



Ingénierie Dirigée par les Modèles

Projet IDM : chaiseMinute

Groupe L34-2

Élèves :

LEBOBE Timothé

LECUYER Simon

SABLAYROLLES Guillaume

THEVENET Louis

Novembre 2024 - Decembre 2024

Table des matières

1. Introduction	3
2. Méta-Modèles	3
2.1. ChaiseMinute	3
2.2. Algorithm	4
2.3. Function	4
2.4. Calculus	5
3. Transformations Texte à Modèle de ChaiseMinute	6
3.1. Syntaxe Textuelle	6
3.2. Transformation M2M	6
4. Transformations M2T	6
4.1. Librairie et script de calcul automatique	6
4.2. Outil de visualisation des données	7
5. Transformations Modèle à Modèle	7
6. Edition graphique	7
6.1. Limitation	8
7. Exemples	8
7.1. Equation du second degré	8
8. Conclusion	12
9. Annexes	12

1. Introduction

Ce projet consiste en la réalisation d'un environnement de calcul Domaine-Scientifique nommé : ChaiseMinute.

ChaiseMinute permet à tous ses utilisateurs de développer des schémas de tables et des librairies pour traiter automatiquement des données respectant ces schémas. Ainsi l'utilisateur pourra produire des outils pour d'autres utilisateurs finaux souhaitant manipuler des données sans avoir à créer leur propres outils.

L'utilisateur de ChaiseMinute dispose d'une syntaxe textuelle ainsi qu'un outil graphique permettant de définir ses schémas de tables. Les schémas de tables pourront être transformés dans des langages plus spécifiques au calcul pour pouvoir transformer et vérifier des données.

2. Méta-Modèles

Nous avons décidé de séparer le modèle en trois sous-méta-modèles :

1. ChaiseMinute.Ecore
 - Point d'entrée, il permet de définir les schémas de table
2. Algorithm.Ecore
 - Définir les algorithmes utilisés pour les ComputedColumns et les Constraints
3. Function.Ecore
 - Pour spécifier les Functions utilisées par les Algorithm

2.1. ChaiseMinute

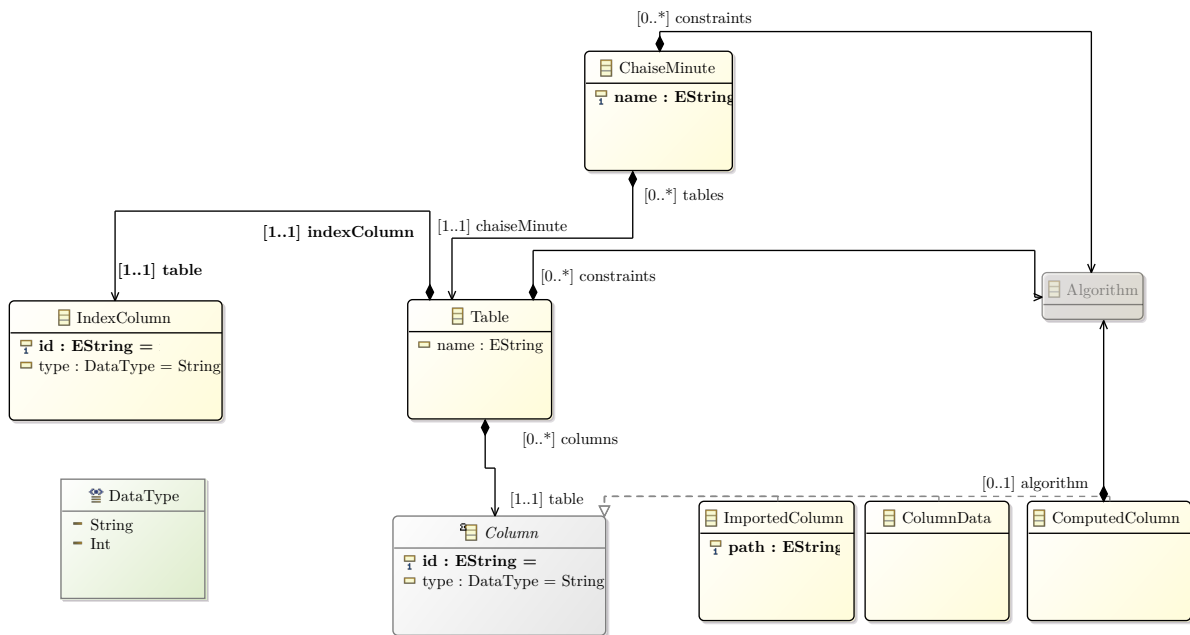


Fig. 1. – ChaiseMinute Ecore Diagram.

