



# Projet Programmation

Responsable du cours : Dawood AL CHANTI  
[dawood.al-chanti@grenoble-inp.fr](mailto:dawood.al-chanti@grenoble-inp.fr)

## Table des matières

<b>1</b>	<b>Présentation générale du projet</b>	<b>3</b>
1.1	Philosophie du projet . . . . .	3
1.2	Organisation . . . . .	3
1.3	Environnement de travail . . . . .	4
<b>2</b>	<b>Sujet n° 1 : Comptage de cellules</b>	<b>8</b>
2.1	Introduction . . . . .	8
2.2	Étapes préparatoires . . . . .	8
2.3	Version minimale . . . . .	12
2.4	Pour aller plus loin . . . . .	17
<b>3</b>	<b>Sujet n° 2 : Simulation du problème à N corps</b>	<b>23</b>
3.1	Introduction . . . . .	23
3.2	Étapes préparatoires . . . . .	24
3.3	Version minimale . . . . .	26
3.4	Pour aller plus loin . . . . .	27
<b>4</b>	<b>Évaluation</b>	<b>35</b>
4.1	Compétences . . . . .	35
4.2	Notation et Barème Indicatif par rapport à la Compétence . . . . .	35
4.3	Évaluation formative . . . . .	37
4.4	Évaluation sommative . . . . .	37
<b>A</b>	<b>Rappels de C</b>	<b>39</b>
<b>B</b>	<b>Format PGM</b>	<b>42</b>

# 1 Présentation générale du projet

## 1.1 Philosophie du projet

Ce projet a pour but de vous mettre dans la situation suivante : vous devez développer un programme informatique **complet** réalisant une tâche donnée, plus complexe que les exercices isolés faits au premier semestre.

L'objectif **principal** est d'aboutir à un projet **finalisé**. Vous devrez donc **livrer** un projet bien organisé, fonctionnel, et facilement utilisable. Nous privilégierons les projets simples, propres, et qui fonctionnent correctement, et ce même s'ils sont moins avancés d'un point de vue algorithmique. Prenez ça en considération lorsque vous développez ! L'intitulé de ce cours est « Projet programmation », pas « Projet informatique », et c'est volontaire.

Notre intention est que vous disposiez à l'issue de ce cours d'un projet que vous maîtrisez, dont vous êtes **fier**, et que vous pourriez montrer lors d'un entretien pour justifier d'une première expérience de développement informatique en C.

## 1.2 Organisation

**Organisation du répertoire du projet** Nous vous conseillons d'organiser le répertoire de votre projet de la manière présentée en Figure 1. C'est cette organisation qui est choisie dans le projet de départ fourni.

- ☐ `bin` \_\_\_\_\_ Contient le(s) exécutable(s), vide au moment du livrable final
- ☐ `include` \_\_\_\_\_ contient tous les fichiers header `*.h`
- ☐ `obj` \_\_\_\_\_ Contient les fichiers objets `.o`, vide au moment du livrable final
- ☐ `src` \_\_\_\_\_ Contient tous les fichiers sources C `*.c`, sauf les tests avec un `main`
- ☐ `test` \_\_\_\_\_ Contient tous les fichiers sources C contenant les programmes et les tests avec un `main`
- ☐ `Makefile` \_\_\_\_\_ Pour la compilation avec `make`
- ☐ `README.md` \_\_\_\_\_ Contient des instructions sur la façon d'exécuter le programme et ce à quoi s'attendre.
- ☐ `rapport` \_\_\_\_\_ Contiennent le rapport de la 1ère séance et le rapport final.

FIGURE 1 – Organisation conseillée pour le répertoire du projet

**Organisation des fichiers sources** Vous pouvez vous fixer les règles suivantes lors de l'écriture de votre code :

- Chaque fonction ne dépassera pas 30 lignes,
- Chaque fichier source contiendra au maximum 10 fonctions.

Pensez également à **indenter** correctement votre code. Même si vous compilez régulièrement, et on vous le conseille vivement, le compilateur passera toujours beaucoup moins de

temps que vous à lire votre code. Écrivez donc votre code pour qu'il soit facile à lire **pour vous**.

**Travailler en mode projet** Le projet programmation nécessite de travailler en mode projet, c'est-à-dire de réaliser quelques travaux en équipe et de gérer les relations individuelles dans ce contexte : définition d'une organisation d'équipe (qui fait quoi ?), d'une organisation dans le temps (planification des tâches et suivi de leur avancement), utilisation d'outils de gestion de projet (Gitlab). **Chaque membre doit réaliser certaines tâches pour valider le projet. Nous demanderons à chaque équipe de proposer un score** pour chaque membre du projet. La somme des notes de l'équipe doit être égale à 20.

**Réusinage de code** Le réusinage de code (*refactoring* en anglais) consiste à réécrire du code afin de le rendre plus lisible. N'hésitez pas à en faire un usage immodéré, que ce soit sur les parties du code que vous avez rédigées ou celles écrites par l'autre membre du binôme. Il est **normal** d'écrire du code peu lisible lors des premières tentatives, **mais** il n'est pas acceptable de le garder dans cet état.

### 1.3 Environnement de travail

**Phelma Machine** Vous travaillerez sous Linux, dans l'environnement de développement de l'école que vous avez utilisé au premier semestre.

Si vous souhaitez retrouver cet environnement sur votre ordinateur personnel, l'installer par vous-même.

**Éditeur de texte** Vous travaillerez avec un éditeur de texte disposant au minimum de la coloration syntaxique. Atom est un bon choix.

**Collaboration avec Git** Pour collaborer efficacement sur une même base de code, vous utiliserez Git. Vous trouverez sur le site [www\\_info](http://www.info.univ-grenoble.fr/~phelma/ressources/s5/) [un guide de démarrage pour l'utilisation de Git](#) ou vous pouvez le trouver sur chamilo également dans la section "C, gdb et Valgrind Ressources de S5". Une [liste des questions fréquentes à propos de Git](#) est également disponible sur ce site.

Quelques exemples de dépôts Git bien tenus :

- <https://github.com/squidfunk/mkdocs-material>
- <https://github.com/Ledger-Donjon/rainbow>
- <https://github.com/etalab/DVF-app>

Votre utilisation de Git se limitera la plupart du temps à cette séquence de commandes :

- `git add fichier` : suivre un fichier ou prendre en compte les modifications apportées sur un fichier,
- `git commit "message"` : regrouper les modifications précédemment validées dans un *commit*,

- `git push` : pousser vos contributions locales sur le dépôt distant,

La commande `git push` échouera si votre collègue a préalablement poussé des modifications que vous n'avez pas encore récupérées. Pour cela, lancez la commande `git pull` pour les récupérer, puis lancez à nouveau la commande `git push`, qui cette fois fonctionnera.

### ⚠ Conflits

Il faut éviter les **conflits** lors de l'écriture du code, c'est à dire travailler simultanément sur une même section de code. En effet, dans ce cas, Git n'a aucun moyen de savoir quelle version conserver et vous demandera de réaliser un *merge* manuel. Travaillez donc sur des sections de code distinctes, en vous répartissant les tâches au préalable.

### ✓ Bonnes pratiques lors de l'utilisation de Git

Nous vous conseillons les bonnes pratiques suivantes :

- faites des *commits* concis, c'est à dire en ne travaillant que sur une seule chose à la fois, et choisissez un message de *commit* explicite décrivant ce que vous venez de coder,
- travaillez sur des sections distinctes du code, pour éviter les conflits,
- utilisez la commande `git status` pour connaître l'état courant de votre projet Git,
- maintenez un fichier `.gitignore` pour éviter de suivre les fichiers inutiles : ne suivez que les fichiers que vous avez écrit vous-même.

**Compilation avec make** Votre projet, puisqu'il est d'une certaine envergure, sera découpé en plusieurs fichiers `.c` et `.h`. Pour simplifier le processus de compilation, vous utiliserez `make`. Ce dernier trouvera les règles de compilation dans le fichier `Makefile` du projet. Vous trouverez sur le site [www\\_info](http://www.info.univ-grenoble.fr/~phelma) [un document expliquant comment rédiger un fichier Makefile](#).

### ✓ Compilation

Pensez à **compiler le plus souvent possible** ! Le compilateur est très utile pour détecter au plus tôt les erreurs lors de l'écriture du code. Exploitez-le !

**Débogage avec gdb** Dans le cas, probable, où le code que vous aurez écrit ne fonctionne pas du premier coup, vous devrez le déboguer. La première solution, simple mais limitée, consiste à faire afficher la valeur des variables avec des `printf`. C'est très souvent suffisant.

La deuxième solution, plus puissante, consiste à utiliser `gdb` pour placer des points d'arrêt dans votre code (breakpoint), faire afficher la valeur des variables (`disp`) et exécuter en mode pas à pas (`n` ou `s`). L'utilisation de `gdb` est **indispensable** en cas d'erreur de segmentation, pour identifier la ligne responsable.

gdb s'utilise de la manière suivante :

```
$ gdb ./bin/executable
```

### ✓ Exécutable avec arguments

Si votre exécutable prend des arguments, il faut passer l'option - -args à gcc :

```
$ gdb --args ./bin/executable arg1 arg2
```

Dans le cas contraire, gdb croira que les arguments arg1 et arg2 sont pour lui.

Vous pouvez **trouver plus d'informations dans le tutoriel** suivant sur le site inria [un tutoriel sur le démarrage de gdb](#).

**Vérification des fuites mémoire avec Valgrind** Lorsqu'on doit allouer la mémoire dynamiquement en faisant des allocation dynamiques (avec malloc), il faut la libérer correctement ensuite (avec free). Valgrind vérifie que toute la mémoire allouée a bien été libérée. Il vérifie également que les accès en lecture et écriture ne sont réalisés que dans la mémoire allouée.

**Exemple** Ce programme alloue un tableau de dix entiers. On tente ensuite d'afficher la case d'indice 20, puis d'écrire à la case d'indice 1000. Enfin, on ne libère pas le tableau.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     int* tableau = malloc(10 * sizeof(int));
6     printf("%d\n", tableau[20]);
7     tableau[1000] = 5;
8 }
```

Voici une partie de la sortie de Valgrind, indiquant qu'à la fin de l'exécution, 40 octets étaient encore utilisés, soit 10 entiers de 4 octets. Nous avons fait une allocation sans libération. Valgrind détecte aussi la lecture et l'écriture illégales et affiche le numéro des lignes fautives.

```
==35802== Invalid read of size 4
==35802==    at 0x10918B: main (test.c:6)
...
==35802== Invalid write of size 4
==35802==    at 0x1091AA: main (test.c:7)
...
==35802== HEAP SUMMARY:
```

```
==35802==      in use at exit: 40 bytes in 1 blocks
==35802==    total heap usage: 2 allocs, 1 frees, 1,064 bytes allocated
==36326==
==36326== LEAK SUMMARY:
==36326==    definitely lost: 40 bytes in 1 blocks
...
==36326== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

## 2 Sujet n° 1 : Comptage de cellules

*D'après des discussions avec M. Michel DESVIGNES.*

### Objectif

L'objectif de ce projet est de **compter** les cellules présentes dans une image en niveaux de gris en les **isolant** par des méthodes de morphologie mathématique.

### 2.1 Introduction

Lorsqu'on réalise une **culture cellulaire**, il est essentiel de pouvoir **compter** les cellules présentes dans la culture à un instant donné. Cela permet de savoir si la culture fonctionne correctement et si de nouvelles cellules s'y développent.

Pour réaliser cette tâche, on peut compter les cellules manuellement. Pour simplifier le processus, on place généralement une grille au dessus de la culture, puis, à l'aide d'un microscope, on compte les cellules dans chaque case de la grille. En sommant finalement tous les comptes intermédiaires, on obtient le nombre total de cellules. Ce processus est long et fastidieux.

Des méthodes de traitement d'images permettent heureusement de réaliser cette tâche **automatiquement**, à partir d'une image de la culture cellulaire comme celle de la Figure 2. C'est l'objectif de ce projet.

