

Conception d'un système robotique avancé pour une application de micro pick-and-place

Travail de Bachelor

Département TIN
Filière Génie Électrique
Orientation Électronique et Automatisation industrielle

Gaëtan Worch

8 février 2025

Supervisé par :
Prof. G. Costanzo (HEIG-VD)

Préambule

Ce travail de Bachelor (ci-après **TB**) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'École.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD
Le Chef du Département

Yverdon-les-Bains, le 8 février 2025

Authentification

Je soussigné, Gaëtan Worch, atteste par la présente avoir réalisé seul ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Gaëtan Worch

A handwritten signature in black ink, appearing to read 'Gaëtan Worch', written over a horizontal line.

Yverdon-les-Bains, le 8 février 2025

Résumé

Table des matières

Préambule	i
Authentification	iii
Résumé	v
1 Introduction	1
1.1 SwissCat+	1
1.1.1 Laboratoire ouest	1
1.2 Contexte et objectifs du projet Storm	1
1.3 Recombinaison de microcapsules	2
2 Problématique de la recombinaison	3
2.1 Objectifs	3
2.2 Cahier des charges fonctionnel	3
3 Hardware	5
3.1 Défis techniques du Hardware	5
3.2 Recherches de solution	5
3.2.1 Saisie et dépose des microcapsule	5
3.2.2 Déplacement des microcapsule	6
3.2.3 Analyse des solutions	8
3.3 Présentation de la solution choisie	13
3.4 Description du matériel utilisé	13
3.5 Intégration	13
4 Software	15
4.1 Algorithme de recombinaison	15
4.1.1 Spécification de l'algorithme	15
4.1.2 Définition du problème	16
4.1.3 Méthode d'optimisation	17
4.2 Robot	21
4.2.1 Objectif	21
4.2.2 Contraintes	21
4.2.3 Méthode de programmation	21
4.2.4 Outils	21
4.2.5 Sécurité	21
4.2.6 Repères de travail	22
4.2.7 Communication avec le robot	22

TABLE DES MATIÈRES

5	Implémentation et intégration Hardware-Software	23
5.1	Développement du logiciel	23
5.1.1	Objets	23
5.1.2	Optimisateur	24
5.2	Tests de validation et calibrage du matériel	24
5.2.1	Tests unitaires	24
6	Sécurité et Préventions des Risques	27
6.1	Identification des risques	27
6.2	Solutions	29
7	Résultat et Analyse	31
7.1	Performance du système	31
7.2	Analyse des erreurs	31
8	Discussion	33
8.1	Points forts	33
8.2	Limitations	33
8.3	Perspectives	33
9	Conclusion	35
	Appendices	41
A	Optimisateur	41
B	Tests unitaires	51

Table des figures

1.1	Schema des modules de STORMS	2
3.1	Exemples de réseau de transport pneumatique par tube	6
3.2	Exemple de convoyeur	7
3.3	Schéma de la <i>glove box</i> Recombinaison	8
3.4	Schéma des solutions (cf. 3.2.3). (En rouge la <i>wellplate</i> , en bleu la paradox)	9
3.5	Temps de cycle moyen des différentes solutions	10
3.6	Comparaison du temps de cycle des solutions en fonction du nombre de microcapsules par plaque	12
3.7	Intégration du robot dans la <i>glove box</i>	14
3.8	Module d'aspiration	14
4.1	Algorithme général de l'optimisateur.	17
4.2	Nombre de combinaisons possible en fonction de la taille des réacteurs	18
4.3	Temps de calcul en fonction de la taille des réacteurs	18
4.4	Position physique du plan « repPickAndPlace »	22
5.1	Diagramme de classe	23
5.3	Couverture du code par les tests unitaires	24
5.2	Diagramme de flux de l'optimisateur implémenté	25

Liste des tableaux

2.1	Cahier des charges fonctionnel	3
3.1	Avantages et inconvénients des solutions de saisie des microcapsules	6
3.2	Anvantages et inconvénients des solution de transport des microcapsules	8
6.1	Analyses des risques	28
6.2	Solutions face aux risques (dans l'ordre de priorité)	29

Liste des codes sources

assets/code/solver.py	42
A.1 Génération d'un diagramme de Bode	42
assets/code/solver.py	43
A.2 Génération d'un diagramme de Bode	43
assets/code/solver.py	44
A.3 Génération d'un diagramme de Bode	44
assets/code/solver.py	45
A.4 Génération d'un diagramme de Bode	45
assets/code/solver.py	46
A.5 Génération d'un diagramme de Bode	46
assets/code/solver.py	47
A.6 Génération d'un diagramme de Bode	47
assets/code/solver.py	48
A.7 Génération d'un diagramme de Bode	48
assets/code/solver.py	49
A.8 Génération d'un diagramme de Bode	49
assets/code/test_batch.py	52
B.1 Test unitaire de batch.py	52
assets/code/test_batch.py	53
B.2 Test unitaire de batch.py	53
assets/code/test_recipe.py	53
B.3 Test unitaire de recipe.py	53
assets/code/test_solve.py	54
B.4 Test unitaire de solve.py	54
assets/code/test_solve.py	55
B.5 Test unitaire de solve.py	55
assets/code/test_solve.py	56
B.6 Test unitaire de solve.py	56
assets/code/test_solve.py	57
B.7 Test unitaire de solve.py	57
assets/code/test_solve.py	58
B.8 Test unitaire de solve.py	58
assets/code/test_storage.py	59
B.9 Test unitaire de storage.py	59
assets/code/test_storage.py	60
B.10 Test unitaire de storage.py	60
assets/code/test_vial.py	61
B.11 Test unitaire de vial.py	61

Chapitre 1

Introduction

1.1 *SwissCat+*

Les catalyseurs est une espèce chimique qui permet ou accélère la réaction chimique sans être consommé dans le processus. Les catalyseurs sont utilisées dans divers domaines (Agriculture, militaire, chimie, traitements des déchets, transformation de polluants, ...). Ils sont indispensables dans notre société, cependant, ces produits requièrent souvent des terres rares. Par exemple, les catalyseurs dans les voitures (qui servent à réduire les émissions de gaz polluant en transformant notamment le monoxyde de carbone, les hydrocarbures imbrûlés et de l'oxyde d'azote en eau, dioxyde de carbone et dioxyde d'azote) sont composée d'alumine, d'oxyde de cérium mais surtout ils sont composés d'au moins trois platinoïde.

C'est dans ce contexte de réduction d'utilisation de terre rare que le laboratoire, SwissCat+, a été créé dans le but d'optimiser la composition des catalyseurs et d'en trouver des nouveaux. Le laboratoire est scindé en deux, le laboratoire Est, située à l'ETHZ à Zurich, s'occupe de faire des recherches sur les catalyseurs hétérogène, tandis que la partie du laboratoire Ouest, situé à l'EPFL à Lausanne, fera des recherches sur les catalyseurs homogènes.

1.1.1 *Laboratoire ouest*

Le laboratoire doit être automatiser afin de pouvoir effectuer de grandes quantités d'expériences pour explorer un maximum l'espace chimique. Pour avoir un espace chimique le plus grand possible, il faut que le laboratoire puisse manipuler des matières liquides et de larges quantités de solides (0.1 à 50 mg).

1.2 *Contexte et objectifs du projet Storm*

Le projet STORMS (*STOchastic Robotized Micro Sampling*) est né d'une collaboration entre l'EPFL, l'HEIG-VD, Chemspeed Technologies et Dietrich Engineering Consultants (DEC) afin de pouvoir manipuler cette plage de quantité de solides.

STORMS est composés de 7 modules (cf. Figure 1.1) :

- la standardisation,
- le stockage,
- le micro-échantillonnage (*microsampling*),
- la recombinaison,

- la synthBox
- 2 box de Chemspeed

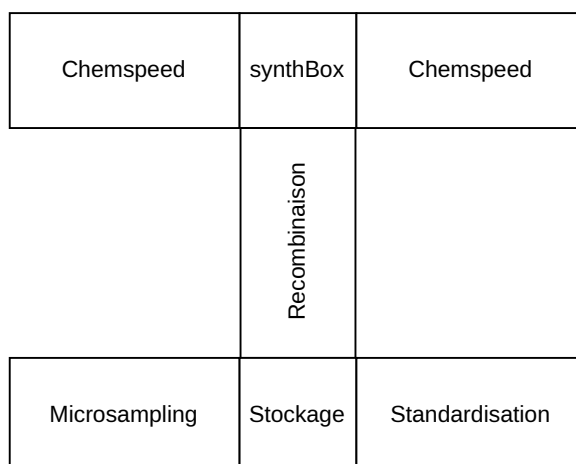


Figure 1.1 – *Schema des modules de STORMS*

La standardisation a pour but de donner des récipients de taille standard aux microsampling qui va créer des microcapsule. Les microcapsule peuvent contenir de 0.1 à 10 mg de produit. Dû aux grandes différences de caractéristique des produits à expérimenter, il n'est pas possible de déterminer précisément à l'avance la quantité qui sera délivrée dans les microcapsule par le *microsampling*. Les microcapsule, une fois fabriquées, sont pesées puis mises sur plaques nommées *wellplate*, qui sont ensuite stockées. Le stockage fait lien entre la standardisation, le *microsampling* et la recombinaison. La recombinaison doit réaliser les recettes voulues à partir du stock disponible.

1.3 *Recombinaison de microcapsules*

Le processus de recombinaison consiste à assembler différentes microcapsule de poudres pour répondre aux besoins expérimentaux. Le logiciel de recombinaison sélectionne parmi les microcapsule disponibles afin d'obtenir une composition entrant dans les tolérances quantités pour une expérience donnée. Le but est de maximiser le nombre d'expériences réalisées. Une fois les microcapsule sélectionnées, la *glove box* recombinaison doit placer ces microcapsule dans les réacteurs correspondant aux recettes réalisées.

Chapitre 2


Problématique de la recombinaison

2.1 Objectifs

Concevoir et développer une solution permettant le transfert automatisé, précis et rapide de microcapsules de réactif chimique entre le stockage et les réacteurs. Un algorithme devra être conçu pour optimiser la sélection des microcapsules afin de maximiser le nombre de recettes réalisées. L'ensemble du système devra garantir un haut niveau de fiabilité et de sécurité dans le processus.



2.2 Cahier des charges fonctionnel

Table 2.1 – Cahier des charges fonctionnel

fonction	énoncé de la fonction	exigence
FP 1	Manipuler des microcapsules de manière automatisée, sans les endommagées	— Ne pas détériorer la microcapsule
FP 1.1	Prélever les microcapsules dans une plaque	— Position de prise arbitraire — Contrôler que la microcapsule soit saisie
FP 1.2	Déplacer les microcapsules	 — Position de dépose arbitraire dans la plaque de réacteurs
FP 1.3	Déposer les microcapsules	
Continued on next page		

CHAPITRE 2. PROBLÉMATIQUE DE LA RECOMBINAISON

Table 2.1– continued from previous page

fonction	énoncé de la fonction	exigence
FP 2	Déterminer les microcapsules les plus adaptées pour chaque réacteur, selon une recette donnée	
FP 2.1	Recevoir la recette pour chaque réacteur	
FP 2.2	Accéder à la base de donnée du stock	
FP 2.3	Déterminer la combinaison de microcapsules optimal pour délivrer la masse de produit donnée	<ul style="list-style-type: none"> — Précision dans la masse délivrée : varie à chaque recette. — Nombre maximale de microcapsules par réacteur : 5
FP 2.4	Transmettre la position des microcapsules dans le stock et sur la plaque	
FP 2.5	Informier le stock des microcapsules prélevées	
FC 1	Respecter les dimensions de l'endroit confiné	Dimension de la boîte : 133 × 95 × 94 cm
FC 2	Utiliser les énergies disponibles 	<ul style="list-style-type: none"> — Électrique : <ul style="list-style-type: none"> — 400 V triphasé — 230 V monophasé — Pneumatique : 8 bar
FC 3	Utiliser les plaques déjà présentes	<ul style="list-style-type: none"> — Nombre de trou sur la plaque de prise : 384 — Nombre de trou sur la plaque de dépose : 48
FC 4	S'adapter aux éléments déjà présents	— API : Beckhoff
FC 5	Dimension des microcapsules : Ø 3 mm	

Chapitre 3

Hardware

3.1 Défis techniques du Hardware

Le hardware doit pouvoir manipuler des *wellplates* ou des paradox, ainsi que des microcapsules en verre de 3 mm de diamètre en garantissant leur intégrité. Le tout doit être dans un espace confiné.

3.2 Recherches de solution

Pour la recherche des solutions, le hardware a été décomposé par les fonctions suivantes :

- saisie et dépose des microcapsules ;
- déplacement des microcapsules ;
- saisie et dépose des plaques.

La position des plaques dans la *glove box* a également été étudié avec 6 configurations différentes.

3.2.1 Saisie et dépose des microcapsule

Pour la saisie des microcapsules, les grandes familles de solutions proposées sont :

- Aspiration ;
- Mécanique ;
 - Pince à doigt ;
 - Pince Gecko.

