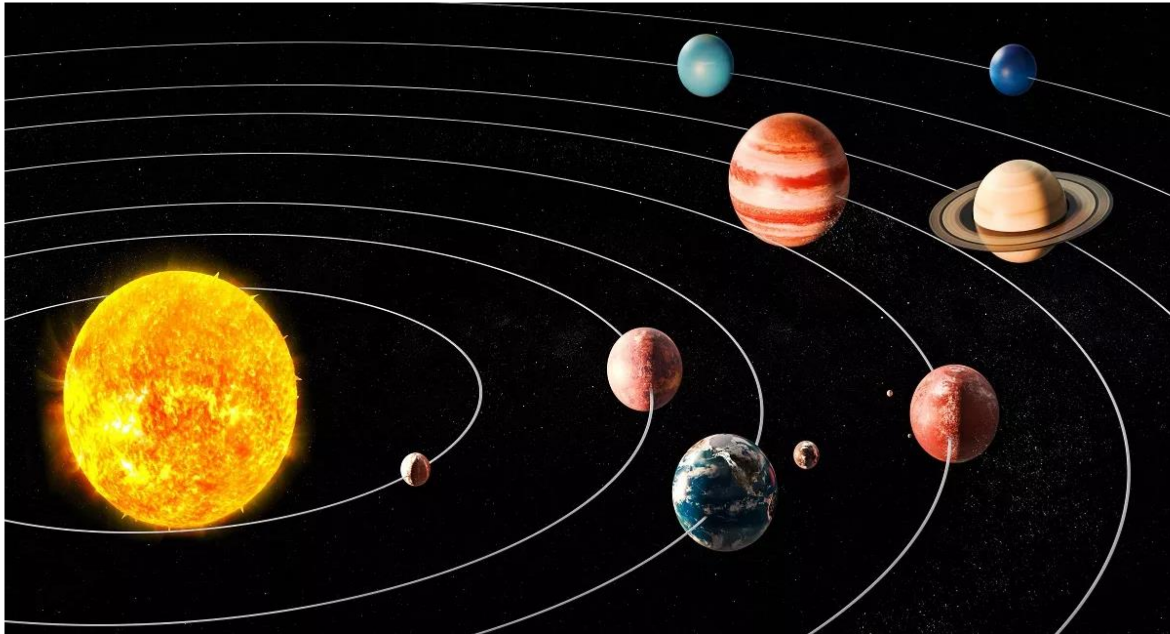
Les orbites



https://www.wondriumdaily.com/orbital-motion-the-orbit-of-a-planet-moves-a-little-after-every-loop/

INTRODUCTION 2

LA DESCRIPTION GÉNÉRALE DU PROGRAMME 3

1.1 La première exécution du programme 3
1.2 L'exécution des JUnit Test 4

LE CALCUL VECTORIEL EN JAVA 5

LES GRANDEURS PHYSIQUES 6

2.1 La quantité de mouvement 6
2.2 L'énergie cinétique 7
2.3 Le moment cinétique 8

L'ALGORITHME D'EULER 9

3.1 Euler de 1er ordre 9

L'INTERACTION GRAVITATIONNELLE 10

4.1 L'énergie gravitationnelle 10
4.2 La force gravitationnelle 11
4.3 La 2e loi de Newton (pour Euler) 12

LES ALTERNATIVES À L'ALGORITHME D'EULER 13

5.1 Euler de 2e ordre 13
5.2 La 2e loi de Newton (pour Euler de 2e ordre) 14
5.3 Runge-Kutta d'ordre 4 (1re position et vitesse finale) 15
5.4 Runge-Kutta d'ordre 4 (position et vitesse à Δt/2) 15
5.5 Runge-Kutta d'ordre 4 (2e position et vitesse finale) 16
5.6 Runge-Kutta d'ordre 4 (la véritable position et vitesse finale) 16
5.7 La 2e loi de Newton (pour Runge-Kutta d'ordre 4) 17

CONCLUSION 18

REMISE DU PROGRAMME 18

Introduction

Le laboratoire *Les orbites* consistera à implémenter des algorithmes de cinématique numérique dans le but de visualiser en trois dimensions le mouvement d'astre tout en évaluant l'évolution de certains paramètre physique d'un système de particule comme la quantité de mouvement, le moment cinétique, l'énergie, etc. Les algorithmes implémentés seront les suivants :

- Euler.
- Euler de second ordre.
- Runge-Kutta d'ordre 4 (RK4).

Pour ce faire, vous devrez utiliser à la classe **SVector3d** représentant un vecteur à trois dimensions où l'ensemble des opérations mathématiques requises auront déjà été implémentée dans la classe.

Pour visualiser les trajectoires des astres en trois dimensions, vous exploiterez la puissance de la librairie graphique ***OpenGL*** implémentée pour JAVA sous le nom de ***JOGL*** (***Java Binding for the OpenGL API***). Ainsi, si votre ordinateur est muni d'une carte graphique, l'exécution de ce programme sera plus rapide.

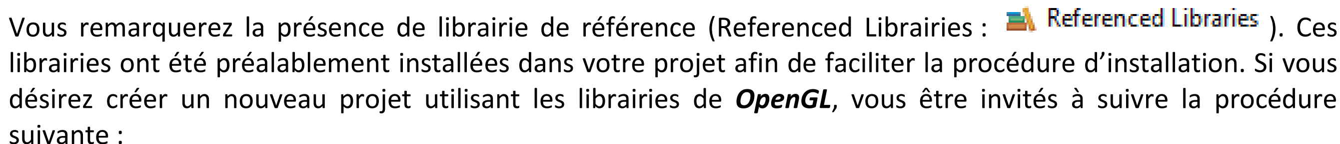
Pour réaliser ce laboratoire, vous avez accès au ***projet Java* SIM**. Vous pouvez télécharger le projet avec lien suivant :

http://physique.cmaisonneuve.qc.ca/svezina/projet/orbites/download/SIM-Orbites.zip

Commencez par vous définir un répertoire (exemple « *java* ») qui vous permettra de définir votre ***workspace*** lors de l'ouverture du logiciel ***Eclipse***. Par la suite, décompressez le fichier « *SIM-Orbite.zip* **»** dans le répertoire de votre ***workspace***. Vous devriez obtenir un répertoire au nom de « *SIM* ».