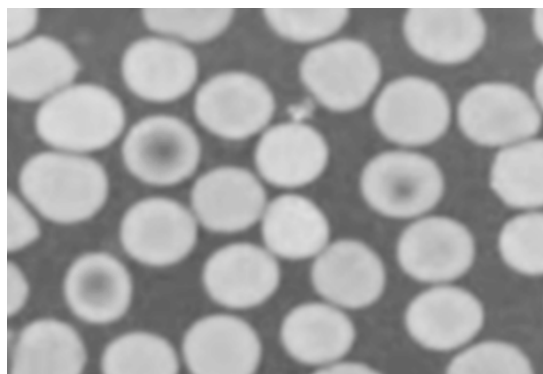


FIGURE 2 – Image d'une culture cellulaire.

### 2.2 Étapes préparatoires

#### 2.2.1 Lecture d'une image au format PGM

Il vous faudra donc dans un premier temps écrire une fonction qui lit une image au format PGM et la stocke en mémoire dans un tableau à deux dimensions. Le format PGM binaire est décrit dans l'annexe B.



Lors de l'allocation de la matrice des pixels, vous prendrez garde à réaliser une allocation mémoire **contiguë**. Ceci est illustré dans la Figure 3. On souhaite stocker une matrice de trois lignes et quatre colonnes (voir Figure 3a). On souhaite également pouvoir accéder à un élément de la matrice avec la syntaxe suivante : `matrice[ligne] [colonne]`.

La première solution, présentée en Figure 3b, consiste à d'abord allouer un tableau de pointeurs pour les lignes, puis à allouer chaque ligne individuellement. Dans ce cas, lors des allocations successives des lignes, rien ne garantit que l'allocateur mémoire les placera côte à côte en mémoire. Cela peut avoir un coût en performances important, à cause des **défauts de cache**.

Une autre solution, présentée en Figure 3c, consiste à d'abord allouer un tableau de pointeurs pour les lignes, comme précédemment, puis à faire **une seule allocation** pour l'intégralité de la matrice, sur le pointeur d'indice 0. Il faut ensuite faire pointer les pointeurs d'indice 1 à `nb_lignes - 1` vers la bonne partie de la zone mémoire allouée.

En plus des raisons de performances évoquées au dessus, le fait de réaliser une allocation contiguë permet, lors de la lecture ou l'écriture d'un fichier, de travailler avec l'intégralité des données à la fois, en les considérant comme un seul bloc binaire.

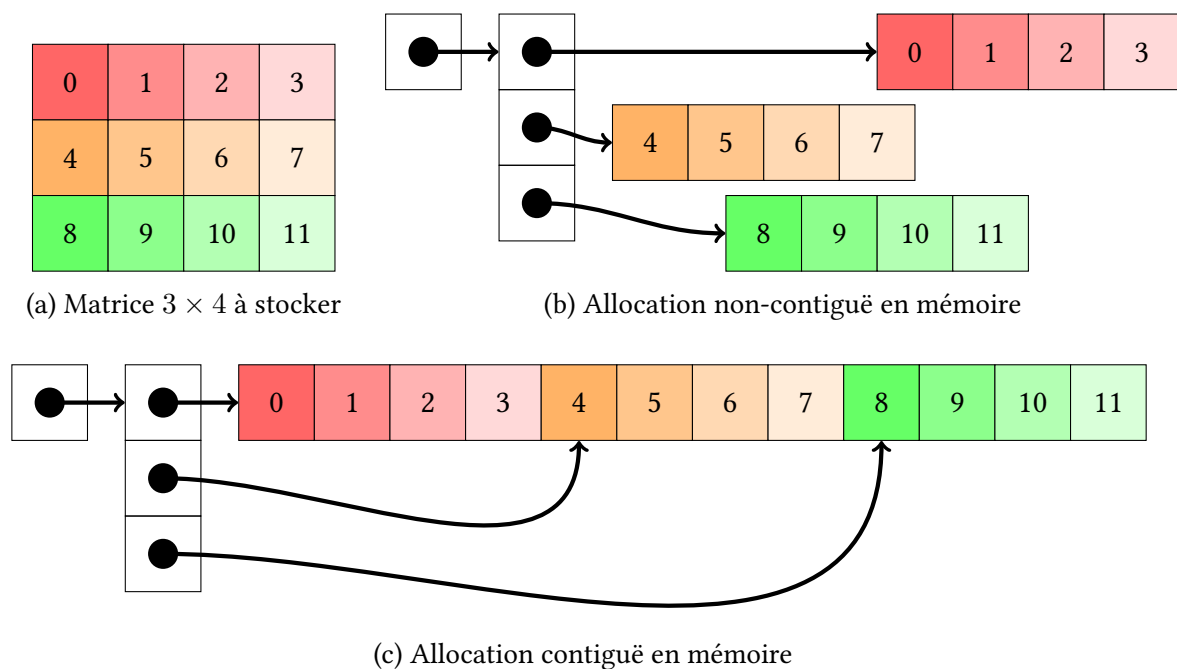


FIGURE 3 – Allocations mémoire (adapté de <http://c-faq.com/aryptr/dynmultidimary.html>).

Lors de la lecture de l'image dans un fichier PGM, il peut être utile de conserver d'autres informations utiles, comme le nombre de lignes et de colonnes. Vous définirez une structure, voire un nouveau type, adaptés pour représenter les images.

Vous écrirez ensuite une fonction dont le rôle sera d'écrire une image dans un fichier. Vous commencerez par écrire directement l'image lue. Cela vous permettra de vérifier que la lecture et l'écriture fonctionnent correctement.

Vous pourrez vérifier que les deux fichiers sont identiques avec la commande suivante :

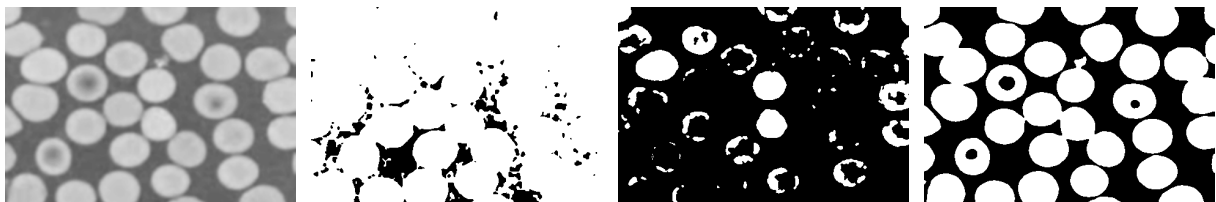
```
$ diff image.pgm copie.pgm
```

Vous n’oubliez pas d’écrire des fonctions pour libérer la mémoire à la fin du programme !

### 2.2.2 Seuillage manuel

Le premier traitement à appliquer à l’image consiste à la seuiller, c’est-à-dire à passer d’une image en niveaux de gris, dont la valeur des pixels va de 0 à 255, à une image binaire (ou “noir et blanc”), dont la valeur des pixels est de 0 ou 255.

Le seuil définit, pour chaque pixel de l’image, s’il sera noir ou blanc après seuillage. Fixer correctement la valeur de ce seuil est important, comme illustré en Figure 4, où l’image de la Figure 4a est seuillée avec différentes valeurs de seuil. Si le seuil est trop bas (Figure 4b) ou trop haut (Figure 4c), on perd des détails de l’image. Un seuil bien choisi (Figure 4d) permet de bien séparer les objets, qui deviennent blancs, de l’arrière-plan, qui devient noir.



(a) Image de départ en niveaux de gris (b) Image seuillée avec un seuil trop bas (c) Image seuillée avec un seuil trop haut (d) Image seuillée avec un seuil bien choisi

FIGURE 4 – Image seuillée avec différentes valeurs de seuil.

Écrivez une fonction qui réalise le seuillage d’une image et le programme de test associé. Vous écrirez le programme de test de façon à ce qu’il puisse prendre en paramètre le nom de l’image à traiter. Vous pourrez faire en sorte que le seuil puisse être passé en paramètre lui aussi. Par exemple, pour un seuil fixé à 150, voici la commande que vous lancerez :

```
$ ./bin/test_seuillage image.pgm 150
```

### 2.2.3 Seuillage automatique : méthode d’Otsu

La **méthode d’Otsu** permet de déterminer automatiquement la valeur du seuil à partir de l’histogramme de l’image.

**Histogramme** L’**histogramme**  $h$  d’une image représente la distribution des valeurs des pixels de l’image. La Figure 5 montre une image et son histogramme. Sur cet histogramme, on observe deux modes, l’un autour de la valeur 100 et l’autre autour de la valeur 200. Le premier correspond à l’arrière-plan sombre, le second aux cellules plus claires. La méthode d’Otsu permet de déterminer la valeur du seuil qui sépare au mieux ces deux modes.

Écrivez une fonction qui calcule l’histogramme de l’image et le stocke dans un tableau.

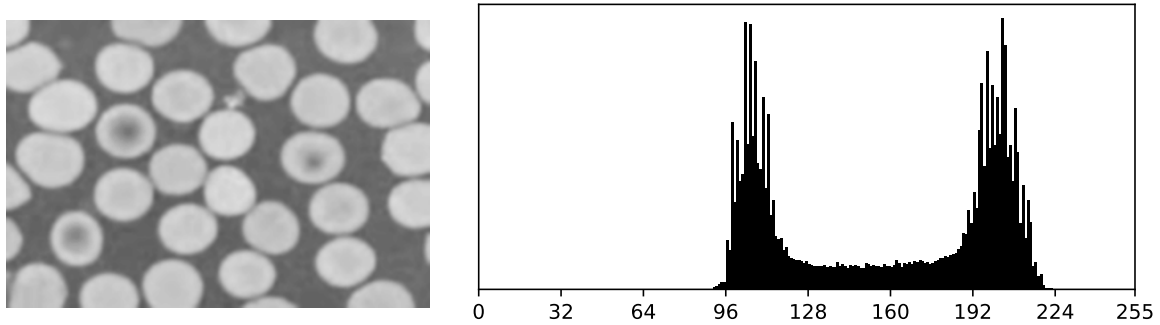


FIGURE 5 – L'image et son histogramme  $h$ .

**Méthode d'Otsu** La méthode d'Otsu énumère toutes les valeurs possibles du seuil, de 0 à 255, et trouve celle qui maximise la variance inter-classe, c'est-à-dire celle qui sépare le mieux les deux modes. La variance inter-classe  $\sigma_b^2$  est définie dans l'équation (1).

$$\sigma_b^2(s) = \omega_0(s)\omega_1(s)[\mu_0(s) - \mu_1(s)]^2 \quad (1)$$

où  $\omega_0(s)$  et  $\omega_1(s)$  sont les probabilités de classe, définies dans l'équation (2)

$$\omega_0(s) = \sum_{i=0}^{s-1} h(i) \quad \text{et} \quad \omega_1(s) = \sum_{i=s}^{255} h(i) \quad (2)$$

et  $\mu_0(s)$  et  $\mu_1(s)$  sont les moyennes empiriques des classes, définies dans l'équation (3).

$$\mu_0(s) = \frac{\sum_{i=0}^{s-1} ih(i)}{\omega_0(s)} \quad \text{et} \quad \mu_1(s) = \frac{\sum_{i=s}^{255} ih(i)}{\omega_1(s)} \quad (3)$$

Le seuil calculé est la valeur de  $s$  pour laquelle la variance inter-classe  $\sigma_b^2(s)$  est maximale.

Vous écrirez une fonction qui réalise le seuillage automatique, puis vous écrirez un programme de test qui permet de seuiller automatiquement une image dont le nom est passé en paramètre.

## 2.2.4 Utilitaire de génération d'images de test

Pour tester le fonctionnement de vos programmes, nous vous fournissons un utilitaire écrit en Python qui permet de générer un image PGM.

Par exemple, pour générer une image de  $3 \times 3$  pixels noire avec un pixel blanc au centre, vous lancerez la commande suivante :

```
$ python2.7 generate_test_image.py NNN NBN NNN
```

## 2.3 Version minimale

### 2.3.1 Opérations booléennes

Nous aurons besoin pour la suite de réaliser des opérations booléennes, ou logiques, entre deux images. Nous définissons donc les trois opérations suivantes : intersection (ET logique), union (OU logique) et OU exclusif.

