Projet Informatique – Sections Electricité et Microtechnique

Printemps 2025 : Linked-Crossing © R. Boulic & collaborateurs

Rendu1 (dimanche 30 mars 23h59)

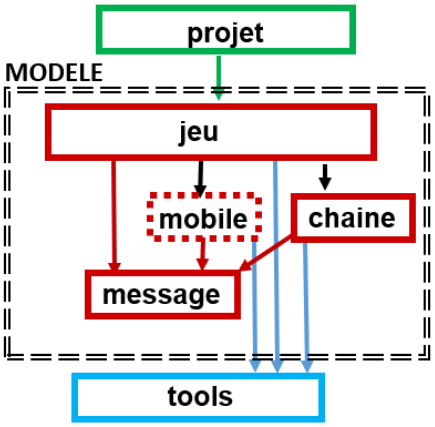
Table des matières :

Objectif de ce document : *Ce document utilise l'approche introduite avec la série théorique sur les méthodes de développement de projet qu'il est important d'avoir faite avant d'aller plus loin.*

En plus de préciser ce qui doit être fait, ce document identifie des ACTIONS à considérer pour réaliser le rendu de manière rigoureuse. Ces ACTIONS sont équivalentes à celles indiquées pour le projet d'automne ; elles ne sont pas notées, elles servent à vous organiser. Vous pouvez adopter une approche différente du moment que vous respectez l'architecture minimale du projet (donnée Fig 11a).

1. Buts du rendu1 : architecture générale

Le *premier* objectif est de disposer d'un module **tools** dont chaque fonction est validée par des tests extensifs (avec du *scaffolding et un test unitaire de ce module*). Le but est de disposer d'un outil stable pour les rendus suivants. Cela implique de définir en priorité les structures de données, assez simples, de **tools** puis les fonctions que ce module va offrir dans son interface. Nous demandons d'y créer le type **S2d** (Donnée section 7.3.3) et la structure de données d'un cercle.



Donnée Fig 11a

N'utilise PAS GTKmm

C'est seulement après la validation du module **tools** que vous pourrez aborder le second objectif d'ébauche du **Modèle** que vous voyez dans la boite en pointillés au dessus du module **tools** à droite. Ce Modèle doit contenir l'ensemble des modules et des dépendances visibles dans la figure ci-contre.

Nous imposons que les modules du **Modèle** mettent en œuvre des **classes** en respectant le *principe d'encapsulation*, ce qui veut dire que les *attributs* seront **private** et qu'il faudra utiliser des *méthodes* pour y accéder. A part cette contrainte stricte, nous acceptons pour ce premier rendu que votre choix de structure de données soit une première ébauche qui pourra être remise en question pour les rendus suivants.

Pour le rendu1, le module de plus haut niveau **projet** contient seulement la fonction **main** : sa tâche est de récupérer le *nom de fichier de test* transmis sur la ligne de commande au moment du lancement de l'exécutable. Ce nom de fichier doit être immédiatement transmis à une fonction ou méthode du module **jeu** qui agit comme point d'entrée du **Modèle**.

Les responsabilités des modules du Modèle (Donnée section 7.2) sont complétées ici :

- **jeu** : c'est le SEUL point d'entrée du Modèle vis-à-vis de l'extérieur (module **projet** pour le rendu1)
  - il gère au plus haut niveau d'abstraction les tâches de (*une*) mise à jour du jeu, dessin, lecture et écriture de fichier. Pour le **rendu1**, on demande seulement de mettre au point l'action de **lecture** d'un fichier pour initialiser une instance de jeu en détectant des erreurs indiquées en section 4.2 de la donnée générale. D'une manière générale, il faudra passer par une méthode du module **jeu** lorsque de l'information devra être échangée entre les entités de **mobile** et celle de **chaine**.
  - en vertu du principe d'abstraction, ce module délègue l'exécution des sous-problèmes qui peuvent être résolus de manière autonome par les modules **mobile et chaine** (cf Fig11a).

