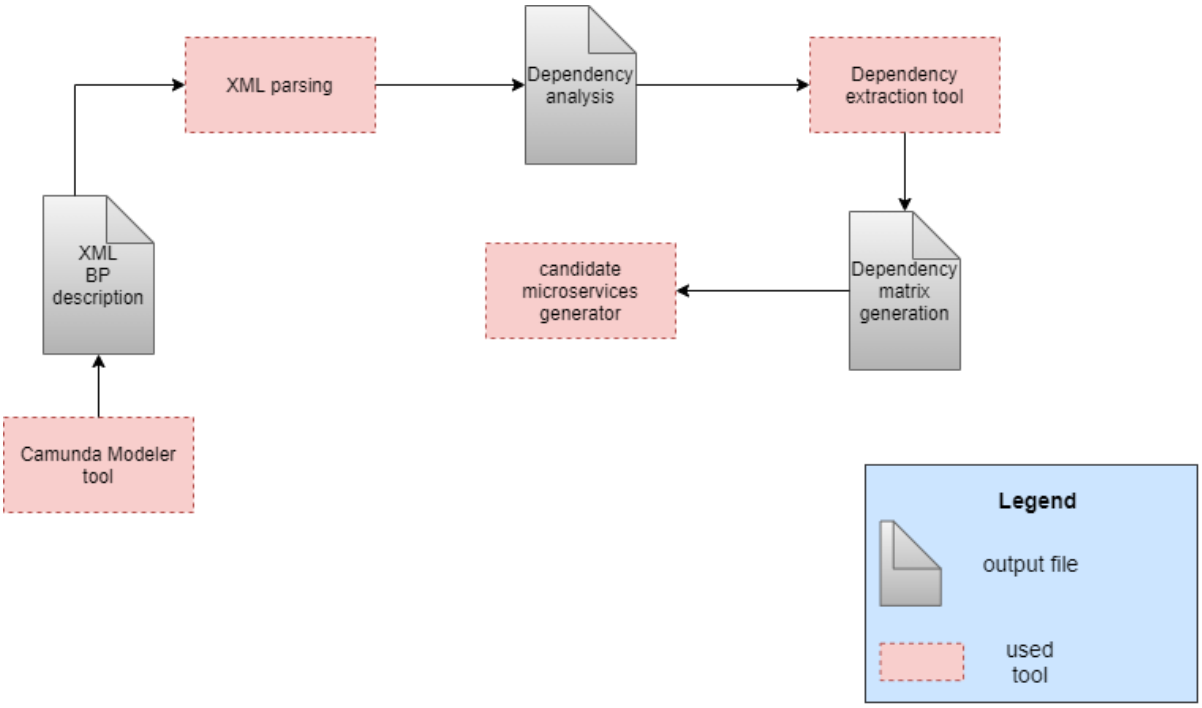
Implementation

A. Architecture



The goal of this part is to provide the global architecture of the MSI (microservices identification) tool to identify automatically the microservices. This tool is composed of:

1) A dependency extractor to apply the correct formula for each operator.
2) And clustering using the K-means algorithm to determine microservices.

Camunda modeler tool : This tool allows us to create an example of BP and generate its XML file. This file describes in detail our BP (connectors, arcs and activities).

XML parsing:

This tool takes the XML file generated by camunda modeler tool in order to analyze it and determine the different types of connectors. In other words, this tool will allow us to clearly and explicitly determine the dependency relationship.

Dependency extraction tool:

Based on the XML parsing output, dependency extraction tool allows to calculate the different structural dependencies between a couple of given activity to generate in the end the dependency matrix.

Candidate microservices generator: It is a tool which is based on the clustering technique and which takes as input the generated dependency matrix to identify the candidate microservices.

