Projet Informatique – Sections Electricité et Microtechnique

Printemps 2023 : *Microrécif* © R. Boulic & collaborators

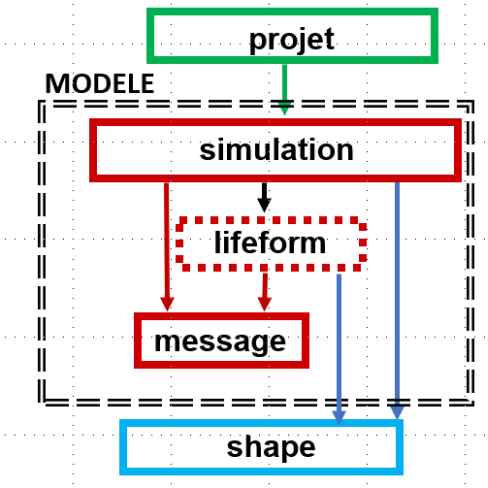
Rendu1 (jeudi 28 mars 23h59)

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 9a).

1. Buts du rendu1 : mise au point de shape et lecture de fichier

Le *premier* objectif est de disposer d'un module **shape** dont chaque fonction est validée par des test 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 **shape** 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 les structures de données pour les segments, carrés et cercles.



Donnée Fig 9a
N'utilise PAS GTKmm

C'est seulement après la validation du module **shape** que vous pourrez aborder le second objectif d'ébauche du **Modèle** que vous voyez dans la boite en pointillés au dessus du module **shape** 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 donnée soit une 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 **simulation** qui agit comme point d'entrée du **Modèle**.

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

- **simulation** : 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 de la simulation, 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 simulation en détectant des erreurs prédéfinies.
  - en vertu du principe d'abstraction, ce module délègue l'exécution des sous-problèmes de ces tâches auprès des modules qui gèrent les entités de **lifeform** (cf Fig9a).

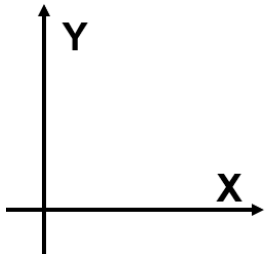
- **lifeform** : pour le rendu final il est demandé de mettre en place *une hiérarchie de classes* pour gérer les différents types d'entités. Cependant, pour le **rendu1**, une approche simplifiée sera acceptée avec un attribut de *type* dans une classe unique **Lifeform**.

- 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 **shape** et comment le tester.

2. But du module shape

Ce module définit le type **S2d** pour modéliser un point ou un vecteur dans le plan (Donnée section 7.3.3). Il s'en sert pour définir les types pour modéliser un segment, un carré ou un cercle dans un espace continu en virgule flottante double précision et selon les convention d'axes visibles à droite.

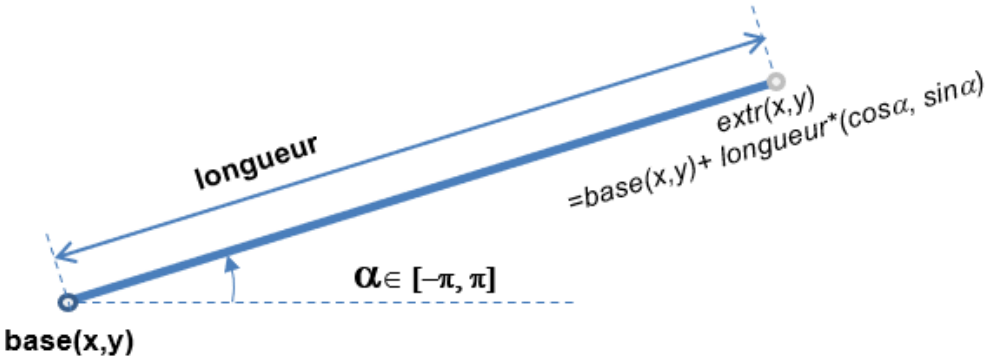


Rendu1 Fig1 : conventions d'axes de coordonnées X et Y

Un **Segment** doit être défini par :