- **mobile**: pour le rendu final il est demandé de mettre en place *une hiérarchie de classes* pour gérer les 2 types d'entités particule et faiseur. Cependant, pour le **rendu1**, une approche simplifiée sera acceptée soit avec un attribut de *type* dans une classe unique **mobile**, soit avec deux classes indépendantes **particule** et **faiseur**.
- **chaine** : ce module définit une classe **chaine** dont un seul exemplaire sera utile pour le jeu ; cela dit il peut être pratique de disposer en interne de plusieurs instances pour les algorithmes manipulant une chaine.
- Le module **message** est *fourni* pour l'affichage de messages standardisés pour la tâche de lecture du Modèle. Il ne faut pas le modifier.

La section suivante précise ce qui est demandé pour le module **tools** et comment le tester.

2. But du module tools

Comme le rappelle l'architecture du projet, ce module doit rester totalement indépendant des concepts du Modèle ; les noms des entités du Modèle ne doivent pas apparaître dans **tools** de même que **constantes.h** ne doit pas être inclus dans **tools** non plus. A la place ce module doit offrir des noms les plus génériques possibles dans l'idée qu'ils pourront servir à d'autres application (Principe de ré-utilisabilité).

Usage de S2d: en particulier, le module **tools** utilise le type **S2d** présenté dans la section 7.3.3 de la donnée générale pour placer un point P(x,y) par rapport à l'origine O du plan ou définir un vecteur v(x,y) dans le plan. On travaille dans un espace continu en virgule flottante double précision selon les conventions d'axes visibles à droite.

Types plus avancés: pour ce projet il sera nécessaire de travailler avec des bipoints en coordonnées cartésiennes ou polaires (avec l'angle en radian $\in [-\pi, \pi]$ par rapport à l'axe X). Vous devez utiliser **S2d** pour construire ces types plus avancés. De même vous devez utiliser la fonction **atan2** de C++ pour obtenir très facilement l'angle α en rd à partir d'un vecteur v(x,y). Pas de problème pour l'usage de **struct** à ce bas-niveau. Vous pouvez définir et nommer vos propres types de données pour pouvoir réaliser les actions suivantes :

- passer d'une représentation cartésienne à une représentation polaire et inversement
- calcul de norme d d'un vecteur v(x,y), etc...

Représentation d'un cercle : ajouter un type permettant de représenter un cercle dans le plan avec les fonctions de test d'inclusion/d'intersection qui seront utiles ensuite au niveau du Modèle (section 3.4 de la donnée générale).

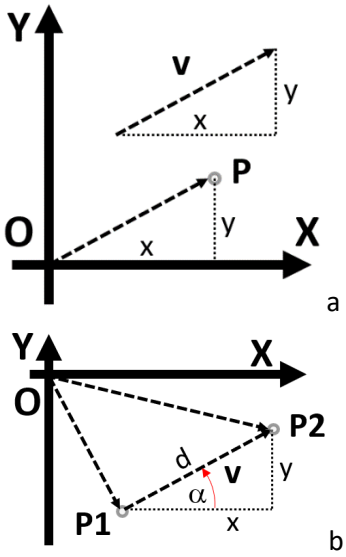


Fig rendu1.1 : (a) conventions d'axes de coordonnées X et Y et usage de S2d pour un point P ou un vecteur v. (b) bipoint en représ. cartésienne avec P1 et P2 ou P1 et v, et représ. polaire avec P1 et (α,d)

Calculs vectoriels avancés : Enfin, un autre ensemble de fonctions sera nécessaire pour effectuer les calculs vectoriels donnant le vecteur réfléchi à partir du vecteur incident en un point à l'intérieur d'un cercle (Fig 2 donnée générale). Les figures suivantes (**rendu1.2**) illustrent deux cas d'un tel calcul pour lesquels la même formule de calcul angulaire est valable.

