Elimination Algorithm.

Let $A \in \mathbb{F}^{(n\times n)}$ be a square matrix. The Gauss Elimination process, that is, the reduction of A in one of its row echelon forms, consists of $n-1$ steps in each of them the under-diagonal elements of a column are transformed in zero elements by the use of elementary operations. For example, let us consider the first step of the algorithm. Matrix A at the first step is denoted by $A^{(1)}$ where the superscript denotes the step number:

$$A = A^{(1)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{n1}^{(1)} & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} \end{bmatrix}.$$

We want to reduce matrix $A^{(1)}$ such that, matrix $A^{(2)}$ will be of the form:

$$A^{(2)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & & a_{2n}^{(2)} \\ \vdots & & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{bmatrix}.$$

This can be done, if $a_{11} \neq 0$, by multiplying the first row by scalars m_{i1} and subtracting the result from the other rows. That is,

$$(\text{row}i)^{(2)} = (\text{row}i)^{(1)} - m_{i1}(\text{row}1)^{(1)} \quad i = 2, \ldots, n. \tag{2.55}$$

In order to obtain the desired effect, constants m_{i1} are chosen to be $m_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}$ (that is the reason why $a_{11} \neq 0$). More precisely, the first step of the Gaussian Elimination can be written as follows:

for $i = 2, \ldots, n$ **do**
  $m_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}$
  for $j = 2, \ldots, n$ **do**
    $a_{ij}^{(2)} = a_{ij}^{(1)} - m_{i1}a_{1j}^{(1)}$
  end for
  $b_i^{(2)} = b_i^{(1)} - m_{i1}b_1^{(1)}$,

end for.

where vector b is the vector of the constant terms in the non homogeneous linear system $Ax = b$. The same strategy is applied to each column of A so that an upper triangular form (echelon form) can be reached. Hence, at the k-th step, we have both the matrix $A^{(k)}$ and the vector $b^{(k)}$ as:

$$A^{(k)} = \begin{bmatrix} a_{11}^{(1)} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & & & & & \vdots \\ \vdots & \ddots & a_{k-1,k-1}^{(k-1)} & & & & a_{k-1,n}^{(k-1)} \\ \vdots & & 0 & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & \vdots & a_{k+1,k}^{(k)} & a_{k+1,k+1}^{(k)} & \cdots & a_{k+1,n}^{(k)} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{nk}^{(k)} & a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}, \tag{2.56}$$

and

$$b^{(k)} = \begin{bmatrix} b_1^1 \\ \vdots \\ b_{k-1}^{(k-1)} \\ b_k^{(k)} \\ b_{k+1}^{(k)} \\ \vdots \\ \vdots \\ b_n^{(k)} \end{bmatrix}. \tag{2.57}$$

What we want now, is to make equal zero the elements of the k-th column from row $k+1$ to row n. To do that, we simply do what we have done in the first step. At the end of the k-th step, matrix $A^{(k+1)}$ and vector $b^{(k+1)}$ are:

$$A^{(k+1)} = \begin{bmatrix} a_{11}^{(1)} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & & & & & \vdots \\ \vdots & \ddots & a_{k-1,k-1}^{(k-1)} & & & & a_{k-1,n}^{(k-1)} \\ \vdots & & 0 & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & \vdots & 0 & a_{k+1,k+1}^{(k+1)} & \cdots & a_{k+1,n}^{(k+1)} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & a_{n,k+1}^{(k+1)} & \cdots & a_{nn}^{(k+1)} \end{bmatrix}, \tag{2.58}$$

and

$$b^{(k)} = \begin{bmatrix} b_1^1 \\ \vdots \\ b_{k-1}^{(k-1)} \\ b_k^{(k)} \\ b_{k+1}^{(k+1)} \\ \vdots \\ \vdots \\ b_n^{(k+1)} \end{bmatrix}. \tag{2.59}$$

Elements of $A^{(k+1)}$ and $b^{(k+1)}$ are defined as follows:

$$a_{ij}^{(k+1)} = \begin{cases} a_{ij}^{(k)}, & i = 1, \ldots, k, \text{ or } j = 1, \ldots, k-1, \\ 0 & i = k, i = k+1, \ldots, n, \\ a_{ij}^{(k)} - m_{ij} a_{ij}^{(k)} & i, j = k+1, \ldots, n, \end{cases} \tag{2.60}$$

and

$$b_i^{(k+1)} = \begin{cases} b_i^{(k)}, & i = 1, \ldots, k, \\ b_i^k - m_{ik} b_k^{(k)}, & i = k+1, \ldots, n, \end{cases} \tag{2.61}$$

where constants m_{ik} are called *multipliers*

$$m_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}. \tag{2.62}$$

The element $a_{kk}^{(k)}$ is called the *pivot element* at step k and the algorithm can go to the next step only if the pivot element at the current step is a non-zero element. The number of required steps is $n-1$ and at the end of the $n-1$-th step, the algorithm supplies an upper triangular matrix $U = A^{(n)}$. Gauss elimination, shown in algorithm 2.1, requires $\mathrm{O}(\frac{1}{3}n^3)$ multiplications and the same number of additions. The main problem of this implementation is that the non-singularity of the matrix being transformed does not ensure that the pivot elements are non-null. This problem can be fixed using techniques like partial or total pivoting.

2.3.1 Partial Pivoting

The Gaussian Elimination Algorithm can be modified in order to avoid a breakdown due to zero pivot elements. In fact, it can be demonstrated that if A

Algorithm 2.1 Gauss Elimination Algorithm

1: **for** $k = 1, \ldots, n-1$ **do**
2: **for** $i = k+1, \ldots, n$ **do**
3: $m_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$
4: **for** $j = k+1, \ldots, n$ **do**
5: $a_{ij}^{(k)} - m_{ij} a_{ij}^{(k)}$
6: **end for**
7: $a_{ij}^{(k+1)} = 0$
8: $b_i^k - m_{ik} b_k^{(k)}$
9: **end for**
10: **end for**

is non-singular, then at step k exists at least one index $k \leq l \leq n$ such that $a_{lk}^{(k)} \neq 0$. Hence, in order to apply Gauss Elimination, it suffices to apply a permutation between rows k and l. Moreover, in order to have a higher numerical stability, l can be chosen such that $a_{lk}^{(k)} \neq 0$ and $|a_{lk}^{(k)}|$ is the highest for $k \leq l \leq n$. In fact, if $|a_{lk}^{(k)}| = \max_{i=k,\ldots,n} |a_{ik}^{(k)}|$, then $|m_{ik}| \leq 1$ that is the lowest possible value for the multipliers. It results in a lower round-off error and avoids even overflows. In equation 2.63 is shown matrix $A^{(k)}$ shows at step k. The highlighted part of the k-th column is the part where the k-th pivot element is searched.

$$A^{(k)} = \left[\begin{array}{ccc|ccc|c} a_{11}^{(1)} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & & & & & \vdots \\ \vdots & \ddots & a_{k-1,k-1}^{(k-1)} & & & & a_{k-1,n}^{(k-1)} \\ \hline \vdots & & 0 & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & & a_{k+1,k}^{(k)} & a_{k+1,k+1}^{(k)} & \cdots & a_{k+1,n}^{(k)} \\ \vdots & & & \vdots & \vdots & & \vdots \\ \vdots & & & \vdots & \vdots & & \vdots \\ 0 & \cdots & & a_{nk}^{(k)} & a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k)} \end{array}\right] \tag{2.63}$$

The strategy described is called *partial pivoting* or *column pivoting*.

Algorithm 2.2 describes the entire procedure. Note that, instead of comparing the pivot candidates with zero, the machine precision ϵ_M has been used for the match. Moreover, in row six an operation of row exchanges tracking has been implemented by the use of a vector of integers.

Algorithm 2.2 Gauss Elimination Algorithm with Partial Pivoting

for $k = 1, \ldots, n-1$ **do**
  $|a_{lk}^{(k)}| = \max_{i=k,\ldots,n} |a_{ik}^{(k)}|$
  if $|a_{lk}^{(k)}| < \epsilon_M$ **then**
    STOP - Singular Matrix
  end if
  if $l \neq k$ **then**
    exchange rows l and k
    $p(k) = l$
    exchange constants b_l and b_k
  end if
  for $i = k+1, \ldots, n$ **do**
    $m_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$
    for $j = k+1, \ldots, n$ **do**
      $a_{ij}^{(k+1)} = a_{ij}^{(k)} - m_{ik} a_{kj}^{(k)}$
    end for
    $b_i^{(k+1)} = b_i^{(k)} - m_{ik} b_k^{(k)}$
  end for
  if $|a_{nn}^{(n)}| < \epsilon_M$ **then**
    STOP - Numerically Singular Matrix
  end if
end for

2.3.2 Total Pivoting

The Gaussian Elimination Algorithm with total pivoting has a different strategy in looking for the pivot candidate. In fact, while the partial pivot strategy determines the pivot element by scanning the current sub-column at the k-th step, total pivoting search for the largest entry $|a_{rs}^{(k)}|$ in the current $(k \times k)$ sub-matrix which is highlighted in equation (2.64)

$$A^{(k)} = \left[\begin{array}{ccc|ccc} a_{11}^{(1)} & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{k-1,k-1}^{(k-1)} & & & a_{k-1,n}^{(k-1)} \\ \hline \vdots & & 0 & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & & a_{k+1,k}^{(k)} & a_{k+1,k+1}^{(k)} & \cdots & a_{k+1,n}^{(k)} \\ \vdots & & & \vdots & \vdots & & \vdots \\ \vdots & & & \vdots & \vdots & & \vdots \\ 0 & \cdots & & a_{nk}^{(k)} & a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k)} \end{array}\right]. \tag{2.64}$$

When the best pivot element is found, it is permuted into the (k, k) position. Total pivoting strategy can be summarized as follows

for each STEP k **do**
  find r, s such that $|a_{rs}^{(k)}| = \max_{i,j=k,\ldots,n} |a_{ij}^{(k)}|$
  exchange row k with row r
  exchange column k with column s
  exchange b_k and b_r in b
  remember that unknown x_k and x_s have been exchanged
end for

where it has been stressed that the exchange of the columns needs to be tracked by remembering that unknown have been exchanged as well. Algorithm 2.3 shows how the total pivoting strategy can be implemented. As it can be seen in rows 16 and 17, exchanges between rows and columns have been tracking by the use of two vectors of integers.

Gauss elimination algorithm with total pivoting requires the same number of flops of the algorithm with partial pivoting strategy. Then, the number of operations required amounts to $\mathrm{O}(\frac{2}{3}n^3)$.

2.3.3 *LU* Factorization

Gauss elimination method, as we have seen, transforms a generic linear system $Ax = b$ in an upper triangular system that is easier to solve. In addition, what Gauss elimination does, is to supply the *LU factorization* of the matrix A. In fact, if we define a lower triangular matrix L and an upper triangular matrix U as

$$l_{ij} = \begin{cases} m_{ij}, & \text{if } i > j, \\ 1, & \text{if } i = j, \\ 0, & \text{if } i < j \end{cases} \tag{2.65}$$

and

$$u_{ij} = \begin{cases} a_{ij}^{(n)}, & \text{if } i \leq j \\ 0, & \text{if } i > j, \end{cases} \tag{2.66}$$

where m_{ij} are the Gauss multipliers and a_{ij}^{n} are the coefficients at the last step of Gauss elimination, then the following equation hold

$$A = LU. \tag{2.67}$$

Algorithm 2.3 Gauss Elimination Algorithm with Total Pivoting

1: **for** $k = 1, \ldots, n-1$ **do**
2: **Determine** r and s with $k \leq r \leq n$ and $k \leq s \leq n$ such that
3: $|a_{rs}^{(k)}| = \max\{|a_{ij}^{(k)}| : i = 1, \ldots, n \text{ and } j = k, \ldots, n\}$
4: **if** $|a_{rs}^{(k)}| < \epsilon_M$ **then**
5: STOP - Singular Matrix
6: **end if**
7: **if** $r \neq k$ **then**
8: **exchange** row k with row r
9: **exchange** constants b_r and b_k
10: **end if**
11: **if** $s \neq k$ **then**
12: **exchange** column k with column s
13: **end if**
14: $p(k) = r$
15: $q(k) = s$
16: **for** $h = k+1, \ldots, n$ **do**
17: $m_{hk} = \frac{a_{hk}^{(k)}}{a_{kk}^{(k)}}$
18: **for** $j = k+1, \ldots, n$ **do**
19: $a_{hj}^{(k+1)} = a_{hj}^{(k)} - m_{hk} a_{kj}^{(k)}$
20: **end for**
21: $b_h^{(k+1)} = b_h^{(k)} - m_{hk} b_k^{(k)}$
22: **end for**
23: **if** $|a_{nn}^{(n)}| < \epsilon_M$ **then**
24: STOP - Numerically Singular Matrix
25: **end if**
26: **end for**

In order to prove that (2.67) is true we need the so called *Gauss Elementary Matrices* which are defined in app. B. In fact, we can use the vector $\mathbf{m}_k = (0, \ldots, 0, m_{k+1,k}, m_{k+2,k}, \ldots, m_{kn})^T$ of the multipliers m_{ik} obtained at the k-th step of the Gaussian Elimination in order to build the Gauss elementary matrix $\mathcal{M}_k$. Then the k-th step itself can be represented as follows

$$A^{(k+1)} = \mathcal{M}_k A^{(k)}. \tag{2.68}$$

The entire elimination process can then be represented by successive left multiplications of a Gauss matrix by the matrix A

$$U = A^{(n)} = \mathcal{M}_{n-1} A^{(n-1)} = \mathcal{M}_{n-1}\mathcal{M}_{n-2} A^{(n-2)} = \mathcal{M}_{n-1}\mathcal{M}_{n-2} \cdots \mathcal{M}_1 A^{(1)}, \tag{2.69}$$

where $L^{-1} = \mathcal{M}_{n-1}\mathcal{M}_{n-2}\cdots\mathcal{M}_1$ is a lower triangular matrix because Gauss matrices are lower triangular. From properties 1 and 2 listed in app. B, it is easy to verify that

$$L = \mathcal{M}_1^{-1}\mathcal{M}_2^{-1}\cdots\mathcal{M}_{n-1}^{-1} = \begin{bmatrix} 1 & & & \\ m_{21} & \ddots & & \\ \vdots & \ddots & \ddots & \\ m_{n1} & \cdots & m_{n,n-1} & 1 \end{bmatrix}, \tag{2.70}$$

and then from equation (2.69) the *LU factorization* of $A = LU$ can be obtained. What it can be seen is that the elements of matrix L are obtained directly from the Gauss Elimination Algorithm without additional computation required.

Gauss elimination with partial pivoting can be represented in matrix form as well. In fact, the generic k-th step of the algorithm consists of a row exchange and a row reduction as we have seen earlier. This operations can be represented in matrix form by elementary matrices (see sec. 2.1) and Gauss transformations (Gauss elementary matrices). Consequently, the k-th step can be written as follows

$$A^{(k+1)} = \mathcal{M}_k E^{(k,l)} A^{(k)}. \tag{2.71}$$

The entire algorithm can be expressed as a sequence of rows exchange and row reductions as the generic step, giving as a result the upper triangular matrix we want

$$U = A^{(n)} = \mathcal{M}_{n-1}E^{(n-1,l_{n-1})}\cdots\mathcal{M}_2 E^{(2,l_2)}\mathcal{M}_1 E^{(1,l_1)}A = NA, \tag{2.72}$$

where $\mathcal{M}_i$ is the generic Gauss transformation and $E^{(i,l_i)}$ is the generic rows exchange. It can be demonstrated that

$$N = \mathcal{M}_{n-1}\tilde{\mathcal{M}}_{n-2}\cdots\tilde{\mathcal{M}}_2\tilde{\mathcal{M}}_1 E^{(n-1,l_{n-1})}\cdots E^{(2,l_2)}E^{(1,l_1)} = L^{-1}P,$$

where

$$L^{-1} = \mathcal{M}_{n-1}\tilde{\mathcal{M}}_{n-2}\cdots\tilde{\mathcal{M}}_2\tilde{\mathcal{M}}_1$$

is a lower triangular matrix and

$$P = E^{(n-1,l_{n-1})}\cdots E^{(2,l_2)}E^{(1,l_1)}$$

is a permutation matrix. Note that matrices $\tilde{\mathcal{M}}_i$ are different from matrices

$\mathcal{M}_i$. Nonetheless, they have the same structure. Finally, we can write

$$U = A^{(n)} = L^{-1}PA,$$

and consequently

$$PA = LU, \tag{2.73}$$

that is the *LU factorization with partial pivoting* of matrix A.

By the same reasoning, we can represent in matrix form Gaussian elimination with total pivoting as well. It results in being not so different from the partial pivoting case. In fact, if we add the column pivoting to equation (2.71) we obtain the generic step

$$A^{(k+1)} = \mathcal{M}_k E^{(k,r)} A^{(k)} E^{(k,s)}. \tag{2.74}$$

Then, the entire algorithm can be written as follows

$$\begin{aligned} U &= A^{(n)} \\ &= \mathcal{M}_{n-1}E^{(n-1,r_{n-1})}\cdots\mathcal{M}_2E^{(2,r_2)}\mathcal{M}_1E^{(1,r_1)}AE^{(1,s_1)}E^{(2,s_2)}\cdots E^{(n-1,s_{n-1})} \\ &= NA, \end{aligned} \tag{2.75}$$

where $N = L^{-1}P$ as before and

$$Q = E^{(1,r_1)}AE^{(1,s_1)}E^{(2,s_2)}\cdots E^{(n-1,s_{n-1}} \tag{2.76}$$

is a permutation matrix like P. As a result, we obtain the *LU factorization with total pivoting*

$$PAQ = LU. \tag{2.77}$$

One of the most important things to do is to determine whether the LU factorization of a generic matrix A exists. The following theorem gives necessary and sufficient conditions in order to ensure that a matrix is LU factorizable.

Theorem 2.5. *Let $A \in \mathbb{F}^{(n\times n)}$ be a non-singular matrix, and let A_k be its leading principal minors of order k. If A_k is non-singular for $k = 1, \ldots, n-1$ then the LU factorization of A exist and is unique.*

Proof: The theorem will be demonstrate by induction over n for a matrix $A \in \mathbb{C}^{(n\times n)}$. The same reasoning can be applied for the real case.

$\boxed{\text{n = 1}}$

If $n = 1$, $A_1 = [a_{11}]$. Then $L = [1]$ and $U = [a_{11}]$.

$\boxed{\text{n = k > 1}}$

If $n = k > 1$, A_k can be written as follows

$$A_k = \begin{bmatrix} A_{k-1} & \mathbf{d} \\ \mathbf{c}^* & \alpha \end{bmatrix},$$

where $A_{k-1} = L_{k-1}U_{k-1}$ with L_{k-1} unit[3] lower triangular matrix and U_{k-1} upper triangular matrix. We assume that $\det(A_{k-1}) \neq 0$. Now write L_k and U_k as follows

$$L_k = \begin{bmatrix} L_{k-1} & 0 \\ \mathbf{u}^* & 1 \end{bmatrix}, \qquad U_k = \begin{bmatrix} U_{k-1} & \mathbf{v} \\ \mathbf{0}^* & \beta \end{bmatrix}.$$

In order to have $A_k = L_kU_k$, $\mathbf{u}, \mathbf{v}$ and β need to be determined. Matrix A_k can also be written as follows

$$L_kU_k = \begin{bmatrix} L_{k-1}U_{k-1} & L_{k-1}\mathbf{v} \\ \mathbf{u}^*U_{k-1} & \mathbf{u}^*\mathbf{v} + \beta \end{bmatrix},$$

and relation $A_k = L_kU_k$ is verified if and only if

1. $L_{k-1}\mathbf{v} = \mathbf{d}$,
2. $U_{k-1}^*\mathbf{u} = \mathbf{c}$,
3. $\mathbf{u}^*\mathbf{v} + \beta = \alpha$.

Relations 1) and 2) univocally define the vectors $\mathbf{u}$ and $\mathbf{v}$ because they are solutions of non homogeneous linear systems whose matrices are non-singular. In fact, $\det(L_{k-1}) = 1$ and $\det(U_{k-1}) = \det(A_{k-1}) \neq 0$ by assumption. Hence, the third relation univocally defines β which can be found from $\beta = \alpha - \mathbf{u}^*\mathbf{v}$. Consequently, since k is a generic index, it has been demonstrated by induction over k what stated in the theorem. □

[3]A *unit lower triangular matrix* is a lower triangular matrix which has ones along the main diagonal.

Chapter 3

Formation Control via Complex Laplacian

Formation control via complex Laplacian has been proposed in [29]. It is a new approach to agents formation and results in easier laws to control a group of agents. In [29] the problem of formation control has been studied in the plane and results both for single-integrator and double-integrator agent internal dynamic model have been supplied. In this chapter such results will be presented in section 3.3, while sections 3.1 and 3.2 present preliminary materials in order to understand the last section.

3.1 Sensing Digraph

Multi Agent Systems can be well modelled by graphs. In fact, *agents* can be represented by the *vertices* of a graph, while *edges (arcs)* can represent *existing interactions (links)* among agents themselves. In different formation control approaches simple graphs are used. Instead, in [29] weighted directed graphs $\mathcal{D} = (\mathcal{V}, \mathcal{A}, w)$ has been used to elaborate the control theory that will be shown. They are more suitable since arcs orientation is used to represent the direction information flow has in agents interaction. Moreover, arc weights play a key role in agents formation control laws as it will be seen in later sections. A complex weight $w_{ij} \in \mathbb{C}$ is associated to each arc $(j, i) \in \mathcal{A}$, and the complex Laplacian associated to $\mathcal{D}$ is

$$l_{ij} = \begin{cases} -w_{ij} & \text{if } i \neq j \text{ and } j \in N_i^-, \\ 0 & \text{if } i \neq j \text{ and } j \notin N_i^-, \\ \sum_{j \in N_i^-} w_{ij} & \text{if } i = j, \end{cases} \tag{3.1}$$

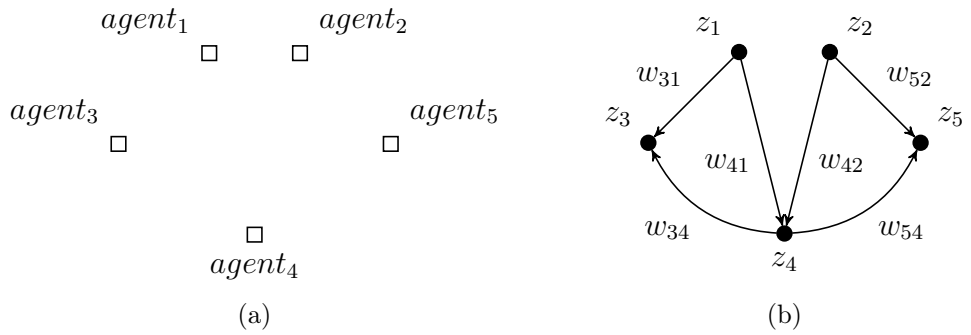


Figure 3.1: Sensing digraph example. In (b) is shown a sensing digraph for the group of agents depicted in (a).

where N_i^- is the in-neighbor set of vertex i. In the graph model, vertices $j \in N_i^-$, which are tails of arcs with i as head, are nothing less than agents from which information is sensed by agent i.

In fig. (3.1) is shown a simple example of a group of agents modelled by a weighted digraph $\mathcal{D}$. As it can be seen in fig. (3.1b), $agent_1$ and $agent_2$ do not receive any information from the rest of the group since they have no incoming arcs, while the other agents do. That is the way the digraph conveys information about agents formation. The weighted digraph which is used to represent an agents formation is called *sensing digraph*.

3.2 Planar Formation

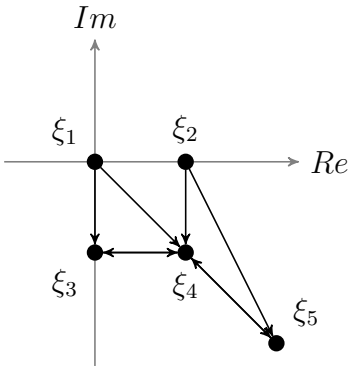
In the plane, a tuple of n complex numbers

$$\xi = [\xi_1, \xi_2, \ldots, \xi_n]^T \tag{3.2}$$

is called a *formation basis* for n agents, which defines a *geometric pattern* in a specific coordinate system. In fig.(3.2) is shown a group of agents and its planar formation. Agents are not randomly scattered in the plane, but are disposed in specific positions so that a global configuration is reached. The formation basis specifies the position for each agent and, indirectly, the relative distance among agents, that characterize the shape of the formation. Usually two agents are not expected to overlap each other, so it is assumed that

$$\xi_i \neq \xi_j \quad \text{for} \quad i \neq j. \tag{3.3}$$

A formation with four degrees of freedom (translation in two main direc-



$$\xi = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \\ \xi_4 \\ \xi_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ -2\iota \\ 2-2\iota \\ 4-4\iota \end{bmatrix}$$

Figure 3.2: Example of a formation basis.

tions, rotation, and scaling) is defined by

$$F_\xi = c_1 \mathbf{1}_n + c_2 \xi, \tag{3.4}$$

where $c_1, c_2 \in \mathbb{C}$. Those complex constants are deputed to make a planar formation move or scale. In fact, it can be seen that

- parameter c_1 controls formation *translation*;
- parameter c_2 controls formation *rotation* and *scaling*. Let c_2 be written in polar form

  $$c_2 = |c_2| \, \mathrm{e}^{\iota\beta} .$$

  Then, we can scale a formation by $|c_2|$ and rotate a formation by the angle β. Note that rotation can be obtained by taking c_2 so that $|c_2| = 1$, while scaling can be obtained by taking c_2 real.

As an example, the planar formations $F_\xi^{(1)}$,$F_\xi^{(2)}$,and $F_\xi^{(3)}$ of four agents in figure (3.3) are obtained from the same basis via translating, rotating, and scaling. When $|c_1| = 1$, then the formation is obtained from the basis via translation and rotation only, a case which is more familiar to everyone.

Denote $z = [z_1, \cdots, z_n]^T \in \mathbb{C}^n$ the aggregate position vector of n agents. It is said that the n agents *form a planar formation* F_ξ with respect to basis ξ if there exist complex constants c_1 and c_2 such that $z = c_1 \mathbf{1}_n + c_2 \xi$. The n agents are said to *asymptotically reach a planar formation* F_ξ if there exist complex constants c_1 and c_2 such that

$$\lim_{t \to \infty} z(t) = c_1 \mathbf{1}_n + c_2 \xi. \tag{3.5}$$

As it will be shown in sec. 3.3, a planar formation is strictly related to the

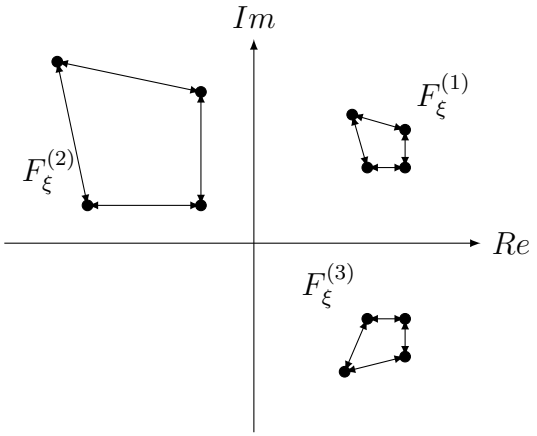


Figure 3.3: Agents Formation up to translation, rotation and scaling. As it can be seen, $F_\xi^{(1)}$ is the original formation, $F_\xi^{(2)}$ is the formation after being translated and scaled and $F_\xi^{(3)}$ is the formation after being translated and rotated.

complex Laplacian of the sensing digraph which represents the multi-agent system. In fact, ξ is another linearly independent eigenvector of L associated with zero eigenvalues in addition to the eigenvector of ones. Planar formation F_ξ is then a linear combination of two independent eigenvectors of the complex Laplacian. For those reasons, in order to uniquely determine the location, orientation and size of the formation in a leader-follower configuration, two co-leaders have to be considered.

Example

To show how a planar formation can be translated, rotated and scaled, let us have an example. Let ξ be the planar formation

$$\xi = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \\ \xi_4 \\ \xi_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ -2\imath \\ 2 - 2\imath \\ 4 - 4\imath \end{bmatrix}, \tag{3.6}$$

which is represented in fig. (3.2). As we have already seen, complex constants c_1 and c_2 are the parameters by which a planar formation can be controlled.

1. **TRANSLATION**

   Let c_1 and c_2 be

$$\begin{cases} c_1 = -2 - 2\imath \\ c_2 = 1, \end{cases}$$

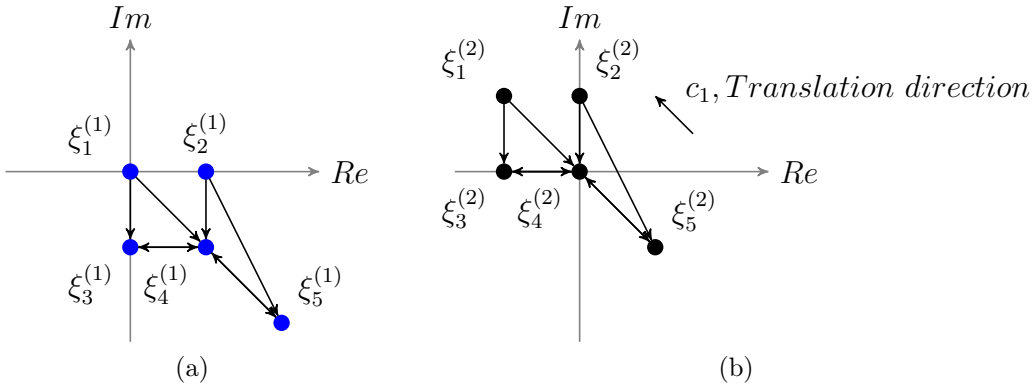


Figure 3.4: Example of an agents formation translation.

where $c_1 = 1$ ensures no rotation, neither scaling. Asymptotically, the agents reach the planar formation

$$F_\xi = c_1 \mathbf{1}_n + c_2 \xi = \begin{bmatrix} -2+2\iota \\ -2+2\iota \\ -2+2\iota \\ -2+2\iota \\ -2+2\iota \end{bmatrix} + \begin{bmatrix} 0 \\ 2 \\ -2\iota \\ 2-2\iota \\ 4-4\iota \end{bmatrix} = \begin{bmatrix} -2+2\iota \\ 2\iota \\ -2 \\ 0 \\ 2-2\iota \end{bmatrix}.$$

As it can be seen in fig. (3.4) the planar formation has been translated in the direction of the vector representing the constant c_1.

2. **ROTATION**

Let c_1 and c_2 be

$$\begin{cases} c_1 = 0 \\ c_2 = -\iota = \mathrm{e}^{-\iota\frac{\pi}{2}}, \end{cases}$$

where $c_1 = 0$ ensures no translation and $|c_2| = 1$ ensures no scaling. Asymptotically, the agents reach the planar formation

$$F_\xi = c_1 \mathbf{1}_n + c_2 \xi = -\iota \begin{bmatrix} 0 \\ 2 \\ -2\iota \\ 2-2\iota \\ 4-4\iota \end{bmatrix} = \mathrm{e}^{-\iota\frac{\pi}{2}} \begin{bmatrix} 0 \\ 2 \\ 2\,\mathrm{e}^{-\iota\frac{\pi}{2}} \\ 2\sqrt{2}\,\mathrm{e}^{-\iota\frac{\pi}{2}} \\ 4\sqrt{2}\,\mathrm{e}^{-\iota\frac{\pi}{2}} \end{bmatrix} = \begin{bmatrix} 0 \\ -2\iota \\ -2 \\ -2-2\iota \\ -4-4\iota \end{bmatrix}.$$

As it can be seen in fig.(3.5) the planar formation has been rotated by the angle $\arg(c_2) = -\frac{\pi}{2}$.

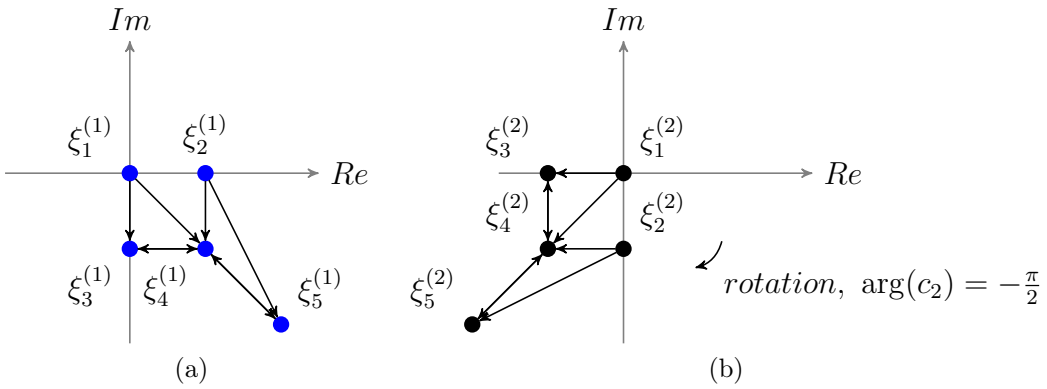


Figure 3.5: Example of an agents formation rotation.

3. **SCALING**

   Let c_1 and c_2 be

$$\begin{cases} c_1 = 0 \\ c_2 = 2, \end{cases}$$

   where c_1 value ensures no translation and c_2 value ensures no rotation. Asymptotically, the agents reach the planar formation

$$F_\xi = c_1 \mathbf{1}_n + c_2 \xi = 2 \begin{bmatrix} 0 \\ 2 \\ -2\iota \\ 2 - 2\iota \\ 4 - 4\iota \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ -4\iota \\ 4 - 4\iota \\ 8 - 8\iota \end{bmatrix}.$$

   As it can be seen in fig. (3.6) the planar formation has been scaled by $|c_2| = 2$.

For the sake of completeness, in fig. (3.7) is shown the formation basis and the planar formation reached by the use of both the parameters already seen $c_1 = -2 + 2\iota$ and $c_2 = -2\iota$, which affect the four degree of freedom at the same time.

3.3 Fundamental Results

Let us consider a group of n agents in the plane labelled $1, \ldots, n$, consisting of leaders and followers. As already said, suppose that there are two leaders in the group (without loss of generality, say 1 and 2) and all the others are followers. The positions of the n agents are denoted by complex numbers $z_1, \ldots, z_n \in \mathbb{C}$.

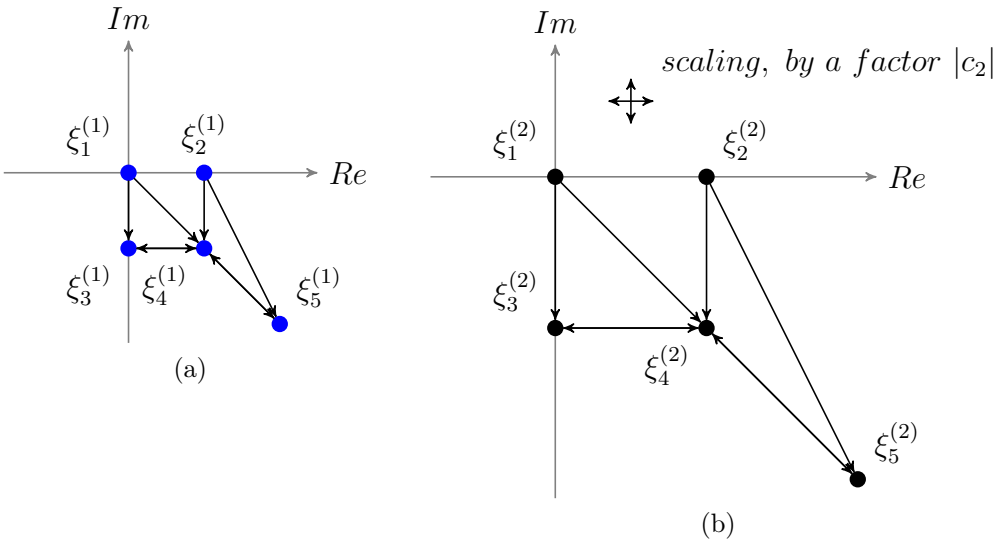


Figure 3.6: Example of an agents formation scaling.

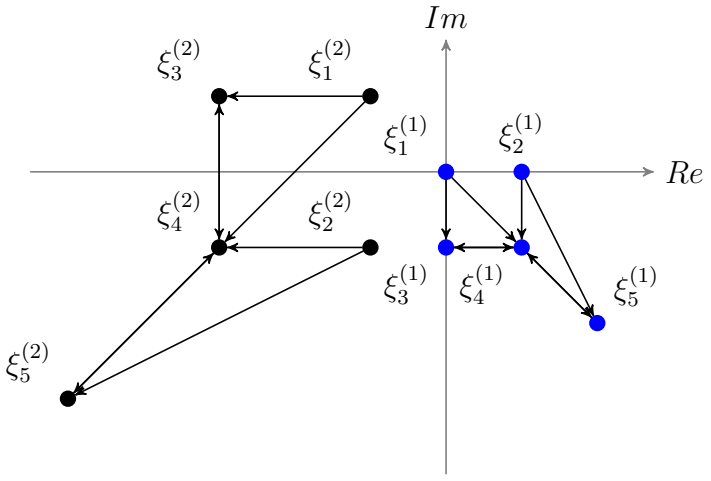


Figure 3.7: Example of the overall behavior of an agents formation under translation, rotation and scaling.

In order to represent the sensing graph, a digraph $\mathcal{D}$ of n nodes has been used, in which $1, 2$ are leader agents, $3, \ldots, n$ are followers agents, and an edge (j, i) indicates that agent i can measure the relative position of agent j, namely, $(z_j - z_i)$. In addition, it has been considered a sensing digraph without self-loops. Since in a leader-follower network, the leader agents do not interact with the follower agents, and do not need to access the information from the follower agents, the sensing graph $\mathcal{D}$ has the following property.

(P1): Leader nodes (1 and 2) do not have incoming edges.

Thus, the Laplacian of $\mathcal{D}$ takes the following form.

$$L = \left[\begin{array}{c|c} 0_{2\times 2} & 0_{2\times(n-2)} \\ \hline L_{lf} & L_{ff} \end{array}\right] \tag{3.7}$$

In the following subsections analysis has been done for two internal dynamic models of the agents, that is for the case of single-integrator kinematics and double-integrator dynamics.

3.3.1 Single-Integrator Kinematics

Suppose that each agent is governed by a single-integrator kinematics

$$\dot{z}_i = v_i, \tag{3.8}$$

where $z_i \in \mathbb{C}$ represents the position of agent i in the plane and $v_i \in \mathbb{C}$ represent the velocity control input. Consider the sensing graph $\mathcal{D}$ and suppose that the agents take the following control laws:

$$\begin{aligned} v_i &= 0, && i = 1, 2; \\ v_i &= \sum_{j\in N_i^-} w_{ij}(z_j - z_i), && i = 3, \ldots, n, \end{aligned} \tag{3.9}$$

where $w_{ij} = k_{ij}\, \mathrm{e}^{\iota\alpha_{ij}}$ is a complex weight with $k_{ij} > 0$ and $\alpha_{ij} \in [-\pi, \pi)$. For a specific formation basis $\xi \in \mathbb{C}$ satisfying $\xi_i \neq \xi_j$, each agent i can arbitrarily choose weights $w_{ij}, j \in N_i^-$, such that

$$\sum_{j\in N_i^-} w_{ij}(\xi_j - \xi_i) = 0. \tag{3.10}$$

The interaction rule in (3.9) can be implemented locally by only accessing the relative position information from its neighbors.

