Ce Mini-Projet est constitué de deux parties : la première est à réaliser en Python et concerne le phénomène de synchronisation entre des oscillateurs de fréquence propre différentes, mais couplés entre eux. La seconde partie est à réaliser en C++ et concerne le modèle de Fermi-Pasta-Ulam-Tsingou illustrant l'existence d'ergodicité dans un modèle simple de particules identiques interagissant avec leur plus proches voisins avec un potentiel non harmonique.

1 Le modèle de Kuramoto

On se propose d'étudier par simulation numérique le modèle standard de la synchronisation, celui de Kuramoto. Il a été introduit en 1975 et illustre la faculté pour des oscillateurs de fréquences propres différentes d'évoluer vers un mode oscillatoire global quand le paramètre de couplage est au delà d'une valeur critique. Nous allons étudier le modèle original mais aussi quelques variantes qui ont été introduites depuis bientôt une cinquante d'années.

Avec le langage Python, les temps de simulations doivent être de l'ordre de quelques minutes si les codes ont été suffisamment bien optimisés.

On considère N rotors déposés sur un anneau. Les équations du mouvement sont données par les équations suivantes :

$$\dot{\theta}_i = \omega_i + \sum_{j=1}^{N} \frac{K}{N} \sin(\theta_j - \theta_i) \tag{1}$$

où θ_i et ω_i sont respectivement la phase et la pulsation du rotor i. K est la constante de couplage et le facteur $1/N$ assure que le modèle est bien définie à la limite thermodynamique. Les pulsations $\{\omega_i\}$ sont choisies au début de la simulation comme des nombres aléatoires issus d'une distribution gaussienne centrée de variance unité.

Afin de visualiser la dynamique collective de ce système, on introduit le paramètre d'ordre complexe suivant :

$$r(t)e^{i\psi(t)} = \frac{1}{N} \sum_{i=1}^{N} e^{i\theta_i(t)} \tag{2}$$

Question Préambule : avant de commencer à coder, montrer que les équations du mouvement peuvent se réécrire comme :

$$\dot{\theta}_i = \omega_i + Kr(t) \sin(\Psi - \theta_i) \tag{3}$$

1. On souhaite tout d'abord visualiser une configuration initiale. Créer un premier programme en Python avec une classe **OSCI** dont le constructeur contient un attribut N représentant le nombre d'oscillateurs initialisé par un paramètre d'entrée à la création de l'objet, et un tableau des angles des différents oscillateurs initialisés par une distribution uniforme entre $-\pi$ et π.

   Créer une fonction membre qui affiche dans un graphique un cercle de rayon unité et les phases de chaque oscillateur par un point sur ce cercle (choisir une couleur différente

pour le cercle et les points). Dans cette même fonction calculer le paramètre d'ordre de cette configuration et afficher celui-ci dans le plan (avec une troisième couleur) et sauver la figure dans un ficher avec un nom de la forme **kura1_N.pdf** où N est le nombre d'oscillateurs.

Faire un programme principal créant successivement deux objets de taille $N = 10$, puis $N = 100$. Pour faciliter la vie de vos enseignants ajouter au programme l'instruction :

```
np.random.seed(40)
```

Sauver le programme sous le nom **kura1.py**.

2. Le temps de relaxation du système est suffisamment court pour que l'on puisse choisir de résoudre le système d'équations différentielles à partir d'une méthode adaptative Runge-Kutta telle que **RK45**. Comme il est nécessaire d'avoir les résultats avec un espacement constant on utilisera l'option **t_eval=t** avec **t** un tableau donné par l'instruction suivante :

```
t=np.linspace(0,100,201)
```

A partir du programme précédent ajouter au constructeur un attribut K pour la constante de couplage de l'objet initialisé par un paramètre d'entrée à la création de l'objet, ainsi qu'un attribut tableau des pulsations propres dont les valeurs sont tirées à partir d'une distribution gaussienne centrée en zéro et de variance unité.

Créer ensuite deux fonctions membres de la classe : une fonction associée à l'équation différentielle à résoudre que l'on appellera **Kura**, et une fonction **solve** qui fait appel à la fonction de **solve_ivp** de **scipy.integrate** et qui renvoie la solution donnée par **solve_ivp**.