Ouvrez le logiciel ***Eclipse***. Dans l'onglet ***File***, choisissez l'option ***Switch Workspace*** et identifiez la localisation de votre répertoire de projet (dans l'exemple : « *java* »). Votre ***workspace*** est maintenant configuré.

Dans l'onglet ***File***, choisissez l'option ***Open Projects from File System…*** . Dans la fenêtre ***Import Projects from File System or Archive***, modifiez le champ ***Import source*** à l'aide du bouton ***Directory…*** et sélectionnez le répertoire du projet contenu dans le fichier décompressé « *SIM-Orbite.zip* » étant de nom *SIM*. Vous avez maintenant configuré votre environnement de développement.



Vous remarquerez la présence de librairie de référence (Referenced Librairies : Referenced Libraries). Ces librairies ont été préalablement installées dans votre projet afin de faciliter la procédure d'installation. Si vous désirez créer un nouveau projet utilisant les librairies de ***OpenGL***, vous être invités à suivre la procédure suivante :

https://physique.cmaisonneuve.qc.ca/svezina/info/JAVA/installation-jogl-win64.pdf

(procédure à réaliser pour installer JOGL à un nouveau projet JAVA)

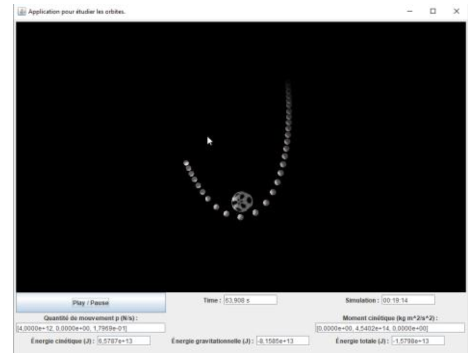
Durant le laboratoire, vous devrez répondre à des questions conceptuelles. Vous pouvez télécharger le cahier de réponse avec lien suivant :

http://physique.cmaisonneuve.qc.ca/svezina/projet/orbites/download/Cahier-Orbites.docx

La description générale du programme

L'application **SIMOrbit** vous permettra de visualiser une scène en trois dimensions contenant des astres. À l'aide d'une caméra contrôlable par la souris, vous pourrez vous déplacer dans cet environnement 3D afin d'observer le mouvement des astres.

Tout au long de la simulation, vous pourrez visualiser des grandeurs physiques d'un système de particule ($\vec{p}$, $\vec{L}$, K, U_g, E) afin de déterminer celles qui sont constantes dans le temps et celles qui divergent en fonction du choix de l'algorithme numérique retenu pour réaliser la simulation.



Aperçu de l'application SIMOrbite.

Dans l'application **SIMOrbit**, on y retrouve une classe abstraite **SParticleSystem** permettant de faire la gestion de particules regroupées dans un système afin de leur permettre de s'appliquer des forces entre elles dans le but de les déplacer selon des règles de cinématique. La classe **SGravitationalParticuleSystem** sera utilisée pour définir le type d'interaction entre les particules étant de nature gravitationnelle. Au fur et à mesure que vous compléterez ce laboratoire, vous pourrez observer le mouvement des particules ainsi que la valeur de différentes grandeurs physiques du système.

Architecture de l'héritage des classes :

SParticleSystem

(Classe abstraite d'un système de particule)

$\mapsto$ **SGravitationalParticleSystem**

(Système de particule gravitationnelle)

1.1 La première exécution du programme

Fichier à modifier : aucun

Classer à exécuter : **SIMOrbit.java**

Prérequis : aucun

Fichier de scène : *ballons.txt*

Dans le répertoire de votre projet, ouvrez le fichier *orbit_configuration.cfg* et constaté la présence de la ligne suivante : *read_data ballons.txt*. Ceci signifie que l'application **SIMOrbit** va ouvrir le fichier de scène *ballons.txt*, y extraire de l'information et affichée le tout dans l'application 3D. Vous devrez modifier ce choix de fichier de scène dans ce laboratoire lorsque vous devrez analyser une autre scène.

Dans la fenêtre ***Package Explorer*** du logiciel ***Eclipse***, ouvrez le répertoire **SIM** puis ouvrez le répertoire **src**. Constatez la présence de quelques ***packages*** nécessaires à l'exécution de ce programme. Ouvrez-le ***package*** **sim.application** et lancez l'exécution de la classe **SIMOrbit.java** à l'aide d'un « clic droit » sur le fichier. Sélectionnez dans le ***pop-pop menu*** l'option ***Run As*** et l'autre option ***Java Application***.

Vous êtes présentement plongés dans un environnement 3d où il y a la présence de 5 ballons que vous visualisez avec un angle d'ouverture de 60°. Vous pouvez lancer la simulation à l'aide du bouton « Action / Pause ». Vous allez constater qu'il n'y a rien qui bouge puisqu'il n'y a pas d'algorithme de cinématique d'implémenté.

Pour manœuvrer la caméra en 3D, vous devez effectuer les commandes suivantes à l'aide de la souris :

- Pour déplacer la caméra, vous devez enfoncer le « ***clic-droit*** » de la souris et déplacez celle-ci sur la fenêtre de l'application horizontalement et verticalement. Vous constaterez que la caméra peut se déplacer vers l'avant et l'arrière ainsi que sur la droite et la gauche.

- Pour faire tourner la caméra, vous devrez enfoncer le « ***clic-gauche*** » de la souris et effectuer un mouvement vertical pour faire tourner la caméra vers le haut ou vers le bas et effectuer un mouvement horizontal pour faire tourner la caméra vers la droite ou vers la gauche.
- Pour faire rouler la caméra, vous devrez enfoncer le « ***clic-gauche*** » et le « ***clic-droit*** » et effectuer un mouvement horizontal pour faire rouler la caméra.