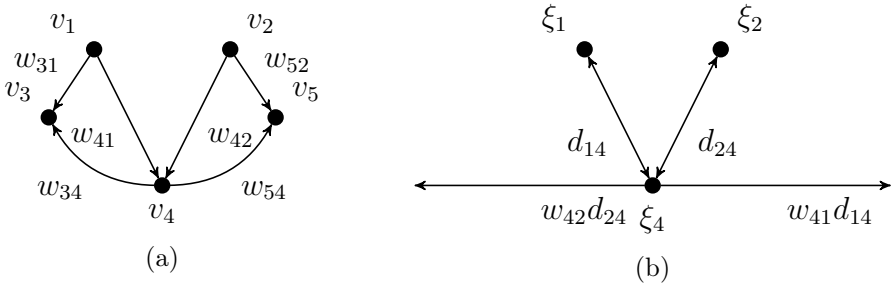


Figure 3.8: Example of a complex weights control law for a formation basis.

Suppose we have the formation basis ξ representing the agents formation whose digraph is depicted in fig.(3.8a)

$$\xi = \begin{bmatrix} -1.5 + 1.5\iota \\ 1.5 + 1.5\iota \\ -2.5 \\ -1.5\iota \\ 2.5 \end{bmatrix}. \tag{3.11}$$

Follower agents will then choose complex weights such that equation 3.10 is verified. For instance, let us consider $agent_4$. $Agent_4$ has two incoming arcs from leaders $agent_1$ and $agent_2$, with complex weights w_{41} and w_{42}. Thus, in order to verify the aforementioned control law, weights have to be chosen such that

$$\begin{aligned} \sum_{j \in N_i^-} w_{ij}(\xi_j - \xi_i) &= w_{41}(\xi_1 - \xi_4) + w_{42}(\xi_2 - \xi_4) \\ &= w_{41}[(-1.5 + 1.5\iota) - (-1.5\iota)] + w_{42}[(1.5 + 1.5\iota) - (-1.5\iota)] \\ &= w_{41}(-1.5 + 3\iota) + w_{42}(1.5 + 3\iota) = 0. \end{aligned}$$

As it can be seen, $agent_4$ can choose one of the weights arbitrarily, the other one is then univocally determined. In this case, a practical choice is

$$\begin{aligned} w_{41} &= -1.5 - 3\iota \\ w_{42} &= -1.5 + 3\iota, \end{aligned}$$

which verify the relation above. In fig.(3.8b) are shown the resulting vectors $w_{41}d_{14}$ and $w_{42}d_{24}$, which are opposite and sum to zero as necessary.

If the two co-leaders were to be translated, the chosen weights would yield a velocity $v \neq 0$, given by interaction rule (3.9). In this case, $agent_4$ will follow

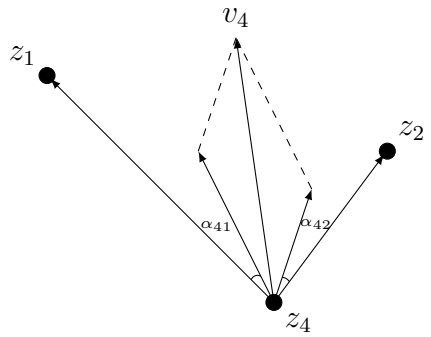


Figure 3.9: Interaction rule for complex weights. The picture exemplifies how complex weights affect the velocity of an agent.

the co-leaders with a velocity

$$v_4 = w_{41}(z_1 - z_4) + w_{42}(z_2 - z_4).$$

Let $z = [z_1, z_2, \ldots, z_n]^T \in \mathbb{C}^n$. Then the overall dynamics of the agents can be written as

$$\dot{z} = -Lz, \tag{3.12}$$

where L is the complex-valued Laplacian of $\mathcal{D}$ defined earlier. Denote

$$\bar{z_1} = z_1(0) \quad \text{and} \quad \bar{z_2} = z_2(0). \tag{3.13}$$

Next, it is shown a necessary and sufficient condition such that any equilibrium state of (3.12) forms a planar formation F_ξ.

Theorem 3.1. *Assume that $\bar{z_1} \neq \bar{z_2}$ and moreover assume that $\xi \in \mathbb{C}^n$ satisfies $\xi_i \neq \xi_j$ for $i \neq j$. Then every equilibrium state of (3.12) forms a planar formation $F_\xi = c_1\mathbf{1}_n + c_2\xi$ with*

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 & \xi_1 \\ 1 & \xi_2 \end{bmatrix}^{-1} \begin{bmatrix} \bar{z_1} \\ \bar{z_2} \end{bmatrix} \tag{3.14}$$

if and only if

$$L\xi = 0 \quad \textit{and} \quad \det(L_{ff}) \neq 0. \tag{3.15}$$

Proof: *(Sufficiency)* From the condition $L\xi = 0$, it is known that L has a zero eigenvalue with an associated eigenvector ξ. On the other hand, the Laplacian L always has a zero eigenvalue with an associated eigenvector $\mathbf{1}_n$. The two eigenvectors $\mathbf{1}_n$ and ξ are linearly independent because $\xi_i \neq \xi_j$. Moreover, it follows from the condition $\det(L_{ff}) \neq 0$ that $\text{rank}(L) = n - 2$ and L has only

two zero eigenvalues. So the null space of L is

$$\ker(L) = \{c_1\mathbf{1}_n + c_2\xi : c_1, c_2 \in \mathbb{C}\} \tag{3.16}$$

and thus every equilibrium state forms a planar formation $F_\xi = c_1\mathbf{1}_n + c_2\xi$. Notice that $z_1(t) = \bar{z}_1$ and $z_2(t) = \bar{z}_2$. Therefore (3.14) follows.

(Necessity) Suppose on the contrary that $L\xi \neq 0$. Then $L(c_1\mathbf{1}_n + c_2\xi) \neq 0$ for any c_1 and c_2, which means a state corresponding to a planar formation F_ξ cannot be an equilibrium state of (3.12). On the other hand, suppose on the contrary that $\det(L_{ff}) = 0$. Thus it could be found a vector $\eta_f \in \mathbb{C}^{(n-2)}$ such that $L_{ff}\eta_f = 0$. As a result $\eta = [0\ 0\ \eta_f^T]^T$ is the null space of L. It can be checked that $\mathbf{1}_n, \xi$ and η are linearly independent since $\xi_i \neq \xi_j$ in ξ. Thus, the equilibrium state η does not correspond to any planar formation F_ξ generated from basis ξ. □

Remark 3.1. *From Theorem 3.1 it can be seen that the equilibrium formation of the n agents is uniquely determined by the two leaders' location. If the two leader agents do not remain stationary but asymptotically converge to two different locations, then the limit positions of two co-leaders specify the planar formation F_ξ. Hence, by controlling the motions of two co-leaders, the group formation can be rotated, translated, and scaled.*

Let us have the formation basis (3.11) already seen in the last example. The inverse of a generic square (2×2) matrix is

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \rightarrow A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

Then, equation (3.14) can be written as

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \frac{1}{(\xi_2 - \xi_1)} \begin{bmatrix} \xi_2 & -\xi_1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \bar{z_1} \\ \bar{z_2} \end{bmatrix}. \tag{3.17}$$

Suppose that the conditions $L\xi = 0 \quad \text{and} \quad \det(L_{ff}) \neq 0$ are satisfied, then theorem 3.1 holds. Substituting the values for ξ_1 and ξ_2 in the expression above we obtain

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1.5 + 1.5\iota & 1.5 - 1.5\iota \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \bar{z_1} \\ \bar{z_2} \end{bmatrix}, \tag{3.18}$$

that is the relation by which parameters c_1 and c_2 can be determined. Let us

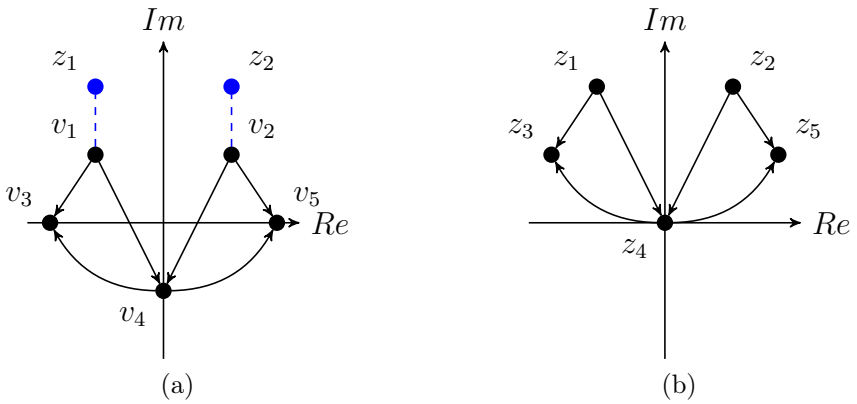


Figure 3.10: Example of a formation control by its leaders. In (a) is shown a digraph for a formation basis. By translating its leaders (blue vertices) the entire formation is being translated, as shown in (b).

consider leaders position $\bar{z}_1 = -1.5+3\iota$ and $\bar{z}_2 = 1.5+3\iota$. Then, the formation will be determined by the following parameters

$$c_1 = \frac{1}{3}(\xi_2\bar{z}_1 - \xi_1\bar{z}_2) = 1.5\iota,$$
$$c_2 = \frac{1}{3}(-\bar{z}_1 + \bar{z}_2) = 1.$$

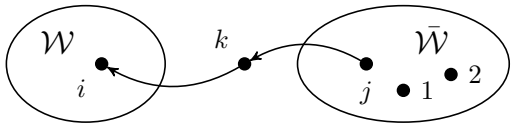
As it can be seen from $\bar{z}_1$ and $\bar{z}_2$, leaders were simply translated along the direction of the half positive imaginary axis. Moreover, parameters c_1 and c_2 obtained from theorem 3.1 affect the entire formation by only translating it, without rotation nor scaling. Then, controlling the leaders movements the whole formation can be controlled. In fig.(3.10) the formation basis before and after translation is shown.

Theorem 3.1 requires to check whether $\det(L_{ff}) \neq 0$. A graphical condition could also be given for the same purpose.

Theorem 3.2. *For a digraph $\mathcal{D}$ and a formation basis $\xi \in \mathbb{C}^n$ satisfying $\xi_i \neq \xi_j$, if $\det(L_{ff}) \neq 0$ for L satisfying $L\xi = 0$, then every follower node in $\mathcal{D}$ is 2-reachable from a leader node.*

Proof: Theorem can be proved in a contrapositive form.

Suppose that not every follower node in $\mathcal{D}$ is 2-reachable from a leader node. That is, a node exists, say k, such that when it is removed from the graph, some nodes are not reachable from a leader node any more. Let $\mathcal{W}$ denote the set of nodes that are not reachable from a leader node after removing node k and let $\bar{\mathcal{W}}$ denote the nodes not in $\mathcal{W} \cup \{k\}$. Then it is certain that nodes in $\mathcal{W}$ are not reachable from any node in $\bar{\mathcal{W}}$.



In particular, for $i \in \mathcal{W}$ and $j \in \bar{\mathcal{W}}$, the (i,j)-th entry of L must be 0. Relabelling the nodes in the order of $1, 2, \ldots, m, \ldots, n$ such that $\bar{\mathcal{W}} = \{1, \ldots, m-1\}$, m is the node k before relabelling, and $\mathcal{W} = \{m+1, \ldots, n\}$. Then the Laplacian L after relabelling must be of the following form

$$L = \left[\begin{array}{c|c|c} * & * & * \\ \hline 0 & l_m & L_{\mathcal{W}} \end{array}\right]$$

where $l_m \in \mathbb{C}^{(n-m)}$ and $L_{\mathcal{W}} \in \mathcal{C}^{(n-m)\times(n-m)}$. Denote the formation basis ξ after relabelling by

$$\xi = \begin{bmatrix} \xi_a \\ \xi_b \end{bmatrix},$$

where $\xi_a \in \mathbb{C}^{(m-1)}$ and $\xi_b \in \mathbb{C}^{(n-m+1)}$. From the definition of L and from the conditions $L\xi = 0$ and $L\mathbf{1}_n = 0$, then it can be seen that

$$[l_m \; L_{\mathcal{W}}]\mathbf{1}_{(n-m+1)} = 0, \tag{3.19a}$$

$$[l_m \; L_{\mathcal{W}}]\xi_b = 0. \tag{3.19b}$$

Since $\mathbf{1}_{(n-m+1)}$ and ξ_b are linearly independent by assumption, it is then known from (3.19) that $\text{rank}[l_m \; L_{\mathcal{W}}] \leq (n-m-1)$ and therefore $\det(L_{ff}) = 0$. □

Combining Theorem 3.1 and Theorem 3.2, it is known that in order to uniquely define a planar formation, the digraph $\mathcal{D}$ should have the property that every follower node is 2-reachable from a leader node. If the property does not hold, then for whatever choice of weights, the planar formation can be deformed and the digraph could not define a unique planar formation. For example, consider a digraph $\mathcal{D}$ of 5 nodes shown in fig.(3.11a). For this digraph, the follower nodes 4 and 5 are not 2-reachable from a leader node as they are not reachable from any leader node when node 3 is removed. Consequently, the digraph could not define a unique planar formation. As it can be seen in fig.(3.11b), in addition to rotation, translation and scaling, the formation can also be bent. Adding at least one of the two arcs $(1,4)$ and $(2,5)$, the formation becomes 2-reachable and cannot be bent any more (fig.(3.11c)).

Theorem 3.2 can be verified from an algebraically point of view as well. A formation basis for the group of agents represented by the digraph in fig.(3.11a)

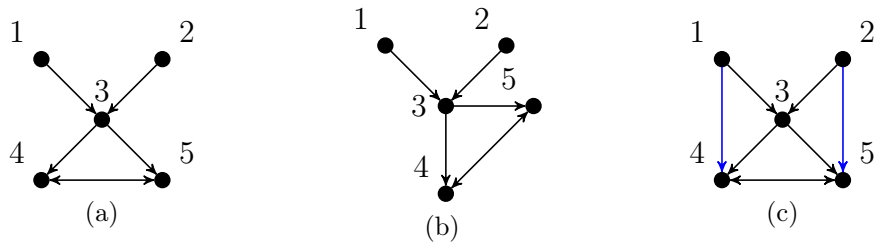


Figure 3.11: Example of a non 2-reachable agents formation. If the sensing digraph is not 2-reachable then the agents formation (a) can be bent (b). Adding at least one of the two blue arcs (c) the formation becomes 2-reachable and cannot be bent any more.

could be

$$\xi = \begin{bmatrix} 0 \\ 2 \\ 1-\iota \\ -2\iota \\ 2-2\iota \end{bmatrix},$$

and the corresponding Laplacian matrix is

$$L = \left[\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline -1-\iota & -1+\iota & 2 & 0 & 0 \\ 0 & 0 & -2 & 1-\iota & 1+\iota \\ 0 & 0 & 2\iota & -1-\iota & 1-\iota \end{array}\right].$$

Note that the condition $L\xi = 0$ holds. Nonetheless, since the digraph is not 2-reachable, the determinant of L_{ff} is zero as stated in theorem 3.2,

$$\det(L_{ff}) = 2(1-\iota)(1-\iota) - 2(1+\iota)(-1-\iota) = 0.$$

Adding at the digraph the arc $(1,4)$, weighted by w_{41}, every follower node becomes 2-reachable from a leader node. The Laplacian which verifies the relation $L\xi = 0$ becomes

$$L = \left[\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline -1-\iota & -1+\iota & 2 & 0 & 0 \\ -1+\iota & 0 & -2 & 1-3\iota & 2+2\iota \\ 0 & 0 & 2\iota & -1-\iota & 1-\iota \end{array}\right],$$

where blue numbers changed from the last configuration because of the new arc. The determinant of the follower-follower sub-matrix is not null any more because of the 2-reachability. Then,

$$\det(L_{ff}) = 2(1-3\iota)(1-\iota) - 2(2+2\iota)(-1-\iota) = -4 \neq 0.$$

as we expected.

The next step is to determine whether the n agents can asymptotically reach a planar formation, i.e., the stability of system $\dot{z} = -Lz$. Before presenting the results on stability, it is provided a result on the invariance property for the operation of pre-multiplying an invertible diagonal matrix D. It is an important property ensuring that the equilibrium formations are preserved.

Theorem 3.3. *Every equilibrium state of system (3.12) forms a planar formation F_ξ if and only if every equilibrium state of the following system*

$$\dot{z} = -DLz \tag{3.20}$$

forms a planar formation F_ξ for all invertible diagonal matrix $D = diag(d_1, d_2, \cdots, d_n) \in \mathbb{C}^{n\times n}$.

Proof: Since D is diagonal and invertible, it follows that the null space of DL is the same as L. So the two systems have the same equilibrium states and form the same planar formation. □

When L is pre-multiplied by D, the complex weights on edges having head at agent i are multiplied by a non-zero complex number d_i. Therefore, the interaction rule is still locally implementable using relative position information only. Generally, for a complex-valued Laplacian L satisfying the conditions of Theorem 3.1, L may have eigenvalues with both negative and positive real parts and thus system (3.12) may not be asymptotically stable with respect to the equilibrium subspace $\ker(L)$. In other words, the n agents may not be able to asymptotically reach a planar formation F_ξ. However, it has been shown in the next result that if certain conditions are satisfied, there exists an invertible diagonal matrix D such that DL has all other eigenvalues with positive real parts in addition to two 0 eigenvalues and thus $\dot{z} = -DLz$ is asymptotically stable with respect to the equilibrium subspace $\ker(L)$. Such a matrix D is called a *stabilizing matrix*. Since pre-multiplying such a matrix D does not change the planar formation at equilibrium states from Theorem 3.3, a local

interaction rule is thus obtained such that the n agents asymptotically reach a desired planar formation F_ξ.

Theorem 3.4. *Consider a formation basis $\xi \in \mathbb{C}^n$ satisfying $\xi_i \neq \xi_j$ and suppose a complex Laplacian L of the sensing graph $\mathcal{D}$ satisfies $L\xi = 0$ and $\det(L_{ff}) \neq 0$. If there exists a permutation matrix P such that all the leading principal minors of $PL_{ff}P^T$ are non-zero, then a stabilizing matrix D for system (3.12) exists.*

Theorem 3.4 requires the following result related to the multiplicative inverse eigenvalue problem.

Theorem 3.5 (Ballantine(1970))**.** *Let A be an $n \times n$ complex matrix all of whose leading principal minors are non-zero. Then there is an $n \times n$ complex diagonal matrix M such that all the eigenvalues of MA are positive and simple.*

Proof of Theorem 3.4: By the condition that there is a permutation matrix P such that all the leading principal minors of $PL_{ff}P^T$ are non-zero, then it follows from Theorem 3.5 that there exists a diagonal matrix M such that $MPL_{ff}P^T$ has all eigenvalues with positive real parts. Note that P^TMPL_{ff} has the same eigenvalues as $MPL_{ff}P^T$ since the permutation transformation does not change the eigenvalues. Also, note that P^TMP is a diagonal matrix as well, and it can be denoted as $M' = P^TMP$. Let

$$D = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & M' \end{bmatrix}. \tag{3.21}$$

Then the system $\dot{z} = -DLz$ has two zero eigenvalues and all the others have negative real parts. As a result D is a stabilizing matrix. □

By numerous simulations it is known that for most complex Laplacians L with weights generated randomly satisfying $L\xi = 0$ and $\det(L_{ff}) \neq 0$, all the leading principal minors of L_{ff} are non-zero. Thus, $P = I$. For rare cases, a relabel of the nodes was needed. Consequently, central to find a stabilizing matrix D such that

$$D = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & M \end{bmatrix} \tag{3.22}$$

is to find the diagonal matrix M as stated in Ballantine's theorem. Denote $M = \mathrm{diag}(m_1, \ldots, m_{n-2})$ and denote $A_{(1\sim i)}$ the sub-matrix formed by the first i rows and columns of a matrix A. Algorithm 3.1 gives the complete description in order to find D.

Algorithm 3.1 Single-Integrator Kinematics

Find a permutation matrix P such that $A = PL_{ff}P^T$ has all non-zero leading principal minors.

for $i = 1, \ldots, n-2$ **do**

Find m_i to assign the eigenvalues of $\text{diag}(m_1, \ldots, m_i)A_{(1\sim i)}$ in the open right half complex plane.

end for

Construct D according to (3.22).

The algorithm suggests to find the diagonal entries of M one by one in an iterative way. The success is ensured by the constructive proof to show the existence of such a diagonal matrix in Ballantine(1970) [5].

Remark 3.2. *It has been discussed how a planar formation is achieved by a complex Laplacian based control law. The results can be simply extended to reach and maintain a formation shape while moving. Suppose that the velocities ,for single integrator kinematics model, of the two co-leaders are synchronized, and say $v_0(t)$. When this synchronized velocity information is available to all the followers, then the following control law is the adjusted one to reach a formation while moving.*

$$\begin{cases} v_i = v_0(t), & i = 1, 2; \\ v_i = \sum_{j \in N_i^-} w_{ij}(z_j - z_i) + v_0(t), & i = 3, \cdots, n. \end{cases} \tag{3.23}$$

When the leaders' velocity is not accessible by all followers, estimation schemes can be adopted to estimate it and then the leaders' velocity in (3.23) can be replaced with the estimated one.

3.3.2 Double-Integrator Dynamics

Suppose that each agent is governed by a double-integrator dynamics

$$\begin{cases} \dot{z}_i = v_i \\ \dot{v}_i = a_i \end{cases} \tag{3.24}$$

where the position $z_i \in \mathbb{C}$ and the velocity $v_i \in \mathbb{C}$ are the state and the acceleration $a_i \in \mathbb{C}$ is the control input. Consider the sensing graph $\mathcal{D}$ and

suppose that each agent takes the control law

$$\begin{aligned} a_i &= -\gamma v_i, && i = 1, 2; \\ a_i &= \sum_{j \in N_i} w_{ij}(z_j - z_i) - \gamma v_i, && i = 3, \cdots, n, \end{aligned} \tag{3.25}$$

where $w_{ij} = k_{ij} e^{\iota \alpha_{ij}}$ is a complex weight with $k_{ij} > 0$ and $\alpha_{ij} \in [-\pi, \pi)$, and $\gamma > 0$ is a real number representing the damping gain. Write $z = [z_1, \ldots, z_n]^T$ and $v = [v_1, \ldots, v_n]^T$. Then the overall system of the n agents under the interaction rule (3.25) can be written as

$$\begin{bmatrix} \dot{z} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0_{n \times n} & I_n \\ -L & -\gamma I_n \end{bmatrix} \begin{bmatrix} z \\ v \end{bmatrix} \tag{3.26}$$

where L is the Laplacian of $\mathcal{D}$ defined in (3.1). The interaction rule (3.25) similar to (3.9) can also be locally implemented, which requires only the relative positions of the neighbors and its own velocity. Denote

$$\bar{z}_1 = \lim_{t \to \infty} z_1(t) \quad \text{and} \quad \bar{z}_2 = \lim_{t \to \infty} z_2(t). \tag{3.27}$$

Next it is shown that the condition in Theorem 3.1 is also a necessary and sufficient condition such that the equilibrium states $(\bar{z}, \bar{v})$ of system (3.26) form a planar formation F_ξ, i.e.,

$$\bar{z} = c_1 \mathbf{1}_n + c_2 \xi \quad \text{and} \quad \bar{v} = 0, \tag{3.28}$$

where c_1 and c_2 can be obtained from (3.14). Moreover, it is shown that the equilibrium formations are invariant to the operation of pre-multiplying by an invertible diagonal complex matrix D.

Theorem 3.6. *Assume that $\bar{z}_1 \neq \bar{z}_2$ and moreover assume that $\xi \in \mathbb{C}^n$ satisfies $\xi_i \neq \xi_j$ for $i \neq j$. Then the following are equivalent.*

1. *$L\xi = 0$ and $\det(L_{ff}) \neq 0$.*

2. *equilibrium state of system (3.26) forms a planar formation F_ξ.*

3. *Every equilibrium state of the following system*

$$\begin{bmatrix} \dot{z} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0_{n \times n} & I_n \\ -DL & -\gamma I_n \end{bmatrix} \begin{bmatrix} z \\ v \end{bmatrix}, \tag{3.29}$$

for all invertible diagonal matrix $D = \text{diag}(d_1, d_2, \ldots, d_n) \in \mathbb{C}^{n\times n}$*, forms a planar formation* F_ξ*.*

Proof: By simply checking system (3.29), it can be obtained that the equilibrium states satisfy $L\bar{z} = 0$ and $\bar{v} = 0$, Thus, the conclusion follows from the same argument in Theorem 3.1 and Theorem 3.3. □

According to Theorem 3.6 the equilibrium formation for the double integrator model is characterized (as was the case with the single integrator model) by $L\xi = 0$ and $\det(L_{ff}) \neq 0$. Then it is clear from Theorem 3.2 that a necessary graphical condition is that every follower node in $\mathcal{D}$ is 2-reachable from a leader node. Also, similar to the single-integrator model, the eigenvalues of system (3.26) may be distributed in the left half complex plane and the right half complex plane such that trajectories of system (3.26) may not converge to the equilibrium formation. Hence, an invertible diagonal matrix D is utilized to assign the eigenvalues of

$$\begin{bmatrix} 0_{n\times n} & I_n \\ -DL & -\gamma I_n \end{bmatrix} \tag{3.30}$$

in the open left half complex plane in addition to two 0 eigenvalues, i.e., to make the n agents asymptotically reach a planar formation F_ξ with the interaction law (3.29). If such a matrix D exists, it is called a *stabilizing matrix*. The following theorem shows the existence of such a matrix.

Theorem 3.7. *Consider a formation basis* $\xi \in \mathbb{C}^n$ *satisfying* $\xi_i \neq \xi_j$ *and suppose a complex Laplacian* L *of the sensing graph* $\mathcal{D}$ *satisfies* $L\xi = 0$ *and* $\det(L_{ff}) \neq 0$*. If there is a permutation matrix* P *such that all leading principal minors of* $PL_{ff}P^T$ *are non-zero, then a stabilizing matrix* D *for system (3.26) exists.*

Proof: Denote

$$A = \begin{bmatrix} 0_{n\times n} & I_n \\ -DL & -\gamma I_n \end{bmatrix} \tag{3.31}$$

Let λ_i be an eigenvalue of A and let $\zeta = [x^T \ y^T]^T$ be the corresponding eigenvector where $x, y \in \mathbb{C}^n$. Then from the equality $A\zeta = \lambda_i\zeta$, it is obtained that

$$y = \lambda_i x, \tag{3.32a}$$

$$-DLx - \gamma y = \lambda_i. \tag{3.32b}$$

Substituting (3.32a) into (3.32b) results in

$$-DLx = (\lambda_i^2 + \gamma\lambda_i)x,$$

which means

$$\lambda_i^2 + \gamma\lambda_i + \sigma_i = 0 \tag{3.33}$$

where σ_i is an eigenvalue of the matrix DL. Since L satisfies $L\xi = 0$ and $\det(L_{ff}) \neq 0$, it is known that DL has two zero eigenvalues for all invertible diagonal matrix D. Without loss of generality, denote $\sigma_1 = \sigma_2 = 0$. For $\sigma_1 = \sigma_2 = 0$, the roots of the characteristic equation (3.33) are

$$\lambda_{i,1} = 0, \quad \lambda_{i,2} = -\gamma < 0, \quad i = 1, 2.$$

Thus, to show the existence of a stabilizing matrix D, it remains to show that σ_i $(i = 3, \ldots, n)$ can be assigned such that the roots of the complex-coefficient characteristic equation (3.33) have negative real part. It has been shown that the roots are in the open half complex plane if and only if

$$\frac{Re(\sigma_i)}{(Im(\sigma_i))^2} > \frac{1}{\gamma^2}.$$

By the assumption that there is a permutation matrix P such that all the leading principal minors of PLP^T are non-zero, then it follows from the same argument as in the Theorem 3.4 that there exists a diagonal matrix M such that the eigenvalues of $MPL_{ff}P^T$ all have positive real parts. Denote the eigenvalues of $MPL_{ff}P^T$ by $\sigma_3', \ldots, \sigma_n'$. Then choose D as

$$D = \begin{bmatrix} I_2 & 0 \\ 0 & \epsilon P^T M P \end{bmatrix} \tag{3.34}$$

where $\epsilon > 0$ is a scalar. Thus, the eigenvalues of DL are

$$\sigma_1 = \sigma_2 = 0, \quad \sigma_i = \epsilon\sigma_i', \quad i = 3, \ldots, n. \tag{3.35}$$

Then it can be checked that for sufficiently small $\epsilon > 0$

$$\frac{Re(\sigma_i)}{(Im(\sigma_i))^2} = \frac{Re(\sigma_i')}{\epsilon(Im(\sigma_i'))^2} > \frac{1}{\gamma^2}, \quad i = 3, \ldots, n. \tag{3.36}$$

Therefore, a stabilizing matrix D is derived, which makes a group of n agents asymptotically reaches the planar formation F_ξ. □

From the proof of theorem 3.7, it is known that a stabilizing matrix can also be obtained for the double-integrator case with a minor modification of Algorithm (3.1). In fact, only the construction of D must be modified.

Algorithm 3.2 Double-Integrator Dynamics

Find a permutation matrix P such that $A = PL_{ff}P^T$ has all non-zero leading principal minors.

for $i = 1, \ldots, n-2$ **do**

Find m_i to assign the eigenvalues of $\text{diag}(m_1, \cdots, m_i)A_{(1\sim i)}$ in the open right half complex plane.

end for

Select an ϵ satisfying conditions in the proof of (3.35)

Construct D according to (3.34).

Remark 3.3. *It has been discussed how a planar formation is achieved by a complex Laplacian based control law. The results can be simply extended to reach and maintain a formation shape while moving. Suppose that the accelerations ,for double integrator dynamics model, of the two co-leaders are synchronized, and say $a_0(t)$. When this synchronized acceleration information is available to all the followers, then the following control law is the adjusted one to reach a formation while moving.*

$$\begin{cases} a_i = -\gamma v_i + a_0(t), & i = 1, 2; \\ a_i = \sum_{j \in N_i^-} w_{ij}(z_j - z_i) - \gamma v_i + a_0(t), & i = 3, \cdots, n. \end{cases} \tag{3.37}$$

When the leaders' acceleration is not accessible by all followers, estimation schemes can be adopted to estimate it and then the leaders' acceleration in (3.37) can be replaced with the estimated one.

Chapter 4

The Isomorphism Problem. Relabelling the Graph nodes

As we have seen in chapter (3), the stability of a multi-agent formation whose single agent is modelled either by a single-integrator kinematics or by a double-integrator dynamics, depends on the eigenvalues of the complex Laplacian matrix that represents the sensing graph modelling the formation. In case of instability, that is, the Laplacian matrix of the systems

$$\dot{z} = -Lz, \tag{4.1}$$

$$\begin{bmatrix} \dot{z} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0_{n\times n} & I_n \\ -L & -\gamma I_n \end{bmatrix} \begin{bmatrix} z \\ v \end{bmatrix}, \tag{4.2}$$

happens to have eigenvalues in the half negative complex plane, by theorem 3.4, theorem 3.5 and theorem 3.7 it is possible to find a complex diagonal matrix D such that the matrix DL has all eigenvalues in the half positive complex plane and systems (4.1) and (4.2) are stable. Thus, the stability of a leader-follower formation can be entirely ensured by finding:

- Single-Integrator Kinematics:

    1. a permutation matrix P (if needed) such that the Laplacian sub-matrix $\hat{L} = PL_{ff}P^T$ has all non-zero principal minors,
    2. a complex diagonal matrix M (if needed) such that the matrix $-DL$ is stable, where D is

$$D = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & P^T M P \end{bmatrix}. \tag{4.3}$$

- Double-Integrator Dynamics:

    1. a permutation matrix P (if needed) such that the Laplacian submatrix $\hat{L} = PL_{ff}P^T$ has all non-zero principal minors,
    2. a complex diagonal matrix M and a scalar $\epsilon > 0$ (if needed) such that the matrix $-DL$ is stable, where D is

$$D = \begin{bmatrix} I_2 & 0 \\ 0 & \epsilon P^T M P \end{bmatrix}. \tag{4.4}$$

It means that central to the stabilization of a formation is to find the permutation matrix P and the diagonal matrix M fitting the given constraints. In the following sections we present some algorithms aimed to solve these problems. In particular, in section 4.1 a general algorithm design technique called backtracking is presented. In section 4.2 two algorithms based on the backtracking design are proposed to solve the permutation matrix problem. Finally, in section 4.3 a comparison between the two proposed algorithms is given. Implementation and testing are realized in **MATLAB®** modelling language.

4.1 Backtracking Algorithm Design Technique

Backtracking (see [27]) represents one of the most general techniques in algorithm design. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. In order to apply the backtracking method, the desired solution must be expressible as an n-tuple $(x_1, \ldots, x_n)$ where x_i are chosen from some finite set S_i. Often the problem to be solved calls for finding one vector which satisfies a criterion function $\mathcal{P}(x_1, \ldots, x_n)$. Sometimes it seeks all such vectors which satisfy $\mathcal{P}$. For example, a simple problem to be solved like sorting integers can be modelled by a backtracking technique. Let us have a vector of n integers, say $a \in \mathbb{N}^n$, the problem is to find a permutation of the a_is elements such that they are sorted in a descending order. To design a backtracking algorithm just take the set S_i as a finite set which includes the integers 1 through n and a criterion function $\mathcal{P}$ of the form $a_i \geqslant a_{i+1}$, for $i = 1, \ldots, n$. Suppose m_i is the size of set S_i ($|S_i| = m_i$). Then there are $m = m_1 m_2 \cdots m_n$ n-tuples which are possible candidates for satisfying the function $\mathcal{P}$. For example, in the sorting problem we have that $m_i = i$, for

$i = 1, \ldots, n$ and consequently $m = n!$. To test all of the possible m solutions of a problem in a brute force approach, would be very time consuming. The backtracking technique algorithm instead, is able to search for the optimal solutions testing for fewer than m n-tuples. The basic idea is to build up the same vector one component at a time and to use modified criterion functions $\mathcal{P}_i(x_1, \ldots, x_i)$ (also called *bounding functions* and indicated by $\mathcal{B}_i$) to test whether the vector being formed has any chance of success. In this way, the algorithm realizes if a partial vector either can or cannot lead to an optimal solution. If the partial vector does not satisfy the bounding function $\mathcal{B}_i$, then it cannot lead to an optimal solution and $m_{i+1} \ldots m_n$ possible test vectors can be ignored entirely. In the design of a backtracking algorithm is often required that the solutions satisfy a complex set of constraints. This set can be usually divided into two categories:

- **explicit constraints** are rules which restrict each x_i to take on values only from a given set. For instance, in the sorting problem we have seen earlier, an explicit constraint is to take values from the set $S_i = \{1, \ldots, n\} - \{x_1, \ldots, x_{i-1}\}$. The explicit constraints may or may not depend on the particular instance I of the problem being solved,

- **implicit constraints** describe the way in which the x_i *must relate to each other.*

Explicit and implicit constraints are related to each other. In fact, explicit constraints define a possible solution space I of tuples, while implicit constraints determine which of the tuples in I actually satisfy the criterion function. From that point of view, a backtracking algorithm determines problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space. Many tree organizations may be possible for I. For example, we can see two trees for a sorting problem of $n = 3$ integers in figure (4.1). Broadly speaking, each node in the tree defines a problem state. All paths from the root to other nodes define the state space of the problem. *Solution states* are those problem states S for which the path from the root to S defines a tuple in the solution space. In the tree of figure (4.1a) only the leaf nodes are solution states. *Answers states* are those solution states S for which the path from the root to S defines a tuple which is a member of the set of solutions (i.e., it satisfies the implicit constraints) of the problem. The tree organization of the solution space will be referred to as the *state space tree.* The example in figure (4.1a) is

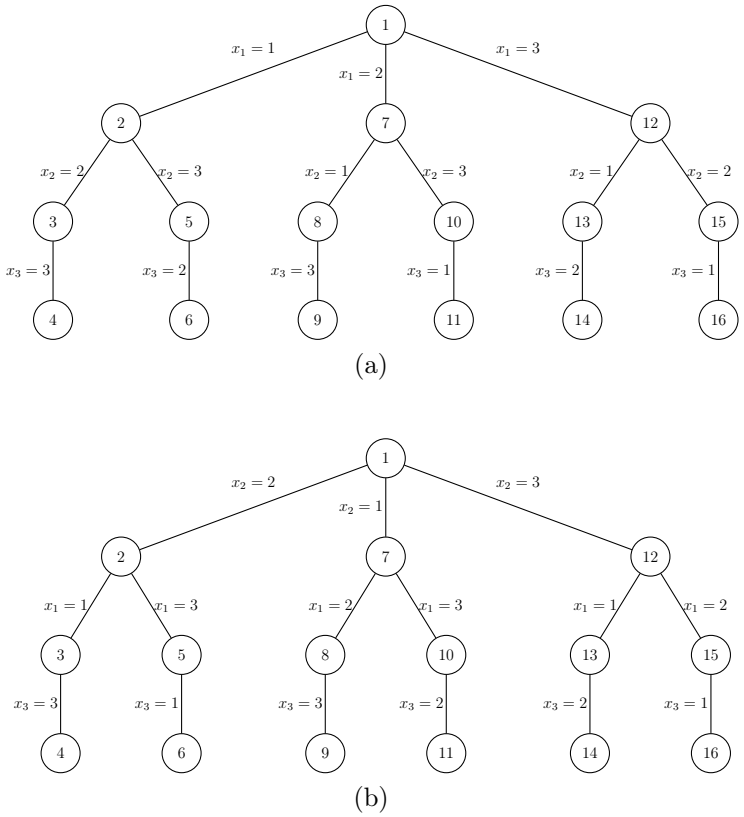


Figure 4.1: Example of two different tree organizations for a solution space in a backtracking algorithm. The problem being represented is the sorting of 3 positive integers.

called a *static tree* because the tree organization is independent of the problem instance being solved. When the tree organization is determined dynamically as the solution space is being searched, then it is called *dynamic tree*. In this case, it depends on the problem instance. For example, if the one instance of the sorting problem is solved with the tree organization of figure (4.1a) and a second instance with the tree organization of figure (4.1b) then the backtrack algorithm has been using a dynamic tree.