Les tables de vérité de ces opérations sont rappelées dans la Table 1.

$im_A[i][j]$	$im_B[i][j]$	$im_{INTERSECTION}[i][j]$	$im_{UNION}[i][j]$	$im_{XOR}[i][j]$
N	N	N	N	N
N	B	N	B	B
B	N	N	B	B
B	B	B	B	N

TABLE 1 – Tables de vérité des opérations intersection, union et OU exclusif.

Toutes ces opérations pré-supposent que les images d'entrée sont de même dimension. Il serait intéressant de le vérifier grâce à `assert`.

#### Mot-clé du langage C

`union` est un mot-clé du langage C, vous ne pouvez donc pas l'utiliser comme nom de fonction.

### 2.3.2 Opérations morphologiques basiques

La **morphologie mathématique** est une branche des mathématiques très utile pour le traitement des images. Elle définit des opérations applicables à des images, en particulier celles en noir et blanc, ce qui nous intéressera dans ce projet. Nous nous limiterons aux opérations simples dans le cadre de ce projet.

**Érosion** L'opération d'érosion consiste à calculer le **minimum local** de chaque groupe de  $3 \times 3$  pixels dans l'image de départ. Si dans un groupe de  $3 \times 3$  pixels **tous** les pixels sont blancs, alors le pixel en sortie de l'opération est blanc. Si dans un groupe de  $3 \times 3$  pixels il y a **au moins un** pixel noir, alors le pixel en sortie de l'opération est noir.

L'opération d'érosion est illustrée en Figure 6 où deux exemples de son application sont donnés. L'érosion réduit la surface des zones blanches dans l'image.

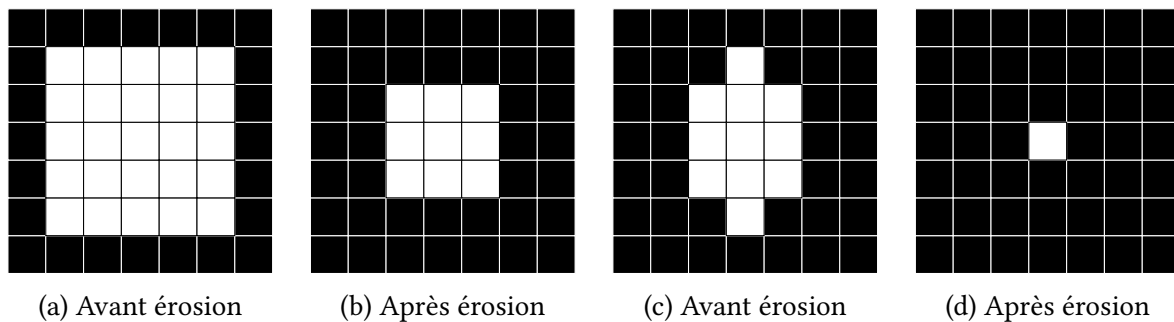


FIGURE 6 – Deux exemples d'application de l'opération d'érosion.

**Dilatation** L'opération de dilatation consiste à calculer le **maximum local** de chaque groupe de  $3 \times 3$  pixels dans l'image de départ. Si dans un groupe de  $3 \times 3$  pixels **tous** les pixels sont noirs, alors le pixel en sortie de l'opération est noir. Si dans un groupe de  $3 \times 3$  pixels il y a **au moins un** pixel blanc, alors le pixel en sortie de l'opération est blanc.

L'opération de dilatation est illustrée en Figure 7 où deux exemples de son application sont donnés. La dilatation augmente la surface des zones blanches dans l'image.

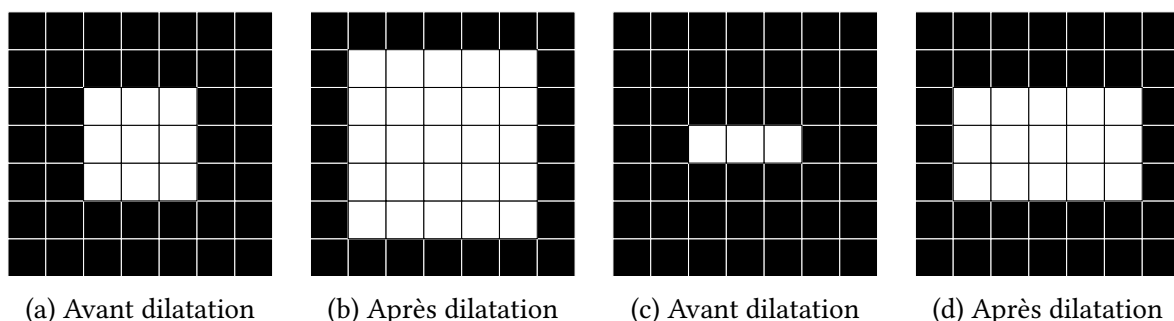


FIGURE 7 – Deux exemples d'application de l'opération de dilatation.

### 2.3.3 Reconstruction

Une autre opération importante de morphologie est la reconstruction. Cette dernière consiste à faire croître une image **graine** dans l'image. On dilate la graine dans les zones blanches de l'image, jusqu'à ce que ces dernières soient complètement remplies. La reconstruction est décrite dans l'Algorithme 1.

---

#### Algorithme 1 Reconstruction morphologique

---

```

1: graine_dilatee = dilatation(graine)
2: image_reconstruite = intersection(image, graine_dilatee)
3: faire
4:   graine_dilatee = dilatation(image_reconstruite)
5:   image_reconstruite = intersection(image, graine_dilatee)
6: tant que image_reconstruite change

```

---

Pour écrire cette fonction, vous aurez besoin de comparer deux images, afin de savoir si `image_reconstruite` a changé. Vous utiliserez `memcmp` pour ça.

La Figure 8 illustre l'opération de reconstruction sur un exemple. À chaque itération de la boucle `while`, la graine croît dans les zones blanches de l'image, jusqu'à remplir complètement la zone du bas. La boucle s'arrête ensuite car l'image reconstruite ne change plus. La reconstruction est terminée : on a bien "reconstruit" la zone blanche du bas de l'image.

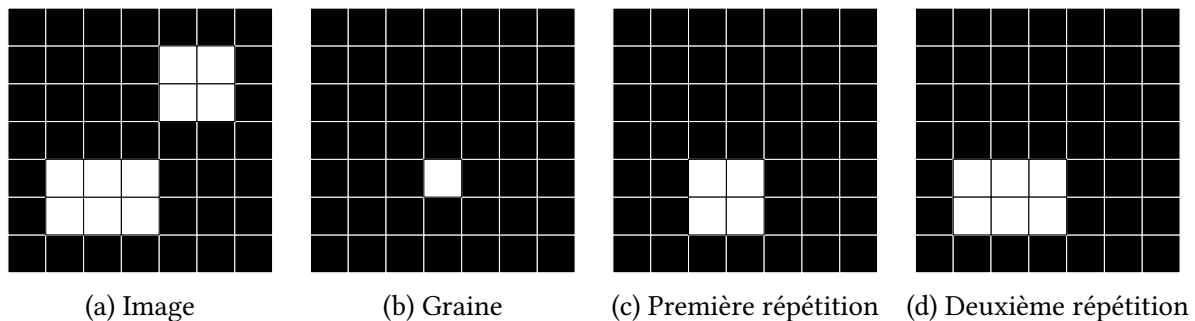


FIGURE 8 – Illustration des étapes de l'opération de reconstruction morphologique.

L'opération de reconstruction définie ci-dessus est très utile pour "sélectionner" certains éléments de l'image. L'effet que la reconstruction aura sur l'image dépend donc de la graine.

### 2.3.4 Suppression des cellules au bord

Pour supprimer les cellules au bord de l'image, on applique l'opération de reconstruction en utilisant comme graine un cadre blanc d'un pixel au bord de l'image, sur fond noir. La Figure 9 montre quelques étapes de cette reconstruction. Les cellules au bord sont "sélectionnées" par la reconstruction. On termine en faisant le OU-exclusif entre l'image reconstruite et l'image initiale (Figure 10).



FIGURE 9 – Étapes de la reconstruction d'un cadre blanc dans l'image seuillée.

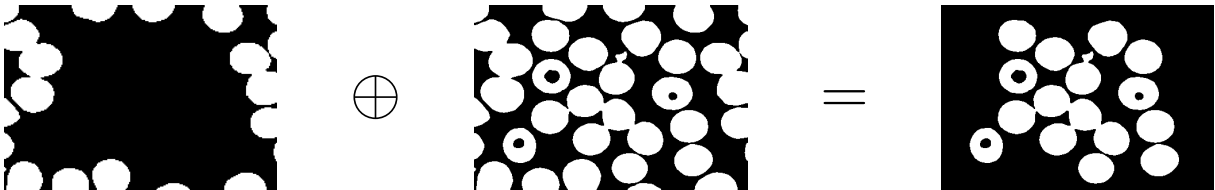
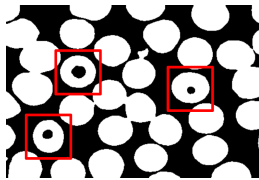


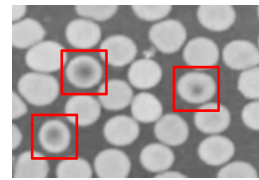
FIGURE 10 – Suppression des cellules au bord par OU-exclusif avec un cadre reconstruit.

### 2.3.5 Bouchage des trous

Certaines cellules de l'image seuillée présentent un **trou**. Ce trou correspond à une zone sombre au milieu de ces cellules dans l'image initiale en niveaux de gris. La Figure 11 montre trois cellules pour lesquelles ce phénomène se produit.



(a) Image seuillée



(b) Image en niveaux de gris

FIGURE 11 – Mise en évidence dans l'image seuillée et l'image en niveaux de gris des cellules présentant un trou dans l'image seuillée.

Pour remplir des cellules avec un trou, on applique l'opération de reconstruction sur l'image **inversée** avec la même graine que précédemment : un cadre blanc d'un pixel sur fond noir, de la taille de l'image. Cela remplit progressivement l'espace entre les cellules, ne laissant finalement que les cellules elles-mêmes.

La Figure 12 montre quelques étapes de la reconstruction du cadre dans l'image inversée. Le fond est progressivement rempli, ne laissant finalement que des cellules sans trou. Il faut finalement inverser à nouveau l'image pour obtenir l'image seuillée initiale avec les trous bouchés.

### 2.3.6 Érosion manuelle

Dans l'image finalement obtenue, certaines cellules se touchent. C'est un problème pour les compter. Pour séparer ces cellules, on érode l'image un certain nombre de fois, fixé ma-



FIGURE 12 – Étapes de la reconstruction d'un cadre blanc dans l'image seuillée.

nuellement. La Figure 13 montre une image avec les cellules bien séparées.

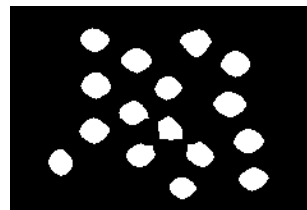


FIGURE 13 – Cellules bien séparées après érosion.

Nous avons fait le choix ici de réaliser cette érosion manuelle à la fin, mais nous aurions aussi pu la réaliser avant de supprimer les cellules au bord.

### 2.3.7 Comptage des composantes connexes par parcours en profondeur

La dernière étape consiste à compter les composantes connexes de l'image. Deux pixels appartiennent à la même composante connexe s'ils sont en contact par un de leurs quatre côtés : on considère donc la **4-connectivité**. On considèrera que des pixels qui sont en contact par un angle n'appartiennent pas à la même composante connexe, ce qui serait le cas si on considèrait la **8-connectivité**.

Pour compter les composantes connexes, on se basera sur un algorithme de parcours en profondeur (DFS pour *depth-first search*) récursif sur les pixels blancs de l'image. Celui-ci est décrit dans l'Algorithme 2.