- Un point **S2d** définissant sa base
- Un angle $\in [-\pi, \pi]$ par rapport à l'axe X
- Sa longueur (positive double précision)

Les 3 informations permettent d'en déduire les coordonnées du point **S2d** définissant l'extrémité du segment. Il est autorisé de mémoriser également les coordonnées de ce second point dans la structure de donnée **Segment** (au choix, structure ou classe) à condition de s'assurer qu'elles sont toujours cohérentes avec les 3 autres données (qui sont considérées comme prioritaires dans le choix de la représentation d'un **Segment** pour ce projet). La formule du calcul de l'extrémité est indiquée dans la figure ci-dessous :



Rendu1 Fig2 : les 3 informations fondamentales définissant un Segment : sa base (**S2d**), son angle par rapport à l'axe X et sa longueur positive. Elles permettent d'en déduire facilement les coordonnées de l'extrémité.

Pour le rendu1, ce module doit offrir des fonctions qui renvoient :

- l'écart angulaire dans l'intervalle $[-\pi, \pi]$ entre 2 segments (section 2.1)
- le booléen de superposition de 2 segments ayant un point commun (section 2.1).
- le booléen d'intersection ou de superposition de 2 segments indépendants (section 2.2).

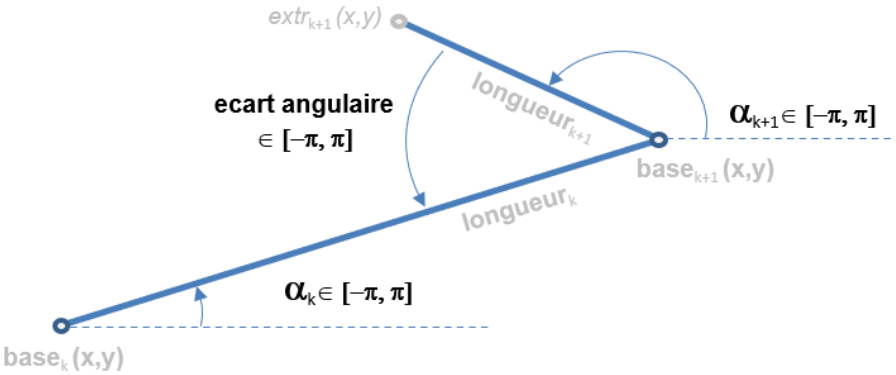
Seule la constante **epsil_zero** est définie par ce module pour gérer les variantes de détection de superposition des entités. Les autres constantes sont définies par le Modèle et ne doivent pas apparaître dans **shape**.

Remarque anticipée sur le rendu2 :

- Les fonctions de dessin de carré, cercle, segment seront aussi à faire dans ce module mais seulement à partir du rendu2 car le rendu1 ne doit pas avoir de dépendances vis-à-vis des bibliothèques de dessin.
- La détermination du point le plus proche (angulairement) d'un segment aura besoin du calcul de l'écart angulaire dans l'intervalle $[-\pi, \pi]$ entre un segment et un vecteur construit à partir du premier point du segment et d'un point supplémentaire.

2.1 Ecart angulaire et détection de la superposition de 2 segments ayant un point commun

L'écart angulaire est définit par la Figure 3 ; il peut être calculé à partir des valeurs α des 2 segments et doit être normalisé dans l'intervalle $[-\pi, \pi]$.



Rendu1 Figure 3 : l'écart angulaire est nul quand le segment k+1 est totalement replié sur le segment k ; son signe change quand le segment k+1 passe de l'autre coté du segment k (en configuration repliée cela se passe autour de zéro ; en configuration alignée le signe change aussi mais avec des valeurs autour de +/-π.)

Ce calcul nous permet de détecter la superposition de 2 segments ayant un point commun:

- Lecture de fichier : la valeur est *nulle* lorsque le segment k+1 est *totalement replié* sur le segment k.
- Simulation : Lorsqu'on est proche du cas replié, cette valeur d'écart angulaire est faible et on peut aussi détecter que le segment k+1 est passé de l'autre coté du segment k si cette valeur change de signe.