Avantage et inconvénients**Table 3.1** – *Avantages et inconvénients des solutions de saisie des microcapsules*

Solution	Avantages	Inconvénient
Aspiration	<ul style="list-style-type: none"> - Exerce moins de pression directe - Simple à installer - Nécessite peu d'entretien 	<ul style="list-style-type: none"> - Apporter l'énergie pneumatique - Bruyant
Pince à doigt	<ul style="list-style-type: none"> - Contrôle précis - Faible coût 	<ul style="list-style-type: none"> - Maintenance fréquente - Ne convient pas au petits objets - Espace limité, pour pouvoir ouvrir et fermer la pince - Nécessite un contrôle de force
Pince Gecko	<ul style="list-style-type: none"> - Saisie non intrusive - Ne nécessite pas de source d'énergie externe 	<ul style="list-style-type: none"> - Capacité de charge - Nécessite un nettoyage pour maintenir l'adhérence - Détachement complexe

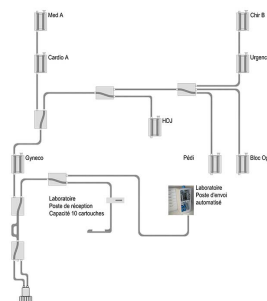
3.2.2 Déplacement des microcapsule

Pour le déplacement des microcapsules de leur plaque jusqu'aux réacteurs, trois idées ont été étudiées :

- Transport pneumatique par tube ;
- convoyeur ;
- robot.

Transport pneumatique par tube

Le système de transport pneumatique par tube, serait des tuyaux dans lesquelles naviguent les microcapsules grâce à une différence de pression de chaque côté de la microcapsule. Ce système est déjà présent dans les hôpitaux et dans les grandes surfaces.

**(a)** Schéma d'un réseau de transport pneumatique¹**(b)** Cartouche de transport²**Figure 3.1** – *Exemples de réseau de transport pneumatique par tube*

1. <https://www.transport-pneumatique.fr/transport-pneumatique-centres-hospitaliers/>
 2. <https://www.transport-pneumatique.fr/cartouches-pochettes/>

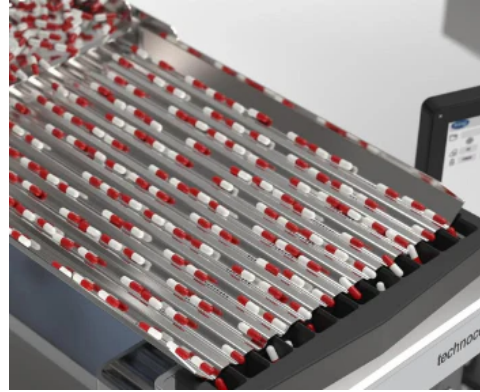
3.2. RECHERCHES DE SOLUTION

Transport par convoyeur

Pour déplacer les microcapsules, un convoyeur peut être utilisé, il faut néanmoins que le convoyeur soit adapté au microcapsule, les microcapsules étant cylindriques, elles risqueraient de rouler sur un convoyeur à bande lisse, mais une bande à tasseau (cf. Figure 3.2a) ou un demi-tube (cf. Figure 3.2b) conviendraient parfaitement.



(a) Convoyeur avec bande à tasseaux³



(b) Convoyeur à tube⁴

Figure 3.2 – Exemple de convoyeur

Quant aux différents moyens de mouvoir les microcapsules, il y a :

- les vibrations ;
- le déplacement de la bande ;
- la gravité.

La dernière option nécessite des surface lisses, que le système soit en pente et le temps de déplacement n'est pas réglable. Les deux autres solutions ne se distinguent pas vraiment pour l'instant, car dans tous les cas, l'utilisation d'un moteur électrique est nécessaire.

Déplacement à l'aide d'un robot

Pour le déplacement des microcapsules, seuls les axes T_x , T_y et T_z sont nécessaires, soit 3 degrés de liberté. Un robot de type *SCARA*, cylindrique ou Delta peuvent correspondre.

2. <https://fr.m.wikipedia.org/wiki/Convoyeur>

3. <https://doser-compter.com/products/ligne-de-comptage-king>

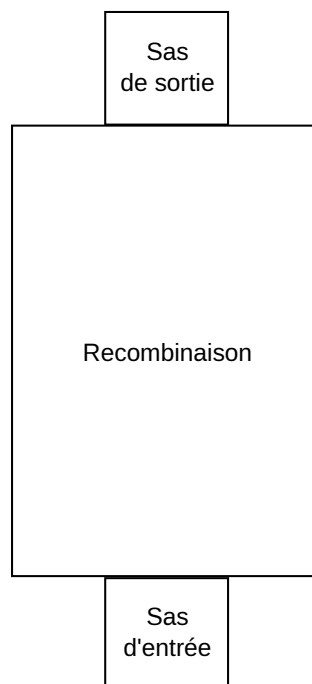
Avantages et inconvénients**Table 3.2** – *Avantages et inconvénients des solution de transport des microcapsules*

Solution	Avantages	Inconvénient
Transport pneumatique par tube		<ul style="list-style-type: none"> - Peu modulable - Bruyant - Aiguillage complexe
Convoyeur	<ul style="list-style-type: none"> - - 	<ul style="list-style-type: none"> - Maintenance fréquente - Ne convient pas au petits objets - Espace limité, pour pouvoir ouvrir et fermer la pince - Nécessite un contrôle de force
Robot	<ul style="list-style-type: none"> - Place - Modulable 	<ul style="list-style-type: none"> - Coût - Nécessite un nettoyage pour conserver l'adhérence dans le temps - Détachement complexe

3.2.3 Analyse des solutions

De par la complexité et le manque de modularité du transport pneumatique et du convoyeur, ~~le choix d'utiliser~~ un robot a été choisi.

La position des plaques dans la *glove box* (cf. Figure 3.3) est importante pour la suite, elle permettra de choisir le type de robot à utiliser. Il est important de savoir comment placer les deux plaques dans la *glove box*.

**Positionnement des plaques dans la glove box****Figure 3.3** – *Schéma de la glove box Recombinaison*

3.2. RECHERCHES DE SOLUTION

Pour la position des plaques, les solutions trouvées sont :

1. Les plaques ne bougent pas et restent dans les sas (cf. Figure 3.4a) ;
2. Les plaques sont positionnées symétriquement par rapport au centre de la *glove box* (cf. Figure 3.4b) ;
3. La plaque de microcapsules est déplacée à côté de la plaque de réacteur, cette dernière ne bouge pas (cf. Figure 3.4c) ;
4. La plaque de microcapsules reste dans le sas tandis que la plaque de réacteur est transportée à ses côtés (cf. Figure 3.4d) ;
5. Les deux plaques sont mises l'une à côté de l'autre en étant plus proches de la sortie (cf. Figure 3.4e) ;
6. Les deux plaques sont mises l'une à côté de l'autre en étant plus proches du stock (cf. Figure 3.4f).

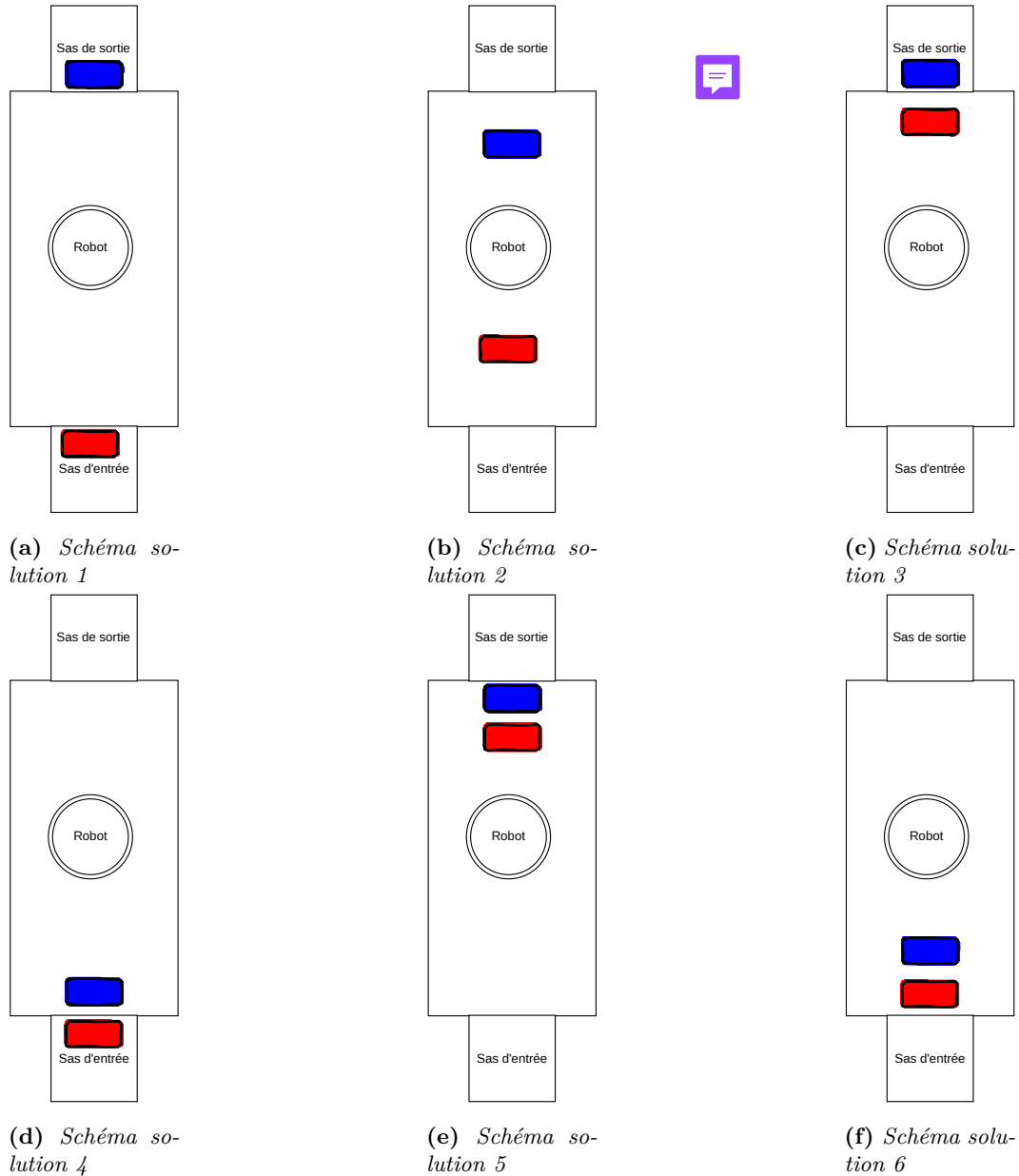


Figure 3.4 – Schéma des solutions (cf. 3.2.3). (En rouge la wellplate, en bleu la paradox)

Le temps de cycle est le suivant :

$$\begin{aligned}
 tpsCycle &= tpsDeplacementPlaquesCapsule + tpsDeplacementCapsule \\
 &+ tpsAttentePlaque + tpsDeplacementPlaquesReacteur \\
 &= nbrePlaques \cdot \frac{d_{PlaqueMicrocapsules}}{v_{robot}} \\
 &+ nbreMicrocapsules \cdot \frac{d_{EntrePlaque}}{v_{robot}} \\
 &+ 2 \cdot nbrePlaques \cdot tpsAttente + \frac{d_{PlaqueReacteur}}{v_{robot}}
 \end{aligned}$$

Certaines valeurs ne peuvent être connues que lors de la mise en route, notamment l'accélération, la vitesse d'approche, il est donc nécessaire de faire certaines hypothèses. Les distances sont également arbitraires afin de se faire une idée du temps de cycle des solutions. Ici, la position optimale n'est pas recherchée.

En utilisant les hypothèses suivantes :

- $v_{robot} = 1 \frac{m}{s}$;
- $d_{EntrePlaque} = 1.33, 0.3, 0.3, 0.3, 0.3$ et $0.3 m s^{-1}$;
- $d_{wellplate} = 0, 0.515, 1.03, 0, 0.83$ et $0.2 m$;
- $d_{paradox} = 0, 0.515, 0, 1.03, 0.2$ et $0.83 m$;
- L'accélération du robot est infinie ;
- La vitesse d'approche n'est pas prise en compte.

Avec ces informations, il est possible de calculer le temps moyen d'un cycle (cf. Figure 3.5) en fonction du nombre de microcapsule demandées ainsi que du nombre de microcapsules par plaque, et ce, pour chaque solution.

