Rapport : Introduction aux Équations Différentielles Partielles

Compte-rendu du projet d'étude

Programmation Orientée Objet - C++

MAM 4
Année 2025 - 2026

Polytech Nice Sophia

Rédigé par :
Elsa Catteau et Charlotte Prouzet

Sommaire

1 Introduction et objectifs

Les équations différentielles partielles jouent un rôle fondamental dans la modélisation de nombreux phénomènes physiques, tels que la diffusion de la chaleur, la propagation d'ondes ou encore le transport de polluants dans l'air et dans l'eau. Chaque EDP traduit une relation entre les variations temporelles et spatiales de quantités physiques et constitue un outil puissant pour analyser, comprendre et prédire des systèmes complexes. Dans le contexte environnemental actuel, elles peuvent également servir à représenter des mécanismes liés au stockage du dioxyde de carbone (CO2) dans les forêts, un processus essentiel pour l'atténuation du changement climatique.

L'un des modèles les plus simples et les plus étudiés est l'équation de convection-diffusion, qui combine deux phénomènes complémentaires :

— la diffusion, qui traduit la tendance d'une quantité à se disperser ;
— la convection, qui représente le transport dirigé par un flux externe.

L'objectif de ce projet d'étude est d'étudier cette équation à travers différents cas tests. En se basant sur le projet effectué l'année précédente en Python, nous avons cette fois-ci implémenté une méthode numérique basée sur les différences finies en langage C++. Cette approche nous a permis de simuler l'évolution de solutions analytiques connues et de comparer les résultats numériques avec les solutions exactes.

2 Equation de diffusion et de convection

Dans le cadre de ce projet, nous avons décidé de nous pencher sur l'équation de convection-diffusion unidimensionnelle suivante :

$$f(x,t) = \frac{\partial u}{\partial t} - D\frac{\partial^2 u}{\partial x^2} + C\frac{\partial u}{\partial x}, \tag{1}$$

où :

— $u(x,t)$ désigne la solution recherchée (par exemple la concentration de CO_2),
— D est le coefficient de diffusion,
— C est le coefficient de convection,
— $f(x,t)$ est un terme source calculé à partir de la solution exacte.

Le terme de diffusion $D\frac{\partial^2 u}{\partial x^2}$ modélise la dispersion de la quantité étudiée, tandis que le terme de convection $C\frac{\partial u}{\partial x}$ représente son transport. Les conditions aux limites et la condition initiale sont fixées à partir de la solution exacte connue, ce qui nous permettra de vérifier la précision de la solution numérique trouvée.

3 Approximations numériques

3.1 Méthode des différences finies

Afin de résoudre numériquement l'équation, nous avons utilisé un schéma aux différences finies explicite. Pour des maillages définis par un pas spatial Δx et un pas temporel Δt, la solution discrète $u_i^n \approx u(x_i, t^n)$ vérifie la relation :

$$u_i^{n+1} = u_i^n + \gamma\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) - \nu\left(u_i^n - u_{i-1}^n\right) + f(x_i, t^n)\,\Delta t, \tag{2}$$

avec

$$\gamma = \frac{D\,\Delta t}{\Delta x^2}, \quad \nu = \frac{C\,\Delta t}{\Delta x}. \tag{3}$$

Les coefficients γ et ν correspondent respectivement aux contributions diffusives et convectives. Leur choix est contraint par les conditions de stabilité :

$$\gamma \leq \frac{1}{2}, \quad \nu \leq 1. \tag{4}$$

3.2 Implémentation en C++

Contrairement à un projet Python pour lequel le code est écrit dans un seul script, l'implémentation en C++ requiert la séparation de notre code en trois fichiers principaux :

- TransportDiffusion.h qui permet la déclaration la classe
- TransportDiffusion.cpp qui permet l'implémentation des méthodes de la classe
- main.cpp, le programme principal qui permet de créer l'objet de la classe

3.2.1 TransportDiffusion.h

Nous commençons ce projet en créant un fichier header (.h) qui correspond à l'interface de la classe. Ce fichier nous permet de déclarer les variables et les méthodes publiques que l'utilisateur pourra par la suite utiliser.

Notre code est le suivant :

```
#ifndef TRANSPORTDIFFUSION_H
#define TRANSPORTDIFFUSION_H

#include <vector>
#include <string>

class TransportDiffusion {
private:
    double C, D, L;
    int Nx, Nt;
    double dx, dt, T;
    double gamma, v;
    std::vector<double> x;
    std::vector<double> t;
    std::vector<std::vector<double>> u_calcule;

public:
    // Constructeur et destructeur
    TransportDiffusion(double C_, double D_, double L_, int Nx_, int Nt_);
    ~TransportDiffusion();

    // Methodes
    double u_exacte(double xi, double ti);
    double f(double xi, double ti);
    void calculer();
    double erreur_L2();
    double erreur_Linf();
    void exporter_csv(const std::string& filename);
    void exportData(const std::string& filename);
};

#endif
```

Listing 1 – TransportDiffusion.h

3.2.2 TransportDiffusion.cpp

Une fois la déclaration de notre classe et de nos méthodes faîtes, nous avons procéder à l'implémentation de ces dernières via le fichier TransportDiffusion.cpp.