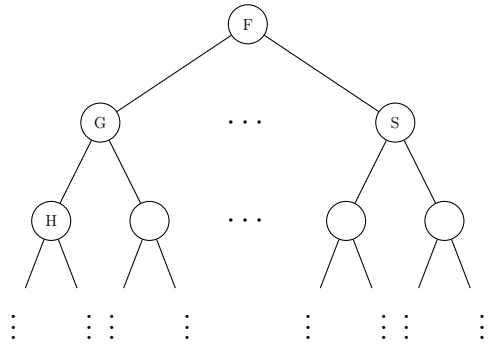
Once a state space tree has been conceived of for any problem, this problem may be solved by systematically generating the problem states, determining which of these are solution states and finally determining which solution states are answer states. The generation process begins with the root node and goes further with his children and so on, until the state space has been completely represented. In particular, there are three kind of nodes:

- a *live node* is a node that has been generated and all of whose children has not yet been generated,
- a *E-node* (node being expanded) is a live node whose children are cur-

rently being generated,

- a *dead node* is a generated node that is neither not to be expanded further nor one for which all of its children have been generated.

For backtracking algorithms, nodes of the state space tree are generated in a *depth first* way. In fact, as soon as a new child H of the current E-node G is generated, this child will become the E-node. G will become the E-node again when the sub-tree H has been fully explored.



For example, let us consider one of the state space tree of figure (4.1), where it can be easily seen that nodes are labelled as in depth first search. Consider node 2. This node is a live node as soon as it is generated. When node 3 is generated, it becomes a live node itself while node 2 becomes an E-node. Because the three is visited in a depth first way, node 3 is the next node to consider. Then it becomes an E-node while its first child is being generated and so on. After the sub-tree 3 has been visited, node 2 becomes an E-node again and so on until all its children have been generated. When the next children of node 1 is considered, that is node 7, node 2 becomes a dead node.

In this process bounding functions will be used to kill live nodes without generating their children. A depth first-like problem generally uses a stack to keep track of the visited or the generated nodes. Hence, a recursive definition of the backtracking algorithm is more natural. Let $(x_1, x_2, \ldots, x_i)$ be a path from the root to a node in a state space tree. Let $T(x_1, x_2, \ldots, x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2, \ldots, x_{i+1})$ is also a path to a problem state. We assume the existence of bounding functions $\mathcal{B}_{i+1}$ (expressed as predicates) such that $\mathcal{B}_{i+1}(x_1, x_2, \ldots, x_{i+1})$ is false for a path $(x_1, x_2, \ldots, x_{i+1})$ from the root node to a problem state only if the path cannot be extended to reach an answer node. Thus, the candidates for position $i+1$ of the solution vectors $x = [x_1, \ldots, x_n]$ are generated by T and satisfied by $\mathcal{B}_{i+1}$. The pseudo-code version of the backtracking can be seen in algorithm 4.1. In the

Algorithm 4.1 Recursive Backtracking Algorithm

```
  recursiveBACKTRACK(k)
Require: global n, x
  for each x_k such that x_k ∈ T(x_1, ..., x_{k-1}) and B_k(x_1, ..., x_k) = true do
    if (x_1, ..., x_k) is a path to an answer node then
      A ← x
      call recursiveBACKTRACK(k + 1)
    end if
  end for
```

backtrack implementation, A is the set of all n-tuples answer to the problem. Note that the solution vector x is treated as a global array. Each recursive call adjoins to the current vector an element and verifies the current bounding function over that. For example, let us consider the k-th call. The k-th element x_k is adjoined to the current tuple $(x_1, x_2, \ldots, x_{k-1})$ and the bounding function $\mathcal{B}_k$ is verified. The kth possible elements are generated one by one at the k-th recursive call. If $\mathcal{B}_k$ is true the next recursive call is made, otherwise the current call keeps on adjoining x_ks possible elements and checking $\mathcal{B}_k$. If no suitable x_k is found, the for loop is exited without invoking the algorithm again, and the last unresolved call is resumed. If a suitable x_k is found, a new call over the tuple $(x_1, x_2, \ldots, x_k)$ is made and the $k+1$th element is searched for. When k exceeds n, $T(x_1, x_2, \ldots, x_{k-1})$ returns an empty set and hence the for loop is never entered. Thus, no more recursive calls are possible, only the resume of unresolved ones.

Remark 4.1. *The algorithm search for all the actual solutions of the problem. In order to have only a single solution, a flag can be added as a parameter to indicate the first occurrence of success. Thus, the unresolved calls can be exited and the program terminated.*

4.1.1 Backtracking Efficiency

The importance of backtracking lies in its ability to solve some instances with large n in a very small amount of time. The only difficulty is in predicting the behavior of the algorithm for the problem instance we wish to solve. The efficiency of a backtracking algorithm depends on 4 main factors:

1. the time to generate the next x_k,
2. the number of x_k satisfying the explicit constraints,
3. the time for the bounding functions $\mathcal{B}_i$,

4. the number of x_k satisfying the $\mathcal{B}_i$ for all i.

In order to decrease the time needed to compute the solutions, we can try to reduce the number of generated nodes by efficient bounding functions. Nonetheless, the most times, a more efficient bounding function means a function that needs a higher computing time to evaluate. However, what is desired is a reduction in the overall execution time and not just a reduction in the number of nodes generated. Hence, a balance between 3 and 4 should be searched for.

Once a state space organization tree is selected, the first three of the mentioned factors are relatively independent of the problem instance being solved. Only the number of nodes generated (fourth condition) varies from one problem instance to another. Hence, the worst case time for a backtracking algorithm will generally depend on condition number 4). In fact,

1. if the number of nodes generated is 2^n, then

$$\mathrm{O}(\mathrm{p}(n)2^n); \tag{4.5}$$

2. if the number of nodes generated is $n!$, then

$$\mathrm{O}(\mathrm{q}(n)n!). \tag{4.6}$$

Note that $\mathrm{p}(n)$ and $\mathrm{q}(n)$ are polynomials in n.

Generally speaking, the main issue with predicting the algorithm behavior is due to the nature of the problem we want ot solve. In fact, the position and the number of the actual solutions in the state space tree is unpredictable. When we are searching for all the solutions, only the number of them affects the algorithm performance. On the other hand, if we are searching for the first occurrence, solutions' position could be of great significance, since the choice of a state space organization tree could degrade or improve algorithm efficiency.

4.1.2 Examples of Backtracking Design

We can see how a backtrack algorithm works by mean of examples. The technique will be applied to solve the problem of sorting the elements of a vector and the 8-Queens problem. The different nature of these problems shows the flexibility of the backtracking design, despite the fact that backtracking itself is not the algorithm of choice for those problems. Before proceeding with the examples, let us have a summary of the main parts of a backtracking design.

- S is the set of the explicit constraints while $S_k \subseteq S$ is the subset of the explicit constraints at the k-th step.
- $\mathcal{B}$ is the criterion function while $\mathcal{B}_k$ is the bounding function at the k-th step.
- $T(x_1, \ldots, x_{k-1})$ is the function which supplies candidates at step k given the explicit constraints satisfied in the previous $k-1$ steps.

Sorting a vector of positive integers

Let $\mathbf{a} = [a_1\ a_2\ a_3\ a_4]^T$ be a vector of $n = 4$ positive integers, that is $\mathbf{a} \in \mathbb{N}^4$. We want to sort the elements of $\mathbf{a}$ in an ascending order (from the smallest to the biggest). The backtracking procedure will be modelled as follows:

- the set of the explicit constraints will be the set of the elements' position in vector $\mathbf{a}$. Then,
$$S = \{1, 2, 3, 4\}.$$
At each step k of the algorithm, the explicit constraints will then be $S_k \subseteq \{1, 2, 3, 4\}$;
- $T(x_1, \ldots, x_{k-1})$ supplies at each step k a node to be examine. In this case, the generic x_i is the position of a vector element;
- the criterion function to satisfy is $\mathcal{B} = \{a_i \leq a_{i+1}\}$ for $i = 1, \ldots, 4$, while the bounding function at the generic k-th step is $\mathcal{B}_k = \{a_k \geqslant a_{k-1}\}$, where $a_k(x_k) \mid x_k \in T(x_1, \ldots, x_{k-1})$.

Let us have $\mathbf{a} = \begin{bmatrix} 7 \\ 1 \\ 4 \\ 3 \end{bmatrix}$. The way in which function T supplies candidates determines the state space organization tree of the backtracking instance. Suppose that T supplies the positions of the elements such that the state space tree will be the one represented in fig.(4.2). The search for the solution will then explore that tree in a depth first way, supplying a step-by-step algorithm as follows:

$\boxed{\text{K} = 1}$

$\rightarrow x_1 = 1$

$a_1(1) = 7$

$\boxed{\text{K} = 2}$

$\rightarrow x_2 = 2$

$a_2(2) = 1 \geqslant a_1(1) \rightarrow$ **FALSE** $\rightarrow$ sub-tree 3 **discharged**

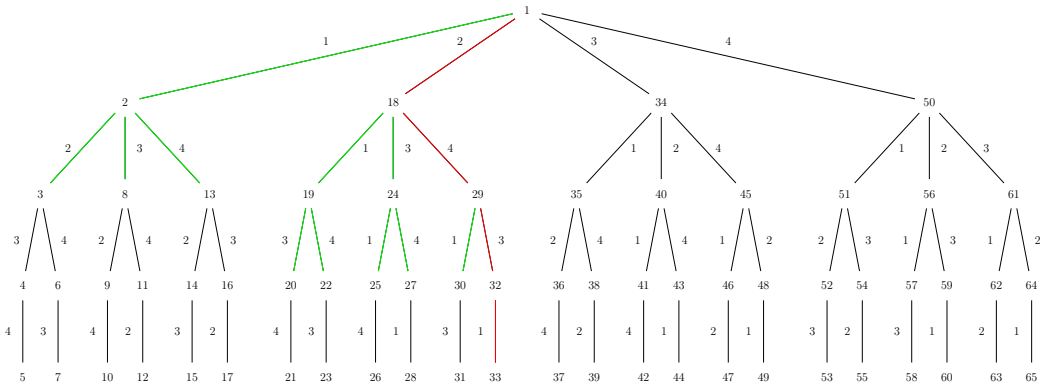


Figure 4.2: Example of a tree organization for the solution space of a sorting problem. The green branches are the visited ones, while the red path is a path to a solution state.

$\rightarrow x_2 = 3$

$a_2(3) = 4 \geqslant a_1(1) \rightarrow$ **FALSE** $\rightarrow$ sub-tree 8 **discharged**

$\rightarrow x_2 = 4$

$a_2(4) = 3 \geqslant a_1(1) \rightarrow$ **FALSE** $\rightarrow$ sub-tree 13 **discharged**

$T =$ empty, **BACK-STEP**

K = 1

$\rightarrow x_1 = 2$

$a_1(2) = 1$

K = 2

$\rightarrow x_2 = 1$

$a_2(1) = 7 \geqslant a_1(2) \rightarrow$ **TRUE** $\rightarrow$ **NEXT CALL**

K = 3

$\rightarrow x_3 = 3$

$a_3(3) = 4 \geqslant a_2(1) \rightarrow$ **FALSE** $\rightarrow$ sub-tree 20 **discharged**

$\vdots$

Without analysing each step of the algorithm, we can easily see that the actual solution is $x = (2, 4, 3, 1)$, such that $\hat{\mathbf{a}} = \begin{bmatrix} 1 \\ 3 \\ 4 \\ 7 \end{bmatrix}$. As it can be seen from the first steps, the algorithm has been able to discharge sub-trees 3, 8 and 13 without visiting the whole sub-trees and finally find the solution in sub-tree 18 (red path).

The problem of sorting a vector of elements is a problem which has only one solution. Hence, not only is the algorithm performance affected by the number of nodes generated during the calls but also by the position of the solution

itself. In the example we have seen, a different state space tree organization could have led to the solution faster. The main issue is that the position of the solution is unpredictable and heuristics are then required in order to learn the characteristic behavior of a specific problem.

The 8-Queens problem

The 8-queens (see [27]) is a classical combinatorial problem where 8 queens must be placed on an 8×8 chessboard so that no two *attack*, that is no two of them are on the same row, column or diagonal. In figure (4.3) a solution for the problem is shown. The chessboard is modelled like a matrix where row and column are numbered from 1 to 8. The queens themselves can be numbered from 1 to 8. Since each queen must be on a different row, we can assume that queen i is to be placed on row i. All solutions to the 8-queens problem can therefore be represented as 8-tuples $(x_1, \ldots, x_8)$ where x_i is the column on which queen i is placed. Using this formulation, the explicit constraints are

$$S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}, \quad 1 \leq i \leq n.$$

Hence, the solution space consists of 8^8 8-tuples.

The *implicit constraints* for this problem are:

1. *no two x_i's can be the same* (that is, *all queens must be in a different column*);

2. *no two queens can be on the same diagonal.*

The first constraint implies that all solutions are permutations of the 8-tuple $(1, 2, 3, 4, 5, 6, 7, 8)$. Hence, the size of the solution space reduces from 8^8 tuples to 8! tuples. The second constraint can be expressed in mathematical terms modelling the chessboard as a two dimensional array, where each row and column can be enumerated like in figure (4.3). In that way, we can see that for every element on the same diagonal which runs from the bottom left to the top right, each element has the same $row - column$ value. Also, every element on the same diagonal which goes from the lower right to the upper left has the same $row + column$. Suppose two queens are placed at positions (i, j) and (k, l). Then by the above they are on the same diagonal only if

$$i - j = k - l \quad \text{or} \quad i + j = k + l.$$

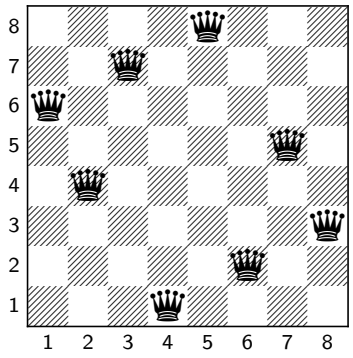


Figure 4.3: Example of a solution for the 8-queens problem.

The first equation implies

$$j - l = i - k,$$

while the second implies

$$j - l = k - i.$$

Therefore two queens lie on the same diagonal if and only if

$$|j - l| = |i - k|. \tag{4.7}$$

Implicit constraints are the bounding functions that are used to verify at each step that the *non attack* configuration holds.

As an example, let us see some steps for an easier instance of the problem, the 4-queens, represented in figure (4.4). The first step (fig. 4.4a) places the first queen in position $(1, 1)$ and a second call is made. The second call (fig. 4.4b) tries to place the second queen in positions $(2, 1)$ and $(2, 2)$ but they are forbidden by the bounding functions. The second call ends to place the queen in position $(2, 3)$. The third call (fig. 4.4c) tries all the positions in row 3 without any success and the backtrack process ends the current call and resumes the last one. The position of queen 2 is then changed (fig. 4.4d), and she is moved in $(2, 4)$ while the next call is finally able to place the third queen in $(3, 2)$. Then the fourth call is made, but no queen can be placed in row 4. Since the other queens cannot be moved further, the backtrack ends the last 3 calls and move back resuming the first one. The algorithm continues as described until a solution is reached. For example, a feasible solution is $x = (2, 4, 1, 3)$.

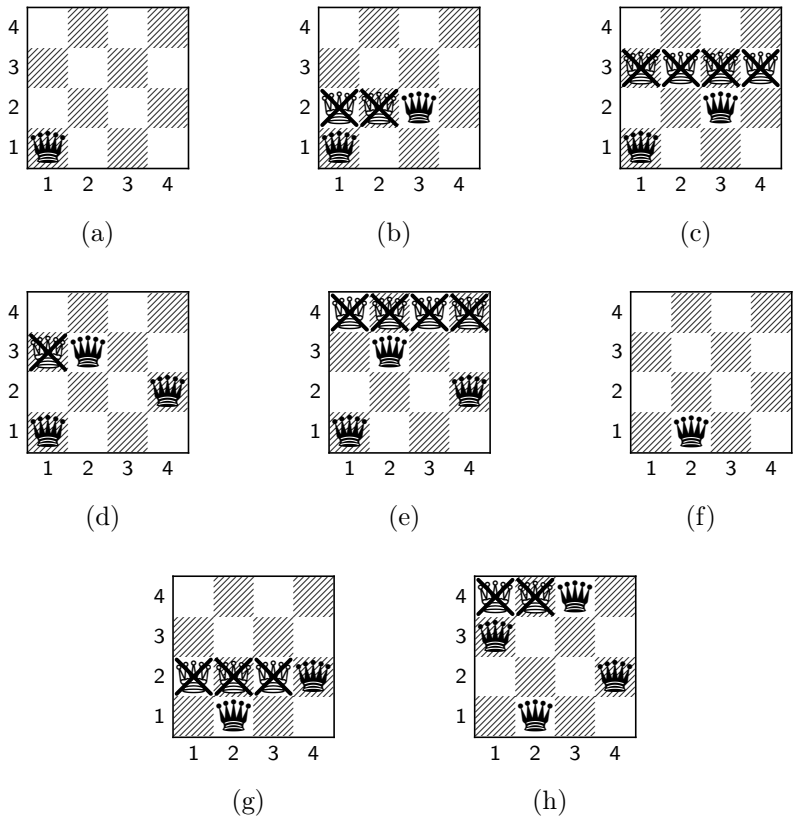


Figure 4.4: Some steps for the 4-queens problem.

4.2 The permutation matrix P

The problem of finding a permutation matrix P such that the matrix $\hat{L}_{ff}$ given from the equation

$$\hat{L}_{ff} = PL_{ff}P^T \tag{4.8}$$

has all non-zero leading principal minors, can be solved by an algorithm designed with a backtracking technique (see sec. 4.1). To understand how this can be made, we can rewrite equation (4.8) as follows:

$$\hat{L}_{ff} = \begin{bmatrix} e_i^T \\ e_j^T \\ e_h^T \\ \vdots \\ e_s^T \end{bmatrix} L_{ff} \begin{bmatrix} e_i & e_j & e_h & \cdots & e_s \end{bmatrix} \tag{4.9}$$

where the generic vector $e_i \in \mathbb{R}^n$ is the ith column vector of the identity matrix I, that is the ith orthonormal vector of the canonical base of $\mathbb{R}^n$ vector space. As it can be seen from equation (4.9), the first row of P is equal to the first column of P^T, the second row is equal to the second column and so on. Hence, we can enumerate the rows of P and consequently the columns of P^T which will have the same sequence of numbers. If the ordered set of rows of P is $(i, j, h, \ldots, s)$ so the ordered set of column of P^T will be $(i, j, h, \ldots, s)$, that is the same. The problem of finding P could be seen as the problem of finding the right sequence of numbers representing the vectors e_i-th such that, the permutation matrix P obtained, allows constraints over $\hat{L}_{ff}$ to hold. That is the starting point for the backtracking algorithm. In fact, sets S_i of explicit constraints for backtracking the solution, are sets of real numbers chosen through 1 to n, where n is the order of the permutation matrix P. Then,

$$S_i = \{1, \ldots, n\}, \quad 1 \leq i \leq \text{max-steps}. \tag{4.10}$$

For the current problem, the overall objective, that is, the *criterion function* $\mathcal{P}(x_1, x_2, \ldots, x_n)$ where $x_i \in \{1, 2, \ldots, n\}$, is to obtain a matrix $\hat{L}_{ff}$ with all non-zero leading principal minors. In the following subsections we will see two methods for verifying the determinant of $\hat{L}_{ff}$ sub-matrices and some different T function used to learn the behavior of the backtracking with the problem being solved.

4.2.1 Computing the Determinant.Algorithm 1

The bounding functions $\mathcal{B}_i$ check for the determinant of the leading principal minor of order i at each i-th step. They can be written in predicate form as follows:

$$\mathcal{B}_i = \{\det(\hat{L}_{ff[1\sim i]}) \neq 0 == TRUE?\}. \tag{4.11}$$

For example, consider a generic step, say the k-th step. The algorithm searches for a feasible x_k to add at the already obtained partial solution $(x_1, x_2, \ldots, x_{k-1})$. When a feasible x_k is added at the $k-1$-tuple, the upper part of $\hat{L}_{ff}$ can be built using rows and columns of L_{ff} addressed by the values of the x_is in the tuple. The bounding function $\mathcal{B}_k$ checks for the determinant of the leading principal minor $\hat{L}_{ff[1\sim k]}$ then. If $\det \hat{L}_{ff[1\sim k]} \neq 0$ then we can proceed to the next step. On the other hand, if $\det \hat{L}_{ff[1\sim k]} = 0$ for all feasible x_k, the backtrack ends the current step and resumes the $k-1$-th step. In resuming the previous call, a new x_{k-1} is chosen from $T(x_1, x_2, \ldots, x_{k-2})$, that is a not yet used row of L_{ff}, and the algorithm verifies again the bounding function. If the bounding function is true, then the k-th step can be reached again, otherwise a new x_{k-1} should be still searched for the $k-1$-th step. The process, described in algorithm 4.2 is the same for each step. The algorithm has been designed

Algorithm 4.2 Permutation Matrix Solver

```
  recursiveBACKTRACK(k)
Require: global n, b, p
  for each x_k such that x_k ∈ T(x_1, ..., x_{k-1}) and B_k = {det(L̂_{ff[1~k]}) ≠ 0} →
  true do
    if (x_1, ..., x_k) is a path to an answer node then
      b ← x
      if k == n then
        p = 1
        END CALL
      end if
      call recursiveBACKTRACK(k + 1)
    end if
    if p == 1 then
      END CALL
    end if
  end for
```

in order to search for the first occurrence of a solution. In fact, when the first node answer is reached in the solution tree space, the algorithm ends. A flag is raised and each of the previous non-terminated calls are ended. As it can

be seen from algorithm 4.2 the flag used is the boolean variable p that is set to 1 only when the answer is found.

The solution is the n-tuple of the first n real numbers, ordered as we need to order the rows of an identity matrix I to obtain the permutation matrix P we are searching for. For instance, let us have $n = 5$ and a solution like $\{2, 3, 4, 1, 5\}$. That means that the permutation matrix we need has the second row of I as first, the third row of I as second, and so on. The algorithm tracks the solution by mean of the vector b whose elements are the integers from 1 to n which are ordered call after call.

Efficiency

The algorithm performance depends largely on the bounding functions, which compute the determinant of each $\hat{L}_{ff}$ sub-matrix.

Provided that a solution is a n-tuple of integers from 1 to n ordered in a specific manner, the solution space will have $n!$ possible solutions to explore. In fact, the way in which n different numbers can be combined is exactly the factorial of n. Hence,the algorithm ,as a worst case, must check all the $n!$ nodes of the state space tree and its overall complexity is

$$\mathrm{O}\left(\mathrm{p}(n)n!\right), \tag{4.12}$$

where $\mathrm{p}(n)$ is a polynomial in n that takes account of nodes generation, x_ks generation and bounding functions evaluation (see sec. 4.1). In this case we assume that only $\mathcal{B}_i$s evaluation accounts for the expression of $\mathrm{p}(n)$. It seems a reasonable simplification given that for each bounding function we have to compute the determinant of a matrix. In fact, the complexity to evaluate the determinant of a matrix of order n is at least $\mathrm{O}(\frac{2}{3}n^3)$, significantly more than the necessary time to make the algorithm pick numbers from a given set. In order to find a solution, we have to evaluate the determinant of all leading principal minors, and it means that $\mathrm{p}(n)$ is the sum of the operations needed to accomplish that. We can write in formulas:

$$\mathrm{p}(n) = \frac{2}{3}(1^3 + 2^3 + 3^3 + \cdots + n^3) = \frac{2}{3}\sum_{i=1}^{n} i^3 \tag{4.13}$$

Equation 4.13 can be written in a closed form by Faulhaber's formula, that for

the case of the sum of cubes takes the form (see appendix D and [41]):

$$\sum_{i=1}^{n} i^3 = \frac{1}{4}(n^4 + 2n^3 + n^2). \tag{4.14}$$

The polynomial $\mathrm{p}(n)$ can then be written as

$$\mathrm{p}(n) = \frac{1}{6}(n^4 + 2n^3 + n^2), \tag{4.15}$$

and, can be expressed asymptotically as

$$\mathrm{O}\left(\mathrm{p}(n)\right) = \mathrm{O}\left(\frac{1}{6}n^4\right). \tag{4.16}$$

Thus, the overall complexity of the backtrack algorithm for the calculus of the permutation matrix P is

$$\mathrm{O}\left(\mathrm{p}(n)n!\right) = \mathrm{O}\left(\frac{1}{6}n^4 n!\right), \tag{4.17}$$

that is higher than a simply factorial time complexity. In order to have better performance, what can be changed is the function T. In fact, T supplies, at each call, the x_i to be added to the path already done. Since we are searching for the first occurrence of the solution, different ways of adding x_i to the tuple $(x_1, \ldots, x_{i-1})$ could result in a better performance for the backtracking algorithm, that is a solution could be found earlier.

Experimental Results

Experimental results in [29] have shown that, for randomly generated Laplacian matrices, the most times the solution is an identity matrix. It means that, the randomly generated Laplacian matrix has already all non-null principal minors. Then, a function T which takes account of this fact would be preferable. In order to study the efficiency of the algorithm another function T has been chosen. The two different T used are:

- T_{d1}, which supplies values x_i in order to search for the nearest solution to the identity matrix. That is, if $x_{k-1} = 5$ the first value supplied at the k-th call will be $x_k = 6$. Then, if the bounding function is not verified, the next value will be $x_k = 7$ and so on. As it will be seen this function T is the one which has better results, according to [29];

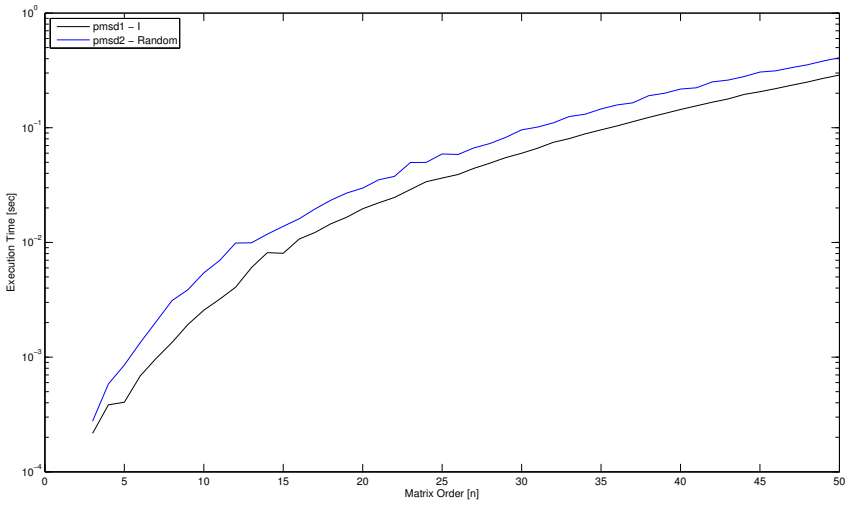


Figure 4.5: Comparison between different state space organization trees in the determinant-based backtracking algorithm. Black line represents algorithm with T_{d1} while blue line represents algorithm with T_{d2}. Function T_{d1} makes the algorithm a little faster.

- T_{d2}, which supplies values x_i randomly picking them from the set of the explicit constraints S_i until all values are tried. That is, at the k-th call $x_k \in S_k = S - (x_1, \ldots, x_{k-1})$.

Note that T_{d1} implements a static state space tree organization while T_{d2}, because of the randomness of the search, implements a dynamic state space tree organization. In fact, T_{d2} changes the state space tree organization for each instance of the problem.

In figure (4.5) a comparison between the two backtracking implementations with different Ts is shown. Algorithm which implements T_{d1} (black line) results to be the fastest of the two. We can explain that by mean of a simple example. Let us have the following real non-singular matrix:

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 3 & 2 & 1 \end{bmatrix}.$$

Its leading principal minors are

$$A_1 = 1,$$
$$A_2 = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix},$$

which are non-singular as well since $\det(A_1) = 1 \neq 0$ and $\det(A_2) = 1 \neq 0$. In this case the two implementations of the algorithm will lead to different solutions, because of the different search methods.

For T_{d1}, the algorithm will lead to the solution

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

that is, as expected, the identity matrix since all leading principal minors are already non-null.

For T_{d2} the algorithm will lead to the solution

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix},$$

that is a solution as well, since $\hat{A} = PAP^T$ has all non-null leading principal minors.

The point is that the first algorithm has to compute a smaller number of determinants in searching for the solution because it searches for the nearest solution to the identity matrix. Since the experiments showed that randomly generated complex Laplacian matrices for the most times do not need any permutation of rows and columns, then algorithm which implements T_{d2} results to be slower because of the higher number of bounding functions evaluations.

4.2.2 Gauss Elimination Method.Algorithm 2

The permutation matrix problem solved with a determinant-based backtracking algorithm has a worst case complexity of $O\left(p(n)n!\right) = O\left(\frac{1}{6}n^4 n!\right)$, as we have already seen. Even though this is an upper bound and is reached only if all combinations are tried (i.e. for a singular matrix); a more fast algorithm would be preferable. We can have a better worst case complexity by modifying the bounding functions B_k used at each call k. To do so, we need to analyse the automorphism equation

$$\hat{L}_{ff} = PL_{ff}P^T. \tag{4.18}$$

In subsections 2.3.2 and 2.3.3 we have seen the Gauss elimination method with total pivoting and the correspondent LU factorization, which have the general expression

$$LU = PAQ, \tag{4.19}$$

where

- $L \in \mathbb{C}^{n\times n}$ is a complex unit lower triangular matrix,
- $U \in \mathbb{C}^{n\times n}$ is a complex upper triangular matrix,
- $A \in \mathbb{C}^{n\times n}$ is a complex square matrix,
- $P, Q \in \mathbb{R}^{n\times n}$ are two permutation matrices.

If we compare equations (4.18) and (4.19) together, we can say that the first one could be seen as a special case of an LU factorization with total pivoting

$$\hat{L}_{ff} = \hat{L}_g \hat{U}_g = P L_{ff} P^T, \tag{4.20}$$

where $Q = P^T$. The question is how we can benefit from the LU factorization of the relabelled Laplacian sub-matrix. Rewrite factors $\hat{L}_g$ and $\hat{U}_g$ as block matrices

$$\begin{aligned} \hat{L}_g &= \begin{bmatrix} \hat{L}_g^{(k)} & \mathbf{0} \\ \mathbf{w}^T & \hat{l}_{nn} \end{bmatrix}, \\ \hat{U}_g &= \begin{bmatrix} \hat{U}_g^{(k)} & \mathbf{p} \\ \mathbf{0}^T & \hat{u}_{nn} \end{bmatrix} \end{aligned}, \tag{4.21}$$

where $\mathbf{0}, \mathbf{w}, \mathbf{p}$ are vectors and $\hat{L}_g^{(k)}, \hat{U}_g^{(k)} \in \mathbb{C}^{(k\times k)}$ are the leading principal minors of order k of $\hat{L}_g$ and $\hat{U}_g$. The block product between the two factors results in

$$\hat{L}_{ff} = \hat{L}_g \hat{U}_g = \begin{bmatrix} \hat{L}_g^{(k)} \hat{U}_g^{(k)} & \hat{L}_g^{(k)} \mathbf{p} \\ \mathbf{w}^T \hat{U}_g^{(k)} & \mathbf{w}^T \mathbf{p} + \hat{l}_{nn} \hat{u}_{nn} \end{bmatrix}, \tag{4.22}$$

where it can be easily seen that $\hat{L}_{ff}^{(k)} = \hat{L}_g^{(k)} \hat{U}_g^{(k)}$ is the leading principal minor of order k of the relabelled Laplacian sub-matrix. Since $\hat{L}_g$ is a unit lower triangular matrix, so are its leading principal minors. Then, the determinant of $\hat{L}_{ff}^{(k)}$ is

$$\det\left(\hat{L}_{ff}^{(k)}\right) = \det\left(\hat{L}_g^{(k)}\right) \det\left(\hat{U}_g^{(k)}\right) = \det\left(\hat{U}_g^{(k)}\right), \tag{4.23}$$

where $\det\left(\hat{L}_g^{(k)}\right) = 1$. Moreover, $\hat{U}_g$ is an upper triangular matrix and so are its leading principal minors. Then, we can write

$$\det\left(\hat{L}_{ff}^{(k)}\right) = \prod_{i=1}^{k} u_{ii}, \tag{4.24}$$

which is valid for $k = 1, \ldots, n$. Equation (4.24) shows that the determinant of each principal minor (and of the entire matrix) can be computed as the product of the diagonal elements of the factor matrix $\hat{U}_g$. Therefore, a leading principal minor of $\hat{L}_{ff}$ has non-zero determinant if and only if all diagonal elements of the corresponding leading principal minor of $\hat{U}_g$ are non-null. It means that, the problem of finding a permutation matrix P such that $\hat{L}_{ff}$ has all non-zero leading principal minors can be formulated as the problem of finding a permutation matrix P such that LU factorization (4.20) exists. The equivalence of the two problems is ensured by theorem (2.5) whose proof can be found in section 2.3.

Theorem 4.1. *Let $A \in \mathbb{F}^{(n\times n)}$ be a non-singular matrix, and let A_k be its leading principal minors of order k. If A_k is non-singular for $k = 1, \ldots, n-1$ then the LU factorization of A exist and is unique.*

The next question that needs an answer is what happens to the elements of L_{ff} after the operation of permutation. This is important in order to define the bounding functions for the backtracking procedure.

In section 2.3 we have seen two main types of LU factorizations:

- LU factorization with partial pivoting, where pivot elements are chosen among the elements in the sub-columns under the main diagonal,

- LU factorization with total pivoting, where pivot elements are chosen in the sub-matrix that has not been triangulated yet.

In the case of equation (4.20) we are wondering where the pivot elements are chosen from. To have an answer, we need to do some mathematics. Permutation matrix P can be written in terms of its row vectors and P^T in terms of its column vectors. Moreover, the Laplacian sub-matrix too can be written in terms of its column vectors as

$$L_{ff} = [l_1\ l_2\ l_3 \cdots\ l_n], \tag{4.25}$$

where $l_i \in \mathbb{C}^n$ for $i = 1, \ldots, n$. Thus, equation (4.18) can be written as follows

$$\hat{L}_{ff} = \begin{bmatrix} e_i^T \\ e_j^T \\ e_h^T \\ \vdots \\ e_s^T \end{bmatrix} \begin{bmatrix} l_1 & l_2 & l_3 & \cdots & l_n \end{bmatrix} \begin{bmatrix} e_i & e_j & e_h & \cdots & e_s \end{bmatrix}. \tag{4.26}$$

By multiplying the first two matrices, the following partial result is obtained

$$\begin{aligned} \hat{L}_{ff} &= \begin{bmatrix} e_i^T l_1 & e_i^T l_2 & e_i^T l_3 & \cdots & e_i^T l_n \\ e_j^T l_1 & e_j^T l_2 & e_j^T l_3 & \cdots & e_j^T l_n \\ e_h^T l_1 & e_h^T l_2 & e_h^T l_3 & \cdots & e_h^T l_n \\ \vdots & & & & \vdots \\ e_s^T l_1 & e_s^T l_2 & e_s^T l_3 & \cdots & e_s^T l_n \end{bmatrix} \begin{bmatrix} e_i & e_j & e_h & \cdots & e_s \end{bmatrix} \\ &= \begin{bmatrix} \bar{l}_{i1} \\ \bar{l}_{j2} \\ \bar{l}_{h3} \\ \vdots \\ \bar{l}_{sn} \end{bmatrix} \begin{bmatrix} e_i & e_j & e_h & \cdots & e_s \end{bmatrix}. \end{aligned} \tag{4.27}$$

Then, matrix $\hat{L}_{ff}$ results to be

$$\hat{L}_{ff} = \begin{bmatrix} \bar{l}_{i1} e_i & \bar{l}_{i1} e_j & \bar{l}_{i1} e_h & \cdots & \bar{l}_{i1} e_s \\ \bar{l}_{j2} e_i & \bar{l}_{j2} e_j & \bar{l}_{l2} e_h & \cdots & \bar{l}_{j2} e_s \\ \bar{l}_{h3} e_i & \bar{l}_{h3} e_j & \bar{l}_{h3} e_h & \cdots & \bar{l}_{h3} e_s \\ \vdots & & & & \vdots \\ \bar{l}_{sn} e_i & \bar{l}_{sn} e_j & \bar{l}_{sn} e_h & \cdots & \bar{l}_{sn} e_s \end{bmatrix}. \tag{4.28}$$

Recalling the properties of vectors $e_i \in \mathbb{R}^n$, we have that

$$\left.\begin{aligned} \hat{l}_{ii} = \bar{l}_{ki} e_k \quad &= e_k^T l_{ki} e_k = l_{kk} \\ \hat{l}_{ij} = \bar{l}_{kj} e_w \quad &= e_k^T l_{kj} e_w = l_{kw} \end{aligned}\right\} \quad \text{elements of } \hat{L}_{ff}, \tag{4.29}$$

and matrix $\hat{L}_{ff}$ results to be

$$\hat{L}_{ff} = \begin{bmatrix} l_{ii} & l_{ij} & l_{ih} & \cdots & l_{is} \\ l_{ji} & l_{jj} & l_{jh} & \cdots & l_{js} \\ l_{hi} & l_{hj} & l_{hh} & \cdots & l_{hs} \\ \vdots & & & \ddots & \vdots \\ l_{si} & l_{sj} & l_{sh} & \cdots & l_{ss} \end{bmatrix} \tag{4.30}$$

The diagonal elements remain along the diagonal even after the pivoting operation. The expression $\hat{L}_{ff} = PL_{ff}P^T$ can then be seen as a *Gauss elimination with diagonal pivoting*. Now we are ready to define the backtracking algorithm for the Gaussian elimination case.

Algorithm 4.3 explains how the backtrack process with Gaussian elimination works. The new bounding functions appear to be simply predicates without any additional computational effort. This is not quite true since the k-th step of Gaussian elimination modifies the elements of the sub-matrix $L_{ff[k\sim n]}$, that is, the pivot candidates for the $k+1$ call as well. Hence, bounding functions can be seen as functions which operate in two steps throughout two consecutive calls. During a generic call $k-1$, the algorithm modifies the pivot candidates and at call k one of them is chosen.

Algorithm 4.3 Permutation Matrix Solver with Gaussian Elimination

```
  recursiveBACKTRACK(k)
Require: global n, b, p
  for each x_k such that x_k ∈ T(x_1, ..., x_{k-1}) and B_k = {û_kk ≠ 0} → true
  do
    if (x_1, ..., x_k) is a path to an answer node then
      b ← x
      EXECUTE the k-th step of Gaussian Elimination
      if k == n then
        p = 1
        END CALL
      end if
      call recursiveBACKTRACK(k + 1)
    end if
    if p == 1 then
      END CALL
    end if
  end for
```

Efficiency

As we have said for algorithm 4.2, the number of nodes generated is unpredictable, so the worst case is considered in order to study the performance of algorithm 4.3. The worst case complexity can be easily obtained from the general formula already seen for the backtrack process when the highest number of tuples that can be generated is $n!$,

$$\mathrm{O}\left(\mathrm{p}(n)n!\right). \tag{4.31}$$

In this case, for each possible tuple, an LU factorization is needed. Thus, for each tuple we have the polynomial complexity

$$\mathrm{O}\left(\mathrm{p}(n)\right) = \mathrm{O}\left(\frac{2}{3}n^3\right), \tag{4.32}$$

Therefore, the overall worst case complexity for algorithm 4.3 results to be

$$\mathrm{O}\left(\mathrm{p}(n)n!\right) = \mathrm{O}\left(\frac{2}{3}n^3 n!\right), \tag{4.33}$$

which is an order of magnitude smaller than the worst case complexity for algorithm 4.2.