1.2 L'exécution des JUnit Test

Fichier à modifier : aucun

Prérequis : aucun

Pour s'assurer de la qualité d'un programme, il est important de tester les fonctionnalités des différentes méthodes implémentées. L'environnement de développement ***Eclipse*** permet l'exécution de batterie de tests unitaires avec une gestion des succès (*succes*) et des échecs (*fail*).

https://www.javacodegeeks.com/2013/12/parameterized-junit-tests-with-junitparams.html

Un test unitaire réalise l'exécution d'une méthode (ou quelques méthodes) d'une classe ou de plusieurs classes dans un scénario particulier. Le résultat de l'exécution (« ***calculated*** ») est alors comparé avec un résultat attendu (« ***expected*** »). Si le résultat calculé est identique au résultat attendu, le test est un succès. Dans le cas contraire, le test est alors un échec.

Afin de vous familiariser avec l'exécution des ***JUnit Test***, vous allez réaliser l'exécution de l'ensemble des tests préalablement écrit pour vous afin de vous guider dans la qualité de vos implémentations.

Pour ce faire, vous allez :

- Dans la fenêtre ***Package Explorer***, ouvrez le répertoire **SIM** contenant l'ensemble des éléments du projet.
- Sur le répertoire **test/src**, effectuez un « clic droit » et sélectionnez dans le pop-pop menu l'option ***Run As*** et réalisez l'exécution de type ***JUnit Test***.
- (**Si les étapes précédentes ne fonctionnent pas**) Établir le lien avec la librairie ***JUnit 4***. Faire un clic droit sur le répertoire **SIM** dans la fenêtre ***Package Explorer*** et choisir l'option ***Properties***. Dans la propriété ***Java Build Path***, choisir l'onglet ***Libraries*** et activer le bouton ***Add Library***. Choisir ***JUnit*** et activer le bouton ***next***. Choisir ***JUnit 4*** et activer le bouton ***Finish***.

Dans la fenêtre ***JUnit***, vous pouvez visualiser l'ensemble des tests effectués pour valider certaines fonctionnalités du projet :

- Si toutes les implémentations sont adéquates, une **couleur verte** sera affichée (Failures : 0).
- S'il y a des implémentations inadéquates, une **couleur rouge** sera affichée (Failures : « nombre > 0 »).

Dans la fenêtre ***Console***, vous remarquerez la présente de messages indiquant que certains tests n'ont pas été effectués, mais qu'ils sont considérés comme des succès. Ce sont des tests reliés à des méthodes que vous devrez implémenter. Ces méthodes retournent présentement une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Lorsque l'implémentation sera réalisée, le test sera effectué « officiellement ».

Pour lancer l'exécution d'un nombre restreint de tests, vous n'avez qu'à effectuer le « clic droit » sur le répertoire ou le fichier désiré. Par exemple, vous pouvez lancer le test de la classe **SMathTest** du package ***sim.math*** situé dans le répertoire ***test/src/math***. Vous remarquerez qu'il est en succès (**couleur verte**), mais qu'il y a des tests non effectués.

Le calcul vectoriel en JAVA

La classe **SVector3d** disponible dans le ***package*** **sim.math** vous permettra, grâce à sa structure ***orientée-objet***, d'écrire sous forme mathématique des équations et les objets de type **SVector3d** employés dans vos équations exécuteront les opérations mathématiques comme vous l'avez appris en mathématique

Voici quelques exemples afin de vous illustrer comment exploiter cette classe dans votre code :

Définition en mathématique	En informatique
$\vec{A} = 2\vec{i} + 3\vec{j} + 4\vec{k}$ $\vec{B} = \vec{i} - 2\vec{j} - \vec{k}$ $a = 8.0$ $\vec{C}$: un vecteur étant la réponse à un calcul w : un scalaire étant la réponse à un calcul	SVector3d A = **new** SVector3d(2.0, 3.0, 4.0); SVector3d B = **new** SVector3d(1.0, -2.0, -1.0); **double** a = 8.0; SVector3d C; **double** w;

Opération mathématique	En mathématique	En informatique
L'addition de vecteur	$\vec{C} = \vec{A} + \vec{B}$ $\Rightarrow \quad \vec{C} = 3\vec{i} + \vec{j} + 3\vec{k}$	C = A.add(B);
La soustraction de vecteur	$\vec{C} = \vec{A} - \vec{B}$ $\Rightarrow \quad \vec{C} = \vec{i} + 5\vec{j} + 4\vec{k}$	C = A.substract(B);
La multiplication par un scalaire	$\vec{C} = a\vec{A}$ $\Rightarrow \vec{C} = 16\vec{i} + 24\vec{j} + 32\vec{k}$	C = A.multiply(a);
Le module d'un vecteur	$w = \left\|\vec{A}\right\|$ $\Rightarrow \quad w = \sqrt{29}$	w = A.modulus();
Le produit scalaire	$w = \vec{A} \cdot \vec{B}$ $\Rightarrow \quad w = -8$	w = A.dot(B);
Le produit vectoriel	$\vec{C} = \vec{A} \times \vec{B}$ $\Rightarrow \quad \vec{C} = 5\vec{i} + 6\vec{j} - 7\vec{k}$	C = A.cross(B);
La normalisation	$\vec{C} = \vec{A} / \left\|\vec{A}\right\| \quad \Rightarrow$ $\vec{C} = \frac{2}{\sqrt{29}}\vec{i} + \frac{3}{\sqrt{29}}\vec{j} + \frac{4}{\sqrt{29}}\vec{k}$	C = A.normalize();

En analysant les champs x, y et z la classe **SVector3d**, vous remarquerez qu'ils sont tous ***final***. Ainsi, un vecteur est un objet immutable (ne pouvant pas changer sous l'action d'une méthode). Tout calcul mathématique avec un objet **SVector3d** va générer un nouveau vecteur qui devra être affecté à une variable si vous désirez conserver le fruit du calcul (un peu comme vous devez faire pour sauvegarder un calcul réalisé avec la classe **String** de ***JAVA***).

