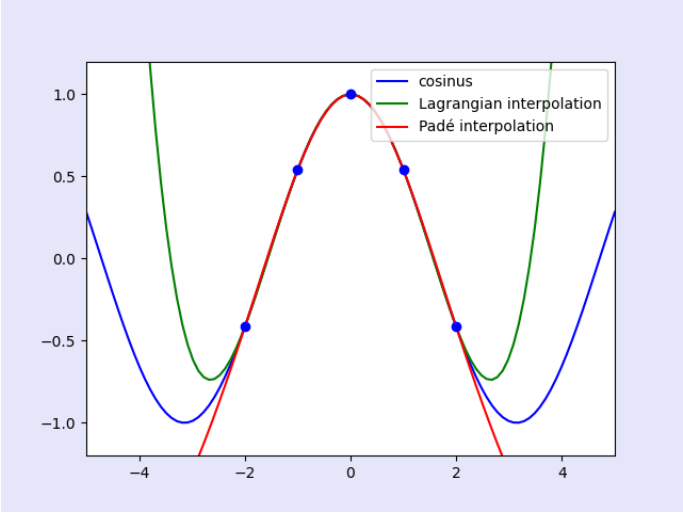
Notebook : Interpolation de Padé

Solution détaillée - Méthodes Numériques

Tom Janssen

Février 2026



Ce document a pour but d'aider les étudiants du cours de l'excellent cours de Méthodes Numériques, donné par M. Vincent Legat à l'Ecole Polytechnique de Louvain, en leur offrant une solution détaillée aux devoirs

1 Le problème à résoudre

L'objectif de ce devoir est de réaliser une interpolation rationnelle à l'aide de l'approximant de Padé. Contrairement à l'interpolation de Lagrange qui utilise un seul polynôme, Padé utilise un quotient de deux polynômes.

1.1 Arguments des fonctions

- **X** : vecteur numpy unidimensionnel des abscisses de longueur $2n+1$.
- **U** : vecteur numpy unidimensionnel des ordonnées de longueur $2n+1$.
- **a** : Le vecteur des coefficients calculés par `padeInterpolationCompute`.
- **x** : Les points d'abscisses où l'on souhaite évaluer l'approximant.

2 Le système à résoudre

Pour un n donné, l'interpolation s'écrit sous la forme :

$$u(x) \approx u^h(x) = \frac{a_0 + a_1 x + \cdots + a_n x^n}{1 + a_{n+1} x + \cdots + a_{2n} x^n} \tag{1}$$

Pour trouver les coefficients a_k, nous imposons que $u^h(X_i) = U_i$ pour chaque point. Pour linéariser le système, on multiplie par le dénominateur (différent de zéro, espérons le...) :

$$(a_0 + a_1 X_i + \cdots + a_n X_i^n) - U_i(a_{n+1} X_i + \cdots + a_{2n} X_i^n) = U_i \tag{2}$$

Ceci nous permet de construire un système matriciel de la forme $Aa = U$, où A est une matrice carrée de taille $(2n+1) \times (2n+1)$. Pour un n quelconque, la matrice A prend la structure suivante :

$$A = \begin{pmatrix} 1 & X_0 & \dots & X_0^n & -U_0 X_0 & \dots & -U_0 X_0^n \\ 1 & X_1 & \dots & X_1^n & -U_1 X_1 & \dots & -U_1 X_1^n \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & X_{2n} & \dots & X_{2n}^n & -U_{2n} X_{2n} & \dots & -U_{2n} X_{2n}^n \end{pmatrix} \tag{3}$$

a est le vecteur colonne des coefficients de notre polynôme d'interpolation et U les différentes ordonneés.

3 Implémentation Python

Une fois qu'on a bien compris comment le système était formé, (il est recommandé de le faire à la main pour des cas simples, le cas n = 2 vous a gentillement été livré dans les consignes), l'implémentation python ne requiert qu'un peu de dextérité concernant les indices dans les boucles.

```python
from numpy import *
from numpy.linalg import solve

def padeInterpolationCompute(X, U):
    N = len(X)
    n = (N - 1) // 2
```

```
7
8       # Construction de la matrice A
9       A = zeros((N, N))
10      for i in range(N):
11          # Partie numerateur (colonnes 0 a n)
12          for j in range(n + 1):
13              A[i, j] = X[i]**j
14          # Partie denominateur (colonnes n+1 a 2n)
15          for j in range(n + 1, N):
16              A[i, j] = -U[i] * (X[i]**(j - n))
17
18      # Resolution du systeme lineaire
19      a = solve(A, U)
20      return a
21
22  def padeEval(a, x):
23      n = (len(a) - 1) // 2
24
25      # Calcul des polynomes dans la fraction num et denom
26      num = 0.0
27      denom = 1.0 # Terme constant fixe a 1
28      for j in range(n + 1):
29          num += a[j] * (x**j)
30      for j in range(n + 1, len(a)):
31          denom += a[j] * (x**(j - n))
32
33      return num / denom
```

