Projet 2 Calcul scientifique numérique

Mariot Clement et Marchand Leo

19 novembre 2023

1 Introduction

De la même manière que dans le projet précédent, notre but est d'approximer un problème à frontière libre en C. En l'occurence, le problème est exactement le même que celui du projet précédent. La différence se fait au niveau du calcul de la fonction f_3 et nous avons donc étudier l'approximation de l'équation de Fredholm de deuxième espèce. Pour cela, nous avons utilisé deux méthodes d'approximation : la décomposition d'Adomain ainsi que la méthode des noyaux itérés et nous avons comparé les résultats obtenus. Ensuite, il a fallu reprendre le problème afin d'obtenir une nouvelle résolution du problème de Cauchy et enfin reprendre l'algorithme de Newton en l'adaptant à ce nouveau projet.

2 Méthode de décomposition d'Adomain

1. La première méthode que nous avons implémenté est la méthode de décomposition d'Adomain. Cette méthode revient à approcher la fonction u par :

$$u(x) = \sum_{n=0}^{\infty} u_n(x)$$

La suite de fonctions u_n est définie par réccurence :

$$\begin{cases} u_0(x) = f(x) \\ u_{n+1}(x) = \lambda \int_a^b k(x,t)u_n(t)dt \end{cases} \quad (1)$$

Nous allons expliciter la version direct de la suite associée à la méthode de décomposition de adomain :

$$f_n^{\alpha} = U_n(x) \quad (2)$$

$$f_n^\alpha = \lambda \int_0^L k(x, s_n) U_{n-1}(s_n) ds_n \tag{3}$$

$$f_n^\alpha = \lambda \int_0^L k(x, s_n) (\lambda \int_0^L k(s_n, sn-1) U_{n-2}(s_{n-1}) ds_{n-1}) ds_n \tag{4}$$

On réarange les intégrales :

$$f_n^\alpha = \int_0^L \int_0^L \lambda^2 k(x, s_n) k(s_n, s_{n-1}) U_{n-2}(s_{n-1}) ds_n ds_{n-1} \tag{5}$$

Par récurrence on obtient :

$$f_n^\alpha = \int_0^L \dots \int_0^L \lambda^n k(x, s_n) U_0(s_1) \prod_{i=1}^{n-1} k(s_i, s_{i+1}) ds_1 \dots ds_n \tag{6}$$

En prenant la convention que $s_0 = x$ et $U_0(x) = \frac{h(x)}{\alpha}$ on a :

$$f_n^\alpha = \int_0^L \dots \int_0^L \lambda^n \frac{h(s_1)}{\alpha} \prod_{i=1}^{n} k(s_{i-1}, s_i) ds_1 \dots ds_n \tag{7}$$

Donc avec $K(x, s_1, ..., s_n)$ une fonction de $h(s_1)$, de $\prod_{i=1}^n k(s_{i-1}, s_i)$ et de λ on a donc le résultat:

$$f_n^\alpha = \int_0^L \dots \int_0^L K(x, s_1, ..., s_n) ds_1 \dots ds_n \tag{8}$$

On a pas utiliser cette méthode de calcul de la suite (U_n) car elle est bien plus compliquée à implementer et nous n'avons pas réussis à la rendre moins coûteuse en temps de calcul que la définition par récurrence. Ainsi par la suite on utilisera un calcul par récurrence de la suite pour la méthode de décompisition de Adomain.

On a donc eu à programmer cette méthode pour approximer f_3. Nous allons donc par la suite détailler le code établi.

2. A l'aide de l'expression précédente, on peut trouver une nouvelle expression de

3. Pour ce code, nous avons repris quelques fonctions du projet précédent : - la solution exacte :

```
//On déclare ici les fonctions associées à la solution excate

double u_ex(double x, double y){
    double pi = 3.14159265358979323851;
    return (cosh(pi*y)*cos(pi*x));
}

double f0(double x){
    double pi = 3.14159265358979323851;
    return (cos(pi*x));
}

double q0(double x){
    return 0;
}
```