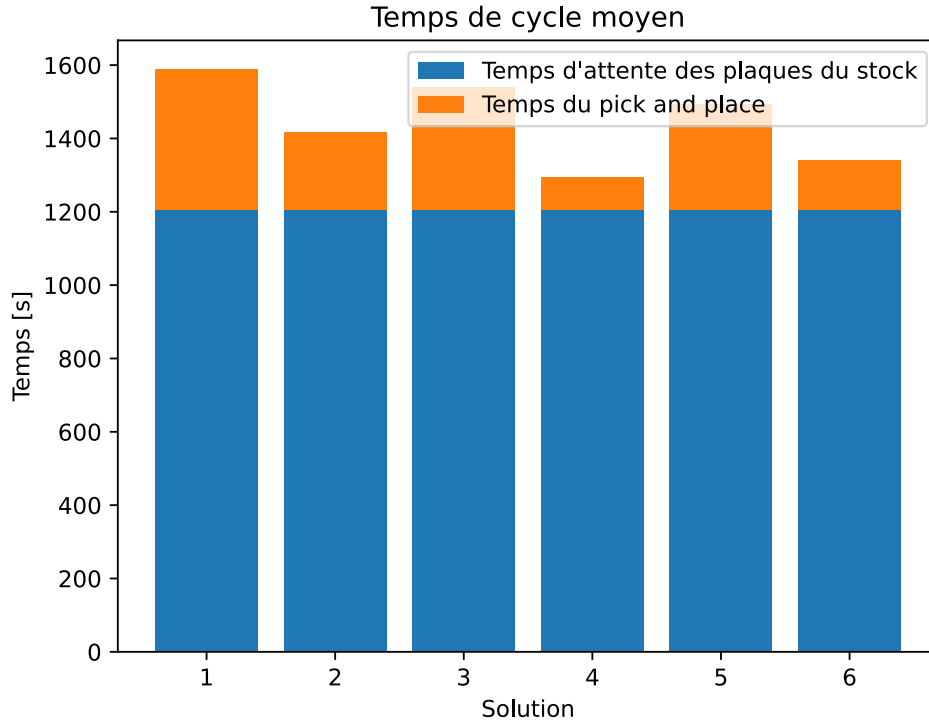


Figure 3.5 – Temps de cycle moyen des différentes solutions

3.2. RECHERCHES DE SOLUTION

Il est possible de voir que les solutions 4 et 6 semblent les plus rapide. Le temps d'attente des plaques est cependant très important, environ 84 % du temps total. Pour réduire ce délai, il peut être intéressant de paralléliser les tâches.



parallélisation des robots

La parallélisation des tâches consiste à effectuer le *pick and place* des microcapsules, la prise et la dépose des plaques indépendamment. De par la configuration des sas, il n'est possible d'y mettre qu'une seule plaque, la parallélisation des solutions 1, 3 et 4 n'est donc pas possible.

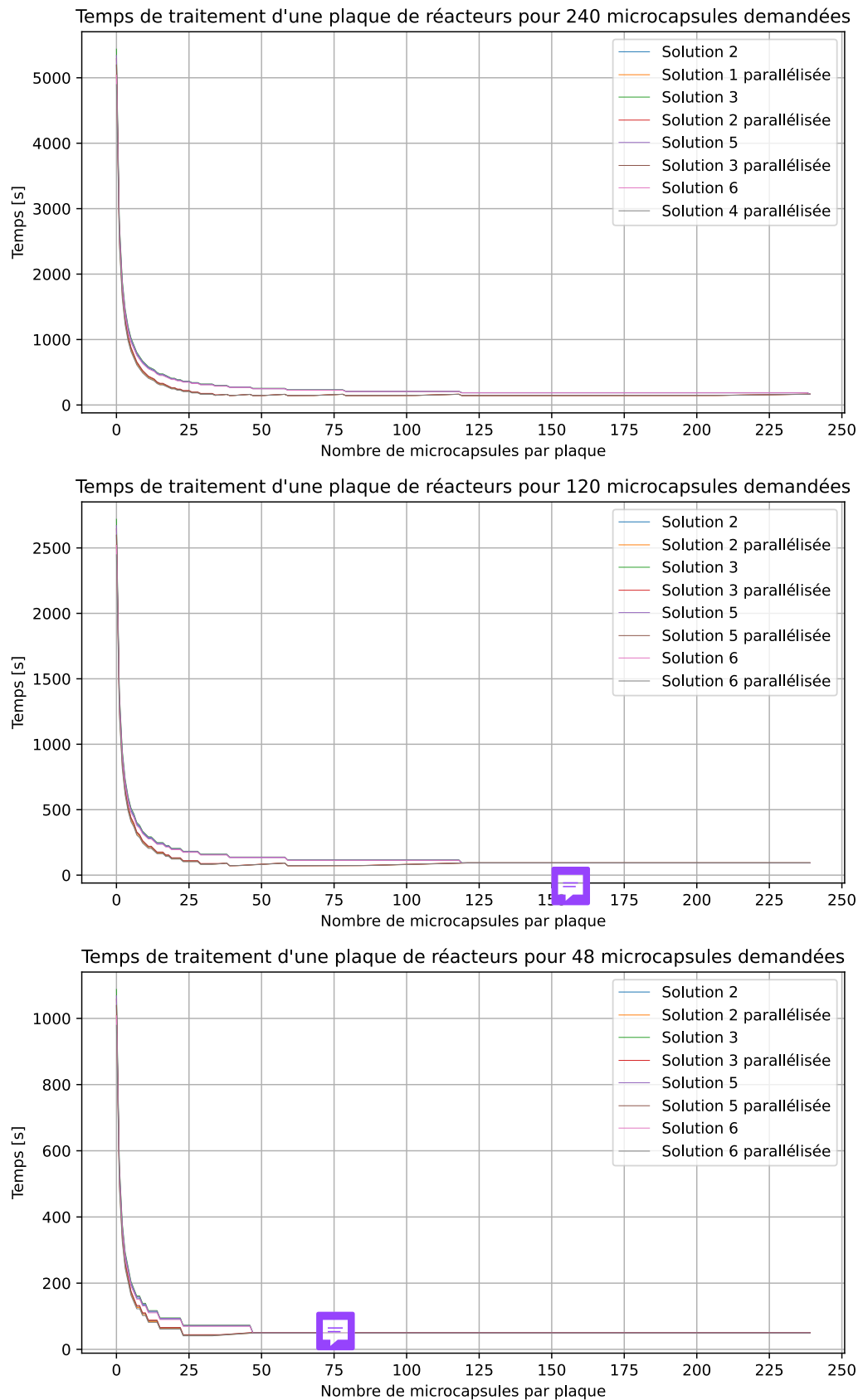


Figure 3.6 – Comparaison du temps de cycle des solutions en fonction du nombre de microcapsules par plaque

3.3. PRÉSENTATION DE LA SOLUTION CHOISIE

Sur la Figure 3.6, il est possible de voir que les solutions parallélisées ont toujours un temps de cycle inférieur aux solutions séquentielles tant que le nombre de microcapsules par plaque est inférieur au nombre de microcapsules demandées. Plus le nombre de microcapsule demandées est élevé, plus le gain de temps est significatif, allant, en moyenne, de 1.3 % pour 48 microcapsules à 27 % pour 240 microcapsules.

De part ces résultats (cf. Figure 3.5 et Figure 3.6), la solution numéro 6, c'est à dire, le déplacement de la *wellplate* et de la *paradox* côte à côte vers le stock est la solution la plus rapide.

3.3 Présentation de la solution choisie

L'utilisation d'un seul robot positionné au centre de la *glove box*, qui s'occupera de déplacer les plaques ainsi que de faire le *pick and place* des microcapsules a été retenue. Cette décision est due à :

- un robot déjà présent dans le laboratoire ;
- la grande flexibilité de cette solution (la *glove box* sera utilisée pour d'autres processus par la suite).

3.4 Description du matériel utilisé

Voici la liste de matériel utilisé lors du TB :

- Robot « UR3e » ;
- Pince « Hand-E » ;
- Adaptation de la pince précédente sur mesure ;
- Module d'aspiration ;
- Support pour les plaques.

3.5 Intégration

Afin de maximiser l'espace accessible par le robot, ce dernier a été surélevé (cf. Figure 3.7). La moitié gauche est réservée pour le *pick and place* des microcapsules, tandis que la partie à droite servira pour la suite du processus. Pour les outils, la pince se met sur la bride du robot et elle possède deux trous taraudés qui seront utilisés pour fixer le *module d'aspiration* (cf. Figure 3.8). L'outil pour saisir les plaques fait un angle à 45° pour pouvoir déposer les plaques sur tout le plan *repPickAndPlace*. Afin de gagner de la place, les deux outils sont placés dans des directions opposées.



Figure 3.7 – *Intégration du robot dans la glove box*



Figure 3.8 – *Module d'aspiration*

Chapitre 4

Software

4.1 Algorithme de recombinaison

4.1.1 Spécification de l'algorithme

Objectif

L'algorithme a pour objectif de sélectionner les microcapsules, dans le stock, à utiliser pour chaque recette d'un batch, soit environ 300 recettes.

Contraintes

Les contraintes de l'algorithme sont les suivantes :

1. Maximiser le nombre de recettes réalisées.
 - Se situer entre dans la plage de tolérance de quantité pour chaque produit ;
 - respecter la quantité maximale de microcapsules de chaque réacteur.
2. Minimiser le nombre de microcapsules utilisées par réacteur.

Entrées

Il possède en entrée :

- les recettes du batch ;
- les microcapsules présentes dans le stock ;
- le nombre maximal de microcapsule par réacteur.

Sorties

Les sorties de l'algorithme incluent :

- Un tableau contenant pour chaque recette un liste avec les **ids** des capsules à utilisées avec le numéros de recettes sous la forme : $[[id_1, id_2, \dots, id_n], n^{\circ}recette_1), \dots, ([id_1, id_2, \dots, id_n], n^{\circ}recette_m)]$ avec m le nombre de recette ;
- Un tableau contenant la quantité de chaque produit pour chaque recette sous la forme : $[("produit_1" : q_{produit_1}, \dots, "produit_n" : q_{produit_n}, n^{\circ}recette_1), \dots, ("produit_1" : q_{produit_1}, \dots, "produit_n" : q_{produit_n}, n^{\circ}recette_m)]$
- Le numéro des recettes qui ne sont pas réalisable
- Les éléments manquants pour réaliser les recettes réalisable avec la forme : $[("Produit_{manquant\ 1}" : q_{produitmanquant1}, \dots, "Produit_{manquant\ n}" : q_{produitmanquantn}, n^{recipe1}), \dots, ("Produit_{manquant\ 1}" : q_{produitmanquant1}, \dots, "Produit_{manquant\ n}" : q_{produitmanquantn}, n^{recipem})]$

- Les éléments manquants pour réaliser les recettes non réalisable avec la forme :
 $(\text{"Produit}_{\text{manquant } 1"} : q_{\text{produitmanquant}1}, \dots, \text{"Produit}_{\text{manquant } n"} : q_{\text{produitmanquant}n}, n^{\text{recipe1}}),$
 $\dots, (\text{"Produit}_{\text{manquant } 1"} : q_{\text{produitmanquant}1}, \dots, \text{"Produit}_{\text{manquant } n"} : q_{\text{produitmanquant}n}, n^{\text{recipe}n})$

4.1.2 Définition du problème

Le problème consiste à trouver une combinaison de microcapsules pour chaque recette qui maximise le nombre de recettes réalisées.

$$\max \left(\sum_i \text{RecetteRealisée}_i \right) \quad (4.1)$$

Maximisation du nombre de recettes réalisées

Avec :

- N , le nombre de microcapsules dans le stockage, $N \in \mathbb{N}^*$;
- k , le nombre moyen de microcapsules par recette, $k \in \mathbb{N}^*$;
- R , le nombre de recette réalisable, $R \in \mathbb{N}^*$.

Le nombre théorique de recettes réalisable est :

$$R_{th} = \left\lfloor \frac{N}{k} \right\rfloor \quad (4.2)$$

L'équation (cf. Equation 4.1), implique une maximisation du nombre théorique de recettes réalisables (cf. Equation 4.3).

$$\max \left(\sum_i \text{RecetteRealisée}_i \right) \Rightarrow \max (R_{th}) \quad (4.3)$$

$$\max (R_{th}) = \max \left(\left\lfloor \frac{N}{k} \right\rfloor \right) \Rightarrow \max (N) \vee \min (k) \quad (4.4)$$

Or, N est constant, donc :

$$\max \left(\sum_i \text{RecetteRealisée}_i \right) \Rightarrow \min (k) \quad (4.5)$$

À des fins de faciliter, l'utilisation de la minimisation de k (cf. Equation 4.5) sera préférée.

Contraintes

La liste des contraintes pour l'optimisateur sont les suivantes :

- La quantité dans chaque réacteur doit être comprises dans la plage souhaitée ;
- le nombre de microcapsules dans un réacteur doit être inférieur ou égal à la quantité maximal de microcapsules dans le réacteur ;
- une microcapsules ne peut être utilisée plusieurs fois.

Knapsack problem

« The knapsack problem (KP) can be formally defined as follows :
 We are given an instance of the knapsack problem with item set N , consisting of n items j with profit P_j and weight W_j , and the capacity value c . (Usually, all these values are taken from the positive integer numbers.) Then the objective is to select a subset of N such that the total profit of the selected items is maximized and the total weight does not exceed c . » (KELLERER HANS 2004, p. 2)

4.1. ALGORITHME DE RECOMBINAISON

Le problème posé ressemble au *knapsack problem*, cependant étant donné qu'il y a plusieurs réacteurs (l'équivalent du sac) le problème est donc plutôt un *Multiple knapsack problem*¹. Il y a encore une nuance entre le problème posé et un *Multiple knapsack problem*, c'est que dans un réacteur, il peut y avoir plusieurs produits dans un seul réacteur. Donc pour chaque batch, il y a plusieurs problème du type *Multiple knapsack problem*.

4.1.3 Méthode d'optimisation

Optimisation générale

L'optimisation générale consiste à traiter chaque produit séparément avec certaines contraintes (cf. Figure 4.1).

