

# Documentation EMS - partie Développeur

youen froger

May 2023

## 1 Introduction

La documentation est séparée en deux parties (Développeur et Exploitation), dont les sujets sont répartis ainsi :

### 1.1 Exploitation

lien : <https://www.overleaf.com/project/646c7383bacf9beacc28712c>

- Comment est constitué l'EMS : description générale des modules et de leur fonctionnement logique (simplifié)
- Sources de données de l'EMS (format et disponibilités) / où mettre les droits d'accès.
- Sorties de l'EMS par ordre d'importance et leur signification
- Fonctionnement nominal (chemin macro) de l'EMS et résilience
- Installation et mise en route de l'EMS (y inclus contraintes sur les appareils et contexte technique)
- Exploitation (paramètres et robustesse/fragilité du code)
- FAQ utilisateur/exploitant

### 1.2 Développeur

- Comment est constitué l'EMS : description générale des modules et de leur fonctionnement logique (détaillé)
- Documentation simplifiée du solveur : détail des consommateurs, fonction objectif...
- Chemin de la donnée plus précis, et comment ajouter de nouvelles sources
- Comment rajouter un nouveau type de consommateur => *à compléter par l'ENS*
- Comment modifier la source des prévisions d'entrée pour ELFE
- Comment relancer un scénario passé
- Compléments sur les sources de données
- FAQ développeur

## 2 Structure du projet

L'EMS est décomposé en différents services et modules. Pour le rôle des différents services et modules, se référer à la documentation "exploitation". Dans cette documentation nous allons nous concentrer sur le module "solveur" qui est le coeur de l'EMS. Ce module a pour but de formuler, de résoudre et d'interpréter les résultats d'un problème d'optimisation, dont le calcul permet de déterminer le placement optimal des consommations dans le but de minimiser l'import d'énergie. Pour ce faire, le temps est découpé en pas de temps d'une taille choisie (par défaut 15 min). La plupart des modules servent à récupérer les informations nécessaire au calcul final.

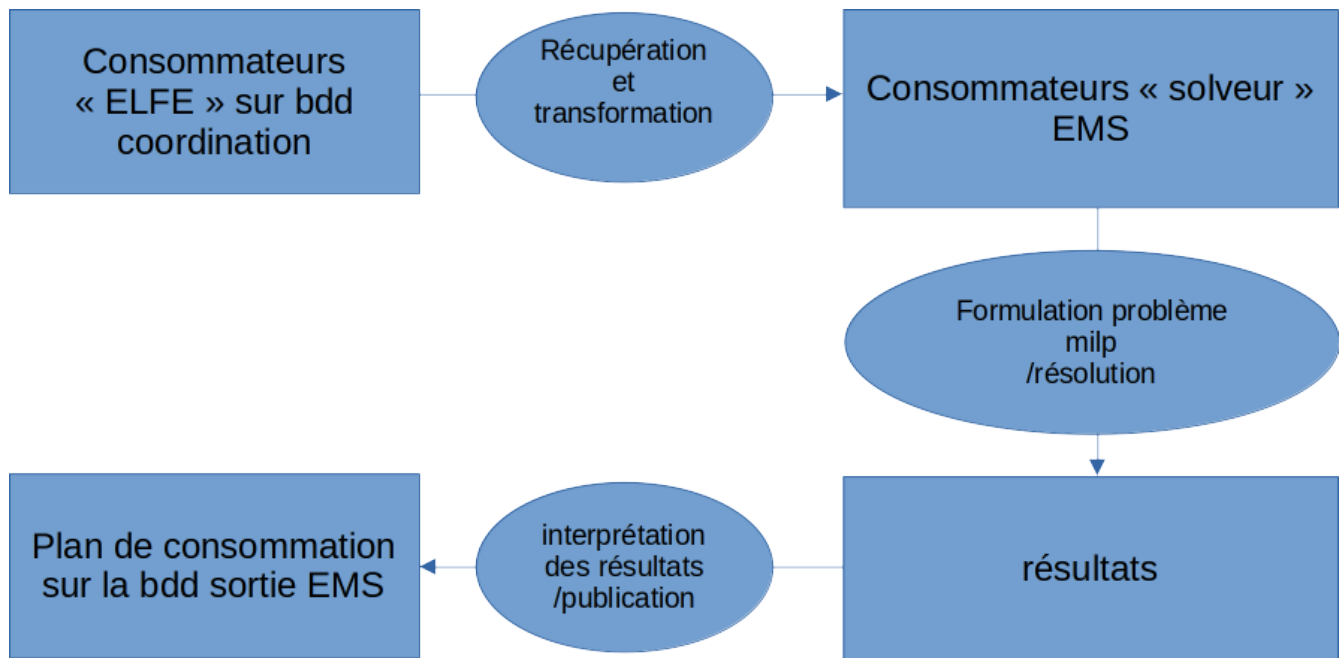


Figure 1: détail du flux de donnée au sein de l'EMS launcher

### 3 Le coeur de l'EMS : Le module solution

#### 3.1 présentation du solveur

Le solveur se présente comme un module qui prend en entrée des paramètres de simulation et une liste de consommateurs sous une certaine forme et qui formule, résout et fournit les résultats d'un problème d'optimisation sous forme des valeurs des variables d'optimisation. Il expose également une interface qui permet de récupérer les décisions de planification des différents consommateurs concernés. Il est fourni avec des classes de base qui permettent de traduire en contraintes

#### 3.2 Formulation du problème mathématique

##### 3.2.1 MILP

Le solveur formule un problème d'optimisation de catégorie MILP (Mixed Integer Linear Programming) en interne et fait appel à un solveur externe sous license MIT (scipy.optimize.milp) qui excelle à résoudre rapidement ce genre de problèmes.

Cette catégorie de problèmes se formule de la manière suivante :

Soit:

- $N_{opt}$  le nombre de variables d'optimisation.
- $\vec{X}$  le vecteur des variables d'optimisation
- $\vec{Integ}$  le vecteur qui contient les informations sur l'"intégralité" des variables de  $\vec{X}$  ( $\vec{Integ}[i] = 0 \Rightarrow \vec{X}[i]$  est une valeur qui n'est pas contrainte de rester entière et  $\vec{Integ}[i] = 1 \Rightarrow \vec{X}[i]$  est contrainte à rester entière ou le plus proche possible d'entière)
- $\vec{C}$  le vecteur des coûts associés aux variables d'optimisation
- $\vec{B}_l$  le vecteur des bornes inférieures pour le respect des contraintes
- $\vec{B}_h$  le vecteur des bornes supérieures pour le respect des contraintes
- $A$  La matrice de contraintes

- $l$  la borne inférieure de valeur pour les variables d'optimisation
- $h$  la borne supérieure de valeur pour les variables d'optimisation

Minimiser  $\vec{C} \cdot \vec{X}$

Tel que:

- $\vec{B}_l \leq A * \vec{X} \leq \vec{B}_h$
- $\forall x \in \vec{X}, l \leq x \leq h$
- Certains éléments spécifiques de  $\vec{X}$  doivent être entiers (ou le plus proche possible d'entiers, approximation de codage des flottants). Ces éléments sont spécifiés dans  $Integ$

Le solveur MILP va fixer les valeurs de  $\vec{X}$  afin de minimiser le produit scalaire  $\vec{C} \cdot \vec{X}$  tout en respectant les contraintes. Cela correspond à la minimisation de  $\sum_{i \in [0, n[} c_i * x_i$ , soit un coût linéaire avec chaque variable d'optimisation. On peut traiter la matrice contraintes lignes par ligne également. Chaque ligne correspond à un contrainte linéaire (la somme des variables d'optimisation chacune multipliée par un coefficient est comprise entre deux valeurs).