Experimental Results

As in the case of the determinant-based backtracking procedure, algorithm 4.3 has been implemented with different T functions. From a Gauss elimination point of view, we chose four different Ts :

- at call k, T_{g1} searches for the next non-null pivot element along the diagonal from position k to n;
- at call k, T_{g2} searches randomly for a non-null pivot element along the sub-diagonal from position k to n. Moreover, the pivot positions are remembered so that when a call is resumed no pivot is chosen two times;
- at call k, T_{g3} searches for the pivot element with maximum modulus along the diagonal, from position k to n. In practice, diagonal elements are ordered in a descending order considering their modulus. Then, at each step of a call, they could be sequentially tried from the biggest to the smallest;

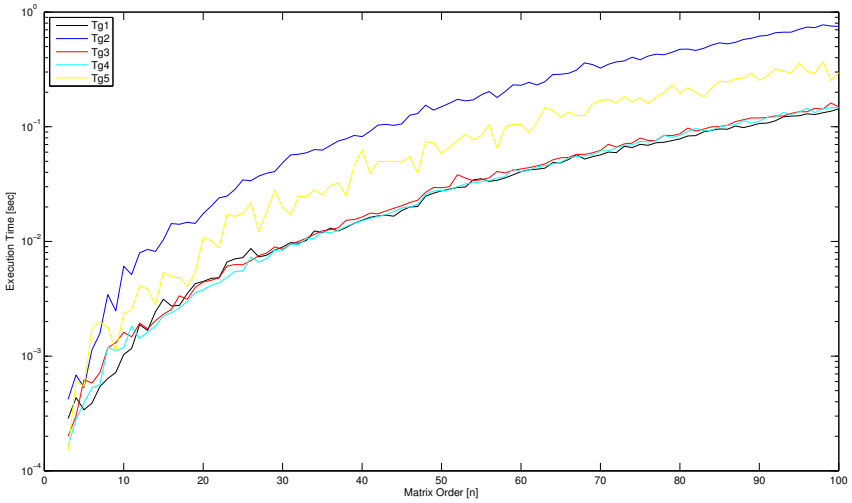


Figure 4.6: Comparison among different state space organization trees in the Gauss-based backtracking algorithm. Black line represents algorithm with T_{g1} while blue line represents algorithm with T_{g2}. Function T_{g1} makes the algorithm the fastest while function $Tg2$ the slowest.

- at call k, T_{g4} searches for the next non-null element along the diagonal from position n to k. It can be seen that function T_{g4} works in the opposite way with respect to function k, T_{g1}.

A fifth function has been defined based on the ones defined above:

- at call k, function T_{g5} behaves like one of the functions among T_{g1}, T_{g2} and T_{g3}. The choice is made randomly.

Note that T_{g1} and T_{g2} are nothing less than the same functions already tried for the determinant-based algorithm.

In figure (4.6) a comparison in terms of time execution among the different T_gs implementations is shown. Functions T_{g1}, T_{g3} and T_{g4} have fundamentally the same performance while, as already seen for the determinant-based case, function T_{g2} has the worst performance. Note that function T_{g5} has a behavior that averages out the performances of the other four functions.

As in the previous case, different node generations can yield different solutions. Let us have the following Laplacian sub-matrix:

$$L_{ff} = \begin{bmatrix} 5-\iota & -2 & -2+\iota & \iota \\ -\iota & -1+2\iota & 2 & 1-\iota \\ -2\iota & -1+2\iota & 2+\iota & -1 \\ 1 & \iota & -2 & -1-2\iota \end{bmatrix},$$

that is non-singular and whose leading principal minors are non-null. Solution yielded from the algorithm that implements T_{g1} is

$$P_{g1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

as we expected since L_{ff} do not need a relabelling and T_{g1} searches for the nearest solution to an identity matrix. The other T functions has yielded the following solutions:

- T_{g2} leads to

$$P_{g2} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix};$$

- T_{g3} leads to

$$P_{g3} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix};$$

- T_{g4} leads to

$$P_{g4} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix};$$

- T_{g5} leads to

$$P_{g5} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

where only P_{g3} and P_{g5} are equal. Then the solution is not unique since five algorithms have yielded four different solutions. This results were expected because already shown by the determinant-based algorithm.

4.3 Comparing Algorithms

The two different implementations of the algorithm in terms of bounding functions, have shown to have different performances for what concern the worst case complexity. In fact, we found that:

- *determinant-based backtracking algorithm* has the worst case complexity

$$\mathrm{O}\left(\mathrm{p}(n)n!\right) = \mathrm{O}\left(\frac{1}{6}n^4 n!\right);$$

- *Gauss-based backtracking algorithm* has the worst case complexity

$$\mathrm{O}\left(\mathrm{p}(n)n!\right) = \mathrm{O}\left(\frac{2}{3}n^3 n!\right).$$

Practical results have simply confirmed what stated above.

Experiments have been made using the following test matrices:

Test 1) complex random follower-follower Laplacian matrices;

Test 2) complex random non-singular matrices;

Test 3) complex random singular matrices.

where matrix values were drawn from the standard uniform distribution on the open interval(0;1). In figure (4.7) a comparison among the proposed algorithm over a Laplacian sub-matrices is shown. Algorithms worked over a Laplacian sub-matrix of variable order, from a minimum of 3 to a maximum of 70. Figure (4.7b) shows a comparison between the best of the two implementations, that are algorithms with T_{d1} and T_{g1}. The difference in terms of time execution between the determinant-based implementation and the Gauss-based implementation is unquestionable. That difference is even more evident in figure (4.9) where algorithms have been tested with random complex singular matrices. In that case, the maximum order of the test matrices has been chosen to be 9. Figure (4.9b) shows the comparison between the best T function of the two implementations. Again, Gauss-based algorithm has an overcoming performance. In figure (4.8) the results of experiment 2 are shown. Algorithm implementations have been tested with random complex non-singular matrices, with a maximum order of 70. As it can be seen, the performance algorithms reached are the same as in experiment 1.

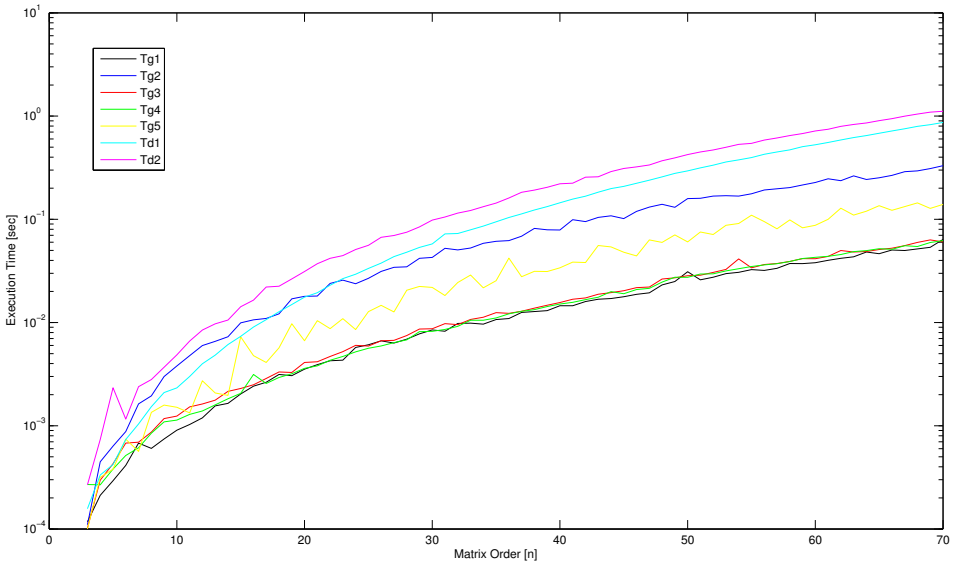
Remark 4.2. *Recall that the complex Laplacian matrix has been defined as the matrix whose elements are:*

$$l_{ij} = \begin{cases} -w_{ij} & \text{if } i \neq j \text{ and } j \in \mathcal{N}_i^-, \\ 0 & \text{if } i \neq j \text{ and } j \notin \mathcal{N}_i^-, \\ \boxed{\sum_{j \in \mathcal{N}_i^-} w_{ij}} & \boxed{\text{if } i = j.} \end{cases}$$

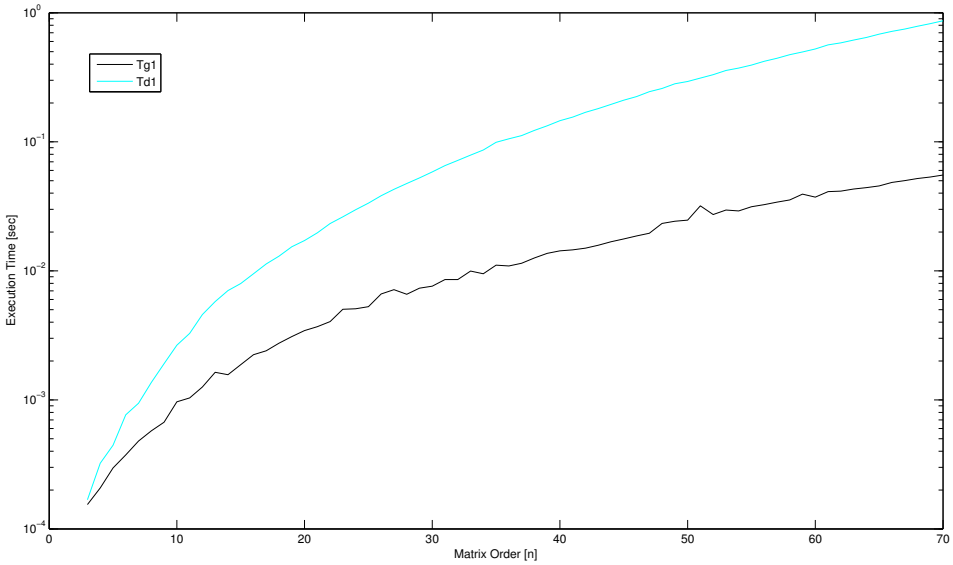
What we notice is that the diagonal elements are the sum of the elements in the same row. Since the Laplacian matrix in the experiments has been built using random values from the range $(0; 1)$*, the complex weights* w_{ij} *result to have both positive real and positive imaginary parts. Then the following condition holds for the diagonal elements:*

$$|l_{kk}| > |l_{kj}| \quad \text{for} \quad k = 1, \ldots, n.$$

It simply means that the diagonal elements are already the elements with the maximum modulus, and then the Gauss elimination with diagonal pivoting has a stability performance in between that of the Gauss with partial pivoting and that of the Gauss with total pivoting.

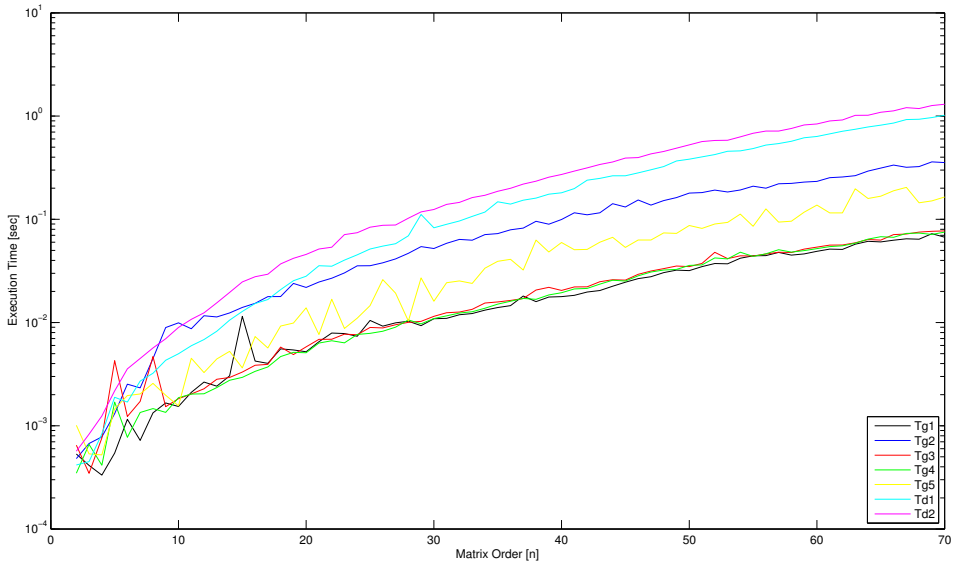


(a)

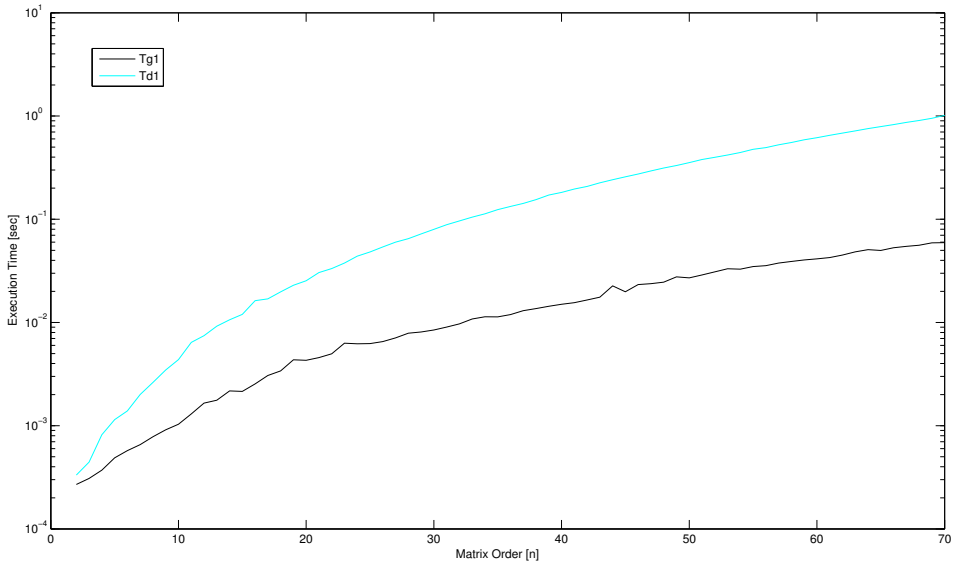


(b)

Figure 4.7: Comparison among different backtracking implementations. Tests have been made over complex random follower-follower Laplacian matrices.

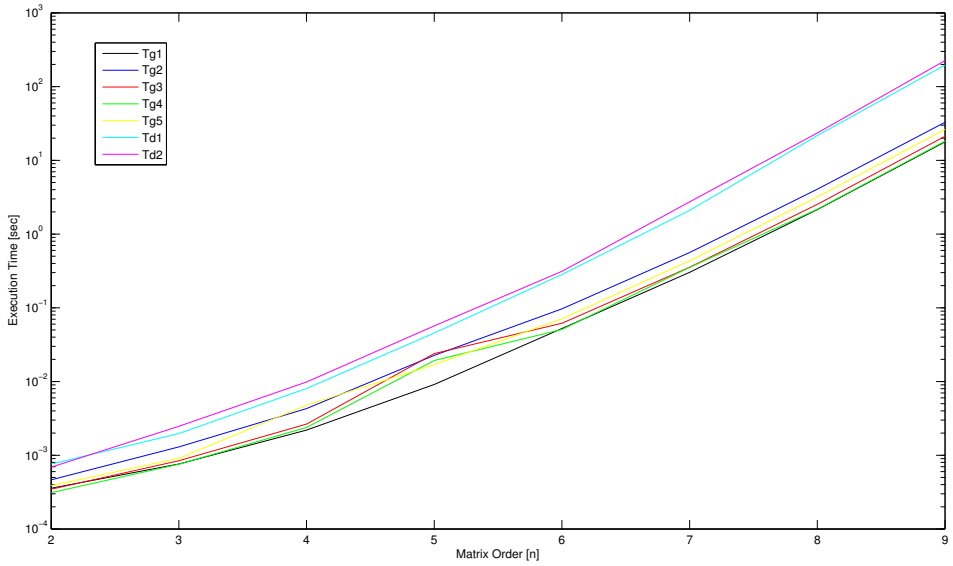


(a)

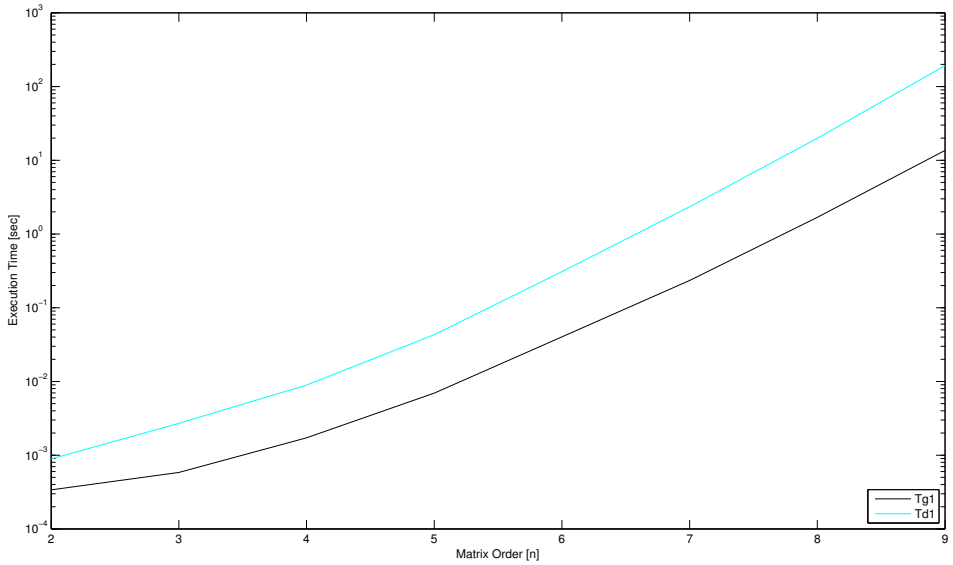


(b)

Figure 4.8: Comparison among different backtracking implementations. Tests have been made over complex random non-singular matrices.



(a)



(b)

Figure 4.9: Comparison among different backtracking implementations. Tests have been made over complex random singular matrices.

Chapter 5

The Multiplicative IEP and Ballantine's Theorem

An *Inverse Eigenvalue Problem* (see [14]) concerns the reconstruction of a matrix from prescribed spectral data, which may consist of the complete or only partial information of eigenvalues or eigenvectors. Such a problem has been encountered in chapter 3 where in order to stabilize the Laplacian matrix $L \in \mathbb{C}^{(n\times n)}$ of a sensing digraph, a complex diagonal matrix $D \in \mathbb{C}^{(n\times n)}$ was needed such that the matrix obtained from the product DL had all its eigenvalues in the right half complex plane. The problem mentioned is called the Multiplicative Inverse Eigenvalue Problem and, in the specific case, only partial information of the desired eigenvalues is given. They must have positive real parts. In this chapter we will analyse the multiplicative problem and give algorithms to find the desired complex diagonal matrix.

5.1 The Inverse Eigenvalue Problem of a Matrix

In a mathematical model (see [13] and [14]), it is generally assumed that there is a correspondence between the *endogenous variables*, that is, the internal parameters, and the *exogenous variables*, that is, the external behavior. As a consequence, when dealing with physical systems, problems can be classified in two main types.

Direct Problems A *direct problem* is the process of analysing and deriving the spectral information and, hence, inducing the dynamical behavior of a system from a priori known physical parameter such as mass, length, elasticity, inductance, capacitance, and so on.

Inverse Problems An *inverse problem* is to validate, determine, or estimate the parameters of the system according to its observed or expected behavior.

The concern in the direct problem is to express the behavior in terms of parameters whereas in the inverse problem the concern is to express the parameters in terms of behavior. In the former the behavior usually is a deterministic consequence of the parameters. In the latter the inverse problem often turns out to be ill-posed in that it has multiple solutions.

Among the inverse problems of various nature, there is the particular class of eigenvalue problems associated with matrices. In this context, an *Inverse Eigenvalue Problem* concerns the *reconstruction of a matrix from prescribed spectral data*, which may consist of *the complete or only partial information of eigenvalues or eigenvectors*. The objective of an inverse eigenvalue problem is *to construct a matrix that maintains a certain specific structure as well as that given spectral property*. To confine the construction to certain special classes of matrices is often necessary for the inverse problem to be more meaningful, either physically or mathematically. The solution to an inverse eigenvalue problem therefore should satisfy two constraints:

1. the *spectral constraint*, referring to the prescribed spectral data;

2. the *structural constraint*, referring to the desirable structure.

In practice, it may occur that one of the two constraints in an **IEP** [1] should be enforced more critically than the other, for example, due to physical realizability. Without this, the physical system simply cannot be built. There are also situations when one constraint could be more relaxed than the other, for example, due to the physical uncertainty. When the two constraints cannot be satisfied simultaneously, the inverse eigenvalue problem could be formulated in a least squares setting, in which one of the two constraints is compromised. Note that the meaning of *"being structured"* can be taken in different ways. Some of the structures, such as Jacobi or Toeplitz, result in matrices forming linear subspaces; structures such as non-negative or stochastic, limit entries of matrices in a certain range and so on.

These constraints define a variety of inverse eigenvalue problems, that can be classified according to characteristics such as additive, multiplicative, parametrized, structured, partially described or least squares. Nonetheless, an

[1] **I**nverse **E**igenvalue **P**roblem.

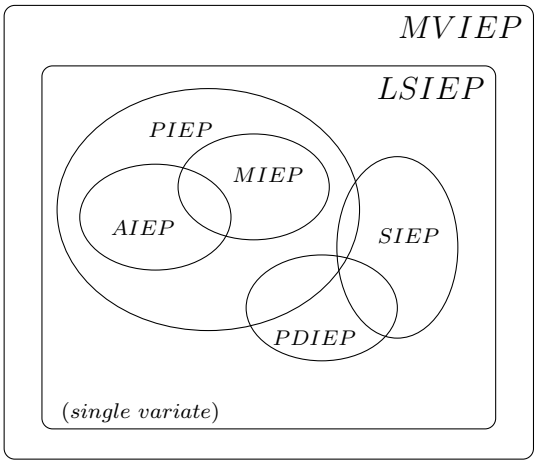


Figure 5.1: Main classification of Inverse Eigenvalue Problems.

IEP often carries overlapping characteristics and it is sometimes difficult to determine which characteristic is the most prominent. In general, the following types can be defined:

MVIEP	=	**M**ulti-**V**ariate **I**nverse **E**igenvalue **P**roblem
LSIEP	=	**L**east **S**quare **I**nverse **E**igenvalue **P**roblem
PIEP	=	**P**arametrized **I**nverse **E**igenvalue **P**roblem
SIEP	=	**S**tructured **I**nverse **E**igenvalue **P**roblem
PDIEP	=	**P**artially **D**escribed **I**nverse **E**igenvalue **P**roblem
AIEP	=	**A**dditive **I**nverse **E**igenvalue **P**roblem
MIEP	=	**M**ultiplicative **I**nverse **E**igenvalue **P**roblem

In figure (5.1) a possible inclusion relationship among the different IEPs is shown. The inclusion diagram implies several points.

- The MVIEP is the most general class of IEP. Nonetheless, the single variate has been the most studied.

- All single variate problems have a natural generalization to the Least Squares formulation.

- The AIEP and the MIEP are two extensively studied special cases of the PIEP.

- The PDIEP is the most difficult to classify. It arises when there are no reasonable tools available to evaluate the entire spectral information due to, for instance, the complexity or the size of the physical system the IEP belongs to.

Note that the given classification is neither definite nor complete. It is simply a representation of the main IEPs. In fact, other characterizations and overlapping are possible. In figure (5.2) a finer classification is depicted and acronyms are explained in table 5.1.

Associated with any **IEP** four fundamental questions of different nature arise:

1. the theory of *solvability*,
2. the practice of *computability*,
3. the analysis of *sensitivity*,
4. the reality of *feasibility*.

The problem of *solvability* consists in finding *necessary* or *sufficient* conditions under which an inverse eigenvalue problem has a solution. Related to the solvability is the issue of uniqueness of a solution. On the other hand, the main concern in *computability* has been to the develop a procedure by which, knowing a priori that the given spectral data are feasible, a matrix can be constructed numerically. The discussion on *sensitivity* concerns perturbation analysis when an IEP is modified by changes in the spectral data. The *feasibility* is a matter of differentiation between whether the given data are exact or approximate, complete or incomplete, and whether an exact value or only an estimate of the parameters of the physical system is needed. Each of these questions is essential but challenging to the understanding of a given IEP. Unfortunately, not many IEPs are comprehensively understood in all these four aspects.

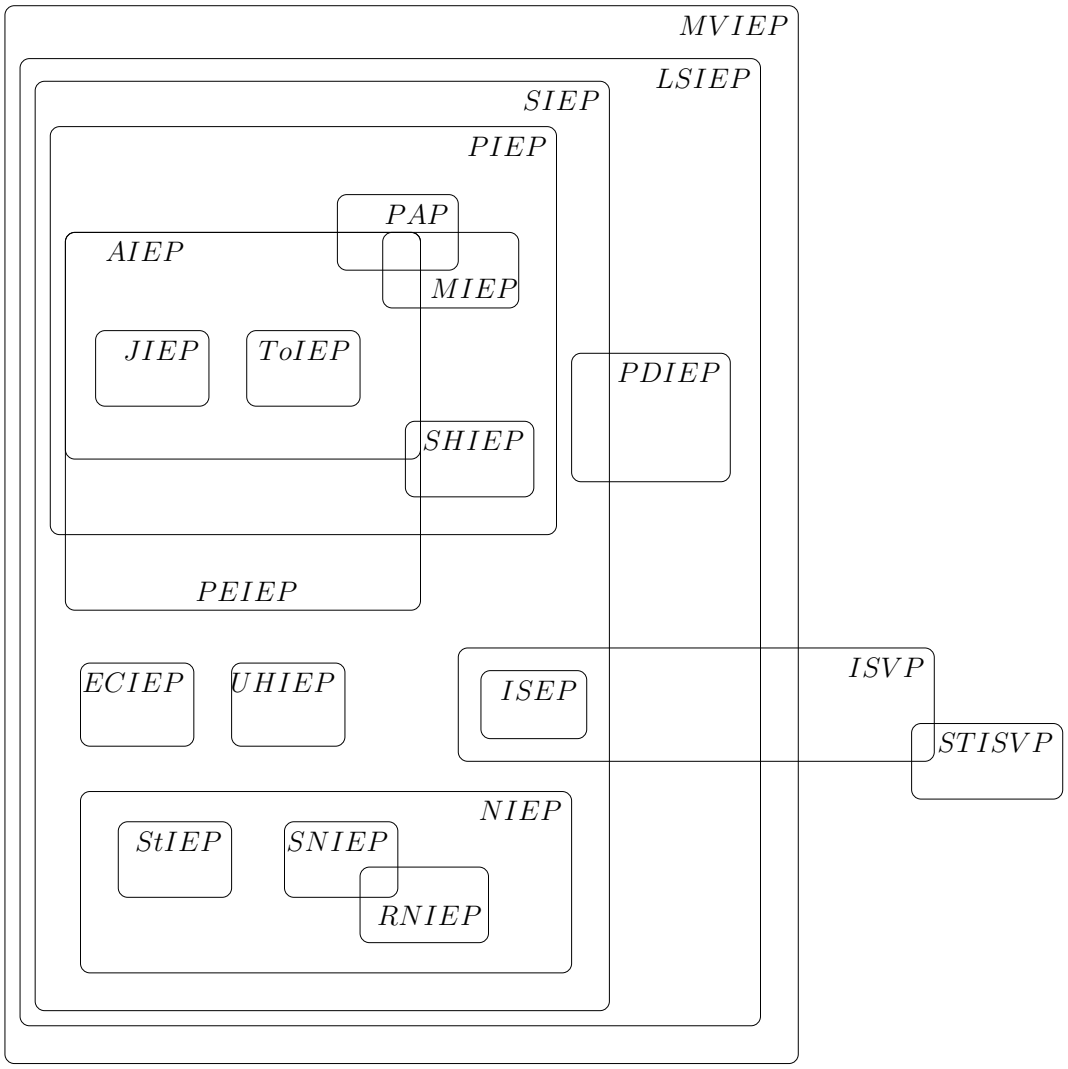


Figure 5.2: Classification of Inverse Eigenvalue Problems.

Acronym	Meaning
AIEP	**A**dditive **IEP**
ECIEP	**E**quality **C**onstrained **IEP**
ISEP	**I**nverse **S**ingular/**E**igenvalue **P**roblem
ISVP	**I**nverse **S**ingular **V**alue **P**roblem
JIEP	**J**acoby **IEP**
LSIEP	**L**east **S**quare **IEP**
MIEP	**M**ultiplicative **IEP**
MVIEP	**M**ulti-**V**ariate **IEP**
NIEP	**N**on-negative **IEP**
PAP	**P**ole **A**ssignment **P**roblem
PEIEP	**IEP** with **P**rescribed **E**ntries
PIEP	**P**arametrized **IEP**
PDIEP	**P**artially **D**escribed **IEP**
RNIEP	**R**eal-valued **N**on-negative **IEP**
SHIEP	**S**chur-**H**orn **IEP**
SIEP	**S**tructured **IEP**
SNIEP	**S**ymmetric **N**on-negative **IEP**
StIEP	**S**tochastic **IEP**
STISVP	**S**ing-**T**hompson **ISVP**
ToIEP	**T**oeplitz **IEP**
UHIEP	**U**nitary **H**essenberg **IEP**

Table 5.1: Summary of the acronyms for IEPs.

5.2 Application: The Pole Assignment Problem

Inverse Eigenvalue Problems arise from a remarkable variety of applications. The list includes applications like:

- control design,
- system identification,
- seismic tomography,
- principal component analysis,
- exploration and remote sensing,
- antenna array processing,
- geophysics,
- molecular spectroscopy,

- particle physics,
- structural analysis,
- circuit theory,
- mechanical simulation.

A common phenomenon that stands out in most of these applications is that the physical parameters of the underlying system are to be reconstructed from knowledge of its *dynamical behavior*. The meaning of *"dynamical behavior"* can be qualified in a different ways. For example, vibrations depend on natural frequencies and normal modes, stability controls depend on the location of eigenvalues, and so on. In that context, the spectral information used to affect the dynamical behavior varies in various ways. If the physical parameters can be, as they are, described mathematically in the form of a matrix, then the problem is an IEP. The structure of the matrix is usually inherited from the physical properties of the underlying system.

A valuable example is the Pole Assignment Problem, extensively studied and documented in the literature because of its importance in practice. Consider first the following dynamic state equation:

$$\dot{x}(t) = Ax(t) + Bu(t), \tag{5.1}$$

where $x(t) \in \mathbb{R}^m$. The two given matrices $A \in \mathbb{R}^{(n\times n)}$ and $B \in \mathbb{R}^{(n\times m)}$ are invariant in time. One classical problem in control theory is to select the input $u(t)$ so that the dynamics of the resulting $x(t)$ is driven into a certain desired state. Depending on how the input $u(t)$ is calculated, there are generally two types of controls:

- the *State Feedback Control*,
- the *Output Feedback Control*.

As it can be seen in figure (5.2), the PAP is a special case of the PIEP.

5.2.1 State Feedback PAP

In *state feedback control*, the input $u(t)$ is selected as a linear function of the current state $x(t)$, that is,

$$u(t) = Fx(t). \tag{5.2}$$

In this way, the system is changed to a closed-loop dynamical system:

$$\dot{x}(t) = (A + BF)x(t). \tag{5.3}$$

A general goal in such a control scheme is to choose the *gain matrix* $F \in \mathbb{R}^{(m\times n)}$ so as to achieve stability or speed up response. There are many ways to do that. One way is to minimize a certain cost function in the so-called linear quadratic regulator. Another way is to directly translate the task to the selection of F so that the spectrum $\sigma(A+BF)$ is bounded in a certain region of the complex plan. Obviously, in the latter the choice of the region affects the degree of difficulty of control. The location of the spectrum can be further restrict by reassigning eigenvalues of the matrix $A + BF$ to a prescribed set. This leads to a special type of inverse eigenvalue problem usually referred to as the *State Feedback Pole Assignment Problem.*

(State Feedback PAP) Given $A \in \mathbb{R}^{(n\times n)}$ and $B \in \mathbb{R}^{(n\times m)}$ and a set of complex numbers $\{\lambda_k\}_{k=1}^n$, closed under complex conjugation, find $F \in \mathbb{R}^{(m\times n)}$ such that

$$\sigma(A + BF) = \{\lambda_k\}_{k=1}^n.$$

5.2.2 Output Feedback PAP

It is often the case in practice that the state $x(t)$ is not directly observable. Instead, only the output $y(t)$ is available. State and output are related by the following equation

$$y(t) = Cx(t), \tag{5.4}$$

where $C \in \mathbb{R}^{(p\times n)}$ is a known matrix. The input $u(t)$ must now be chosen as a linear function of the current output $y(t)$, that is,

$$u(t) = Ky(t). \tag{5.5}$$

The closed-loop dynamical system thus becomes

$$\dot{x}(t) = (A + BKC)x(t). \tag{5.6}$$

The goal is to select the *output matrix* $K \in \mathbb{R}^{(m\times p)}$ so as to reassign the eigenvalues of $A + BKC$. This output feedback PAP once again gives rise to a special type of IEP.

(Output Feedback PAP) $A \in \mathbb{R}^{(n\times n)}$, $B \in \mathbb{R}^{n\times m}$, and $C \in \mathbb{R}^{(p\times n)}$, and a set of complex numbers $\{\lambda_k\}_{k=1}^{n}$, closed under complex conjugation, find $K \in \mathbb{R}^{(m\times p)}$ such that

$$\sigma(A + BKC) = \{\lambda_k\}_{k=1}^{n}.$$

5.3 Multivariate IEP

What can be noted first both from figures (5.1) and (5.2) is that the main distinction in IEPs is between *single variate and multivariate* inverse problems. Nonetheless, the single variate one has been the most studied because of its importance in practice.

A *Multivariate Eigenvalue Problem* is to find real scalars $\{\lambda_1, \ldots, \lambda_m\}$ and a real vector $x \in \mathbb{R}^n$ such that equations

$$Ax = \Lambda x \tag{5.7}$$

$$\| x_i \| = 1, \ i = 1, \ldots, m. \tag{5.8}$$

are satisfied, in which $A \in \mathcal{S}(n)$ [2] is a given positive definite matrix partitioned into blocks:

$$A = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & & \vdots \\ A_{m1} & A_{m2} & \ldots & A_{mm} \end{bmatrix},$$

Λ is the diagonal matrix

$$\Lambda = \text{diag}\{\lambda_1 I_{n_i}, \ldots, \lambda_m I_{n_m}\}$$

with I_{n_i} the identity matrix of size n_i, and $x \in \mathbb{R}^n$ is partitioned into blocks

$$x = [x_1^T, \ldots, x_m^T]^T$$

with $x_i \in \mathbb{R}^{n_i}$. Note that, the single variate case when $m = 1$ is simply a classical symmetric eigenvalue problem.

[2] $\mathcal{S}(n)$:= { all symmetric matrices in $\mathcal{R}(n)$}.

5.4 Single variate IEP

In this section the main IEPs showed in figure (5.1) are listed together with some of their variations and definitions. In order to define the IEPs, symbols $\mathcal{M}, \mathcal{N}$ will be used to denote the following subsets of square matrices:

$$\begin{aligned}
\mathcal{R}(n) &:= \mathbb{R}^{(n\times n)}, \\
\mathcal{S}(n) &:= \{ \text{ all symmetric matrices in } \mathcal{R}(n)\}, \\
\mathcal{O}(n) &:= \{ \text{ all orthogonal matrices in } \mathcal{R}(n)\}, \\
\mathcal{D}_{\mathcal{R}}(n) &:= \{ \text{ all diagonal matrices in } \mathcal{R}(n)\}, \\
\mathcal{C}(n) &:= \mathbb{C}^{(n\times n)}, \\
\mathcal{H}(n) &:= \{ \text{ all Hermitian matrices in } \mathcal{C}(n)\}, \\
\mathcal{D}_{\mathcal{C}}(n) &:= \{ \text{ all diagonal matrices in } \mathcal{C}(n)\}.
\end{aligned}$$

5.4.1 Parametrized IEP

The class of *Parametrized Inverse Eigenvalue Problem* can include many other IEPs since many of them can be regarded as a parameter estimation problem. Nonetheless, the way these parameters regulate the problem allows us to narrow down the list of included IEPs.

A generic PIEP can be described as follows:

(PIEP) Given a family of matrices $A(c) \in \mathcal{M}$ with $c = [c_1, \ldots, c_m] \in \mathbb{F}^m$ and scalars $\{\lambda_1, \ldots, \lambda_n\} \subset \mathbb{F}$, find a parameter c such that $\sigma(A(c)) = \{\lambda_1, \ldots, \lambda_n\}$.

Note that the number m of parameters in c may be different from n. Depending on how the family of matrices $A(c)$ is specifically defined in terms of c, the PIEP can appear and be solved very differently. Nonetheless, a common feature in all variations is that the parameter c is used as a *control* that modulates to the underlying problem in a certain specific way. Some variations are:

(PIEP1) $A(c) = A_0 + \sum_{i=1}^{n} c_i A_i$ where $A_i \in \mathcal{R}(n)$ and $\mathbb{F} = \mathbb{R}$.

(PIEP2) $A(c) = A_0 + \sum_{i=1}^{n} c_i A_i$ where $A_i \in \mathcal{S}(n)$ and $\mathbb{F} = \mathbb{R}$.

A different and more complicated formulation of the PIEP is the following one:

(PIEP3) Given matrices $A \in \mathcal{C}(n)$, $B_i \in \mathbb{C}^{(n\times m_i)}$, $C_i \in \mathbb{C}^{(l_i\times n)}$, $i = 1, \ldots, q$, and scalars $\{\lambda_i, \ldots, \lambda_n\} \subset \mathbb{C}$, find matrices $K_i \in \mathbb{C}^{(m_i\times l_i)}$ such that $\sigma(A + \sum_{i=1}^{q} B_i K_i C_i) = \{\lambda_i, \ldots, \lambda_n\}$.

Note that, for $q = 1$, the PIEP3 includes as special cases the state feedback as well as the output feedback pole assignment problems.

A few interesting special cases of the PIEP are:

(AIEP) Given a matrix $A \in \mathcal{M}$, scalars $\{\lambda_1, \ldots, \lambda_n\} \subset \mathbb{F}$, and a class of matrices $\mathcal{N}$, find a matrix $X \in \mathcal{N}$ such that $\sigma(X + A) = \{\lambda_1, \ldots, \lambda_n\}$.

(MIEP) Given a matrix $A \in \mathcal{M}$, scalars $\{\lambda_1, \ldots, \lambda_n\} \subset \mathbb{F}$, and a class of matrices $\mathcal{N}$, find a matrix $X \in \mathcal{N}$ such that $\sigma(XA) = \{\lambda_1, \ldots, \lambda_n\}$.