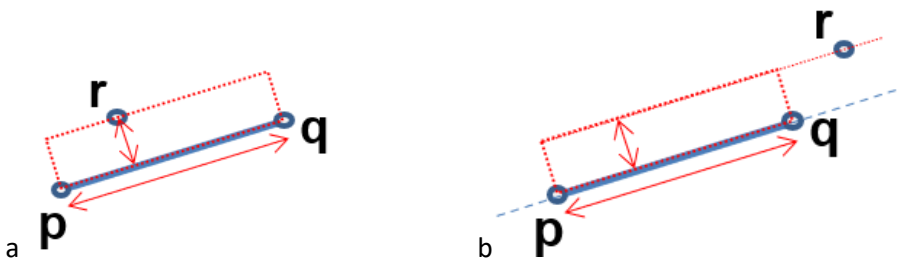
2.2 Détection de l'intersection /superposition de 2 segments indépendants

Nous nous sommes basés sur la méthode de détection d'intersection décrite ici car elle est générale contrairement à l'alternative naïve qui utilise des équations de droite (car cette alternative ne peut pas traiter les segments verticaux de pente infinie).

Malgré son élégance, l'approche en référence est insuffisante car nous travaillons avec des coordonnées en virgule flottante (double), ce qui veut dire qu'il faut introduire **epsil_zero** dans la détection des cas d'alignement.

2.2.1 Modification de la détection de l'alignement de 3 points p, q et r avec la fonction orientation()

Adapter le prototype de cette fonction à notre type **S2d** et utiliser un type **double** pour **val**. Un peu de géométrie nous apprend que **val** calcule la *surface rectangulaire (signée) donnée par le produit de la longueur du segment **pq** par la distance séparant le point **r** de la droite passant par le segment **pq*** (Fig 4a).



Rendu1 Figure 4 : variable **val** dans la fonction **orientation** calcule la surface (signée) du rectangle rouge ; on en déduit la distance de **r** à la droite passant par les points **p** et **q** (a) ; il faut compléter ce test car **r** se projette peut être en dehors du segment **pq** (b).

Il suffit de diviser le résultat obtenu pour **val** par la norme du vecteur **pq** pour obtenir la distance (signée) de **r** à la droite passant par **pq**. Cette fonction renvoie l'entier **0** si la valeur absolue de cette distance est inférieure à **epsil_zero**. Sinon pas d'autre changement, elle renvoie la même expression.

Il faut noter que **r** peut se projeter en dehors du segment **pq** c'est pourquoi il faut d'autres tests pour compléter celui-ci (Fig 4b). C'est le but de la fonction onSegment discutée maintenant.

2.2.2 Modification de la fonction onSegment de détection d'appartenance d'un point q à un segment pr

L'ordre des paramètres n'est pas modifié pour pouvoir ré-utiliser cette fonction telle quelle dans la fonction doIntersection() qui donne le résultat final sur la présence d'une intersection ou d'un alignement. On utilise notre type **S2d** pour ses paramètres.

La formule proposée n'est pas satisfaisante pour traiter la tolérance **epsil_zero** pour garantir que le point **q** se projette bien à l'intérieur du segment **pr**. A la place on exploite le produit scalaire **s = pr.pq** . La valeur de la projection **x** de **pq** sur **pr** est alors donnée en divisant **s** par la norme de **pr**. La fonction renvoie true si :

$$-\text{epsil\_zero} \leq x \leq (\text{norme de } \mathbf{pr}) + \text{epsil\_zero}$$

Sinon elle renvoie false.

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 **simulation.**

Deux stratégies sont autorisées concernant *l'instance* de la classe **Simulation** 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 **simulation** (car elle est unique) et une fonction **lecture** du module simulation 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 2.2 et 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)**.

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 / simulation)

Le choix d'effectuer la sauvegarde de la simulation dans un fichier formaté introduit dans 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 de la simulation*.