Les grandeurs physiques

En premier temps, vous allez vous familiarisez avec les fonctionnalités de la classe **SVector3d** en programmant les trois grandeurs physiques suivantes :

quantité de mouvement $\vec{p}$, énergie cinétique K et moment cinétique $\vec{L}$

La classe statique **SMechanics** sera utilisée pour implémenter ces grandeurs physiques et elles seront utilisées dans la classe abstraite **SParticuleSystem** pour évaluer ces grandeurs sur l'ensemble des particules d'un système.

2.1 La quantité de mouvement

Fichier à modifier : **SMechanics.java**

Classer à exécuter : **SIMOrbit.java**

Prérequis : aucun

Fichier de scène : *ballons.txt*

Dans la classe **SMechanics** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
public SVector3d momentum(double mass, SVector3d speed)
```

L'objectif de cette méthode sera de réaliser le calcul de la quantité de mouvement

$$\vec{p} = m\vec{v}$$

où $\vec{p}$ correspond à la quantité de mouvement (*momentum*), m correspond à la masse (mass) et $\vec{v}$ correspond à la vitesse (speed).

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction `throw new SNoImplementationException( );` et complétez l'implémentation en utilisant la masse et la vitesse pour déterminer la quantité de mouvement.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SMechanicsTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Si vous êtes curieux, vous pouvez localiser l'utilisation de cette méthode dans la classe **SParticuleSystem** du ***package*** **sim.physics** dans la méthode

```
public SVector3d momentum( )
```

où l'on retrouve la somme de toutes les quantités de mouvement ($\sum_{i=0}^{N-1} \vec{p}_i$) de toutes les particules du système.

Exécutez l'application **SIMOrbit** avec la scène *ballons.txt*. Constatez l'affichage de la quantité de mouvement.

Cependant, puisqu'il n'y a pas d'algorithme de cinématique numérique d'implémenté, c'est normal que rien ne bouge !

Répondez à la question suivante dans le cahier de réponse :

> **Question 2.1 :**
>
> Selon les lois de la physique, dans un système de particules s'appliquant des forces gravitationnelles, est-ce que la quantité de mouvement totale doit être conservée ? Justifiez votre réponse à l'aide d'un argument physique.

2.2 L'énergie cinétique

Fichier à modifier : **SMechanics.java**

Classer à exécuter : **SIMOrbit.java**

Prérequis : aucun

Fichier de scène : *ballons.txt*

Dans la classe **SMechanics** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
public SVector3d kineticEnergy(double mass, SVector3d speed)
```

L'objectif de cette méthode sera de réaliser le calcul de l'énergie cinétique

$$K = \frac{1}{2}mv^2$$

où K correspond à l'énergie cinétique (*kineticEnergy*), m correspond à la masse (mass) et $\vec{v}$ correspond à la vitesse (speed).

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction `throw new SNoImplementationException( );` et complétez l'implémentation en utilisant la masse et la vitesse pour déterminer l'énergie cinétique.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SMechanicsTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Si vous êtes curieux, vous pouvez localiser l'utilisation de cette méthode dans la classe **SParticuleSystem** du ***package*** **sim.physics** dans la méthode

```
public double kineticEnergy( )
```

où l'on retrouve la somme de toutes les énergies cinétique ($\sum_{i=0}^{N-1} K_i$) de toutes les particules du système.

Exécutez l'application **SIMOrbit** avec la scène *ballons.txt*. Constatez l'affichage de l'énergie cinétique.

Répondez à la question suivante dans le cahier de réponse :

> **Question 2.2 :**
>
> Selon les lois de la physique, dans un système de particules s'appliquant des forces gravitationnelles, est-ce que l'énergie cinétique totale doit être conservée ? Justifiez votre réponse à l'aide d'un argument physique.

2.3 Le moment cinétique

Fichier à modifier : **SMechanics.java**

Classer à exécuter : **SIMOrbit.java**

Prérequis : aucun

Fichier de scène : *ballons.txt*

Dans la classe **SMechanics** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
public SVector3d angularMomentum(double mass, SVector3d position, SVector3d speed)
```

L'objectif de cette méthode sera de réaliser le calcul du moment cinétique d'une particule

$$\vec{L} = \vec{r} \times \vec{p}$$

où $\vec{L}$ correspond au moment cinétique d'une particule (*angular momentum*), $\vec{r}$ correspond à la position de la masse (position) et $\vec{p}$ correspond à la quantité de mouvement de la particule.

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction `throw new SNoImplementationException( );` et complétez l'implémentation en utilisant la masse (mass) et la vitesse (speed) pour déterminer votre quantité de mouvement (voir 2.1 La quantité de mouvement en page 6) et la position pour déterminer le moment cinétique.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SMechanicsTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Si vous êtes curieux, vous pouvez localiser l'utilisation de cette méthode dans la classe **SParticuleSystem** du ***package*** **sim.physics** dans la méthode