The Additive IEP is a special case of the PIEP with $A(X) = A + X$ and X playing the role of c, and the Multiplicative IEP corresponds to the case where $A(X) = XA$. By being more specific on the class $\mathcal{N}$ of matrices, the problems themselves can be divided into further subclasses.

Additive IEP

The key feature of the Additive IEP is that a given matrix A is perturbed by the *addition* of a specially structured matrix X in order to match the desired eigenvalues. The eigenvalue information can provide at most n equations, so sometimes it may be desirable to limit the number of free parameters in X. Nonetheless, the set $\mathcal{N}$ can be taken quite liberally. Set $\mathcal{N}$ can then be used to impose a certain structural constraint on the solution matrix X. For example, structure on $\mathcal{N}$ sometimes arises naturally because of engineers' design constraints. In that sense, the Additive IEP presents itself various special cases:

(AIEP1) $\mathcal{M} = \mathcal{R}(n)$, $\mathbb{F} = \mathbb{R}$, $\mathcal{N} = \mathcal{D}_{\mathcal{R}}(n)$,

(AIEP2) $\mathcal{M} = \mathcal{S}(n)$, $\mathbb{F} = \mathbb{R}$, $\mathcal{N} = \mathcal{D}_{\mathcal{R}}(n)$,

(AIEP3) $\mathcal{M} = \mathcal{C}(n)$, $\mathbb{F} = \mathbb{C}$, $\mathcal{N} = \mathcal{D}_{\mathcal{C}}(n)$,

(AIEP4) $\mathcal{M} = \mathcal{H}(n)$, $\mathbb{F} = \mathbb{R}$, $\mathcal{N} = \mathcal{D}_{\mathcal{R}}(n)$.

Another interesting variant of the AIEP arises in, for example, control theory or algorithm design, where the stability is at issue. In such a problem it is more practically critical to have eigenvalues located in a certain region than at a certain points. One such problem can be stated as follows:

(AIEP5) Given $A \in \mathcal{R}(n)$, find $X \in \mathcal{N}$ with $\sigma(A + X)$ lies in a certain fixed region, say the right half, of the complex plane.

From a solvability point of view, Friedland proved the following theorem for the AIEP3:

Theorem 5.1. *For any specified* $\{\lambda_1, \ldots, \lambda_n\}$*, the AIEP3 is solvable. The number of solutions is finite and does not exceed* $n!$*. Moreover, for almost all* $\{\lambda_1, \ldots, \lambda_n\}$*, there are exactly* $n!$ *solutions.*

Multiplicative IEP

The Multiplicative IEP arises when the task is to *pre-multiply* a given matrix A by a specially structured matrix X to reposition or to precondition the distribution of its eigenvalues. This is very similar to but more general than the idea of preconditioning the matrix A where it is desired to find an efficient preconditioner M for A so that the product $M^{-1}A$ approximates the identity. Although the sense in which $M^{-1}A$ should approximate the identity differs according to the underlying method to be used, the general setting in the MIEP can be applied to the optimal preconditioning of a given matrix A. Perhaps, the simplest possible preconditioners are the diagonal scaling:

(MIEP1) $\mathcal{M} = \mathcal{R}(n)$, $\mathbb{F} = \mathbb{R}$, $\mathcal{N} = \mathcal{D}_{\mathcal{R}}(n)$,

(MIEP2) $\mathcal{M} = \mathcal{S}(n)$, $\mathbb{F} = \mathbb{R}$, $\mathcal{N} = \mathcal{D}_{\mathcal{R}}(n)$,

(MIEP3) $\mathcal{M} = \mathcal{C}(n)$, $\mathbb{F} = \mathbb{C}$, $\mathcal{N} = \mathcal{D}_{\mathcal{C}}(n)$.

Some other types of multiplicative inverse eigenvalue problems are worth to mention:

(MIEP4) Given a matrix $A \in \mathcal{H}_n$ and scalars $\{\lambda_1, \ldots, \lambda_n\} \subset \mathbb{R}$, find a matrix $X \in \mathcal{D}_{\mathcal{R}}(n)$ such that $\sigma(X^{-1}AX^{-1}) = \{\lambda_1, \ldots, \lambda_n\}$.

(MIEP5) Given a matrix $A \in \mathcal{R}(n)$, find $X \in \mathcal{D}_{\mathcal{R}}(n)$ with positive entries such that $\sigma(XA)$ lies in the right-half complex plane.

From a solvability point of view, Friedland proved the following theorem for the MIEP3:

Theorem 5.2. *If all principal minors of* A *are distinct from zero, then the MIEP3 is solvable for arbitrary* $\{\lambda_1, \ldots, \lambda_n\}$ *and there exist at most* $n!$ *distinct solutions.*

5.4.2 Structured IEP

A generic Structured Inverse Eigenvalue Problem may be stated as follows:

(SIEP) Given scalars $\{\lambda_1, \ldots, \lambda_n\} \in \mathbb{F}$, find $X \in \mathcal{N}$ which consists of specially structured matrices such that $\sigma(X) = \{\lambda_1, \ldots, \lambda_n\}$.

By demanding X to belong to $\mathcal{N}$, where a structure is defined, the SIEP is required to meet both the spectral constraints and the structural constraints. The structural constraint usually is imposed due to the realizability of the underlying physical system.

Many types of structures have been considered for the SIEP. The following are of the most interesting:

(SIEP1) $\mathbb{F} = \mathbb{R}$ and $\mathcal{N} = \{$all Toeplitz matrices in $\mathcal{S}(n)\}$,

(SIEP2) $\mathbb{F} = \mathbb{R}$ and $\mathcal{N} = \{$all pre-symmetric Jacobi matrices in $\mathcal{S}(n)\}$,

(SIEP3) $\mathbb{F} = \mathbb{R}$ and $\mathcal{N} = \{$all non-negative matrices in $\mathcal{S}(n)\}$,

(SIEP4) $\mathbb{F} = \mathbb{R}$ and $\mathcal{N} = \{$all non-negative matrices in $\mathcal{R}(n)\}$,

(SIEP5) $\mathbb{F} = \mathbb{C}$ and $\mathcal{N} = \{$all row-stochastic matrices in $\mathcal{R}(n)\}$.

The spectra of structured matrices may also be structured. So sometimes additional spectral information is given. The following problems are of example:

(SIEP6a) Given scalars $\{\lambda_1, \ldots, \lambda_n\}$ and $\{\mu_1, \ldots, \mu_{n-1}\} \subset \mathbb{R}$ that satisfy the interlacing property $\lambda_i \leq \mu_i \leq \lambda_{i+1}$ for $i = 1, \ldots, n-1$, find a Jacobi matrix J so that $\sigma(J) = \{\lambda_1, \ldots, \lambda_n\}$ and $\sigma(\tilde{J}) = \{\mu_1, \ldots, \mu_{n-1}\}$ where $\tilde{J}$ is the leading $(n-1) \times (n-1)$ principal sub-matrix of J.

(SIEP6b) Given scalars $\{\lambda_1, \ldots, \lambda_{2n}\}$ and $\{\mu_1, \ldots, \mu_{2n-2}\} \subset \mathbb{C}$, find tridiagonal symmetric matrices C and K such that the determinant $\det(Q(\lambda))$ of the λ-matrix $Q(\lambda) = \lambda^2 I + \lambda C + K$ has zeros precisely $\{\lambda_1, \ldots, \lambda_{2n}\}$ and $\det(\tilde{Q}(\lambda))$ has zeros precisely $\{\mu_1, \ldots, \mu_{2n-2}$ where $\tilde{Q}(\lambda)$ is obtained by deleting the last row and the last column of $Q(\lambda)$.

5.4.3 Least Squares IEP

An Inverse Eigenvalue Problem, especially for the real-valued case, may not necessarily have an exact solution. Moreover, the spectral information, in practice, often is obtained by estimation and hence it does not need to be rigorously

obtained. That is, there are situations where an approximate solution in the sense of *least squares* would be satisfactory. Problems we have hereinbefore seen can be generalized to the least squares formulation. However, any inverse eigenvalue problem has two constraints. Thus depending on which constraint is to be enforced explicitly, two ways of defining a least squares approximation are possible.

One natural way is to measure and to minimize the discrepancy among the eigenvalues:

(LSIEPa) Given a set of scalars $\{\lambda_1^*, \ldots, \lambda_m^*\} \subset \mathbb{F}(m \leq n)$, find a matrix $X \in \mathcal{N}$ and a set $\sigma = \{\sigma_1, \ldots, \sigma_m\}$ of indices with $1 \leq \sigma_1 < \cdots < \sigma_m \leq n$ such that the function

$$F(X, \sigma) := \frac{1}{2} \sum_{i=1}^{m} (\lambda_{\sigma_i}(X) - \lambda_i^*)^2, \tag{5.9}$$

where $\lambda_i(X), i = 1, \ldots, n$ are eigenvalues of the matrix X, is minimized.

Note that the set of prescribed eigenvalues has cardinality m which might be less than n. Consequently, associated with the LSIEP for each fixed X is always a combinatorics problem

$$\min_{1 \leq \sigma_1 < \cdots < \sigma_m \leq n} \sum_{i=1}^{m} (\lambda_{\sigma_i}(X) - \lambda_i^*)^2, \tag{5.10}$$

that looks for the closest match between a subset of spectrum of X and the prescribed eigenvalues.

Another way to formulate the least squares approximation is to measure and to minimize the discrepancy between the matrices:

(LSIEPb) Given the set $\mathcal{M}$ whose elements satisfy a certain spectral constraint and a set $\mathcal{N}$ that defines a structural constraint, find $X \in \mathcal{M}$ that minimizes the function

$$F(X) := \frac{1}{2} \parallel X - P(X) \parallel^2, \tag{5.11}$$

where $P(X)$ is the projection of X onto $\mathcal{N}$.

The spectral constraint could be,for example, the isospectral surface

$$\mathcal{W}(\Lambda) = \{X \in \mathcal{R}(n) \mid X = Q^T \Lambda Q, Q \in \mathcal{O}(n)\} \subset \mathcal{S}(n)$$

where the complete spectral $\Lambda := \text{diag}\{\lambda_1, \ldots, \lambda_n\}$ is given, or the set

$$\mathcal{W}(\Gamma, V) := \{X \in \mathcal{R}(n) \text{ or } \mathcal{S}(n) \mid XV = V\Gamma\},$$

where only a portion of eigenvalues $\Gamma := \text{diag}\{\lambda_1, \ldots, \lambda_k\}$ and eigenvalues $V := [v_1, \ldots, v_k]$ are given. Note that if $F(X) = 0$ at a least square solution, then we have also solved the inverse eigenvalue problem of finding $X \in \mathcal{N}$ that satisfies $\mathcal{M}$. So a general SIEP can be solved through the setup of an LSIEPb.

For engineering applications, it is mostly the case that the realizability of the physical system is more critical than the accuracy of the eigenvalues. That is, the structural constraint $\mathcal{N}$ has to be enforced in order that the construction of a physical system be realizable whereas a discrepancy in the eigenvalues is sometimes tolerable because often these eigenvalues are an estimate anyway.

There are several variations to the LSIEP. In the LSIEPa it can be noted that the number of variable parameters for adjusting the matrix X, for example, the degree of freedom in $\mathcal{N}$, could be different from the dimension n. One special case of the LSIEPa where the number l of free parameters might also differ from the number m of the partially prescribed eigenvalues:

(LSIEPa1) $\mathcal{N} = \{A(d) = A_0 + \sum_{i=1}^{l} d_i A_i \mid A_0, A_1, \ldots, A_l \in \mathcal{S}(n) \text{ given}\}, \mathbb{F} = \mathbb{R}.$

Problem LSIEPa1 may be seen in terms of LSIEPb. For a given $\Lambda_m^* := \text{diag}\{\lambda_1^*, \ldots, \lambda_m^*\}$, consider the subset

$$\Gamma := \{Q \,\text{diag}(\Lambda_m^*, \Lambda_c) Q^T \in \mathcal{O}(n), \Lambda \in \mathcal{D}_{\mathcal{R}}(n-m)\} \tag{5.12}$$

and the affine subspace

$$\mathcal{A} := \{A(d) \mid d \in \mathbb{R}^l\} \tag{5.13}$$

with $A(d)$ defined in LSIEPa1. Since Γ contains all symmetric matrices in $\mathbb{R}^{(n\times n)}$ with $\lambda_1^*, \ldots, \lambda_m^*$ ass part of the spectrum, finding the shortest distance between $\mathcal{A}$ and Γ would be another meaningful least squares approximation. The problem can be formulate as follows:

(LSIEPb1) Find $d \in \mathbb{R}^l$, $Q \in \mathcal{O}(n)$, and $\Lambda_c \in \mathcal{D}_{\mathcal{R}}(n-m)$ such that the function

$$G(d, Q, \Lambda) := \frac{1}{2} \parallel A(d) - Q \,\text{diag}(\Lambda_m^*, \Lambda_c) Q^T \parallel_F^2, \tag{5.14}$$

is minimized.

Other variations of LSIEPb include:

(LSIEPb2) $\mathcal{M} = \mathcal{W}(\Lambda), \mathcal{N} = \{A\}$.

(LSIEPb3) $\mathcal{M} = \mathcal{W}(\Lambda), \mathcal{N} = \{\text{all Toeplitz matrices in } \mathcal{S}(n)\}$.

(LSIEPb4) $\mathcal{M} = \mathcal{W}(\Gamma, V), \mathcal{N} = \{A\}$ and $\mathcal{N} = \mathcal{R}(n)$ or $\mathcal{S}(n)$.

5.4.4 Partially Described IEP

In the reconstruction of a system, instead of knowing the complete spectrum, there are also situations where only a portion of eigenvalues and eigenvectors are available. This is especially the case when due to the complexity or the size of the physical system, no reasonable analytical tools are available to evaluate the entire spectral information. This is the case where a Partially described IEP arises. A generic PDIEP is as follows:

(PDIEP) Given vectors $\{v^{(1)}, \ldots, v^{(k)}\} \subset \mathbb{F}^n$ and scalars $\{\lambda_1, \ldots, \lambda_k\} \subset \mathbb{F}$ where $1 \leq k < n$, find a matrix $X \in \mathcal{N}$ such that $Xv^{(i)} = \lambda_i v^{(i)}$ for $i = 1, \ldots, k$.

We could also consider the following variations:

(PDIEP1) $\mathbb{F} = \mathbb{R}, \mathcal{N} = \{\text{all Toeplitz matrices in } \mathcal{S}(n)\}$.

(PDIEP2) $\mathbb{F} = \mathbb{R}, \mathcal{N} = \{\text{all Jacobi matrices in } \mathcal{S}(n)\}$.

(PDIEP3) $\mathbb{F} = \mathbb{R}, \mathcal{N} = \{\text{all pre-symmetric Jacobi matrices in } \mathcal{S}(n)\}$.

Other variations of the PDIEP include:

(PDIEP4) Given two distinct scalars $\lambda, \mu, \in \mathbb{R}$ and two non-zero vectors $x, y \in \mathbb{R}^n$, find two Jacobi matrices J and $\bar{J}$ so that $Jx = \lambda x$ and $\bar{J}y = \mu y$, where J and $\bar{J}$ differ only in the (n, n) position.

(PDIEP5) Given distinct scalars $\{\lambda_1, \ldots, \lambda_n\} \subset \mathbb{R}$ and a non-zero vector $x \in \mathbb{R}^n$, find a Jacobi matrix J such that $\lambda(J) = \{\lambda_1, \ldots, \lambda_n\}$ and that either $Jx = \lambda_1 x$ or $Jx = \lambda_n x$.

(PDIEP6) Construct an $n \times n$ symmetric band matrix of bandwidth p from the knowledge of all the eigenvalues and the first p components of all the normalized eigenvectors.

5.5 Ballantine's Theorem and Stability

In chapter 3 we have seen that a multi-agent system can be controlled by relatively simply laws when its formation is modelled by a 2-reachable complex weighted digraph. More specifically, the control of a multi-agent formation can be done via the complex Laplacian associated to the sensing digraph modelling the formation itself. In both single-integrator kinematics and double-integrator dynamics what it turned out is that the stability of the whole system, that is, the capability of the agents to reach the desired planar formation depends on the stability of the Laplacian matrix. In other words, the multi-agent system is able to reach a planar formation if and only if the matrix

$$-DL \tag{5.15}$$

has all stable eigenvalues. Moreover, since the formation of n agents has two co-leaders and none of them have incoming edges, the Laplacian matrix can then be written as

$$L = \left[\begin{array}{c|c} 0_{2\times 2} & 0_{2\times(n-2)} \\ \hline L_{lf} & L_{ff} \end{array}\right] \tag{5.16}$$

from which it is clear that two of the n eigenvalues are zero. Then the stability issue involves only a Laplacian sub-matrix, the follower-follower matrix L_{ff}. What is needed to find, is a complex diagonal matrix M such that matrix D is:

Single-Integrator Kinematics

$$D = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & M \end{bmatrix}, \tag{5.17}$$

Double-Integrator Dynamics

$$D = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & \epsilon M \end{bmatrix}, \tag{5.18}$$

and matrix (5.15) is stable.

The problem we must solve is a Multiplicative Inverse Eigenvalue Problem, a special case of the Parametrized Inverse Eigenvalue Problem we have already encountered in section 5.4.1. More precisely, having that

$$L_{ff} \in \mathbb{C}^{(m\times m)} \quad \text{and} \quad M \in \mathcal{D}_c(m), \quad m = n-2, \tag{5.19}$$

the problem is what has been denoted as MIEP3, a variant of the MIEP. Fortunately, a theorem for the existence of the solution of such a problem has been yielded by Friedland (theorem 5.2) which states that the solution exist if and only if all principal minors of L_{ff} are non-null. Unfortunately, there is no efficient algorithm to find such a solution (see [14]). In order to solve the stabilization problem then, a different way should be found. As stated in [29], Ballantine's theorem ensure that the diagonal matrix can be computed one element at a time and it can suggest a valuable algorithm for our purpose.

The existence of a stabilizing matrix has been stated by Ballantine too [5] both for the real and the complex case. More precisely, Ballantine stated that a complex diagonal matrix which makes eigenvalues positive or with positive real parts could exist. This is what we need since, with such a matrix, matrix (5.15) would have all stable eigenvalues.

Theorem 5.3 states the existence for the real case.

Theorem 5.3 (Real case)**.** *Let A be an $m \times m$ real matrix all of whose leading principal minors are positive. Then there is an $m \times m$ positive diagonal matrix M such that all the roots of MA are positive and simple.*

Proof: In order to prove the statement induction on m is used. For $m = 1$ the result is trivial, so suppose that $m \geq 2$ and that the result holds for matrices of order $m - 1$. Let A be an $m \times m$ real matrix all of whose leading principal minors are positive and let A_1 be its leading principal sub-matrix of order $m - 1$. Then all the lpm's of A_1 are positive, so by our induction assertion there is a positive diagonal matrix M_1 of order $m - 1$ such that all roots of $M_1 A_1$ are positive and simple. Let d be a real number to be determined later (but treated as a variable for the present). Let A be partitioned as follows:

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix},$$

where A_4 is 1×1. Define an $m \times m$ diagonal matrix M (depending on d) by conformable partition:

$$M = \begin{bmatrix} M_1 & 0 \\ 0 & d \end{bmatrix}.$$

Let $MA = C(d)$, where now we emphasize the dependence on d. Then

$$C(0) = \begin{bmatrix} M_1 A_1 & M_1 A_2 \\ 0 & 0 \end{bmatrix},$$

so the non-zero roots of $C(0)$ are just those of M_1A_1, hence are positive and simple (and there are exactly $m-1$ of them). $C(0)$ also has a simple root at zero. Thus for all sufficiently small $d > 0$ the real parts of the roots of $C(d)$ are (still) m distinct real numbers at least $n-1$ of which are positive. (This follows from the fact that the roots of $C(d)$ are continuous functions of d.) Choose some such d. Then the roots of $C(d)$ are still real and simple (since non-real roots must occur in conjugate pairs) and at least $m-1$ of them are positive. But the determinant of $C(d)$ is positive since those of A and M are, so in fact all m roots of $C(d)$ are positive. This conclude the proof of the induction step and hence of the theorem. □

Note that this same kind of argument can be used to prove that, when A is an $m \times m$ complex matrix all of whose *leading principal minors are non-zero* and also an open sector containing the positive real axis is prescribed, this same kind of argument yields a complex $m \times m$ diagonal matrix M such that all the roots of MA lie in the prescribed sector. Theorem 5.4 states the existence for the complex case.

Theorem 5.4 (Ballantine(1970))**.** *Let A be an $m \times m$ complex matrix all of whose leading principal minors are non-zero. Then there is an $m \times m$ complex diagonal matrix M such that all the eigenvalues of MA are positive and simple.*

As described in the proof of theorem 5.3, the diagonal matrix M can be found step by step computing the diagonal elements one at a time, both in the real and the complex case. This leads to algorithm 5.1 which fundamentally works searching for the diagonal elements of M and verifying whether the real part of the eigenvalues is positive. The algorithm has been implemented in two different manners. What distinguishes the two realizations is the way in which eigenvalues are considered. In the first implementation the eigenvalues are computed for each d_i tried, while in the second implementation (sec. 5.6) no eigenvalue is computed but a bounding for them is considered. Moreover, in order to improve the algorithm convergence, different ways to pick elements d_i from $\mathcal{W}$ are tried.

5.5.1 Simulations

The first implementation of the Ballantine's algorithm has been made by computing the eigenvalues of the matrix $M_{[1\sim i]}L_{ff[1\sim i]}$ at each step i. The elements d_i to be tried has been picked from the set $\mathcal{W}$ in different ways.

Algorithm 5.1 Stabilization by a complex Diagonal matrix

Require: L_{ff}, $\dim(L_{ff}) = m$
  for $i = 1, \ldots, m$ **do**
    for $d_i \in \mathcal{W} \subset \mathbb{C}$ **do**
      $M_{[1\sim i]} = \text{diag}(d_1, \ldots, d_i)$
      $A_{[1\sim i]} = M_{[1\sim i]} L_{ff[1\sim i]}$
      $\sigma(A_{[1\sim i]}) = \{\lambda_1, \ldots, \lambda_i\}$
      if $\Re\{\lambda_k\} > 0$, for $k = 1, \ldots i$ **then**
        step to the next i
      else
        try a new $d_i \in \mathcal{W}$
      end if
    end for
  end for
  return $M = \text{diag}(d_1, \ldots, d_m)$

In figure (5.3) an example of search into $\mathcal{W}$ can be seen. Elements d_i are chosen so that they belongs to straight lines parallel to the imaginary axis. In fact, as soon as the real part $d_{r,i}$ of the element is chosen, all the imaginary parts that belong to the segment which points have real part $d_{r,i}$ are picked, and elements d_i are tried.

Figures (5.4) and (5.5) show instead a circular set $\mathcal{W}$. In the first, elements d_i are searched along the straight lines which cross the axis in their origin. The direction can be modified so that the entire set $\mathcal{W}$ is spanned. In figure (5.5) instead, elements d_i are chosen along circles centred in the origin of the complex plane. Set $\mathcal{W}$ is completely spanned by varying the radius of the circle.

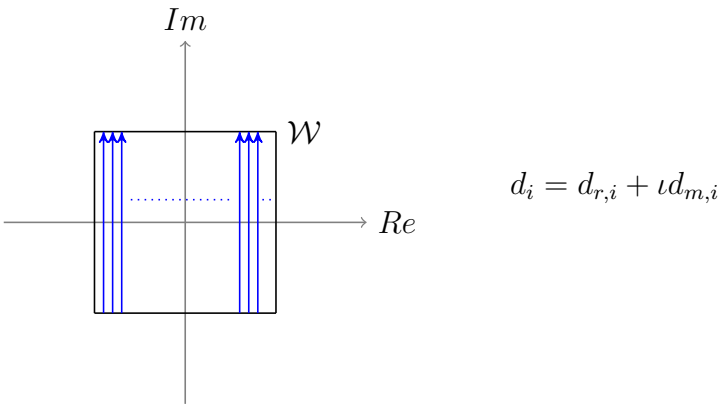
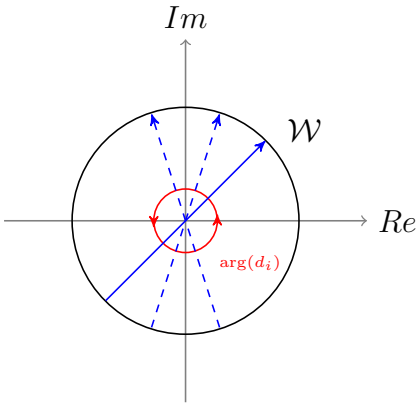


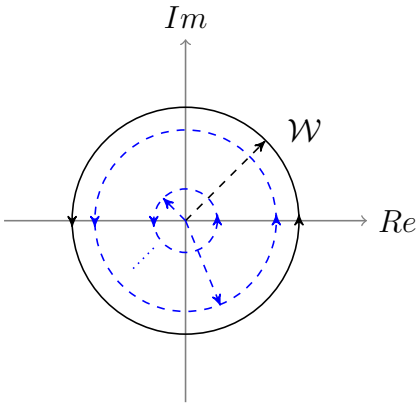
Figure 5.3: Set $\mathcal{W}$ and search strategy for Ballantine's-based algorithm. Elements d_i are chosen from segments parallel to the imaginary axis.

Let us have the sensing digraph and the formation basis depicted in figure



$$d_i = d_{r,i} + \iota d_{m,i} = |d_i| \, \mathrm{e}^{\iota \arg(d_i)}$$

Figure 5.4: Set $\mathcal{W}$ and search strategy for Ballantine's-based algorithm. Elements d_i are chosen from segments with variable direction.



$$d_i = d_{r,i} + \iota d_{m,i} = |d_i| \, \mathrm{e}^{\iota \arg(d_i)}$$

Figure 5.5: Set $\mathcal{W}$ and search strategy for Ballantine's-based algorithm. Elements d_i are chosen from circles with variable radius.

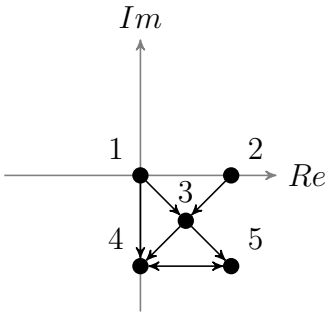
(5.6). The corresponding Laplacian matrix is

$$L = \left[\begin{array}{cc|ccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline -1-\iota & -1+\iota & 2 & 0 & 0 \\ -1+\iota & 0 & -2 & 1-3\iota & 2+2\iota \\ 0 & 0 & 2\iota & -1-\iota & 1-\iota \end{array}\right], \tag{5.20}$$

and conditions $L\xi = 0$ and $L\mathbf{1}_n = 0$ hold. The eigenvalues of L are

$$\sigma(L) = \{0, 0, 2, 2.2496 - 3.6\iota, -0.2496 - 3.6\iota\}, \tag{5.21}$$

where two of them are zero as expected. However, one of the eigenvalues of the follower-follower Laplacian has negative real part so a diagonal matrix must be found in order to condition that eigenvalue and have all eigenvalues



$$\xi = \begin{bmatrix} 0 \\ 2 \\ 1-\iota \\ -2\iota \\ 2-2\iota \end{bmatrix}$$

Figure 5.6: Formation basis and sensing digraph.

with positive real part. Matrix L_{ff} has non-zero leading principal minors so the Ballantine's theorem holds without a relabelling being necessary. The algorithm yields the following solution

$$M_1 = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.1\iota & 0 \\ 0 & 0 & 0.2\iota \end{bmatrix}$$

and the new eigenvalues of the conditioned follower-follower Laplacian matrix $M_1 L_{ff}$ are

$$\sigma(L_{ff}) = \{1, 0.0564 - 0.0436\iota, 0.4436 + 0.3436\iota\},$$

with all positive real parts as desired. This solution has been obtained by the algorithm with the search model depicted in figure (5.3), that is the one which search elements d_i along straight lines parallel to the imaginary axis. If the search method depicted in figure (5.4) is used, the following solution is found

$$M_2 = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.2 + 0.0405\iota & 0 \\ 0 & 0 & -0.2 + 0.4370\iota \end{bmatrix}$$

and the eigenvalues of $M_2 L_{ff}$ are

$$\sigma(L_{ff}) = \{1, 0.0096 - 0.2954\iota, 0.5490 + 0.3729\iota\}.$$

The third search method, depicted in figure (5.5), yields the following so-

lution instead

$$M_3 = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.0980 + 0.0199\iota & 0 \\ 0 & 0 & 0.0732 + 0.0682\iota \end{bmatrix},$$

and the new eigenvalues of $M_3 L_{ff}$ are

$$\sigma(L_{ff}) = \{1, 0.2974 - 0.2256\iota, 0.0015 - 0.0536\iota\}.$$

What we have obtained, is the proof that not only does Ballantine's algorithm ensure that a stabilizing matrix can be found computing its diagonal elements one at a time, but also verifying the existence of different solutions to the problem.

The main drawback on this approach is that Ballantine-based algorithm does not converge as the matrix order increase. This is a problem that could depend on factors such as the difficulties in computing exactly the eigenvalues of high order complex matrices and the finiteness of the set $\mathcal{W}$ in which elements d_i are searched for. Moreover, it has been noted that many times a change in the first diagonal element d_1 changes the entire solution or at least some other diagonal elements d_i, especially when d_1 is chosen out of the set $\mathcal{W}$. It could suggest that for matrices of higher order not only would be important where the solution is searched but also from which point the search starts. This fact could affect the convergence of the algorithm as well as the other two aforementioned reasons. Unfortunately the positions of the solutions are unpredictable and then there is not a rigorous way to choose where to search for a solution and where to start from.

5.6 Bounding the Eigenvalues of a Complex Matrix

Eigenvalues of a complex matrix can be bounded in several ways. Examples of bounds can be seen in [36], [37], and [43] where circular and rectangular regions containing the eigenvalues are defined for a complex square matrix.

In particular, in [36] the following bound is presented.

Rectangular Region Let A be a complex matrix of order n, with eigenvalues

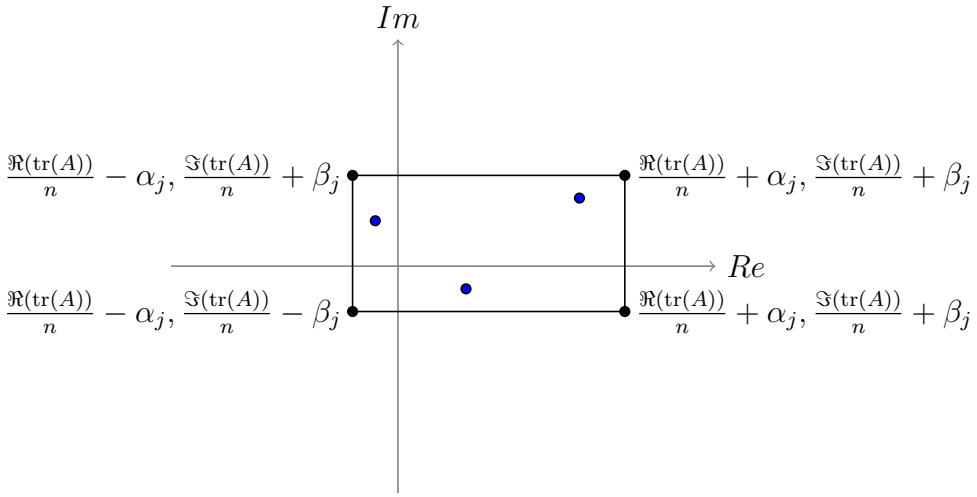


Figure 5.7: Rectangular bound for the eigenvalues of a complex square matrix.

$\lambda_1, \ldots, \lambda_n$. Then all of the eigenvalues lie in the rectangle

$$\left[\frac{\Re(\mathrm{tr}(A))}{n} - \alpha_j, \frac{\Re(\mathrm{tr}(A))}{n} + \alpha_j\right] \times \left[\frac{\Im(\mathrm{tr}(A))}{n} - \beta_j, \frac{\Im(\mathrm{tr}(A))}{n} + \beta_j\right], \quad (5.22)$$

Parameters α_j and β_j are

$$\alpha_j = \left(\frac{n-1}{n}\right)^{\frac{1}{2}} \left(\left(k_j(A) + \Re(\mathrm{tr}(A^2))\right) - \frac{(\Re(\mathrm{tr}(A)))^2}{n}\right)^{\frac{1}{2}}, \quad (5.23)$$

and

$$\beta_j = \left(\frac{n-1}{n}\right)^{\frac{1}{2}} \left(\left(k_j(A) + \Re(\mathrm{tr}(A^2))\right) - \frac{(\Im(\mathrm{tr}(A)))^2}{n}\right)^{\frac{1}{2}}, \quad (5.24)$$

for $j = 1, 2, 3$. The rectangular region then varies with j and is tighter for $k_3(A)$ that has the following expression:

$$k_3(A) = \left(\left(|A|^2 - \frac{|\,\mathrm{tr}(A)|^2}{n}\right)^2 - \frac{|AA^* - A^*A|^2}{2}\right)^{\frac{1}{2}} + \frac{|\,\mathrm{tr}(A)|^2}{n}. \quad (5.25)$$

This bound describes, in the complex plane, a rectangular region which contains all the eigenvalues of a complex square matrix. As it can be seen from the equations above, α_j and β_j are positive quantities, and the rectangle is the one in figure (5.7).

The main idea is to use algorithm 5.1 with the described bound instead of computing the eigenvalues at each step. That is, at each step, element d_i must be found so that the rectangular region for matrix $M_{[1\sim i]}L_{ff[1\sim i]}$ is in the

right half of the complex plane. To ensure that, it suffices to find elements d_i for the left side of the rectangle so that $\frac{\Re(\mathrm{tr}(A))}{n} - \alpha_j$ is positive. That is, the conditions to be verified are:

$$\frac{\Re(\mathrm{tr}(A))}{n} > 0, \tag{5.26}$$

$$\frac{\Re(\mathrm{tr}(A))}{n} > \alpha_j. \tag{5.27}$$

From condition (5.26) we obtain a bound for the set $\mathcal{W}$. Suppose we are at the step k of the algorithm and $A_{[1\sim k-1]}$ has been already stabilized. Since $n > 0$, we are interested in the sign of the trace of $A_{[1\sim k]}$. Then, we can write

$$\frac{\Re(\mathrm{tr}(A))}{n} > 0 \Rightarrow \eta_{kR} + d_{kR} l_{kkR} - d_{kI} l_{kkI} > 0 \Rightarrow d_{kR} l_{kkR} > d_{kI} l_{kkI} - \eta_{kR}, \tag{5.28}$$

where $\eta_{kR} = \sum_{h=1}^{k-1} d_h l_{hh}$ is the real part of the trace of matrix $M_{[1\sim k-1]} L_{ff[1\sim k-1]}$. Depending on the sign of l_{kkR}, we have:

if $l_{kkR} > 0$

$$d_{kR} > d_{kI} \frac{l_{kkI}}{l_{kkR}} - \frac{\eta_{kR}}{l_{kkR}}, \tag{5.29}$$

if $l_{kkR} < 0$

$$d_{kR} < d_{kI} \frac{l_{kkI}}{l_{kkR}} - \frac{\eta_{kR}}{l_{kkR}}. \tag{5.30}$$

Consequently, we can write

$$\mathrm{sign}(l_{kkR}) d_{kR} > d_{kI} \frac{l_{kkI}}{|l_{kkR}|} - \frac{\eta_{kR}}{|l_{kkR}|}. \tag{5.31}$$

Equation (5.31) is describing nothing more than a sub-plane bounded from a straight line of the form

$$y_k = d_{kI} m_k - q_k, \tag{5.32}$$

where $y_k = \mathrm{sign}(l_{kkR}) d_{kR}$, $m_k = \frac{l_{kkI}}{|l_{kkR}|}$ and $q_k = \frac{\eta_{kR}}{|l_{kkR}|}$. That sub-plane is our set $\mathcal{W}$. In figures (5.8) and (5.9) the two possible sub-planes are depicted.

In summary, elements d_i can be found in the following way:

1. chose a range for d_{kI},

2. for each d_{kI} compute

$$\hat{y}_k = d_{kI} m_k - q_k,$$

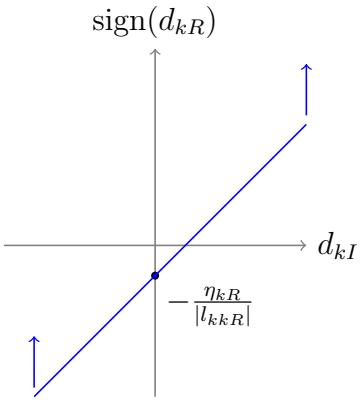


Figure 5.8: Set $\mathcal{W} = \{d_k \in \mathbb{C} : \text{sign}(l_{kkR})d_{kR} > d_{kI}\frac{l_{kkI}}{|l_{kkR}|} - \frac{\eta_{kR}}{|l_{kkR}|}\}$ and $\frac{l_{kkI}}{|l_{kkR}|} > 0$.

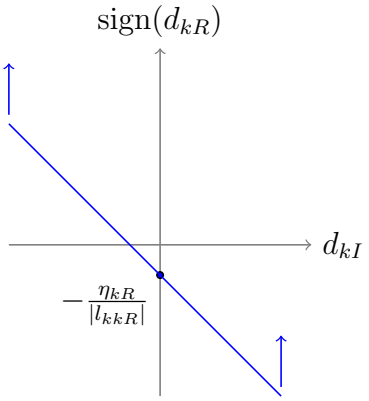


Figure 5.9: Set $\mathcal{W} = \{d_k \in \mathbb{C} : \text{sign}(l_{kkR})d_{kR} > d_{kI}\frac{l_{kkI}}{|l_{kkR}|} - \frac{\eta_{kR}}{|l_{kkR}|}\}$ and $\frac{l_{kkI}}{|l_{kkR}|} < 0$.

that is the minimum value for $\hat{y}_k$ such that $\hat{d_{kR}}\,\text{sign}(l_{kkR})\hat{y}_k$,

3. chose $y_k = \hat{y}_k + e$, where $e > 0$ and let e varies in a positive range,

4. compute $d_{kR} = \text{sign}(l_{kkR})y_k$.

The algorithm has been tested in Matlab and the code is shown in appendix G. Unfortunately the bound results to be too loose and the algorithm is not able to converge.

Chapter 6

Application to a Planar Multi-Agent Formation

In chapter 3 we have seen how a multi-agent planar formation can be controlled by simply laws that exploits the complex Laplacian of the sensing digraph which represents the formation itself. In addition, in chapters 4 and 5 algorithms to make the formation control work in practice have been discussed.

In this chapter simulations are presented both for the single-integrator kinematics and the double-integrator dynamics case, in sections 6.1 and 6.2 respectively. We will apply what we have hereinbefore seen to the agents' planar formation of figure (6.1), where both formation basis and sensing digraph are shown. As it can be verified from figure (6.1a) the sensing digraph is 2-reachable. In fact, for the Laplacian matrix (6.1) of the sensing digraph, algebraic conditions $L\xi = 0$ and $\det(L_{ff}) \neq 0$ are verified and then theorem 3.2 holds.