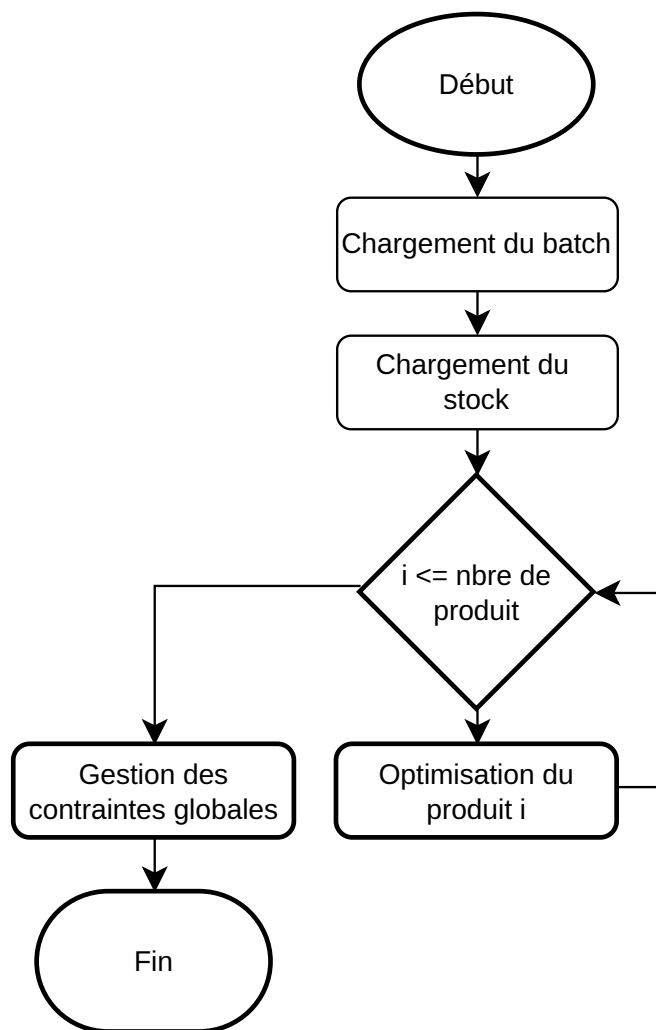


Figure 4.1 – Algorithme général de l'optimisateur.

Optimisateur

L'approche la plus intuitive pour optimiser le problème consiste à calculer toutes les combinaisons possibles puis de sélectionner la solution qui répond le mieux aux critères définis.

1. (KELLERER HANS 2004, p. 285)

Le nombre de combinaisons C possibles pour k microcapsules et un stock n , se calcul comme suit :

$$C_{k,n} = \frac{n!}{k! \cdot (n-k)!} \quad (4.6)$$

$$C_n = \sum_{k=1}^l C_{k,n} = \sum_{k=1}^l \frac{n!}{k! \cdot (n-k)!} \quad (4.7)$$

Nombre de recombinaison des microcapsules (échelle semi-logarithmique)

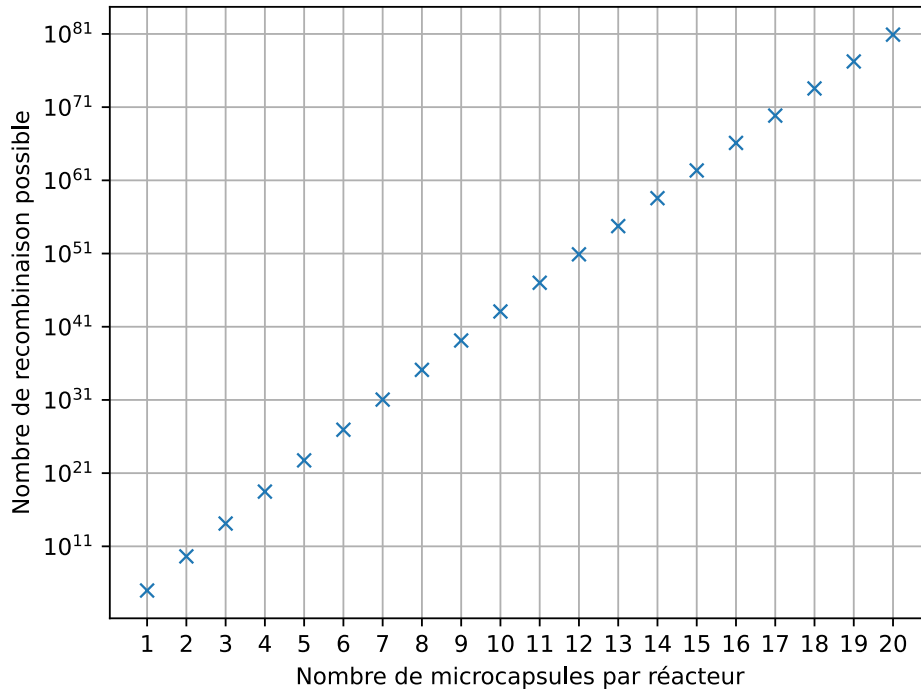
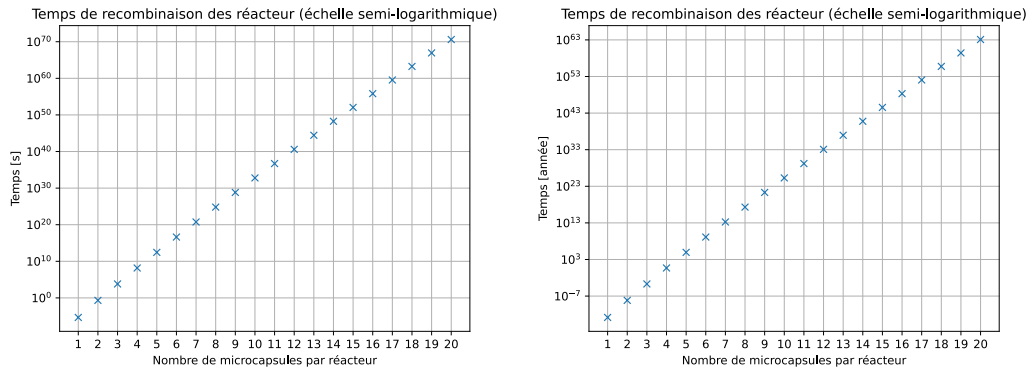


Figure 4.2 – Nombre de combinaisons possible en fonction de la taille des réacteurs



(a) Temps d'exécution en seconde

(b) Temps d'exécution en année

Figure 4.3 – Temps de calcul en fonction de la taille des réacteurs

4.1. ALGORITHME DE RECOMBINAISON

La Figure 4.3 montre le temps nécessaire pour calculer toutes les combinaisons possibles (en prenant en compte le nombre de combinaisons possible (cf. Figure 4.2) et le nombre des calcul par secondes moyen pour les ordinateurs en 2020 environ 10^{11} 2 opérations par secondes). Il est possible d'observer que le temps nécessaire pour les combinaisons dépassant x microcapsules maximal par réacteur, devient des durées non concevable pour cette application. Cette approche, bien qu'elle trouve toujours la solution optimale et qu'elle soit facilement compréhensible, n'est pas adapté au projet.

Plusieurs algorithmes existes pour résoudre ce type de problème :

- glouton ;
- programmation dynamique ;
- optimisation linéaire ;
- heuristique ;
- *branch and cut* ;
- optimisation linéaire en nombres entiers ;
- un algorithme génétique.

Le problème peut être interprété comme un problème contraint d'entier (*Constraints Integer Problems* (CIPs)), car la sélection des microcapsules se fait de manière binaire (une microcapsule est sélectionner ou non). Pour les CIPs, il existe des frameworks (notamment SCIP (*Solver Constraint Integer Programs*)) utilisant certains des algorithmes citées précédemment Figure 4.1.3.

L'utilisation de SCIP se fait avec la forme de l'optimisation avec contraintes :

$$\begin{array}{ll} \min & x \\ \text{subject to} & \sum_i (a_i x_i) \leq b \\ \text{and} & x \in \mathbb{N} \end{array}$$

ou plus généralement :

$$\begin{array}{ll} \min & x \\ \text{subject to} & Ax \leq b \\ \text{and} & x \in \mathbb{N} \end{array}$$

Avec b un vecteur et A une matrice. Pour utiliser l'optimisateur, il faut définir la fonction de coût (cf. section 4.1.3) à minimiser, et les contraintes (cf. section 4.1.3).

Matrice de décision

L'optimisateur doit utilisé un vecteur de décision pour optimiser l'utilisation des microcapsules. Pour une recette le vecteur \vec{x} est définie :

$$x_i \in \{0, 1\}, \forall i \in \{1, 2, \dots, n\} \text{ avec } n \text{ le nombre de microcapsules en stock. (4.8)}$$

Ce vecteur (Equation 4.8) est valable pour 1 seul recette. Idéalement pour plusieurs recettes, il faudrait une matrice X (cf. Equation 4.9), qui est par la suite « vectorisée » afin d'obtenir le vecteur de décision final (cf. Equation 4.10).

$$X = \begin{bmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,n} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,0} & x_{m,1} & \cdots & x_{m,n} \end{bmatrix} \quad (4.9)$$

$$\vec{x} = [x_{0,0}, \dots, x_{0,n}, x_{1,0}, \dots, x_{1,n}, x_{m,0}, \dots, x_{m,n}], \quad (4.10)$$

avec n le nombre de microcapsules, et m le nombre de recettes.

Matrices de contraintes

Pour définir les deux matrices de contraintes (A et b), il faut commencer par définir b car la structure de A en dépendra. b est un vecteur colonne qui comprends pour chaque recette :

1. La quantité maximal souhaitée (noté Q_{max} suivi du numéro de recette).
2. La quantité minimale souhaitée (noté Q_{min} suivi du numéro de recette).
3. Le nombre maximal de microcapsule par réacteur (noté l).

Puis, s'en suit une colonne de n ligne de 1, correspondant aux nombre de fois qu'une microcapsule peut être utilisé.

$$b = \begin{bmatrix} Q_{max}1 \\ -Q_{min}1 \\ l \\ \vdots \\ Q_{max}n \\ -Q_{min}n \\ l \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad (4.11)$$

En sachant que $Ax \leq b$, il est possible d'en déduire que A sera décomposée en sous matrices (des matrices identité I_n et une autre matrice nommées m_1).

$$A = \begin{bmatrix} m_1 & 0 & \dots & 0 \\ 0 & m_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & m_1 \\ I_n & I_n & \dots & I_n \end{bmatrix} \quad (4.12)$$

Pour la composition de m_1 , la première ligne sert à déterminé si la somme de la quantité des capsules sélectionner est inférieur à la quantité limite, la deuxième à déterminé si la somme de quantité des microcapsules sélectionnées est supérieur à la quantité minimale et la dernière ligne est présente pour verifier que la somme des microcapsules sélectionner ne dépasse pas la limite de microcapsule maximum par réacteur.

$$m_1 = \begin{bmatrix} Q_1 & Q_2 & \dots & Q_n \\ -Q_1 & -Q_2 & \dots & -Q_n \\ 1 & 1 & \dots & 1 \end{bmatrix} \quad (4.13)$$

4.2. ROBOT

Fonction de coût

L'objectif de l'optimisation est de réduire le nombre de microcapsules utilisé, avec x le vecteur de décision, le fonction de coût $f(\vec{x})$ est donc :

$$f(\vec{x}) = \sum \vec{x} \quad (4.14)$$

Cependant en utilisant la fonction (cf. Equation 4.14), si une seule des recettes du batch n'est pas réalisable, l'optimisateur retournera le fait que le problème n'est résoluble, sans fournir les résultats des recettes dont il a trouvé des solutions. Pour résoudre ce problème, l'ajout d'une *slack variable*, nommé z est indispensable. Cette variable prendra la quantité manquante de certaines recettes non réalisable. Afin de ne pas tomber dans l'utilisation excessive de \vec{z} , il est nécessaire de rendre le coût de celle-ci plus importante grâce à un ratio α . La fonction de coût définitive devient :

$$f(\vec{x}) = \sum (\vec{x} + \alpha \vec{z}) \quad (4.15)$$

Avec α définit arbitrairement à 10^4 .

4.2 Robot

4.2.1 Objectif

Le robot doit manipuler les microcapsules en effectuant des tâches de *pick and place* depuis leur zone de stockage jusqu'aux réacteurs. Il doit également être capable de manipuler des plaques de microcapsules et de « Para-dox ».

4.2.2 Contraintes

1. Respect des limites de la *glove box* pour éviter toute collision ;
2. Gestion de deux outils (pince et module d'aspiration) sans changement manuel ;

4.2.3 Méthode de programmation

La programmation est effectuée en deux parties :

- Les programmes intégrés au robot gèrent les déplacements : « PickAndPlaceVial », « TakePlate », et « GivePlate » ;
- Un programme externe en Python appelle les fonctions du robot et transmet les informations nécessaires via l'interface RTDE (cf. subsection 4.2.7).

4.2.4 Outils

Les outils nécessaires pour réaliser la manipulation sont :

- une pince pour saisir les *well plate* et les plaques « Para-dox » ;
- un module pour l'aspiration des microcapsules.

Une pointe a également été fabriquée, pour faciliter la programmation des différents repères nécessaires. [Ajout de détaille sur le matériel](#)

4.2.5 Sécurité

Limites de sécurité

Les robots d'Universal Robot permettent de configurer des plans de sécurité, qui ne seront pas franchissables par le TCP du robot. Dans ce cas, deux limites des sécurités sont nécessaires (une limite par porte de la *glove box*). Pour définir ces limites, il faut :

1. Définir un point en mettant l'outil perpendiculaire au plan souhaité.
2. Définir un plan de sécurité qui passe par ce point et qui sera normal à l'outil du robot.
3. Définir la sécurité à mettre en place (Normal, réduit, les deux ou mode de déclenchement réduit). Dans notre cas, nous ne voulons pas que le TCP dépasse ces plans, le mode « les deux » doit donc être appliqué.