B. Algorithmic description of the developed tool

Algorithme *calculateMatrix*

```
// Algorithme qui calcule la valeur entre les différents taches
```

Variables

```
alltasks : List<Task> // la liste des tous les taches dans le graph

allEvents : List<Event> // la liste des évenements dans le graph
```

Début

```
matrix <- initiateMatrix() ; // Phase 1 : Initialiser la matrice avec des valeurs égales à 0.0

matrix <- calculateDirectDependeciesFromEvent (matrix) ; // Phase 2 : mettre à jour la matrice en calculant les valeurs entre tous les taches qui ont une relation directe entre eux.

matrix <- calculateIndirectCosts (matrix) ; // Phase 3 :; mettre à jour la matrice en calculant les valeurs entre les taches qui n'ont pas une relation directe entre eux en s'appuyant sur les valeurs qui ont été déjà calculés( les valeurs entre les taches ayant des relations directes)

matrix <- calculateAnotherPath (matrix) ; // Phase 4 : mettre à jour la matrice en cherchant s'il y a des autres alternatives pour passer entre deux taches marqués dans l'étape précédente comme liaison impossible en s'appuyant sur les valeurs de matrice.

matrix <- calculateInverseCosts (matrix) ; //Phase 5 : mettre à jour la matrice pour que la matrice soit symétrique
```

Fin

Méthode *calculateDirectDependeciesFromEvent*

```
// Méthode qui calcule les valeurs entre tous les taches qui ont une relation directe entre eux
```

Variables

```
matrix : une matrice de réels // la matrice à mettre à jour

alltasks : List<Task> // la liste des tous les taches dans le graph

allEvents : List<Event> // la liste des évenements dans le graph
```

Début

```
Pour i allant de 0 à allEvents.size() - 1 faire

    event <- allEvents(i) ; // L'évènement à la position i dans la liste de tous les évènements

    Si (event instanceof SequenceFlow) alors
```

matrix <- calculateDependecyFromSequenceEvent(event, matrice) // mettre à jour la matrice en calculant la valeur entre les taches reliés à cet évènement Sequence

Si non si** (event instanceof ExclusiveGateway) **alors

matrix <- calculateDependecyFromXOREvent (event, matrice) // mettre à jour la matrice en calculant la valeur entre les taches reliés à cet évènement XOR

Fin si

Fin pour

Fin

Méthode *calculateDependecyFromSequenceEvent*

// Méthode qui calcule les valeurs entre tous les taches reliés à un évènement séquence

Variables

matrix : une matrice de réels // la matrice à mettre à jour

alltasks : List<Task> // la liste des tous les taches dans le graph

allEvents : List<Event> // la liste des évenements dans le graph

p : réel //la valeur initial entre deux processus ayant une liaison directe

Début

sourceTask <- event.getSourceRef() // Trouver le processus source de l'évènement

targetTask <- event.getTargetRef() // trouver le processus destination de l'évènement

matrice[sourceTask.getId_matrice()][targetTask.getId_matrice()] <- p // la valeur entre deux processus d'un évenement sequence est égale à p
matrice[targetTask.getId_matrice()][sourceTask.getId_matrice()] = -2.0 // Une fois on calcule la valeur entre i et j dans la matrice, on mets le valeur entre j et i comme -2.0 pour dire que sa symétrie dans la matrice a été calculée. Ce valeur sera mis dans la dernière étape de l'algorithme.

Fin

Méthode *calculateDependecyFromXOREvent*

// Méthode qui calcule les valeurs entre tous les taches reliés à un évènement séquence

Variables

matrix : une matrice de réels // la matrice à mettre à jour

alltasks : List<Task> // la liste des tous les taches dans le graph

allEvents : List<Event> // la liste des évenements dans le graph

p : réel //la valeur initial entre deux processus ayant une liaison directe

Début

incomingTask<- event.getIncomingList().get(0)// Trouver le processus entrant à l'évènement (une seule processus à l'éntrée d'un evenement xor)

List<Task> outgoingTasks <- event. getOutgoingList() // Trouver la liste des processus à la sortie de l'évenement xor

Pour** i allant de 0 à outgoingTasks.size() -1 **faire

outgoingTask<- outgoingTasks.get(i) // le processus à la position i de la liste des processus sortants de l'évenement

matrice[incomingTask.getId_matrice()][outgoingTask.getId_matrice()] = p(1/ outgoingTasks.size());* // Calculer la relation entre le processus entrant et le processus sortant

matrice[outgoingTask.getId_matrice()][incomingTask.getId_matrice()] = -2.0; // Une fois on calcule la valeur entre i et j dans la matrice, on mets le valeur entre j et i comme -2.0 pour dire que sa symétrie dans la matrice a été calculée. Ce valeur sera mis dans la dernière étape de l'algorithme.