```
public SVector3d angularMomentum( )
```

où l'on retrouve la somme de tous les moments cinétiques ($\sum_{i=0}^{N-1} \vec{L}_i$) de toutes les particules du système.

Exécutez l'application **SIMOrbit** avec la scène *ballons.txt*. Constatez l'affichage du moment cinétique.

Répondez à la question suivante dans le cahier de réponse :

Question 2.3 :

Considérons la situation où un satellite est en orbite autour de la planète Terre. Parmi les quantités physiques suivantes applicables au satellite en orbite, précisez si elles sont conservées ou non en fonction du type d'orbite :

Pour une orbite circulaire :

Énergie cinétique		Quantité de mouvement		Moment cinétique	
OUI	NON	OUI	NON	OUI	NON

Pour une orbite elliptique :

Énergie cinétique		Quantité de mouvement		Moment cinétique	
OUI	NON	OUI	NON	OUI	NON

L'algorithme d'Euler

L'algorithme d'Euler est l'un des algorithmes d'intégration numérique applicable à la cinématique qui est le plus connu. Cependant, la simplicité de son implémentation donne de résultat souvent très limité en qualité.

Nous allons quand même débuter par implémenter celui-ci. Par la suite, nous allons ajouter des algorithmes plus sophistiqués afin de mieux comparer leurs résultats.

3.1 Euler de 1er ordre

Fichier à modifier : **SNumericalIntegrationMotion.java**

Classer à exécuter : **SIMOrbit.java**

Prérequis : aucun

Fichier de scène : *ballons.txt*

Dans la classe **SNumericalIntegrationMotion** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
public SKineticComponent euler(SKineticComponent c_i, SVector3d a_i, double dt)
```

L'objectif de cette méthode sera d'implémenter l'algorithme d'Euler de 1er ordre :

$$\vec{r}_f = \vec{r}_i + \vec{v}_i \Delta t$$

$$\vec{v}_f = \vec{v}_i + \vec{a}_i \Delta t$$

où $\vec{r}$ représente la position, $\vec{v}$ représente la vitesse, $\vec{a}$ représente l'accélération et Δt représente l'écoulement du temps dans une itération (un pas d'Euler).

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction `throw new SNoImplementationException( );` et complétez l'implémentation afin de déterminer la position finale $\vec{r}_f$ et la vitesse finale $\vec{v}_f$. Vous pourrez accéder à la position initiale $\vec{r}_i$ et la vitesse initiale $\vec{v}_i$ à partir de l'objet c_i de type **SKineticComponent**.

Afin de permettre à votre méthode de retourner la position finale et la vitesse finale ensemble, vous allez retourner un objet de type **SKineticComponent** à l'aide du constructeur

```
public SKineticComponent(SVector3d r, SVector3d v);
```

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNumericalIntegrationMotionTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Exécutez l'application **SIMOrbit** avec la scène *ballons.txt*. Constatez maintenant que vos ballons se déplacent, mais strictement en ligne droite.

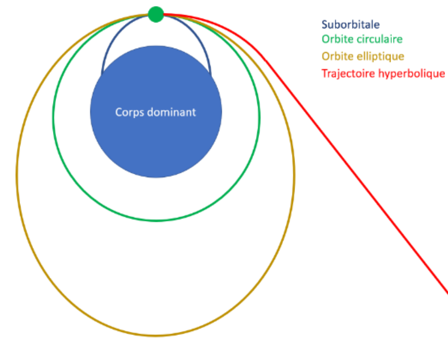
Répondez à la question suivante dans le cahier de réponse :

Question 3.1 :

Après avoir complété l'étape précédente (Euler de 1er ordre), vous constatez que vos ballons se déplacent strictement en ligne droite. Dans le contexte de l'implémentation de ce système de particules, formulez une justification permettant d'expliquer cette observation.

L'interaction gravitationnelle

L'interaction gravitationnelle est à l'origine du mouvement des astres dans l'espace. Son comportement radial où la force est inversement proportionnelle à la distance au carré qui sépare les astres assure des trajectoires de type elliptique ou hyperbolique.



https://venautics.space/complement/bases-de-la-mecanique-orbitale/generalites/
Type de trajectoire près d'un corps attracteur.

Dans la prochaine section, vous allez implémenter l'interaction gravitationnelle sous la forme de l'énergie gravitationnelle U_g (un scalaire) et de la force gravitationnelle F_g (vecteur).

4.1 L'énergie gravitationnelle

Fichier à modifier : **SGravitation.java**

Classer à exécuter : **SIMOrbit.java**

Prérequis : aucun

Fichier de scène : *ballons.txt*

Dans la classe **SGravitation** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
public double gravitationalEnergy(double M, SVector3d r_M, double m, SVector3d r_m)
```

L'objectif de cette méthode sera d'implémenter l'expression de l'énergie gravitationnelle

$$U_g = -G\frac{Mm}{r}$$

où U_g représente l'énergie gravitationnelle, M représente la masse qui attire, m représente la masse attirée et r représente la distance qui sépare les deux masses.

Remarquez que dans la définition de la méthode, vous n'avez pas accès à la distance r. Cependant, vous avez accès aux positions $\vec{r}_M$ (r_M) et $\vec{r}_m$ (r_m) des deux masses M et m.

À partir de la définition d'un vecteur déplacement

$$\vec{s} = \vec{r}_f - \vec{r}_i$$

et de son module

$$s = \left\|\vec{s}\right\| \quad ,$$

calculez la distance r qui sépare les deux masses pour évaluer votre énergie. Utilisez la constante