- ChaiseMinute représente le container permettant de regrouper les schémas de tables entre eux.
- Table défini un schéma de table comme un ensemble de Column et d'une IndexColumn, des contraintes peuvent également être ajoutées pour vérifier les données d'entrée
- IndexColumn : est une colonne spéciale pour les index des lignes
- Column est une super-classe représentant une colonne, elle contient un type de données et d'éventuelles contraintes sur les données d'entrée ou de sortie
 - ColumnData : une colonne simple pour représenter une donnée,

- **ImportedColumn** : une colonne provenant d'une autre **Table**,
- **ComputedColumn** : représentant une colonne calculée avec un **Algorithm**

Toutes les contraintes sont représentées par des algorithmes qui renvoient un booléen.

2.2. Algorithm

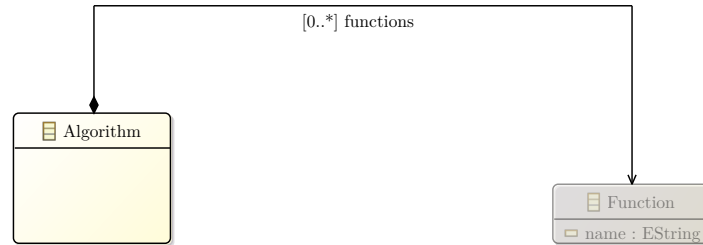


Fig. 2. – Algorithm Diagram.

Algorithm représente le point d'entrée d'un algorithme, il contient les fonctions qui seront appliquées.

L'application des fonctions respecte les règles suivantes :

- La première fonction est appliquée à ses arguments déclarés dans le modèle
- Les fonctions suivantes reçoivent en premier argument la sortie de la fonction précédente, puis leur arguments déclarés

Ces règles nous permettent de chaîner les fonctions dans un algorithme.

2.3. Function

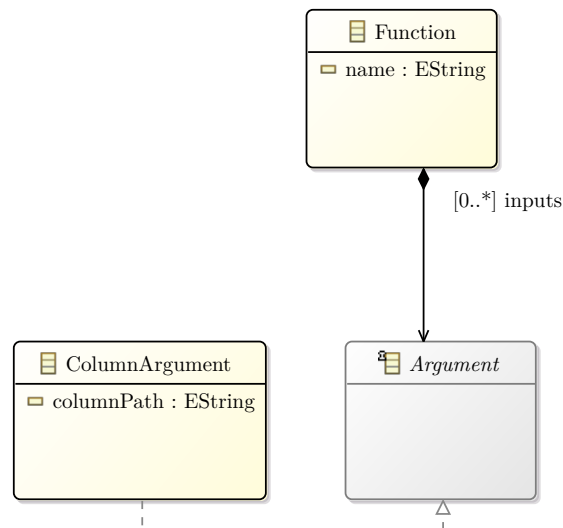


Fig. 3. – Algorithm Diagram.

Une **Function** est représentée par un identifiant qui référence un programme Python, elle contient des arguments, qui sont des références vers des colonnes. Les colonnes sont représentées sous la forme `nomTable.nomColonne` dans tout le projet (pour les références croisées de Fig. 1, les arguments de fonctions, etc).

2.3.1. Limitations

On aurait pu faire en sorte que les **Function** soient des arguments, ainsi on aurait pu construire un arbre d'appels et prendre la sortie de plusieurs fonctions à la fois en arguments. On peut quand même obtenir ce résultat avec le système actuel en adaptant les fonctions Python pour qu'elles renvoient plusieurs colonnes, ainsi en adaptant la fonction suivante pour qu'elle récupère ces deux colonnes, on imite le fonctionnement d'un arbre d'appels.