- la fonction d'approximation d'intégrale de Gauss Legendre que nous avons adaptée pour fonctionner avec un vecteur de cinq points en entrée :

```
double integrate_vect(double borne_inf, double borne_sup, double U[5]){
    //Méthode de Gauss-Legendre à 5 points avec un vecteur U de taille 5 contenant la valeur de la fonction à integrer aux 5 points de Legendre
    double wi[] = {(322.0 - 13.0*sqrt(70.0))/900.0, (322.0 + 13.0*sqrt(70.0))/900.0, 128.0/225.0,
    (322.0 + 13.0*sqrt(70.0))/900.0, (322.0 - 13.0*sqrt(70.0))/900.0};
    int taille_i = 5;
    double result = 0;
    int i=0;
    for ( i = 0; i < 5; i++)
    {
        result = result + wi[i]*U[i];
    }
    result = ((borne_sup - borne_inf)/2)*result;
    return result;
```

- la définition des fonctions k et h utile pour le calcul de f_3 :

```
double k(int it_max, double H, double L, double x, double t){
    int i;
    double somme = 0;
    double pi = 3.14159265358979323851;
    for ( i = 1; i < it_max; i++)
    {
        somme = somme + (i*cos(i*pi*x/L)*cos(i*pi*t/L))/sinh(i*pi*H/L);
    }
    return (1/(H*L) + 2*pi/(L*L)*somme);
}
```

```c
double h(int it_max, double xh, double H, double L){
    double pi = 3.14159265358979323851;
    double somme = 0.0;
    int k;
    //on constuit l'intégrande
    double integrande_h(double t, int i){
        return (cos(i*pi*t/L)*cos(pi*t));
    }
    for (k = 1; k < it_max; k++)
    {
        somme = somme + ((k*cos(xh*k*pi/L))/sinh(k*pi*H/L))*cosh(H*k*pi/L)*integrate_m(0, L, k, integrande_h);
    }
    double f_0(double x){
        double pi = 3.14159265358979323851;
        return (cos(pi*x));
    }
    double A_0(double L){
        return ((1/L)*integrate(0, L, f_0));
    }

    return ((1/H)*A_0(L) + (2*pi/(L*L))*somme);
}
```

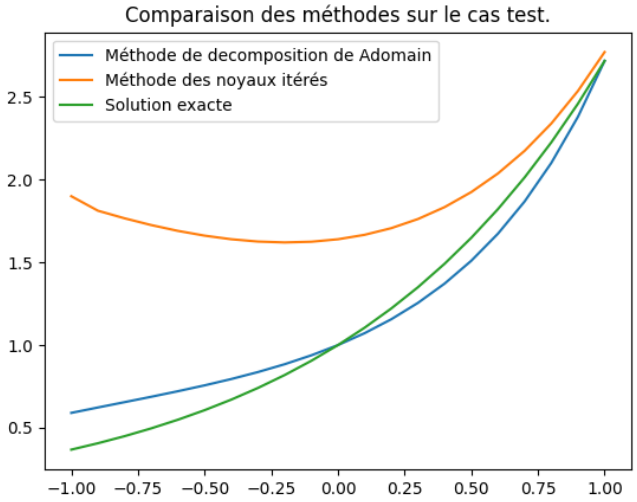
On a ensuite pu définir la fonction Adomain qui est donc notre nouvel outil d'approximation de f_3 :

```c
double adomain(double x, double lambda, int n, double a, double b, double (k)(int, double, double, double, double),
double (f)(double), double H, double L, double it_max){
    int i;
    int j;
    int l;
    //On définit les points xi de Legendre
    double ti[] = {-(1.0/3.0)*sqrt(5 + 2*sqrt(10.0/7.0)), -(1.0/3.0)*sqrt(5 - 2*sqrt(10.0/7.0)), 0,
     (1.0/3.0)*sqrt(5 - 2*sqrt(10.0/7.0)), (1.0/3.0)*sqrt(5 + 2*sqrt(10.0/7.0))};

    //On renormalise pour être sur l'intervalle [0,1]
    for ( i = 0; i < 5; i++)
    {
        ti[i] = ((b - a)/2)*ti[i] + (a + b)/2;
    }

    //On initialise la somme de U_n
    double somme = f(x);
```

```c
//on test le cas où f(x)=0
if (somme == 0)
{
    somme = 1e-5;
}

//On initialise la boucle sur n
double U[5];
double U_plus[5];
double W[5];
double Z[5];
for ( i = 0; i < 5; i++)
{
    for ( l = 0; l < 5; l++)
        {
            W[l] = k(it_max, H, L, ti[i], ti[l])*f(ti[l]);
        }
    U[i] = lambda*integrate_vect(a,b,W);
    //printf("U[%d] = %lf\n", i, U[i]);
    //printf("k(x,ti[%d]) = %lf\n", i, k(it_max, H, L, x, ti[i]));
}
```

```
for ( i = 0; i < 5; i++)
{
    Z[i] = k(it_max, H, L, x, ti[i])*f(ti[i]);
}
somme = somme + lambda*integrate_vect(a,b,Z);

//On fait la boucle sur n
for ( i = 1; i < n; i++)
{
    for ( j = 0; j < 5; j++)
    {
        for ( l = 0; l < 5; l++)
        {
            W[l] = k(it_max, H, L, ti[j],ti[l])*U[l];
        }

        U_plus[j] = lambda*integrate_vect( a, b, W);
        //printf("U_plus[%d] = %lf\n", j, U_plus[j]);
    }

    for ( l = 0; l < 5; l++)
    {
        W[l] = k(it_max, H, L, x, ti[l])*U_plus[l];
    }

    somme = somme + lambda*integrate_vect( a, b, W);
    //printf("somme partielle = %lf\n", somme);
    //printf(" n = %d\n", i);

    //On fait une seconde boucle pour décaler les indices
    for ( j = 0; j < 5; j++)
    {
        U[j] = U_plus[j];
    }
}
return somme;
```