Les directives

Dans un premier temps nous incluons le contenu du fichier TransportDiffusion.h dans notre fichier source, puis nous rajoutons les fonctions standard :

```cpp
#include "TransportDiffusion.h"
#include <cmath>
#include <iostream>
#include <fstream>
#include <algorithm>
```

Listing 2 – Déclaration des directives

Le constructeur

Ensuite nous implémentons le constructeur de la classe "TransportDiffusion" en lui passant en argument les paramètres C, D, L, Nx et Nt. Ce constructeur nous permet de créer un objet TransportDiffusion et d'initialiser ses variables membres principales avec les valeurs fournies par l'utilisateur :

```cpp
TransportDiffusion::TransportDiffusion(double C_, double D_, double L_, int Nx_,
    int Nt_)
    : C(C_), D(D_), L(L_), Nx(Nx_), Nt(Nt_)
```

Listing 3 – Initialisation du constructeur

Nous avons ensuite définit les paramètres dérivés nécessaires à notre simulation :

```cpp
// Pas spatial
dx = L / Nx;

// Coefficient pour la diffusion
gamma = 0.25;

// Coefficient pour le transport
v = 0.5;

// Pas de temps limite par la diffusion
double dt1 = gamma * dx * dx / D;

// Pas de temps limite par le transport
double dt2 = v * dx / std::abs(C);

// On prend le plus petit pour garantir la stabilite
dt = std::min(dt1, dt2);

// Temps total de la simulation
T = Nt * dt;
```

Listing 4 – Calcul des paramètres dérivés

Nous procédons alors à l'initialisation des vecteurs `x` et `t`, qui correspondent respectivement aux vecteurs des positions et des instants de temps. De même, nous initialisons `u_calcule`, qui est une matrice de taille (N_x, N_t) et dans laquelle nous stockerons la solution numérique :

```cpp
x.resize(Nx);
t.resize(Nt);
u_calcule.resize(Nx, std::vector<double>(Nt, 0.0));
```

Listing 5 – Initialisation des vecteurs et de la matrice

Enfin, nous remplissons les vecteurs x et t avec des valeurs uniformes. De plus, on initialise la solution au temps t=0 avec la solution exacte :

```cpp
// Initialisation des vecteurs x et t
for (int i = 0; i < Nx; ++i)
    x[i] = i * dx;
```

```
4
5  for (int n = 0; n < Nt; ++n)
6      t[n] = n * dt;
7
8  // Initialisation de la premiere colonne de u_calcule avec la solution exacte
9  for (int i = 0; i < Nx; ++i)
10     u_calcule[i][0] = u_exacte(x[i], 0);
```

Listing 6 – Remplissage des vecteurs et conditions initiales

Le destructeur

Une fois le constructeur implémenté, nous avons ajouté un destructeur à la classe TransportDiffusion. Bien que vide, il garantit la gestion de la fin de vie d'un objet et contribue donc à l'encapsulation de a classe :

```
1  TransportDiffusion::~TransportDiffusion() {}
```

Listing 7 – Implémentation du destructeur

Calcul numérique

La méthode `calculer()` applique quant à elle un schéma numérique explicite afin de résoudre notre équation de transport-diffusion. Au sein de cette méthode, on peut noter que la solution `u_calcule` est régulièrement mise à jour.
Dans le cas test n°2, on impose 0 aux frontières $i = 0$ et $i = N_x - 1$. Cela correspond aux conditions limites de nos bords, qui changent en fonction des cas tests (cf. partie 4 sur les cas tests) :

```
1  void TransportDiffusion::calculer() {
2      for (int n = 0; n < Nt-1; ++n) {
3          for (int i = 1; i < Nx-1; ++i) {
4              double du = (C > 0) ? (u_calcule[i][n] - u_calcule[i-1][n])
5                                  : (u_calcule[i+1][n] - u_calcule[i][n]);
6              u_calcule[i][n+1] = u_calcule[i][n]
7                                + gamma * (u_calcule[i+1][n] - 2*u_calcule[i][n] +
                                     u_calcule[i-1][n])
8                                - v * du
9                                + f(x[i], t[n]) * dt;
10         }
11         u_calcule[0][n+1] = 0.0;
12         u_calcule[Nx-1][n+1] = 0.0;
13     }
14 }
```

Listing 8 – Implémentation de la méthode calculer()