Dans le programme principal, créer un objet de taille $N = 100$ et $K = 0$, résoudre l'équation différentielle pour une durée égale à 100, puis afficher la configuration finale ainsi que le paramètre d'ordre. Vous sauverez la figure sous le nom **kura2_100.pdf** et le programme sous le nom **kura2.py**.

Suggestion : avec l'instruction suivante dans le programme principal, vous pouvez recopier la dernière configuration des phases et utiliser la première fonction membre pour afficher la configuration.

```
oscillateur.phi=oscillateur.solve().y[:,-1]
```

3. Exécuter le programme précédent avec les paramètres $N = 100$ et $K = 2$. Sauver la figure sous le nom **kura3_100.pdf.** et le programme sous le nom **kura3.py**. Que constate-t-on ? Pourquoi peut-t-on parler de synchronisation ?
4. On cherche maintenant à obtenir une description plus quantitative du phénomène en parcourant différentes valeurs de K. Pour supprimer des oscillations non essentielles, il est nécessaire pour chaque simulation la somme totale des pulsations propres reste *strictement* égale à zéro. Modifier le constructeur de la classe pour obtenir ce résultat.

On souhaite avoir l'évolution temporelle de la valeur absolue du paramètre d'ordre pour différentes valeurs de K. On prendra 6 valeurs de K entre 1 et 2. Modifier le

code précédent et afficher les trajectoires $r(t)$ pour les différentes valeurs de K dans un graphique que l'on enregistrera sous le nom **kura_4.pdf** et le programme sous le nom **kura4.py**. Que constate-t-on ?

5. On cherche à tracer pour 51 valeurs K allant de 1 à 2 la valeur absolue du paramètre d'ordre moyennée sur les temps supérieurs à $t = 50$. Modifier le programme précédent et tracer cette courbe dans une figure nommée **kura5.pdf** et sauvegarder le programme sous le nom **kura5.py**. Quelle est la tendance que l'on observe ?

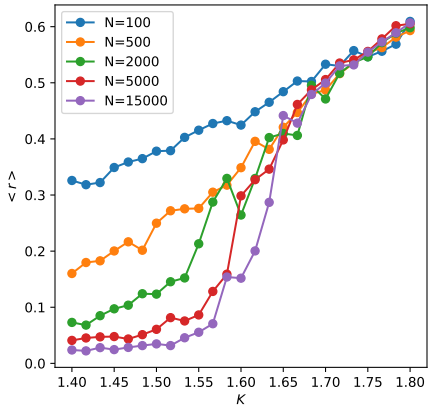
6. Pour supprimer les fluctuations dues au désordre, il est nécessaire de faire la moyenne sur un ensemble de $Nrep$ trajectoires indépendantes sur toutes les valeur de K. Obtenir pour chaque valeur de K la moyenne sur le désordre de la moyenne temporelle de la valeur absolue du paramètre d'ordre.

   Tracer à nouveau $\bar{r}$ en fonction de K avec $Nrep = 10$ pour commencer. Une fois le script opérationnel augmenter à $Nrep = 50$ (ce dernier calcul prend environ 1 minute). Sauver le graphique et le programme sous les noms **kura6.pdf** et **kura6.py**.

7. Modifier le code précédent pour calculer et tracer sur le même graphe le paramètre d'ordre moyenné en fonction de K pour différentes tailles N du système. Pour se focaliser sur la région la plus pertinente, on prendra 25 valeurs de K entre 1.4 et 1.8. Pour les différentes valeurs de N et $Nrep$ à réaliser, voir la table 1.

   Ce calcul nécessite un temps de calcul d'environ 12 minutes et peut être exécuté sur les ordinateurs de l'UFR si votre ordinateur personnel est un peu lent.

   Sauver la figure et le programme sous les noms **kura7.pdf** et **kura7.py**. Vous obtiendrez une figure comparable à la figure suivante :



8. On peut montrer qu'à la limite thermodynamique, on a une transition du second ordre avec une valeur de la constante de couplage K_c

$$K_c = \frac{2}{\pi g(0)} \tag{4}$$

N	Nrep
100	200
500	50
2000	20
5000	15
15000	10

TABLE 1 – Nombre d'oscillateurs et de trajectoires indépendantes

où $g(0)$ est la valeur de la distribution de probabilité à l'origine. De plus, le paramètre d'ordre r se comporte approximativement comme :

$$r = \sqrt{\frac{-16(K - K_c)}{\pi K_c^4 g''(0)}} \tag{5}$$

pour $K > K_c$, où g'' est la dérivée seconde de la distribution de probabilité.