La notation  $\vec{C}$  a ici été choisie pour correspondre à la documentation du module MILP. Par la suite le vecteur  $\vec{C}$  sera appelé  $\vec{D}$  ou "le vecteur coût",  $C$  désignant des consommateurs dans le contexte de ELFE.

### 3.2.2 MILP, pour ELFE

Voici la tant attendue section de comment ça marche en vrai ! Prenez votre courage à deux mains, on rentre dans le concret !

Pour tous le problème, on imposera que les valeurs des variables d'optimisation sont strictement positives. Cette contrainte n'est pas exprimée dans la matrice contrainte mais de manière globale.

Pour le placement des différentes machines, on va découper la durée de simulation ( $t_{sim} = 24h$ ) en pas de temps d'une durée constante ( $time\_delta = 900s$ ). On discrétise donc le temps en  $N_{step} = 96$  pas.

Le vecteur des variables d'optimisation est séparé en deux parties : La première partie va contenir par construction l'import d'énergie prévu à chaque pas de temps une fois l'optimisation réalisée. La seconde va être séparée en blocs, chacun représentant les décisions et les variables internes d'un consommateur (au sens "une machine que l'EMS doit placer"). L'EMS doit placer  $N_{conso}$  consommateurs au total, numérotées de 0 à  $N_{conso} - 1$ . Il est important de savoir que le nombre de variables d'optimisations nécessaires à l'optimisation n'est pas constant d'un consommateur à l'autre, même au sein de la même catégorie de consommateurs. On va définir pour la suite de ce document le nombre de variables d'optimisation d'un consommateur  $i$   $N_{var}[i]$  (dans le code de l'EMS il est appelé *minimizing\_variable\_count* ).

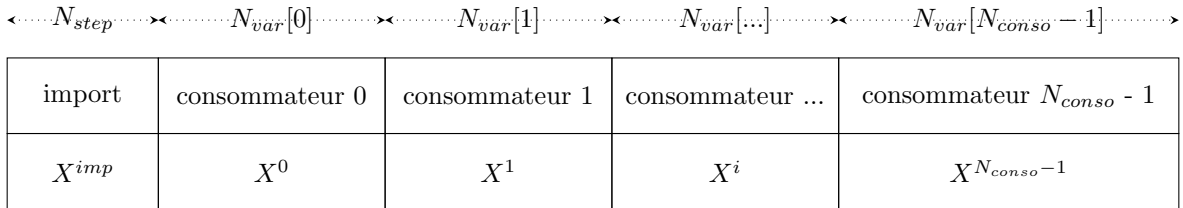


Figure 2: vecteur d'optimisation

Le vecteur coût est défini par blocs de manière analogue. L'objectif étant de minimiser l'import, un coût de 1 est affecté à chaque import. Les sous-vecteurs  $D^i$  sont fixés à 0 sur tous les consommateurs actuels, mais il est possible d'affecter un coût à un nouveau modèle de consommateur

$$\leftarrow N_{step} \rightarrow \leftarrow N_{var}[0] \rightarrow \leftarrow N_{var}[1] \rightarrow \leftarrow N_{var}[\dots] \rightarrow \leftarrow N_{var}[N_{conso}-1] \rightarrow$$

import	consommateur 0	consommateur 1	consommateur ...	consommateur $N_{conso} - 1$
1	$D^0$	$D^1$	$D^i$	$D^{N_{conso}-1}$

Figure 3: vecteur coût

Pour ce qui est du vecteur  $\vec{Integ}$ , il est également décomposé en blocs. Les imports à chaque instant étant des valeurs continues, le sous vecteur  $Integ^{imp}$  est rempli de 0

$$\leftarrow N_{step} \rightarrow \leftarrow N_{var}[0] \rightarrow \leftarrow N_{var}[1] \rightarrow \leftarrow N_{var}[\dots] \rightarrow \leftarrow N_{var}[N_{conso}-1] \rightarrow$$

import	consommateur 0	consommateur 1	consommateur ...	consommateur $N_{conso} - 1$
0	$Integ^0$	$Integ^1$	$Integ^i$	$Integ^{N_{conso}-1}$

Figure 4: vecteur  $\vec{integ}$

La matrice contraintes est elle aussi définie par bloc. Chaque ligne de cette matrice correspond à l'expression d'une contrainte linéaire. Les  $N_{step}$  premières contraintes servent à construire l'import à chaque instant en prenant en compte les consommations flexibles. Cela s'exprime de la manière suivante au pas de temps  $i$  :

$$import[i] - \sum_j conso_{flex,j}[i] \geq conso_{non\ flex}[i] - prod[i] \quad (1)$$

ce qui implique que

$$import[i] \geq \sum_j conso_{flex,j} + conso_{non\ flex}[i] - prod[i] \quad (2)$$

Donc l'import est supérieur ou égal au déficit de production une fois les consommations flexibles placées. Comme l'import est positif et qu'on minimise la somme des imports (qui revient à minimiser les imports à chaque pas de temps, les imports étant tous positifs) l'import peut prendre deux valeurs :

$$import[i] = 0 \quad ssi \sum_i conso_{flex,j} + conso_{non\ flex}[i] - prod[i] \leq 0 \quad (3a)$$

$$import[i] = \sum_i conso_{flex,j} + conso_{non\ flex}[i] - prod[i] \quad sinon \quad (3b)$$

Cela signifie que l'import est nul lorsque l'on produit plus que ce que l'on consomme, sinon il est égal au déficit de production une fois les consommations flexibles placées.

Dans ce problème, on part d'une prévision de consommation et de production non flexibles, considérées connues par la suite. Il ne reste plus qu'à déterminer les différentes consommations flexibles ( $conso_{flex}$ ) à partir des différentes variables d'optimisation. Pour que le solveur puisse résoudre, cette relation sera linéaire.