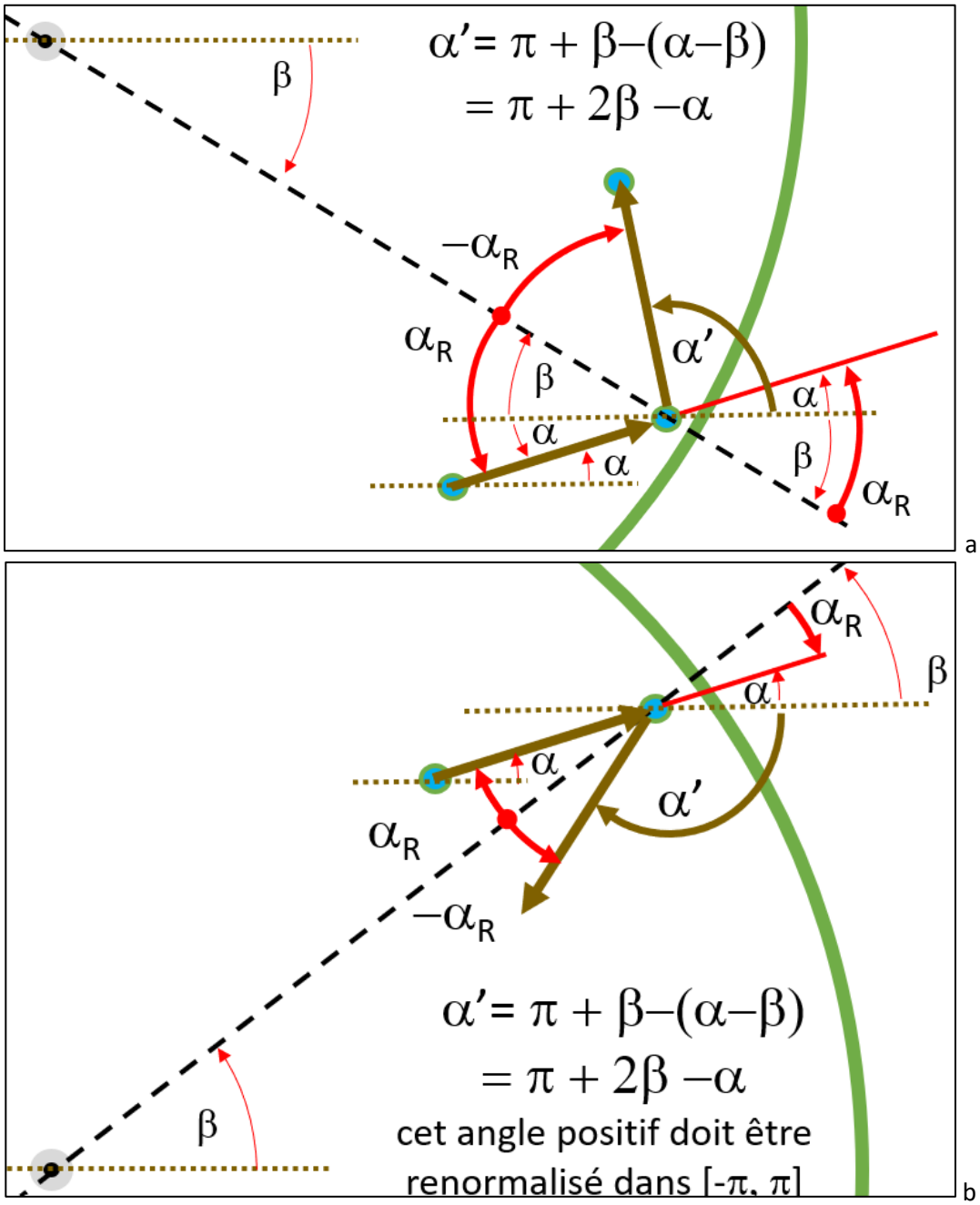


Fig rendu1.2 : (a) calcul du vecteur réfléchi d'angle α' à partir du vecteur incident d'angle α et du vecteur reliant le centre du cercle au point de réflexion, d'angle β. Les angles sont signés. (b) second cas ; attention : il faut toujours renormaliser le résultat d'un calcul angulaire entre $[-\pi, \pi]$

Justification : ce type de calcul est nécessaire dès le rendu1 pour effectuer certains des tests demandés à la lecture de fichier. En effet, le format du fichier pour le type faiseur n'indique que les données de sa tête. Or, pour obtenir la position des autres éléments circulaires du faiseur, il faut construire les positions précédentes qu'a prises la tête du faiseur. *Ces positions sont situées aux multiples de la distance exprimée par le déplacement du faiseur mais dans la direction opposée au déplacement du faiseur pendant le jeu.* De plus, si une de ces positions antérieures ne respecte pas la condition d'être à l'intérieur de l'arène, il faut alors calculer la direction réfléchie dans la direction opposée au déplacement pendant le jeu. Les fonctions que vous proposerez devront être paramétrées pour pouvoir être utilisées dans les 2 scénarios d'utilisation au niveau du Modèle (calcul des éléments précédents ou calcul du déplacement du faiseur).