Dans un premier temps, coder cette fonction dans un script séparé **traceexact.py**, et tracer là dans un graphe **traceexact.pdf**.

Joindre ensuite cette fonction au programme précédent pour tracer sur le même graphe le résultat de la limite thermodynamique et les résultats des simulations. Nommer le programme complet **kura8.py** et la figure complète **kura8.pdf**.

9. *Question Bonus :* En fait l'équation donnée plus haut correspond à la solution à l'ordre le plus petit dans un développement en $K-K_c$. L'équation *exacte* pour une distribution gaussienne centrée de variance 1 est donnée par :

$$r = \frac{y\sqrt{2\pi}}{4} e^{y^2/4}(I(0, y^2/4) + I(1, y^2/4)) \tag{6}$$

où $y = K.r$ et $I(n, x)$ est la fonction de Bessel modifiée d'ordre n de première espèce. Montrer en faisant un développement limité en y, que l'on retrouve bien l'équation 5.

Dans un script séparé **traceexact2.py** créer un vecteur y allant de 0.01 à 1.2 de 120 éléments, puis calculer le vecteur r avec la fonction ci-dessus. Les valeurs de K correspondantes peuvent ensuite être calculée avec $K = y/r$. Tracer sur un même graphe **traceexact2.pdf** le résultat exact et le résultat approché de la question précédente.

Intégrer cette second courbe aux résultats de simulation et vérifier que celle-ci représente bien une enveloppe inférieure à tous vos résultats. Nommer le programme final **kura9.py** et la figure complète **kura9.pdf**.

2 Le modèle FPUT

Jusque dans les années 50, on pensait que tout système qui n'était pas intégrable relaxait vers l'équilibre avec le temps. Un exemple de modèle intégrable est la chaîne d'oscillateurs couplés avec un potentiel harmonique. C'est ainsi que le modèle d'une chaîne d'oscillateurs couplés avec une interaction **non** harmonique, qui a été le premier à être étudié par simulation numérique, a crée la surprise. Historiquement, attribué à Fermi, Pasta et Ulam, ce n'est que tardivement que le nom de Mary Tsingou, qui a réalisé la programmation du premier code, a enfin été associé à ce modèle. Je renvoie tous ceux qui seraient intéressés à cet article en ligne qui restaure une vérité trop longtemps oubliée.

Nous allons voir quelques propriétés de ce modèle qui a ouvert le sujet de la physique non linéaire et reste encore d'actualité pour beaucoup de modèles.

On considère un modèle constitué de $N+1$ masses dont le Hamiltonien est donné par :

$$H = \sum_{i=1}^{N-1} \frac{p_i^2}{2m} + \sum_{i=0}^{N-1} V(x_{i+1} - x_i) \tag{7}$$

avec :

$$V(x) = k\frac{x^2}{2} + g\frac{x^3}{3} \tag{8}$$

où p_i représente la quantité de mouvement de la particule i et x_i la position par rapport à sa position d'équilibre. Les particules aux extrémités sont fixes $x_0 = x_N = 0$ et donc $N-1$ particules sont mobiles. Avec un changement d'unités, on peut écrire le Hamiltonien comme :

$$H = \sum_{i=1}^{N-1} \frac{v_i^2}{2} + \sum_{i=0}^{N-1} V(x_{i+1} - x_i) \tag{9}$$

avec :

$$V(x) = \frac{x^2}{2} + \alpha\frac{x^3}{3} \tag{10}$$

Les équations du mouvement sont alors données par :

$$\begin{aligned} \dot{x}_i &= v_i \\ \dot{v}_i &= -2x_i + x_{i+1} + x_{i-1} + \alpha((x_{i+1} - x_i)^2 - (x_{i-1} - x_i)^2) \end{aligned} \tag{11}$$