```
public static final double G
```

de cette classe pour obtenir la valeur de la constante G avec la précision exigée.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SGravitationTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Si vous êtes curieux, vous pouvez localiser l'utilisation de cette méthode dans la classe **SGravitationalParticleSystem** du ***package*** **sim.physics** dans la méthode

public double interactionEnergy(SParticle p1, SParticle p2) **throws** IllegalArgumentException

et que cette dernière est utilisée dans la classe **SParticleSystem** du ***package*** **sim.physics** dans la méthode

public double potentialEnergy( )

où l'on retrouve la somme de tous les énergies potentielles ($\sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} U_{ij}$) de toutes les particules du système.

Exécutez l'application **SIMOrbit** avec la scène *ballons.txt*. Constatez que vos ballons se déplacent toujours en ligne droite et c'est normal !

Répondez à la question suivante dans le cahier de réponse :

> **Question 4.1 :**
>
> Dans la méthode **public double** potentialEnergy( ) de la classe **SParticleSystem**, on retrouve deux boucles ***for*** imbriquée avec des indices i et j initialisés de façon particulière (i = 0 et j = i + 1). Expliquez en mots l'effet de ce choix sur la façon de calculer l'énergie potentielle totale pour un système de *N* particules.

4.2 La force gravitationnelle

Fichier à modifier : **SGravitation.java**

Prérequis : aucun

Dans la classe **SGravitation** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

public SVector3d gravitationalLaw(**double** M, SVector3d r_M, **double** m, SVector3d r_m)

L'objectif de cette méthode sera d'implémenter l'expression vectorielle de la force gravitationnelle :

Expression avec vecteur unitaire $\hat{r}$	Expression avec vecteur déplacement $\vec{r}$
$\vec{F}_g = -G\dfrac{Mm}{\|\vec{r}\|^2}\hat{r}$	$\vec{F}_g = -G\dfrac{Mm}{\|\vec{r}\|^3}\vec{r}$

avec $\vec{r}$: Vecteur déplacement de la source M de la force vers la cible m de la force ($\vec{r} = \vec{r}_m - \vec{r}_M$).

$\hat{r}$: Vecteur orientation de la force (M vers m) ($\hat{r} = \vec{r} / \|\vec{r}\|$).

Exploitez la position $\vec{r}_M$ (r_M) de la masse M et la position $\vec{r}_m$ (r_m) de la masse m pour implementer l'une ou l'autre des expressions de la force gravitationnelle. N'oubliez pas qu'une SVector3d possède une fonctionnalité de normalisation ($\vec{r} \rightarrow \hat{r}$).

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SGravitationTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Si vous êtes curieux, vous pouvez localiser l'utilisation de cette méthode dans la classe **SGravitationalParticleSystem** du ***package*** **sim.physics** dans la méthode

public SVector3d interactionForce(SParticle p1, SParticle p2) **throws** IllegalArgumentException

et que cette dernière est utilisée dans la classe **SParticleSystem** du ***package*** **sim.physics** dans la méthode

public SVector3d force(SParticle p)

provenant de l'interface **SApplyForce** où l'on retrouve la somme des forces sur la particule *j* ($\sum_{i=0,\, i \neq j}^{N-1} \vec{F}_{ij}$) provenant de toutes les autres particules.

4.3 La 2e loi de Newton (pour Euler)

Fichier à modifier : **SNumericalIntegrationMotion.java**

Classer à exécuter : **SIMOrbit.java**

Prérequis : 3.1 et 4.2

Fichier de scène : *ballons.txt*

Dans la classe **SNumericalIntegrationMotion** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

public SKineticComponent eulerForceIntegration(SParticle p, SApplyForce F, **double** dt)

L'objectif de cette méthode sera d'implémenter la 2e loi de Newton ($\sum \vec{F} = m\vec{a}$) afin d'y obtenir l'accélération et d'utiliser la méthode

public SKineticComponent euler(SKineticComponent c_i, SVector3d a_i, **double** dt)

pour déterminer les composants cinétiques finaux (position et vitesse) d'une particule selon la méthode d'Euler.

Utilisez l'objet F (l'applicateur de force) pour déterminer la force résultante applicable sur la particule p.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNumericalIntegrationMotionTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Exécutez l'application **SIMOrbit** avec la scène *ballons.txt*. Constatez que ça fonctionne maintenant très bien.

Question 4.3 :

Dans la simulation de la scène *ballons.txt*, vous remarquerez qu'il y a des ballons qui tournent autour du ballon de soccer mais que celui-ci bouge également. Est-ce normal ? Formulez une explication pour justifier votre position.

Indice : Vous pouvez lire le contenu du fichier *ballons.txt* disponible dans le répertoire « orbit » pour obtenir le descriptif de la scène.

Les alternatives à l'algorithme d'Euler

Afin d'améliorer les résultats proposés par l'algorithme d'Euler, vous allez implémenter les deux algorithmes numériques suivants :

Euler de 2e ordre	Runge-Kutta d'ordre 4 (RK4)
$\vec{r}(t) \to x_0 + v_{x0}t + \frac{1}{2}a_{x0}t^2$ (approximation de la position par un polynôme du 2e ordre) $\vec{v}(t) \to v_{x0} + a_{x0}t$ (approximation de la vitesse par un polynôme de 1er ordre)	$\vec{r}(t) \to x_0 + v_{x0}t + \frac{1}{2}a_{x0}t^2 + \frac{1}{6}\dot{a}_{x0}t^3 + \frac{1}{24}\ddot{a}_{x0}t^4$ (approximation de la position par un polynôme du 4e ordre) $\vec{v}(t) \to v_{x0} + a_{x0}t + \frac{1}{2}\dot{a}_{x0}t^2 + \frac{1}{6}\ddot{a}_{x0}t^3$ (approximation de la vitesse par un polynôme de 1er ordre)

Remarque[1] :

- $\dot{a}_{x0}$ correspond à la 1re dérivée de l'accélération par rapport au temps et porte le nom de *secousse* (*jerk*) .
- $\ddot{a}_{x0}$ correspond à la 2e dérivée de l'accélération par rapport au temps et porte le nom de *saut* (*jounce*, *snap*).

Pour analyser l'effet de l'évolution du choix de l'algorithme, nous allons utiliser un autre fichier de scène. Dans le répertoire de votre projet, ouvrez le fichier *orbit_configuration.cfg* et changer la ligne *read_data* de scène *ballons.txt* par la scène *soleil-terre-lune.txt*.

Exécutez l'application **SIMOrbit** avec la scène *soleil-terre-lune.txt*. Cette scène contient trois astres de dimension *non-réaliste* simulant le mouvement du Soleil, de la Terre et de la Lune. L'algorithme de Euler est présentement utilisé (paramètre *simulation_parameter* -> *numerical_algorithm EULER*).

Qu'est-ce qui ne fonctionne pas dans cette simulation ?

5.1 Euler de 2e ordre

Fichier à modifier : **SNumericalIntegrationMotion.java**

Prérequis : aucun

Dans la classe **SNumericalIntegrationMotion** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
public SKineticComponent eulerSecondOrder(SKineticComponent c_i, SVector3d a_i, double dt)
```

L'objectif de cette méthode sera d'implémenter l'algorithme d'Euler de 2e ordre :

$$\vec{r}_f = \vec{r}_i + \vec{v}_i \Delta t + \frac{1}{2}\vec{a}_i \Delta t^2$$

$$\vec{v}_f = \vec{v}_i + \vec{a}_i \Delta t$$

où $\vec{r}$ représente la position, $\vec{v}$ représente la vitesse, $\vec{a}$ représente l'accélération et Δt représente l'écoulement du temps dans une itération (un pas d'Euler).