Pour chaque consommateur, le sous-vecteur des variables d'optimisation contient les variables de décision et les variables internes, regroupées par type. La puissance consommée à chaque instant  $i$  par un consommateur  $j$  n'est fonction que de ses variables de décision (pour les consommateurs considérés dans le projet ELFE). Pour chaque variable de décision  $k$ , son impact sur la consommation à l'instant  $i$  est linéaire par design et son coefficient est noté  $P_{i,k}^j$ .

$$conso_{flex,j}[i] = \sum_k P_{i,k}^j * X^j[k] \quad (4)$$

Et on a donc :

$$conso_{flex}[i] = \sum_j \sum_k P_{i,k}^j * X^j[k] \quad (5)$$

On peut donc réécrire l'équation 1 de la manière suivante :

$$import[i] - \sum_j \sum_k P_{i,k}^j * X^j[k] \geq conso_{non\ flex}[i] - prod[i] \quad (6)$$

Pour la suite on notera  $M_j$  le nombre de variables d'optimisations dans le rôle des variables de décision pour le consommateur j

Les lignes suivantes sont les contraintes internes à chaque consommateurs qui permettent d'exprimer les contraintes utilisateurs, les contraintes physiques et celles qui assurent le bon fonctionnement du consommateur. Ces contraintes s'expriment uniquement à partir des variables d'optimisation du consommateur. Pour la suite on nommera  $C^i$  la matrice contraintes liée au consommateur i et  $N_{Co}[i]$  le nombre de contraintes du consommateur i (voir figure 5)

$\leftarrow N_{step} \rightarrow$		$\leftarrow N_{var}[0] \rightarrow$		$\leftarrow N_{var}[\dots] \rightarrow$		$\leftarrow N_{var}[N_{conso}-1] \rightarrow$	
	import	consommateur 0		...consommateur i...		consommateur $N_{conso} - 1$	
$\begin{matrix} \uparrow \\ \vdots \\ \uparrow \end{matrix}$ $N_{step}$	ID( $N_{step}$ )	$-P_{0,0}^0 \quad \dots \quad -P_{0,m_0}^0 \quad 0 \dots 0$	$-P_{0,0}^i \quad \dots \quad -P_{0,m_i}^i$	$-P_{0,0}^{N_{conso}-1} \quad -P_{0,1}^{N_{conso}-1} \quad 0$			
		$\vdots$	$\vdots$	$\vdots$			
		$-P_{N_{steps},0}^0 \quad \dots \quad -P_{N_{steps},m_0}^0 \quad 0 \dots 0$	$-P_{N_{steps},0}^i \quad \dots \quad -P_{N_{steps},m_i}^i$	$-P_{N_{steps},0}^{N_{conso}-1} \quad -P_{N_{steps},1}^{N_{conso}-1} \quad 0$			
$\begin{matrix} \uparrow \\ \vdots \\ \uparrow \end{matrix}$ $N_{Co}[0]$	0	$C_{0,0}^0 \quad \dots \quad C_{0,N_{var}[0]}^0$	0	0			
	$\vdots$	$\vdots$					
	$C_{N_{Co}[0],0}^0 \quad \dots \quad C_{N_{Co}[0],N_{var}[0]}^0$						
$\begin{matrix} \uparrow \\ \vdots \\ \uparrow \end{matrix}$ $N_{Co}[i]$	0	0	$C_{0,0}^i \quad \dots \quad C_{0,N_{var}[i]}^i$	0			
	$\vdots$	$\vdots$					
	$C_{N_{Co}[i],0}^i \quad \dots \quad C_{N_{Co}[i],N_{var}[i]}^i$						
$\begin{matrix} \uparrow \\ \vdots \\ \uparrow \end{matrix}$ $N_{Co}[N_{conso}-1]$	0	0	0	$C^{N_{conso}-1}$			

Figure 5: matrice contraintes

La première ligne-bloc correspond ici à l'expression de l'équation 6. Les ligne-blocs suivantes correspondent à l'expression des contraintes sur les variables internes de chaque consommateur.

Appelons  $Conso_{nonflex}[i]$  la prévision de consommation non flexible pour le i-ème pas de temps et  $Prod_{nonflex}[i]$  la prévision de production non flexible pour le i-ème pas de temps. Appelons  $B_l^j[i]$  la borne basse (respectivement  $B_h^j[i]$  la borne haute) pour la i-ème contrainte du consommateur j. Les bornes hautes et basses des contraintes sont définies par bloc comme dans la figure 6.

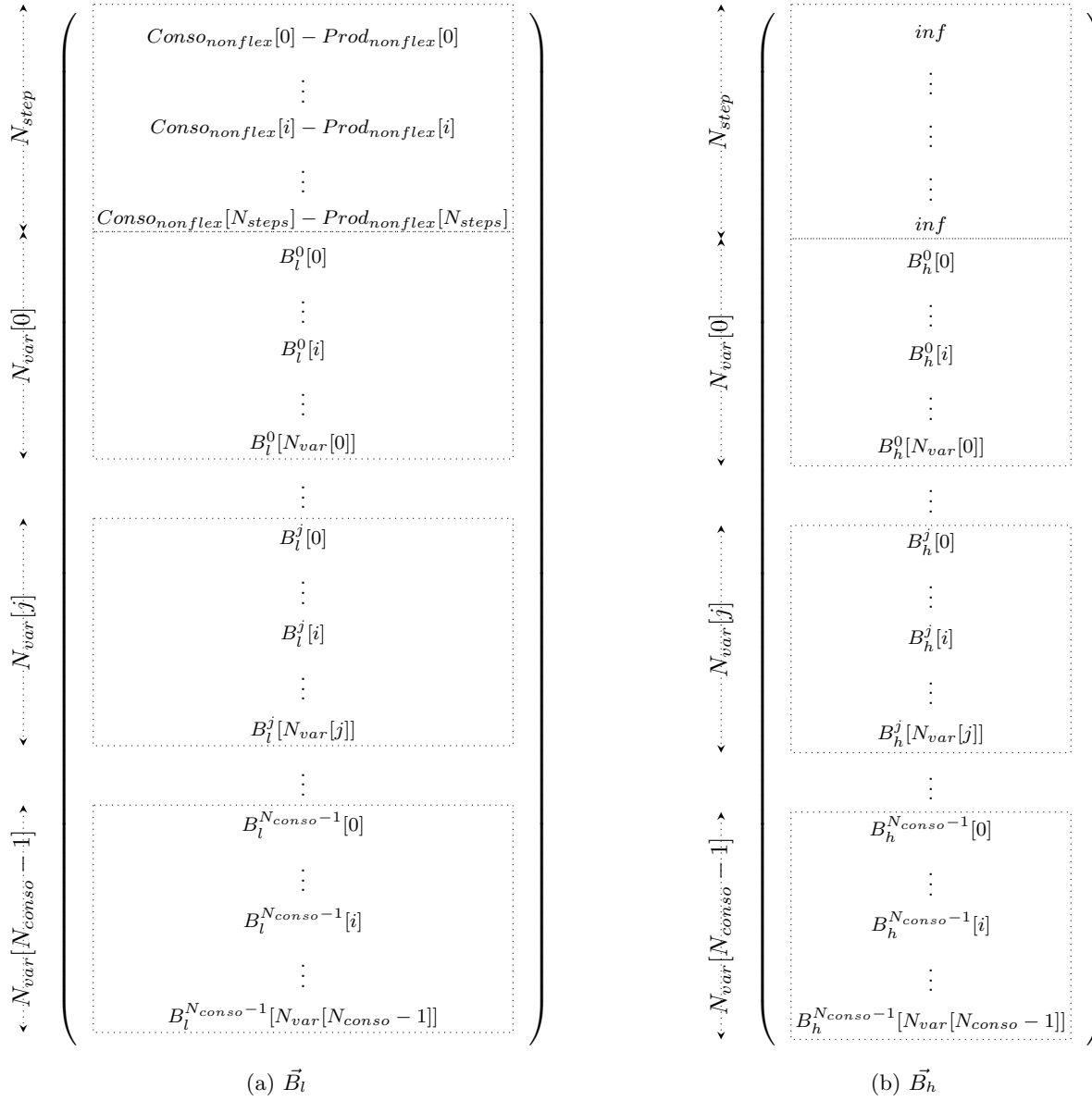


Figure 6: bornes des contraintes

### 3.2.3 Un exemple concret de matrice : on programme le placement d'une machine sur 5 pas de temps

Admettons que la prévision de consommation soit (en W) 1200 700 800 1000 200 et que la prévision de production soit (toujours en W) 3000 5000 3000 0 000