---

#### Algorithme 2 Algorithme de recherche en profondeur dans les pixels d'une image

---

```

1: fonction DFS(image, ligne, colonne)
2:   si image[ligne][colonne] n'a pas encore été visité alors
3:     si image[ligne][colonne] est blanc alors
4:       Marquer image[ligne][colonne] comme visité
5:       DFS(image, ligne-1, colonne)           ▷ Visite du voisin du haut
6:       DFS(image, ligne+1, colonne)           ▷ Visite du voisin du bas
7:       DFS(image, ligne, colonne-1)         ▷ Visite du voisin de gauche
8:       DFS(image, ligne, colonne+1)         ▷ Visite du voisin de droite

```

---



Une fois cet algorithme implémenté, on comptera les composantes connexes en effectuant une recherche à partir de tous les pixels de l'image. On prendra garde de ne pas visiter des pixels que l'on aura déjà visités. Chaque nouvelle recherche réussie correspondra à une nouvelle composante connexe.

## 2.4 Pour aller plus loin

### 2.4.1 Érodés ultimes

Dans la section 2.3.6, nous avons réalisé une érosion **manuelle** : nous avons fixé arbitrairement le nombre de répétitions afin de séparer suffisamment les cellules qui se touchaient, sans les faire disparaître. Il serait plus intéressant de réaliser ces érosions successives en déterminant le nombre d'itérations de manière automatique.

Un problème qui apparaît alors est que certaines cellules sont plus grosses que d'autres. En conséquence, les érosions successives supprimeront les plus petites cellules, tandis qu'elles conserveront les plus grosses. Comment savoir alors quand arrêter l'érosion ?

Une solution consiste à identifier les **érodés ultimes**, c'est-à-dire, pour chaque cellule, sa dernière version érodée avant qu'elle ne disparaisse. L'Algorithme 3 décrit cette méthode. Son principe est le suivant :

- on érode l'image (ligne 4) puis on reconstruit la version érodée dans l'image de départ (ligne 6),
- on détecte avec un OU-exclusif les éléments que l'érosion avait fait disparaître (ligne 7) : ce sont les résidus,
- on accumule les résidus avec une union (ligne 8),
- l'image courante devient l'image érodée (ligne 9),
- on répète ces opérations tant que l'image courante n'a pas été complètement érodée (ligne 10), c'est-à-dire tant qu'elle n'est pas vide.

Pour une cellule de taille suffisante, l'érosion puis la reconstruction réalisées aux lignes 4 et 6 s'annulent. En revanche, pour un érodé ultime, la reconstruction est impossible puisque l'érosion l'a fait disparaître. Ces deux cas sont discriminés par l'opération OU-exclusif puis les cas où la cellule a disparu, correspondant aux érodés ultimes, sont accumulés.

---

### Algorithme 3 Érodés ultimes

---

```

1: erodes_ultimes, erodee, residus, reconstruite : des images vides
2: courante = image
3: faire
4:   erodee = erosion(courante)
5:   Effacer reconstruite
6:   reconstruite = reconstruction(courante, erodee)
7:   residus = OU_exclusif(reconstruite, courante)
8:   erodes_ultimes = union(residus, erodes_ultimes)
9:   échanger erodee et courante
10: tant que courante n'est pas vide

```

---

La Figure 14 montre un exemple d'érodés ultimes.



FIGURE 14 – Érodés ultimes.

**Dilatation finale** Les érodés ultimes obtenus peuvent avoir des formes variées. En particulier, il est possible d'obtenir trois pixels, disposés en diagonale. Dans ce cas, ils ne seront pas identifiés comme faisant partie de la même composante connexe par l'Algorithme 2, car on considère seulement la 4-connectivité.

Une solution consiste à réaliser une dilatation finale, qui connectera ces pixels. Il ne faut pas faire plusieurs dilatations à cette étape, sinon on risque de faire se toucher à nouveau des cellules qui ont été séparées lors de la détermination des érodés ultimes.

#### 2.4.2 Étiquetage en composantes connexes : algorithme de Hoshen–Kopelman

La méthode utilisée précédemment pour l'étiquetage des composantes connexes, décrite dans la section 2.3.7, n'est pas efficace, car elle parcourt chaque case beaucoup plus de fois que nécessaire.

L'algorithme de Hoshen-Kopelman permet d'étiqueter les composantes connexes en seulement deux passes sur l'image. Cet algorithme fait appel à une structure de données *union-find*, qui permet d'exprimer des relations d'équivalence entre des éléments. Ces éléments sont alors regroupés, ou non, dans des **classes d'équivalence**. Il faut donc dans un premier temps implémenter cette structure de données.

**Structure de données *union-find*** Une structure *union-find* permet d'exprimer des relations d'équivalence entre des éléments. Deux méthodes sont définies sur cette structure de données :

- `find(i)` : trouve la classe de l'élément `i`,
- `union(i, j)` : fusionne les classes des éléments `i` et `j`.

**Implémentation d'une structure *union-find* avec un tableau** Il est possible d'implémenter une structure *union-find* de différentes manières. L'une d'elles est d'utiliser un tableau.

On fixe la convention suivante : si une case du tableau, correspondant à un élément, contient son index, alors cet index est la classe d'équivalence de l'élément. À l'inverse, si une case du tableau **ne contient pas son index**, elle contient l'index de la case qui stocke sa classe d'équivalence.

On peut implémenter les deux fonctions `find` et `union` de la manière suivante en C :

```

1 unsigned uf_find(unsigned i, unsigned* classes)
2 {
3     while (classes[i] != i) {
4         i = classes[i];
5     }
6     return i;
7 }
```

```

1 void uf_union(unsigned i, unsigned j, unsigned* classes)
2 {
3     unsigned classe_i = uf_find(i, classes);
4     unsigned classe_j = uf_find(j, classes);
5     classes[classe_i] = classe_j;      /* Affecter la classe de j à i */
6 }
```

**Exemple** Soit quatre éléments indépendants, alors le tableau `classes` est : `[0, 1, 2, 3]`. Un appel à `uf_find` sur chaque case du tableau confirme qu'il y a bien quatre classes.

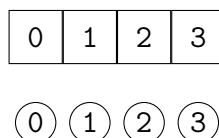


FIGURE 15 – Situation initiale.

On souhaite grouper les éléments 1 et 3 dans la même classe d'équivalence. On appelle alors `uf_union(1, 3, classes)`. Le tableau `classes` vaut ensuite : `[0, 3, 2, 3]`.

Les appels `uf_find(1, classes)` et `uf_find(3, classes)` renvoient bien la même valeur, 3, indiquant que les éléments 1 et 3 font bien partie de la même classe d'équivalence.

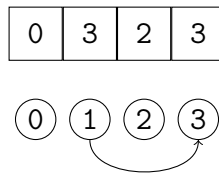


FIGURE 16 – `uf_union(1, 3, classes)`.

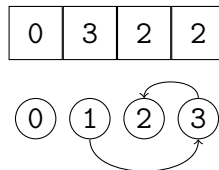


FIGURE 17 – `uf_union(1, 2, classes)`.

On souhaite grouper les éléments 1 et 2 dans la même classe d'équivalence. Le tableau `classes` vaut ensuite : `[0, 3, 2, 2]`.

Les appels `uf_find(1, classes)`, `uf_find(2, classes)` et `uf_find(3, classes)` renvoient bien la même valeur, 2, indiquant que les éléments 1, 2 et 3 font bien partie de la même classe d'équivalence.

**Première passe : étiquetage temporaire et construction des classes d'équivalence** Lors de la première passe sur l'image, à chaque pixel rencontré :

- soit ce pixel est à la frontière entre deux classes, dans ce cas on fusionnera ces deux classes avec `union` puis on lui affectera l'une des deux classes,
- soit ce pixel est en contact avec une classe existante, dans ce cas on la lui affectera après l'avoir identifiée avec `find`,
- soit on lui affectera une nouvelle classe.

Cette méthode est décrite dans l'Algorithme 4.

---

**Algorithme 4** Étiquetage en composantes connexes : première passe

---

- 1: **pour** chaque pixel de l'image en commençant en haut à gauche **faire**
  - 2:   **si** le voisin haut est étiqueté **et** le voisin gauche est étiqueté **alors**
  - 3:     Étiqueter le pixel avec le minimum des deux classes
  - 4:     Fusionner les deux classes
  - 5:   **sinon si** le voisin haut est étiqueté **et** le voisin gauche n'est pas étiqueté **alors**
  - 6:     Étiqueter le pixel avec la classe du voisin haut
  - 7:   **sinon si** le voisin gauche est étiqueté **et** le voisin haut n'est pas étiqueté **alors**
  - 8:     Étiqueter le pixel avec la classe du voisin gauche
  - 9:   **sinon**
  - 10:    Étiqueter le pixel avec une nouvelle classe
-

**Seconde passe : étiquetage final** On parcourra ensuite la grille à nouveau, en remplaçant l'étiquette de chaque pixel par sa classe d'équivalence. On pourra alors compter les classes d'équivalence, ce qui nous donnera le nombre de composantes connexes, et donc le nombre de cellules.

### 2.4.3 Optimisation des performances

**Méthode d'Otsu** La méthode d'Otsu peut être largement optimisée en évitant les calculs redondants. Pour cela, au lieu de recalculer les probabilités de classe et les moyennes empiriques pour chaque valeur du seuil, mettez-les à jour.

**Structure *union-find*** Un problème qui peut apparaître lors de la construction du tableau des équivalences dans la structure *union-find* est que de longues "chaînes" menant à la classe d'équivalence peuvent se former. Ceci était visible dans la Figure 17, où, pour obtenir la classe de l'élément 1, il faut passer par l'élément 3, pour arriver finalement à l'élément 2. On aimerait que l'élément 1 soit directement connecté à l'élément 2, comme dans la Figure 18.

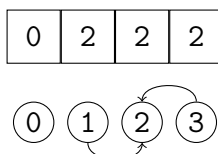


FIGURE 18 – Ce que l'on aimerait à la place de la Figure 17

Pour ça, il faut faire en sorte de ne parcourir cette longue chaîne **qu'une fois**, puis, une fois la classe d'équivalence identifiée, de **mettre à jour** la classe de l'élément de départ.

### 2.4.4 Marquage des cellules comptées

**Numérotation** Une fois les cellules comptées, vous écrirez une fonction qui placera un nombre au milieu de la cellule. Vous pourrez utiliser les chiffres donnés en Figure 19 pour savoir quels pixels colorer.

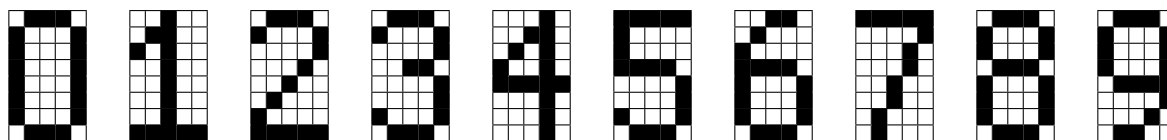


FIGURE 19 – Chiffres.

**Coloration** Une autre option pour identifier les cellules comptées est de leur affecter une couleur. Le format PGM étant dédié aux images en niveaux de gris, vous devrez vous tourner vers le **format PPM**, très similaire, mais où chaque pixel est codé par trois valeurs, correspondant aux canaux rouge, vert et bleu.

Obtenir  $n$  couleurs distinctes aussi différentes que possible les unes des autres est un problème difficile. Il ne suffit pas de tirer des couleurs équidistantes dans l'espace RGB, car cet espace est non-linéaire pour l'œil humain : des couleurs proches dans cet espace ne le sont pas forcément pour nous, et inversement. Dans un article<sup>1</sup> de 1965, Kenneth Kelly a proposé une liste de 22 couleurs de “contraste maximal”. Elles sont données, avec leurs valeurs en RGB, dans la Table 2. Vous utiliserez ces couleurs pour marquer les cellules.























