Interpolation polynomiale - Étude du phénomène de Runge

Importation de packages pour Python

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg as npl
from scipy import interpolate
```

Retour sur le phénomène de Runge

On a vu dans le cours le phénomène de Runge qui se traduit par une mauvaise interpolation, lorsque l'on augmente le degré du polynôme d'interpolation de Lagrange, de la fonction f définie sur $\mathbb{R}$ par

$$f(x) = \frac{1}{1+x^2}.$$

Le but du TP est d'observer ce phénomène mais aussi de mettre en oeuvre une meilleure répartition des points d'interpolation à l'aide des racines des polynômes de Chebyshev et de constater l'atténuation du phénomène de Runge. Plus particulièrement, vous implémenterez des fonctions permettant de calculer le polynôme d'interpolation de Lagrange par la méthode directe. L'interpolation par la méthode de Lagrange et celle de Newton vous est ensuite donnée. Enfin, à partir de la méthode de Newton, vous ferez l'interpolation en utilisant les points de Chebyshev.

Fonction de Runge

> **À faire :** Implémenter une fonction **Runge** qui à un vecteur x sous format numpy retourne le vecteur $f(x)$. Puis, tracer de cette fonction sur l'intervalle $[-3, 3]$.

In [1]:
```
def Runge(x):
    return
```

Construction de points d'interpolation équirépartis

Dans un premier temps, nous allons considérer des points d'interpolation uniformément répartis sur l'intervalle d'interpolation $[a, b]$, $a < b$. L'ensemble des points d'interpolation $(x_j)_{0 \leq j \leq n}$ va donc être donné, pour un certain $n \geq 1$, par

$$x_j = a + (b - a)\frac{j}{n}, \quad 0 \leq j \leq n.$$

> **À faire :** Implémenter une fonction **Interp_Equi** qui prend en arguments d'entrée les valeurs a et b (qui définissent l'intervalle d'interpolation) ainsi que m de manière à retourner un vecteur x de m points d'interpolations équirépartis sur $[a, b]$.

In [2]:
```
def Interp_Equi(a,b,m):
    return
```

Méthode directe de construction d'un polynôme d'interpolation

On rappelle que la méthode directe de construction consiste à simplement poser le système d'équations suivant

$$p(x_j) = y_j, \quad 0 \leq j \leq n,$$

sous la forme d'un système linéaire dont la solution correspond au vecteur $(a_j)_{0 \leq j \leq n}$ de coefficients du polynôme d'interpolation. Plus précisément, le système s'écrit

$$\begin{bmatrix} 1 & x_0 & \cdots & x_0^{n-1} & x_0^n \\ 1 & x_1 & \cdots & x_1^{n-1} & x_1^n \\ \vdots & \vdots & & & \vdots \\ 1 & x_n & \cdots & x_n^{n-1} & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

> **À faire :** Implémenter une fonction **Vandermonde** qui prend en arguments d'entrée un vecteur x de points d'interpolation et qui rend, en sortie, la matrice de Vandermonde associée au système linéaire précédent.

In [4]:
```
def Vandermonde(x):
    return
```

À faire : À l'aide de la fonction **npl.solve**, implémenter une fonction **Methode_directe** qui prend en argument les points d'interpolations x ainsi qu'une fonction f à interpoler et qui rend en sortie un vecteur a qui contient les coefficients du polynôme d'interpolation de f aux points x.

In [5]:
```
def Methode_directe(x,f):
    return
```

À faire : Grâce aux fonctions **PtsInterp_Equi** et **Methode_directe**, tracer sur une même figure la fonction **Runge** sur l'intervalle $[-3, 3]$ ainsi que son polynôme d'interpolation pour 5, 10 et 15 points d'interpolation.

In [ ]:
```

```

À faire : Utiliser à présent une boite noire de python pour retrouver ces résultats. Pour cela, s'appuyer sur la méthode interpolate.KroghInterpolator de scipy.