4. Enfin, pour valider la méthode, nous l'avons utilisée pour approcher une fonction dont on connaît la valeur exacte de l'intégrale. Nous avons pris l'exemple du poly page 10 qui est : $u(x) = e^x - x + x \int_0^1 tu(t)dt$. On devait alors trouver comme approximation $u(x) = e^x$. Voici ce que l'on obtient:



5. On remarque que la méthode de décomposition de Adomain approche le comportement de la solution exacte mais la méthode des noyaux itérés ne semble pas convenir pour x lorsque ce dernier n'est pas dans le voisinage de 1. Ces erreurs peuvent avoir plusieurs origines telles qu'un bug dans le code ou alors le fait que l'on itère seulement vingt fois, etc...

3 Méthode des noyaux itérés

1. La méthode des noyaux itérés est semblable à la méthode précédente. La différence se fait sur la création de la suite. En effet, il n'est plus question d'utiliser la série des u_n pour trouver la fonction recherchée. La fonction va ici être approché par la limite de la suite lorsque $n \longrightarrow +\infty$. Il y'a donc un changement dans l'expression de la suite des u_n qui devient :

$$u_{n+1}(x) = f(x) + \lambda \int_a^b k(x,t)u_n(t)dt$$

   On prendra $u_o(x) = f(x)$ comme pour la méthode d'Adomain.