4 Résultats

Et zou ! On obtient des jolis graphes d'interpolation. L'un des avantages majeurs de l'approche de Padé est qu'elle est souvent plus efficace qu'un développement en série de Taylor pour évaluer des fonctions comme le cosinus. Comme on peut le voir sur les graphiques de test, elle suit la courbure de la fonction originale de manière très fidèle, même avec peu de points.
Ce premier devoir était relativement simple et n'est qu'une mise en bouche des autres, je vous conseille vivement d'essayer les suivants par vous-mêmes... Pour toute question, n'hésitez pas à demander à vos super tuteurs et tutrices qui sont particulièrement bienveillants cette année :)

5 Version Vectorisée (Bonus)

L'énoncé suggère que les étudiants habiles et futéspeuvent proposer une implémentation plus rapide. Voici comment vectoriser et ainsi supprimer les boucles for en utilisant le broadcasting de Numpy :).

5.1 L'astuce du Broadcasting

Au lieu de calculer X_i^j élément par élément, on considère X comme une colonne et le vecteur des puissances $J = [0, 1, \ldots, n]$ comme une ligne. On obtient ainsi une matrice A telle que $A_{i,j} = X_i^j$.

En pratique, si $X \in \mathbb{R}^N$, alors `X[:,None]` est de la forme $(N, 1)$ et `arange(n+1)` a la forme $(n+1,)$. Lorsqu'on calcule `X[:,None] ** arange(n+1)`, Numpy 'diffuse' (broadcast) automatiquement ces tableaux : la colonne $(N, 1)$ est répétée sur $(n+1)$ colonnes et la ligne des puissances est répétée sur N lignes, ce qui produit directement une matrice de forme $(N, n+1)$.

```
def padeInterpolationCompute(X, U):
    N = len(X)
    n = (N - 1) // 2

    # Preparation des formes pour le broadcasting
    # X devient une colonne (N, 1)
    X_col = X[:, newaxis]

    # Partie Numerateur : X^0 a X^n
    # On eleve la colonne X a la puissance de la ligne [0, 1, ..., n]
    # Resultat : Matrice (N, n+1)
    pow_num = arange(n + 1)
    A_num = X_col ** pow_num

    # Partie Denominateur : -U * X^1 a -U * X^n
    pow_den = arange(1, n + 1)
    # On multiplie par -U (colonne) element par element:)
    A_den = -U[:, newaxis] * (X_col ** pow_den)

    # Assemblage final de la matrice A
    A = hstack((A_num, A_den))

    return solve(A, U)
```

5.2 Évaluation vectorisée de $u_h(x)$

Une fois les coefficients a calculés, on veut évaluer

$$u_h(x) = \frac{a_0 + a_1 x + \cdots + a_n x^n}{1 + a_{n+1} x + \cdots + a_{2n} x^n}.$$

Là encore, on évite les boucles en construisant la matrice des puissances de x grâce au broadcasting.

```
def padeEval(a, x):
    # n est tel que len(a)=2n+1
    n = (len(a) - 1) // 2

    # Force x a etre un tableau 1D (fonctionne aussi si x est un scalaire)
    x_in = array(x, dtype=float, copy=False, ndmin=1)

    # Preparation des puissances pour le broadcasting
    x_col = x_in[:, newaxis]          # (M, 1) si x contient M points

    # Numerateur : a0 + a1 x + ... + an x^n
    pow_num = arange(n + 1)           # (n+1,)
    V_num = x_col ** pow_num          # (M, n+1)
    num = V_num @ a[:n + 1]           # (M,)

    # Denominateur : 1 + a_{n+1} x + ... + a_{2n} x^n
    pow_den = arange(1, n + 1)        # (n,)
    V_den = x_col ** pow_den          # (M, n)
```

```
    den = 1.0 + V_den @ a[n + 1:]      # (M,)

    u = num / den

    # Si x est scalaire, on renvoie un scalaire
    if isscalar(x) or x_in.shape == (1,):
        return float(u[0])
    return u
```

On construit implicitement une matrice de type Vandermonde (comme vu au CM), puis on effectue des produits matrice-vecteur.