2.4. Calculus

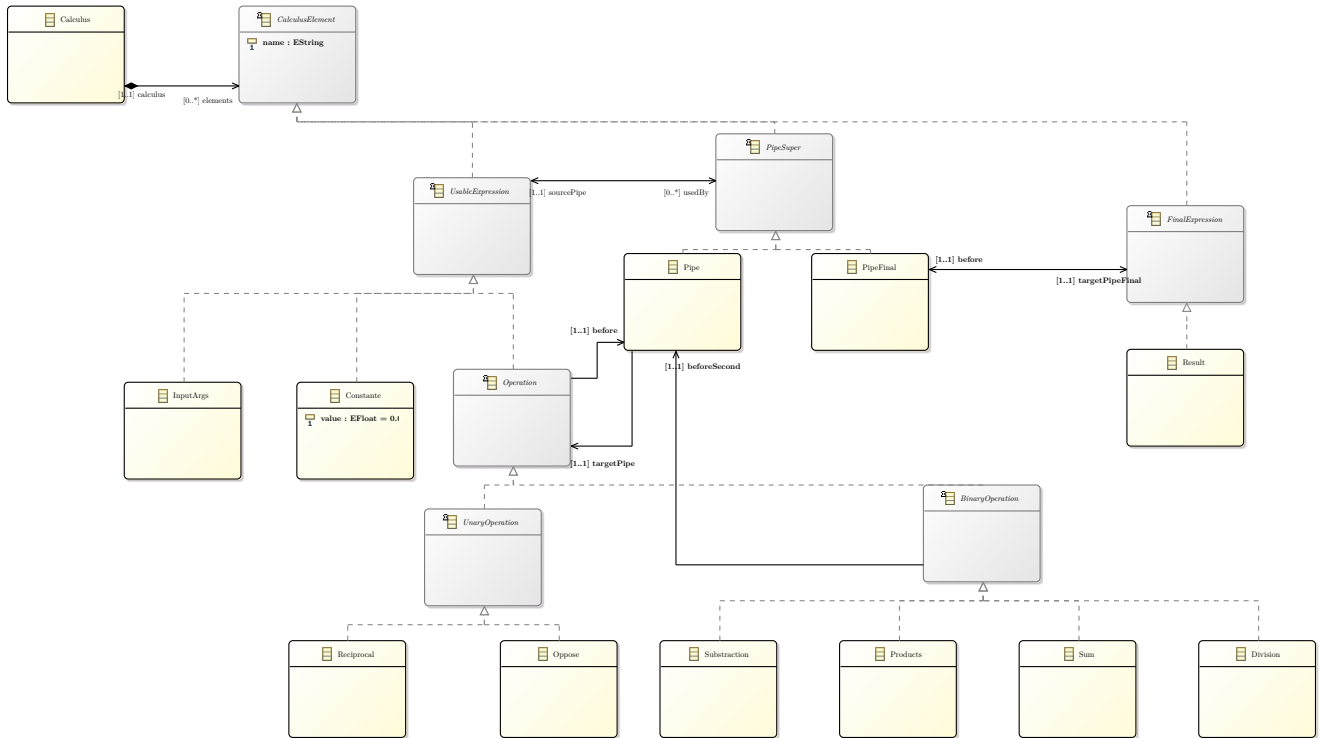


Fig. 4. – Calculus Diagram.

Calculus représente le méta-modèle des fonctions/calculs éditables à l'aide d'une syntaxe graphique par un utilisateur. Un Calculus représente un ensemble de CalculusElement qui peuvent se décomposer en trois grandes parties : UsableExpression, PipeSuper et FinalExpression.