In [ ]:
```

```

Méthode de Lagrange

Nous avons vu dans le cours que le polynôme d'interpolation de Lagrange peut être calculé par la méthode de Lagrange. Cette méthode consiste à trouver une base de polynômes, qui sont les polynômes de Lagrange, dans laquelle on exprime le polynôme d'interpolation. Étant donné une fonction f et un ensemble $(x_j)_{0\leq j\leq n}$ de points d'interpolation, le polynôme d'interpolation p s'exprime comme

$$p(x) = \sum_{j=0}^{n} f(x_j) L_{j,n}(x), \quad \forall x \in \mathbb{R},$$

avec $(L_{j,n})_{0\leq j\leq n}$ la base de polynôme de Lagrange donnée par

$$L_{j,n}(x) = \prod_{\substack{k=0 \\ k\neq j}}^{n} \frac{x - x_k}{x_j - x_k}.$$

> **À faire :** Écrire une fonction **Methode_Lagrange** qui implémente la méthode précédente avec en arguments d'entrée un vecteur x_interp de points d'interpolation et un vecteur x de points d'abscisse. Cette fonction returnera la matrice L dont chaque colonne correspond au vecteur $\ell_i = L_{i,n}(x)$.

On obtient alors le vecteur $y = p(x)$ grâce au produit matrice-vecteur $y = Lz$ où $z = f(x\_interp)$. Le produit matrice-vecteur peut se faire grâce à la fonction **np.dot**.

In [3]:
```
def Methode_Lagrange(x_interp,x):
    m = len(x_interp)
    n = len(x)
    L = np.ones((n,m))
    for i in range(m):
        for j in range(m):
            if j != i:
                L[:,i] = L[:,i]*(x - x_interp[j])/(x_interp[i] - x_int
    return L
```

> **À faire :** Tracer ensuite sur une même figure la fonction **Runge** sur l'intervalle $[-3, 3]$ ainsi que son polynôme d'interpolation obtenue avec la méthode de Lagrange pour 5, 10 et 15 points d'interpolation.

In [4]:
```
x = np.linspace(-3,3,1000)
plt.plot(x,Runge(x))
for n in [5,10,15]:
    x_interp = Interp_Equi(-3,3,n)
    z = Runge(x_interp)
    L = Methode_Lagrange(x_interp,x)
    P = L.dot(z)
    plt.plot(x,P)
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-4-0766abb9cd7c> in <module>()
----> 1 x = np.linspace(-3,3,1000)
      2 plt.plot(x,Runge(x))
      3 for n in [5,10,15]:
      4     x_interp = Interp_Equi(-3,3,n)
      5     z = Runge(x_interp)

NameError: name 'np' is not defined
```

Méthode de Newton

La méthode de Newton, comme pour celle de Lagrange, permet d'exprimer le polynôme d'interpolation de Lagrange dans une base de polynômes, qui sont les polynômes de Newton. Étant donné une fonction f et un ensemble $(x_j)_{0\leq j\leq n}$ de points d'interpolation, le polynôme d'interpolation p s'exprime comme

$$p(x) = \sum_{j=0}^{n} z_j N_{j,n}(x), \quad \forall x \in \mathbb{R},$$

avec $(N_{j,n})_{0\leq j\leq n}$ la base de polynôme de Newton donnée par

$$N_{j,n}(x) = \prod_{k=0}^{j}(x - x_k),$$

et $(z_j)_{0\leq j\leq n}$ les coefficients qui sont solution du système triangulaire

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & (x_1 - x_0) & 0 & \cdots & 0 \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 1 & (x_n - x_0) & (x_n - x_0)(x_n - x_1) & \cdots & \prod_{j=0}^{n-1}(x_n - x_j) \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

> **À faire :** Écrire une fonction **Methode_Newton** qui implémente la méthode précédente avec en arguments d'entrée un vecteur x_interp de points d'interpolation et un vecteur x de points d'abscisse. Cette fonction returnera la matrice N dont chaque colonne correspond au vecteur $n_i = N_{i,n}(x)$.

In [5]:
```
def Methode_Newton(x_interp,x):
    m = len(x_interp)
    n = len(x)
    N = np.ones((n,m))
    for i in range(m):
        for j in range(i):
            N[:,i] = N[:,i]*(x - x_interp[j])
    return N
```

> **À faire :** Écrire une fonction **Coeff_Newton** qui permet de calculer les coefficients $(z_j)_{0\leq j\leq n}$ en résolvant le système triangulaire inférieur par un algorithme de descente. Cette fonction prendra en entrée le vecteur x_interp des points d'interpolation ainsi que la fonction f à interpoler et rendra le vecteur z des coefficients.