Constante définie par le module tools : seule la constante **epsil_zero** est définie par ce module **tools** pour gérer les variantes de détection de superposition des entités. Les autres constantes sont définies par le Modèle et ne doivent pas apparaître dans **tools**.

3. lecture de fichiers de configuration

3.1 Module projet

Le module **projet** contient la fonction **main()** en charge d'analyser si la ligne de commande est bien conforme à la syntaxe suivante : **`./projet t01.txt`**

Où **`t01.txt`** est un fichier de configuration (Donnée **Section 4**).

Nous ne testerons pas l'absence de l'argument. Pour le rendu1, votre programme doit se terminer en cas d'absence d'argument. Si l'argument est présent nous supposons qu'il correspond à un fichier de test présent dans le répertoire courant. Ce nom de fichier doit être transmis à une fonction ou méthode de **lecture** du module **jeu.**

Deux stratégies sont autorisées concernant *l'instance* de la classe **Jeu** qui pilote le Modèle :

- Soit elle existe dans le module **projet** et une méthode **lecture** est appelée sur cette instance.
- Soit elle est cachée dans le module **jeu** (car elle est unique) et une fonction **lecture** du module **jeu** est appelée sans avoir besoin de préciser en paramètre sur quelle instance elle travaille.

3.2 Liste des erreurs à détecter au niveau du Modèle (cf sections 4.2 de la donnée générale):

Le programme cherche à initialiser l'état du **Modèle** avec les données lues dans le fichier de configuration. Pour le Rendu1 il **s'arrête** dès la ***première*** erreur trouvée dans le fichier en appelant la fonction mise à disposition pour l'affichage du message d'erreur puis on quitte le programme en appelant **exit(EXIT_FAILURE)** comme montré en 3.3.

Le programme **s'arrête** aussi après la lecture du fichier s'il n'y a aucune erreur ; dans ce cas, il faut appeler la fonction **success()** du module message qui affiche un message indiquant le succès de la lecture puis on quitte en appelant **exit(0**).

3.2.1 usage différencié de epsil_zero selon le but des tests (lecture / jeu)

Le choix d'effectuer la sauvegarde du jeu dans un fichier formaté introduit des arrondis sur les valeurs sauvegardées. Ces arrondis nous imposent d'effectuer des tests de collisions *moins stricts quand on lit un fichier* en comparaison de ces mêmes tests *pendant une mise à jour du jeu*.

C'est pourquoi la valeur **epsil_zero** doit être considérée comme nulle pendant les tests de la lecture de fichier du rendu1. Cela veut dire que vos fonctions du module **tools** qui font les tests d'intersection/superposition doivent prendre en compte un paramètre supplémentaire qui active ou désactive l'utilisation de **epsil_zero** dans les tests.

3.2.2 Appartenance des entités l'arène du jeu

Les particules et les articulations de la chaîne sont considérées comme des points (cercle de rayon nul). Un faiseur est considéré comme un ensemble de cercles calculables à partir des informations du fichier. Il n'y a aucun test à faire sur le segment qui relie deux articulations de la chaîne.

3.2.3 Collisions interdites

Les articulations de la chaîne ne doivent pas être incluses dans un élément de faiseur.
Un élément d'un faiseur A ne doit pas intersecter un élément d'un faiseur B.

3.2.4 Autres tests à effectuer

La lecture doit aussi vérifier (section 4.2 de la donnée générale) :
- que le score est dans **]0, score_max]**

- Mobile : **d** compris dans **[0, d_max]**
- Particule : nombre dans **[0, nb_particule_max]** ; compteur **c** dans **[0, time_to_split[**
- Faiseur : intervalle de validité du rayon **r** ; ***nbe*>0**
- Chaine : racine distance intérieure et longueur entre articulations consécutives ≤ **r_capture**

Nous ne testerons pas vos projets sur la longueur des lignes de fichier ni sur d'autres éventuelles incohérences.