- UsableExpression décrit les expressions pouvant être appelées pour réaliser le calcul :
 - InputArgs pour les arguments en entrée du Calculus,
 - Constante pour décrire les constantes pouvant être écrite en dure dans les suites de calculs. Ces deux premiers ne peuvent que fournir des valeurs à l'intérieur du Calculus et donc n'ont pas d'expression en entrée,
 - Operation décrit une opération possible à l'intérieur du calcul, pouvant être à deux entrées (BinaryExpression) ou à une seule entrée (UnaryExpression)
- PipeSuper représente les liens entre les UsableExpression pour décrire l'ordre des calculs et l'ordre des appels :
 - Pipe pour décrire les liens, les entrées et les sorties, entre les calculs,
 - PipeFinal indique que les calculs redirige vers la sortie finale, la donnée retournée du Calculus,
- FinalExpression indique la fin du calcul.

2.4.1. Limitations

Nous avons donné des opérations de calcul « classique » utilisable directement par l'utilisateur : `Sum`, `Substraction`, `Products`, `Division`, `Oppose` et `Reciprocal` mais nous aurions pu rajouter des opérations mathématiques plus avancées (`sin`, `cos`, `exp`, `modulo`, ...) ou juste définir des opérations unaire ou binaire en les décrivant par un argument `opération`.

Les `Pipe` sont utiles car il permettent de définir des expressions pendantes non reliées qui sont facilement éditables ensuite mais cela rajoute du poids sur l'architecture du méta-modèles qui nous pourrions corriger dans une version future.

3. Transformations Texte à Modèle de ChaiseMinute

3.1. Syntaxe Textuelle

En utilisant le langage Xtext, nous avons pu définir une syntaxe textuelle pour les schémas de tables comme suit :

```
1  nomSchemaTable {
2    nomTable1 (
3      nomColonne1 of typeColonne1,
4      nomColonne2 of typeColonne2,
5      nomColonne3 of typeColonne3 computed with [
6        nomFonction1("nomTable1.nomColonne1", "nomTable1.nomColonne2") >
7        nomFonction2(nomTable1.nomColonne2)
8      ]
9    )
10   constrained by [nomFonction3("nomTable1.nomColonne1", "nomTable1.nomColonne2")]
11
12   nomTable2 (
13     nomColonne1 of typeColonne1 imported from nomTable1.nomColonne1
14   )
15 }
16 constrained by [nomFonction4("nomTable2.nomColonne1" "nomTable1.nomColonne2")]
```

La syntaxe permet de définir des modèles de Table dans un méta-modèles proche de `ChaiseMinute` mais néanmoins différent : `TabouretSeconde`. En effet, Xtext ne supportant pas certaines spécificités des ecore, comme les e-references, il est nécessaire d'effectuer une transformation M2M pour obtenir un modèle de `ChaiseMinute` à partir d'un modèle conforme à `TabouretSeconde`.

3.2. Transformation M2M

Pour finaliser la transformation texte à modèle, nous avons écrit une transformation modèle à modèle en `Acceleo` (la menuiserie) qui consiste simplement à rétablir les e-references et avoir un modèle conforme à `ChaiseMinute`.

4. Transformations M2T

4.1. Librairie et script de calcul automatique

Nous avons décidé d'utiliser le langage Python pour nos algorithmes. Ainsi, pour générer une librairie de calcul à partir d'un schéma de table, il nous suffit de générer un programme Python qui appelle les fonctions référencées par les algorithmes et les contraintes.

La librairie propose différentes fonctions pour manipuler des données à l'aide du schéma de table.

```

1  def load(files):
2      """Load all tables as CSV files. All tables must have a corresponding CSV file
3      with the same name in working dir."""
4  def check_constraints(input):
5      """Ensure all constraints are respected in input and output data"""
6
7  def generate_output(input):
8      """Returns a Dict<TableName, Dict<ColumnName, Data>> by applying the the model
9      on the input data."""
10
11 def save_to_csv(tables):
12     """Saves each table from he input argument as a CSV file."""
13
14 def main():
15     input = load(sys.argv[1:])
16     out = generate_output(input)
17     res, msg = check_constraints(input)
18     print(msg)
19     save_to_csv(out)
20     print("Exported files")

```

Liste 1. – Signatures des fonctions de la librairie

Nous générons aussi une fonction `main` dans notre librairie qui prend en entrée des fichier au format CSV et qui en génèrent de nouveaux en appliquant le schéma de table, résultant en un script de transformation automatique des données.