2. Voici donc le code utilisé pour la méthode; ce code est très similaire à celui de la méthode d'Adomain.

```c
double noyaux_iteres(double x, double lambda, int n, double a, double b, double (k)(int, double, double, double, double),
 double (f)(double), double H, double L, double it_max){

    //On utilise quasiment le même code que pour adomain mais on évalue en x seulement le dernier itéré Un

    int i;
    int j;
    int l;
    //On définit les points xi de Legendre
    double ti[] = {-(1.0/3.0)*sqrt(5 + 2*sqrt(10.0/7.0)), -(1.0/3.0)*sqrt(5 - 2*sqrt(10.0/7.0)), 0,
    (1.0/3.0)*sqrt(5 - 2*sqrt(10.0/7.0)), (1.0/3.0)*sqrt(5 + 2*sqrt(10.0/7.0))};

    //On renormalise pour être sur l'intervalle [a,b]
    for ( i = 0; i < 5; i++)
    {
        ti[i] = ((b - a)/2)*ti[i] + (a + b)/2;
    }

    //On initialise la boucle sur n
    double U[5];
    double U_plus[5];
    double W[5];
    double resultat;
    for ( i = 0; i < 5; i++)
    {
        for ( l = 0; l < 5; l++)
            {
                W[l] = k(it_max, H, L, ti[i], ti[l])*f(ti[l]);
            }
        U[i] = f(ti[i]) + lambda*integrate_vect(a,b,W);
        //printf("U[%d] = %lf\n", i, U[i]);
        //printf("k(x,ti[%d]) = %lf\n", i, k(it_max, H, L, x, ti[i]));
```

```
for ( j = 0; j < 5; j++)
    {
        W[j] = k(it_max, H, L, x, ti[j])*U[j];
    }


resultat = f(x) + integrate_vect(a,b,W);
//printf("U_app_1(1) = %lf\n", resultat);

//On fait la boucle sur n
for ( i = 2; i < (n - 1); i++)
{
    for ( j = 0; j < 5; j++)
    {
        for ( l = 0; l < 5; l++)
        {
            W[l] = k(it_max, H, L, ti[j],ti[l])*U[l];
        }

        U_plus[j] = f(ti[j]) + lambda*integrate_vect( a, b, W);
        //printf("U_plus[%d] = %lf\n", j, U_plus[j]);
    }

    for ( j = 0; j < 5; j++)
    {
        W[j] = k(it_max, H, L, x, ti[j])*U[j];
    }


    resultat = f(x) + integrate_vect(a,b,W);
   // printf("U_app_%d(1) = %lf\n", i, resultat);

    //On fait une seconde boucle pour décaler les indices
    for ( j = 0; j < 5; j++)
    {
        U[j] = U_plus[j];
    }
}
```

```
//on évalue l'itéré suivant en x
for ( i = 0; i < 5; i++)
{
    W[i] = k(it_max, H, L, x, ti[i])*U[i];
}


resultat = f(x) + integrate_vect(a,b,W);
return resultat;
```

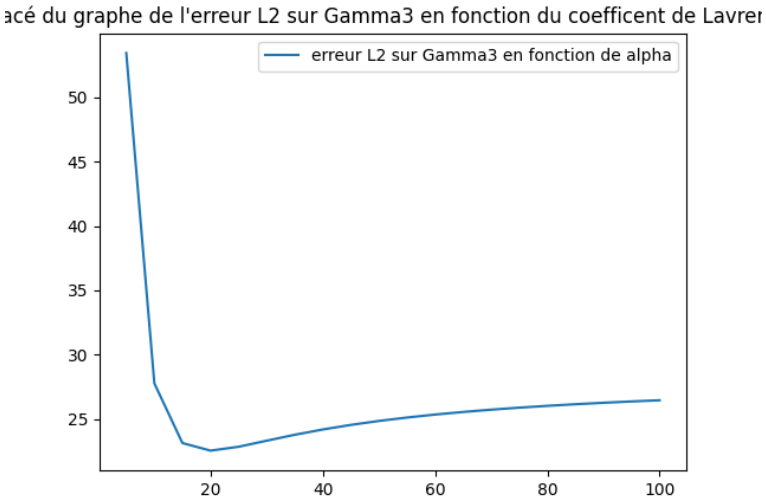
3. Enfin, de même que pour Adomain, nous avons tester cette méthode sur un cas simple où la solution exacte est facile à calculer. Nous sommes alors tomber sur un résultat similaire.

4 Résolution du problème de Cauchy

Dans cette partie, nous avons repris la structure du code du projet précédent en modifiant juste l'approximation de f_3 avec les deux nouvelles méthodes. Nous allons donc comparer les différentes erreurs obtenues avec les deux méthodes ainsi que la valeur du α optimal.

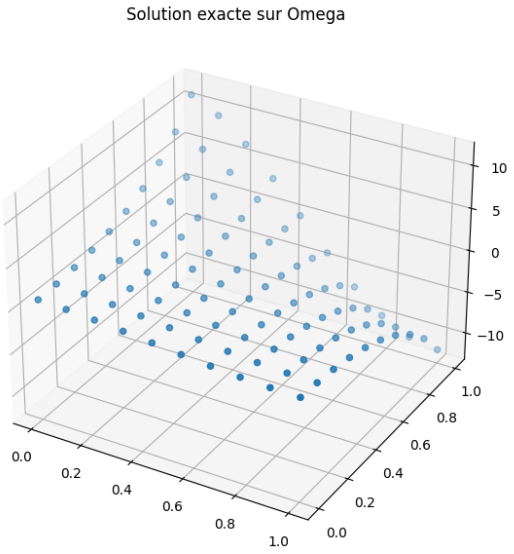
- **Pour la méthode d'Adomain** :