4.2.6 Repères de travail

Deux repères seront utilisés :

- le repère de base qui sera utilisé pour déplacer les plaques ;
- le repère « repPickAndPlace » (cf. Figure 4.4), sera utilisé pour le déplacement des microcapsules.



Figure 4.4 – Position physique du plan « repPickAndPlace »

4.2.7 Communication avec le robot

L'interface *Real-Time Data Exchange* (RTDE) permet l'échange bidirectionnel en temps réel des données entre le robot et le système externe, facilitant ainsi le démarrage des programmes robot et la lecture/écriture des registres nécessaires.

Chapitre 5

Implémentation et intégration Hardware-Software

5.1 Développement du logiciel

5.1.1 Objets

Afin de faciliter l'améliorabilité et la clareté du code, une programmation orientée objet à été choisi (Diagramme de classe Figure 5.1)

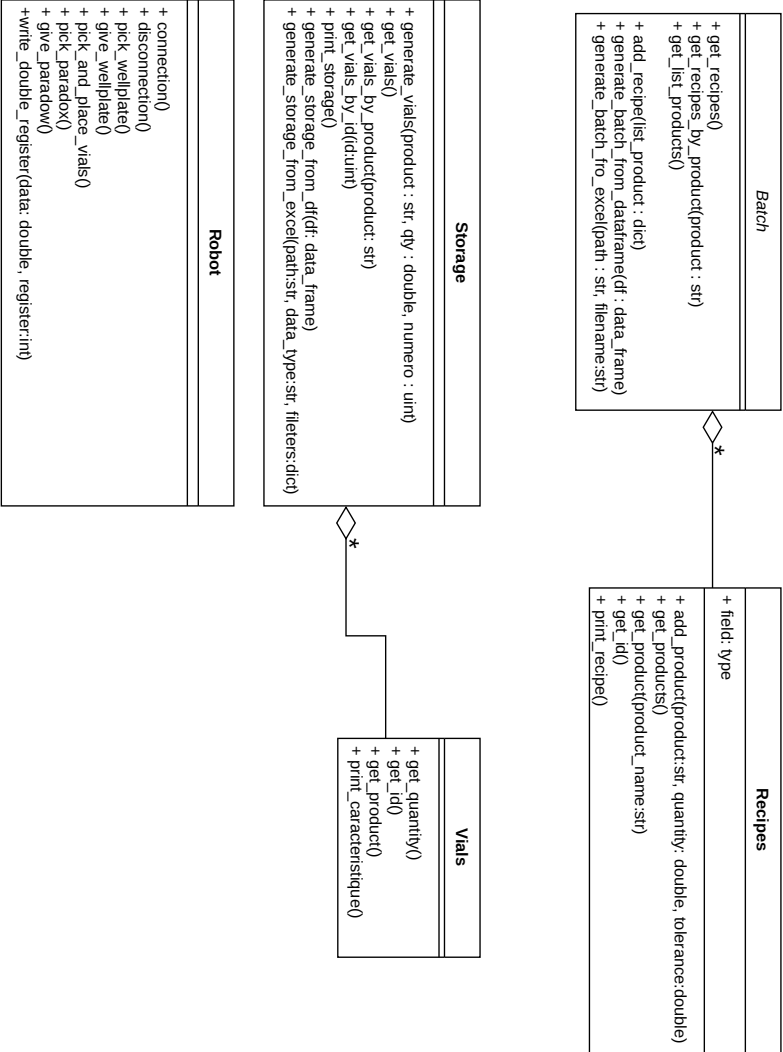


Figure 5.1 – Diagramme de classe

5.1.2 Optimisateur

Pour l'optimisateur, celui-ci suit le diagramme de flux (cf. Figure 5.2) avec a la suite une traitement des données récoltées. L'utilisation de SCIP¹ (qui permettra de ne pas implémenter chaque algorithme indépendamment) à travers le wrapper or-tools² sera utilisé afin de faciliter l'intégration en python.

5.2 Tests de validation et calibrage du matériel

5.2.1 Tests unitaires

Les tests unitaires des classes et de l'optimisateur se font avec les modules « py-test » ([Ajout de référence code](#)) et « Coverage ». Ce dernier permet de connaître le pourcentage de lignes de code tester (94% dans ce cas (cf. Figure 5.3)).

File ▲	statements	missing	excluded	coverage
apps/shared/models/batch.py	37	0	0	100%
apps/shared/models/read_write_data.py	42	6	0	86%
apps/shared/models/recipe.py	23	0	0	100%
apps/shared/models/solver.py	176	12	0	93%
apps/shared/models/storage.py	30	0	0	100%
apps/shared/models/vials.py	17	0	0	100%
Total	325	18	0	94%

Figure 5.3 – Couverture du code par les tests unitaires

Pour le robot, les tests se sont fait en lançant les programmes et en vérifiant visuellement que le comportement soit correct.

1. ZUSE-INSTITUTE-BERLIN 2005.

2. GOOGLEAI s. d.

5.2. TESTS DE VALIDATION ET CALIBRAGE DU MATÉRIEL

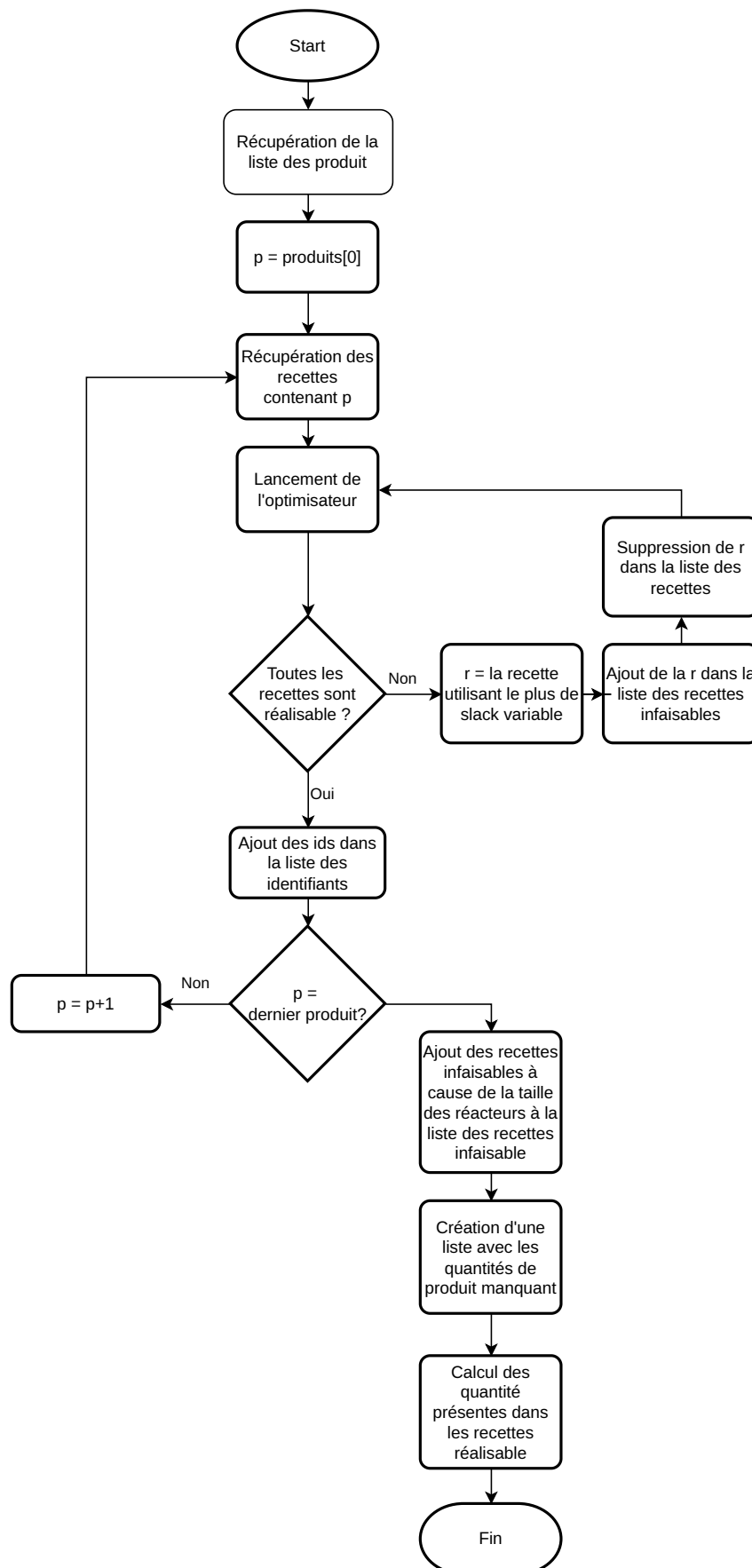


Figure 5.2 – Diagramme de flux de l'optimisateur implémenté

Chapitre 6

Sécurité et Préventions des Risques

La sécurité est aspect important de ce projet, en raison de l'utilisation d'un robot UR3 dans un espace confiné pour manipuler des composants chimiques. Cette section détaille les mesures mises en oeuvre ou à mettre en oeuvre pour assurer la sécurité des opérateurs, du matériel et des échantillons.

6.1 Identification des risques

Pour faire cette analyse, une approche ascendante¹ avec une matrice de risques (cf. Table 6.1) sont utilisées. Pour la matrice de risque, la criticité du risque est le produit de la gravité et de l'occurrence de ce risque. Cette criticité servira à prioriser l'ordre de réduction des risques (plus la criticité est élevée, plus le risque est important et doit être réduit).

1. L'approche ascendante consiste à partir de la base (qu'est-ce qui peut être dangereux) puis remonter pour en trouver les effets. Inversement à l'approche descendante qui cherche les causes à partir d'un accident

Id	Description	Impact	Gravité [1-5]	Occurence [1-5]	Criticité [1-25]
1	Déchirement des gants par le robot	Perte de l'étanchéité	5	1	5
2	microcapsulecapsule non saisie	Erreur lors des analyses	1	2	2
3	microcapsulecapsule non déposée	Erreur lors des analyses Détérioration des prochaines microcapsules	3	2	6
4	Perte de microcapsule lors du déplacement	Erreur lors des analyses Contamination de la <i>glove box</i>	4	3	12
5	Renversement d'une wellplate	Perte de la totalité des capsules de la wellplate Contamination de la <i>glove box</i>	5	2	10
6	Renversement d'une paradox	Perte de la totalité des capsules présentes dans la paradox Contamination de la <i>glove box</i>	5	3	15
7	Collision entre les gants et le robot	Contusions	5	1	5

Table 6.1 – Analyses des risques

6.2. SOLUTIONS

6.2 Solutions

Cette section a pour but de donner des solutions envisageables aux risques déterminés précédemment (cf. section 6.1). Les solutions trouvées doivent impacter, la gravité, le nombre d'occurrence ou les deux. Par exemple, en prenant le risque à traiter en priorité (le numéro 6 qui possède une criticité de 15 (cf. Table 6.1)), il est possible de modifier la paradox afin d'y ajouter des ergots et ainsi améliorer le maintien de celle-ci dans la pince du robot, réduisant le nombre d'occurrence de ce risque, mais il est également possible de rajouter un couvercle par dessus la paradox afin de ne pas éparpiller le contenu dans la *glove box*, réduisant ainsi la gravité du risque. Toutes les solutions proposées n'ont pas forcément été appliquées lors du travail de Bachelor.

Id du risque	Solutions proposées	Action[Gravité, Occurrence ou les deux]
6	Améliorer l'ergonomie de la paradox par rapport à la pince. Placer un couvercle sur la paradox lors des déplacements.	Occurrence Gravité
4	Augmenter la pression pour l'aspiration. Mettre un bac récupérateur sous la pince.	Occurrence Gravité
5	Ajout de trous sur la <i>wellplate</i> pour améliorer la prise de la pince.	Occurrence
3	Ajouter un système de soufflerie lors de la dépose.	Occurrence
7	Mettre des gants plus épais.	Gravité
1	Arrondir tous les angles présents sur la pince.	Occurrence
2	Augmenter la pression pour l'aspiration.	Occurrence

Table 6.2 – Solutions face aux risques (dans l'ordre de priorité)

Chapitre 7

Résultat et Analyse

7.1 Performance du système

7.2 Analyse des erreurs

Chapitre 8

Discussion

8.1 *Points forts*

La réalisation de ce travail de Bachelor a permis de concevoir et d'implémenter un système robotisé pour la manipulation de microcapsules dans un environnement confiné. Bien que les objectifs initiaux aient été globalement atteints, certaines limitations et axes d'amélioration méritent d'être discutés.

- **Précision et efficacité** : Le système robotisé mis en place s'est révélé capable de manipuler les *wellplates* avec une précision satisfaisante. L'intégration du robot avec les outils développés a permis d'automatiser efficacement le transfert des capsules entre les *wellplates* et les réacteurs.
- **Flexibilité du système** : La configuration retenue, avec un robot positionné au centre de la *glove box*, offre une grande modularité, permettant de l'adapter à d'autres processus futures au laboratoire SwissCar+.
- **Optimisation logicielle** : L'algorithme de recombinaison a démontré sa capacité à maximiser le nombre de recettes réalisables, grâce à une approche rigoureuse basée sur des techniques d'optimisation contraintes.