Pour résoudre celles-ci, on utilisera l'algorithme de Verlet qui est adapté la dynamique moléculaire aux temps longs. Pour un pas de temps constant Δt, celui-ci s'écrit comme :

$$\begin{aligned} x_i(t+\Delta t) =& x_i(t) + \Delta t v_i(t) + \frac{(\Delta t)^2}{2}(-F((x_{i+1} - x_i)(t)) + F((x_i(t) - x_{i-1})(t))) \\ v_i(t+\Delta t) =& v_i(t) + \frac{\Delta t}{2}(-F((x_{i+1} - x_i))(t)) + F((x_i - x_{i-1})(t)) \\ & - F((x_{i+1} - x_i)(t+\Delta t)) + F((x_i - x_{i-1})(t+\Delta t))) \end{aligned} \tag{12}$$

où

$$F(x) = -x - \alpha x^2 \tag{13}$$

1. Écrire une classe **FPUT** dont les attributs sont un entier N pour le nombre de particules, le paramètre α, un double stockant l'énergie totale du système, et des tableaux qui concernent les positions et les vitesses. Pour rendre l'écriture du code plus lisible, il est recommandé de créer une structure **double2** de la manière suivante :

```
typedef struct {
       double x,v;
} double2;
```

   Avec cette définition, on peut créer un **vector <double2> par** qui sera un vecteur où chaque particule aura deux valeurs x et v pour caractériser sa position dans l'espace et sa vitesse à l'instant t. Le constructeur de la classe doit fixer la taille de ce vecteur à $N+1$ et initialiser à 0 les vitesses. Les positions initiales sont données par l'équation :

$$x_i = a \sin\left(\frac{i\pi}{N}\right) \tag{14}$$

   où a est un autre attribut que l'on pourra choisir à l'initialisation de la classe. Ajouter une fonction membre qui calcule l'énergie totale du système pour toute configuration. Dans le programme principal, créer deux objets de type **FPUT** avec $\alpha = 0$, $a = 1$, $N = 32$ et $\alpha = 0.25$, $a = 1$, $N = 32$ et afficher pour ces deux objets le résultat du calcul de l'énergie totale. Enregistrer le code source sous le nom **fput1.cpp**. On doit trouver $E = 0.0770444$ (comme Mary Tsingou l'a obtenue à l'époque) pour ces deux objets. Donner une explication justifiant que l'énergie est la même pour différentes valeurs de α.

2. Pour un potentiel donné, on sait que la solution est donnée par des modes propres $Q_k(t)$ qui s'expriment comme :

$$Q_k(t) = \sqrt{\frac{2}{N}} \sum_{j=1}^{N-1} \sin\left(\frac{jk\pi}{N}\right) x_j(t) \tag{15}$$

   On peut alors associer une énergie par mode qui s'exprime de la manière suivante

$$E_k(t) = \frac{1}{2}\dot{Q}_k^2(t) + \frac{\omega_k^2}{2} Q_k^2(t) \tag{16}$$

   avec

$$\omega_k = 2 \sin\left(\frac{k\pi}{2N}\right) \tag{17}$$

   Ajouter au programme précédent un vecteur des différentes énergies E_k et ajouter une fonction membre pour calculer les différentes énergies pour une configuration du système.
   Pour éviter de calculer des sinus inutilement, il est conseillé d'ajouter dans le constructeur de la classe un vecteur ω de taille $N+1$ donné par la formule ci-dessus, et une matrice $(N+1)\times(N+1)$ dont les éléments sont donnés par : $A[i,j] = \sqrt{\frac{2}{N}} \sin(ij\pi/N)$.

Dans le programme principal calculer les énergies des modes propres et afficher les énergies des 4 premiers modes. Enregistrer le code source sous le nom **fput2.cpp**. Que note-ton ? Pouvez-vous donner une explication à ce résultat ?

