Compte rendu UE Informatique

COUËRON Lola
JOLY Marine

L3 Physique, Université Grenoble Alpes

Novembre 2024

Sommaire

1 Introduction

1.1 Contexte

La simulation informatique est un outil prépondérant dans l'étude de phénomènes physiques. Elle permet de comparer des modèles théoriques et expérimentaux avec précision. Dans le cadre de l'UE Informatique nous présentons une simulation à N corps ainsi que deux exemples d'applications : un problème à deux corps et un système solaire. L'objectif initial est de simuler l'interaction Terre - Soleil. Pour se faire nous utilisons le Principe Fondamental de la Dynamique de Newton qui nous permet d'obtenir les équations du mouvement. Cependant afin de décrire des systèmes plus complexes nous utilisons l'algorithme de Runge-Kutta d'ordre 4 qui permet de simuler l'interaction à N corps. Ces deux interactions sont visuellement décrite via l'outil PyGame.

1.2 Structure générale du code

Ce projet étant collaboratif, nous avons travaillé sur
GitHub (https ://github.com/MJ240103/solarSystem) afin d'optimiser la répartition des tâches. Ce projet est divisé en 4 fichiers :

1. le fichier contenant l'application réalisée avec Tkinter (**UIsolarSystem.py**)
2. le fichier contenant la simulation réalisée avec PyGame (**mainEngine.py**)
3. le fichier contenant les équations et les paramètres physiques nécessaires à la simulation (**bodies.py**)
4. les fichier permettant de stocker les paramètres de simulations déjà existantes (fichiers json d'extension **.mj** et **Simulation1.py**)

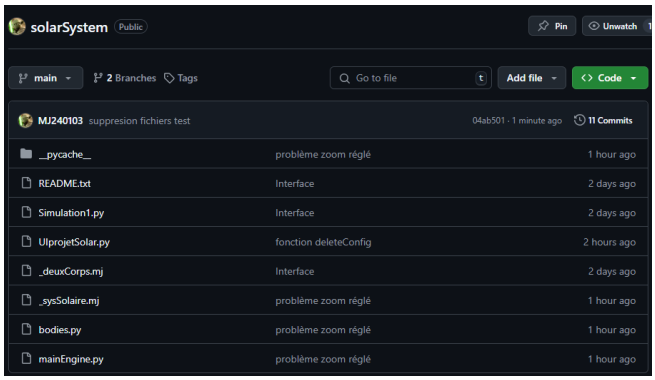

FIGURE 1 – Fichiers de la branche main du repository solarSystem.

Le fichier **mainEngine.py** utilise les équations et paramètres physiques du fichier **bodies.py** afin de simuler les interactions attractives entre plusieurs corps et plusieurs paramètres par le biais de PyGame. Le nombre de corps et Les paramètres sont définis par l'utilisateur grâce à l'interface contenue dans le fichier **IUsolarSystem**. Celle-ci utilise Tkinter pour lancer la simulation PyGame. Les simulations peuvent être stockées dans des fichiers python (ex : **Simulation1.py**) ou dans des fichiers json (ex : **sysSolaire.mj)**. Les librairies os, tkinter, json, pygame, sys, collections, math et copy ont été utilisées.

2 Physique

2.1 Implémentation naïve

En premier lieu, nous utilisons une implémentation naïve des équations du mouvement. Nous considérons un corps central autour duquel gravite un autre corps. Le mouvement de ce corps est régi par la force de gravitation $\vec{F} = -G\frac{m_A m_B}{r^2}\vec{U_r}$. Les coordonnées cartésiennes sont utilisées afin de simplifier l'implémentation des équations du mouvement en Python.