Fin pour

// Pour un évènement XOR, le nombre des processus sortants sont soit 1 soit 2. Si le nombre est 2, il faut spécifier que la relation entre les deux processus sortant de cet évènement est impossible. Une valeur de -1.0 veut dire un chemin non existant ou une relation impossible.

si**(outgoingTasks.size() == 2) **alors

matrice[outgoingTasks.get(0).getId_matrice()][outgoingTasks.get(1).getId_matrice()] = -1.0

matrice[outgoingTasks.get(1).getId_matrice()][outgoingTasks.get(0).getId_matrice()] = -1.0

fin si

Fin

Méthode *calculateIndirectCosts*

// Méthode qui calcule les valeurs entre tous les processus qui n'ont pas une relation directe entre eux

Variables

```
matrix : une matrice de réels // la matrice à mettre à jour

alltasks : List<Task> // la liste des tous les taches dans le graph

allEvents : List<Event> // la liste des évenements dans le graph

p : réel //la valeur initial entre deux processus ayant une liaison directe
```

Début

```
Pour i allant de 0 à allTasks.size()-2 faire
      Pour j allant de i +1 à allTasks.size()-1 faire

            Si (matrix[i][j] == 0.0) alors //le cost entre i et j n'a pas été
      calculé
                  Task taskI = findTaskByMatriceIndex(i); // trouver le processus
            à la position i
                  Task taskJ = findTaskByMatriceIndex(j); //Trouver le processus
            à la position j

                  matrice[i][j] = calculateIndirectCost(taskI, taskJ, matrice);
// Appeler une methode recursive pour calculer le cout indirect entre les deux
procesus

                  matrice[j][i] = -2.0; // Le valeur inverse sera mis à -2.0
            Fin si

      Fin pour
Fin pour
Fin
```

Méthode `calculateIndirectCost`

```
// Méthode qui calcule la valeur entre deux processus non voisins
```

Variables

```
matrix : une matrice de réels // la matrice à mettre à jour

alltasks : List<Task> // la liste des tous les taches dans le graph

allEvents : List<Event> // la liste des évenements dans le graph

p : réel //la valeur initial entre deux processus ayant une liaison directe

taskI : le premier processus

taskJ : le deuxième processus
```

Début

```
i <- taskI.getId_matrice(); //la position du premier processus dans la matrice

j <- taskJ.getId_matrice() //la position du deuxième processus dans le matrice

si (matrix [i][j] != 0.0) alors // la valeur a été déjà calculé

      retourner matrice[i][j]
```

```
Fin si

sucessiveTask <- getNextTaskFromSourceIndex(i,j, matrice) // trouver le procesus intermédiaire qui peut nous amener à la destination.

Si (sucessiveTask == null) alors

Retourner -1.0 // pas des taches intermidiares : Il est fort probable qu'il y a aucun liaison entre les deux processus ( à vérifier dans l'étape suivante)

Fin si

retourner matrice[i][sucessiveTask.getid_Matrice()] * calculateIndirectCost(sucessiveTask, taskJ, matrice)) // le valeur entre les deux processus i et j sera la multiplication de la valeur entre i et le processus intermédiaire et entre la valeur entre le processus intermédiaire et j.

Fin
```