8.2 *Limitations*

- **Prise des microcapsules** : Le système pneumatique pour saisir les microcapsules n'as pas pu être développé par manque de temps.
- **Problème de manipulation** : Certains risques liés à la manipulation, comme le renversement des *wellplates*, des paradoxs ou la perte de microcapsules lors du transport, ont été identifiés. Des solutions ont été proposées (cf. Table 6.2), leur implementation reste partielle voir inexistante et nécessiterait une validation plus approfondie.
- **Temps de traitement** : Bien que l'algorithme permette une recombinaison efficace des microcapsules, lors de problèmes trop complexe, la recombinaison pourrait devenir un goulot d'étranglement. Une optimisation supplémentaire de l'algorithme, notamment en utilisant une méthode heuristique plus avancée pourrait améliorer les performances.

8.3 *Perspectives*

- **Amélioration matérielle** : L'amélioration de la pince pour pouvoir saisir la paradox (un entraxe plus grand suffirait).

- **Développement du réseau pneumatique :** Le développement d'un réseau pneumatique permettant au robot de faire le plus de déplacement possible est nécessaire.
- **Validation expérimentale :** Des tests à plus grandes échelle avec des lot de microcapsules et des recettes variées permettront d'évaluer plus précisément les performances du systèmes en conditions réelles.
- **Algorithme et intelligence artificielle :** L'intégration d'un algorithme d'apprentissage pour sélectionner les microcapsules permettrait de diminuer le temps de traitement de l'algorithme.

Chapitre 9

Conclusion

Ce projet de Bachelor a permis de développer un système robotisé innovant pour la recombinaison des microcapsules chimiques dans un laboratoire de recherche. Les contributions principales incluent la conception et l'intégration d'un robot dans une *glove box*, et la création d'un algorithme d'optimisation permettant de maximiser le nombre de recette réalisable.

L'ensemble du système répond aux exigences initiales initiales en matières de précision, de fiabilité et de sécurité, tout en offrant une grande flexibilité pour des applications futures. Les résultats obtenus démontrent que l'automatisation de la recombinaison peut considérablement accélérer le processus expérimental, réduisant les erreurs humaines et augmentant l'efficacité global.

Cependant certains points, comme le réseau pneumatique, n'ont pas pu être réalisée à cause d'un manque de temps.

Des axes d'amélioration existent. Le temps de calcul de l'algorithme, la robustesse des outils et la gestion des risques liés à la manipulation sont des aspects à améliorer. L'ajout d'un module d'intelligence artificielle et l'amélioration des outils physique pourrait augmenter le rendement de la recombinaison.

En conclusion, ce travail de représente une avancée vers l'automatisation complète des tâches dans le laboratoire.

Gaëtan Worch

Bibliographie

- KELLERER HANS PFERSCHY Ulrich, PISINGER David (2004). *Knapsack Problems*. Springer. ISBN : 978-3-642-07311-3 (cf. p. 16, 17).
- ZUSE-INSTITUTE-BERLIN (2005). *SCIP, Solving Constraint Integer Programs*. URL : <https://scipopt.org/>. (accessed : 06.02.2025) (cf. p. 24).
- C., Johan (2023). *Lvolution de la puissance de calcul en informatique*. URL : <https://KnapsackProblemsBooklespetitesanalyses.com/2023/10/14/levolution-de-la-puissance-de-calcul-en-informatique/>. (accessed : 05.02.2025) (cf. p. 19).
- GOOGLEAI (s. d.). *Interface MPSolver*. URL : https://developers.google.com/optimization/lp/lp_example?hl=fr. (accessed : 06.02.2025) (cf. p. 24).

Annexes

Annexe A



Optimisateur

```

1 from ortools.linear_solver import pywraplp
2 import numpy as np
3 from ..models import batch as b, storage as s
4
5 def split_array(arr, chunk_size) :
6     """
7     Splits a list `arr` into smaller sublists (chunks) of a specified
8     ↪ size (`chunk_size`).
9
10    Parameters:
11        arr (list): List of elements to be split.
12        chunk_size (int): The size of each chunk (sublist).
13
14    Returns:
15        list: A list of sublists, each containing at most `chunk_size`
16        ↪ elements from the original list.
17    """
18    return [arr[i:i + chunk_size] for i in range(0, len(arr),
19    ↪ chunk_size)]
20
21 def solve(batch, storage, reactor_capacity) :
22     """
23     Solves the optimization problem related to vial assignment, using
24     ↪ a decision matrix
25     and considering constraints on product quantities.
26
27    Parameters:
28        batch (object): An object representing a batch with products
29        ↪ and recipes.
30        storage (object): An object representing the storage
31        ↪ containing vials.
32
33    Returns:
34        tuple: A tuple containing:
35        - list_id (list): A list of the decision matrix.
36        - quantities (list): A list of quantities calculated based
37        ↪ on the decision matrix.
38        - id_unfeasible_recipe (list): The IDs of recipes that
39        ↪ could not be completed.
40        - missing_elements (list): A list of missing products/
41        ↪ quantities for unfeasible recipes.

```

Listing A.1 – Génération d'un diagramme de Bode


```

1      """
2      list_id = []
3      list_product = batch.get_list_products()
4      list_recipes_no_solution = []
5      all_recipes = batch.get_recipes()
6      decision_matrix = np.zeros([len(all_recipes), len(storage.
7          ↪ get_vials())])
8      nbre_product = len(list_product)
9      for i in range(nbre_product) :
10         product = list_product[i]
11         recipes = batch.get_recipes_by_product(product)
12         while(True) :
13             recipes = [recipe for recipe in recipes if recipe not in
14                 ↪ list_recipes_no_solution]
15             decision_matrix_solver, slack_variable, status =
16                 ↪ find_vials_matrix_test(recipes, storage.
17                 ↪ get_vials_by_product(product),
18                 ↪ nbre_vial_maxium_per_reactor=reactor_capacity)
19             if sum(slack_variable) == 0 :
20                 break
21             index_max = np.argmax(slack_variable)
22             list_recipes_no_solution.append(recipes[index_max])
23             cleaned_matrix = clean_matrix(decision_matrix_solver)
24             list_id.append(cleaned_matrix)
25
26             decision_matrix = update_matrix_decision(decision_matrix,
27                 ↪ cleaned_matrix, recipes)
28         recipe_with_too_much_vials, decision_matrix =
29             ↪ verifying_number_of_vial_by_recipe(decision_matrix,
30             ↪ reactor_capacity)
31         list_recipe_too_much_microcaps = [(all_recipes[i].get_id(),
32             ↪ all_recipes[i].get_products()) for i in range(len(
33             ↪ all_recipes)) if all_recipes[i].get_id() in
34             ↪ recipe_with_too_much_vials]
35
36         list_unfeasible_recipes = fusion_array(
37             ↪ list_recipe_too_much_microcaps, list_recipes_no_solution)
38         id_unfeasible_recipe, missing_elements =
39             ↪ creation_list_missing_elements(list_unfeasible_recipes)
40         list_id = creation_list_id(decision_matrix)
41         quantities = calcul_quantities(storage, list_id,
42             ↪ list_unfeasible_recipes)
43         return list_id, quantities, id_unfeasible_recipe, missing_elements
44
45 def update_matrix_decision(A, list_ids, list_recipes) :
46     """
47     Updates a decision matrix `A` based on selected IDs from `list_ids
48     ↪ ` and corresponding recipes in `list_recipes`.
49
50     Parameters:
51
52     A (ndarray): The decision matrix to be updated.
53     list_ids (list): A list of selected IDs.
54     list_recipes (list): A list of recipes.
55
56     Returns:
57
58     ndarray: The updated decision matrix `A`.
59     """
60     for index, recipe in enumerate(list_recipes):

```

Listing A.2 – Génération d'un diagramme de Bode

```

1         num_recipe = recipe[0]
2         for i in list_ids[index] :
3             A[num_recipe, i-1] = 1
4
5     return A
6
7 def create_block_diagonal(pattern, nbreRecipes, sizeStorage):
8     """
9     Creates a block diagonal matrix based on a given pattern,
10    ↪ repeating it for a specified number of recipes and storage
11    ↪ size.
12
13    Parameters:
14    pattern (ndarray): A 2D array representing the pattern to be
15    ↪ repeated.
16    nbreRecipes (int): The number of recipes (how many times the
17    ↪ pattern should be repeated).
18    sizeStorage (int): The size of the storage.
19
20    Returns:
21    ndarray: A 2D matrix where the given pattern is repeated
22    ↪ diagonally.
23    """
24    block_rows, block_cols = pattern.shape
25
26    rows = nbreRecipes * block_rows
27    cols = nbreRecipes * sizeStorage
28
29    result = np.zeros((rows, cols))
30
31    for i in range(nbreRecipes):
32        start_row = i * block_rows
33        start_col = i * sizeStorage
34        result[start_row:start_row + block_rows, start_col:start_col +
35        ↪ block_cols] = pattern
36
37    return result
38
39 def find_vials_matrix_test(recipes, stock,
40 ↪ nbre_vial_maxium_per_reactor):
41     """
42     Solves an optimization problem using the Google OR-Tools solver to
43     ↪ assign vials to recipes while considering the constraints.
44
45     Parameters:
46     recipes (list): A list of recipes with product quantities.
47     stock (list): The list of available vials in storage.
48     nbre_vial_maxium_per_reactor (int): The maximum number of
49     ↪ vials allowed per reactor.
50
51     Returns:
52     tuple: A tuple containing:
53     - split_soluce_ids (list): Solution IDs for the vials
54     ↪ assigned to the recipes.
55     - z_sum (list): A list of slack variables (unused
56     ↪ quantities).
57     - status (int): The status of the solver (optimal,
58     ↪ infeasible, etc.).
59     """
60     nbre_recipes = len(recipes)

```

Listing A.3 – Génération d'un diagramme de Bode

```

1 size_storage = len(stock)
2 m1 = np.zeros([2, size_storage])
3 for index, vial in enumerate(stock) :
4     m1[0, index] = vial.get_quantity()
5     m1[1, index] = -vial.get_quantity()
6
7 I = np.identity(size_storage)
8 nbre_rep_I_block = nbre_recipes
9 I_block = np.tile(I, (1, nbre_rep_I_block))
10 qty_bounds = np.zeros([0, 0])
11 for index, recipe in enumerate(recipes) :
12     product = recipe[1]
13     y = np.array([[product['quantity_max'], -product['quantity_min']
14     ↪ '']]])
15     qty_bounds = np.append(qty_bounds, y)
16 A = np.block([[m1 if i == j else np.zeros_like(m1) for i in range(
17     ↪ nbre_recipes)] for j in range(nbre_recipes)])
18
19 b = np.ones([1, nbre_recipes + size_storage])
20 b[:, 0:nbre_recipes] = nbre_vial_maxium_per_reactor
21
22 B = np.zeros([nbre_recipes, nbre_recipes*size_storage])
23
24 for i in range(nbre_recipes) :
25     B[i, i*size_storage:i*size_storage+size_storage] = 1
26
27 B = np.append(B, I_block, axis = 0)
28
29 pattern = np.array([[1 for _ in range(size_storage)], [-1 for _ in
30     ↪ range(size_storage)]])
31 C = create_block_diagonal(pattern, nbre_recipes, size_storage)
32 solver = pywraplp.Solver.CreateSolver('SCIP')
33 x = [solver.IntVar(0, 1, f"x{i}") for i in range(nbre_recipes*
34     ↪ size_storage)]
35 y = [solver.NumVar(-solver.infinity(), solver.infinity(), f"y{i}")
36     ↪ for i in range(nbre_recipes*size_storage)]
37 for i in range(len(qty_bounds)) :
38     solver.Add(solver.Sum(A[i, j] * x[j] + C[i, j] * y[j] for j in
39     ↪ range(len(x))) <= qty_bounds[i])
40 for i in range(B.shape[0]) :
41     solver.Add(solver.Sum(B[i, j] * x[j] for j in range(len(x)))
42     ↪ <= b[0][i])
43
44 z = [solver.NumVar(0, solver.infinity(), f"z{i}") for i in range(
45     ↪ len(y))]
46 for i in range(len(y)):
47     solver.Add(z[i] >= y[i])
48     solver.Add(z[i] >= -y[i])
49 solver.SetTimeLimit(3600000)
50 solver.Minimize(solver.Sum(x) + 100*solver.Sum(z))
51 status = solver.Solve()
52 if status == pywraplp.Solver.OPTIMAL or status == pywraplp.Solver.
53     ↪ FEASIBLE:
54     soluce_ids = [int(x[i].solution_value() * stock[i%len(stock)].
55     ↪ get_id()) if x[i].solution_value() == 1 else -1 for i
56     ↪ in range(len(x))]
57     split_soluce_ids = split_array(soluce_ids, size_storage)
58     z_value = [z[i].solution_value() for i in range(len(z))]
59     split_slack_variable = split_array(z_value, size_storage)