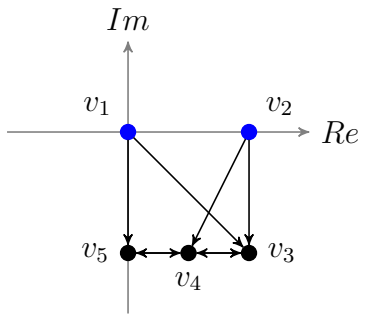
Laplacian Matrix

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -2-2\iota & 2-2\iota & -12 & 12+4\iota & 0 \\ 0 & -1+2\iota & 1 & 4-2\iota & -4 \\ -2-2\iota & 0 & 0 & -4+4\iota & 6-2\iota \end{bmatrix} \tag{6.1}$$

Follower-Follower Laplacian Matrix

$$L_{ff} = \begin{bmatrix} -12 & 12+4\iota & 0 \\ 1 & 4-2\iota & -4 \\ 0 & -4+4\iota & 6-2\iota \end{bmatrix} \tag{6.2}$$

The 2-reachability condition is very important because assures that the formation can asymptotically reach the desired planar formation. The only



(a)

$$\xi = \begin{bmatrix} 0 \\ 4 \\ 4-4\iota \\ 2-4\iota \\ -4\iota \end{bmatrix}$$

Figure 6.1: Planar formation and sensing digraph for a MAS. The blue dots represent the leaders of the formation while the black ones are the followers.

problem is in the eigenvalues of the matrix L_{ff} which are

$$\sigma(L_{ff}) = \{-12.7411 - 0.2966\iota, 1.0328 - 0.0201\iota, 9.7083 - 3.6833\iota\}.$$

In order to reach the desired planar formation these eigenvalues must have positive real parts. In this case however, the follower-follower Laplacian matrix has one eigenvalue with negative real part, and then L needs to be stabilized by a complex diagonal matrix. This is possible since matrix L_{ff} has non-null leading principal minors and then Ballantine's theorem holds.

A possible solution that leads to stability is the following complex diagonal matrix

$$M = \begin{bmatrix} -3.233 & 0 & 0 \\ 0 & 1.14 + 0.58\iota & 0 \\ 0 & 0 & 1.326 - 0.02\iota \end{bmatrix}, \tag{6.3}$$

which gives new eigenvalues

$$\sigma(DL) = \{0, 0, 37.6328 - 1.243\iota, 1.4205 + 0.3924\iota, 13.3787 - 1.8814\iota\},$$

with positive real parts as desired.

Thus, the Laplacian matrix will be stabilized by the diagonal matrix

$$D = \left[\begin{array}{cc|ccc} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & -3.233 & 0 & 0 \\ 0 & 0 & 0 & 1.14 + 0.58\iota & 0 \\ 0 & 0 & 0 & 0 & 1.326 - 0.02\iota \end{array}\right]. \tag{6.4}$$

The agents formation will asymptotically converge to the planar formation F_ξ

given by equation (6.9) with the complex Laplacian matrix $-DL$.

In order to test the Laplacian-based control, both examples of a planar formation with and without a input are shown. For this reason, it is useful to remember which is the internal state evolution for a system modelled by a state space representation. The general form of a *time invariant system state space* is

$$\dot{x} = Ax + Bu, \tag{6.5}$$

where vector $x = [x_1, \ldots, x_k]^T$ is the system state vector and $u = [u_1, \ldots, u_m]$ is the input vector. In case that no inputs are given, the state space equation is

$$\dot{x} = Ax, \tag{6.6}$$

that is a homogeneous differential equation. The solution of equation (6.6) is

$$x(t) = \mathrm{e}^{A(t-\tau)}\, x(\tau), \tag{6.7}$$

where τ is a generic instant of time for which the value of the internal state x is known. In case of non-null inputs, equation (6.5) is a non-homogeneous differential equation which has *forced solution*

$$x(t) = \mathrm{e}^{A(t-\tau)}\, x(\tau) + \int_{\tau}^{t} \mathrm{e}^{A(t-\alpha)}\, Bu(\alpha)\, \mathrm{d}\alpha\,. \tag{6.8}$$

The convolution integral shown in equation (6.8) is the forced response of the system, which takes account of the system response due to the non-null input u.

Those equations will be of importance in the following sections when the convergence of a planar formation will be shown both for the null and non-null input cases.

6.1 Single-Integrator Kinematics

The single-integrator kinematics case has been fully described in subsection 3.3.1 where conditions to make possible the control of a planar formation were given. The 2-reachability is one of them and it has an algebraic form that is described in theorem 3.1 which also gives a practical formula to find the planar formation F_ξ. In fact, the aforementioned theorem states that if and only if conditions $L\xi = 0$ and $\det(L_{ff}) \neq 0$ are satisfied, then the planar formation

$F_\xi = c_1\mathbf{1}_n + c_2\xi$ can be reached and constants c_1 and c_2 can be found from the following expression:

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 & \xi_1 \\ 1 & \xi_2 \end{bmatrix}^{-1} \begin{bmatrix} \bar{z_1} \\ \bar{z_2} \end{bmatrix}. \tag{6.9}$$

where $\bar{z_1}$ and $\bar{z_2}$ are the leaders positions. Since $\det(L_{ff}) =\neq 0$, then a planar formation can be reached. Nonetheless, the eigenvalues of L are

$$\sigma(L) = \{0, 0, -12.7411 - 0.2966\iota, 1.0328 - 0.0201\iota, 9.7083 - 3.6833\iota\} \tag{6.10}$$

and then matrix L needs to be stabilized by a diagonal matrix D of the form

$$D = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & M \end{bmatrix}, \tag{6.11}$$

as previously shown. Since M has been found, every planar formation F_ξ with the complex Laplacian matrix DL can be asymptotically reached.

The single-integrator kinematics case yields the following interaction laws for the agents in order to make them reach a planar formation while moving

$$\begin{cases} v_i = v_0(t), & i = 1, 2; \\ v_i = \sum_{j\in N_i} w_{ij}(z_j - z_i) + v_0(t), & i = 3, \cdots, n. \end{cases} \tag{6.12}$$

Note that $v_0(t)$ is the synchronized velocity of the leaders and it has to be known by the followers as well. Since the control law is $\dot{z}_i = v_i$, equation (6.12) takes the following compact form

$$\dot{z} = -DLz + bv_0(t), \tag{6.13}$$

where $b = [1, 1, \ldots, 1]^T$ is an n-dimensional vector of ones so that the velocity is available to all the agents, as a control input for the leaders and as a parameter for the followers. In case of $v_0(t) = 0$, that is no control input is given, the state space representation will be of the form

$$\dot{z} = -DLz. \tag{6.14}$$

Equations (6.13) and (6.14) represent the forced and unforced case for which

the solutions are the ones given in equations (6.8) and (6.7) respectively:

$$z(t) = \mathrm{e}^{-DL(t-\tau)}\, z(\tau) + \int_{\tau}^{t} \mathrm{e}^{-DL(t-\alpha)}\, b v_0(\alpha)\, \mathrm{d}\alpha, \tag{6.15}$$

and

$$z(t) = \mathrm{e}^{-DL(t-\tau)}\, z(\tau). \tag{6.16}$$

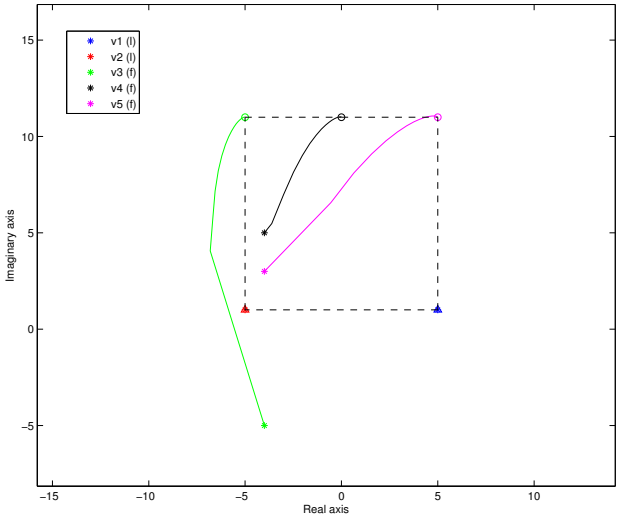
From equations (6.14) and (6.15) it can be easily seen why the matrix system must have all stable eigenvalues. In fact the agents converge to a planar formation with the eigenvalues of matrix $-DL$ and if they are unstable no convergence is possible at all.

In figure (6.2) a group of 5 agents randomly disposed is shown to converge to the planar formation of figure (6.1). The asterisks represent the agents before reaching the planar formation while the triangles and the circles represent respectively the leaders and the followers after reached the formation. Both in experiment (6.2a) and (6.2b) leader agents have not been moved so asterisks and triangles have superposed one another. As it can be seen, when the followers sense the leader agents they suddenly move to reach the designated position next to them.

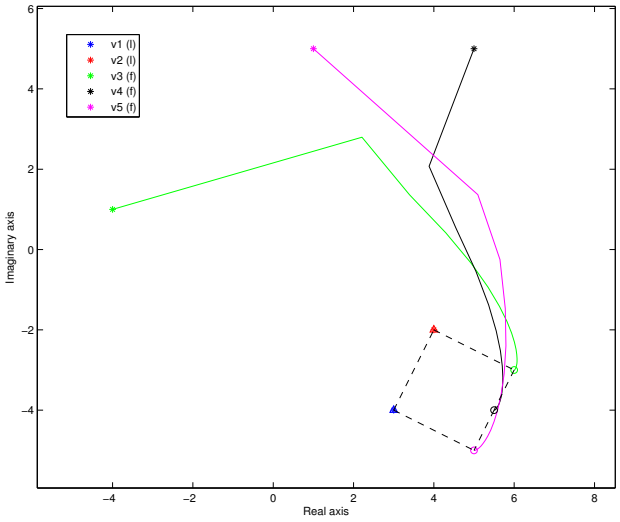
In figure (6.3) the forced evolution of a group of 5 agents is shown. The leader agents are controlled by the following velocity input

$$v_0(t) = 2t\cos(0.1t) + \iota 0.5t\sin(0.1t). \tag{6.17}$$

In figure (6.3a) the evolution of the system in a time interval $t \in [0, 12]$ is shown. In figure (6.3b) the agents randomly disposed are shown to start moving while in figure (6.3c) the reached planar formation is shown. As it can be seen, controlling the leader agents only, the followers react following them and reaching the desired planar formation with the eigenvalues of $-DL$.

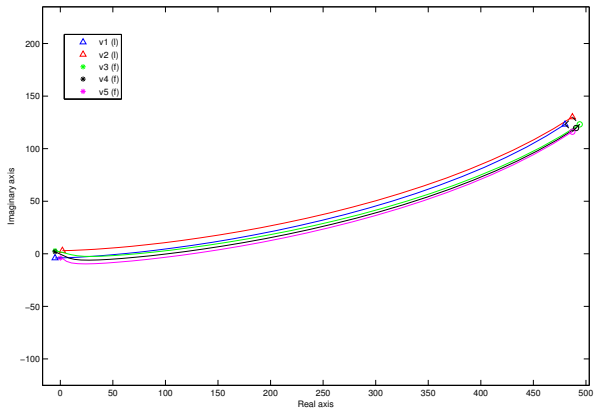


(a)

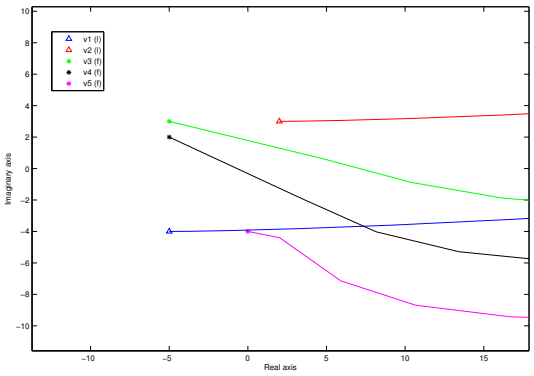


(b)

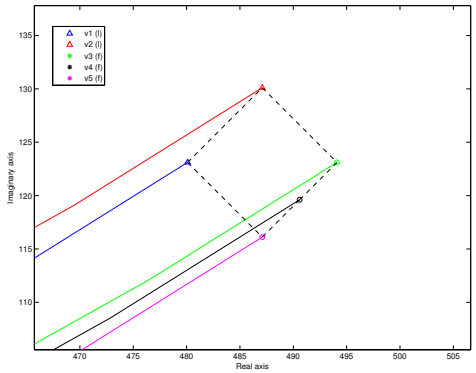
Figure 6.2: Single integrator kinematics case. Agents reaching a planar formation.



(a)



(b)



(c)

Figure 6.3: Single integrator kinematics case. Agents reaching a moving formation.

6.2 Double-Integrator Dynamics

The double-integrator dynamics case has been fully described in subsection 3.3.2 where similar conditions to the single-integrator case have been given for the formation to reach a stable planar formation. For example, theorem 3.6 gives conditions for a planar formation to be reached which are substantially the same as the single-integrator case. In fact a planar formation $F_\xi = c_1\mathbf{1}_n + c_2\xi$ can be reached if and only if $L\xi = 0$ and $\det(L_{ff}) \neq 0$, where constants c_1 and c_2 are again the solutions of system (6.9).

The Laplacian matrix of the sensing digraph is the same as before and it needs to be stabilized by a diagonal matrix, although this time D have the form

$$D = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & \epsilon M \end{bmatrix}, \tag{6.18}$$

where $\epsilon > 0$ is a scalar which can be chosen equal 1 in this case. Matrix M has been previously found so that the non-null eigenvalues of DL are in the open right half complex plane.

The double-integrator dynamics case yields the following interaction laws for the agents in order to make them reach a planar formation while moving

$$\begin{cases} a_i = -\gamma v_i + a_0(t), & i = 1, 2; \\ a_i = \sum_{j\in N_i} w_{ij}(z_j - z_i) - \gamma v_i + a_0(t), & i = 3, \cdots, n. \end{cases} \tag{6.19}$$

Since the control laws are $\dot{z}_i = v_i$ and $\dot{v}_i = a_i$ then equation (6.19) can be written in the following form

$$\begin{bmatrix} \dot{z} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0_{n\times n} & I_n \\ -DL & -\gamma I_n \end{bmatrix} \begin{bmatrix} z \\ v \end{bmatrix} + ba_0(t), \tag{6.20}$$

where $b = [1, \ldots, 1]^T$ is a vector of ones. In case of $a_0(t) = 0$, that is no control input is given, the state space represents an unforced system as previously seen. The forced case have solution

$$\begin{bmatrix} z(t) \\ v(t) \end{bmatrix} = \mathrm{e}^{H(t-\tau)} \begin{bmatrix} z(\tau) \\ v(\tau) \end{bmatrix} + \int_\tau^t \mathrm{e}^{H(t-\alpha)}\, ba_0(\alpha)\, \mathrm{d}\alpha, \tag{6.21}$$

while the unforced one has solution

$$\begin{bmatrix} z(t) \\ v(t) \end{bmatrix} = \mathrm{e}^{H(t-\tau)} \begin{bmatrix} z(\tau) \\ v(\tau) \end{bmatrix}. \tag{6.22}$$

In both cases the matrix H is the matrix of the system (6.20). Thus matrix H is

$$H = \begin{bmatrix} 0_{n\times n} & I_n \\ -DL & -\gamma I_n \end{bmatrix}. \tag{6.23}$$

Note that, given the non-null eigenvalues λ_i of matrix DL, the constant γ must verify the condition given in theorem (3.7), that is

$$\frac{Re(\lambda_i)}{(Im(\lambda_i))^2} > \frac{1}{\gamma^2}, \quad i = 3, \ldots, n. \tag{6.24}$$

where $n = 5$ in our experiments. Since for the eigenvalues of the Laplacian matrix (6.1) we have that

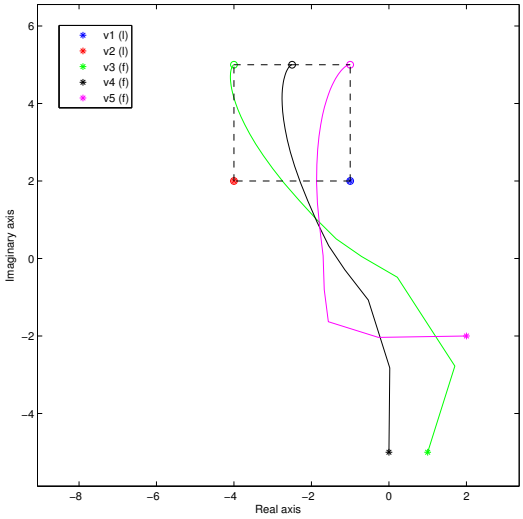
$$\frac{Re(\lambda_3)}{(Im(\lambda_3))^2} = 24.357, \quad \frac{Re(\lambda_4)}{(Im(\lambda_4))^2} = 9.2254, \quad \frac{Re(\lambda_5)}{(Im(\lambda_5))^2} = 3.7797,$$

then we are quite free in the choice of γ.

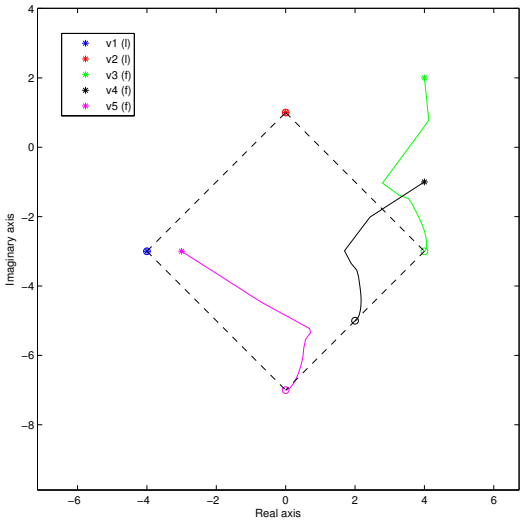
In figure (6.4) two groups of random agents are shown to reach a planar formation. Both experiments have $\gamma = 5$. In figure (6.5) a group of random agents is shown to reach a moving formation. The leaders were controlled by an acceleration which have the following expression

$$a_0(t) = 2t\cos(0.1t) + \iota 1.5t\sin(0.1t). \tag{6.25}$$

The experiment have been done with $\gamma = 5$ and the evolution of the system has been observed in a time interval $t \in [0, 60]$. In figure (6.5a) the trajectory followed by the agents can be seen while in figure (6.5b) and (6.5c) the agents can be observed in a random formation before the control input were applied and in the moving desired formation afterwards.

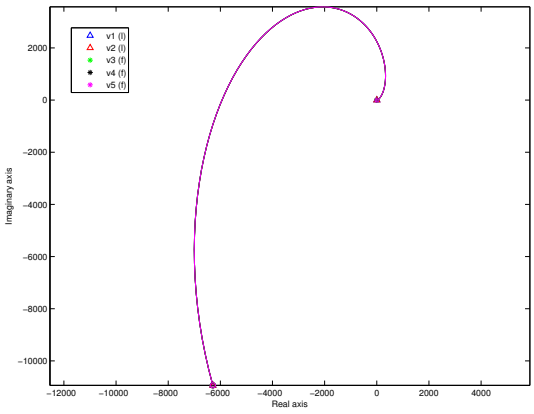


(a)

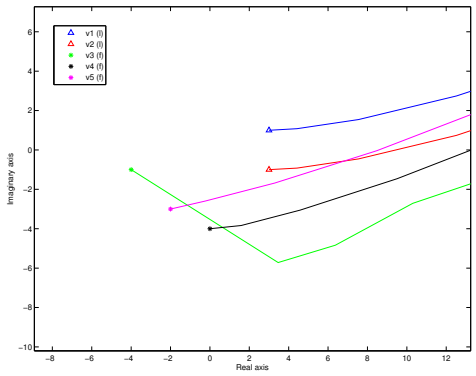


(b)

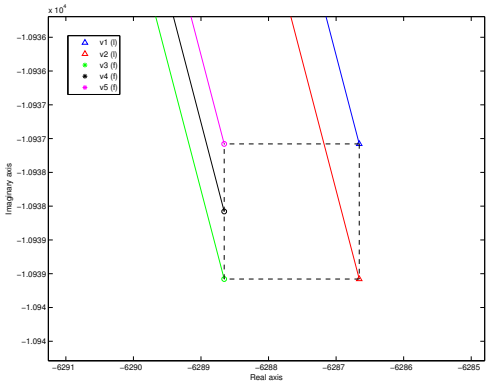
Figure 6.4: Double integrator dynamics case. Agents reaching a planar formation.



(a)



(b)



(c)

Figure 6.5: Double integrator dynamics case. Agents reaching a moving formation.

Chapter 7

Conclusion

In this work we have seen a new formation control approach that is based on complex weighted sensing digraphs and their Laplacian matrices. The control laws that this approach yields are locally and easily implementable with few information required from the other agents such as relative distance and leaders' speed or acceleration. In chapter 3 the theory developed in [29] has been fully presented both for the single-integrator kinematics and the double-integrator dynamics case. What resulted is that in both cases the control of the agents formation depends fundamentally on Ballantine's theorem, to which is related the permutation matrix problem. So, if an agents formation were to be controlled and its Laplacian matrix were to have unstable eigenvalues, they would be stabilized. In chapters 4 and 5 algorithms aimed to the stabilization of a Laplacian matrix have been discussed.

In chapter 4 the permutation matrix problem has been discussed. The problem is to find a permutation matrix P such that the matrix $\hat{L_{ff}} = PL_{ff}P^T$ has non-null leading principal minors. Two implementations have been yielded both based on the backtracking design technique. Even though the number of the solutions can be higher than one as experiments have shown, the worst case complexity of both implementations is very high as shown from the following expressions

Determinant-based algorithm

$$\mathrm{O}\left(\mathrm{p}(n)n!\right) = \mathrm{O}\left(\frac{1}{6}n^4 n!\right), \tag{7.1}$$

Gauss-based algorithm

$$\mathrm{O}\left(\mathrm{p}(n)n!\right) = \mathrm{O}\left(\frac{2}{3}n^3 n!\right). \tag{7.2}$$

Nonetheless, experimental results have shown that for random generated Lapalcian matrices the most of the times a relabelling of the nodes were not necessary.

In chapter 5 the stabilization problem has been discussed. System stabilization, turned out to be a well known inverse eigenvalue problem, the MIEP. The encountered multiplicative inverse eigenvalue problem consists in finding a complex diagonal matrix that multiplied for the unstable matrix L_{ff} yields a new stable matrix. Unfortunately this problem, brilliantly mathematically described by Friedland, has not an efficient algorithm in practice. To solve the problem, an algorithm has been presented which is derived from Ballantine's theorem. Nonetheless, the practical issues in complex numbers computation limit the order of the matrix to be stabilized. A second problem is that, even if a solution is found, the resulting stable eigenvalues cannot be chosen as desired. This results to be an issue for a practical use of the control technique. In fact, if the stable eigenvalues cannot be chosen, the agents could converge to a stable formation in unpredictable ways. In order to have a smooth and rapid convergence, eigenvalues should have a small imaginary part, in order to suffer from oscillations the less as possible and a high real part. For example, let us have the formation basis and the sensing digraph already seen in chapter 6. A possible Laplacian matrix is

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 4+1\iota & -5-3\iota & 5+6\iota & -4-4\iota & 0 \\ 0 & -5-2\iota & 5-4\iota & -4+22\iota & 4-16\iota \\ -5-2\iota & 0 & 0 & -4+10\iota & 9-8\iota \end{bmatrix}$$

which has eigenvalues

$$\sigma(L) = \{0, 0, -1.5268 + 18.3388\iota, 9.4885 - 0.7229\iota, 2.0383 + 2.3842\iota\}.$$

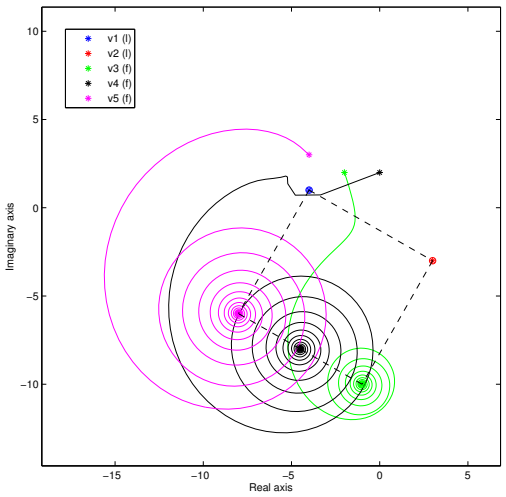


Figure 7.1: Example of a group of 5 double-integrator agents with a slow convergence to a planar formation.

A possible solution matrix for the stabilization of L_{ff} is

$$M = \begin{bmatrix} 0.0820 - 0.0984\iota & 0 & 0 \\ 0 & -10\iota & 0 \\ 0 & 0 & 0.2\iota \end{bmatrix}$$

yielded from Ballantine-based algorithm. The new eigenvalues of $-DL$ are

$$\sigma(-DL) = \{0, 0, -218.74 - 39.45\iota, -0.56 + 0.16\iota, -0.11 + 1.09\iota\}.$$

In figure (7.1) a group of 5 random agents modelled by the double-integrator dynamics is shown to converge to a planar formation with eigenvalues $\sigma(-DL)$. The system has been observed for a time interval $[0, 320]$ and $\gamma = 5$ has been used. As it can be seen, the agents converge to the right planar formation but the eigenvalues with the small real part make the convergence longer and many cycles forms before the follower agents converging to the final positions.

In summary, agents formation control via complex Laplacian is a powerful formation control approach in order to make a group of agents move in the plane in a coordinated and predetermined way. The algorithms discussed have their limits which can make difficult to apply the technique in practice. Still they can be used for further research, for example in practical applications such as collision avoidance and limited sensing capability. Other interesting problems to be address includes for example how to avoid collisions and maintain

the links by simply adjusting the complex weights in the control law [29].

Appendix A

Direct Matrices Operations

Let $A_1 \in \mathbb{F}^{(n_1 \times n_1)}, A_2 \in \mathbb{F}^{(n_2 \times n_2)}, \ldots, A_k \in \mathbb{F}^{(n_k \times n_k)}$ be k square matrices of different dimensions such that $n = \sum_{i=1}^{k} n_i$. The *matrix direct sum* of those matrices (see [26]),

$$A = A_1 \oplus A_2 \oplus \cdots \oplus A_k, \tag{A.1}$$

is the *block diagonal matrix* with the matrices A_i down the diagonal. That is,

$$A = \begin{bmatrix} A_1 & & 0 \\ & \ddots & \\ 0 & & A_k \end{bmatrix}, \tag{A.2}$$

where $A \in \mathbb{F}^{(n \times n)}$. As it can be seen, matrix A is nothing else than a matrix written in the Jordan canonical form. Hence, properties of that form holds for the result of a direct sum of matrices.

Let $A \in \mathbb{F}^{(mtimesn)}$ and $B \in \mathbb{F}^{(p \times q)}$ be two matrices. The *direct product* of A and B (see [26]), is defined to be the matrix $C \in \mathbb{F}^{(mp \times qn)}$ such that

$$C = A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}. \tag{A.3}$$

This matrix operation is also known as the *Kronecker product* or the *tensor product*. Let λ_i and β_j be the eigenvalues of A and B respectively. Then, the following properties hold:

- eigenvalues of $A \otimes B$ are $\{\lambda_i \beta_j\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$;

- eigenvalues of $(A \otimes I_n) + (I_m \otimes B)$ are $\{\lambda_i + \beta_j\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

Appendix B

Gauss Elementary Matrices

A *Gauss Elementary Matrix (Gauss transformation)*(see [35]) is defined as follows

$$\mathcal{M}_k = I - \mathbf{m}_k \mathbf{e}_k^T, \tag{B.1}$$

where I is the identity matrix, $\mathbf{e}_k = (0, \ldots, 1, \ldots, 0)^T$ is the k-th vector of the canonical basis and

$$\mathbf{m}_k = (0, \ldots, 0, m_{k+1,k}, m_{k+2,k}, \ldots, m_{kn})^T. \tag{B.2}$$

If we pre-multiply a square matrix A for a Gauss elementary matrix, we will have the following result:

$$\mathcal{M}_k A = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -m_{k+1,k} & 1 & & \\ & & -m_{k+2,k} & & 1 & \\ & & \vdots & & & \ddots \\ & & -m_{n,k} & & & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a}^1 \\ \vdots \\ \mathbf{a}^k \\ \mathbf{a}^{k+1} \\ \mathbf{a}^{k+2} \\ \vdots \\ \mathbf{a}^n \end{bmatrix} = \begin{bmatrix} \mathbf{a}^1 \\ \vdots \\ \mathbf{a}^k \\ \mathbf{a}^{k+1} - m_{k+1,k}\mathbf{a}^k \\ \mathbf{a}^{k+2} - m_{k+2,k}\mathbf{a}^k \\ \vdots \\ \mathbf{a}^n - m_{n,k}\mathbf{a}^k \end{bmatrix},$$

where matrix A has been represented by its vector rows $\mathbf{a}^i$. as it can be seen, $\mathcal{M}_k$ affects only the rows of A from k (index of the Gauss elementary matrix) to n.

Matrix $\mathcal{M}_k$ has the following properties:

1. the inverse of $\mathcal{M}_k$ is still a Gauss Elementary matrix

$$\mathcal{M}_k^{-1} = I + \mathbf{m}_k \mathbf{e}_k^T; \tag{B.3}$$

2. the product of the inverse of two Gauss Elementary matrices with increasing index, can be obtained in the following way

$$\mathcal{M}_k^{-1}\mathcal{M}_{k+r}^{-1} = I + \mathbf{m}_k\mathbf{e}_k^T + \mathbf{m}_{k+r}\mathbf{e}_{k+r}^T. \tag{B.4}$$

Gauss Elementary matrices are usually used in order to represent the *LU* factorization of a matrix.

Appendix C

Matrices and Vector Spaces

Let us have a matrix $A \in \mathbb{F}^{(m\times n)}$. Rows and columns of A can be seen as vectors which describe vector spaces (see [26]). The following definitions are related to that point of view.

- The *range* of A, denoted by range(A), is the set of all linear combination of the columns of A. That is,

$$\text{range}(A) = \{Ax | x \in \mathbb{F}^n\} \subseteq \mathbb{F}^m. \tag{C.1}$$

  If $A = [a_1\ a_2\ \ldots\ a_n]$, then range$(A) = \text{Span}(a_1, a_2, \ldots, a_n)$. The range of A is also called *column space* or *image space*.

- The *row space* of A, denoted by RS(A), is the set of all linear combinations of the rows of A. That is,

$$\text{RS}(A) = \{y^T A | y^T \in \mathbb{F}^m\} \subseteq \mathbb{F}^n. \tag{C.2}$$

  If $A = [b_1\ b_2\ \ldots\ b_m]$, then RS$(A) = \text{Span}(b_1, b_2, \ldots, b_m)$. Note that the row space of A equals the column space of the transpose of A, that is, RS$(A) = \text{range}(A^T)$.

- The *null space* or *kernel* of A, denoted by ker(A), is the set of all solutions to the homogeneous equation $Ax = 0$. That is,

$$\ker(A) = \{x_{n\times 1} | Ax = 0\} \subseteq \mathbb{F}^n. \tag{C.3}$$

  The dimension of the null space is called the *nullity* of A and is denoted by null(A).

- The *left-hand null space* of A is the set of all solutions to the left-hand homogeneous system $y^T A = \mathbf{0}^T$. That is,

$$\ker(A^T) = \{y_{m\times 1} | A^T y = 0\} \subseteq \mathbb{F}^m. \tag{C.4}$$

- The *rank* of A, denoted by $\text{rank}(A)$, is the number of leading entries in the reduced row echelon form of A (or any row echelon form of A).

Let us have a matrix $A \in \mathbb{F}^{(m\times n)}$. The following properties hold (see [26]).

1. The range of A is a subspace of $\mathbb{F}^m$.
2. The columns of A corresponding to the pivot columns in the reduced row echelon form of A (or any row echelon form of A) give a basis for $\text{range}(A)$. Let $v_1, v_2, \ldots, v_k \in \mathbb{F}^m$. If matrix $A = [v_1\ v_2\ \ldots\ v_k]$, then a basis for $\text{range}(A)$ will be a linearly independent subset of $v_1, v_2, \ldots, v_k$ having the same span.
3. $\dim(\text{range}(A)) = \text{rank}(A)$.
4. The kernel of A is a subspace of $\mathbb{F}^n$.
5. If the reduced row echelon form of A (or any row echelon form of A) has k pivot columns[1], then $\text{null}(A) = n - k$.
6. If two matrices A and B are row equivalent, then $\text{RS}(A) = \text{RS}(B)$.
7. The row space A is a subspace of $\mathbb{F}^n$.
8. The pivot rows[2] in the reduced row echelon form of A (or any row echelon form of A) give a basis for $\text{RS}(A)$.
9. $\text{RS}(A) = \text{rank}(A)$.
10. $\text{rank}(A) = \text{rank}(A^T)$.
11. *Dimension Theorem* For any $A \in \mathbb{F}^{(m\times n)}$,

$$n = \text{rank}(A) + \text{null}(A).$$

Similarly,

$$m = \dim(\text{RS}(A)) + \text{null}(A^T).$$

[1] A pivot column in a matrix is a column which contains a pivot element.
[2] A pivot row in a matrix is a row which contains a pivot element.

12. A vector $b \in \mathbb{F}^m$ is in range(A) if and only if the equation $Ax = b$ has a solution. So range$(A) = \mathbb{F}^m$ if and only if the equation $Ax = b$ has a solution for every $b \in \mathbb{F}^m$.

13. A vector $a \in \mathbb{F}^n$ is in $\mathbb{RS}(A)$ if and only if the equation $A^T y = a$ has a solution. So RS$(A) = \mathbb{F}^n$ if and only if the equation $A^T y = a$ has a solution for every $a \in \mathbb{F}^n$.

14. If a is a solution to the equation $Ax = b$, then $a + v$ is also a solution for any $v \in \ker(A)$.

15. If $A \in \mathbb{F}^{m \times n}$ is rank 1, then there are vectors $v \in \mathbb{F}^m$ and $u \in \mathbb{F}^n$ so that $A = vu^T$.

16. If $A \in \mathbb{F}^{(m \times n)}$ is rank k, then A is a sum of k rank 1 matrices. That is, there exist $A_1, \ldots, A_k$ with $A = A_1 + A_2 + \cdots + A_k$ and rank$(A_i) = 1$, for $i = 1, \ldots, k$.

17. The following are all equivalent statements about a matrix $A \in \mathbb{F}^{(m \times n)}$.

    (a) The rank of A is k.

    (b) dim(range(A)) $= k$.

    (c) The reduced row echelon form of A has k pivot columns.

    (d) A row echelon form of A has k pivot columns.

    (e) The largest number of linearly independent columns of A is k.

    (f) The largest number of linearly independent rows of A is k.

18. *Rank Inequalities* (Unless specified otherwise, assume that $A, B \in \mathbb{F}^{(m \times n)}$.)

    (a) rank$(A) \leq \min(m, n)$.

    (b) If a new matrix B is created by deleting rows and/or columns of a matrix A, then rank$(B) \leq$ rank(A).

    (c) rank$(A + B) \leq$ rank$(A) +$ rank(B).

    (d) If A has a $p \times q$ submatrix of 0s, then rank$(A) \leq (m - p) + (n - q)$.

    (e) If $A \in \mathbb{F}^{(k \times n)}$, then

    $$\text{rank}(A) + \text{rank}(B) - k \leq \text{rank}(AB) \leq \min\{\text{rank}(A), \text{rank}(B)\}.$$

19. *Rank Equalities*

(a) If $A \in \mathbb{C}^{(m\times n)}$, then $\text{rank}(A^*) = \text{rank}(A^T) = \text{rank}(\bar{A}) = \text{rank}(A)$.

(b) If $A \in \mathbb{C}^{(m\times n)}$, then $\text{rank}(A^*A) = \text{rank}(A)$. If $A \in \mathbb{R}^{(m\times n)}$, then $\text{rank}(A^TA) = \text{rank}(A)$.

(c) Rank is unchanged by left or right multiplication by a nonsingular matrix. That is, if $A \in \mathbb{F}^{(n\times n)}$ and $B \in \mathbb{F}^{(m\times m)}$ are nonsingular, and $M \in \mathbb{F}^{(m\times n)}$, then

$$\text{rank}(AM) = \text{rank}(M) = \text{rank}(MB) = \text{rank}(AMB).$$

(d) If $A, B \in \mathbb{F}^{(m\times n)}$, then $\text{rank}(A) = \text{rank}(B)$ if and only if there exist nonsingular matrices $X \in \mathbb{F}^{(m\times m)}$ and $Y \in \mathbb{F}^{(n\times n)}$ such that $A = XBY$ (i.e., if and only if A is equivalent to B).

(e) If $A \in \mathbb{F}^{(m\times n)}$ has rank k, then $A = XBY$, for some $X \in \mathbb{F}^{(m\times k)}$, $Y \in \mathbb{F}^{(k\times n)}$, and nonsingular $B \in \mathbb{F}^{(k\times k)}$.

(f) If $A_1 \in \mathbb{F}^{(n_1\times n_1)}, \ldots, A_k \in \mathbb{F}^{(n_k\times n_k)}$, then $\text{rank}(A_1 \oplus \cdots \oplus A_k) = \text{rank}(A_1) + \cdots + \text{rank}(A_k)$.

20. Let $A, B \in \mathbb{F}^{(n\times n)}$ with A similar to B.

(a) A is equivalent to B.

(b) $\text{rank}(A) = \text{rank}(B)$.

(c) $\text{tr}(A) = \text{tr}(B)$.

21. Equivalence of matrices is an equivalence relation on $\mathbb{F}^{(m\times n)}$.

22. Similarity of matrices is an equivalence relation on $\mathbb{F}^{(n\times n)}$.

23. If $A \in \mathbb{F}^{(m\times n)}$ and $\text{rank}(A) = k$, then A is equivalent to $\left[\begin{smallmatrix} I_k & 0 \\ 0 & 0 \end{smallmatrix}\right]$, and so any two matrices of the same size and rank are equivalent.

24. If $A \in \mathbb{R}^{(n\times n)}$, then for any $x \in \text{RS}(A)$ and any $y \in \ker(A)$, $x^Ty = 0$. So the row space and kernel of a real matrix are orthogonal to one another.

Appendix D

Faulhaber's Formula

The general formula for the *power sum* of the first n positive integers was named after Johann Faulhaber (see [41]). He published in a 1631 edition of *Academiae Algebrae* the derivation of the first seventeen polynomials of the form

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + 3^p + \cdots + n^p,$$

but the general formula was written in a closed form when Bernoulli numbers where discovered. The formula is

$$\sum_{k=1}^{n} k^p = H_{n,-p} = \frac{1}{p+1} \sum_{i=1}^{p+1} (-1)^{\delta_{ip}} \binom{p+1}{i} B_{p+1-i} n^i, \tag{D.1}$$

where $H_{n,r}$ is a generalized *harmonic number*, δ_{ip} is the *Kronecker delta*, $\binom{n}{i}$ is a *binomial coefficient* and B_i is the i-th *Bernoulli number*.