Méthode *getNextTaskFromSourceIndex*

```
// Méthode qui trouve la tache suivante sur lequel on peut s'apuyer pour trouver la valeur entre i et j

Variables

matrix : une matrice de réels // la matrice à mettre à jour

alltasks : List<Task> // la liste des tous les taches dans le graph

allEvents : List<Event> // la liste des évenements dans le graph

p : réel //la valeur initial entre deux processus ayant une liaison directe

sourceIndex:entier // la position du premier processus

targetIndex: entier // la position de deuxième processus

Début

//Initialiser la liste des positions de taches sucessuers

allIndexes : list des entiers,

//Trouver toutes les cellules dans la matrice dont i = sourceIndex et leurs valeurs ont été déja calculés
Pour i allant de 0 à alltasks.size()-1 faire

        si(matrix[sourceIndex][j] != 0.0) alors

                allIndexes.add(j)

        Fin si

Fin Pour

Exclure les sucessuers qui ne peuvent pas me prendre vers le target (liaison non existante)
allIndexes <- allIndexes.filter(index -> matrice[index][targetIndex] != 0.0 && matrice[index][targetIndex] != -1.0 && matrice[index][targetIndex] != -2.0)
```

```
si(!allIndexes.isEmpty()) alors // sucuesseurs existants pour calculer la valeur entre i et le target

//Trouver le successeur avec cout maximal vers le taregt
        indexMax <- targetInts.get(0)
        Pour i allant de 1 à allIndexes.size() -1 faire
        Index <- allIndexes.get(i)
                Si( matrice[sourceIndex][index] * matrice[index][targetIndex] >
        matrice[sourceIndex][indexMax] * matrice[indexMax][targetIndex] ) alors
                        indexMax = index
                Fin si
        Fin pour

        retourner findTaskByMatriceIndex(indexMax)// retourner le processus dont sa position dans la matrice est indexMax
Si non
        retourner null
Fin si
Fin
```

Méthode *calculateAnotherPath*

```
// Méthode qui cherhce une autre aletrnative pour les processus procesus ont été marqué dans l'étape suivante comme non liées ou independants
```

Variables

```
matrix : une matrice de réels // la matrice à mettre à jour

alltasks : List<Task> // la liste des tous les taches dans le graph

allEvents : List<Event> // la liste des évenements dans le graph

p : réel //la valeur initial entre deux processus ayant une liaison directe
```

Début

```
Pour i allant de 0 à alltasks.size()-1 faire

        Pour j allant de 0 à alltasks.size()-1 faire

                Si matrix[i][j]== -1.0 // la lisaison est marqué comme non existant dans la phase précedente

                        Task_k <- getNextTaskFromSourceIndex(i,j)   // Trouver la tache intermidiare qui peut nous améner à j à partir de i : Trouver un tache intermidaire dont son valeur avec i et j est calculé

                        Si (task_k != null) alors // si cette tache existe

                                Matrix[i][j]= matrice[i][task_k.getId_matrice()] *
                        matrice[task_k.getId_matrice()][j]

                        Si non

                                Task_k <- getNextTaskFromSourceIndex(j,i)  // trouver une tache intermidaire qui peut nous améner de j vers i : on teste si le valeur inverse est calculé
```

```
                    Si (task_k != null) alors // si cette tache existe
                    Matrix[i][j]= matrice[j][task_k.getId_matrice()] *
                    matrice[task_k.getId_matrice()][i]
                    Fin si,

            Fin si
        Fin si
    Fin pour
Fin Pour
```

Fin

Méthode calculateInverseCosts

```
// Méthode qui calcule les valeurs inverses pour rendre le matrice symétrique : si
le valeur entre i et j a été cacluclé ,on mets à jour la valeur entre j et i et si
ce dernier a été calculé, on met à jour celui entre i et j
```

Variables

```
matrix : une matrice de réels // la matrice à mettre à jour
alltasks : List<Task> // la liste des tous les taches dans le graph
```

Début

```
Pour i allant de 0 à alltasks.size()-1 faire
    Pour j allant de 0 à alltasks.size()-1 faire
        Si matrix[i][j] != -2.0 // la valeur entre i et j est calculé
            matrix[j]i] = matrice [i][j]
        Si non // la veleur entre i et j est non calculé
            matrice [i][j] = matrix[j]i] ;
            Fin si
Fin pour
    Fin pour
Fin Pour
```