La prévision de consommation - production vaut donc -1800 -4300 -2200 1000 200

La machine que l'on va chercher à placer a un cycle bien connu, qui 3 pas de temps : 100, 200, 100. On va choisir de lui attribuer une variable d'optimisation par placement possible, qui correspond à l'information "doit-on lancer la machine à ce pas de temps ?". On va choisir des contraintes de telle sorte que la variable de décision vaille 1 si la machine doit être lancée à cet instant, 0 sinon. On a 3 placements possibles étant donné qu'elle n'aura pas le temps de finir son cycle avant la fin si on la place sur les deux derniers créneaux. Pour cette machine, on a donc  $N_{var} = 3$ . Comme on va imposer aux variables internes de la machine d'être entières, le vecteur  $\vec{integ}$  vaut donc

import					machine		
0	0	0	0	0	1	1	1

Figure 7:  $\vec{integ}$  exemple

Les variables internes de la machine ne contribuent pas directement au coût (elles le font indirectement par le biais de leur consommation, mais cela sera exprimé dans la matrice contrainte). Le vecteur coût devient est donc

import					machine		
1	1	1	1	1	0	0	0

Figure 8: vecteur coût exemple

Afin de lancer la machine au moins et seulement une fois, comme les variables internes correspondent à la décision booléenne "déclenche-t-on la machine à cet instant", on impose comme contrainte que la somme des variables internes soit égale à 1 (supérieure ou égale et inférieure ou égale à 1).

La matrice contrainte s'exprime donc

$$\begin{pmatrix} -1800 \\ -4300 \\ -2200 \\ 1000 \\ 200 \\ 1 \end{pmatrix} \leq \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & -100 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -200 & -100 & 0 \\ 0 & 0 & 1 & 0 & 0 & -100 & -200 & -100 \\ 0 & 0 & 0 & 1 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -100 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} imp[0] \\ \vdots \\ imp[5] \\ X^0[0] \\ X^0[1] \\ X^0[2] \end{pmatrix} \leq \begin{pmatrix} inf \\ inf \\ inf \\ inf \\ inf \\ 1 \end{pmatrix}$$


 : contraintes venant de la machine exemple

Figure 9: Matrice contrainte,  $B_l$  et  $B_h$  pour le cas exemple

La première ligne-bloc définie par le consommateur est celle qui permet d'exprimer l'impact de la décision sur la consommation / l'import comme défini par l'équation 6. On peut la traiter colonne par colonne : la première colonne représente ici la consommation au cours du temps si la machine commence au premier pas de temps

$$\begin{pmatrix} -1800 \\ -4300 \\ -2200 \\ 1000 \\ 200 \\ 1 \end{pmatrix} \leq \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & -100 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -200 & -100 & 0 \\ 0 & 0 & 1 & 0 & 0 & -100 & -200 & -100 \\ 0 & 0 & 0 & 1 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -100 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} imp[0] \\ \vdots \\ imp[5] \\ X^0[0] \\ X^0[1] \\ X^0[2] \end{pmatrix} \leq \begin{pmatrix} inf \\ inf \\ inf \\ inf \\ inf \\ 1 \end{pmatrix}$$


 : contraintes correspondant à l'expression de la consommation de la machine à chaque instant

Figure 10: Contraintes de la puissance consommée pour le cas exemple

La dernière ligne-bloc contient la contrainte qui fait qu'on ne place qu'une seule fois la machine

$$\begin{pmatrix} -1800 \\ -4300 \\ -2200 \\ 1000 \\ 200 \\ 1 \end{pmatrix} \leq \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & -100 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -200 & -100 & 0 \\ 0 & 0 & 1 & 0 & 0 & -100 & -200 & -100 \\ 0 & 0 & 0 & 1 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -100 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} imp[0] \\ \vdots \\ imp[5] \\ X^0[0] \\ X^0[1] \\ X^0[2] \end{pmatrix} \leq \begin{pmatrix} inf \\ inf \\ inf \\ inf \\ inf \\ 1 \end{pmatrix}$$


 : contrainte d'unicité du placement du départ de la machine

Figure 11: Contrainte d'unicité du placement du départ de la machine pour le cas exemple