```

Listing A.4 – Génération d'un diagramme de Bode

```

1      split_slack_variable = split_array(z_value, size_storage)
2      z_sum = [sum(split_slack_variable[i]) for i in range(len(
3          ↪ split_slack_variable))]
4      return split_soluce_ids, z_sum, status
5  elif status == pywraplp.Solver.INFEASIBLE:
6      print("Problème infaisable : aucune solution ne respecte
7          ↪ toutes les contraintes.")
8  elif status == pywraplp.Solver.UNBOUNDED:
9      print("Le problème est non borné.")
10 elif status == pywraplp.Solver.ABNORMAL:
11     print("Arrêt anormal du solveur.")
12 elif status == pywraplp.Solver.NOT_SOLVED:
13     print("Le problème n'a pas encore été résolu.")
14 else:
15     print("Statut inconnu.")
16 return []
17
18 def clean_matrix(matrix) :
19     """
20     Cleans the decision matrix by removing `-1` values (representing
21     ↪ unassigned vials).
22
23     Parameters:
24         matrix (list): The decision matrix to be cleaned.
25
26     Returns:
27         list: A cleaned decision matrix without `-1` values.
28     """
29     filtered_matrix = []
30     for i in range(len(matrix)) :
31         filtered_line = []
32         line = matrix[i]
33         for j in range(len(line)) :
34             if line[j] != -1 :
35                 filtered_line.append(line[j])
36
37         filtered_matrix.append(filtered_line)
38     return filtered_matrix
39
40 def verifying_number_of_vial_by_recipe(decision_matrix,
41     ↪ nbre_vial_maximum_per_reactor) :
42     """
43     Verifies that the number of vials per recipe does not exceed the
44     ↪ specified limit.
45
46     Parameters:
47         decision_matrix (numpy array): The decision matrix indicating
48             ↪ vial assignments to recipes.
49         nbre_vial_maximum_per_reactor (int): The maximum allowed
50             ↪ number of vials per reactor.
51
52     Returns:
53         tuple: A tuple containing:
54             - unfeasible_recipe (list): A list of recipes that
55                 ↪ exceeded the vial limit.
56             - decision_matrix (list): The updated decision matrix.
57     """
58     unfeasible_recipe = []
59     for i in range(decision_matrix.shape[0]) :

```

Listing A.5 – Génération d'un diagramme de Bode

```

1   for i in range(decision_matrix.shape[0]) :
2       if np.sum(decision_matrix[i]) > nbre_vial_maximum_per_reactor
3           ↪ :
4           decision_matrix[i] = 0
5           unfeasible_recipe.append(i)
6
7   return unfeasible_recipe, decision_matrix
8
9   def fusion_array(array_1, array_2) :
10       """
11       Merges two arrays, ensuring no duplicate entries based on the
12       ↪ first element (ID).
13
14       Parameters:
15         array_1 (list): The first array.
16         array_2 (list): The second array.
17
18       Returns:
19         list: A merged array without duplicate entries.
20       """
21       index_array = {entry[0] for entry in array_1}
22
23       for entry in array_2 :
24         if entry[0] not in index_array :
25           array_1.append(entry)
26
27       return array_1
28
29   def creation_list_id(decision_matrix) :
30       """
31       Creates a list of IDs from the decision matrix, mapping recipes to
32       ↪ their corresponding vials.
33
34       Parameters:
35         decision_matrix (list): The decision matrix indicating vial
36         ↪ assignments to recipes.
37
38       Returns:
39         list: A list of tuples, where each tuple contains a list of
40         ↪ vial IDs assigned to a recipe and the recipe's index.
41       """
42       array_id = []
43       for i in range(decision_matrix.shape[0]) :
44         array_id_by_recipes = []
45         for j in range(decision_matrix.shape[1]) :
46           if decision_matrix[i][j] == 1 :
47             array_id_by_recipes.append(j)
48
49         array_id.append((array_id_by_recipes, i))
50       return array_id
51
52   def fusion_dictionary_by_product(dict) :
53       """
54       Merges a dictionary of products and quantities into a new
55       ↪ structure, grouping quantities by product.
56
57       Parameters:
58         dict (list): A list of dictionaries containing products and
59         ↪ their quantities.

```

Listing A.6 – Génération d'un diagramme de Bode

```

1
2 Returns:
3     list: A list of dictionaries, each containing a product and
4         ↪ its corresponding list of quantities.
5     """
6     all_products = {}
7     for element in dict :
8         product = element['product']
9         quantity = element['quantity']
10        if product not in all_products :
11            all_products[product] = []
12            all_products[product].append(quantity)
13        fusionned_dictionnary = [{'product': product, 'quantities':
14            ↪ quantities} for product, quantities in all_products.items()
15            ↪ ]
16    return fusionned_dictionnary
17
18 def creation_list_missing_elements(list_unfeasible_recipes) :
19     """
20     Creates a list of missing elements (products and quantities) for
21     ↪ unfeasible recipes.
22
23     Parameters:
24         list_unfeasible_recipes (list): A list of unfeasible recipes.
25
26     Returns:
27         tuple: A tuple containing:
28             - index_unfeasible_recipe (list): A list of unfeasible
29             ↪ recipe IDs.
30             - missing_element (list): A list of missing product
31             ↪ quantities for each unfeasible recipe.
32     """
33     index_unfeasible_recipe = []
34     missing_element = []
35     for _, recipe in enumerate(list_unfeasible_recipes) :
36         index = recipe[0]
37         products = [recipe[1]] if isinstance(recipe[1], dict) else
38             ↪ recipe[1]
39         index_unfeasible_recipe.append(index)
40         for i in range(len(products)) :
41             missing_element.append({'product':products[i]['product'],
42                 ↪ 'quantity':products[i]['quantity_target']})
43
44     return index_unfeasible_recipe, missing_element
45
46 def calcul_quantities(storage, list_id, list_unfeasible_recipe) :
47     """
48     Calculates the quantities of products that are required based on
49     ↪ the storage and recipe data.
50
51     Parameters:
52         storage (object): The storage object containing vial data.
53         list_id (list): A list of IDs representing the decision matrix
54             ↪ .
55         list_unfeasible_recipe (list): A list of unfeasible recipes.
56
57     Returns:

```

Listing A.7 – Génération d'un diagramme de Bode

```

1      list: A list of calculated quantities for the unfeasible
2          ↪ recipes.
3      """
4      quantities = []
5      for i in list_id :
6          ids = i[0]
7          num_recip = i[1]
8          products = {}
9          for id in ids :
10             microcapsule = storage.get_vials_by_id(id)
11             product = microcapsule.get_product()
12             quantity = microcapsule.get_quantity()
13             if product not in products :
14                 products[product] = 0
15             products[product] += quantity
16             quantities.append((products, num_recip))
17
18     for unfeasible in list_unfeasible_recipe :
19         num_unfeasible_recipe = unfeasible[0]
20         products = [unfeasible[1]] if isinstance(unfeasible[1], dict)
21         ↪ else unfeasible[1]
22     for i, (product, idx) in enumerate(quantities) :
23         if idx == num_unfeasible_recipe :
24             for i in range(len(products)) :
25                 product[products[i]['product']] = 0
26     return quantities

```

Listing A.8 – Génération d'un diagramme de Bode

Annexe B

Tests unitaires

```

1  from apps.shared.models.batch import Batch
2
3  batch = Batch()
4
5  def test_batch_init() :
6      b = Batch()
7
8      assert b.recipes == []
9      assert b.list_product == []
10
11 def test_generate_batch() :
12     path = "tests/input/recipe/"
13     filename = "chemical_recipes_test.xlsx"
14     batch.generate_batch_from_excel(path, filename)
15
16     assert len(batch.recipes) == 1
17     product = batch.recipes[0].get_products()[0]
18     assert product["product"] == "Farine"
19     assert product["quantity_target"] == 51236
20     assert product["quantity_min"] == 46112.4
21     assert product["quantity_max"] == 56359.6
22
23 def test_get_recipes() :
24     products = batch.get_recipes()
25     product = products[0].get_products()
26     assert len(products) == 1
27     assert product[0]["product"] == "Farine"
28     assert product[0]["quantity_target"] == 51236
29     assert product[0]["quantity_min"] == 46112.4
30     assert product[0]["quantity_max"] == 56359.6
31
32 def test_get_recipes_by_product() :
33     products_farine = batch.get_recipes_by_product("Farine")
34     products_surce = batch.get_recipes_by_product("Sucre")
35
36     assert len(products_farine) == 1
37     assert len(products_surce) == 0
38
39 def test_get_list_product() :
40     products = batch.get_list_products()
41
42     assert products == ["Farine"]

```

Listing B.1 – Test unitaire de batch.py

```

1
2 def test_print(capsys) :
3     batch.print()
4     captured = capsys.readouterr()
5
6     assert captured.out == "Recipe\nř0\nProduct: Farine, Quantity
    ↳ target: 51236, Quantity_min: 46112.4, Quantity_max:
    ↳ 56359.6.\n"

```

Listing B.2 – *Test unitaire de batch.py*

```

1 from apps.shared.models.recipe import Recipe
2
3 recipe = Recipe(0)
4
5 def test_recipe_init() :
6     r = Recipe(10)
7     assert r.list_product == []
8     assert r.id == 10
9
10 def test_add_product() :
11     recipe.add_product("a", 10, 0.1)
12     recipe.add_product("b", 100, 0.1)
13     assert len(recipe.list_product) == 2
14     assert recipe.list_product[0]["product"] == "a"
15     assert recipe.list_product[0]["quantity_target"] == 10
16     assert recipe.list_product[0]["quantity_min"] == 9
17     assert recipe.list_product[0]["quantity_max"] == 11
18
19 def test_get_product() :
20     products = recipe.get_products()
21     assert len(products) == 2
22     assert products[0]["product"] == "a"
23     assert products[0]["quantity_target"] == 10
24     assert products[0]["quantity_min"] == 9
25     assert products[0]["quantity_max"] == 11
26
27 def test_get_recipe_by_product() :
28     product = recipe.get_product("b")
29     assert product["product"] == "b"
30     assert product["quantity_target"] == 100
31     assert product["quantity_min"] == 90
32     assert product["quantity_max"] == 110
33
34 def test_get_id() :
35     assert recipe.get_id() == 0
36
37 def test_print_recipe(capsys) :
38     recipe.print_recipe()
39     captured = capsys.readouterr()
40
41     assert captured.out == "Recipe\nř0\nProduct: a, Quantity_target:
    ↳ : 10, Quantity_min: 9.0, Quantity_max: 11.0.\nProduct: b
    ↳ , Quantity_target: 100, Quantity_min: 90.0, Quantity_max:
    ↳ : 110.0.\n"

```

Listing B.3 – *Test unitaire de recipe.py*

```

1 import numpy as np
2 import pytest
3 from apps.shared.models import batch, storage
4
5 # Importer les fonctions à tester depuis votre module.
6 # Adaptez le chemin d'import à votre projet.
7 from apps.shared.models.solver import (
8     split_array,
9     solve,
10    update_matrix_decision,
11    create_block_diagonal,
12    find_vials_matrix_test,
13    clean_matrix,
14    verifying_number_of_vial_by_recipe,
15    fusion_array,
16    creation_list_id,
17    fusion_dictionary_by_product,
18    creation_list_missing_elements,
19    calcul_quantities,
20 )
21
22 class DummyVial:
23     def __init__(self, id, quantity, product):
24         self._id = id
25         self._quantity = quantity
26         self._product = product
27
28     def get_quantity(self):
29         return self._quantity
30
31     def get_id(self):
32         return self._id
33
34     def get_product(self):
35         return self._product
36
37 class DummyStorage:
38     def __init__(self, vials):
39         self.vials = vials
40
41     def get_vials(self):
42         return self.vials
43
44     def get_vials_by_product(self, product):
45         return [v for v in self.vials if v.get_product() == product]
46
47     def get_vials_by_id(self, id):
48         for vial in self.vials:
49             if vial.get_id() == id:
50                 return vial
51         return None
52
53 class DummyRecipe:
54     def __init__(self, id, product_info):
55         self._id = id
56         self._product_info = product_info
57
58     def get_id(self):
59         return self._id

```

Listing B.4 – Test unitaire de solve.py

```

1     def get_products(self):
2         return self._product_info
3
4 class DummyBatch:
5     def __init__(self, products, recipes):
6         """
7         products : liste de noms de produits, ex: ['A']
8         recipes : liste de tuples (id, product_info)
9         """
10        self.products = products
11        self.recipes = recipes
12
13    def get_list_products(self):
14        return self.products
15
16    def get_recipes(self):
17        return [DummyRecipe(r[0], r[1]) for r in self.recipes]
18
19    def get_recipes_by_product(self, product):
20        return [r for r in self.recipes if r[1].get('product') ==
21                ↪ product]
22
23    def test_split_array_with_list():
24        arr = [1, 2, 3, 4, 5, 6]
25        result = split_array(arr, 3)
26        assert result == [[1, 2, 3], [4, 5, 6]]
27
28    def test_split_array_with_numpy_array():
29        arr = np.array([1, 2, 3, 4, 5, 6])
30        result = split_array(arr, 2)
31        result_as_list = [chunk.tolist() for chunk in result]
32        assert result_as_list == [[1, 2], [3, 4], [5, 6]]
33
34    def test_update_matrix_decision():
35        A = np.zeros((2, 5))
36        list_ids = [[2, 3], [5]]
37        list_recipes = [(0, {}), (1, {})]
38        updated_A = update_matrix_decision(A.copy(), list_ids,
39        ↪ list_recipes)
40        expected = np.zeros((2, 5))
41        expected[0, 1] = 1
42        expected[0, 2] = 1
43        expected[1, 4] = 1
44        np.testing.assert_array_equal(updated_A, expected)
45
46    def test_create_block_diagonal():
47        pattern = np.array([[1, 2], [3, 4]])
48        nbreRecipes = 3
49        sizeStorage = 4
50        result = create_block_diagonal(pattern, nbreRecipes, sizeStorage)
51        assert result.shape == (6, 12)
52        for i in range(nbreRecipes):
53            start_row = i * pattern.shape[0]
54            start_col = i * sizeStorage
55            block = result[start_row:start_row+pattern.shape[0], start_col
56            ↪ :start_col+pattern.shape[1]]
57            np.testing.assert_array_equal(block, pattern)
58
59    def test_clean_matrix():
60        matrix = [