4.2. Outil de visualisation des données

En s'appuyant sur la même librairie, on crée un outil de visualisation du schéma de table en Python. (Voir Fig. 7)

5. Transformations Modèle à Modèle

6. Edition graphique

Nous avons développé un outils graphique permettant de modifier des fichiers `.cml` (ChaiseMinute) pour modifier les différents schémas de tables et obtenir une visualisation plus pratique pour l'utilisateur.

Les Tables et les Columns sont visualisées comme des `containers`, des boîtes, pour montrer l'imbrication des Columns dans les Tables et la fraternités des Columns.

- Les Tables sont représentées par des `containers` verts clairs,
- Les IndexColumn sont représentées par des `containers` bleus clairs,
- Les DataColumn sont représentées par des `containers` rouges clairs,
- Les ImportedColumn sont représentées par des `containers` violets clairs,
- Les ComputedColumn sont représentées par des `containers` jaunes,
- Les Algorithm sont représentés par des `containers` marrons dans les ComputedColumn
 - Les informations dans les `containers` Algorithm affichent le nom des Fonctions qu'ils utilisent.

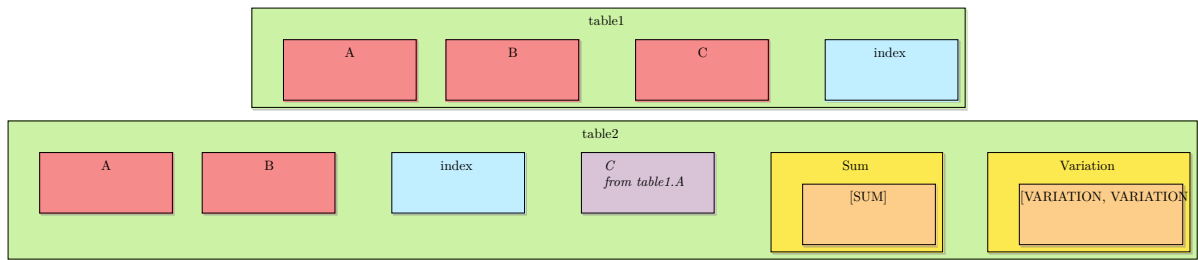


Fig. 5. – Sirius de ExempleComplice.cml.

6.1. Limitation

Il est possible pour l'utilisateur de rajouter des fonctions utiles pour une `ComputedColumn`. Cependant nous avons rencontré des difficultés à choisir des fonctions inutile. En effet, nous ajoutons et enlevons les fonctions en écrivant leur chemin dans une boîte de dialogue texte mais pour l'enlever nous n'avons pas réussi à utiliser la valeur renvoyée pour vérifier si elle correspondait à une `Function` présente et donc la supprimer en conséquence.

7. Exemples

7.1. Equation du second degré

7.1.1. Modèle `TabouretSeconde` (syntaxe textuelle)

On utilise un schéma de table qui représente des polynômes du second degré dont on cherche les racines. Les données d'entrée sont un fichier CSV qui contient une colonne par coefficient (a , b , c).

```

1  exemple_equations {
2      equations indexed on Index of Int(
3          Solution of String computed with [
4              compute_delta("equations.a","equations.b", "equations.c")
5              > solve("equations.a","equations.b", "equations.c")
6          ],
7      delta of Int computed with [
8          compute_delta("equations.a","equations.b", "equations.c")
9      ],
10     a of Int imported from "table.a",
11     b of Int imported from "table.b",
12     c of Int imported from "table.c"
13 )
14 }
15 constrained by [ensure_is_not_null("table.a") ]

```

Liste 2. – Description textuelle du schéma de table

Les colonnes a , b et c sont importées de la table d'entrée.