On commence par la recherche du α optimal. On obtient ainsi la courbe suivante :

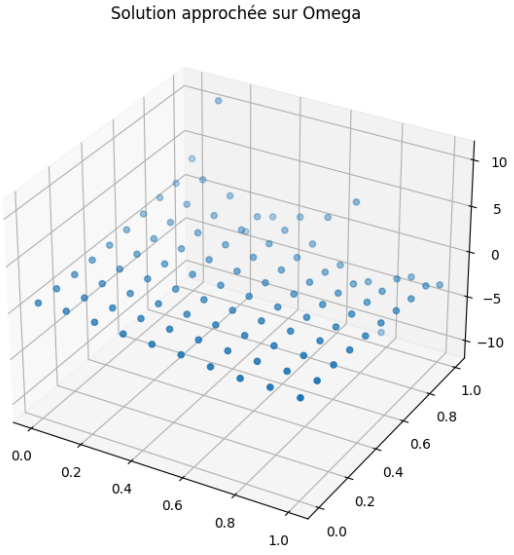


On en déduit ainsi empiriquement que $\alpha = 20$ et c'est celui que l'on utilisera par la suite pour la méthode de décomposition de Adomain.

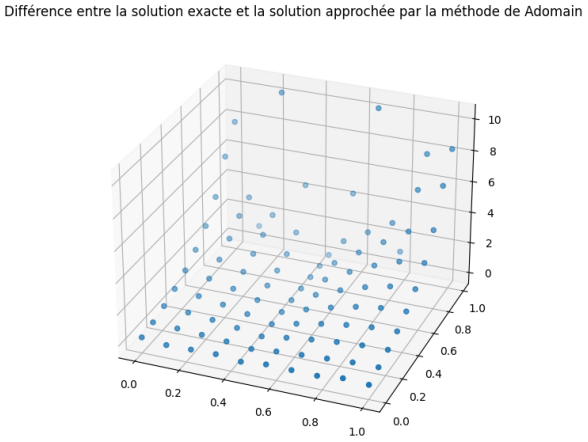
On trace la solution exacte sur Ω :



Puis la solution approchée par la méthode de décomposition de Adomain :



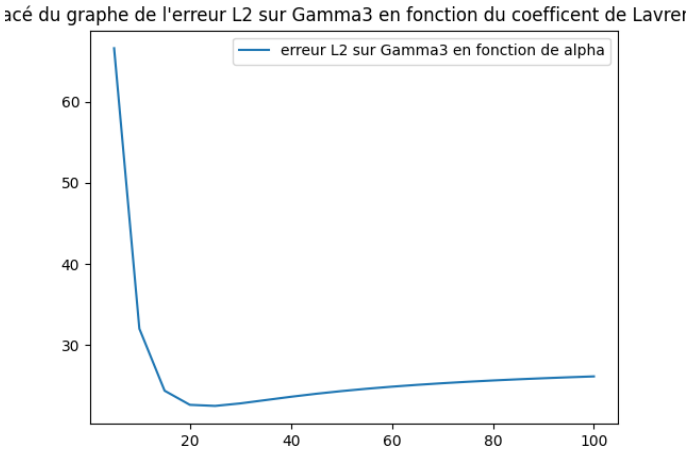
Et enfin la différence entre la solution approchée et exacte :