Couleur	Nom	R	G	B	Couleur	Nom	R	G	B
	blanc	242	243	244		bleu	0	103	165
	noir	34	34	34		rose orangé	249	147	121
	jaune	243	195	0		violet	96	78	151
	indigo	135	86	146		doré	246	166	0
	orange	243	132	0		fuschia	179	68	108
	bleu ciel	161	202	241		jaune vert	220	211	0
	rouge	190	0	50		marron rouge	136	45	23
	beige	194	178	128		vert jaune	141	182	0
	gris	132	132	130		marron jaune	101	69	34
	vert	0	136	86		orange sanguine	226	88	34
	rose	230	143	172		vert olive	43	61	38

TABLE 2 – 22 couleurs de “contraste maximal” par Kenneth Kelly.

1. Kenneth L KELLY. “Twenty-two colors of maximum contrast”. In : **Color Engineering** 3.26 (1965), p. 26-27.

### 3 Sujet n° 2 : Simulation du problème à N corps

*D'après des discussions avec M. Richard BRESSOUX et M. Patrick KOCELNIAK.*

#### Objectif

L'objectif de ce projet est de **résoudre numériquement** le problème à N corps, c'est-à-dire d'utiliser des **méthodes numériques** pour résoudre les **équations du mouvement** de N corps interagissant dans le cadre de la théorie de la **gravitation de Newton**.

#### 3.1 Introduction

##### 3.1.1 Grandeurs astronomiques

La valeur de la constante gravitationnelle en unités SI est donnée dans l'équation (4).

$$G = 6.674\,30 \times 10^{-11} \text{ m}^3 \cdot \text{kg}^{-1} \cdot \text{s}^{-2} \quad (4)$$

Puisqu'on va simuler des systèmes à l'échelle astronomique, les mètres et les kilogrammes ne sont pas très adaptés. Nous utiliserons une constante gravitationnelle modifiée, donnée dans l'équation (5).

$$\tilde{G} = 9.508\,29 \times 10^3 \text{ AL}^3 \cdot \text{M}_{\odot}^{-1} \cdot \text{T}^{-2} \quad (5)$$

avec :

- AL : une année-lumière,
- $\text{M}_{\odot}$  : la masse du soleil, environ  $2 \times 10^{30} \text{ kg}$ .
- T : la période de révolution d'une étoile située à  $1 \times 10^3 \text{ AL}$  du trou noir central de la Voie lactée

##### 3.1.2 Équations de la mécanique

Nous définissons des vecteurs à deux composantes car nous travaillons dans le plan. C'est adapté pour simuler une galaxie par exemple, où tous les objets peuvent être considérés comme étant sur le même plan : lorsqu'on l'observe de la Terre, la Voie lactée nous apparaît comme une "ligne".

**Notations** Nous utilisons les notations suivantes :

- la masse d'un corps  $i$  est notée  $m_i$
- la force gravitationnelle exercée par un corps  $j$  sur un corps  $i$  est un vecteur noté  $\vec{F}_{j \rightarrow i}$
- la position d'un corps est un vecteur noté  $\vec{r}_i$
- la vitesse d'un corps est un vecteur noté  $\vec{v}_i$
- l'accélération d'un corps est un vecteur noté  $\vec{a}_i$

**Principe fondamental de la dynamique** Selon la deuxième loi de Newton, la somme des forces extérieures exercées sur un corps est égale à sa masse multipliée par son accélération (voir équation (6)).

$$\sum_{i \neq j} \vec{F}_{j \rightarrow i}(t) = m_i \times \vec{a}_i(t) \quad (6)$$

où la force gravitationnelle est définie dans l'équation (7).

$$\vec{F}_{j \rightarrow i}(t) = -m_i \times m_j \times G \times \frac{\vec{r}_i - \vec{r}_j}{d^3} \quad (7)$$

où  $G$  est la constante universelle de la gravitation et  $d$  est la distance entre les deux corps (voir équation (8)).

$$d = \|\vec{r}_i - \vec{r}_j\| \quad (8)$$

**Systèmes d'équations** Les deux systèmes d'équations qu'on va résoudre sont les suivants :

$$\forall i \in \{1; \dots; n\} \quad \frac{\partial \vec{r}_i(t)}{\partial t} = \vec{v}_i(t) \quad (9)$$

$$\frac{\partial \vec{v}_i(t)}{\partial t} = \vec{a}_i(t) = \frac{1}{m_i} \sum_{i \neq j} \vec{F}_{j \rightarrow i}(t) \quad (10)$$

Nous allons les résoudre approximativement avec des méthodes numériques itératives qui, à partir des conditions initiales, donnent à chaque pas de simulation les nouvelles valeurs des positions  $\vec{r}_i$  et des vitesses  $\vec{v}_i$  de tous les corps.

## 3.2 Étapes préparatoires

### 3.2.1 Représentation des grandeurs physiques et des objets

**Représentation des scalaires et des vecteurs** Les nombres réels seront représentés avec une représentation à virgule flottante de précision moyenne, c'est-à-dire en utilisant le type `double` (plus précis que `float`, moins que `long double`). Vous définirez une structure, voire un nouveau type, adaptée pour représenter les vecteurs.

**Représentation des corps** Vous définirez une structure, voire un nouveau type, adaptée pour représenter les corps et leurs propriétés (position, vitesse, etc).

**Représentation de l'univers** On considérera que l'univers est un carré, dont les côtés opposés sont connectés (voir Figure 20). Tout corps qui sort par le bord gauche de l'univers y rentre à nouveau par le bord droit et inversement. De même, tout corps qui sort par le bord haut de l'univers y rentre à nouveau par le bord bas et inversement.



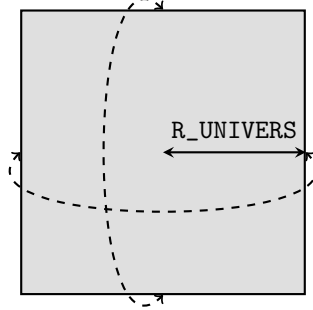


FIGURE 20 – Univers.

### 3.2.2 Méthode d'Euler

La méthode d'Euler est la méthode la plus simple de résolution numérique d'équations différentielles. C'est aussi une des moins précises. Nous l'utiliserons pour résoudre les systèmes d'équations pour les vitesses et les positions des corps donnés dans les équations (9) et (10).

Soit une équation différentielle définie de la manière suivante :

$$y'(t) = f(t, y(t)) \quad \text{avec comme condition initiale } y(t_0) = y_0 \quad (11)$$

alors pour un pas de simulation  $\Delta_t$  on a, après un développement au premier ordre :

$$y(t + \Delta_t) = y(t) + \Delta_t \times f(t, y(t)) \quad (12)$$

En appliquant cette méthode aux équations (9) et (10), on a :

$$\vec{r}(t + \Delta_t) = \vec{r}(t) + \Delta_t \times \frac{\partial \vec{r}(t)}{\partial t} = \vec{r}(t) + \Delta_t \times \vec{v}(t) \quad (13)$$

$$\vec{v}(t + \Delta_t) = \vec{v}(t) + \Delta_t \times \frac{\partial \vec{v}(t)}{\partial t} = \vec{v}(t) + \Delta_t \times \vec{a}(t) \quad (14)$$

On peut donc en déduire l'algorithme présenté dans l'Algorithme 5, qui calcule les positions successives des corps à partir de leur position initiale.

### 3.2.3 Génération d'un GIF animé

**Génération des images** À partir de la configuration initiale des positions et des vitesses pour chaque corps, nous allons calculer une suite de positions et de vitesses. À chaque pas de simulation, on générera une image de l'univers. Vous utiliserez le format PGM binaire, décrit dans l'annexe B.

---

**Algorithme 5** Méthode d'Euler appliquée au calcul des positions et des vitesses (corrigée)

---

- 1: Initialiser les positions et les vitesses  $\vec{r}_i(0)$  et  $\vec{v}_i(0)$  des corps.
  - 2: **pour** pas  $\leftarrow 1$  à NB\_PAS **faire**
  - 3:   **pour** corps  $\leftarrow 1$  à NB\_CORPS **faire**
  - 4:      $\vec{F}_i \leftarrow$  Calculer la force gravitationnelle totale sur le corps  $i$  ▷ Équation (7)
  - 5:   **pour** corps  $\leftarrow 1$  à NB\_CORPS **faire**
  - 6:      $\vec{r}_i \leftarrow \vec{r}_i + \Delta t \cdot \vec{v}_i$  ▷ Équation (13)
  - 7:     Corriger la position  $\vec{r}_i \leftarrow \text{correct\_position}(\vec{r}_i)$
  - 8:   **pour** corps  $\leftarrow 1$  à NB\_CORPS **faire**
  - 9:      $\vec{v}_i \leftarrow \vec{v}_i + \Delta t \cdot \frac{\vec{F}_i}{m_i}$  ▷ Équation (14)
- 

En partant d'une image de base noire, vous placerez chaque corps d'après sa position  $\vec{r}_i$ . Attention, l'image n'a pas les mêmes dimensions que l'univers : à vous de faire la conversion. Vous écrirez ensuite dans un fichier pour y stocker l'image.

**Génération du GIF animé** Le programme imagemagick permet de générer un GIF animé à partir d'une liste d'images. Vous vous inspirerez de la commande suivante pour cela :

```
$ convert -delay 5 -loop 0 ./images/*.pgm anim.gif
```

### 3.3 Version minimale

#### 3.3.1 Paramètre en ligne de commande

Le nombre de pas de simulation devra être passé en ligne de commande à l'exécutable. Par exemple, pour simuler cent pas, on lancera la commande suivante, qui génère cent images :

```
$ ./simulation_N_corps 100
```

#### 3.3.2 Configuration initiale basique

On pourra simuler, comme base de départ, un univers de rayon  $2 \times 10^3$  AL comportant :

- un trou noir central immobile, de masse  $m_{\text{trou noir}} = 4.152 \times 10^6 M_{\odot}$ ,
- une étoile, de 1 masse solaire, distante de  $d = 1 \times 10^3$  AL du trou noir et placée à sa droite, et dont le vecteur vitesse initial, normal au vecteur position, a pour norme  $\sqrt{\frac{m_{\text{trou noir}} \times G}{d}}$ . Cela donne une trajectoire circulaire à l'étoile.

En simulant ce système avec un pas de  $\Delta_t = 0.01$ , l'étoile tourne autour du trou noir et revient à sa position initiale après cent pas de simulation.

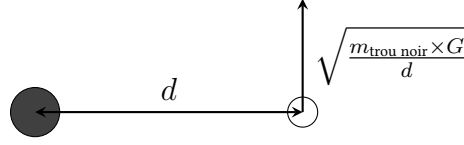


FIGURE 21 – Configuration initiale avec un trou noir central et une étoile en orbite.

### 3.3.3 Méthode de Verlet

La méthode d'Euler décrite en Section 3.2.2 a l'inconvénient d'induire une erreur assez élevée. Une autre méthode plus précise et bien adaptée aux calculs de gravitation est la méthode de Verlet. Les équations (9) et (10) sont cette fois développées au deuxième ordre :

$$\begin{aligned}\vec{r}(t + \Delta_t) &= \vec{r}(t) + \Delta_t \times \frac{\partial \vec{r}(t)}{\partial t} + \frac{\Delta_t^2}{2} \times \frac{\partial^2 \vec{r}(t)}{\partial t^2} \\ &= \vec{r}(t) + \Delta_t \times \vec{v}(t) + \frac{\Delta_t^2}{2} \times \vec{a}(t)\end{aligned}\quad (15)$$

$$\vec{v}(t + \frac{\Delta_t}{2}) = \vec{v}(t) + \frac{\Delta_t}{2} \times \frac{\partial \vec{v}(t)}{\partial t} + \frac{\Delta_t^2}{8} \times \frac{\partial^2 \vec{v}(t)}{\partial t^2} \quad (16)$$

$$\vec{v}(t + \frac{\Delta_t}{2}) = \vec{v}(t + \Delta_t) - \frac{\Delta_t}{2} \times \frac{\partial \vec{v}(t + \Delta_t)}{\partial t} + \frac{\Delta_t^2}{8} \times \frac{\partial^2 \vec{v}(t + \Delta_t)}{\partial t^2} \quad (17)$$

Par soustraction de l'équation (17) et de l'équation (16), on obtient :

$$\begin{aligned}\vec{v}(t + \Delta_t) &= \vec{v}(t) + \frac{\Delta_t}{2} \times \left( \frac{\partial \vec{v}(t)}{\partial t} + \frac{\partial \vec{v}(t + \Delta_t)}{\partial t} \right) \\ &= \vec{v}(t) + \frac{\Delta_t}{2} \times (\vec{a}(t) + \vec{a}(t + \Delta_t))\end{aligned}\quad (18)$$

#### ⚠ Et les dérivées secondes de la vitesse ?

La différence des dérivées secondes de la vitesse  $\frac{\partial^2 \vec{v}(t)}{\partial t^2}$  et  $\frac{\partial^2 \vec{v}(t + \Delta_t)}{\partial t^2}$  étant en  $\mathcal{O}(\Delta_t)$ , la multiplication par  $\frac{\Delta_t^2}{8}$  donne un terme en  $\mathcal{O}(\Delta_t^3)$ , que l'on exclut donc dans le cadre du développement limité à l'ordre 2.