Le schéma de table spécifie deux nouvelles colonnes `Solution` et `delta` qui représentent les racines et les déterminants de chaque polynôme. Elles sont produites à partir des fonctions `compute_delta` et `solve`.

```

1  import numpy as np
2
3  def compute_delta(a,b,c):
4      return np.multiply(b,b) - 4 * np.multiply(a, c)

```



```

1 import numpy as np
2 import math
3
4 def solve(deltas, a,b,c):
5     solutions = []
6     for i in range(len(a)):
7
8         if deltas[i] > 0:
9             sqrt_delta = math.sqrt(deltas[i])
10            solutions.append([
11                (-b[i] - sqrt_delta)/(2*a[i]),
12                (-b[i] + sqrt_delta)/(2*a[i])
13            ])
14        elif deltas[i] < 0:
15            solutions.append(["No solution"])
16        else:
17            solutions.append([-b[i] / (2*a[i])])
18    return solutions

```

Finalement, une contrainte est vérifiée sur la colonne a , dont tous les éléments doivent être non nuls.

```

1 import numpy as np
2
3 def ensure_is_not_null(x):
4     return (np.array(x) != 0).all()

```

7.1.2. Modèle ChaiseMinute

On transforme ensuite ce modèle intermédiaire en ChaiseMinute par notre transformation ATL `cmtToCm`

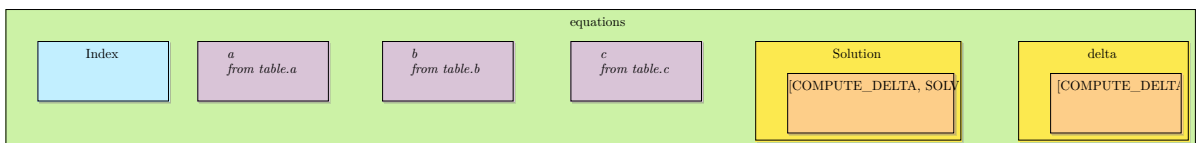


Fig. 6. – Modèle chaiseMinute obtenu vu sous Sirius

7.1.3. Génération de la librairie Python

On génère ensuite la librairie via la transformation modèle vers texte, un extrait est disponible ci-dessous.

La fonction `generate_output` applique le schéma de table à partir des données chargées en entrée.

```

1 def generate_output(input):
2     """Returns a Dict<TableName, Dict<ColumnName, Data>> by applying the the model
3     on the input data."""
4     tables = {}
5     out = {}
6     #####
7     # Table: equations
8     #####
9     ### Imported column: a from table.a ###
10    out["a"]=input["table"]["a"]
11
12    ### Imported column: b from table.b ###

```

```

12 out["b"]=input["table"]["b"]
13
14 ### Imported column: c from table.c ###
15 out["c"]=input["table"]["c"]
16
17 #####
18 ## Computed column: Solution
19 #####
20 ### Apply compute_delta ##
21 from compute_delta import compute_delta
22 out["Solution"] = compute_delta(
23     search(input, out, "equations", "a"),
24     search(input, out, "equations", "b"),
25     search(input, out, "equations", "c"),
26 )
27 #####
28 ### Apply solve ##
29 from solve import solve
30 out["Solution"] = solve(
31     out["Solution"], # Previous result used in next function
32     search(input, out, "equations", "a"),
33     search(input, out, "equations", "b"),
34     search(input, out, "equations", "c"),
35 )
36 #####
37 #####
38 ## Computed column: delta
39 #####
40 ### Apply compute_delta ##
41 from compute_delta import compute_delta
42 out["delta"] = compute_delta(
43     search(input, out, "equations", "a"),
44     search(input, out, "equations", "b"),
45     search(input, out, "equations", "c"),
46 )
47 #####
48 tables["equations"] = out
49
50 return tables

```

La fonction `check_constraints` applique les contraintes et vérifie la cohérence des types.

```

1 def check_constraints(input):
2     """Ensure all constraints are respected in input and output data"""
3     #####
4     # Verifying Input Constraints
5     #####
6     ### Apply ensure_is_not_null ##
7     from ensure_is_not_null import ensure_is_not_null
8     res = ensure_is_not_null(
9         input["table"]["a"].to_list(),
10    )
11
12    if not res:
13        return (False, ("ensure_is_not_null constraints failed"))
14
15    #####
16    # Verifying Table Constraints
17    #####
18    out = generate_output(input)
19
20    ### Verify data types