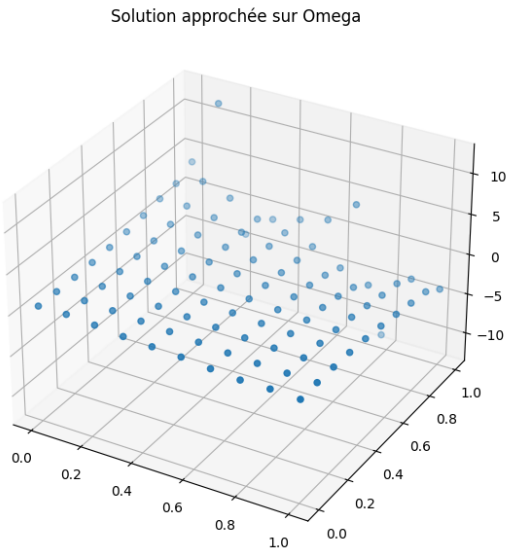
On remarque ainsi que la grande partie de l'erreur commise par la solution approchée est au voisinage de Γ_3. Cela nous indique potentiellement une mauvaise approximation de la fonction f_3 qui est la condition sur ce bord. Maintenant, regardons si la méthode des noyaux itérés donne de meilleures approximations.

- **Pour la méthode des noyaux itérés** :

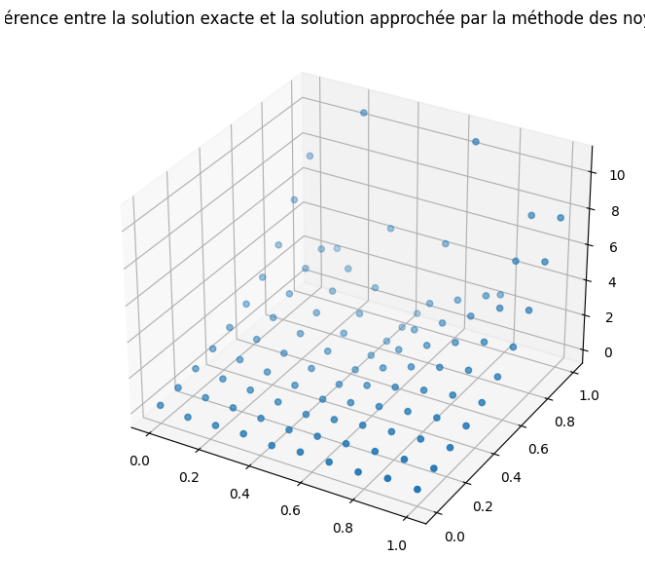
On recommence par une recherche du α optimal pour cette nouvelle méthode de résolution de l'intégrale de seconde espèce de Fredholm régularisée, ainsi on obtient :



Cela mène à choisir $\alpha = 21$ comme optimum. On trace maintenant la solution approchée par la méthode des noyaux itérés sur Ω :



Enfin voici la différence entre la solution exacte et la solution approchée par la méthode des noyaux itérés:



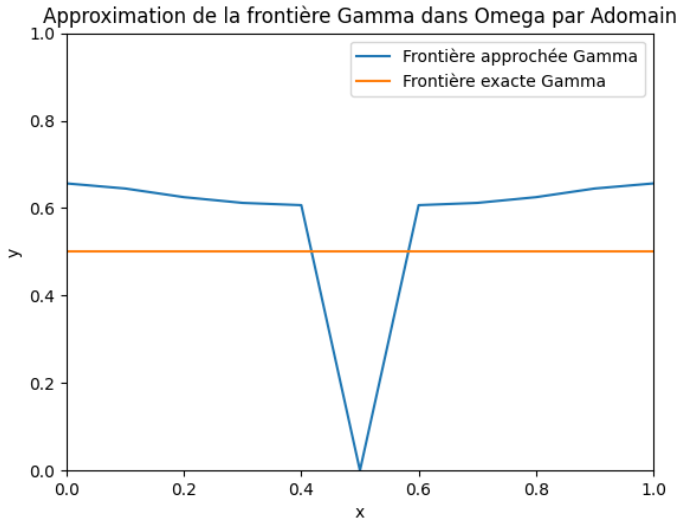
On remarque que l'erreur commise en chaque point du maillage est sensiblement la même que pour la méthode de décomosition de Adomain ce qui révèlerait que la résolution du problème de cauchy n'est pas valide ou bien le code contient encore quelques bugs.