On peut donc en déduire la méthode présentée dans l'Algorithme 6.

## 3.4 Pour aller plus loin

### 3.4.1 Configuration initiale avancée

À partir de la configuration initiale basique présentée ci-dessus en Section 3.3.2, on peut construire une configuration initiale plus avancée, ressemblant à une galaxie. Pour cela, on

---

**Algorithme 6** Méthode de Verlet appliquée au calcul des positions et des vitesses (corrigée)

---

- 1: Initialiser les positions et les vitesses  $\vec{r}_i(0)$  et  $\vec{v}_i(0)$  des corps
  - 2: Initialiser les forces gravitationnelles  $\vec{F}_i^{\text{prev}}$  à  $t = 0$
  - 3: **pour** pas  $\leftarrow 1$  à NB\_PAS **faire**
  - 4:   **pour** corps  $\leftarrow 1$  à NB\_CORPS **faire**
  - 5:      $\vec{F}_i^{\text{prev}} \leftarrow$  Calculer la force gravitationnelle totale sur le corps  $i$  ▷ Équation (7)
  - 6:   **pour** corps  $\leftarrow 1$  à NB\_CORPS **faire**
  - 7:      $\vec{r}_i \leftarrow \vec{r}_i + \Delta t \cdot \vec{v}_i + \frac{\Delta t^2}{2} \cdot \frac{\vec{F}_i^{\text{prev}}}{m_i}$  ▷ Équation (15)
  - 8:     Corriger la position  $\vec{r}_i \leftarrow \text{correct\_position}(\vec{r}_i)$
  - 9:   **pour** corps  $\leftarrow 1$  à NB\_CORPS **faire**
  - 10:      $\vec{F}_i \leftarrow$  Calculer la force gravitationnelle totale sur le corps  $i$
  - 11:   **pour** corps  $\leftarrow 1$  à NB\_CORPS **faire**
  - 12:      $\vec{v}_i \leftarrow \vec{v}_i + \frac{\Delta t}{2} \cdot \left( \frac{\vec{F}_i}{m_i} + \frac{\vec{F}_i^{\text{prev}}}{m_i} \right)$  ▷ Équation (18)
- 

placera à nouveau un trou noir central, puis des étoiles dans une couronne périphérique. On procédera ainsi pour chaque étoile à placer :

- A. Tirer aléatoirement une distance  $d$  dans la couronne et un angle  $\alpha \in [0; 2\pi)$
- B. Dédire la position initiale en projetant le vecteur de norme  $d$  sur les directions  $x$  et  $y$
- C. Dédire la vitesse initiale en projetant le vecteur de norme  $\sqrt{\frac{m_{\text{trou noir}} \times G}{d}}$  normal à  $d$

**Paramètre de *softening*** Dans le tirage aléatoire précédent, il peut arriver que certaines étoiles soient très proches les unes des autres. Cela arrive plus fréquemment dans les univers petits. Or, la force gravitationnelle entre ces étoiles peut alors devenir très importante, entraînant des mouvements très rapides sur les images générées. Une solution pour remédier à ce problème consiste à ajouter une constante  $C_{\text{soft}}$  dite de *softening* (adoucissement) à la distance entre les deux corps, calculée via l'équation (8). On a alors :

$$d = \|\vec{r}_i - \vec{r}_j + C_{\text{soft}}\| \quad (19)$$

Ainsi, même si les corps sont arbitrairement proches, la force gravitationnelle sera égale à celle calculée dans le cas où les corps sont au moins distants de  $C_{\text{soft}}$ . En pratique, on pourra choisir  $C_{\text{soft}} = 0.5 \text{ AL}$ .

### 3.4.2 Quadtree : calcul approché de la force gravitationnelle

Lors du calcul de la force gravitationnelle totale s'exerçant sur un corps, la configuration présentée en Figure 23 peut se produire : le corps de gauche est très éloigné des autres corps, et ces derniers sont proches les uns des autres. Dans ce cas, le groupe des corps de droite est considéré comme un seul corps, en calculant son barycentre et sa masse totale.

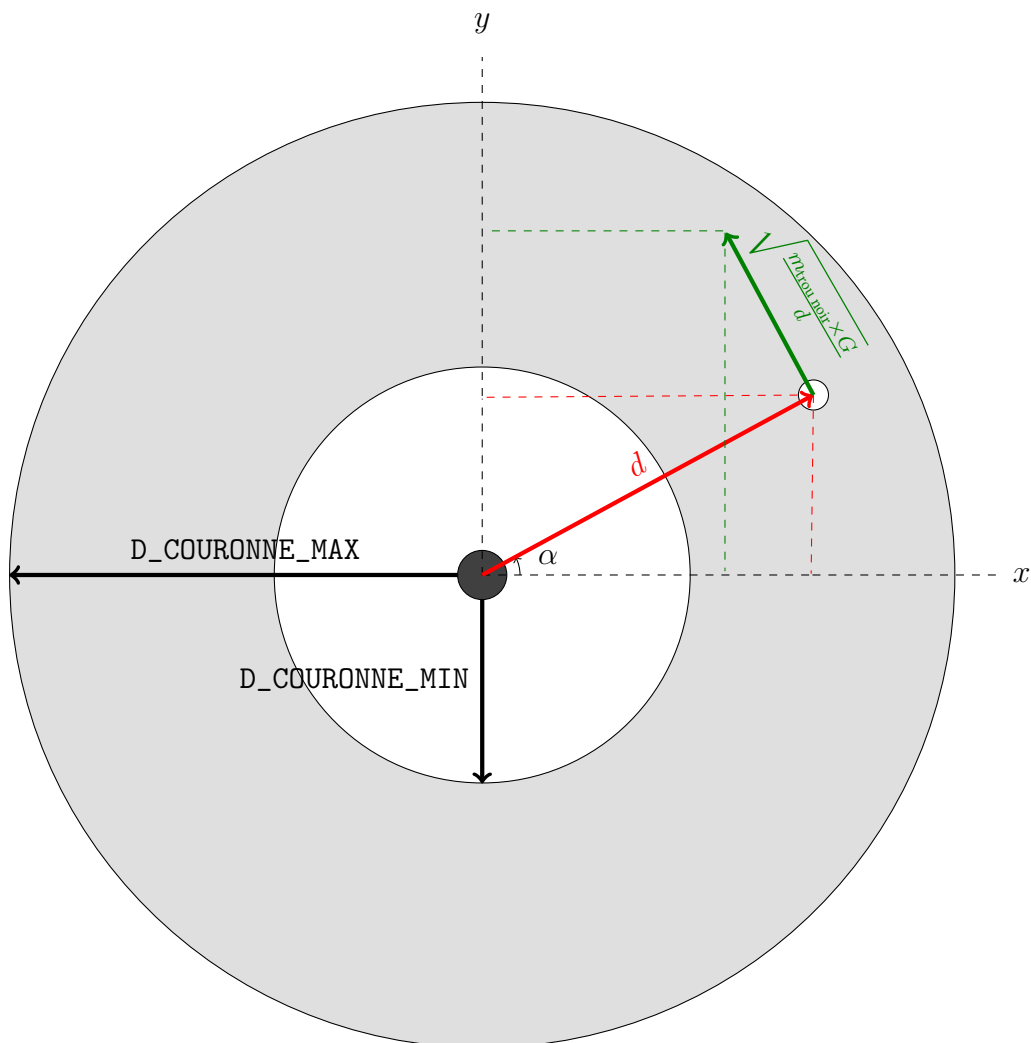


FIGURE 22 – Configuration initiale avancée avec un trou noir central et un grand nombre d'étoiles en orbite dans une couronne (une seule étoile est représentée).



FIGURE 23 – Situation dans laquelle le calcul individuel des forces peut être approché.

**Structure de données** Afin de savoir quels corps peuvent être regroupés, nous allons organiser les corps de l'univers en **quadtree**. Un quadtree est un arbre dont chaque nœud a **quatre** successeurs : nord-ouest, nord-est, sud-ouest, sud-est. La Figure 24 montre comment les corps de l'univers peuvent être organisés en quadtree.

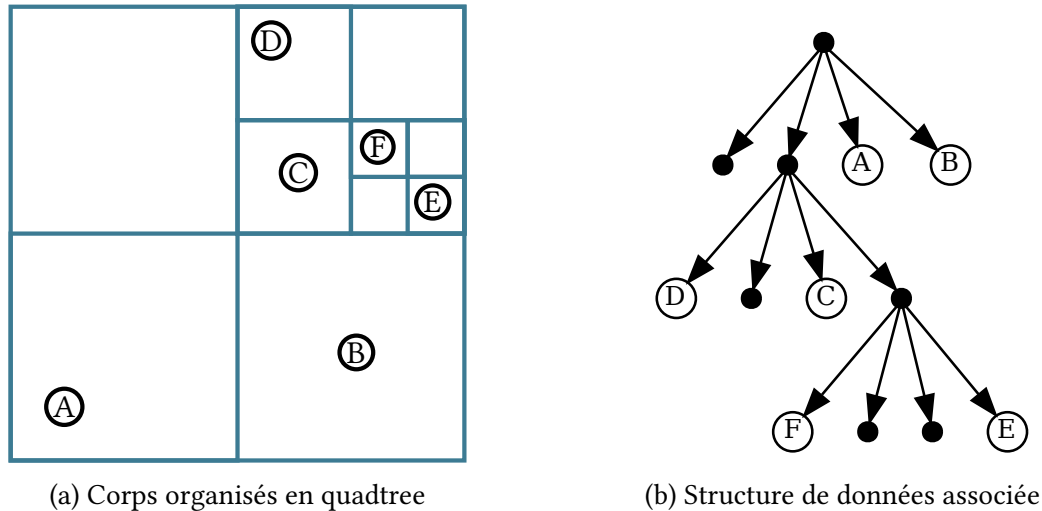


FIGURE 24 – Quadtree

**Construction** Pour construire le quadtree, on y ajoute les corps les uns après les autres, connaissant leur position. On doit respecter les deux règles suivantes lors de la construction du quadtree :

- chaque corps doit être **seul** dans un quadrant du quadtree à un niveau donné,
- les corps se trouveront nécessairement **sur les feuilles** du quadtree.

Si deux corps se retrouvent dans le même quadrant, alors il faut le **subdiviser** et tenter de faire descendre les deux corps plus bas dans le quadtree, jusqu'à ce qu'ils se séparent. Pour savoir dans quel quadrant un corps doit descendre, on compare ses coordonnées à celles du centre du quadtree.

Les étapes successives de la construction du quadtree de la Figure 24b sont :

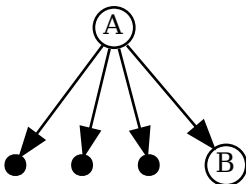
Étape 1 : Initialement, le quadtree est vide.



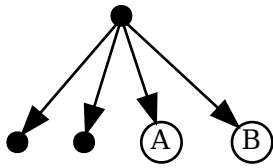
Étape 2 : On ajoute le corps A.