3.3 Utilisation du module message :

Le module **message** est fourni dans un fichier archive sur moodle ; il ne doit pas être modifié. Son interface **message.h** détaille l'ensemble des fonctions à appeler il faudra utiliser les messages de cette façon :

```
if(une détection d'erreur est vraie)
{
      cout << message::appel_de_la_fonction(paramètres éventuels);
     std ::exit(EXIT_FAILURE) ; // Rendu1
}
```

Dans le cas de succès de la lecture du fichier, c'est la fonction **success()** qu'il faut appeler.

A partir du rendu2, il ne faudra pas quitter le programme mais renvoyer un booléen d'échec pour les cas d'échec et un booléen de succès quand la lecture de fichier et toutes les validations ont été effectuées avec succès.

Le message affiché par ces fonctions se termine par un passage à la ligne. Il ne faut pas en ajouter un.

3.4 Méthode de travail

Plusieurs approches sont possibles ; utilisez le document *méthodes de développement de projet* comme guide. Au niveau de l'exécution, nous fournissons un nombre limité de fichiers de tests sur lesquels votre programme sera évalué. Le succès de ces tests ne peut pas garantir l'absence de bugs pour d'autres fichiers de tests. Donc, commencez à *organiser votre propre batterie de tests* indépendamment de ce que nous mettons à disposition.

3.4.1 ACTION : test unitaire de tools

Pour effectuer le test unitaire d'un module vous devez écrire un programme de test (*scaffolding*) qui effectue des appels de toutes les fonctions offertes par le module et ce programme compare le résultat renvoyé par chaque appel au résultat attendu (que vous avez calculé indépendamment par un autre moyen).

A partir des indications de ce rendu1 décidez du nom des fonctions et leur prototype et mettez les fonctions exportées dans **tools.h**. Une fois cela fait on peut inclure **tools.h** dans un programme de test pour tester chacune des fonctions de **tools.cc**. C'est votre responsabilité de trouver un nombre suffisant d'exemples pertinents pour s'assurer qu'on n'oublie pas de cas particuliers.

Il est essentiel d'effectuer ce test unitaire sur **tools** avant d'utiliser ses fonctions dans le Modèle.

3.4.2 ACTION : test du Makefile de l'architecture du rendu1

A partir du dessin de l'architecture du rendu1 (page 1, Fig9a de la Donnée) en déduire les dépendances et écrire le fichier Makefile. Testez-le avec des modules contenant le minimum pour être compilable et exécutable, c'est-à-dire les includes et, au plus haut niveau, une fonction main() vide ou simplement avec affichage d'un message.

3.4.3 ACTION : tests du module jeu

Au stade du rendu1, seul le constructeur et la fonction/méthode de lecture de fichier sont absolument nécessaires. Dans ce module l'opération de lecture met en place *l'automate de lecture* (cours Topic3) qui filtre les lignes inutiles du fichier et délègue l'analyse fine de lecture d'une ligne aux autres modules plus spécialisés

(mobile et de chaine qui ont la responsabilité de faire les vérifications nécessaires). L'automate peut être testé avec des *stubs* de ces fonctions/méthodes plus spécialisées de lecture qui renvoient toujours true pour indiquer que le décodage de la ligne de fichier s'est bien passé.

A ce stade l'intégration avec le module projet est immédiate puisqu'il suffit de remplacer le stub de la fonction/méthode de lecture du fichier par un appel de celle mise au point pour le module jeu.

3.4.4 ACTION : tests du module mobile

Les entités de mobile peuvent être représentées par une seule classe à l'aide d'un attribut de type mais cela n'est accepté que pour le rendu1 ; il faut proposer une hiérarchie de classes (2 niveaux suffisent) pour les rendus 2 et 3. Ecrivez du code de scaffolding pour initialiser des instances des différents types d'entités et vérifier la valeur des attributs avec une méthode d'affichage dans le terminal.

Ensuite seulement écrivez la méthode qui décode la ligne du fichier pour chaque type d'entité de mobile. Testez cette méthode en transmettant une ligne avec la syntaxe du fichier et vérifiez avec la méthode d'affichage que le décodage s'est bien passé.