In [6]:
```
def Coeff_Newton(x_interp,f):
    m = len(x_interp)
    T = np.zeros((m,m))
    for i in range(m):
        for j in range(m):
            if j == 0:
                T[i,j] = 1
            elif j<=i:
                T[i,j] = T[i,j-1]*(x_interp[i] - x_interp[j-1])
    z = npl.solve(T,f(x_interp))
    return z
```

On obtient alors le vecteur $y = p(x)$ grâce au produit matrice-vecteur $y = Lz$.

> **À faire :** Tracer ensuite sur une même figure la fonction **Runge** sur l'intervalle $[-3, 3]$ ainsi que son polynôme d'interpolation obtenue avec la méthode de Newton pour 5, 10 et 15 points d'interpolation.

In [7]:
```
x = np.linspace(-3,3,1000)
plt.plot(x,Runge(x))
for n in [5,10,15]:
    x_interp = Interp_Equi(-3,3,n)
    z = Coeff_Newton(x_interp,Runge)
    N = Methode_Newton(x_interp,x)
    P = N.dot(z)
    plt.plot(x,P)
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-7-08722acc7fd3> in <module>()
----> 1 x = np.linspace(-3,3,1000)
      2 plt.plot(x,Runge(x))
      3 for n in [5,10,15]:
      4     x_interp = Interp_Equi(-3,3,n)
      5     z = Coeff_Newton(x_interp,Runge)

NameError: name 'np' is not defined
```

Points d'interpolation de Chebyshev

Les points de Chebyshev dans l'intervalle $[-1, 1]$ sont donnés par la formule suivante

$$y_j = \cos\left(\frac{2i+1}{2(n+1)}\pi\right), \quad 0 \leq j \leq n.$$

Afin d'adapter ces points à un intervalle $[a, b]$, on se basera sur la formule suivante

$$x_j = \frac{a+b}{2} + \frac{a-b}{2} y_j, \quad 0 \leq j \leq n.$$

> **À faire :** Implémenter une fonction **Interp_Chebyshev** qui prend en arguments d'entrée les valeurs a et b (qui définissent l'intervalle d'interpolation) ainsi que m de manière à retourner un vecteur x de m points d'interpolations répartis selon les points de Chebyshev sur $[a, b]$.

In [14]:
```
def Interp_Chebyshev(a,b,m):
    return
```

> **À faire :** Tracer ensuite sur une même figure la fonction **Runge** sur l'intervalle $[-3, 3]$ ainsi que son polynôme d'interpolation obtenue avec la méthode de Newton pour 5, 10 et 15 points d'interpolation.

In [ ]:
```

```

Comparaison d'erreur selon la répartition des points d'interpolation

Nous sommes à présent en mesure de quantifier numériquement la réduction de l'erreur qu'apporte la répartition suivant les points de Chebyshev en comparaison avec les points équirépartis. On introduit pour cela l'erreur

$$e(n) = \sup_{x\in[-3,3]} |f(x) - p(x)|,$$

où f est la fonction à interpoler et p est le polynôme d'interpolation de Lagrange de degré n. On veut comparer l'évolution de l'erreur en fonction de n et du choix de la répartition des points d'interpolation.

> **À faire :** Écrire une fonction **Erreur_Interp** permettant de calculer l'erreur $e(n)$ dans le cas de points d'interpolation équirépartis et le cas des points de Chebyshev. Cette fonction prendra en entrée l'entier n ainsi qu'un entier p et donnera en sortie un scalaire correspondant à l'erreur $e(n)$ pour les points équirépartis si $p = 0$ ou pour les points de Chebyshev si $p = 1$.
> L'interpolation se fera à l'aide de la méthode de Newton sur l'intervalle $[-3, 3]$.

In [8]:
```
def Erreur_Interp(n,p):
    return
```

> **À faire :** Tracer sur une même figure l'évolution de l'erreur $\log(e(n))$, en fonction de n, pour les points d'interpolation équirépartis et pour les points de Chebyshev de 1 jusqu'à 100 points.

In [17]:
```

```

In [ ]:
```

```