Fin

C. some screenshots of the application

1. Input Data

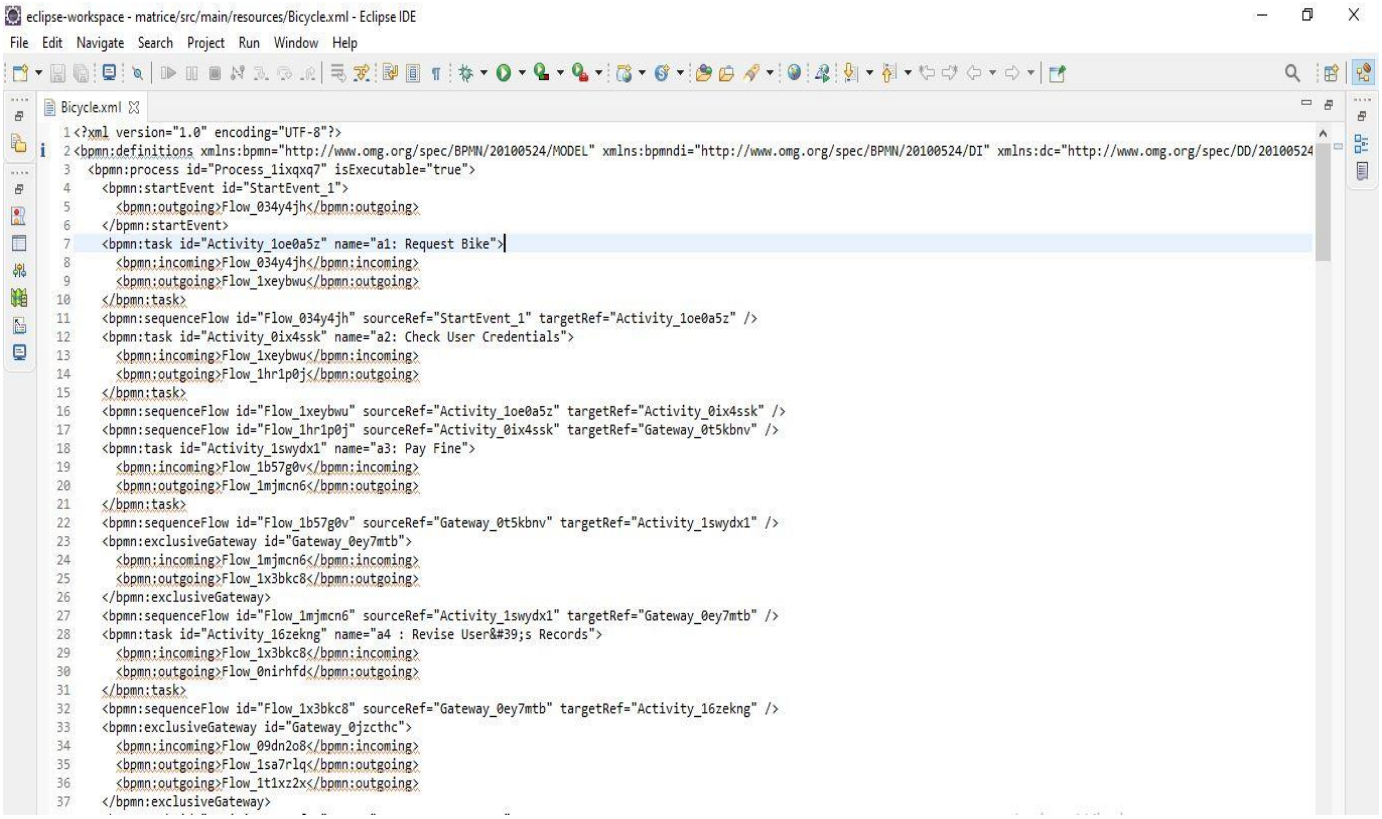


Figure 1: XML file of bicing system

2. Dependency calculation: sequence case

```
1  package com.example.matrice.models;
2
3  import com.example.matrice.services.TaskService;
9
10
11 public class SequenceFlow extends Event{
12     public String id;
13     public String sourceRef;
14     public String targetRef;
15
16     @Autowired
17     TaskService taskService;
18
19     @Override
20     public List<Dependence> calcul(List<Task> taskList, BigDecimal p, List<Event>events,List<S
21         Task taskIncomoing=this.findByIncoming(taskList,this.getId());
22         Task taskOutgoing=this.findByOutGoing(taskList,this.getId());
23         List<Dependence> dependences=new ArrayList<>();
24         if(taskIncomoing != null && taskOutgoing != null) {
25             Dependence dependence = new Dependence();
26             dependence.setAct1(taskIncomoing.name);
27             dependence.setAct2(taskOutgoing.name);
28             dependence.valeur = p;
29             dependences.add(dependence);
30         }
31         return dependences;
32     }
33
34     public Task findByIncoming(List<Task> tasks,String incoming){
35         for (Task task : tasks) {
36             if(task.getIncomingList().contains(incoming)){
37                 return task;
38             }
39         }
40         return null;
41     }
42
43     public Task findByOutGoing(List<Task> tasks, String outgoing){
44         for (Task task : tasks) {
```

Figure 2: sequence case

3. Dependency calculation: Xor_case

```java
package com.example.matrice.models;



import java.math.BigDecimal;


public class ExclusiveGateway extends Event {

    public String id;
    public List<String > incomingList=new ArrayList<>();
    public List<String > outgoingList=new ArrayList<>();

    @Override
    public List<Dependence> calcul(List<Task> taskList, BigDecimal p, List<Event>events,List<St
        List<Task> incomingTasks=new ArrayList<>();
        incomingTasks=getIncomingAllTask(taskList,this.getIncomingList(),events);
        List<Task> outGoingTasks=new ArrayList<>();