```

```

21 for x in out["equations"]["Solution"]:
22     break
23 for x in out["equations"]["delta"]:
24     if (type(x)!=int and type(x)!=numpy.int64):
25         return (False, "Type constraints failed on equations.delta")
26
27 for x in out["equations"]["a"]:
28     if (type(x)!=int and type(x)!=numpy.int64):
29         return (False, "Type constraints failed on equations.a")
30
31 for x in out["equations"]["b"]:
32     if (type(x)!=int and type(x)!=numpy.int64):
33         return (False, "Type constraints failed on equations.b")
34
35 for x in out["equations"]["c"]:
36     if (type(x)!=int and type(x)!=numpy.int64):
37         return (False, "Type constraints failed on equations.c")
38
39
40 return (True, "Constraints are respected")

```

7.1.4. Application de la librairie

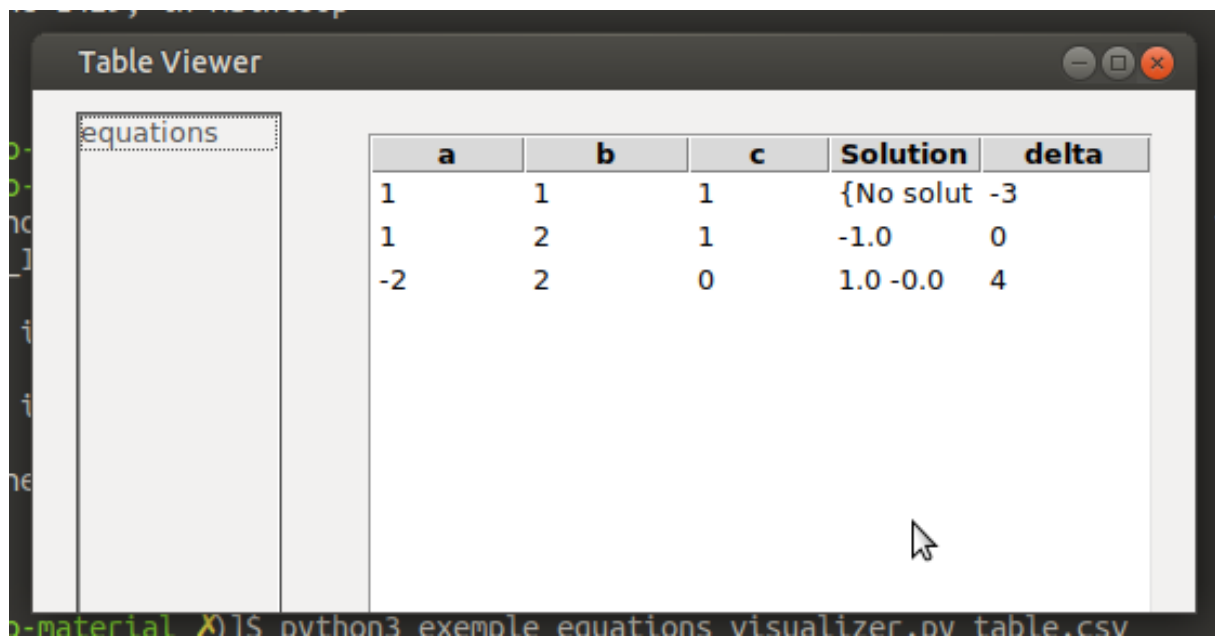
On peut finalement appeler le script automatique (fonction `main` de la librairie), qui à partir des données d'entrée au format CSV :

Index	a	b	c
0	1	1	1
1	1	2	1
2	-2	2	0

Produit le fichier :

Index	a	b	c	Solution	del- ta
0	1	1	1	['No solution']	-3
1	1	2	1	[-1.0]	0
2	-2	2	0	[1.0, -0.0]	4

On peut également voir les données dans l'outil de visualisation généré



a	b	c	Solution	delta
1	1	1	{No solut	-3
1	2	1	-1.0	0
-2	2	0	1.0 -0.0	4

Fig. 7. – Visualisation des données générées

8. Conclusion

9. Annexes