La position d'une particule en coordonnées cartésiennes est donnée par :

$$\vec{r}(t) = x(t)\vec{i} + y(t)\vec{j} + z(t)\vec{k} \tag{1}$$

La vitesse est la dérivée de la position par rapport au temps :

$$\vec{v}(t) = \frac{d\vec{r}(t)}{dt} = \frac{dx(t)}{dt}\vec{i} + \frac{dy(t)}{dt}\vec{j} + \frac{dz(t)}{dt}\vec{k} \tag{2}$$

L'accélération est la dérivée de la vitesse par rapport au temps :

$$\vec{a}(t) = \frac{d\vec{v}(t)}{dt} = \frac{d^2x(t)}{dt^2}\vec{i} + \frac{d^2y(t)}{dt^2}\vec{j} + \frac{d^2z(t)}{dt^2}\vec{k} \tag{3}$$

Pour obtenir les équations du mouvement, on applique le Principe Fondamental de la Dynamique (PFD) de Newton et on projette les forces en cartésien : $\vec{F} = m\vec{a}$

Enfin, nous implantons ces équations en Python. L'appel à la fonction `applyGravity` effectue les calculs de position, de vitesse et d'accélération du corps en orbite. Quant à la fonction `setPosition`, celle-ci est chargée de mettre à jour la position du corps en orbite en fonction de sa vitesse et de l'intervalle de temps indiquant la durée d'une révolution autour de l'astre central.

```
168     # Mise à jour des planètes
169     for planet in solarSystem:
170         planet.applyGravity(masseSoleil, sun_position, G)
171         planet.setPosition(dt)
```

FIGURE 2 – Execution de `applyGravity` et `setPosition` afin de générer les coordonnées des corps

```
14      def setPosition(self, dt):
15          """Met à jour la position de la planète en fonction de sa vitesse et de l'intervalle de temps."""
16          self.vitesse[0] += self.acceleration[0] * dt
17          self.vitesse[1] += self.acceleration[1] * dt
18          self.position[0] += self.vitesse[0] * dt
19          self.position[1] += self.vitesse[1] * dt
20
21      def applyGravity(self, masse_soleil, position_soleil, G):
22          """Met à jour l'accélération de la planète en fonction de la gravité exercée par le Soleil."""
23          dx = position_soleil[0] - self.position[0]
24          dy = position_soleil[1] - self.position[1]
25          distance = math.sqrt(dx**2 + dy**2)
26          force = G * masse_soleil * self.masse / distance**2
27          theta = math.atan2(dy, dx)
28          self.acceleration[0] = math.cos(theta) * force / self.masse
29          self.acceleration[1] = math.sin(theta) * force / self.masse
```

FIGURE 3 – Fonctions `setPosition` et `applyGravity` du module bodies.py

Ces fonctions ne sont pas présente dans le code final car elles sont désormais remplacées par de nouvelles fonctions utiles pour une implémentation Runge-Kutta d'ordre 4.
`applyGravity` est désormais rémplacée par `gravite` et `setPosition` par `selfDraw` .

2.2 Implémentation Runge-Kutta ordre 4

L'implémentation naïve étant un outil limité nous utilisons l'algorithme de Runge-Kutta d'ordre 4 (RK4) afin de décrire des systèmes plus complexes comme l'interaction entre plusieurs corps. La méthode d'Euler ou bien celle de Runge-Kutta d'ordre 4 semble ainsi plus adaptées.

La méthode d'Euler permet à partir d'une condition initiale de résoudre une équation différentielle du premier ordre. Cette méthode est simple mais peu précise notamment lorsque l'on choisit des pas trop grand car son erreur est de l'ordre de h (avec h le pas). C'est pourquoi si l'on divise le pas par 2 l'erreur est divisée par 2.

Par conséquent, du fait de sa précision la méthode RK4 a retenu toute notre attention. En effet, cette méthode utilise des itérations. La première solution éstimée est utilisée pour éstimer une deuxième solution plus précise, et ainsi de suite. La méthode RK4 est une méthode d'ordre 4, ce qui signifie que l'erreur totale accumulée est de l'ordre de h^4. Donc si on divise le pas par 2, l'erreur est divisée par 16. Ainsi, RK4 est plus précise que la méthode d'Euler.