5 Algorithme de Newton

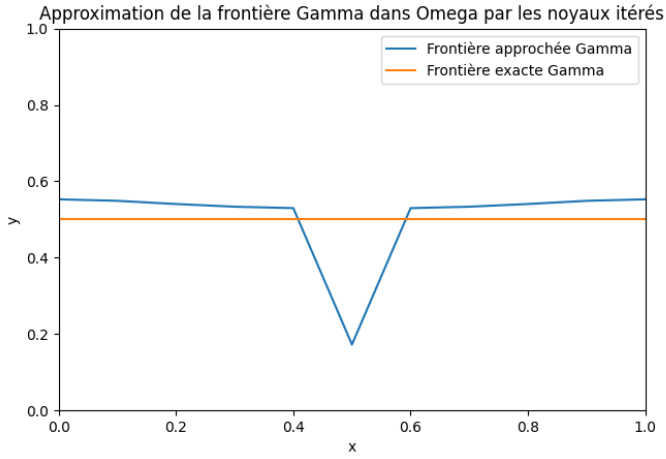
De même que dans la section précédente, nous avons repris le programme du projet précédent pour réaliser la recherche de la forma de la frontière à l'aide de l'algorithme de Newton. Ce dernier est donc codé comme celà :

```c
//On implémente l'algorithme de newton
double Newton(int max, double H, int m, double (f)(double, int), double (f_prime)(double, int)){
    double U_k;
    int i;
    //on initialise la suite avec un nombre aléatoire compris entre 0 et L
    /*
    srand(time(NULL));
    U_k = H*rand()/(RAND_MAX+1.0);
    */
   U_k = 0.5;
    //On itère jusqu'à max pour éviter d'avoir à faire de test logique
    for ( i = 0; i < max; i++)
    {
        U_k = U_k - (f(U_k, m)/f_prime(U_k, m));
    }
    /*
    if ((U_k < 0.0) || (U_k > H))
    {
        U_k = Newton(max, H, m, f, f_prime);
    }
    */

    return U_k;
}
```

On rappelle que la frontière ne peut être trouvée en $x = 0.5$ car il n'y a pas unicité de la solution car la solution exacte donne $u_{ex}(0.5, y) = 0$, $\forall y \in [0; H]$. On recherche maintenant une frontière constante d'équation $y = 0.5$. Ce qui nous donne pour la méthode de décompostion de Adomain :

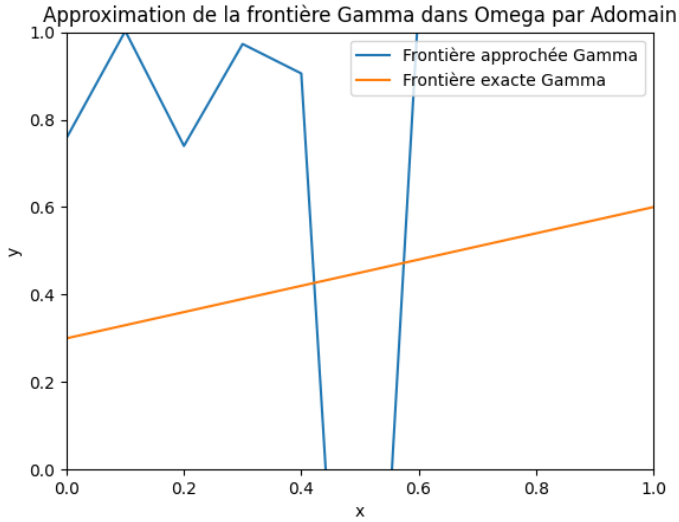


Et de la même manière pour la méthode des noyaux itérés :

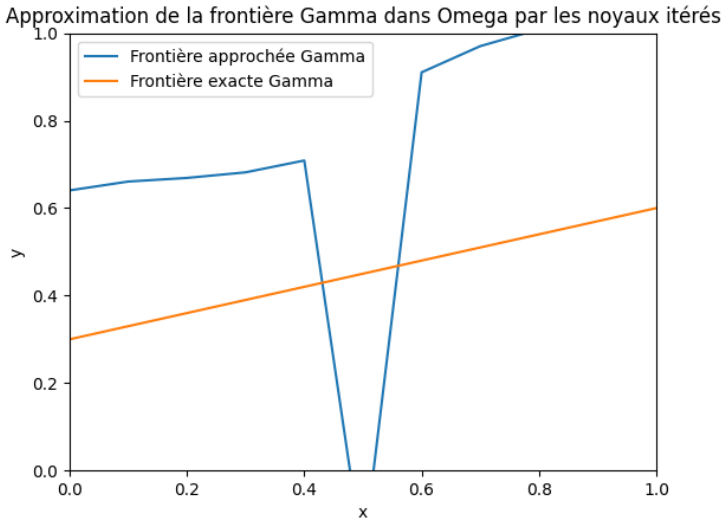


Pour la recherche de cette forme de frontière la méthode des noyaux itérés semble donner de meilleurs resultats même si les deux méthodes capturent le comportement global de la frontière exacte.