The sums for $p = 1, \ldots, 7$ result in:

$\boxed{p = 1}$ $$\sum_{k=1}^{n} k = \frac{1}{2}(n^2 + n)$$

$\boxed{p = 2}$ $$\sum_{k=1}^{n} k^2 = \frac{1}{6}(2n^3 + 3n^2 + n)$$

$\boxed{p = 3}$ $$\sum_{k=1}^{n} k^3 = \frac{1}{4}(n^4 + 2n^3 + n^2)$$

$\boxed{p = 4}$ $$\sum_{k=1}^{n} k^4 = \frac{1}{30}(6n^5 + 15n^4 + 10n^3 - n)$$

$\boxed{p=5}$ $$\sum_{k=1}^{n} k^5 = \frac{1}{12}(2n^6 + 6n^5 + 5n^4 - n^2)$$

$\boxed{p=6}$ $$\sum_{k=1}^{n} k^6 = \frac{1}{42}(6n^7 + 21n^6 + 21n^5 - 7n^3 + n)$$

$\boxed{p=7}$ $$\sum_{k=1}^{n} k^7 = \frac{1}{24}(3n^8 + 12n^7 + 14n^6 - 7n^4 + 2n^2).$$

Appendix E

Matlab: The Laplacian matrix of a Weighted Digraph

The Laplacian matrix of a graph can be defined in different ways, but we are interested in the definition already seen in sec. (2.2.4) concerning weighted digraphs. A complex digraph is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ we associate a complex number $w_{ij} \in \mathbb{C}$ to each edge (j, i). His Laplacian matrix can be defined as follows:

$$L(i,j) = \begin{cases} -w_{ij} & \text{if } i \neq j \text{ and } j \in N_i \\ 0 & \text{if } i \neq j \text{ and } j \notin N_i \\ \sum_{j \in N_i} w_{ij} & \text{if } i = j, \end{cases} \tag{E.1}$$

where $N_i = \{j : (j, i) \in \mathcal{E}\}$ is the in-neighbor set of node i. For our purpose, we need to generate Laplacian matrices that represent sensing digraphs of formations with the following characteristics:

1. the leaders are in number of 2,
2. every follower node has to be 2-reachable from at least one leader node,
3. the followers can sense a different number of other followers. The following cases are considered:
   (a) 1 in-neighbor follower,
   (b) 2 in-neighbor followers,
   (c) 3 in-neighbor followers,
   (d) each follower is an in-neighbor for every follower.

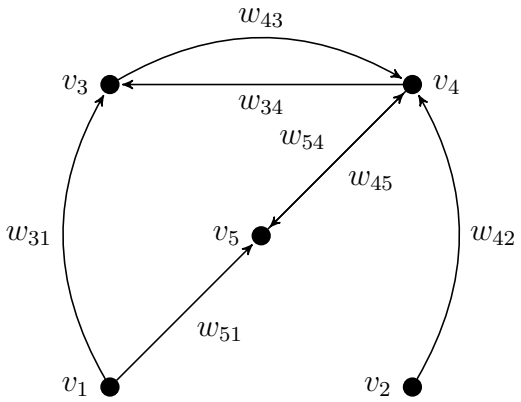


Figure E.1: Sensing digraph represented by the Laplacian matrix (E.3).

Without loss of generality, we can label the 2 co-leader nodes as 1 and 2. Reminding that, leaders do not have incoming edges, the Laplacian matrix of an agents formation with 2 co-leaders can be generally represented, as we have already seen in sec.(3), in the following form:

$$L = \left[\begin{array}{c|c} 0_{2\times 2} & 0_{2\times(n-2)} \\ \hline L_{lf} & L_{ff} \end{array}\right], \tag{E.2}$$

where:

- L_{lf} is the leader-follower sub-matrix,
- L_{ff} is the follower-follower sub-matrix.

For instance, the Laplacian matrix of a formation of 5 agents, 2 of which are co-leaders, could be:

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -w_{31} & 0 & \alpha_{33} & -w_{34} & 0 \\ 0 & -w_{42} & -w_{43} & \alpha_{44} & -w_{45} \\ -w_{51} & 0 & 0 & -w_{54} & \alpha_{55} \end{bmatrix}, \tag{E.3}$$

where $\alpha_{kk} = \sum_{j\in N_k} w_{kj}$. The sensing digraph represented by the Laplacian (E.3) is depicted in figure (E.1). As we can see, each follower has at least two in-neighbor nodes. The most important thing is that the digraph is 2-reachable.

The sub-digraph formed by the followers is represented by the sub-matrix L_{ff} that is, in this case, a tridiagonal form:

$$L_{ff} = \begin{bmatrix} \alpha_{33} & -w_{34} & 0 \\ -w_{43} & \alpha_{44} & -w_{45} \\ 0 & -w_{54} & \alpha_{55} \end{bmatrix}. \tag{E.4}$$

We can generate Laplacian matrices representing sensing digraphs with different connections. For our purpose, we will generate Laplacians with the aforementioned characteristics and give them a standard structure. Hence, we will choose which are the possible connections among agents over the aforementioned constrains. Remember that constraints 1) and 2) are common to every digraph we want. The real difference among the Laplacians is given from number 3). The leader-follower sub-matrix is a $(n-2)\times 2$ matrix representing the incoming edges from leaders to followers. When the ijth element of L_{lf} is non-zero, it means that the ith follower is sensing the jth leader. We can choose to connect all followers to each leader or simply to just one of them (in order to have 2-reachable followers). Hence, L_{lf} will have no zero elements at all or one zero element per row respectively, as shown below:

$$L_{lf} = \begin{bmatrix} -w_{31} & -w_{32} \\ -w_{41} & -w_{42} \\ \vdots & \vdots \\ \vdots & \vdots \\ -w_{n1} & -w_{n2} \end{bmatrix} \tag{E.5a}$$

$$L_{lf} = \begin{bmatrix} -w_{31} & 0 \\ 0 & -w_{42} \\ \vdots & \vdots \\ -w_{n-1,1} & 0 \\ 0 & -w_{n2} \end{bmatrix}. \tag{E.5b}$$

The follower-follower sub-matrix, is an $(n-2)\times(n-2)$ matrix, representing the connections among followers. That is, when the ijth element is non-zero, then it means that the ith follower is sensing the jth follower. We want to generate Laplacians where L_{ff} has the characteristics in 3) and for each of them we must make assumptions about the possible connections:

3a) when each follower can sense only another follower, the L_{ff} matrix has only 2 non-zero elements per row. We can assume that each follower

senses its predecessor in the L matrix. Follower number 3 can be assumed to sense follower n as it has not a predecessor (node number 2 is a leader). Thus, L_{ff} will be of the form:

$$L_{ff} = \begin{bmatrix} \alpha_{33} & 0 & \dots & 0 & -w_{3n} \\ -w_{43} & \alpha_{44} & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -w_{n,n-1} & \alpha_{nn} \end{bmatrix} \tag{E.6}$$

3b) when each follower can sense only two followers, the L_{ff} matrix has 3 non-zero elements per row. We can assume that each follower senses its predecessor and its successor in the L matrix. We can also assume that follower 3 senses follower n as its predecessor, and that follower n senses follower 3 as its successor. Thus, L_{ff} will be of the form:

$$L_{ff} = \begin{bmatrix} \alpha_{33} & -w_{34} & 0 & \dots & 0 & -w_{3n} \\ -w_{43} & \alpha_{44} & -w_{45} & 0 & \dots & 0 \\ 0 & -w_{54} & \alpha_{55} & -w_{56} & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \vdots \\ -w_{n3} & 0 & \dots & 0 & -w_{n,n-1} & \alpha_{nn} \end{bmatrix} \tag{E.7}$$

3c) when each follower can sense only 3 followers, the L_{ff} matrix has 4 non-zero elements per row. We can assume that each follower senses its predecessor and its 2 successors in the L matrix. We can also assume that follower 3 senses follower n as its predecessor and follower n follower 3 and 4 as its successors. Thus, L_{ff} will be of the form:

$$L_{ff} = \begin{bmatrix} \alpha_{33} & -w_{34} & -w_{35} & 0 & \dots & 0 & -w_{3n} \\ -w_{43} & \alpha_{44} & -w_{45} & -w_{46} & 0 & \dots & 0 \\ 0 & -w_{54} & \alpha_{55} & -w_{56} & -w_{57} & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & & & & & \\ & & & & & & \\ -w_{n3} & -w_{n4} & 0 & \dots & 0 & -w_{n,n-1} & \alpha_{nn} \end{bmatrix} \tag{E.8}$$

3d) when each follower can sense every other follower it means that the follower-follower sub-matrix L_{ff} has no zero off-diagonal elements. Consequently, L_{ff} will be as follows:

$$L_{ff} = \begin{bmatrix} \alpha_{33} & -w_{34} & \dots & \dots & -w_{3n} \\ -w_{43} & \alpha_{44} & -w_{45} & \dots & -w_{4n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ -w_{n3} & \dots & \dots & -w_{n,n-1} & \alpha_{nn} \end{bmatrix} \tag{E.9}$$

We can generate Laplacian matrices like the ones described below by the **MATLAB®** implementation shown in file E.1.

File E.1: **lm.m**.

```
%LM - Laplacian Matrix.
% m  = matrix order,
% in = # of in-neighbor followers of each follower.
%      in = 1 -> each follower senses its predecessor,
%      in = 2 -> each follower senses its predecessor
%                and its successor,
%      in = 3 -> each follower senses its predecessor
%                and its two successors.
%      in = 0 -> each follower senses every other follower.
%NOTE: for each value of the variable in Lff and,
%      consequently, L must
%have a minimum dimension. Hence, for the cases described
%above, we have:
%      for in = 1,  m ≥ 4,
%      for in = 2,  m ≥ 5,
%      for in = 3,  m ≥ 6,
%      for in = 0,  m ≥ 3.

function  L = lm(m,in)

L = zeros(m);

switch in
    case 1
        if(m < 4)
            disp('Wrong matrix dimension, m must be ≥ 4');
            return;
```

```
28          end
29          L(3,m) = -(rand + rand*1i);
30          x = -ones(m,1);
31          for h = 3 : m-1
32              L(h,randi([1 2],1,1)) = -(rand + rand*1i);
33
34              L(h+1,h)= -(rand + rand*1i);
35
36              L(h,h) = L(h,:)*x;
37          end
38          L(m,randi([1 2],1,1)) = -(rand + rand*1i);
39          L(m,m) = L(m,:)*x;
40      case 2
41          if(m < 5)
42              disp('Wrong matrix dimension, m must be ≥ 5');
43              return;
44          end
45          L(3,m) = -(rand + rand*1i);
46          L(3,4) = -(rand + rand*1i);
47          L(3,randi([1 2],1,1)) = -(rand + rand*1i);
48
49          L(m,3) = -(rand + rand*1i);
50          L(m,m-1) = -(rand + rand*1i);
51          L(m,randi([1 2],1,1)) = -(rand + rand*1i);
52
53          x = -ones(m,1);
54          for h = 4 : m-1
55              L(h,randi([1 2],1,1)) = -(rand + rand*1i);
56
57              L(h,h-1)= -(rand + rand*1i);
58              L(h,h+1)= -(rand + rand*1i);
59
60              L(h,h) = L(h,:)*x;
61          end
62          L(3,3) = L(3,:)*x;
63          L(m,m) = L(m,:)*x;
64      case 3
65          if(m < 6)
66              disp('Wrong matrix dimension, m must be ≥ 6');
67              return;
68          end
69          L(3,4)   = -(rand + rand*1i);
70          L(3,5)   = -(rand + rand*1i);
71          L(3,m)   = -(rand + rand*1i);
72          L(m,3)   = -(rand + rand*1i);
```

```
73          L(m,4)    = -(rand + rand*1i);
74          L(m,m-1) = -(rand + rand*1i);
75          x = -ones(m,1);
76          for h = 4 : m-1
77              L(h,randi([1 2],1,1)) = -(rand + rand*1i);
78
79              L(h,h-1) = -(rand + rand*1i);
80              L(h,h+1) = -(rand + rand*1i);
81              L(h,h+2) = -(rand + rand*1i);
82
83              L(h,h) = L(h,:)*x;
84          end
85          L(3,randi([1 2],1,1)) = -(rand + rand*1i);
86          L(m,randi([1 2],1,1)) = -(rand + rand*1i);
87          L(3,3) = L(3,:)*x;
88          L(m,m) = L(m,:)*x;
89      case 0
90          if(m < 3)
91              disp('Wrong matrix dimension, m must be ≥ 3');
92              return;
93          end
94          L(3:m,:) = complex(rand(m-2,m),rand(m-2,m));
95          for h = 3 : m
96              x = ones(m,1);
97              x(h) = 0;
98              lhh = L(h,:)*x;
99              L(h,:) = -L(h,:);
100             L(h,h) = lhh;
101         end
102     otherwise
103         disp('Wrong number of in-neighbor followers');
104 end
105 end
```

A different way to obtain a complex Laplacian matrix involves the adjacency matrix of the digraph itself. When a multi-agent formation is given with the corresponding formation basis ξ desired, we need a Laplacian matrix L for which conditions $L\xi = 0$ and $L\mathbf{1}_n = 0$ hold. This is necessary in order that L be suitable for the formation control laws given in chapter 3, so that the desired planar formation is reachable. To do so, we can exploit the fact that, for a formation basis, the condition that ξ must be an eigenvector for L can be written as

$$\sum_{j \in \mathcal{N}_i^-} w_{ij}(\xi_j - \xi_i) = 0. \tag{E.10}$$

For each row, all weights are randomly generated but the last one, that is computed via equation (E.10) so that $L\xi = 0$ is verified. File E.2 shows a possible Matlab implementation of the Laplacian matrix generator. The function *lmg* has inputs A and z, that are the adjacency matrix of the graph's formation and the desired formation basis. The adjacency used is defined as follows

$$a_{ij} = \begin{cases} 1 & \text{if } (v_j, v_i) \in \mathcal{A}, \\ 0 & \text{if } (v_j, v_i) \notin \mathcal{A}. \end{cases} \tag{E.11}$$

What function *lmg* does is to search for the ones in A and assign to L random weights w_{ij} in the same position of them. Once this process is done diagonal elements are built so that $L\mathbf{1}_n = 0$ holds.

File E.2: **lmg.m**.

```
1  %LMG - Laplacian Matrix Generator.
2
3  %Function lmg takes an adjacency matrix and a formation
4  %basis z so that the complex Laplacian matrix L which
5  %satisfies conditions L1=0 and Lz=0 is randoml built.
6
7  function L = lmg(A,z)
8
9  [m,n] = size(A);
10
11 mini = -10;
12 maxi = 10;
13
14 %Verifying if the adjacency matrix is square.
15 if(m ~= n)
16     disp('Adjacency matrix is not square.');
17     L = 0;
18     return;
19 end
20
21 %Verifying if leaders have incoming edges.
22 for i = 1 : 2
23     for j = 1 : n
24         if(A(i,j) ~= 0)
25 disp('Wrong sensing digraph. Leaders have incoming edges.');
26             L = 0;
27             return;
28         end
```

```
    end
end

L = zeros(m);

%Counting how many weights there will be in each row
%of the Laplacian matrix.
count = zeros(n-2,1);
for h = 1 : n-2
    count(h) = nnz(A(h+2,:));
end

%Assigning complex random weights such as matrix
%L has z as eigenvector.
for i = 3 : n
        par_sum = 0;
        s = count(i-2);
        for j = 1 : n
            if(s == 0)
                break;
            end
             if(A(i,j) ~= 0)
                 if(s == 1)
                    y = solve('y*(z(i) - z(j)) + par_sum');
                    L(i,j) = eval(y);
                 else
                    %L(i,j) = -(rand + rand*1i);
L(i,j) = -(randi([mini maxi],1,1) + randi([mini maxi],1,1)*1i);
                    par_sum = par_sum + L(i,j)*(z(i) - z(j));
                 end
                 s = s - 1;
             end
        end
end

%Assigning diagonal elements such as a vector of ones
%is an eigenvalue for L.
x = -ones(n,1);
for i = 3 : n
    L(i,i) = L(i,:)*x;
end

end
```

Appendix F

Matlab: Code from Chapter 4

In chapter 4 we have seen the problem of finding a permutation matrix P such that the matrix

$$\hat{L}_{ff} = PL_{ff}P^T \tag{F.1}$$

has non-null principal minors, and backtracking has been used in order to solve the problem. Backtracking algorithm for the permutation matrix problem has been implemented considering two different ways to compute the bounding functions:

1. B_is have been implemented computing the determinant of a sub-matrix of order i;

2. B_is have been implemented as the i-th step of Gaussian elimination with diagonal pivoting.

Nonetheless, for each of them, the same T function have been tried in order to study the behavior of the algorithm with respect to the problem being solved.

The actual implementations and simulations have been made in the Matlab environment. In both cases, the algorithm has been split in two files:

- the *main function*, named *pmsgx.m* or *pmsdx.m*. The task of the main file is to generate the needed set up in order to make the algorithm work and to return the solution found;

- the *recursive function* called from the main one that implements the recursive backtrack process. Files are called *nnpsgx.m* and *nnpsdx.m*.

> In file names d, g and x are:

  - d identifies the determinant-based implementation, that is the one which uses bounding functions number 1);

- g identifies the gauss-based implementation, that is that is the one which uses bounding functions number 2);
- x is a number that indicates different T functions used.

In the following subsections implementations both for the determinant-based algorithm (subsection F.1) and the Gauss-based algorithm (subsection F.2) are shown.

F.1 Determinant-based backtrack.Algorithm 1

The algorithm aimed to find a permutation matrix P in order to verify equation (F.1) has been implemented with two different bounding functions.

Files *nnpsd1.m* (F.2) and *nnpsd2.m* (F.3) implements the backtracking procedure using bounding functions that compute the determinant of the principal minors of the matrix. As it can be seen from the code, the functions call themselves in a recursive way, that is the natural implementation for the backtrack procedure. Function F.2 search for the nearest solution to an identity matrix, while function F.3 search randomly in the solutions space. In fact, as it can be seen in row 23 of file F.3 the sequence of node generation is given from the function *irp.m* that simply creates a randomly ordered vector of integers whose maximum is the input integer. Hence, T_{d1} defines a static state space organization tree while T_{d2}, because of the randomness of the search, defines a dynamic state space organization tree. Note that functions *nnpsdx* compute the determinant of the leading principal minors via function *dtr* other than exploit the native Matlab function *det*. This is for experimental purpose. In fact, Matlab function *det* is not implemented in Matlab code and then its speed execution cannot be fairly compared to other Matlab code-based function. Hence for the tests shown in chapter 4 we needed to reimplement that function in Matlab code.

File F.1 is the main function which call the recursive function desired. In the code it calls function *nnpsd1.m* but it could have called function *nnpsd1.m* as well. In fact, *psmd1.m* and *pmsd2.m* are the same function.

File F.1: **pmsd1.m**

```
1 %PMSD1 - Permutation Matrix Solver with Determinant function.
2 %The following function solves the problem of finding the
3 %permutation matrix P, such that matrix A has all non-zero
```

```
%leading principal minors:
%           A = P*Lf*P'.

%The algorithm has been implemented using the function dtr
%in the core function nnpsd1.
%NOTE: dtr has been implemented with the Gauss elimination
%algorithm  with column pivoting.


function P = pmsd1(Lf)

[m,n] = size(Lf);

%Permutations vector.
b = zeros(m,1);
for i = 1 : m
    b(i) = i;
end

p = 0;
s = 1;

[b,p] = nnpsd1(Lf,m,b,p,s);

if(p == 0)
    disp('The permutation Matrix P does not exist');
    P = zeros(m);
    return;
else
    I = eye(m);
    P = zeros(m);
    for j = 1 : m
        P(j,:) = I(b(j),:);
    end
end
end
```

File F.2: **nnpsd1.m**

```
%NNPSD1 — Non—null leading Principal Minors Solver, with
%         dtr function.
%Function nnpsd1 is the core function of the backtracking
%algorithm used to find a permutation matrix P such that
%matrix A has all non—zero leading principal minors:
```

```
%          A = P*Lf*P'.
%The function nnpsd1 has been designed using recursion
%principle. In order to compute the value of each
%determinant, dtr function has been implemented using
%Gauss with partial pivoting. The solution space is
%searched using for each call an ascending  ordered
%sequence from the set {1,2,...,n}, where n is the order
%of input matrix Lf.

function [per,p] = nnpsd1(Lf,m,b,p,s)

D = Lf;
per = b;

%Pivot selection over the matrix diagonal.
for i = s : m+1

    d = dtr(D(1:s,1:s));

    if(abs(d) > eps)

        if(s == m)
            p = 1;
            return;
        end

        [per,p] = nnpsd1(D,m,per,p,s+1);

        if(p == 1)
            return;
        else
            if(p == 0)
                if(i == m+1)
                    per = b;
                    return;
                else
                    per = b;
                end
            end
        end
    else
         if(i < m+1)
            r = per(s);
            per(s) = per(i);
            per(i) = r;
```

```
            D = excn(D,s,i);
            p = 0;
        else
            p = 0;
            return;
        end
    end
end
end
```

File F.3: **nnpsd2.m**

```
%NNPSD2 - Non-null leading Principal Minors Solver, with
%         dtr function.
%Function nnpsd1 is the core function of the backtracking
%algorithm  used to find a permutation matrix P such that
%matrix A has all   non-zero leading principal minors:
%          A = P*Lf*P'.
%The function nnpsd1 has been designed using recursion
%principle. In order to compute the value of each
%determinant, dtr function has been implemented using
%Gauss with partial pivoting. The solution space is
%searched using for each call a randomly  orderes
%sequence from the set {1,2,...,n}, where n is the order
%of input matrix Lf.

function [per,p] = nnpsd2(Lf,m,b,p,s)

D = Lf;
per = b;


%-----------------------------------------%
%Choose randomly the sequence of the
%pivots among the diagonal elements.
sequence = irp(m-s+1);
%-----------------------------------------%

%-Pivot selection over the matrix diagonal-%
for i = s : m

    index = sequence(i-s+1) + (s-1);
    r = per(s);
    per(s) = per(index);
```

```
33      per(index) = r;
34      D = excn(D,s,index);
35
36      d = dtr(D(1:s,1:s));
37
38      if(abs(d) > eps)
39
40          if(s == m)
41              p = 1;
42              return;
43          end
44
45          [per,p] = nnpsd2(D,m,per,p,s+1);
46
47          if(p == 1)
48              return;
49          else
50              if(p == 0)
51                  if(i == m)
52                      per = b;
53                      return;
54                  else
55                      per = b;
56                  end
57              end
58          end
59      else
60          if(i == m)
61              p = 0;
62              return;
63          end
64      end
65  end
66  end
```

F.2 Gauss-based Backtrack.Algorithm 2

The algorithm aimed to find a permutation matrix P in order to verify equation (F.1) has been also implemented with bounding functions based on Gaussian elimination (see chapter 4). Matlab implementations can be seen in the files below. As in the case of the determinant-based algorithm, the implementation has been split in two files. The main function, *pmsgx.m*, sets up some global

variables and calls function *nnpsgx.m* that is the function which implements the backtracking procedure. The x is a number that refers to the T function implemented within the file. As it can be seen in subsection (4.2.2), there are five T_{gx} functions used:

- file *nnpsg1.m* (F.5) implements T_{g1};
- file *nnpsg2.m* (F.6) implements T_{g2};
- file *nnpsg3.m* (F.7) implements T_{g3};
- file *nnpsg4.m* (F.8) implements T_{g4};
- file *nnpsg5.m* (F.9) implements T_{g5}.

The main function is the same for each function *nnpsgx*, then only one has been listed. It can be seen in (F.4).

File F.4: **pmsg1.m**

```
%PMSG1 - Permutation Matrix Solver.
%The following function solves the problem of
%finding the permutatio matrix P, such that the
%relation below is satisfied.
%          A = P*Lf*P'.

%PMSG1 exploits the recursive function nnpsg1.


function [P,A,U] = pmsg1(Lf)

[m,n] = size(Lf);

%Permutations vector.
b = zeros(m,1);
for i = 1 : m
    b(i) = i;
end

p = 0;
L = eye(m);
U = zeros(m);
s = 1;

[b,p,L,U] = nnpsg1(Lf,m,b,L,p,s);
```

```
26
27 if(p == 0)
28     disp('The permutation Matrix P does not exist');
29     P = zeros(m);
30     A = zeros(m);
31     return;
32 else
33     A = L*U;
34     I = eye(m);
35     P = zeros(m);
36     for j = 1 : m
37         P(j,:) = I(b(j),:);
38     end
39 end
40 end
```

File F.5: **nnpsg1.m**

```
1  %NNPSG1 - Non-nnull pivot selector,
2  %The following function is aimed to implement a step of the
3  %Gaussian elimination with diagonal pivoting. The pivot
4  %element is searched only along the diagonal. Consecutive steps
5  %are executed via recursive calls.
6
7  %Function nnpsg1 implements Tg1 as searching method for the
8  %pivot.
9
10 function [per,p,T,U] = nnpsg1(Lf,m,b,L,p,s)
11
12 U = Lf;
13 T = L;
14 per = b;
15
16 %-------------------------------------------------------------%
17 if(((m-s) == 0) && (abs(Lf(s,s)) > eps))
18     p = 1;
19     return;
20 end
21
22 if(((m-s) == 0) && (abs(Lf(s,s)) < eps))
23     p = 0;
24     return;
25 end
26 %-------------------------------------------------------------%
```

```
27
28 %-----Pivot selection over the matrix diagonal.----------%
29 for i = s : m
30     if(abs(Lf(i,i)) > eps)
31         if(i ~= s)
32             %-----------------------------%
33             U = excn(U,s,i);
34             %-----------------------------%
35             r = per(s);
36             per(s) = per(i);
37             per(i) = r;
38             %-----------------------------%
39             t = T(s,1:s-1);
40             T(s,1:s-1) = T(i,1:s-1);
41             T(i,1:s-1) = t;
42             %-----------------------------%
43         end
44         %-------------------------------------------------------%
45         %NOTE: (s,s) is the position of the current pivot
46         %to be used in the elimination process
47         %(triangularization).
48
49         %Gauss elimination over the column s.
50         l = ones(m-s+1,1);
51         for h = s+1 : m
52             l(h-s+1) = U(h,s) / U(s,s);
53             U(h,s:m) = U(h,s:m) - l(h-s+1)*U(s,s:m);
54         end
55         T(s:m,s) = l;
56
57         %-----Recursive Call--------------------------------%
58         [per,p,T,U] = nnpsg1(U,m,per,T,p,s+1);
59         %-------------------------------------------------------%
60
61         if(p == 1)
62             return;
63         else
64             if(p == 0)
65                 if(i == m)
66                     return;
67                 else
68                     U = Lf;
69                     T = L;
70                     per = b;
71                 end
```

```
            end
        end
    else
        if((i == m) && (abs(Lf(i,i) < eps)))
            p = 0;
            return;
        end
    end
end
end
```

File F.6: **nnpsg2.m**

```
%NNPSG1 - Non-nnull pivot selector,
%The following function is aimed to implement a step of the
%Gaussian elimination with diagonal pivoting. The pivot
%element is searched only along the diagonal. Consecutive steps
%are executed via recursive calls.

%Function nnpsg2 implements Tg2 as searching method for the
%pivot.

function [per,p,T,U] = nnpsg2(Lf,m,b,L,p,s)

U = Lf;
T = L;
per = b;

%-------------------------------------------------------%
if(((m-s) == 0) && (abs(Lf(s,s)) > eps))
    p = 1;
    return;
end

if(((m-s) == 0) && (abs(Lf(s,s)) < eps))
    p = 0;
    return;
end
%-------------------------------------------------------%

%-------------------------------------------------------%
%Choose randomly the sequence of the pivots among
%the diagonal elements.
sequence = irp(m-s+1);
```

```
32
33 %-----Pivot selection over the matrix diagonal.---%
34 for i = s : m
35
36     index = sequence(i-s+1) + (s-1);
37
38     %-------------------------------------------------%
39     if(abs(Lf(index,index)) > eps)
40         if(index ~= s)
41             %-----------------------------%
42             U = excn(U,s,index);
43             %-----------------------------%
44             r = per(s);
45             per(s) = per(index);
46             per(index) = r;
47             %-----------------------------%
48             t = T(s,1:s-1);
49             T(s,1:s-1) = T(index,1:s-1);
50             T(index,1:s-1) = t;
51             %-----------------------------%
52         end
53         %-------------------------------------------------%
54         %NOTE: (s,s) is the position of the current
55         %pivot to be used in the elimination process
56         %(triangularization).
57
58         %Gauss elimination over the column s.
59         l = ones(m-s+1,1);
60         for h = s+1 : m
61             l(h-s+1) = U(h,s) / U(s,s);
62             U(h,s:m) = U(h,s:m) - l(h-s+1)*U(s,s:m);
63         end
64         T(s:m,s) = l;
65
66         %-----Recursive Call-----------------------------%
67         [per,p,T,U] = nnpsg2(U,m,per,T,p,s+1);
68         %-------------------------------------------------%
69
70         if(p == 1)
71             return;
72         else
73             if(p == 0)
74                 if(i == m)
75                     return;
76                 else
```

```
77                      U = Lf;
78                      T = L;
79                      per = b;
80                  end
81              end
82          end
83      else
84          if((i == m) && (abs(Lf(i,i) < eps)))
85              p = 0;
86              return;
87          end
88      end
89  end
90  end
```

File F.7: **nnpsg3.m**

```
1  %NNPSG1 - Non-nnull pivot selector,
2  %The following function is aimed to implement a step of the
3  %Gaussian elimination with diagonal pivoting. The pivot
4  %element is searched only along the diagonal. Consecutive steps
5  %are executed via recursive calls.
6
7  %Function nnpsg3 implements Tg3 as searching method for the
8  %pivot.
9
10 function [per,p,T,U] = nnpsg3(Lf,m,b,L,p,s)
11
12 U = Lf;
13 T = L;
14 per = b;
15
16 %-------------------------------------------------------------%
17 if(((m-s) == 0) && (abs(Lf(s,s)) > eps))
18     p = 1;
19     return;
20 end
21
22 if(((m-s) == 0) && (abs(Lf(s,s)) < eps))
23     p = 0;
24     return;
25 end
26 %-------------------------------------------------------------%
27
```

```
28 diag = zeros(m-s+1,1);
29 for j = s : m
30    diag(j-s+1) = abs(Lf(j,j));
31 end
32 [piv,index] = sort(diag,'descend');
33
34 %-----Pivot selection over the matrix diagonal.---%
35 for i = s : m
36     %-------------------------------------------------%
37     %Choose the pivot element with maximum modulus
38     %among the diagonal elements.
39     maxe = piv(i-s+1);
40     maxi = index(i-s+1) + (s-1);
41     %-------------------------------------------------%
42     if(abs(Lf(maxi,maxi)) > eps)
43         if(maxi ~= s)
44             %-----------------------------%
45             U = excn(U,s,maxi);
46             %-----------------------------%
47             r = per(s);
48             per(s) = per(maxi);
49             per(maxi) = r;
50             %-----------------------------%
51             t = T(s,1:s-1);
52             T(s,1:s-1) = T(maxi,1:s-1);
53             T(maxi,1:s-1) = t;
54             %-----------------------------%
55         end
56         %-----------------------------------------------%
57         %NOTE: (s,s) is the position of the current
58         %pivot to be used in the elimination process
59         %(triangularization).
60
61         %Gauss elimination over the column s.
62         l = ones(m-s+1,1);
63         for h = s+1 : m
64             l(h-s+1) = U(h,s) / U(s,s);
65             U(h,s:m) = U(h,s:m) - l(h-s+1)*U(s,s:m);
66         end
67         T(s:m,s) = l;
68
69         %-----Recursive Call-------------------------%
70         [per,p,T,U] = nnpsg3(U,m,per,T,p,s+1);
71         %-------------------------------------------%
72
```

```
73          if(p == 1)
74              return;
75          else
76              if(p == 0)
77                  if(i == m)
78                      return;
79                  else
80                      U = Lf;
81                      T = L;
82                      per = b;
83                  end
84              end
85          end
86      else
87          if((i == m) && (abs(Lf(maxi,maxi) < eps)))
88              p = 0;
89              return;
90          end
91      end
92  end
93  end
```

File F.8: **nnpsg4.m**

```
1  %NNPSG1 - Non-nnull pivot selector,
2  %The following function is aimed to implement a step of the
3  %Gaussian elimination with diagonal pivoting. The pivot
4  %element is searched only along the diagonal. Consecutive steps
5  %are executed via recursive calls.
6
7  %Function nnpsg4 implements Tg4 as searching method for the
8  %pivot.
9
10 function [per,p,T,U] = nnpsg4(Lf,m,b,L,p,s)
11
12 U = Lf;
13 T = L;
14 per = b;
15
16 %-------------------------------------------------------%
17 if(((m-s) == 0) && (abs(Lf(s,s)) > eps))
18     p = 1;
19     return;
20 end
```

```
21
22 if(((m-s) == 0) && (abs(Lf(s,s)) < eps))
23     p = 0;
24     return;
25 end
26 %-------------------------------------------------------%
27
28 %-----Pivot selection over the matrix diagonal.----%
29 for i = m : -1 : s
30     if(abs(Lf(i,i)) > eps)
31         if(i ~= s)
32             %-----------------------------%
33             U = excn(U,s,i);
34             %-----------------------------%
35             r = per(s);
36             per(s) = per(i);
37             per(i) = r;
38             %-----------------------------%
39             t = T(s,1:s-1);
40             T(s,1:s-1) = T(i,1:s-1);
41             T(i,1:s-1) = t;
42             %-----------------------------%
43         end
44         %-------------------------------------------------%
45         %NOTE: (s,s) is the position of the current
46         %pivot to be used in the elimination process
47         %(triangularization).
48
49         %Gauss elimination over the column s.
50         l = ones(m-s+1,1);
51         for h = s+1 : m
52             l(h-s+1) = U(h,s) / U(s,s);
53             U(h,s:m) = U(h,s:m) - l(h-s+1)*U(s,s:m);
54         end
55         T(s:m,s) = l;
56
57         %-----Recursive Call-----------------------------%
58         [per,p,T,U] = nnpsg4(U,m,per,T,p,s+1);
59         %-------------------------------------------------%
60
61         if(p == 1)
62             return;
63         else
64             if(p == 0)
65                 if(i == s)
```

```
66                      return;
67                  else
68                      U = Lf;
69                      T = L;
70                      per = b;
71                  end
72              end
73          end
74      else
75          if((i == s) && (abs(Lf(i,i) < eps)))
76              p = 0;
77              return;
78          end
79      end
80  end
81  end
```

File F.9: **nnpsg5.m**

```
1  %NNPSG1 - Non-nnull pivot selector,
2  %The following function is aimed to implement a step of the
3  %Gaussian elimination with diagonal pivoting. The pivot
4  %element is searched only along the diagonal. Consecutive steps
5  %are executed via recursive calls.
6
7  %Function nnpsg5 implements Tg5 as searching method for the
8  %pivot.
9
10 function [per,p,T,U] = nnpsg5(Lf,m,b,L,p,s)
11
12 U = Lf;
13 T = L;
14 per = b;
15
16 %-------------------------------------------------------%
17 if(((m-s) == 0) && (abs(Lf(s,s)) > eps))
18     p = 1;
19     return;
20 end
21
22 if(((m-s) == 0) && (abs(Lf(s,s)) < eps))
23     p = 0;
24     return;
25 end
```

```
26 %-------------------------------------------------------%
27
28 choice = randi([1 3],1,1);
29 switch choice
30     case 1
31         diag = zeros(m-s+1,1);
32         for j = s : m
33             diag(j-s+1) = abs(Lf(j,j));
34         end
35         [piv,sequence] = sort(diag,'descend');
36     case 2
37         sequence = irp(m-s+1);
38     case 3
39         sequence = zeros(m-s+1,1);
40         for h = 1 : m-s+1
41             sequence(h) = h;
42         end
43 end
44
45 %-----Pivot selection over the matrix diagonal.------%
46 for i = s : m
47
48     index = sequence(i-s+1) + (s-1);
49
50     if(abs(Lf(index,index)) > eps)
51         if(index ~= s)
52             %-----------------------------%
53             U = excn(U,s,index);
54             %-----------------------------%
55             r = per(s);
56             per(s) = per(index);
57             per(index) = r;
58             %-----------------------------%
59             t = T(s,1:s-1);
60             T(s,1:s-1) = T(index,1:s-1);
61             T(index,1:s-1) = t;
62             %-----------------------------%
63         end
64         %-------------------------------------------------%
65         %NOTE: (s,s) is the position of the current pivot
66         %to be used in the elimination process
67         %(triangularization).
68
69         %Gauss elimination over the column s.
70         l = ones(m-s+1,1);
```

```
71          for h = s+1 : m
72              l(h-s+1) = U(h,s) / U(s,s);
73              U(h,s:m) = U(h,s:m) - l(h-s+1)*U(s,s:m);
74          end
75          T(s:m,s) = l;
76
77          %-----Recursive Call--------------------------------%
78          [per,p,T,U] = nnpsg5(U,m,per,T,p,s+1);
79          %---------------------------------------------------%
80
81          if(p == 1)
82              return;
83          else
84              if(p == 0)
85                  if(i == m)
86                      return;
87                  else
88                      U = Lf;
89                      T = L;
90                      per = b;
91                  end
92              end
93          end
94      else
95          if((i == m) && (abs(Lf(i,i) < eps)))
96              p = 0;
97              return;
98          end
99      end
100 end
101 end
```

Appendix G

Matlab: Code from Chapter 5

In chapter 5 the problem of finding a stabilizing matrix for the stabilization of the follower-follower Laplacian matrix L_{ff} has been addressed. More precisely, the problem was to find a diagonal matrix M such that, the matrix $-DL$ has all stable eigenvalues. D is a slightly different diagonal matrix, depending on the agent's dynamic we are considering. The two expressions are

Single-Integrator Kinematics

$$D = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & M \end{bmatrix}, \tag{G.1}$$

Double-Integrator Dynamics

$$D = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & \epsilon M \end{bmatrix}, \tag{G.2}$$

where $\epsilon > 0$ is a constant to be found.

In the following sections implementations of the proposed algorithms for the MIEP are shown.

G.1 Ballantine's Theorem.Algorithm 1

In this section are proposed some Matlab implementations of the algorithm 5.1 proposed in chapter 5. All listed files use a Matlab function in order to compute, at each step i, the eigenvalues of the matrix $M_{[1\sim i]}L_{ff[1\sim i]}$. The difference is in the search methods for the diagonal elements in the set $\mathcal{W}$. In fact, function

smse1 (and smse12) implement search method of figure (5.3), that is, elements d_i are chosen from segments parallel to the imaginary axis.

smse2 implement search method of figure (5.4), that is, elements d_i are chosen from segments with variable direction;

smse3 implement search method of figure (5.5), that is, elements d_i are chosen from circles with variable radius.