On considère le problème suivant : $y' = f(t, y)$ avec $y(t_0) = y_0$.

La méthode RK4 est donnée par l'équation : $y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ où

— $k_1 = f(t_n, y_n)$

— $k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1)$

— $k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2)$

— $k_4 = f(t_n + h, y_n + hk_3)$

Le fonctionnement se base sur l'idée que la valeur suivante (y_{n+1}) est approchée par la somme de la valeur courante (y_n) et du produit de la taille de l'intervalle (h) par la pente estimée. Cette pente est obtenue en effectuant une moyenne pondérée des pentes :

— k_1 correspond à la pente au début de l'intervalle.

```python
def runge_kutta(planetes, G, dt):
    k1v = []
    k1x = []
    k2v = []
    k2x = []
    k3v = []
    k3x = []
    k4v = []
    k4x = []
    acc = {}

    #Etape 1
    for planete in planetes: acc[planete] = planete.gravite(planetes, G)
    clone_planetes = [copy.copy(p) for p in planetes]
    for planete in planetes:
        k1v.append(acc[planete])
        k1x.append(planete.vitesse)
```

FIGURE 4 – Etape 1 de la méthode RK4.

— k_2 correspond à la pente au milieu de l'intervalle. Elle s'obtient en utilisant k_1 pour calculer la valeur de y au point $t_n + \frac{h}{2}$ par le biais de la méthode d'Euler.

```
#Etape 2
for i, planete in enumerate(planetes):
    clone_planetes[i].vitesse[0] += k1v[i][0] * dt / 2
    clone_planetes[i].vitesse[1] += k1v[i][1] * dt / 2
    clone_planetes[i].position[0] += k1x[i][0] * dt / 2
    clone_planetes[i].position[1] += k1x[i][1] * dt / 2

    acc[planete] = planete.gravite(planetes, G)
    k2v.append(acc[planete])
    k2x.append(clone_planetes[i].vitesse)
```

FIGURE 5 – Etape 2 de la méthode RK4.

— k_3 est la pente au milieu de l'intervalle, mais obtenue en utilisant k_2 pour y.

```
# Etape 3
for i, planete in enumerate(planetes):
    clone_planetes[i].vitesse[0] += k2v[i][0] * dt / 2
    clone_planetes[i].vitesse[1] += k2v[i][1] * dt / 2
    clone_planetes[i].position[0] += k2x[i][0] * dt / 2
    clone_planetes[i].position[1] += k2x[i][1] * dt / 2

    acc[planete] = planete.gravite(planetes, G)
    k3v.append(acc[planete])
    k3x.append(clone_planetes[i].vitesse)
```

FIGURE 6 – Etape 3 de la méthode RK4.

— k_4 correspond à la pente de la fin de l'intervalle, obtenue avec la valeur de y calculée en utlisant k_3.

```
# Etape 4
for i, planete in enumerate(planetes):
    clone_planetes[i].vitesse[0] += k3v[i][0] * dt / 2
    clone_planetes[i].vitesse[1] += k3v[i][1] * dt / 2
    clone_planetes[i].position[0] += k3x[i][0] * dt / 2
    clone_planetes[i].position[1] += k3x[i][1] * dt / 2

    acc[planete] = planete.gravite(planetes, G)
    k4v.append(acc[planete])
    k4x.append(clone_planetes[i].vitesse)
```

FIGURE 7 – Etape 4 de la méthode RK4.

Lorsque la moyenne pondérée des 4 pentes est effectuée, un poids plus important est donné aux pentes du milieu.

```
for i, planete in enumerate(planetes):
    planete.vitesse[0] += ((k1v[i][0] + 2 * k2v[i][0] + 2 * k3v[i][0] + k4v[i][0]) * dt / 6)
    planete.vitesse[1] += ((k1v[i][1] + 2 * k2v[i][1] + 2 * k3v[i][1] + k4v[i][1]) * dt / 6)
    planete.position[0] += ((k1x[i][0] + 2 * k2x[i][0] + 2 * k3x[i][0] + k4x[i][0]) * dt / 6)
    planete.position[1] += ((k1x[i][1] + 2 * k2x[i][1] + 2 * k3x[i][1] + k4x[i][1]) * dt / 6)
```

FIGURE 8 – Itération de la méthode RK4.