Regardons ce que la méthode de décomposition de Adomain donne pour une frontière d'équation $y = 03.x + 0.3$:

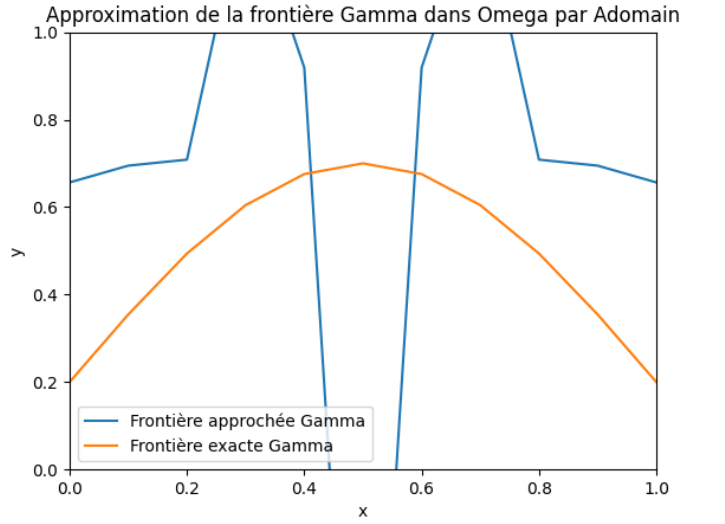


On a ici un soucis probablement causé par la mauvaise qualité de l'approximation car l'algorithme de Newton a été testé et est normalement performante.

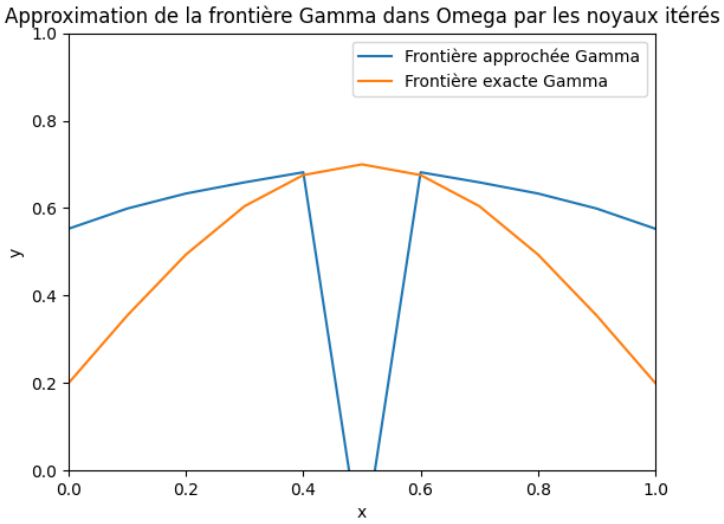


On a le même problème dans ce cas ci mais cette méthode semble capturer légèrement le comportement de la frontière pour des x assez petits.

Enfin maintenant on regarde ce que l'on obtient pour une frontière d'équation $y = 0.5sin(\pi x) + 0.2$ pour la méthode de décomposition de Adomain:



La méthode des noyaux itérés nous donne :



Encore une fois la méthode des noyaux itérés donne une meilleure approximation de la frontière Γ_3.

6 Conclusion

Pour conclure cette comparaison entre la méthode de décomposition de Adomain et la méthode des noyaux itérés, on remarque que la méthode des noyaux itérés donne de meilleures approximations de frontières de diverses formes. D'un point de vue technique les deux méthodes ont les mêmes difficultées d'implémentation, c'est-à-dire d'itéré les quadratures sur $u_n(x)$ pour obtenir une somme d'itérés ou bien la limite de la suite (u_n). Ainsi nous préconisons l'utilisation de la méthode des noyaux itérés.