        this.getOutgoingList().forEach(s -> {
            Task task=findByOutGoing(taskList,s);
            if(task != null){
                outGoingTasks.add(task);
            }
        });

        BigDecimal valeur=new BigDecimal(1).divide(new BigDecimal(outgoingList.size())).multipl
        List<Dependence> dependences=new ArrayList<>();
        incomingTasks.forEach(task -> {
            outGoingTasks.forEach(task1 -> {
                Dependence dependence=new Dependence();
                dependence.setAct1(task.name);
                dependence.setAct2(task1.name);
                dependence.setValeur(valeur);
                dependences.add(dependence);
            });
        });
        Dependence dependence=new Dependence();
```

Figure 3: Xor case

4. Dependency calculation: Or_case

```java
package com.example.matrice.models;



import java.math.BigDecimal;


public class InclusiveGateway extends Event {

    public String id;
    public List<String > incomingList=new ArrayList<>();
    public List<String > outgoingList=new ArrayList<>();

    @Override
    public List<Dependence> calcul(List<Task> taskList, BigDecimal p, List<Event>events,List<St
        List<Task> incomingTasks=new ArrayList<>();
        incomingTasks=getIncomingAllTask(taskList,this.getIncomingList(),events);
        List<Task> outGoingTasks=new ArrayList<>();
        this.getOutgoingList().forEach(s -> {
            Task task=findByOutGoing(taskList,s);
            if(task != null){
                outGoingTasks.add(task);
            }
        });

        BigDecimal valeur=new BigDecimal(1).divide(new BigDecimal(outgoingList.size())).multipl
        List<Dependence> dependences=new ArrayList<>();
        incomingTasks.forEach(task -> {
            outGoingTasks.forEach(task1 -> {
                Dependence dependence=new Dependence();
                dependence.setAct1(task.name);
                dependence.setAct2(task1.name);
                dependence.setValeur(valeur);
                dependences.add(dependence);
            });
        });
        Dependence dependence=new Dependence();
        return dependences;
```