```

Listing B.5 – Test unitaire de solve.py

```

1      [1, -1, 2],
2      [-1, -1, 3],
3      [4, 5, -1]
4  ]
5  cleaned = clean_matrix(matrix)
6  expected = [
7      [1, 2],
8      [3],
9      [4, 5]
10 ]
11  assert cleaned == expected
12
13  def test_verifying_number_of_vial_by_recipe():
14      decision_matrix = np.array([
15          [1, 1, 1, 0],
16          [0, 1, 0, 0]
17      ])
18      reactor_capacity = 2
19      unfeasible, updated_matrix = verifying_number_of_vial_by_recipe(
20          ↪ decision_matrix.copy(), reactor_capacity)
21      np.testing.assert_array_equal(updated_matrix[0], np.zeros(4))
22      np.testing.assert_array_equal(updated_matrix[1], decision_matrix
23          ↪ [1])
24      assert unfeasible == [0]
25
26  def test_fusion_array():
27      array_1 = [(1, {'product': 'A'})], (2, {'product': 'B'})]
28      array_2 = [(2, {'product': 'B'})], (3, {'product': 'C'})]
29      result = fusion_array(array_1.copy(), array_2)
30      ids = {entry[0] for entry in result}
31      assert ids == {1, 2, 3}
32
33  def test_creation_list_id():
34      decision_matrix = np.array([
35          [1, 0, 1, 0],
36          [0, 1, 0, 0]
37      ])
38      result = creation_list_id(decision_matrix)
39      expected = [(0, 2], 0), ([1], 1)]
40      assert result == expected
41
42  def test_fusion_dictionary_by_product():
43      input_list = [
44          {'product': 'A', 'quantity': 10},
45          {'product': 'B', 'quantity': 5},
46          {'product': 'A', 'quantity': 7},
47      ]
48      result = fusion_dictionary_by_product(input_list)
49      result_dict = {entry['product']: entry['quantities'] for entry in
50          ↪ result}
51      assert 'A' in result_dict and 'B' in result_dict
52      assert sorted(result_dict['A']) == [7, 10]
53      assert result_dict['B'] == [5]
54
55  def test_creation_list_missing_elements():
56      unfeasible_recipes = [
57          (0, {'product': 'A', 'quantity_target': 10}),
58          (1, {'product': 'B', 'quantity_target': 5}),
59      ]

```

Listing B.6 – Test unitaire de solve.py

```

1      ]
2      ids, missing = creation_list_missing_elements(unfeasible_recipes)
3      assert ids == [0, 1]
4      missing_products = {elem['product'] for elem in missing}
5      assert missing_products == {'A', 'B'}
6
7  def test_calcul_quantities():
8      list_id = [
9          ([1, 2], 0),
10         ([3], 1)
11     ]
12     vial1 = DummyVial(1, 10, 'A')
13     vial2 = DummyVial(2, 15, 'A')
14     vial3 = DummyVial(3, 20, 'B')
15     storage = DummyStorage([vial1, vial2, vial3])
16     list_unfeasible_recipe = [
17         (1, {'product': 'B', 'quantity_target': 5})
18     ]
19     quantities = calcul_quantities(storage, list_id,
20                                     ↪ list_unfeasible_recipe)
21
22     expected = [
23         ({'A': 25}, 0),
24         ({'B': 0}, 1)
25     ]
26     assert quantities == expected
27
28 def test_split_array_empty():
29     arr = []
30     result = split_array(arr, 3)
31     assert result == []
32
33 def test_create_block_diagonal_with_zero_recipes():
34     pattern = np.array([[1, 2], [3, 4]])
35     result = create_block_diagonal(pattern, 0, 5)
36     assert result.size == 0
37
38 def test_update_matrix_decision_empty_ids():
39     A = np.zeros((1, 3))
40     list_ids = [[]]
41     list_recipes = [(0, {})]
42     updated_A = update_matrix_decision(A.copy(), list_ids,
43                                       ↪ list_recipes)
44     np.testing.assert_array_equal(updated_A, A)
45
46 def test_find_vials() :
47     recipes = [{"chemical_element": "Farine", "quantity": 10, "tolerance": 0.1}]
48     b = batch.Batch()
49     b.add_recipe(recipes)
50
51     s = storage.Storage()
52     s.generate_vials("Farine", 10, 0)
53     decision_matrix, slack_variable, status = find_vials_matrix_test(b
54                               ↪ .get_recipes_by_product("Farine"), s.get_vials(), 2)
55
56     assert status == 0
57     assert slack_variable == [0]
58     assert decision_matrix == [[0]]

```

Listing B.7 – Test unitaire de solve.py

```

1
2 def test_solve() :
3     recipes = [{"chemical_element": "Farine", "quantity": 10, "tolerance"
4                 ↪ ":0.1"}]
5     b = batch.Batch()
6     b.add_recipe(recipes)
7
8     s = storage.Storage()
9     s.generate_vials("Farine", 10, 0)
10    list_id, quantities, id_undecidable_recipe, missing_elements =
11    ↪ solve(b, s, 2)
12    print(quantities)
13    print(id_undecidable_recipe)
14    print(missing_elements)
15    assert list_id == [[0], 0]
16    assert quantities == [{"Farine" : 10}, 0]
17    assert id_undecidable_recipe == []
18    assert missing_elements == []

```

Listing B.8 – Test unitaire de solve.py


```

1
2 import pandas as pd
3 from apps.shared.models.storage import Storage
4
5 storage = Storage()
6
7 def test_storage_init() :
8     assert storage.vials == []
9
10 def test_generate_vials() :
11     storage.generate_vials("a", 1, 0)
12
13     assert len(storage.vials) == 1
14
15 def test_get_vials(capsys) :
16     assert len(storage.get_vials()) == 1
17
18     storage.generate_vials("b", 1, 2)
19     assert len(storage.get_vials()) == 2
20
21     (storage.get_vials()[0]).print_caracteristique()
22     captured = capsys.readouterr()
23     assert captured.out == "product_:a_qty_:1_id_:0\n"
24     (storage.get_vials()[1]).print_caracteristique()
25     captured = capsys.readouterr()
26     assert captured.out == "product_:b_qty_:1_id_:2\n"
27
28 def test_get_vials_by_product() :
29     vials_a = storage.get_vials_by_product("a")
30     vials_b = storage.get_vials_by_product("b")
31     vials_c = storage.get_vials_by_product("c")
32
33     for vial in vials_a :
34         assert vial.get_product() == "a"
35
36     for vial in vials_b :
37         assert vial.get_product() == "b"
38
39     assert vials_c == []
40
41 def test_get_vial_by_id() :
42     vial0 = storage.get_vials_by_id(0)
43     vial1 = storage.get_vials_by_id(1)
44     vial2 = storage.get_vials_by_id(2)
45
46     assert vial0.get_id() == 0
47     assert vial1 == None
48     assert vial2.get_id() == 2
49
50 def test_print_storage(capsys) :
51     storage.print_storage()
52
53     captured = capsys.readouterr()
54     assert captured.out == "product_:a_qty_:1_id_:0\nproduct_:b_
55         ↪ qty_:1_id_:2\n"
56
57 def test_generation_with_dataframe() :
58     new_storage = Storage()

```

Listing B.9 – Test unitaire de storage.py

```

1     parameters = [0, 7, "drawer", "Standardization", "caps", "sodium_
      ↪ fluoride", "7681-49-4",
2         "SwissCAT-649433", "[F-].[Na+]", "1S/FH.Na/h1H;/q
      ↪ ;+1/p-1", "NaF", 41.99,
3         2.56, 3.327]
4
5     df = pd.DataFrame(columns=["vialID", "vialGloveBoxLocationID", "
      ↪ vialGloveBoxLocationName", "gloveBox",
6         "vialType", "Chemical_name", "CAS_
      ↪ Number", "SwissCATNumber", "
      ↪ Smiles", "Inchi",
7         "Molecular_formula", "Molecular_mass_[g
      ↪ /mol]", "Density_[mg/L]", "
      ↪ Quantity_[mg]"])
8
9     df.loc[0] = parameters
10
11     new_storage.generate_storage_from_df(df)
12     assert len(new_storage.get_vials()) == 1
13     vial = new_storage.get_vials()[0]
14     assert vial.get_id() == parameters[0]
15     assert vial.get_product() == parameters[5]
16     assert vial.get_quantity() == parameters[-1]
17
18     def test_generation_with_excel() :
19         parameters = [7014, 7, "drawer", "Standardization", "caps", "
      ↪ sodium_fluoride", "7681-49-4",
20             "SwissCAT-649433", "[F-].[Na+]", "1S/FH.Na/h1H;/q
      ↪ ;+1/p-1", "NaF", 41.99,
21             2.56, 3.327]
22         new_storage = Storage()
23         path = "tests/input/data_base/"
24         new_storage.generate_storage_from_excel(path, 'vial', {'vialType':
      ↪ 'caps'})
25
26         assert len(new_storage.get_vials()) == 1
27         vial = new_storage.get_vials()[0]
28         assert vial.get_id() == parameters[0]
29         assert vial.get_product() == parameters[5]
30         assert vial.get_quantity() == parameters[-1]
31
32         parameters = [7014, 7, "drawer", "Standardization", "caps", "
      ↪ sodium_fluoride", "7681-49-4",
33             "SwissCAT-649433", "[F-].[Na+]", "1S/FH.Na/h1H;/q
      ↪ ;+1/p-1", "NaF", 41.99,
34             2.56, 3.327]
35         new_storage2 = Storage()
36         path = "tests/input/data_base/"
37         new_storage2.generate_storage_from_excel(path, 'vial')
38
39         assert len(new_storage2.get_vials()) == 1
40         vial2 = new_storage2.get_vials()[0]
41         assert vial2.get_id() == parameters[0]
42         assert vial2.get_product() == parameters[5]
43         assert vial2.get_quantity() == parameters[-1]

```

Listing B.10 – Test unitaire de storage.py

```

1 import pytest
2 from apps.shared.models.vials import *
3
4 vial = Vials("a", 1, 0)
5
6 def test_generate_vial() :
7
8     assert vial.id == 0
9     assert vial.product == "a"
10    assert vial.quantity == 1
11
12 def test_get_id() :
13     assert vial.get_id() == vial.id
14
15 def test_get_product() :
16     assert vial.get_product() == vial.product
17
18 def test_get_quantity() :
19     assert vial.get_quantity() == vial.quantity
20
21 def test_raise_quantity() :
22     with pytest.raises(ValueError) :
23         Vials("a", -1, 0)
24
25 def test_raise_id() :
26     with pytest.raises(ValueError) :
27         Vials("a", 1, -1)
28
29 def test_print_vial(capsys) :
30     vial.print_caracteristique()
31     captured = capsys.readouterr()
32     assert captured.out == "product: a quantity: 1 id: 0\n"

```

Listing B.11 – *Test unitaire de vial.py*

N° de test	Fonction	Description	Entrées	Tests effectués	Résultat attendu	Résultat	Validation
1	connection()	Test de la fonction	ip : "192.168.1.2"	- Tentative de connexion avec une bonne ip	"Connection with the robot (ip : 192.168.1.2) is established"	Connection with the robot (ip : 192.168.1.2) is established	
			ip : "192.168.1.3"	- Tentative de connexion avec une mauvaise ip	"Connection error : Timeout connecting to UR dashboard server."	"Connection error : Timeout connecting to UR dashboard server."	
2	pick_wellplate()	Test du fonctionnement normal de la fonction pick_wellplate		- Appel de la fonction avec URRRTDE	Prise de la wellplate dans le sas et dépose à la place dans la glove box	Prise de la wellplate dans le sas et dépose à la place dans la glove box	
3	give_wellplate()	Test du fonctionnement normal de la fonction give_wellplate		- Appel de la fonction avec URRRTDE	Prise de la wellplate dans la glove box puis dépose dans le sas	Prise de la wellplate dans la glove box puis dépose dans le sas	
4	pick_and_place_v	Test du fonctionnement normal de la fonction pick_and_place_v	("P24", "C3")	- Appel de la fonction avec URRRTDE	Prise de la capsule situé en P24 sur la wellplate puis dépose de celle-ci en C3 dans la paradox	Prise de la capsule situé en P24 sur la wellplate puis dépose de celle-ci en C3 dans la paradox	
5	pick_paradox()	Test du fonctionnement normal de la fonction pick_paradox		- Appel de la fonction avec URRRTDE	Prise de la paradox dans le sas et dépose à la place dans la glove box		
6	give_paradox()	Test du fonctionnement normal de la fonction give_paradox		- Appel de la fonction avec URRRTDE	Prise de la paradox dans la glove box et dépose dans le sas		

Glossaire

glovebox Boîte scellée permettant de manipuler les objets à l'intérieur grâce à des gants.. ix, 2, 5, 8, 9, 13, 14, 28, 29, 33, 35

microcapsule Petite capsule en verre servant à contenir des produits chimiques.. ix, x, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 18, 19, 20, 21, 22, 28, 33, 34, 35

paradox Nom de la boîte contenant les réacteurs.. ix, 5, 9, 13, 28, 29, 33

wellplate Plaque contenant les microcapsules.. ix, 2, 5, 9, 13, 33