3 Interface graphique de l'application de simulation

3.1 Paramétrage des simulations

3.1.1 Body

Nous appelons `Body` un objet regroupant les caractéristiques suivante :
— Nom (string) : Nom de l'objet
— Masse (float) : Masse de l'objet
— Rayon (float) : Rayon de l'objet
— Position (float tuple) : Position (cartésienne) de l'objet
— Vitesse (float tuple) : Vitesse (cartésienne) de l'objet
— Acceleration (float tuple) : Accélération (cartésienne) de l'objet
— couleur (int 3-uplet) : Couleur

Un `Body` peut par exemple représenter la Terre, le Soleil ou bien un satellite artificiel.

3.1.2 Simulation

Nous appelons `Simulation` un objet regroupant les caractéristiques suivantes :
— Nom_Simu (string) : Nom de la simulation
— FPS (int) : Frame Per Second (nombre d'image par seconde) de la simulation
— G (float) : Constante gravitationnelle
— dt (float) : Intervalle de temps utilisé par RK4
— SPACE_X (float) : Taille horizontale de l'espace simulé
— SPACE_Y (float) : Taille vecticale de l'espace simulé
— ECHELLE_RAYON (float) : Echelle de rendu des rayons des `body`
— UNIVERSE_CENTER (float tuple) : Centre de l'univers (au centre de l'affichage en début de simulation)
— planetes (`Bodylst`) : Liste des objets, tels que définit avec le constructeur `body`

Pour enregistrer une simulation, nous utilisons le format `JavaScript Object Notation` (JSON).

Python a l'avantage de directement prendre en charge ce format à l'aide du module Json. Ainsi, en important le fichier simulation et en signifiant à python qu'il s'agit de json, on peut accéder aux différents paramètres de la simulation.

On accède par exemple au nom d'une simulation représentée par la variable sim de la manière suivante : nomSimu = sim["Nom_Simu"] (voir Figure 9).

Nous transformons ensuite notre objet directement convertis depuis le Json en un objet Simulation à l'aide d'une classe. L'avantage de cette méthode est que la classe Simulation possède une méthode `launch()` qui lance directement l'exécution de la simulation dans le moteur. (voir Figure 10)

```
def listToSim(self):
    fileName = self.Lb.get(tk.ACTIVE)
    dirPath = self.dirVar.get()
    path = dirPath+"/"+fileName
    with open(path,"r",encoding="utf-8") as f:
        data = f.read().replace("\n","").replace("\t","")
        obj = json.loads(data)
        sim = Simulation(obj)
    return sim
```

FIGURE 9 – Importation d'un fichier via le module Json

```
class Simulation(): #Objet simulation, que l'on passe à notre mainEngine
    def __init__(self, json="{}"):
        self.nom = json["Nom_Simu"]
        self.FPS = json["FPS"]
        self.G = json["G"]
        self.dt = json["dt"]
        self.SPACE_X = json["SPACE_X"]
        self.SPACE_Y = json["SPACE_Y"]
        self.ECHELLE_RAYON = json["ECHELLE_RAYON"]
        self.UNIVERSE_CENTER = json["UNIVERSE_CENTER"]
        self.solarSystem = []

        for planet in json["planetes"]:
            self.solarSystem.append(Planet(
                nom=planet["nom"],
                masse=planet["masse"],
                rayon=planet["rayon"],
                position=planet["position"],
                vitesse=planet["vitesse"],
                acceleration=planet["acceleration"],
                couleur=planet["couleur"]
            ))

    def launch(self): #Lancer la simulation
        mainEngine.simulation(self)
```

FIGURE 10 – Classe simulation

3.2 Modification, Création

L'interface, réalisée à l'aide de Tkinter se présente comme suit :

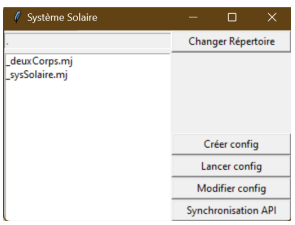

FIGURE 11 – Interface Fenêtre - principale

Sur la fenêtre principale de l'interface, on a la possibilité de se placer dans un répertoire en cliquant sur "Changer Répertoire". Le répertoire par défaut est celui du fichier interface.