3. On cherche à résoudre les équations du mouvement par l'algorithme de Verlet. Afin de faciliter cette partie, la fonction membre **Verlet** donnée ci-dessous permet de calculer la position et la vitesse des $N-1$ particules mobiles à l'instant $t+\Delta t$ en fonction de celles à l'instant t.

```
#include<iostream>
#include<fstream>
#include<vector>
#include<cmath>
#include <iomanip>
using namespace std;

typedef struct {
      double x,v;
    } double2;

class FPUT
{
public:
    int N;
    double alpha=0.25;
    vector <double2> par;
    vector <double2> parn; // next time step
    vector <double> Ek;
    vector <double> omega;
    vector <vector <double>> AA;
    double nrj=0;
    double ampli=0;
    double pi=acos(-1);
    double Deltat=0.01;
    FPUT(int n,double Alpha,double ampli,double Deltat):N(n), alpha(Alpha),
        ↪ ampli(ampli),Deltat(Deltat)
    {
      .....
    }
    double calculE()
    {
      ....

    }
    vector <double> calculEk()
    {
```

```
      .....
    }

    double force(double x)
    {
        return(-x-alpha*x*x);
    }
    void verlet()
    {
        double Deltat2=Deltat*Deltat;
        for (int i=1; i<N; ++i)
        {
            double tmp =-force(par[i+1].x-par[i].x)+force(par[i].x-par[i-1].x
                ↪ );
            parn[i].x = par[i].x+Deltat*par[i].v+0.5*Deltat2*tmp;
        }
        for (int i=1; i<N; ++i)
        {
            double tmp =-force(par[i+1].x-par[i].x) +force(par[i].x-par[i-1].
                ↪ x);
            double tmp2=-force(parn[i+1].x-parn[i].x)+force(parn[i].x-parn[i
                ↪ -1].x);
            parn[i].v=par[i].v+0.5*Deltat*(tmp+tmp2);
        }

        for (int i=0;i<N+1;i++)
        {
            par[i].x=parn[i].x;
            par[i].v=parn[i].v;
        }
    }
};
int main()
{
    int N=32;
    int Nsteps=100000;
    double pi=acos(-1);
    double DeltaT=0.01;
    FPUT fput(N,0.25,1.0,DeltaT);
    ....

    return (0);
}
```

Intégrer et adapter cette partie du code dans votre programme. Dans le programme

principal faire une boucle sur 10^6 pas de temps qui, à chaque pas de temps, appelle la fonction **verlet** de l'objet pour calculer la prochaine configuration.

Pour suivre l'évolution du système, tous les 100 pas de temps enregistrer dans un fichier nommé **energies_alpha.dat** le temps, l'énergie totale, et les énergies des quatre premiers modes (*note :* on évitera si possible d'enregistrer un million de lignes - environ 60 Mo - pour rien !) Enregistrer le code source sous le nom **fput3.cpp**.

Réaliser une simulation avec $\alpha = 0$ puis dans un second temps avec $\alpha = 0.25$. Interpréter les deux résultats en visualisant les énergies en fonctions du temps avec un script python (enregistrer les figures avec les noms **fput_0.00.pdf** et **fput_0.25.pdf**).

Comparer la seconde figure avec la figure historique obtenue par Mary Tsingou. Penser à utiliser les mêmes axes que dans la figure historique, c'est plus simple et physiquement plus judicieux.

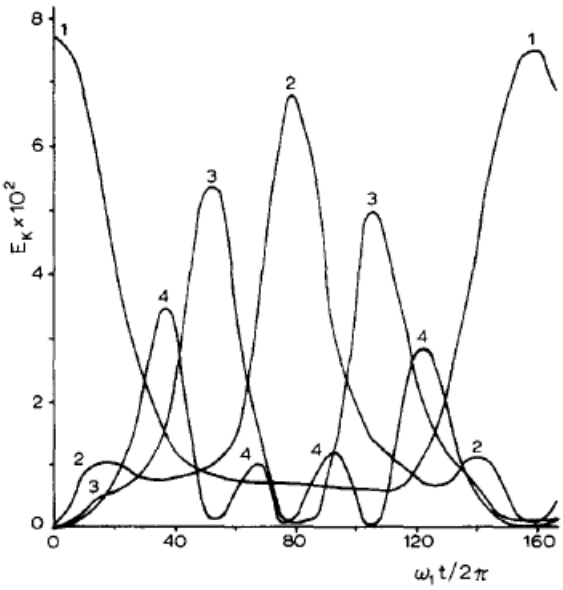


FIGURE 1 – Résultats de Mary Tsingou sur l'évolution des premiers modes normaux d'un modèle FPUT

4. En quoi ce résultat était très inattendu ?

   Faire une simulation avec 30 fois plus de pas de temps et tracer l'évolution temporelle des énergies (fichier **fput_long.pdf**). Que peut-on conclure ?