Pour le type faiseur, ce module dispose de toutes les informations pour tester les collisions inter-faiseur ; cette opération est donc une responsabilité de ce module.

Ensuite intégrez de proche en proche avec les niveaux supérieurs.

3.4.5 ACTION : tests du module chaine

Une seule classe suffit pour représenter une chaîne. Ecrivez du code de scaffolding pour initialiser une chaîne et vérifier la valeur des attributs avec une méthode d'affichage dans le terminal.

Une fois les tests internes à une chaîne effectués, intégrez avec le niveau supérieur de jeu qui sera en charge des tests qui utilisent des informations de différents modules. Dans ce but prévoyez une méthode qui fournit un accès en lecture seule à l'ensemble des articulations pour que le module **jeu** puisse les utiliser.

3.4.6 ACTION : tests de niveau supérieur

Le module **jeu** est responsable du test entre la chaine et les faiseurs. Il doit demander l'ensemble des articulations au module chaine et il peut le transmettre en paramètre à une méthode du module faiseur pour qu'il détecte si l'un des points transmis est dans un élément de faiseur.

4. Forme du rendu1

Convention de style : il est demandé de respecter les conventions de programmation du cours.

- On autorise *une seule* fonction/méthode de plus de 40 lignes (max 80 lignes)

Documentation : l'entête de vos fichiers source doit indiquer les noms des membres du groupe, etc.

Rapport (pdf max 0.5 page) décrit l'activité individuelle de chaque membre selon ce message edstem

Rendu : pour chaque rendu **UN SEUL membre d'un groupe** (noté **SCIPER1** ci-dessous) doit téléverser un fichier **zip**[1] sur moodle (pas d'email). Le non-respect de cette consigne sera pénalisé de plusieurs points. Le nom de ce fichier **zip** a la forme :

SCIPER1_ SCIPER2.zip

Compléter le fichier fourni **mysciper.txt** en remplaçant 111111 par le numéro SCIPER de la personne qui télécharge le fichier archive et 222222 par le numéro SCIPER du second membre du groupe.

[1] Nous exigeons le format zip pour le fichier archive

Le fichier archive du rendu1 doit contenir (**aucun répertoire**) :

- Fichier texte édité **mysciper.txt**
- Fichier pdf du rapport d'activité
- Votre fichier **Makefile** produisant un executable **projet**
- Tout le code source (.cc et .h) nécessaire pour produire l'exécutable.

*On doit obtenir l'exécutable **projet** après décompression du fichier **zip** en lançant la commande* **make** *dans un terminal de la VM (sans utiliser VSCode).*

Auto-vérification : Après avoir téléversé le fichier **zip** de votre rendu sur moodle (upload), récupérez-le (download), décompressez-le et assurez-vous que la commande **make** produit bien l'exécutable lorsqu'elle est lancée dans un *terminal de la VM* (sans utiliser VSCode) et que celui-ci fonctionne correctement.

Exécution sur la VM: votre projet sera évalué sur la VM à distance (compilation avec l'option -std=c++17).

Backup : Il y a un backup automatique sur votre compte myNAS.

Debugging : Visual Studio Code offre un outil intéressant pour la recherche de bug (VSCode tuto).

5. Travail en groupe

Gestion du code au sein d'un groupe

- **Solution recommandée, simple mais limitée** : créer un répertoire sur **gdrive.epfl.ch** qui est partagé seulement par les 2 membres du groupe. Avec cette approche, chacun dispose de sa copie personnelle du projet et de celle du répertoire partagé. Cependant il n'y a pas d'éditeur de code en mode partagé.

- **Solution plus ambitieuse mais qui demande un apprentissage non négligeable (niveau avancé)** : créer un compte sur gitlab.epfl.ch . Attention : il FAUT **restreindre** l'accès du code aux seuls 2 membres du groupes sinon il est public par défaut. Ensuite il faut utiliser l'outil **git** avec **gitlab** comme expliqué dans cette video YouTube . Nous ne pouvons malheureusement pas dédier du temps d'assistant pour du support sur cet outil. C'est pourquoi il est optionnel. Une initiation pour débutants est prévue le 6 mars en CM 1 105 par l'association gnugen.