### 3.3 interface du solveur

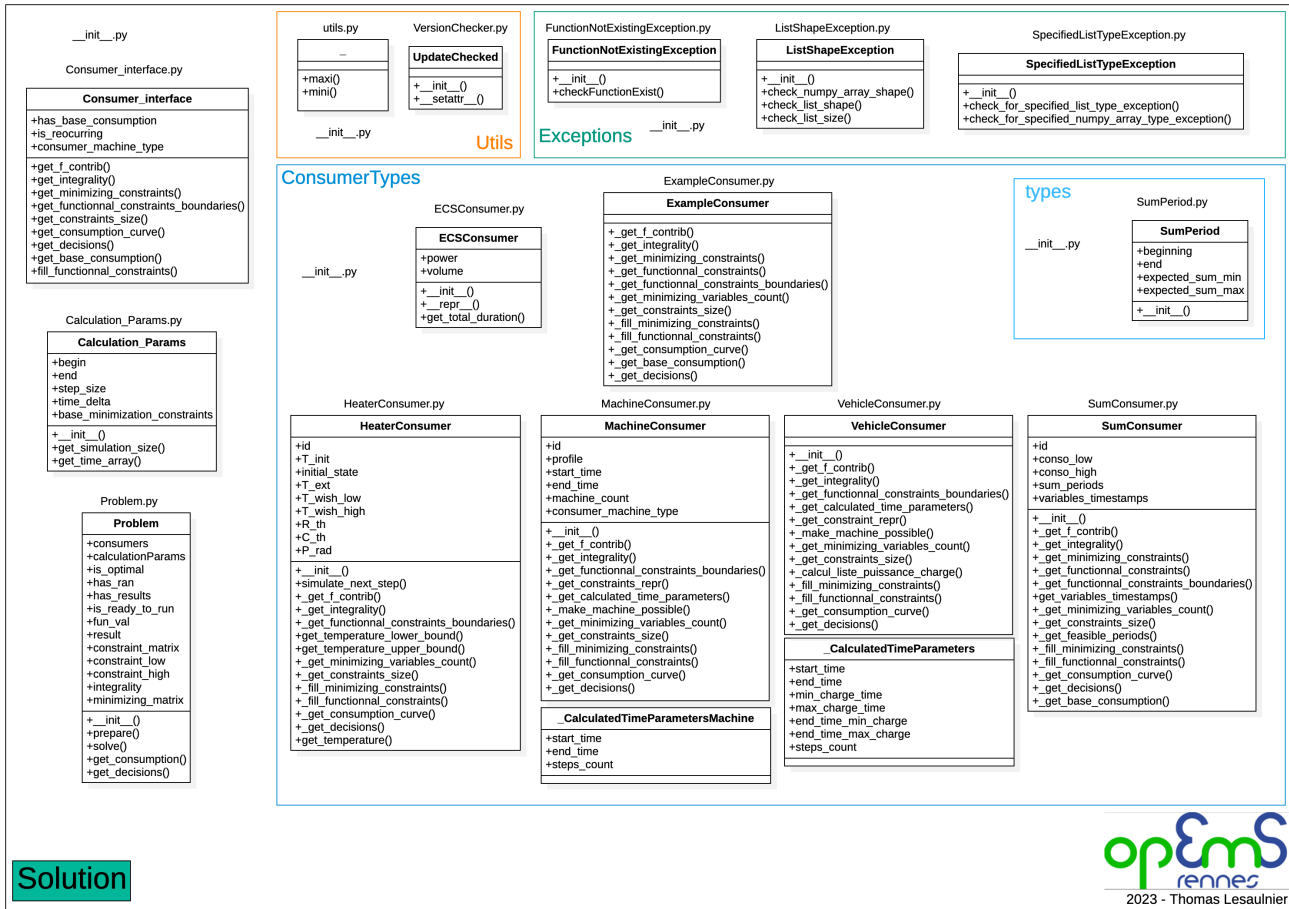


Figure 12: diagramme UML du solveur

Le solveur traite différents types de consommateurs qui ont chacun leurs propres contraintes. Chaque consommateur est représenté dans le code par une instance d'une sous-classe de `ConsumerInterface`. Il existe aujourd'hui 5 types de consommateurs dont voici la définition et le rôle :

- **MachineConsumer** : Il s'agit du consommateur le plus fréquemment utilisé. Comme dans l'exemple de 3.2.3, il s'agit de placer la consommation d'un appareil dont la courbe de consommation est connue à l'avance, ininterrompue et indivisible, en respectant des contraintes sur la date de début et la date de fin.
- **ECSCConsumer** (Eau Chaude Sanitaire) : Il s'agit d'un `MachineConsumer`, qu'il étend en rajoutant des informations supplémentaires de volume et de puissance pour pouvoir passer ces informations à une autre partie du module EMS Launcher, car on n'envoie pas uniquement la date de déclenchement mais également chacune des dates où l'EMS autorise ce consommateur à consommer. De plus, la durée autorisée est plus importante que celle du cycle qui a été utilisé pour placer la consommation, afin permettre au ballon d'eau chaude une recharge complète depuis 15°C. En résumé, on place la consommation en se basant sur celle de la veille, mais on autorise la ballon d'eau chaude à consommer suffisamment pour éviter les problèmes sanitaires.
- **VehicleConsumer** : Il s'agit d'un consommateur de type "véhicule électrique". Ce consommateur ressemble au `MachineConsumer` dans le sens qu'une fois la consommation lancée, on ne l'arrête pas. Cependant, la contrainte étant sur le niveau minimal de charge de la batterie, la courbe de consommation n'est pas fixe en fonction du moment de départ de la charge.
- **SumConsumer** : Il s'agit d'un consommateur dont on veut effacer une partie de la consommation sur une période. On va donc prendre des décisions entières en imposant que la somme des décisions soit comprise



entre une borne min et une borne max sur chaque période considérée. C'est actuellement utilisé pour faire un arbitrage entre un mode "eco" et un mode "confort" sur des chauffages de maison sur des périodes de présence des personnes. Les décisions prises pendant des pas de temps sur lesquels il n'y a pas de contraintes sont forcées à 0.

- HeaterConsumer : Il s'agit d'un consommateur sur lequel on place des contraintes de température. On va prendre les décisions en supposant un modèle RC simple. Les paramètres d'entrées sont la température initiale, la température extérieure, la puissance de chauffage, les valeurs de Rth et Cth et les bornes de températures souhaitées par l'utilisateur.

En plus de ces classes, deux classes sont indispensables pour le bon fonctionnement de l'EMS:

- La classe CalculationParams (solution.Calculation.Params, pour un exemple voir Fig.16) qui contient les paramètres de calculs :

- begin : int, timestamp de fin de la simulation, incluse (en secondes)
- end : int, timestamp de fin de la simulation, incluse (en secondes)
- step\_size : int, la durée d'un pas de simulation, en secondes. Doit être égal à time\_delta
- time\_delta : int, la durée entre deux pas de simulations en secondes. Doit être égale à sim\_size
- base\_minimization\_constraints : List[List[float]], contient une liste de listes d'entier. Actuellement seul le premier élément de la liste de listes est utilisé et contient la liste de consommation non flexible - production (considérée non flexible). Cela a été prévu pour pouvoir faire (ultérieurement) de l'optimisation multi-objectifs.

Cette classe fournit les fonctions suivantes :

- get\_simulation\_size qui retourne le nombre de pas de temps de la simulation.
- get\_time\_array qui fournit une liste contenant les timestamps des différents pas de temps de la simulation.

- La classe Problem, qui permet de formuler un problème (voir Fig.18 pour un exemple)

- Consumers : List[ConsumerInterface] : la liste des consommateurs à considérer
- calculationParams : CalculationParams, Les paramètres de simulation comme mentionnés ci-dessus
- La fonction prepare qui prend un booléen en entrée pour déterminer si le problème doit être re-préparé (si on veut réutiliser cet objet, pas conseillé). Cette fonction a pour rôle de préparer la formulation mathématique du problème avant de résoudre le problème
- La fonction solve qui prend 3 paramètres : le temps en secondes que l'on autorise au solveur, un booléen qui détermine le type de solveur à utiliser (toujours laisser à false, l'autre solveur ne respecte pas les contraintes d'intégralité, voué à disparaître dans de futures versions), et un booléen qui force le solveur à recalculer.
- une fois la fonction solve appelée, on peut récupérer les résultats en utilisant la fonction get\_decisions(). Cette fonction renvoie une liste d'objets sous forme de dictionnaires qui contiennent chacun les résultats d'un consommateur sous la forme suivante :
  - "id" : int, id elfe du consommateur
  - "reoccurring": un booléen qui n'est plus utilisé
  - "is\_ECS" : un booléen qui contient l'information quand à si l'équipement piloté est un chauffage ECS
  - "decisions": les décisions à chaque pas de temps pour le consommateur
  - "minimizing\_variables": les valeurs des variables d'optimisation associées au consommateurs, calculées pas l'EMS
  - "consumer": l'objet qui correspond au consommateur actuel.