File G.1: **smse1.m**

```
%SMSE1 - Stabilizing Matrix Solver.
%-----Ballantine & Eigenvalue Computation-----%
%Elements di are chosen from straight lines
%parallel to the imaginary axis and eigenvalues
%are computed with the Matlab function.

function D = smse1(A)

[m,n] = size(A);

eigen = eig(A);

for i = 1 : m
    if(real(eigen(i)) < 0)
        break;
    end
    if(i == m)
        disp('No need of stabilisation');
        D = eye(m);
        return;
    end
end

diagonal = zeros(m,1);
%diagonal(1) = m*conj(A(1,1));
diagonal(1) = 1/A(1,1);

for k = 2 : m
    %for a = 0 : 0.1 : m*18
    %for a = -10 : 0.1 : 10
    for a = 0 : 0.1 : 10
        [a k]
        %for b = -18*k : 0.1 : 18*k
```

```
34          for b = -10 : 0.1 : 10
35                  diagonal(k) = complex(a,b);
36                  M = diag(diagonal(1:k))*A(1:k,1:k);
37                  eigen = eig(M);
38                  [mine,mini] = min(real(eigen));
39                  if(mine < 0)
40                     diagonal(k) = 0;
41                  else
42          if((a≠0 || b≠0) && (diagonal(k) ≠ diagonal(k-1)))
43                                  break;
44                      end
45                  end
46          end
47          if(mine < 0)
48              diagonal(k) = 0;
49          else
50              break;
51          end
52      end
53      if(diagonal(k) == 0)
54         disp('No D matrix found');
55         break;
56      end
57  end
58
59  D = diag(diagonal);
60
61  end
```

File G.2: **smse12.m**

```
1   %SMSE12 - Stabilizing Matrix Solver.
2   %-----Ballantine & Eigenvalue Computation-----%
3   %Elements di are chosen from straight lines
4   %parallel to the imaginary axis and eigenvalues
5   %are computed with the Matlab function.
6
7   function D = smse12(A)
8
9   [m,n] = size(A);
10
11  eigen = eig(A);
12
13  for i = 1 : m
```

```
    if(real(eigen(i)) < 0)
        break;
    end
    if(i == m)
        disp('No need of stabilisation');
        D = eye(m);
        return;
    end
end

diagonal = zeros(m,1);
diagonal(1) = 15*m / A(1,1);

for k = 2 : m
    for a = 0 : 0.1 : m*18
        [a k]
        for b = -18*k : 0.1 : 18*k
                diagonal(k) = complex(a,b);
                M = diag(diagonal(1:k)) * A(1:k,1:k);
                eigen = eig(M);
                [mine,mini] = min(real(eigen));
                if(mine > 0)
                    if(a~=0 || b~=0)
                        break;
                    end
                end
        end
        if(mine > 0)
            break;
        end
    end
end

D = diag(diagonal);

end
```

File G.3: **smse2.m**

```
%SMSE2 - Stabilizing Matrix Solver.
%-----Ballantine & Eigenvalue Computation-----%
%Elements di are chosen from straight lines
%with different directions and eigenvalues
%are computed with the Matlab function.
```

```
6
7  function D = smse2(A)
8
9  [m,n] = size(A);
10
11 eigen = eig(A);
12
13 for i = 1 : m
14     if(real(eigen(i)) < 0)
15         break;
16     end
17     if(i == m)
18         disp('No need of stabilisation');
19         D = eye(m);
20         return;
21     end
22 end
23
24 diagonal = zeros(m,1);
25 % diagonal(1) = m*15*conj(A(1,1));
26 %diagonal(1) = m*15 / A(1,1);
27 diagonal(1) = 1/A(1,1);
28
29 for k = 2 : m
30
31 %var = pi;
32 var = 2*pi;
33
34     for angle = 0.1*k : 0.05 : var + 0.1*k
35     %for angle = 0.5 : 0.1 : var + 0.5
36         %for a = -k*20 : 0.2 : k*20
37         %for a = -10 : 0.1 : 10
38         for a = 0 : 0.1 : 10
39             [angle  a  k]
40             b = a*tan(angle);
41             diagonal(k) = complex(a,b);
42             M = diag(diagonal(1:k)) * A(1:k,1:k);
43             eigen = eig(M);
44             [mine,mini] = min(real(eigen));
45             if(mine < 0)
46                 diagonal(k) = 0;
47             else
48         if((a~=0 || b~=0) && (diagonal(k) ~= diagonal(k-1)))
49                     break;
50                 end
```

```
51              end
52          end
53          if(mine < 0)
54              diagonal(k) = 0;
55          else
56              break;
57          end
58      end
59      if(diagonal(k) == 0)
60         disp('No D matrix found');
61         break;
62      end
63  end
64
65  D = diag(diagonal);
66
67  end
```

File G.4: **smse3.m**

```
1   %SMSE3 – Stabilizing Matrix Solver.
2   %-----Ballantine & Eigenvalue Computation-----%
3   %Elements di are chosen from circles
4   %with different radius and eigenvalues
5   %are computed with the Matlab function.
6
7   function D = smse3(A,par)
8
9   [m,n] = size(A);
10
11  eigen = eig(A);
12
13  for i = 1 : m
14      if(real(eigen(i)) < 0)
15          break;
16      end
17      if(i == m)
18          disp('No need of stabilisation');
19          D = eye(m);
20          return;
21      end
22  end
23
24  diagonal = zeros(m,1);
```

```
% diagonal(1) = m*15*conj(A(1,1));
%diagonal(1) = m*15 / A(1,1);
diagonal(1) = 1/A(1,1);

for k = 2 : m

var = 2*pi;

    %for r = 0.1 : 0.2 : k*30
    %for r = 5 : 0.2 : k*30
    %for r = 2 : 0.2 : k*30
    for r = 0.2+par : 0.2 : 10+par
        for angle = 0.1*k : -0.05 : var + 0.1*k
        %for angle = -0.1*k : -0.05 : -var - 0.1*k
            [angle  r  k]
            a = (r^2 / (1 + tan(angle)^2))^0.5;
            b = a*tan(angle);
            diagonal(k) = complex(a,b);
            M = diag(diagonal(1:k)) * A(1:k,1:k);
            eigen = eig(M);
            [mine,mini] = min(real(eigen));
            if(mine < 0)
                diagonal(k) = 0;
            else
        if((a≠0 || b≠0) && (diagonal(k) ≠ diagonal(k-1)))
                    break;
                end
            end
        end
        if(mine < 0)
            diagonal(k) = 0;
        else
            break;
        end
    end
    if(diagonal(k) == 0)
       disp('No D matrix found');
       break;
    end
end

D = diag(diagonal);

end
```

G.2 Bounding the Eigenvalues. Algorithm 2

File G.5: **smsb.m**

```
%SMSB - Stabilizing Matrix Solver.
%-----Ballantine & Eigenvalue Localisation-----%
%Rojo - Rectangular localisation.

function D = smsb(L)

[m n] = size(L);
eigen = eig(L);

if(min(real(eigen)) > 0)
    disp('No need of stabilisation');
    D = eye(m);
    return;
else
    disp('Computing the stabilising matrix D');
end

diagonal = zeros(m,1);
% diagonal(1) = m*15*conj(L(1,1));
diagonal(1) = 1 / L(1,1);

for k = 2 : m
    min_dki = -80 + (k^2*20);
    max_dki =  80 + (k^2*20);
    %-------------------------------------------------------------%
    A1 = [diag(diagonal(1:k-1))*L(1:k-1,1:k) ; zeros(1,k)];
    trA1 = trace(A1);
    %-------------------------------------------------------------%
    for dki = min_dki : 0.5 : max_dki
        lkkre = real(L(k,k));
        yk_min = dki*imag(L(k,k)/abs(lkkre) - ...
            real(trA1)/lkkre);
        min_yk = yk_min + (k^2*20);
        max_yk = min_yk + (k*20);
        for yk = min_yk : 0.5 : max_yk
            dk = sign(lkkre)*yk + dki*1i;
            trA = trA1 + dk*L(k,k);
            modtrA = abs(trA)^2;
            trAre = real(trA);
            A = [A1(1:k-1,1:k) ; dk*L(k,1:k)];
```

```
            K3A = ((norm(A)^2 - modtrA/k)^2 - ...
                norm(A*A'-A'*A)^2/2)^0.5;
 f3 = (((k-1)/k)^0.5)*((K3A + real(trace(A*A)))/2 - ...
     trAre^2/k)^0.5;
            ckkres = (trAre/k);
            [ckkres    f3    dki    yk_min  k]
            response = ckkres - f3 > 0;
            if(response && (dki ~= 0 || real(dk) ~= 0))
                diagonal(k) = dk;
                break;
            end
        end
        if(response)
            diagonal(k) = dk;
            break;
        end
    end
end

D = diag(diagonal);

end
```

Appendix H

Matlab: Code from Chapter 6

In chapter 6 experimental results were shown for the complex Laplacian formation control of a group of agents. The experiments were implemented in Matlab code and in this chapter the files are shown.

Note that in files H.2 and H.4 the convolution integrals

SIK case

$$\int_{\tau}^{t} \mathrm{e}^{-DL(t-\alpha)}\, bv_0(\alpha)\, \mathrm{d}\alpha,$$

DID case

$$\int_{\tau}^{t} \mathrm{e}^{H(t-\alpha)}\, ba_0(\alpha)\, \mathrm{d}\alpha,$$

have been made discrete and implemented as a sum in the following way

SIK case

$$\sum_{\alpha=\tau}^{t} \mathrm{e}^{-DL(t-\alpha)}\, bv_0(\alpha),$$

DID case

$$\sum_{\alpha=\tau}^{t} \mathrm{e}^{H(t-\alpha)}\, ba_0(\alpha).$$

In both experiments τ has been chosen to be zero.

File H.1: **sikplanarformation.m**

```
1 %Symulation of a group of agents
2 %reaching a planar formation.
3
```

```
4  %Time interval
5  step = 0.2;
6  time = 50;
7  max = (time / step) + 2;
8
9  %Random group of 5 agents
10 %scale = 1;
11 %z = scale*(rand(5,1) + rand(5,1)*1i);
12 z = randi([-5 5],5,1) + randi([-5 5],5,1)*1i;
13
14 %Stabilizing diagonal matrix D
15 M_s = G;
16 D = [1 0 0 0 0; 0 1 0 0 0; 0 0 M_s(1,:); 0 0 M_s(2,:); 0 0 ...
       M_s(3,:)];
17
18
19 %Defining agents' positions vector
20 agent1 = zeros(max,1);
21 agent2 = zeros(max,1);
22 agent3 = zeros(max,1);
23 agent4 = zeros(max,1);
24 agent5 = zeros(max,1);
25
26 agent1(1) = z(1);
27 agent2(1) = z(2);
28 agent3(1) = z(3);
29 agent4(1) = z(4);
30 agent5(1) = z(5);
31
32 %Transition matrix H
33 Lap = L;
34 H = -D*Lap;
35
36 index = 0;
37 for t = 0 : step : time
38
39     %Unforced response
40     w = expm(H*t)*z;
41
42     agent1(index+2) = w(1);
43     agent2(index+2) = w(2);
44     agent3(index+2) = w(3);
45     agent4(index+2) = w(4);
46     agent5(index+2) = w(5);
47
```

```
48      index = index + 1;
49  end
50
51  %Plotting agents' initial position
52  plot(z(1),'b*');
53  hold on
54  plot(z(2),'r*');
55  hold on
56  plot(z(3),'g*');
57  hold on
58  plot(z(4),'k*');
59  hold on
60  plot(z(5),'m*');
61  hold on
62  %Plotting agents' positions evolution
63  plot(real(agent1),imag(agent1),'b-');
64  hold on
65  plot(real(agent2),imag(agent2),'r-');
66  hold on
67  plot(real(agent3),imag(agent3),'g-');
68  hold on
69  plot(real(agent4),imag(agent4),'k-');
70  hold on
71  plot(real(agent5),imag(agent5),'m-');
72  hold on
73  %Plotting agents' symbols in the last position
74  plot(real(agent1(max)),imag(agent1(max)),'b^');
75  hold on
76  plot(real(agent2(max)),imag(agent2(max)),'r^');
77  hold on
78  plot(real(agent3(max)),imag(agent3(max)),'go');
79  hold on
80  plot(real(agent4(max)),imag(agent4(max)),'ko');
81  hold on
82  plot(real(agent5(max)),imag(agent5(max)),'mo');
83  hold on
84
85  xlabel('Real axis');
86  ylabel('Imaginary axis');
87  legend('v1 (l)','v2 (l)','v3 (f)','v4 (f)','v5 (f)',4);
88
89  %Plotting agents formation shape
90  arc1 = zeros(2,1);
91  arc2 = zeros(2,1);
92  arc3 = zeros(2,1);
```

```
93  arc4 = zeros(2,1);
94  arc5 = zeros(2,1);
95
96  arc1(1) = agent1(max);
97  arc1(2) = agent5(max);
98
99  arc2(1) = agent1(max);
100 arc2(2) = agent2(max);
101
102 arc3(1) = agent2(max);
103 arc3(2) = agent3(max);
104
105 arc4(1) = agent3(max);
106 arc4(2) = agent4(max);
107
108 arc5(1) = agent4(max);
109 arc5(2) = agent5(max);
110
111 %Plotting the shape formation
112 hold on
113 plot(real(arc1),imag(arc1),'k--');
114 hold on
115 plot(real(arc2),imag(arc2),'k--');
116 hold on
117 plot(real(arc3),imag(arc3),'k--');
118 hold on
119 plot(real(arc4),imag(arc4),'k--');
120 hold on
121 plot(real(arc5),imag(arc5),'k--');
122 hold on
123
124 %Disabling stretch to fill option
125 % h_axes = gca;
126 % set(h_axes,'PlotBoxAspectRatio',[1 1 1]);
127
128 axis equal;
129
130 %Changing scale axis
131 axis([-15 15 -15 10]);
```

File H.2: **sikmovingplanarformation.m**

```
1  %Symulation of a group of agents
2  %reaching a planar formation while moving.
```

```
%SIK case.

%Time interval
step = 0.4;
step_conv = 0.2;
time = 12;
max = (time / step) + 2;
% time = 25;
% max = time + 2;

%Random group of 5 agents
%scale = 1;
%z = scale*(rand(5,1) + rand(5,1)*1i);
z = randi([-5 5],5,1) + randi([-5 5],5,1)*1i;

%Stabilizing diagonal matrix D
M_s = G;
D = [1 0 0 0 0; 0 1 0 0 0; 0 0 M_s(1,:); 0 0 M_s(2,:); 0 0 ...
    M_s(3,:)];

%Defining agents' positions vector
agent1 = zeros(max,1);
agent2 = zeros(max,1);
agent3 = zeros(max,1);
agent4 = zeros(max,1);
agent5 = zeros(max,1);

agent1(1) = z(1);
agent2(1) = z(2);
agent3(1) = z(3);
agent4(1) = z(4);
agent5(1) = z(5);

%Transition matrix H
Lap = L;
H = -D*Lap;

b = ones(5,1);
index = 0;
for t = 0 : step : time

    %Forced responce (convolution integral)
    sum = 0;
    for tao = 0 : step_conv : t
        input = b*(2*tao*cos(0.1*tao) + ...
```

```
                0.5*tao*sin(0.1*tao)*1i);
47          sum = sum + expm(H*(t - tao))*input;
48      end
49
50      %Total response of the forced system
51      w = expm(H*t)*z + sum;
52
53      agent1(index+2) = w(1);
54      agent2(index+2) = w(2);
55      agent3(index+2) = w(3);
56      agent4(index+2) = w(4);
57      agent5(index+2) = w(5);
58
59      index = index + 1;
60  end
61
62  %Plotting agents' initial position
63  plot(z(1),'b^');
64  hold on
65  plot(z(2),'r^');
66  hold on
67  plot(z(3),'g*');
68  hold on
69  plot(z(4),'k*');
70  hold on
71  plot(z(5),'m*');
72  hold on
73  %Plotting agents' positions evolution
74  plot(real(agent1),imag(agent1),'b-');
75  hold on
76  plot(real(agent2),imag(agent2),'r-');
77  hold on
78  plot(real(agent3),imag(agent3),'g-');
79  hold on
80  plot(real(agent4),imag(agent4),'k-');
81  hold on
82  plot(real(agent5),imag(agent5),'m-');
83  hold on
84  %Plotting agents' symbols in the last position
85  plot(real(agent1(max)),imag(agent1(max)),'b^');
86  hold on
87  plot(real(agent2(max)),imag(agent2(max)),'r^');
88  hold on
89  plot(real(agent3(max)),imag(agent3(max)),'go');
90  hold on
```

```
91  plot(real(agent4(max)),imag(agent4(max)),'ko');
92  hold on
93  plot(real(agent5(max)),imag(agent5(max)),'mo');
94  hold on
95
96  xlabel('Real axis');
97  ylabel('Imaginary axis');
98  legend('v1 (l)','v2 (l)','v3 (f)','v4 (f)','v5 (f)',4);
99
100 %Plotting agents formation shape
101 arc1 = zeros(2,1);
102 arc2 = zeros(2,1);
103 arc3 = zeros(2,1);
104 arc4 = zeros(2,1);
105 arc5 = zeros(2,1);
106
107 arc1(1) = agent1(max);
108 arc1(2) = agent5(max);
109
110 arc2(1) = agent1(max);
111 arc2(2) = agent2(max);
112
113 arc3(1) = agent2(max);
114 arc3(2) = agent3(max);
115
116 arc4(1) = agent3(max);
117 arc4(2) = agent4(max);
118
119 arc5(1) = agent4(max);
120 arc5(2) = agent5(max);
121
122 %Plotting the shape formation
123 hold on
124 plot(real(arc1),imag(arc1),'k--');
125 hold on
126 plot(real(arc2),imag(arc2),'k--');
127 hold on
128 plot(real(arc3),imag(arc3),'k--');
129 hold on
130 plot(real(arc4),imag(arc4),'k--');
131 hold on
132 plot(real(arc5),imag(arc5),'k--');
133 hold on
134
135 axis equal;
```

```
136
137 %Changing scale axis
138 axis([-20 500 -100 260]);
139
140 %NOTE: Be careful defining step and step_conv.
```

File H.3: **didplanarformation.m**

```
1  %Symulation of a group of agents
2  %reaching a planar formation.
3  %DID case.
4
5  %Time interval
6  step = 0.5;
7  time = 320;
8  max = (time / step) + 2;
9  % time = 360;
10 % max = time + 2;
11
12 %Random group of 5 agents
13 %scale = 1;
14 %z = scale*(rand(5,1) + rand(5,1)*1i);
15 z = randi([-5 5],5,1) + randi([-5 5],5,1)*1i;
16
17 %Stabilizing diagonal matrix D
18 M_s = M1;
19 D = [1 0 0 0 0; 0 1 0 0 0; 0 0 M_s(1,:); 0 0 M_s(2,:); 0 0 ...
       M_s(3,:)];
20
21 %Transition matrix H
22 gamma = 5;
23 E = eye(5);
24 N = zeros(5);
25 Lap = L;
26 H = [N E ; -D*Lap  -gamma*E];
27
28 %Defining agents' positions vector
29 agent1 = zeros(max,1);
30 agent2 = zeros(max,1);
31 agent3 = zeros(max,1);
32 agent4 = zeros(max,1);
33 agent5 = zeros(max,1);
34
35 agent1(1) = z(1);
```

```
agent2(1) = z(2);
agent3(1) = z(3);
agent4(1) = z(4);
agent5(1) = z(5);

%v = ones(5,1);
v = zeros(5,1);
st = [z ; v];

index = 0;
for t = 0 : step : time

    %Unforced response
    w = expm(H*t)*st;

    agent1(index+2) = w(1);
    agent2(index+2) = w(2);
    agent3(index+2) = w(3);
    agent4(index+2) = w(4);
    agent5(index+2) = w(5);

    index = index + 1;
end

%Plotting agents' initial position
plot(z(1),'b*');
hold on
plot(z(2),'r*');
hold on
plot(z(3),'g*');
hold on
plot(z(4),'k*');
hold on
plot(z(5),'m*');
hold on
%Plotting agents' positions evolution
plot(real(agent1),imag(agent1),'b-');
hold on
plot(real(agent2),imag(agent2),'r-');
hold on
plot(real(agent3),imag(agent3),'g-');
hold on
plot(real(agent4),imag(agent4),'k-');
hold on
plot(real(agent5),imag(agent5),'m-');
```

```
hold on
%Plotting agents' symbols in the last position
plot(real(agent1(max)),imag(agent1(max)),'bo');
hold on
plot(real(agent2(max)),imag(agent2(max)),'ro');
hold on
plot(real(agent3(max)),imag(agent3(max)),'go');
hold on
plot(real(agent4(max)),imag(agent4(max)),'ko');
hold on
plot(real(agent5(max)),imag(agent5(max)),'mo');
hold on

xlabel('Real axis');
ylabel('Imaginary axis');
legend('v1 (l)','v2 (l)','v3 (f)','v4 (f)','v5 (f)',4);

%Plotting agents formation shape
arc1 = zeros(2,1);
arc2 = zeros(2,1);
arc3 = zeros(2,1);
arc4 = zeros(2,1);
arc5 = zeros(2,1);

arc1(1) = agent1(max);
arc1(2) = agent5(max);

arc2(1) = agent1(max);
arc2(2) = agent2(max);

arc3(1) = agent2(max);
arc3(2) = agent3(max);

arc4(1) = agent3(max);
arc4(2) = agent4(max);

arc5(1) = agent4(max);
arc5(2) = agent5(max);

%Plotting the shape formation
hold on
plot(real(arc1),imag(arc1),'k--');
hold on
plot(real(arc2),imag(arc2),'k--');
hold on
```

```
126 plot(real(arc3),imag(arc3),'k--');
127 hold on
128 plot(real(arc4),imag(arc4),'k--');
129 hold on
130 plot(real(arc5),imag(arc5),'k--');
131 hold on
132
133 axis equal;
134
135 %Changing scale axis
136 axis([-18 8  -8 18]);
```

File H.4: **didmovingplanarformation.m**

```
1  %Symulation of a group of agents
2  %reaching a planar formation while moving.
3  %DID case.
4
5  %Time interval
6  step = 0.4;
7  step_conv = 0.2;
8  time = 60;
9  max = (time / step) + 2;
10 % time = 400;
11 % max = time + 2;
12
13 %Random group of 5 agents
14 %scale = 1;
15 %z = scale*(rand(5,1) + rand(5,1)*1i);
16 z = randi([-5 5],5,1) + randi([-5 5],5,1)*1i;
17
18 %Stabilizing diagonal matrix D
19 M_s = G;
20 D = [1 0 0 0 0; 0 1 0 0 0; 0 0 M_s(1,:); 0 0 M_s(2,:); 0 0 ...
       M_s(3,:)];
21
22 %Transition matrix H
23 gamma = 20;
24 E = eye(5);
25 N = zeros(5);
26 Lap = L;
27 H = [N E ; -D*Lap -gamma*E];
28
29 %Defining agents' positions vector
```

```
agent1 = zeros(max,1);
agent2 = zeros(max,1);
agent3 = zeros(max,1);
agent4 = zeros(max,1);
agent5 = zeros(max,1);

agent1(1) = z(1);
agent2(1) = z(2);
agent3(1) = z(3);
agent4(1) = z(4);
agent5(1) = z(5);

v = ones(5,1);
%v = zeros(5,1);
st = [z ; v];
b = ones(10,1);

index = 0;
for t = 0 : step : time
    t
    %Forced responce (convolution integral)
    sum = 0;
    for tao = 0 : step_conv : t
        input = b*(2*tao*cos(0.1*t) + 1.5*t*sin(0.1*t)*1i);
        sum = sum + expm(H*(t - tao))*input;
    end

    %Total response of the forced system
    w = expm(H*t)*st + sum;

    agent1(index+2) = w(1);
    agent2(index+2) = w(2);
    agent3(index+2) = w(3);
    agent4(index+2) = w(4);
    agent5(index+2) = w(5);

    index = index + 1;
end

%Plotting agents' initial position
plot(z(1),'b^');
hold on
plot(z(2),'r^');
hold on
plot(z(3),'g*');
```

```
75 hold on
76 plot(z(4),'k*');
77 hold on
78 plot(z(5),'m*');
79 hold on
80 %Plotting agents' positions evolution
81 plot(real(agent1),imag(agent1),'b-');
82 hold on
83 plot(real(agent2),imag(agent2),'r-');
84 hold on
85 plot(real(agent3),imag(agent3),'g-');
86 hold on
87 plot(real(agent4),imag(agent4),'k-');
88 hold on
89 plot(real(agent5),imag(agent5),'m-');
90 hold on
91 %Plotting agents' symbols in the last position
92 plot(real(agent1(max)),imag(agent1(max)),'b^');
93 hold on
94 plot(real(agent2(max)),imag(agent2(max)),'r^');
95 hold on
96 plot(real(agent3(max)),imag(agent3(max)),'go');
97 hold on
98 plot(real(agent4(max)),imag(agent4(max)),'ko');
99 hold on
100 plot(real(agent5(max)),imag(agent5(max)),'mo');
101 hold on
102
103 xlabel('Real axis');
104 ylabel('Imaginary axis');
105 legend('v1 (l)','v2 (l)','v3 (f)','v4 (f)','v5 (f)',4);
106
107 %Plotting agents formation shape
108 arc1 = zeros(2,1);
109 arc2 = zeros(2,1);
110 arc3 = zeros(2,1);
111 arc4 = zeros(2,1);
112 arc5 = zeros(2,1);
113
114 arc1(1) = agent1(max);
115 arc1(2) = agent5(max);
116
117 arc2(1) = agent1(max);
118 arc2(2) = agent2(max);
119
```

```
120 arc3(1) = agent2(max);
121 arc3(2) = agent3(max);
122
123 arc4(1) = agent3(max);
124 arc4(2) = agent4(max);
125
126 arc5(1) = agent4(max);
127 arc5(2) = agent5(max);
128
129 %Plotting the shape formation
130 hold on
131 plot(real(arc1),imag(arc1),'k--');
132 hold on
133 plot(real(arc2),imag(arc2),'k--');
134 hold on
135 plot(real(arc3),imag(arc3),'k--');
136 hold on
137 plot(real(arc4),imag(arc4),'k--');
138 hold on
139 plot(real(arc5),imag(arc5),'k--');
140 hold on
141
142 axis equal;
143
144 %Changing scale axis
145 %axis([-900 1000 -100 1800]);
146
147 %NOTE: Be careful defining step and step_conv.
```

Appendix I

Matlab: Additional Functions

In chapter 4 we have seen how to solve the problem of finding a permutation matrix P such that the matrix $\hat{L_{ff}}$ resulting from the expression $\hat{L_{ff}} = PL_{ff}P^T$ has non-null leading principal minors. Moreover, in appendix F Matlab implementations of the algorithms have been given. In this chapter some Matlab functions used in appendix F are given.

dpm1.m Function *dpm1(A)* returns a boolean variable which is one if matrix A has non-null leading principal minors and zero otherwise. It returns a vector too that contains the value of all leading principal minors, including the determinant of the entire matrix in the last position. File I.1 shows the Matlab implementation.

File I.1: **dpm1.m**

```
1  %DPM1 - Determinant of the leading
2  %principal minors of a matrix.
3
4  %Function dpm1 computes the determinants
5  %of all leading principal minors of
6  %a square matrix passed as an argument.
7  %The output function is a vector whose
8  %elements are the aforementioned determinants.
9  %The first element is the determinant of the
10 %smallest minor (that is the matrix element A(1,1)),
11 %the second the determinant of the minor of order 2
12 %and so on. The last one is the determinant of the
13 %matrix itself. The algorithm has been designed
14 %as an iterative one and the det() Matlab function
15 %has been used.
```

```
16
17 function [nn,dmp1,d1] = dpm1(A)
18
19 [m,n] = size(A);
20
21 %d1 = 0 if the matrix is not a square one.
22 if(m ≠ n)
23     d1 = 0;
24 else
25     d1 = 1;
26 end
27
28 dmp1 = zeros(m,1);
29
30 nn = 1;
31 for i = 1 : m
32     dmp1(i) = det(A(1:i,1:i));
33     if(abs(dmp1(i)) < eps)
34         nn = 0;
35     end
36 end
37
38 end
```

excn.m Function *excn(A,h,k)* simply exchanges row and column h with row and column k in the matrix A passed as an argument. File I.2 shows the Matlab implementation.

File I.2: **excn.m**

```
1  %EXCN - EXCHANGE
2  %The function excn exchanges row and column h
3  %with row and column k in the
4  %matrix A passed as an argument.
5  %NOTE: h < k.
6
7  function [A,w] = excn(A,h,k)
8
9  if(h > k)
10     w = 0;
11     return;
12 else
13     [m,n] = size(A);
```

```
14      if((h > m) || (k > n))
15          w = 0;
16          return;
17      end
18      rh = A(h,:);
19      A(h,:) = A(k,:);
20      A(k,:) = rh;
21      ch = A(:,h);
22      A(:,h) = A(:,k);
23      A(:,k) = ch;
24      w = 1;
25  end
26
27  end
```

dtr.m Function *dtr(A)* simply computes the determinant of the matrix A passed as an argument. File I.3 shows the Matlab implementation based on Gauss elimination with partial pivoting.

File I.3: **dtr.m**

```
1  %DTR -Determinant.
2  %The function dtr computes the determinant of a square
3  %matrix using the Gauss elimination with partial
4  %pivoting.Both real or complex matrices can be passed
5  %to the function. The output variables are:
6  % d   = value of the determinant,
7  % L,U = triangular factors of input matrix A,
8  % P   = permutation matrix for partial pivoting,
9  % w   = boolean variable for success / insuccess.
10
11 function [d,L,A,P,w] = dtr(A)
12
13 [m,n] = size(A);
14
15 if(m ~= n)
16      disp('The matrix is not a square one');
17      d = 0;
18      w = 0;
19      return;
20 end
21
22 L = eye(m);
```

```
perm = 0;
b = zeros(m,1);
for i = 1 : m
    b(i) = i;
end

for k = 1 : m-1
    %-------------------------------------------------------%
    %Choose the pivot element with maximum modulus.

    [maxe,index] = max(abs(A(k:m,k)));
    maxi = index + k -1;

    if(maxe < eps)
        break;
    end
    %-------------------------------------------------------%

    %-------------------------------------------------------%
    %Column pivoting.
    if(k ~= maxi)
        %For the vector used to build P.
        p = b(k);
        b(k) = b(maxi);
        b(maxi) = p;
        %For the sub-matrix A[k~m].
        r = A(k,:);
        A(k,:) = A(maxi,:);
        A(maxi,:) = r;
        %Pivoting for matrix L.
        t = L(k,1:k-1);
        L(k,1:k-1) = L(maxi,1:k-1);
        L(maxi,1:k-1) = t;
        %Counting the number of permutations.
        perm = perm +1;
    end
    %-------------------------------------------------------%

    %-------------------------------------------------------%
    %Gauss elimination step.
    l = ones(m-k+1,1);
    for j = k+1 : m
        l(j-k+1) = A(j,k) / A(k,k);
        A(j,k:m) = A(j,k:m) - l(j-k+1)*A(k,k:m);
    end
```

```
68      L(k:m,k) = l;
69      %-------------------------------------------------------%
70  end
71  %-----------------------------------------------------------%
72  %Determinant.
73  d = (-1)^perm;
74  w = 1;
75  for j = 1 : m
76      d = d*A(j,j);
77  end
78  %-----------------------------------------------------------%
79
80  %-----------------------------------------------------------%
81  %Permutation Matrix.
82  I = eye(m);
83  P = zeros(m);
84  for h = 1 : m
85      P(h,:) = I(b(h),:);
86  end
87  %-----------------------------------------------------------%
88  end
```

irp.m Function *irp(m)* generates a vector of m integers from 1 to m. The sequence of the vector elements is chosen randomly. File I.4 shows the Matlab implementation.

File I.4: **irp.m**

```
1  %IRP - Integer Random Permutation.
2  %The function irp generates a vector of m
3  %integers from 1 to m.The sequence
4  %of the elements of the vector is chosen randomly.
5
6  function index = irp(m,n)
7
8  if(nargin < 2)
9      n = 1;
10 end
11
12 index = zeros(m,1);
13 index(1) = randi([n m],1,1);
14
15 for i = 2 : m
```

```
16      ok = 0;
17      while(ok == 0)
18          y = randi([n m],1,1);
19          if(¬search(index(1:i−1),y))
20              index(i) = y;
21              ok = 1;
22          else
23              ok = 0;
24          end
25      end
26  end
27
28  end
```

Bibliography

[1] *TEX Live 2011*, June 2011. www.tug.org/texlive/doc.

[2] F. Alkhateeb, E. Al Maghayreh, and I. A. Doush, editors. *Multi-agent Systems - Modeling, Interactions, Simulations and Case Sstudies*. InTech, Croatia, 2011.

[3] $\mathcal{AMS}$, American Mathematical Society. *User's Guide for the* ***amsmath*** *Package*, Feb 2002.

[4] K. Balasubramanian. Computational Techniques for the Automorphism Groups of Graphs. *J. Chem. Inf. Comput. Sci.*, (34):621–626, 1994.

[5] C. S. Ballantine. Stabilization by a Diagonal Matrix. *Proceedings of the American Mathematical Society*, 25(4):728–734, 1970.

[6] J. Bang-Jensen and G. Gutin. *Digraphs Theory, Algorithms and Applications*. Springer-Verlag, 2007.

[7] R. B. Bapat, D. Kalita, and S. Pati. On weighted directed graphs. May 2011.

[8] D. Bini, M. Capovani, and O. Menchi. *Metodi Numerici per l'Algebra Lineare*. Zanichelli, Bologna, 1988.

[9] R. A. Brualdi and H. J. Ryser. *Combinatorial Matrix Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1991.

[10] R. A. Brualdi and H. Schneider. Determinantal Identities: Gauss, Schur, Cauchy, Sylvester, Kronecker, Jacobi, Binet, Laplace, Muir, and Cayley. *Linear Algebra and its Applications*, 52(53):769–791, 1983.

[11] P. G. Charlet, editor. *Foundations of Computational Mathematics*, volume XI of *Handbook of Numerical Analysis*. Elsevier Science B. V., 2003.

[12] X. Chen and M. T. Chu. On the Least Squares Solution of Inverse Eigenvalue Problems. 1996.

[13] M. T. Chu. Inverse Eigenvalue Pproblems. 1998.

[14] M. T. Chu and G. H. Golub. *Inverse Eigenvalue Problems. Theory, Algorithms and Apllications.* Numerical Mathematics and Scientific Computation. Oxford University Press, 2005.

[15] R. W. Cottle. Manifestations of the Schur complement. *Linear Algebra and its Applications*, 8:189–211, 1974.

[16] G. N. De Oliveira. On the Multiplicative Inverse Eigenvalue Problem. *Canad. Math. Bull.*, 15(2), 1972.

[17] R. Diestel. *Graph Theory.* Springer, fourth edition, 2010.

[18] W. Ding, G. Yan, and Z. Lin. Collective motions and formations under pursuit strategies on directed acyclic graphs. *Elsevier*, (46):174–181, 2009.

[19] W. Ding, G. Yan, and Z. Lin. Formations on Two-Layer Pursuit Systems. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 3496–3501, Kobe, Japan, May 2009.

[20] J. A. Fax and R. M. Murray. Information flow and cooperative control of vehicle formations. *IEEE Transactions on Automatic Control*, 49(9):1465–1476, 2004.

[21] S. Fortin. The Graph Isomorphism Problem. Technical report, Department of Computing Science, the University of Alberta, Edmond, Alberta, Canada, July 1996.

[22] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, third edition, 1996.

[23] E. Gregorio. *Il pacchetto frontespizio*, 2011. http://ctan.mackichan.com.

[24] J. M. Hendrickx, B. D. Anderson, J. C. Delvenne, and V. D. Blondel. Directed graphs for the analysis of rigidity and persistence in autonomous agent systems. *Int. J. of Robust and Nonlinear Control*, (17):960–981, Nov 2007.

[25] L. Hogben. Spectral Graph Theory and the Inverse Eigenvalue Problem of a Graph. *Electronic Journal of Linear Algebra*, 14:12–31, Jan 2005.

[26] L. Hogben, editor. *Handbook of Linear Algebra*. Discrete Mathematics and its Applications. Chapman & All/CRC, Taylor and Francis Group, 2007.

[27] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

[28] Z. Lin. Lecture Notes on Distributed Control of Multi-Agent Systems. University of Cagliari, Sept 2009.

[29] Z. Lin, W. Ding, G. Yan, C. Yu, and A. Giua. Leader-Follower Formation via Complex Laplacian. Submitted in october, 2011.

[30] C. Moler. *Experiments with MatLab*. Mathworks Inc., electronic edition, 2011. http://www.mathworks.com/moler.

[31] L. Pantieri and T. Gordini. *L' Arte di scrivere con LaTeX*, Dec 2011. www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf.

[32] M. Passaquindici. Sulla stabilità dei polinomi e delle matrici. *Annali della scuola Normale Superiore di Pisa*, 13(1):77–88, 1959.

[33] M. Pavone and E. Frazzoli. Decentralized policies for geometric pattern formation. *ASME Journal of Dynamic Systems, Measurement, and Control*, 129(5):633–643, Sept 2007.

[34] W. Ren. Collective Motion from Consensus with Cartesian Coordinate Coupling - Part I: Single-integrator Kinematics. In *Proocedings of the 47th IEEE Conference on Decision and Control*, Cancun, Messico, Dec 2008.

[35] G. Rodriguez. *Algoritmi Numerici*. Pitagora Editrice, Bologna, 2008.

[36] P. Sargolzaei and R. Rakhshanipur. Improving the Localization of Eigenvalues for Complex Matrices. *Applied Mathematical Science*, 5(38):1857–1864, 2011.

[37] R. M. Tailor and P. H. Bhathawala. Location of the Eigenvalue of Complex Matrices. *International Journal of Advanced Research and Studies*, 1(2):277–278, Jan–Mar 2012.

[38] T. Tantau. *The TikZ and PGF Packages*. Institut für Theoretische Informatik Universität zu Lübeck, Oct 2010.

[39] M. J. Tsatsomeros. Principal Pivot Transforms: properties and applications. *Linear Algebra and its Applications*, 307(1–3):151–165, Mar 2000.

[40] T. Vicset, A. Czirok, E. Ben-Jacob, I. Cohen, and O. Shochet. Novel Type of Phase Transition in a System of Self-Driven Particles. *Physical Review Letters*, 75(6), Aug 1995.

[41] E. W. Weisstein. Faulhaber's Formula. MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/FaulhabersFormula.html.

[42] C. R. Werner. *Methods for Solving Systems of Nonlinear Equations*. SIAM, Philadelphia, second edition, 1998.

[43] J. Wu, P. Zhang, and Y. Wang. The Location for Eigenvalues of Complex Matrices by a Numerical Method. *J. Appl. Math. & Informatics*, 29(1–2):49–53, 2011.