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 **shape** 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 au récipient de la simulation

On ignore le rayon des algues et des scavengers pour ces tests d'appartenance au récipient de la simulation car ces paramètres sont simplement destinés à la visualisation de la simulation. Cette première famille de test

n'a besoin d'être faite que lors de la lecture de fichier car la mise à jour de la simulation va garantir de ne pas ré-introduire cette classe de problèmes :

- inclusion des centres des algues et scavengers, et l'origine des coraux dans le domaine **[1, max-1]**

La famille de tests qui suit doit aussi être fait pendant la simulation mais au moins ils garantissent qu'on n'a pas besoin de tester l'intersection des segments de coraux avec les segments des bords du récipient :

- inclusion des autres centre de rotation et de l'effecteur des coraux dans le domaine **]epsil_zero, max – epsil_zero[**

3.2.3 Test de collision/superposition de segments de coraux

Les segments des coraux ne doivent pas se superposer ou s'intersecter ni être en collision.

- Par construction, deux segments consécutifs d'un corail possèdent un point commun. Pour ces cas, le test doit porter sur l'éventuelle superposition de tels segments consécutifs (section 2.1 de ce rendu1). On considère qu'il y a superposition de 2 segments consécutifs si *l'écart angulaire* entre les 2 segments (normalisé dans **[-π, π])** est :
    - nul **en lecture de fichier,**
    - s'il appartient à l'intervalle **[-delta_rot, delta_rot]** et change de signe lors de la mise à jour en simulation.

- Pour tous les autres cas intra ou inter corail, le projet doit détecter les intersection/superposition par une méthode reposant sur une détection d'intersection avec un calcul de distance pour traiter correctement les cas de superposition/collision (section 2.2 de ce rendu1).

3.2.4 Autres tests à effectuer

La lecture doit aussi vérifier (section 4.2 de la donnée générale) :
- que l'age est strictement positif
- que la longueur des segments est comprise dans **[l_repro-l_seg_interne, l_repro[**
- que l'angle des segments est compris dans **[-π, π]**
- que le rayon des scavengers est compris dans **[r_sca, r_sca_repro[**
- L'unicité des identificateurs de coraux
- L'existence d'un corail identifié avec l'identificateur mémorisé par un scavenger dans l'état MANGE

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. Comme pour la détection d'erreur au niveau de **shape**, 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
}
```

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. Dans ce cas, c'est la fonction **success()** qu'il faut appeler.

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 shape

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 le nom des fonctions et leur prototype et mettez les fonctions exportées dans **shape.h**. Une fois cela fait on peut inclure **shape.h** dans un programme de test pour tester chacune des fonctions de **shape.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 **shape** 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 simulation

Au stade du rendu1, seul le constructeur et la fonction/méthode de lecture de fichier sont 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 (lifeform 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 simulation.

3.4.4 ACTION : tests du module lifeform

Les entités de lifeform 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 différentes 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 lifeform. 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. Ensuite intégrez de proche en proche avec les niveaux supérieurs.

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.

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.

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

- Fichier texte édité **mysciper.txt**
- 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** en lançant la commande* **make** *après décompression du fichier **zip**.*

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 et que celui-ci fonctionne correctement.

Exécution sur la VM: votre projet sera évalué sur la VM à distance.

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

Responsabilité de contribution et de compréhension de l'ensemble du code

Nous exigeons que les deux membres du groupe contribuent à la mise au point du projet.
Nous recommandons de combiner :

- des réunions périodiques pour de la *planification* des tâches pour documenter ***qui fait quoi pour quand***
- du *travail indépendant* sur des tâches de codage et de test bien définies par votre planification
- des périodes de *codage/test simultané* devant un ordinateur (surtout pour debugging)
  - deux rôles : une personne tape au clavier ; l'autre l'assiste. Les deux verbalisent.
  - deux règles : personne ne reste passif ; permuter les rôles régulièrement.

L'oral final individuel portera sur la compréhension de ***l'ensemble du code*** du projet

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 . A partir de ce compte vous créez un projet. 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.

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