La `ListBox` à gauche liste les fichiers de simulations compatibles (.mj) dans le répertoire courant. Ces fichiers contiennent un code JSON correspondants à l'objet `Simulation` définit plus haut.

Lorsqu'on clique sur le nom d'un fichier, il devient le fichier actif. On peut ensuite cliquer sur :
— "Lancer config" : Exécute la configuration dans l'engine
— "Modifier config" : Ouvre la fenêtre secondaire avec les paramètres de la simulation active.

Indépendamment du fichier actif, on peut cliquer sur :
— "Créér config" : Ouvre la fenêtre secondaire avec les paramètres par défaut.
— "Synchronisation API" : Récupère les positions et vitesse actuelles des astres du système solaire , créé un fichier de simulation, et l'execute.

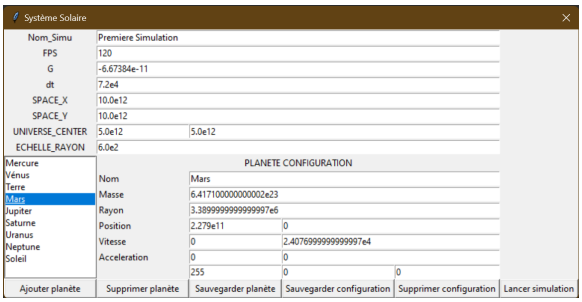

FIGURE 12 – Interface Fenêtre - secondaire

Une fenêtre secondaire s'ouvre lors de la création ou de la modification d'une simulation, avec le même comportement dans les deux cas. Les constantes peuvent être modifiées en ajustant directement les champs correspondants.
Pour éditer une planète, il suffit de cliquer sur son nom, de modifier les champs, puis de valider en cliquant sur "Sauvegarder planète". Il est aussi possible d'ajouter ou de supprimer une planète.

Une fois les modifications terminées, on clique sur "Sauvegarder configuration", qui permet de sauvegarder la simulation via une fenêtre de dialogue. Il est possible d'écraser l'ancienne version ou de lancer la simulation. Avant la sauvegarde, on peut tester la simulation en cliquant sur "Lancer Simulation" .

4 Interface graphique de la simulation par le biais d'exemples d'utilisation

Comme explicité en 2.2, on utilise RK4 dont la mise à jour en temps réelle permet la détermination des trajectoires.

4.1 Simulation à deux corps

4.1.1 Mise à jour de Runge-Kutta

La simulation des mouvements des corps est effectuée à chaque itération de la boucle principale. La méthode de Runge-Kutta est utilisée pour mettre à jour les positions des planètes en fonction des forces gravitationnelles. Cela se produit à la ligne 120 : `runge_kutta(config.solarSystem, config.G, config.dt)`

4.1.2 Affichage

La fonction `realToDisplay` transforme les coordonnées réelles (en fonction de l'espace simulé) en coordonnées adaptées à l'affichage sur la fenêtre Pygame. Ensuite, la méthode `selfDraw` de l'instance de

la classe `Planet` est utilisée pour afficher la simulation à l'écran.