### 3.3.1 classes de consommateurs

#### 3.3.1.1 MachineConsumer

Ce consommateur est utilisé pour placer entre deux pas de temps la consommation d'une machine dont on connaît le cycle et qu'on ne peut pas interrompre. Il s'agit par exemple de placer une machine à laver, un sèche-linge, un lave-vaisselle, etc...

##### 3.3.1.1.1 Définition

- `id:int`, id de l'équipement piloté ou mesuré pour ELFE
- `profile:List[int]`, la liste des consommations du cycle à placer au cours du temps, échantillonné à `time_delta`
- `start_time:int`, date de départ minimale pour le placement de la consommation
- `end_time:int` date de fin maximale souhaitée de la courbe de consommation
- `machine_count:int`, nombre de machines à placer en respectant les contraintes précédemment exprimées
- `consumer_machine_type:int`, (optionnel) non utilisé, le type de machine ELFE, utile pour le résultat

##### 3.3.1.1.2 Exemple

```
consumers=[MachineConsumer(id=1, profile=[300,500,600],start_time=3000, end_time=6000, machine_count=1, consumer_machine_type=0)]
```

#### 3.3.1.2 ECSCConsumer

Comme dit ci-dessus, il s'agit d'un MachineConsumer modifié qui permet de transmettre la durée totale de recharge complète d'un ballon d'eau chaude à partir de son volume et de sa puissance de chauffe

##### 3.3.1.2.1 Définition

- `id: int`, voir MachineConsumer
- `profile: List[int]`
- `start_time : int`, voir MachineConsumer
- `end_time : int`, voir MachineConsumer
- `power : int`, la puissance de chauffe du ballon en W
- `volume : int`, le volume du ballon en L
- `consumer_machine_type : int`, voir MachineConsumer

##### 3.3.1.2.2 Exemple

```
from solution.ConsumerTypes.ECSCConsumer import ECSCConsumer

power=500
volume=200
consumers=[ECSCConsumer(id=1, profile=[300,500,600],start_time=3000, end_time=6000, consumer_machine_type=0, power=power,volume=volume)]
```

#### 3.3.1.3 VehicleConsumer

Ce consommateur est utilisé pour placer la recharge d'une batterie de véhicule électrique

### 3.3.1.3.1 Définition

- id : int, l'id ELFE du consommateur
- power\_watt : int, la puissance en Watt du chargeur
- capacity\_watt\_hour : int, la capacité de la batterie en watt\*heures
- initial\_charger\_pourc : int, la charge initiale en pourcentage de la batterie
- end\_charge\_pourc : int, la charge souhaitée à la fin, en pourcentage de la batterie
- start\_time:int, date de départ minimale pour le placement de la consommation
- end\_time:int date de fin maximale souhaitée de la courbe de consommation

### 3.3.1.3.2 Exemple

```
from solution.ConsumerTypes.VehicleConsumer import VehicleConsumer
consumers=[VehicleConsumer(id=1, start_time=0, end_time=40, initial_charge_pourc=30, end_charge_pourc=80, capacity_watt_hour=4000, power_watt=1500)]
```

### 3.3.1.4 SumConsumer

#### 3.3.1.4.1 Définition

#### 3.3.1.4.2 Exemple

### 3.3.1.5 HeaterConsumer

#### 3.3.1.5.1 Définition

#### 3.3.1.5.2 Exemple

### 3.3.1.6 Votre propre consommateur

Comme vu dans la section 3.2.2, il y a un certain nombre et qui doit exposer un certain nombre de méthodes définies dans ExampleConsumer. Voici la liste des méthodes à implémenter pour définir un consommateur.

- def \_get\_minimizing\_variables\_count(self, calculationParams : CalculationParams) -> int: cette fonction calcule et retourne le nombre de variables d'optimisations nécessaires (il s'agit de définir  $N_{var}[i]$  dans la figure 5) pour la période considérée par la simulation dont les paramètres sont stockés dans la variable calculationParams
- def \_get\_constraints\_size(self, calculationParams : CalculationParams) -> int: Cette fonction calcule et retourne le nombre de contraintes (lignes de contraintes) à exprimer pour assurer le bon fonctionnement du consommateur (il s'agit de définir  $N_{Co}[i]$  dans la figure 5) pour la période considérée par la simulation dont les paramètres sont stockés dans la variable calculationParams
- def \_get\_f\_contrib(self, calculationParams : CalculationParams) -> List[float]: cette fonction calcule et retourne la contribution à la fonction coût des différentes variables d'optimisations sous forme d'une liste de flottants. Il s'agit de la partie "machine" telle que définie dans la partie "machine" du vecteur coût ( $D^i$  dans la Fig 3). Cette liste doit avoir un nombre d'éléments **égal** au nombre de variables d'optimisations définit dans la fonction \_get\_minimizing\_variables\_count
- def \_get\_integrality(self, calculationParams : CalculationParams) -> List[int] : Cette fonction calcule et retourne la contribution au vecteur  $\vec{integ}$  à partir des paramètres de simulation, sous la forme d'une liste d'entiers contenant 0 si la variable d'optimisation n'est pas forcée à être entière et 1 si elle doit être forcée à l'être. Il s'agit de la partie "machine" telle que définie dans la partie "machine" du vecteur  $\vec{integ}$  ( $Integ^i$  dans la Fig 4). Cette liste doit avoir un nombre d'éléments **égal** au nombre de variables d'optimisations définit dans la fonction \_get\_minimizing\_variables\_count