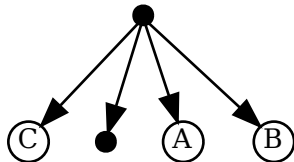
Étape 3 : On veut ajouter le corps B, or le quadrant est plein, car il est occupé par A. Il faut donc le subdiviser. Le corps B descend alors dans le quadrant sud-est.



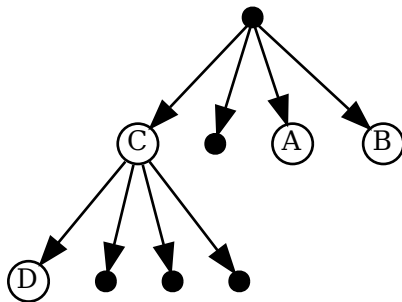
Étape 4 : Il faut ensuite faire descendre A, car il n'est pas sur une feuille. Le corps A descend alors dans le quadrant sud-ouest.



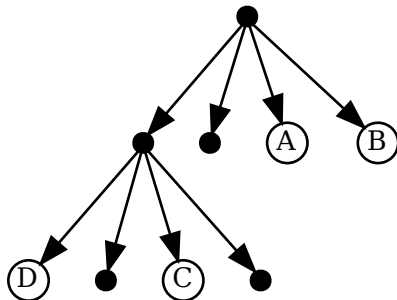
Étape 5 : On ajoute le corps C,



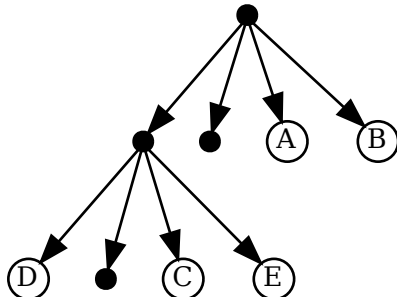
Étape 6 : On veut ajouter le corps D, or le quadrant est plein, car il est occupé par C. Il faut donc le subdiviser. Le corps D descend alors dans le quadrant nord-ouest.



Étape 7 : Il faut ensuite faire descendre C, car il n'est pas sur une feuille. Le corps C descend alors dans le quadrant sud-ouest.

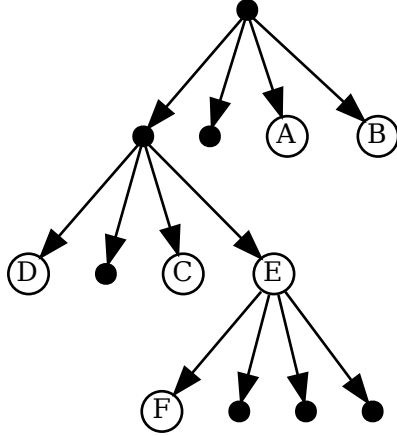


Étape 8 : On ajoute le corps E,

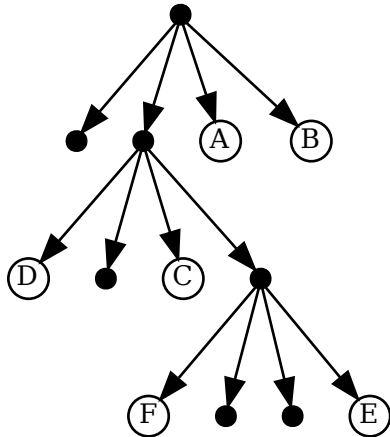


Étape 9 : On veut ajouter le corps F, or le quadrant est plein, car il est occupé par E. Il faut donc

le subdiviser. Le corps F descend alors dans le quadrant nord-ouest.



Étape 10 : Il faut ensuite faire descendre E, car il n'est pas sur une feuille. Le corps E descend alors dans le quadrant sud-est.



**Propagation des masses** Une fois les corps ajoutés, il faut calculer les barycentres et les masses totales des quadrants peuplés. La masse totale  $m_{\text{totale}}$  est la somme des masses. Les coordonnées du barycentre sont calculées grâce à l'équation (20).

$$\vec{r}_{\text{barycentre}} = \frac{1}{m_{\text{totale}}} \sum m_i \vec{r}_i \quad (20)$$

**Utilisation** Une fois le quadtree construit, on l'utilise pour calculer la force gravitationnelle totale s'exerçant sur un corps. Pour cela, au lieu d'utiliser une boucle **for** qui parcourt systématiquement tous les corps, comme aux lignes ?? des Algorithmes 5 et 6, on parcourt le quadtree en sommant les contributions des corps des différents quadrants.

Si on s'en tient à cela, le quadtree ne présente pas d'avantage par rapport au parcours simple, car on descend systématiquement jusqu'à arriver aux corps individuels. Il faut donc introduire une **condition** qui, si elle est vérifiée, stoppe la descente dans le quadtree. On consi-



dèrera alors le barycentre et la masse totale des corps présents dans le sous-quadtrees, au lieu de poursuivre la descente.

La condition que l'on utilise pour savoir si l'on continue à descendre dans le quadtree ou si l'on approxime est donnée dans l'équation (21) et paramétrée par un seuil noté  $\theta$ .

$$\frac{\|\vec{r}_{\text{corps}} - \vec{r}_{\text{barycentre}}\|}{\text{côté}_{\text{quadtree}}} > \theta \quad (21)$$

Ainsi, si le corps est suffisamment éloigné du quadtree considéré et que ce quadtree est suffisamment petit, alors on approxime la contribution de chacun des corps du quadtree en utilisant le barycentre et la masse totale (voir Figure 25). Il vous faudra fixer le paramètre  $\theta$ . Plus  $\theta$  est petit, plus les calculs sont précis, le cas où  $\theta = 0$  correspondant au parcours exhaustif. Plus  $\theta$  est grand, plus les calculs sont rapides, mais on perd alors en précision.

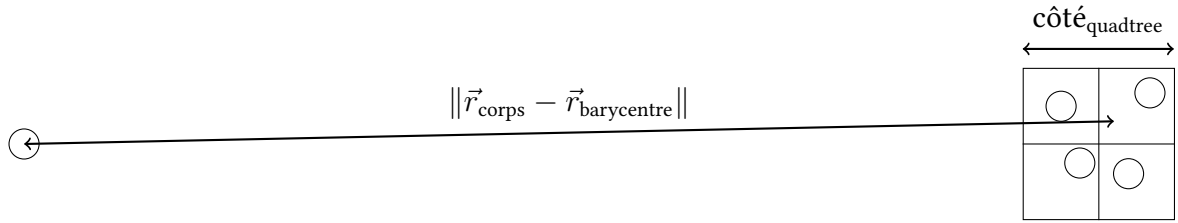


FIGURE 25 – Situation dans laquelle le calcul individuel des forces peut être approché.

Utiliser une structure de quadtree fait gagner un temps significatif lors du calcul des forces, et permet ainsi de simuler des systèmes avec un nombre bien plus important de corps en un temps raisonnable.

### 3.4.3 Lecture de la configuration initiale dans un fichier

La configuration initiale des corps est pour l'instant fixée dans le code source. Il serait beaucoup plus pratique de pouvoir la lire dans un fichier de configuration.

### 3.4.4 Mesure du temps d'exécution

Mesurez le temps pris par votre programme pour simuler un système à  $n$  corps, en faisant varier  $n$ . Tracez ensuite l'évolution du temps de calcul en fonction de  $n$ , pour déduire la **complexité** de votre implémentation.

### 3.4.5 Optimisations

**Troisième loi de Newton ou principe d'action-réaction** Souvenez-vous que la force exercée par un corps A sur un corps B est égale à la force exercée par le corps B sur le corps A, mais de sens opposé, comme résumé dans l'équation (22).

$$\vec{F}_{j \rightarrow i} = -\vec{F}_{i \rightarrow j} \quad (22)$$

**Pré-calcul du produit  $m_i \times G$**  On calcule très souvent le produit  $m_i \times G$ . Il serait judicieux de la calculer une seule fois pour chaque corps, puis de la stocker pour plus tard.

**Double calcul de la force gravitationnelle totale** Dans la méthode de Verlet, on calcule à chaque pas la force gravitationnelle totale à l'instant  $t$  et à l'instant  $t + \Delta_t$ . Or l'instant  $t$  pour le pas  $n$  est l'instant  $t + \Delta_t$  du pas  $n - 1$ . On peut donc garder en mémoire la valeur de la force gravitationnelle totale à un pas donné pour la réutiliser au pas suivant.

### 3.4.6 Méthodes numériques avancées

Mettez en œuvre des méthodes numériques plus avancées, comme les **méthodes de Runge-Kutta**, souvent plus lourdes en calcul mais plus précises.

## 4 Évaluation

### 4.1 Compétences

Ce projet a pour ambition de renforcer ou vous faire acquérir les compétences suivantes.

**Concevoir/réaliser des solutions techniques** Cette compétence valide vos capacités à :

- ✓ produire une solution fonctionnelle à un problème technique : votre code répond aux attentes sujet et fonctionne correctement,
- ✓ vous appuyer sur vos connaissances : mettre en œuvre sa connaissance du langage C,
- ✓ respecter le délai imparti : livrer le projet à la date fixée,
- ✓ tester la solution proposée : utiliser les outils de débogage, de gestion de version, de gestion de la mémoire, prouver que les codes sont robustes, mettre en place des procédures de test explicites,
- ✓ proposer une solution compréhensible et réutilisable : un code structuré en plusieurs fichiers source et d'en-tête cohérents, des répertoires séparés contenant les fichiers de données, sources, exécutables, en-tête, objets, un fichier `Makefile` pour compiler, un fichier `README` expliquant comment compiler et utiliser le programme.

**Coopérer dans une équipe en mode projet** Cette compétence valide vos capacités à :

- ✓ trouver votre place dans l'équipe : chacun(e) réalise des tâches mais participe à toutes les grandes tâches du projet (*il n'y a pas un codeur, un rédacteur, un testeur : tous les membres doivent participer à toutes les tâches du projet*),
- ✓ travailler avec rigueur en respectant les délais : les tâches sont effectuées correctement et dans les temps,
- ✓ présenter de manière claire et concise les résultats obtenus : le rapport est clair, bien écrit, concis et explicite tous les éléments exprimés dans le sujet/cahier des charges, la démonstration de votre projet prouve qu'il fonctionne correctement, le fichier `README` est précis.

### 4.2 Notation et Barème Indicatif par rapport à la Compétence

#### A. Qualité et performance des codes

- (a) **Factorisation du code** : découpage en fonctions, en fichiers source et entête cohérents, structuration en répertoire (sources, entete, tests, objet, binaire).
- (b) **Fichiers d'en-tête** : les fichiers `.h` avec le préprocesseur `#define` et `#include` sont correctement utilisés sans problème de référencement.
- (c) **Lisibilité et style du code**
  - Facile de découvrir la fonctionnalité du code sans commentaires.

- Code commenté lorsque la fonctionnalité n'est pas explicite.
- Refactoring du code : réécriture du code pour le rendre plus court et plus lisible.
- Indentation régulière.
- Les fonctions respectent la règle d'être  $\leq 30$  lignes de code.
- (d) **Programmation défensive** : Exécution d'une vérification préalable et/ou postérieure à la condition sur les arguments des fonctions et le bloc de code lorsque cela est nécessaire (par exemple, division par 0, condition if et exit(-1), utilisation de assert()).
- (e) **État du logiciel** : ce qui fonctionne et testées, ce qui ne fonctionne pas parmi les fonctionnalités demandées.
- (f) **Performance du code** : vérification du temps d'exécution et vérification de la consommation de mémoire.
- (g) **Structure des données** :
  - Choix justifié des structures de données.
  - Utilisation correcte de types précis (e.g : float au lieu de double si double n'est pas nécessaire).
  - Types qui gèrent les tableaux et la manipulation dynamique de la mémoire pour produire un code performant.