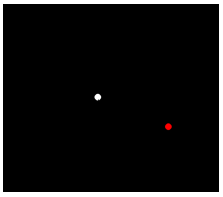

FIGURE 13 – Interaction deux corps sans trace

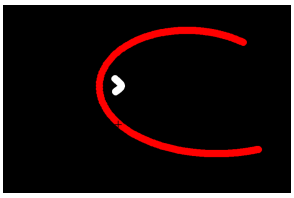

FIGURE 14 – Interaction deux corps avec trace

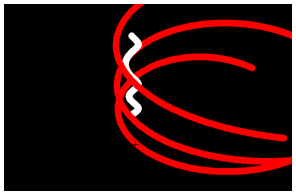

FIGURE 15 – Evolution de l'interaction deux corps

On peut aussi déplacer la caméra par rapport au centre de l'écran en utilisant les touches (z, q, s et d).

Au lancement de la simulation, les trajectoires des corps ne sont pas affichées. Toutefois, en appuyant sur la touche espace du clavier, il est possible de les afficher ou de les supprimer. En effet, appuyer sur espace permet de choisir si l'on veut remplir le fond en noir ou non à chaque mise à jour de la position des corps.

```python
show_orbits = False # Variable pour contrôler l'affichage des trajectoires

...

window.fill(BLACK)
while running:
    # Gestion des événements
    for event in pygame.event.get():
        if event.type == pygame.QUIT:  # Si la fenêtre d'animation est fermée
            running = False  # Termine la boucle principale
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_g:
                zoom -= 0.1  # Augmenter le zoom pour rapprocher
                print(zoom)
            elif event.key == pygame.K_h:
                zoom += 0.1  # Réduire le zoom pour éloigner
                print(zoom)
            elif event.key == pygame.K_SPACE:
                show_orbits = not show_orbits  # Bascule de l'affichage des trajectoires

...

# Mettre le fond en noir uniquement si les orbites ne sont pas affichées
    if not show_orbits:
        window.fill(BLACK)
```

FIGURE 16 – Code source du tracé des trajectoires

4.1.3 Option ZOOM

On utilise les touches (g,h) pour zoomer/dézoomer. Cette fonctionnalité est rendue possible par les lignes suivantes :

```python
if event.key == pygame.K_g:
    zoom -= 0.1  # Augmenter le zoom pour rapprocher
    print(zoom)
elif event.key == pygame.K_h:
    zoom += 0.1  # Réduire le zoom pour éloigner
```

FIGURE 17 – Code source de la fonctionnalité

Le zoom influence la manière dont les positions des planètes sont converties pour l'affichage à l'écran. Cette transformation est effectuée dans la méthode `planet.selfDraw()`, qui prend en compte la valeur du zoom pour ajuster la taille et la position des planètes dans la fenêtre Pygame.

```python
# Affichage des planètes
for planet in config.solarSystem:
    planet.selfDraw(
        window,
        config.ECHELLE_RAYON,
        curCenter,
        SCREEN_WIDTH,
        SCREEN_HEIGHT,
        zoom,
        lambda x, y: realToDisplay(x,y,SCREEN_WIDTH,SCREEN_HEIGHT,config.SPACE_X,config.SPACE_Y))
```

FIGURE 18 – Prise en compte de la fonctionnalité zoom

La fonction lambda `realToDisplay()` calcule les nouvelles positions des objets en fonction de l'échelle du zoom, permettant de zoomer à l'intérieur ou à l'extérieur du système solaire simulé tout en ajustant l'affichage de manière cohérente.

4.1.4 Choix d'une planète centrale

On peut choisir laquelle des deux planètes est placée au centre du système.