- `def _fill_minimizing_constraints(self, calculationParams: CalculationParams, tofill: np.ndarray, xpars: List[int], ypars: List[int]):` Cette fonction doit remplir la partie  $P^i$  de la matrice contrainte e dans la Fig.5. Pour rappel, si on appelle  $j$  le numéro de la ligne et  $k$  celui de la colonne, la matrice doit contenir  $-P_{j,k}^i$ , ou  $P_{j,k}^i$  représente la contribution de la valeur de la même variable d'optimisation à la consommation du jème pas de temps. la valeur écrite dans la matrice doit donc être négative si il y a consommation et positive si il y a production d'énergie. Pour ce faire, le solveur passe en paramètre à la fonction les paramètre de simulation au moyen de `calculationParams`, et passe en paramètre la matrice contrainte **Complète** en paramètre. Pour savoir ou se situe le bloc dans lequel on doit écrire, les variables `xpars` et `ypars` sont passé en paramètre. Il s'agit des coordonnées de  $P_{0,0}^i$ . `xpars` et `ypars` contiennent une liste d'index qui, en l'état sera toujours constituée d'une seul élément, mais cela permettra peut être ensuite d'optimiser sur plusieurs facteurs en ayant plusieurs lignes-blocs correspondant par exemple a plusieurs seuils d'import. Attention a ne pas dépasser  $N_{step}$  lignes et  $N_{var}[i]$  colonnes ou d'autres consommateurs pourraient être impactés. les `xpars` correspondent au numéro de la colonne et les `ypars` au numéro de la ligne. La matrice se remplit de bas en haut. Pour écrire  $P_{j,k}^i$ , il faut écrire `tofill[ypars[0] + j, xpars[0] + k] = - P_{j,k}^i`.
- `def _fill_functionnal_constraints(self, calculationParams: CalculationParams, tofill: np.ndarray, xpar: int, ypar: int) :` Il s'agit de remplir la partie  $C^i$  de la matrice contrainte tel que définit dans la Fig.5 pour ce consommateur. Pour ce faire, le solveur passe en paramètre à la fonction les paramètre de simulation au moyen de `calculationParams`, et passe en paramètre la matrice contrainte **Complète** en paramètre. Pour savoir ou se situe le bloc dans lequel on doit écrire, les variables `xpar` et `ypar` sont passé en paramètre. Cette fois il s'agit simplement d'entiers car on a définit le nombre de contraintes nécessaires au moyen de `_get_constraints.size`. Pour rappel, si on appelle  $j$  le numéro de la ligne et  $k$  celui de la colonne, la matrice doit contenir les  $C_{j,k}^i$ . Les valeurs de `xpar` et `ypar` correspondent respectivement à l'indice de la colonne et a celui de la ligne de l'élément  $C_{0,0}^i$ . Attention a ne pas dépasser  $N_{Co}[i]$  lignes et  $N_{var}[i]$  colonnes ou d'autres consommateurs pourraient être impactés. La matrice se remplit de haut en bas et de gauche à droite Pour remplir  $C_{j,k}^i$ , il faut écrire `tofill[ypar + j, xpar + k] = C_{j,k}^i`
- `def _get_functionnal_constraints_boundaries(self, calculationParams : CalculationParams) -> List[List[float]]` Il s'agit de définir les bornes min et max des contraintes telles que définies dans la fig.6. Cette fonction retourne une liste contenant **deux** listes contenant **chacune**  $N_{var}[i]$  valeurs. La première doit contenir les valeurs de  $B_l^i$  et la deuxième les valeurs de  $B_h^i$ . Il est important que l'ordre soit le même que celui utilisé dans `_fill_functionnal_constraints`
- `def _get_consumption_curve(self, calculationParams : CalculationParams, variables : List[float]) -> np.ndarray:` Cette fonction prend en paramètre les variables d'optimisations qui ont été calculées par le solveur et doit retourner la courbe de consommation du consommateur sous forme d'un tableau numpy contenant une valeur par pas de temps (pour un producteur, remplir avec des valeurs négatives). Le tableau doit contenir exactement  $N_{step}$  valeurs.
- `def _get_decisions(self, calculationParams : CalculationParams, variables : List[float]) -> np.ndarray:` Cette fonction prend en paramètre les variables d'optimisations qui ont été calculées par le solveur et doit retourner les décisions prises à chaque instant quand au contrôle du consommateur sous forme d'un tableau numpy contenant une valeur par pas de temps. Le tableau doit contenir exactement  $N_{step}$  valeurs.
- (optionnel, sinon définir la variable `has_base_consumption` a `False`) `def _get_base_consumption(self, calculationParams : CalculationParams) -> np.ndarray:` Cette fonction prend en paramètre les paramètres de simulation et retourne un tableau numpy qui contient un point par pas de simulation. Cette consommation correspond à la consommation non déplaçable du consommateur.
- les variables `id`, `is_reoccurring` et `has_base_consumption` : `id` est l'id de la machine, ressorti dans le résultat tel quel, `is_reoccurring` n'est plus utilisé et `has_base_consumption` sert pour définir une consommation de base au consommateur (consommation non flexible). Attention la consommation de base n'est utilisé que dans les résultats pas dans le calcul dans la version 1.0 !!!

### 3.4 Exemple d'utilisation du module solution sans le reste de l'EMS

to be done, voir section 5

## 4 Comment modifier la source des prévisions d'entrée ELFE

Les prévisions sont faites par persistance dans le service power\_prediction en suivant l'algorithme suivant :

```
def persistance_prediction (curve : List[int]) -> List[int]:
    #il s'agit d'une fonction (persistance_prediction) qui prend en parametre une liste d'entiers appelee curve
    #qui correspond a la liste des valeurs de puissance sur les dernieres 24h
    #dans l'ordre croissant des dates :
    #l'indice 0 correspond a la donnee d'il y a 24h,
    #l'indice 1 a celle d'il y a 23h45 etc...
    liste_des_predictions : List[int] = [] # on initialise la liste des valeurs de la prediction comme une liste vide

    valeur_actuelle : int = curve[-1]#valeur_actuelle est un entier qui contient la prevision en se basant sur la variation observee 24h avant

    valeur_precedente : int = curve[0]
    #On initialise la valeur precedente a la derniere valeur de la courbe
    #cette valeur va permettre de calculer la variation au fur et a mesure

    nombre_de_points_de_la_courbe : int = len(curve)#on stocke le nombre de points de la courbe dans une variable

    for i in range(nombre_de_points_de_la_courbe):

        valeur_actuelle += curve[i] - valeur_precedente #on ajoute la variation de production entre le point 24h avant la prevision et
        #celui de 24h15 avant
        #l'exception c'est le premier point ou i = 0 et valeur_precedente vaut aussi curve[0], ce qui correspond
        #a recopier la derniere consommation observee,
        #sinon valeur_precedente vaut curve[i-1] a chaque instant

        valeur_precedente = curve[i] #On stocke la valeur de consommation qu'on vient d'observer pour l'iteration suivante

        prediction : int = valeur_actuelle * (nombre_de_points_de_la_courbe - i) / nombre_de_points_de_la_courbe
        + curve[i] * i / nombre_de_points_de_la_courbe
        #la prediction est la moyenne ponderee entre la valeur calculee par variation precedemment et la valeur observee 24h avant la prediction
        #plus on est proche du point qu'on connait, plus le mecanisme de variation a une influence
        #plus on en est loin, plus on se fie au point observee
        #par exemple la prediction de dans 30 min sera composee a 95/96 du mecanisme de variation et 1/96 de la valeur observee il y a 23:30
        valeur_actuelle = prediction
        #on utilise la derniere prediction comme base pour le mecanisme de variation de la prochaine iteration
        liste_des_predictions.append(prediction)
        #on ajoute la prevision actuelle a la fin de la liste des previsions
    # on renvoie la liste de prediction pour un autre programme
    # cet autre programme va mettre cette liste 2x bout a bout pour construire une prediction sur 48 h au cas ou
    #mais seule la premiere prevision devrait etre utilisee
    return liste_des_predictions
```

Les données d'entrée de cet algorithme sont celles de la courbe "Equilibre General P=C bis" du projet ELFE, qui est historisée dans l'outil Zabbix.