Figure 4: Or case

5. Dependency calculation: And_case

```java
package com.example.matrice.models;



import java.math.BigDecimal;


public class ParallelGateway extends Event {

    public String id;
    public List<String > incomingList=new ArrayList<>();
    public List<String > outgoingList=new ArrayList<>();

    @Override
    public List<Dependence> calcul(List<Task> taskList, BigDecimal p, List<Event>events,List<St
        List<Task> incomingTasks=new ArrayList<>();
        incomingTasks=getIncomingAllTask(taskList,this.getIncomingList(),events);
        List<Task> outGoingTasks=new ArrayList<>();
        this.getOutgoingList().forEach(s -> {
            Task task=findByOutGoing(taskList,s);
            if(task != null){
                outGoingTasks.add(task);
            }
        });

        BigDecimal valeur=new BigDecimal(1).divide(new BigDecimal(outgoingList.size())).multipl
        List<Dependence> dependences=new ArrayList<>();
        incomingTasks.forEach(task -> {
            outGoingTasks.forEach(task1 -> {
                Dependence dependence=new Dependence();
                dependence.setAct1(task.name);
                dependence.setAct2(task1.name);
                dependence.setValeur(valeur);
                dependences.add(dependence);
            });
        });
        Dependence dependence=new Dependence();
        return dependences;
```

6. Matrix generation

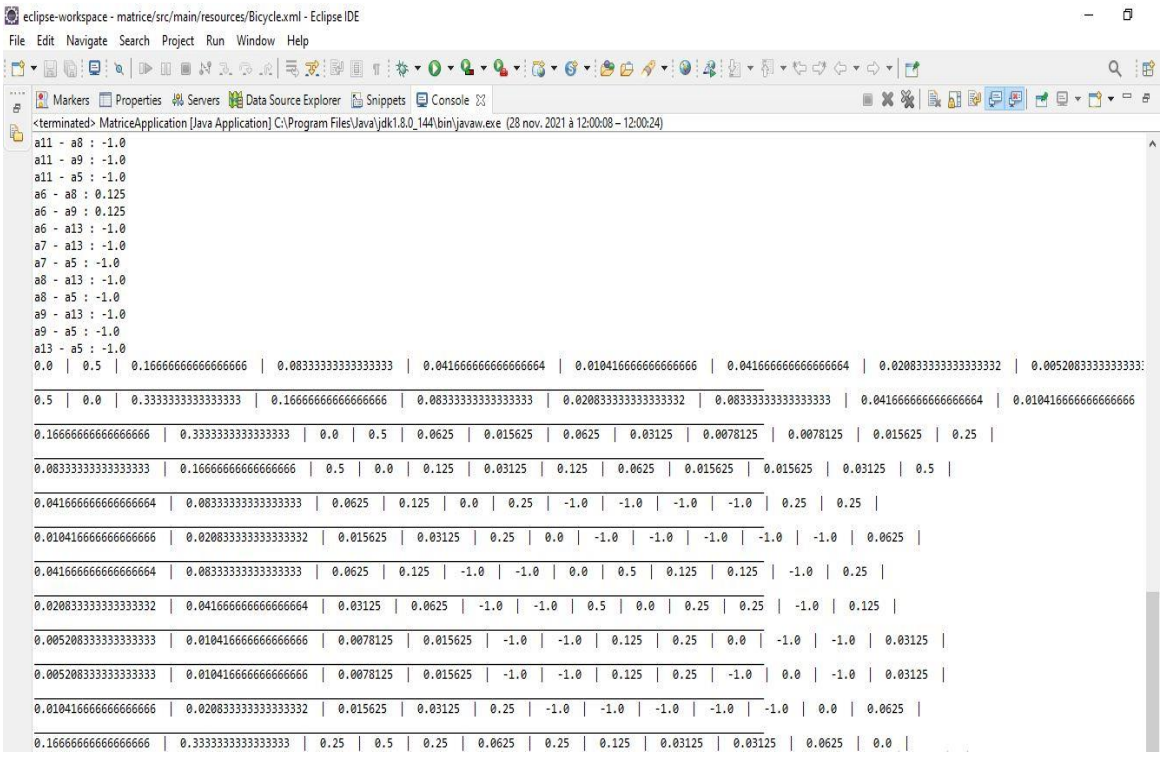


Figure 5: Matrix generation