## B. Outils de développement

- (a) **Compilation** : Compilation sans avertissement ni erreur. Utilisation correcte du Makefile pour intégrer toutes les fonctions dans les fichiers .c et les fichiers d'entête .h.
- (b) **Git** : Utilisation fréquente de Git avec des messages explicites de commit. Fichier ReadMe à jour qui illustre comment compiler et exécuter leur projet.
- (c) **Valgrind** :
  - Vérifier les défauts de mémoire.
  - S'assurer de la libération de la mémoire allouée
  - Vérifier l'accès correct à la lecture et à l'écriture sur une zone de mémoire spécifique.
- (d) **Gdb** : déboguer les programmes C avec Gdb notamment en cas d'erreur de segmentation et non pas en utilisant un simple printf pour l'affichage.

## C. Gestion de projet et avancement

- (a) Programmation en binôme.
- (b) Répartition des tâches dans le temps et au sein des binômes.
- (c) Planification : ce qui a été fait, ce qui doit être fait.

VALIDATION COMPETENCES (pour A, B et C)	
0	Absence, ou travail pas fait
1p	NON ACQUIS - insuffisant (travail de qualité minimaliste, incomplet, erroné)
2p	PARTIELLEMENT ACQUIS - travail fait en baclant, à moitié, ou fait mais pas justifié
3p	ACQUIS - très bien voire exceptionnel, selon attentes

TABLE 3 – Barème indicatif

### 4.3 Évaluation formative

Lors de la séance 3, nous réaliserons une évaluation **formative**. L'objectif de cette évaluation est de vous permettre :

- d'estimer votre **progression** dans le projet : êtes-vous dans les temps ou en retard ?
- d'identifier **rapidement et clairement** vos **points forts** et vos **points faibles** : qu'est-ce que vous faites bien, et qu'est-ce que vous devriez améliorer.

Cette évaluation formative n'est **pas notée**, par définition. Son rôle est uniquement de vous accompagner dans le processus d'apprentissage.

La **barème indicatif d'évaluation** que nous utiliserons est donnée en Table 3. Pour chaque critère évalué (A, B ou C), nous indiquerons s'il est acquis, partiellement acquis ou non acquis par rapport aux compétences identifiées en Section 4.1 et sur la base de la notation dans Section 4.2.

### 4.4 Évaluation sommative

L'évaluation **sommative** consiste à vous attribuer une note pour évaluer la qualité de votre projet. **La note finale sera la somme de cinq notes :**

- Rapport de la première séance** : analyse du problème (10% de la note finale). A la fin de la première séance, vous devez avoir une vision claire des grandes étapes de votre programme et vous ferez un court document décrivant :
  - quelle est votre vision des étapes principales ? (vous pouvez dessiner un diagramme concernant la conception),
  - les types de données utilisées, en explicitant le rôle de chaque élément des structures,
  - les modules (couple de fichiers .c/.h) et le rôle de chaque module, et dans chaque module, les prototypes de fonctions essentielles (le rôle exact de la fonction, le rôle de chaque paramètre et son mode de passage (par valeur ou par adresse)),
  - la répartition du travail entre les membres de l'équipe : quelles sont les fonctions qui seront réalisées par chaque membre de l'équipe et pour quelle échéance ?
- Suivi de l'avancement** (15% de la note finale) Le suivi de l'avancement noté sera effectué lors de la **sixième séance**.

- C. **Soutenance (20% de la note finale)** : Lors de la soutenance, d'une durée de dix minutes, vous présenterez le fonctionnement de votre programme en en faisant une **démonstration**. Vous ne montrerez pas votre code. Vous répondrez ensuite à quelques questions posées à propos de la mise en œuvre de votre projet.
- D. **Rapport final (15% de la note finale)**. Vous ferez un rapport court (3 à 5 pages) explicitant les points suivants :
- (a) État du logiciel : ce qui fonctionne, ce qui ne fonctionne pas parmi les fonctionnalités demandées.
  - (b) Justifier les structures de données que vous avez choisi.
  - (c) Méthodologie utilisée pour les tests, commandes qui permettent de tester les fonctionnalités demandées.
  - (d) Analyse des performances et mémoire.
  - (e) Outils de développement utilisés.
  - (f) Organisation au sein de l'équipe, Réunions et Planning effectif, qui a fait quoi, ce qui prouve que vous avez acquis la compétence Coopérer dans une équipe ou en mode projet.
  - (g) Notes proposées pour chacun des membres de l'équipe. La somme des notes de l'équipe doit être égale à 20.
  - (h) Conclusion
- E. **Évaluation de votre dépôt Git final** du projet (40% de la note finale). Le rendu final se fait sur votre dépôt git commun aux membres du projet, qui constitué :
- (a) du code source du projet (fichiers source et de test .c et fichiers d'en-tête .h),
  - (b) d'un fichier Makefile pour la compilation,
  - (c) d'un fichier README.md qui indiquera comment compiler et utiliser le programme.

Les enseignants récupéreront votre projet sur votre dépôt Git et l'évalueront. Lors de la notation, l'accent sera mis sur la cohérence, la propreté, en un mot le **professionnalisme** du rendu final. Ce projet n'est **pas un exercice d'informatique** : le but n'est pas seulement qu'à la fin « ça marche ». Attachez-vous à **livrer** un projet cohérent et qui fonctionne parfaitement.

### **Plagiat**

Tout plagiat, qu'il soit interne ou externe à Phelma, est interdit. Le logiciel anti-plagiat Compilatio sera utilisé pour le détecter. Si un plagiat manifeste est avéré, le ou les binômes concernés seront évidemment **sanctionnés**.

## A Rappels de C

Nous vous (re)donnons ici quelques informations sur le langage C qui vous seront utiles. Pour chaque fonction, nous vous donnons une brève explication de ce qu'elle fait. Pour en savoir plus, en particulier pour connaître les paramètres, visitez les pages de manuel. Par exemple, pour accéder au manuel de `printf` :

```
$ man 3 printf
```

Vous trouverez des informations **fiables** sur le langage C sur les sites suivants :

- <https://www.cplusplus.com/reference/clibrary/>
- <https://en.cppreference.com/w/c>
- <http://c-faq.com/>

Vous serez prudents quant aux informations trouvées sur <https://stackoverflow.com>. Vous éviterez les sites suivants :

- <https://www.tutorialspoint.com/>
- <https://www.geeksforgeeks.org/>
- <https://koor.fr>
- <https://www.delftstack.com/>

**Précédence** Les opérateurs du langage C sont considérés par ordre de précédence. Celui-ci est rappelé dans le tableau 4. Plus un opérateur est haut dans ce tableau, plus il est prioritaire.

### Travailler avec des fichiers

`fopen` : ouvrir un fichier

`fgets` : lire une chaîne de caractères de longueur maximale donnée dans un fichier

`fputs` : écrire une chaîne de caractères dans un fichier

`fprintf` : écrire une chaîne de caractères formatée dans un fichier

`fread` : lire dans un fichier binaire

`fwrite` : écrire dans un fichier binaire

`fclose` : fermer un fichier

### Allocation dynamique de la mémoire

`malloc` : allouer une zone mémoire de  $n$  octets

- ne pas caster la valeur de retour de `malloc` ([voir ici pourquoi](#))
- utiliser l'un des **types principaux** du langage C comme paramètre de `sizeof`
- `malloc` prend un paramètre : la taille de la zone en octets.

Opérateur(s)	Associativité
() [] -> . ++ --	de gauche à droite
! ~ ++ -- + - (type) * & sizeof	de droite à gauche
* / %	de gauche à droite
+ -	de gauche à droite
<< >>	de gauche à droite
< <= > >=	de gauche à droite
== !=	de gauche à droite
&	de gauche à droite
^	de gauche à droite
	de gauche à droite
&&	de gauche à droite
	de gauche à droite
?:	de droite à gauche
= += -= *= /= %= <<= >>= & = ^=  =	de droite à gauche
,	de gauche à droite

TABLE 4 – Précédence des opérateurs en C.

`calloc` : allouer une zone mémoire de  $n$  octets et l’initialiser à zéro

- `calloc` prend deux paramètres : le nombre de “cases” mémoire dans la zone, et la taille d’une “case”, en octets.

`free` : libère une zone mémoire

### Nombres (pseudo-)aléatoires

`srand` : initialise le générateur de nombres pseudo-aléatoires

`rand` : renvoie un entier aléatoire entre 0 et `RANDMAX` inclus

`rand_r` : renvoie un entier aléatoire entre 0 et `RANDMAX` inclus (remplace `rand` lorsqu’on utilise plusieurs *threads*)

### Pointeurs et adresses

`*p` : la valeur pointée par `p`

`&a` : l’adresse de la variable `a`

**Structures et nouveaux types** Une structure se définit avec le mot-clé `struct`. Par exemple, une structure représentant une personne pourrait être :

```
struct personne {
    char* nom;
    char* prenom;
```



```
unsigned pointure;  
};
```

On déclarera une `struct` `personne` nommée `albert` de la manière suivante, en utilisant à nouveau le mot-clé `struct` :

```
struct personne albert = {"Dupont", "Albert", 43};
```

On accède aux champs de la structure avec un point :

```
printf("%s %s chaussure du %d\n", albert.prenom, albert.nom, albert.pointure);
```

Il est possible de définir un nouveau type, pour éviter d’avoir à utiliser le mot-clé `struct`. On ajoute souvent par convention le suffixe “\_t” à ces nouveaux types.

```
typedef struct personne personne_t;
```

On peut alors écrire l’initialisation :

```
personne_t albert;
```

Il est même possible déclarer la structure et le nouveau type en même temps :

```
typedef struct personne {  
    char* nom;  
    char* prenom;  
    unsigned pointure;  
} personne_t;
```

À vous d’évaluer la lisibilité d’une telle construction.

Enfin, pour initialiser une structure lors de sa déclaration, on peut écrire :

```
personne_t albert = {.nom = "Dupont", .prenom = "Albert", .pointure = 43};
```

Attention, cela ne fonctionne qu’à la déclaration. Le code suivant est invalide :

```
personne_t albert;  
albert = {.nom = "Dupont", .prenom = "Albert", .pointure = 43};
```

**Pointeur vers une structure** Dans le cas où l'on dispose d'un pointeur vers une structure, on accède aux membres de la structure pointé avec l'opérateur "flèche" : `pointeur->membre`, ce qui est une notation est équivalente à `(*pointeur).membre`. On accède donc à la structure **pointée par** `pointeur`, puis à son membre.

## B Format PGM

Le format PGM (pour *portable graymap file format*), permet de stocker des images en niveaux de gris. Vous l'avez déjà rencontré au premier semestre. Nous utiliserons le format PGM **binaire**, par opposition au format PGM ASCII, car plus compact sur disque. Un fichier au format PGM est composé :

- d'un en-tête :
  - ligne 0 : P5 \_\_\_\_\_ *Le nombre magique du format PGM binaire*
  - ligne 1 : NB\_COLONNES NB\_LIGNES \_\_\_\_\_ *Les dimensions de l'image*
  - ligne 2 : DEPTH \_\_\_\_\_ *La valeur pour chaque pixel va de 0 à DEPTH*
- des valeurs des pixels, qu'on considèrera appartenir à l'intervalle  $[0, 255]$ , et qu'on stockera donc dans des octets.

### Affichage du contenu de fichiers binaires

La commande `od` permet d'afficher le contenu d'un fichier binaire, et donc en particulier d'un fichier PGM binaire. Elle s'utilise comme suit, l'option `-x` groupant les octets par 2 et l'option `-c` affichant les caractères imprimables :

```
$ od -xc image.pgm
```

**Exemple** La Figure 26 montre un exemple d'image, un carré noir de 3 pixels de côté avec un pixel blanc au centre.

■

FIGURE 26 – Image de test.

```
$ od -xc image.pgm
0000000  3550  330a  3320  320a  3535  000a  0000  ff00
          P  5  \n  3      3  \n  2  5  5  \n  \0  \0  \0  \0 377
0000020  0000  0000
          \0  \0  \0  \0
0000024
```

On retrouve bien l'en-tête du fichier PGM binaire (P5), le nombre de lignes et de colonnes (3 3), la profondeur (255) puis les pixels.