Complétez l'implémentation afin de déterminer la position finale $\vec{r}_f$ et la vitesse finale $\vec{v}_f$. N'oubliez pas de retourner vos résultats dans le format d'un objet de type **SKineticComponent**.

[1]Référence : http://wearcam.org/absement/Derivatives_of_displacement.htm#:~:text=4th%20derivative%20is%20jounce,jerk%20with%20respect%20to%20time.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNumericalIntegrationMotionTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

5.2 La 2e loi de Newton (pour Euler de 2e ordre)

Fichier à modifier : **SNumericalIntegrationMotion.java**

Classer à exécuter : **SIMOrbit.java**

Prérequis : 4.2 et 5.1

Fichier de scène : *soleil-terre-lune.txt*

Dans la classe **SNumericalIntegrationMotion** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
public SKineticComponent eulerSecondOrderForceIntegration(SParticle p, SApplyForce F, double dt)
```

L'objectif de cette méthode sera d'implémenter la 2e loi de Newton ($\sum \vec{F} = m\vec{a}$) afin d'y obtenir l'accélération et d'utiliser la méthode

```
public SKineticComponent eulerSecondOrder(SKineticComponent c_i, SVector3d a_i, double dt)
```

pour déterminer les composants cinétiques finaux (position et vitesse) d'une particule selon la méthode d'Euler de 2e ordre.

Vous constaterez que le code est sensiblement le même que ce que vous avez déjà programmé.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNumericalIntegrationMotionTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Dans le répertoire « orbit »de votre projet, ouvrez le fichier *soleil-terre-lune.txt*. Dans la description de la scène, localisez le paramètre *simulation_parameter* et modifiez le champ *numerical_algorithm* (présentement *EULER*) par la *EULER_SECOND_ORDER*.

Exécutez l'application **SIMOrbit** avec la scène *soleil-terre-lune.txt*.

Répondez à la question suivante dans le cahier de réponse :

Question 5.2 :

Selon vous, est-ce que l'algorithme d'Euler de second ordre est présentement adéquat pour réaliser la « simulation désirée » de la scène *soleil-terre-lune.txt* avec les paramètres actuels ? Si non, proposez une très solution simple en exploitant cet algorithme pour régler la situation.

5.3 Runge-Kutta d'ordre 4 (1re position et vitesse finale)

Fichier à modifier : **SNumericalIntegrationMotion.java**

Prérequis : aucun

Dans la classe **SNumericalIntegrationMotion** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
protected SKineticComponent firstFinalStateRK4(SKineticComponent c_i, SVector3d a_i, double dt)
```

L'objectif de cette méthode sera d'implémenter les instructions #3 et #4 de l'algorithme du RK4[2] :

$$\vec{r}_{f(1)} = \vec{r}_i + \vec{v}_i \Delta t$$

$$\vec{v}_{f(1)} = \vec{v}_i + \vec{a}_i \Delta t$$

Complétez l'implémentation afin de déterminer la position finale $\vec{r}_{f(1)}$ et la vitesse finale $\vec{v}_{f(2)}$. Vous remarquerez que ce sont les mêmes instructions que la méthode d'Euler de 1er ordre (voir

3.1 Euler de 1er ordre en page 9).

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNumericalIntegrationMotionTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

5.4 Runge-Kutta d'ordre 4 (position et vitesse à Δt/2)

Fichier à modifier : **SNumericalIntegrationMotion.java**

Prérequis : aucun

Dans la classe **SNumericalIntegrationMotion** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
protected SKineticComponent middleTimeStateRK4(SKineticComponent c_i, SVector3d a_i, SVector3d a_f1, double dt)
```

L'objectif de cette méthode sera d'implémenter les instructions #7 et #8 de l'algorithme du RK4 :

$$\vec{r}_{f(mid)} = \vec{r}_i + \vec{v}_i \frac{\Delta t}{2} + \frac{1}{2}\vec{a}_i \left(\frac{\Delta t}{2}\right)^2$$

$$\vec{v}_{f(mid)} = \vec{v}_i + \left(\frac{3}{4}\vec{a}_i + \frac{1}{4}\vec{a}_{f(1)}\right)\frac{\Delta t}{2}$$

Complétez l'implémentation afin de déterminer la position finale $\vec{r}_{f(mid)}$ et la vitesse finale $\vec{v}_{f(mid)}$.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNumericalIntegrationMotionTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

[2] Lien vers les notes de cours : https://physique.cmaisonneuve.qc.ca/svezina/nya/note_nya/NYA_XXI_Chap%201.X1.pdf

5.5 Runge-Kutta d'ordre 4 (2e position et vitesse finale)

<u>Fichier à modifier :</u> **SNumericalIntegrationMotion.java**

<u>Prérequis :</u> aucun

Dans la classe **SNumericalIntegrationMotion** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
protected SKineticComponent secondFinalStateRK4(SKineticComponent c_i, SVector3d a_i, SVector3d a_f1, double dt)
```

L'objectif de cette méthode sera d'implémenter les instructions #11 et #12 de l'algorithme du RK4 :

$$\vec{r}_{f(2)} = \vec{r}_i + \vec{v}_i \Delta t + \frac{1}{2}\vec{a}_i \Delta t^2$$

$$\vec{v}_{f(2)} = \vec{v}_i + \left(\frac{1}{2}\vec{a}_i + \frac{1}{2}\vec{a}_{f(1)}\right)\Delta t$$