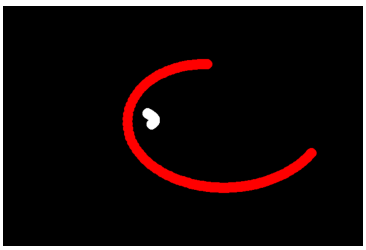

FIGURE 19 – Centrage 1

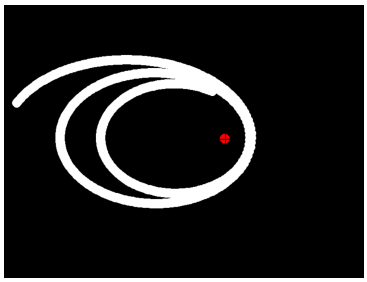

FIGURE 20 – Centrage 2

4.2 Simulation du système solaire

Tout ce qui à été décrit précédemment est valable dans la simulation du système solaire. A partir de l'interface principale de la fenêtre Tkinter on peut lancer la simulation `sysSolaire`.On utilise le fichier `sysSolaire.mj` de type JSON qui contient les caractéristiques d'une simulation du système solaire. On peut également la modifier pour y ajouter d'autres corps tels que les lunes de Jupiter.


```
{
    "Nom_Simu":"Premiere Simulation",
    "FPS":120,
    "G":-6.67384e-11,
    "dt":7.2e4,
    "SPACE_X":1e13,
    "SPACE_Y":1e13,
    "UNIVERSE_CENTER":[5e12,5e12],
    "ECHELLE_RAYON":600,
    "planetes":[
        {"nom":"Mercure",
        "masse":3.3011e23,
        "rayon":2.4397e6,
        "position":[57.9e9,0],
        "vitesse":[0,47.87e3],
        "acceleration":[0,0],
        "couleur":[255,255,255]
        },
        {"nom":"Vénus",
```


FIGURE 21 – Fichier JSON

La simulation du système solaire permet de mettre en évidence le caractère attracteur du Soleil du fait de sa masse prépondérante par rapport aux autres corps ainsi que les trajectoires cycloïdales des planètes lorsque l'on change de référentiel d'observation.

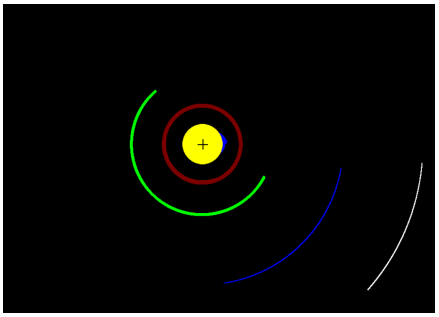

FIGURE 22 – Centrage Soleil

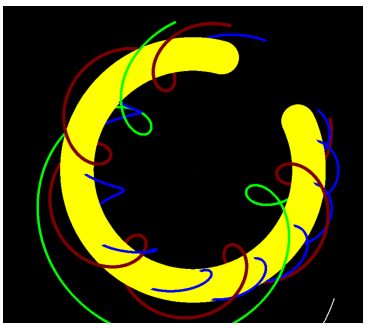

FIGURE 23 – Changement de configuration

5 Conclusion

5.1 Extension du programme

Il est possible de simuler n'importe quel système mécanique avec notre application puisque le RK4 est indépendant de la force utilisée. On pourrait par exemple simuler l'intéraction de 42 particules avec différentes charges.

5.2 Conclusion

Ce projet nous a permis de calculer et simuler des intéractions gravitationnelles entre différents corps. La méthode de résolution des équations différentielles RK4 s'est montrée efficace et suffisante afin d'obtenir un niveau de précision conséquent. Cet algorithme est indépendant du nombre d'objet contenu dans l'ensemble du système et des expressions des différentes forces qui s'appliquent au système. C'est pourquoi il est aisé de généraliser l'interaction de deux corps à N corps.

Enfin, l'outil d'interface graphique PyGame à permis de mettre en évidence certaines caractéristiques. C'est notamment le cas de la vitesse de la trajectoire elliptique en fonction de la distance corps central - corps en orbite (système deux corps). On a aussi pu observer l'influence de la masse d'un corps sur les autres et l'influence du centrage concernant le corps le plus massif (système N corps, Figure 21).

Par conséquent, cet outil est très utile dans un but pédagogique puisqu'il permet de mettre en mouvement des équations qui peuvent sembler abstraites aux premiers abords.

Toutefois, afin d'améliorer cette application, nous pourrions tenir compte de l'impact de la collision entre deux corps. De plus, dans le cas du système solaire, nous pourrions utiliser une API afin de récupérer des données précises concernant les positions et vitesses des corps.