Afin de changer la source de données dans le service power\_prediction, voici deux scénarios possibles :

Le premier est de garder le code de power\_prediction, mais d'en changer les données d'entrées, et par exemple d'arrêter de faire appel à la persistance\_prediction définie ci-dessus. Dans ce cas power\_prediction doit :

- récupérer les données des sources pour sa prévision
- faire une prévision au pas de temps du quart d'heure
- historiser cette prévision dans la base de données

Le second est d'avoir un nouveau service par division de la production et de n'utiliser le service power\_prediction que pour assembler les résultats. Dans ce cas les courbes de données intermédiaires doivent être historisées dans la base de données interne de l'EMS, au moyen notamment des types défini dans database.history\_db\_types.

Voici des outils de l'EMS qui peuvent être utiles : -Les classes annotées serializableThroughDatabase qui sont dotées des méthodes suivantes :

- get\_create\_table\_str(name): qui permet de récupérer la requête SQL qui permet de créer une table de nom name qui contient les bons champs pour pouvoir stocker une instance de la classe
- get\_append\_in\_table\_str(self, name): qui génère la requête pour pouvoir ajouter l'objet actuel dans la table de nom name
- get\_update\_in\_table\_str(self, name): qui génère la requête pour pouvoir modifier la sauvegarde de l'objet actuel dans la table de nom name
- get\_create\_or\_update\_in\_table\_str(self, name): qui génère la requête pour pouvoir sauvegarder l'objet actuel dans la table de nom name, quitte à créer une nouvelle ligne si il n'existe pas d'élément qui partage une clef primaire l'objet actuel.

- `create_from_select_output(output)`: qui crée un objet en se basant sur le résultat d’une requête de type ”SELECT \* FROM nom.table”

-Les méthodes `execute_queries`/`fetch` du module `database.query` qui permettent respectivement d’exécuter une requête de modification/exécuter une requête en récupérant les résultats dans la base de données. -le module `zabbix_reader` qui permet de lire des données depuis `zabbix`

## 4.1 Exemples

Un exemple concret d’utilisation des méthodes des classes `serializableThroughDatabase` est la sauvegarde des prévisions de prod-conso dans la table ”prediction” de la base de données de l’EMS:

```
from database.EMS.db_types import EMSPowerCurveData
from database.query import execute_queries
from credentials.db_credentials import db_credentials
#je recopie la definition pour la lisibilite du document, mais ce bloc n'est pas necessaire
@serializableThroughDatabase
@dataclass(init=True, repr=True)
class EMSPowerCurveData():
    data_timestamp : Union[int, create.DB_Annotation(is_primary=True)]
    power : int

for i in range(len(prediction)):
    #on suppose ici que timestamp et prediction sont rempli respectivement par les dates et les predictions de prod-conso en W a ce stade
    curve_data = EMSPowerCurveData(timestamp[i], prediction[i])#exemple legerement modifie dans un soucis de clarete
    data_to_post_to_database.append(curve_data.get_create_or_update_in_table_str(CURVE_DATA_TABLE))#CURVE_DATA_TABLE = "prediction"
execute_queries(db_credentials["EMS"], data_to_post_to_database)
```

Figure 13: exemple d’envoi de données dans la table prediction de l’EMS

Voici un exemple de l’utilisation du module `zabbix_reader` pour récupérer les données de la journée précédente :

```
from learning.zabbix_reader import ZabbixReader
from credentials.zabbix_credentials import zabbix_credentials
from learning.curve import get_full_curve_snapped
from datetime import datetime

CURVE_KEY="Equilibre General P=C bis"
CURVE_PERIOD= 15*60 #on vise une periode du quart d'heure
DAY_DURATION=24*60*60 # duree d'une journee en heures

zr = ZabbixReader("http://mqtt.projet-elfe.fr/api_jsonrpc.php", zabbix_credentials["username"], zabbix_credentials["password"])
zr.get_token()

items = zr.get_items()#on liste les itemid des courbes qui peuvent etre utiles a l'EMS.
#Pour ajouter une courbe qui n'est pas dans celles-ci on devra modifier la fonction get_items
item_id = items[CURVE_KEY]#on recupere l'id zabbix de la courbe
current_timestamp = int(datetime.now().timestamp()) #on recupere la date actuelle
current_timestamp = round(current_timestamp/CURVE_PERIOD) * CURVE_PERIOD #on arrondi la date actuelle au quart d'heure
data = zr.readData(item_id, current_timestamp - DAY_DURATION, current_timestamp)#on recupere les donnees entre hier et maintenant
base_timestamp = data["timestamps"][0] - (data["timestamps"][0] % CURVE_PERIOD) #on recupere le timestamp au quart d'heure
#de la premiere donnee disponible
curve = get_full_curve_snapped(data["timestamps"], data["values"], CURVE_PERIOD, current_timestamp - DAY_DURATION)
#on recupere la courbe moyennée sur un quart d'heure
curve_base_timestamps = [base_timestamp + i * CURVE_PERIOD for i in range(len(curve)) ]# on genere les timestamps associes
```

Figure 14: récupération d’une courbe depuis `zabbix`

```

from database.EMS_db_types import EMSPowerCurveData
from database.query import execute_queries
from credentials.db_credentials import db_credentials
from datetime import datetime
from psycopg2 import sql

#je recopie la definition pour la lisibilite du document, mais ce bloc n'est pas necessaire
@serializableThroughDatabase
@dataclass(init=True, repr=True)
class EMSPowerCurveData():
    data_timestamp : Union[int, create_DB.Annotation(is_primary=True)]
    power : int

CURVE_DATA_TABLE = "prediction"
current_timestamp = int(datetime.now().timestamp()) #on recupere la date actuelle
current_timestamp = round(current_timestamp/CURVE_PERIOD) * CURVE_PERIOD #on arrondi la date

query = (sql.SQL("SELECT * FROM {0} WHERE data_timestamp >= %s").format(
    sql.Identifier(CURVE_DATA_TABLE)
), [current_timestamp])

result = fetch(db_credentials["EMS"], query) #on recupere les donnees depuis la base de donnees
curve_formated = [EMSPowerCurveData.create_from_select_output(i) for i in result] # on converti le resultat sous forme de
#liste de points EMSPowerCurveData qu'on peut ensuite modifier et renvoyer sur la table souhaitee

```

Figure 15: récupération d’une courbe depuis la base de données

## 5 Comment relancer un scénario passé ?

On récupère un des fichier qui a été généré par l’EMS dans le dossier data/run\_conditions. Ces fichiers sont nommés sous le format TimestampDeLancement\_TimestampDeValidite\_NombreDeTypesEnlevePourCauseDeCrash.py. Cela se présente sous la forme suivante :





```
problem : Problem = Problem(consumers, sim_params)
problem.prepare()
problem.solve(15*60)
```

Figure 18: Code de lancement d'un problème a partir des conditions initiales

Et ensuite on peut traiter les résultats en faisant appel aux fonctions de la classe `Probleme` comme par exemple `problem.get_consumption` ou `problem.get_decisions`. Pour le lancer, il faut se placer dans la racine de l'EMS, se mettre dans l'environnement conda "milp" et lancer le fichier :

```
$>conda activate milp
$>python nom_du_fichier
```

Voici un fichier complet :

[illegible]