Complétez l'implémentation afin de déterminer la position finale $\vec{r}_{f(2)}$ et la vitesse finale $\vec{v}_{f(2)}$.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNumericalIntegrationMotionTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

5.6 Runge-Kutta d'ordre 4 (la véritable position et vitesse finale)

<u>Fichier à modifier :</u> **SNumericalIntegrationMotion.java**

<u>Prérequis :</u> aucun

Dans la classe **SNumericalIntegrationMotion** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
protected SKineticComponent finalStateRK4(SKineticComponent c_i,
SVector3d a_i, SVector3d a_mid, SVector3d a_f2, double dt)
```

L'objectif de cette méthode sera d'implémenter les instructions #16 et #17 de l'algorithme du RK4 :

$$\ldots.\ \vec{r}_f = \vec{r}_i + \vec{v}_i \Delta t + \frac{1}{2}\left(\frac{1}{3}\vec{a}_i + \frac{2}{3}\vec{a}_{mid}\right)\Delta t^2$$

$$\ldots\ \vec{v}_f = \vec{v}_i + \left(\frac{1}{6}\vec{a}_i + \frac{4}{6}\vec{a}_{mid} + \frac{1}{6}\vec{a}_{f(2)}\right)\Delta t$$

Complétez l'implémentation afin de déterminer la position finale $\vec{r}_f$ et la vitesse finale $\vec{v}_f$.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNumericalIntegrationMotionTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

5.7 La 2e loi de Newton (pour Runge-Kutta d'ordre 4)

Fichier à modifier : **SNumericalIntegrationMotion.java**

Classer à exécuter : **SIMOrbit.java**

Prérequis : 4.2, 5.3, 5.4, 5.5 et 5.7

Fichier de scène : *soleil-terre-lune.txt* et *systeme_solaire.txt*

Dans la classe **SNumericalIntegrationMotion** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

protected SKineticComponent RK4ForceIntegration(SParticle p, SForceCalculator F, **double** dt)

L'objectif de cette méthode sera d'implémenter l'ensemble des instructions de l'algorithme du RK4 (#1 à #17) et d'y retourner les valeurs de

$$\ldots\ \vec{r}_f = \vec{r}_i + \vec{v}_i \Delta t + \frac{1}{2}\left(\frac{1}{3}\vec{a}_i + \frac{2}{3}\vec{a}_{mid}\right)\Delta t^2 \qquad \text{et} \qquad \ldots\ \vec{v}_f = \vec{v}_i + \left(\frac{1}{6}\vec{a}_i + \frac{4}{6}\vec{a}_{mid} + \frac{1}{6}\vec{a}_{f(2)}\right)\Delta t\ .$$

Pour calculer $\vec{r}_f$ et $\vec{v}_f$, vous devrez calculer plusieurs accélérations différentes ($\vec{a}_i$, $\vec{a}_{f1}$, $\vec{a}_{mid}$ et $\vec{a}_{f2}$). Pour ce faire, utilisez la force F pour évaluer vos accélérations avec la 2e loi de Newton.

IMPORTANT :

- Puisque la force calculée par F sera gravitationnelle, elle dépend de la position de la particule. Ainsi, avant de calculer une accélération, assurez-vous de bien définir les composants cinétiques (**SKineticComponent**) de votre particule avant d'utiliser la force F avec cette particule. Utilisez l'instruction

  p.setKineticComponent(…);

  pour modifier la position et la vitesse de la particule avant d'évaluer son accélération à cette position et vitesse là. Vous obtiendrez bien entendu ces positions et vitesses avec les méthodes que vous avez programmées précédemment.
- Quand vous aurez déterminé $\vec{r}_f$ et $\vec{v}_f$, avant de retourner ce résultat en objet **SKineticComponent**, assurez-vous de redéfinir la position et la vitesse de la particule avec ses valeurs initiale (exemple : p.setKineticComponent(c_i);). Autrement, le 1re JUnit Test sera en échec (mais vous aurez quand même une version fonctionnelle de l'implémentation du RK4).

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNumericalIntegrationMotionTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Dans le répertoire « orbit »de votre projet, ouvrez le fichier *soleil-terre-lune.txt*. Dans la description de la scène, localisez le paramètre *simulation_parameter* et modifiez le champ *numerical_algorithm* par la *RK4*.

Exécutez l'application **SIMOrbit** avec la scène *soleil-terre-lune.txt*.

Répondez à la question suivante dans le cahier de réponse :

Question 5.7a :

Avec l'implémentation du RK4, est-ce que le mouvement de Lune est maintenant adéquat ? Justifiez votre réponse à l'aide d'un court texte.

Dans le répertoire de votre projet, ouvrez le fichier *orbit_configuration.cfg* et changer la ligne *read_data* pour la scène *systeme_solaire.txt*. Cette scène représente à l'échelle notre système solaire où le rayon du Soleil a été multiplié par 10 et le rayon des planètes et la Lune a été multiplié par 100 afin que les astres ne soient pas trop petits en raison des distances astronomiques.

Exécutez l'application **SIMOrbit** avec la scène *système_solaire.txt*.

Répondez à la question suivante dans le cahier de réponse :

Question 5.7b :

Formulez une affirmation permettant de justifier que la simulation du système solaire de la scène *systeme_solaire.txt* est réaliste (si l'on fait abstraction des dimensions des astres).

Conclusion

Félicitations ! Vous avez complété l'implémentation de l'interaction gravitationnelle qui fut intégré à un système de particules pour déplacer celle-ci grâce à différentes intégrales numériques de cinématique et ce, en trois dimensions.

Remise du programme

Pour effectuer la remise électronique de votre programme, envoyer le fichier ci-dessous dans l'espace de dépôt exigé par votre enseignant (exemple : OMNIVOX/LÉA) :

- Le **répertoire SIM** de votre projet dans un **format compressé** « zip » sous le nom *SIM-Orbites.zip* comme vous l'avez initialement téléchargé.