Calcul des erreurs

À la fin de notre simulation, afin de mesurer à quel point la solution numérique obtenue `u_calcule` se rapproche de la solution exacte `u_exacte`, nous introduisons deux indicateurs d'erreur :

— Erreur L2 : Elle calcule la différence entre la solution numérique et la solution exacte sur tout le domaine, puis en fait une moyenne "quadratique". Cela nous donne alors une mesure globale sur la précision de la simulation :

```
1          double TransportDiffusion::erreur_L2() {
2      double sum = 0.0;
3      for (int i = 0; i < Nx; ++i) {
4          double e = std::abs(u_calcule[i][Nt-1] - u_exacte(x[i], T));
5          sum += e * e;
6      }
7      return std::sqrt(dx * sum);
8  }
```

Listing 9 – Erreur L2

— Erreur Linf : Elle correspond à la plus grande erreur ponctuelle entre les deux solutions. ECela nous permet de détecter les écarts les plus important :

```
        double TransportDiffusion::erreur_Linf() {
    double max_err = 0.0;
    for (int i = 0; i < Nx; ++i) {
        double e = std::abs(u_calcule[i][Nt-1] - u_exacte(x[i], T));
        if (e > max_err) max_err = e;
    }
    return max_err;
}
```

Listing 10 – Erreur Linf

Ces deux mesures nous permettent ainsi de juger si le schéma numérique est fiable et suffisamment précis (ou non). En effet, L2 évalue la qualité globale de la solution tandis que Linf met en évidence la pire erreur locale.

3.2.3 main.cpp

Dans le fichier main.cpp, on définit nos différentes constatntes (C, D, L Nx, Nt) et on crée un objet de type TransportDiffusion qu'on a appelé td. Ensuite, on appelle la méthode caluler qui réalise tous les calculs numériques pour résoudre l'équation de convection-diffusion avec les paramètres donnés. Enfin, on affiche les valeurs de l'erreur L^2 et l'erreur L^∞.

```
#include "TransportDiffusion.h"
#include <iostream>

int main() {
    double C = 1.0, D = 1.0, L = 1.0;
    int Nx = 10000, Nt = 50;
    TransportDiffusion td(C, D, L, Nx, Nt);
    td.calculer();
    std::cout << "Erreur L2 : " << td.erreur_L2() << std::endl;
    std::cout << "Erreur Linf : " << td.erreur_Linf() << std::endl;
    return 0;
}
```

Listing 11 – main.cpp

4 Cas Tests

Nous avons décidé d'effectuer 3 cas tests.

4.1 Cas test 1

La solution analytique considérée pour ce premier cas test est

$$u(x,t) = \cos(\pi x)(1+t).$$

Le domaine spatial est $\partial\Omega = \{0,1\}$. La condition initiale, pour $t = 0$, est donnée par $u_i^0 = \cos(\pi x)$. Les conditions aux limites sont définies par $u_L^n = 1 + t$ à gauche et $u_R^n = -(1+t)$ à droite.
Le terme source associé à cette solution est

$$f(x,t) = \cos(\pi x) + D\,(\pi^2 \cos(\pi x)(1+t)) - C\,(\pi \sin(\pi x)(1+t)).$$

Déclaration de la solution exacte et du terme source :

```cpp
#include <cstdio>
#include <vector>

// Solution exacte
double TransportDiffusion::u_exacte(double x, double t) {
    return std::cos(M_PI*x)*(1 + t);
}

// Terme source
double TransportDiffusion::f(double x, double t) {
    return std::cos(M_PI*x)+D*(M_PI*M_PI)*std::cos(M_PI*x)*(1+t)-C*M_PI*std::sin
        (M_PI*x)*(1+t);
}
```

La suite du code est correspond au fichier TransportDiffusion.cpp, il y a juste une modification a apporté dans la fonction calculer, il faut mettre les conditions limites de notre cas test soit :

```cpp
u_calcule[0][n+1] = 1.0 + t[n+1];
u_calcule[Nx-1][n+1] = -(1.0 + t[n+1]);
```

En exécutant notre programme, on obtient les erreurs suivantes : l'erreur L^2 vaut 0.00544224 et l'erreur L^∞ vaut 0.00769649.

Ensuite, nous avons souhaité comparer la solution exacte avec celle calculée. Pour cela, il était nécessaire de tracer un graphe. Nous nous sommes alors documentés pour savoir comment afficher un graphe à partir du langage C++. Nous avons installé `gnuplot` et ajouté dans nos programmes `Cas_test_1.cpp` et `main.cpp` des instructions permettant de générer un fichier `.dat`, qui sera ensuite lu par `gnuplot`. Voici ce que nous avons ajouté dans le fichier `Cas_test_1.cpp` :

```cpp
void TransportDiffusion::exportData(const std::string& filename) {
    std::ofstream file(filename);
    for (int i = 0; i < Nx; ++i) {
        file << x[i] << " " << u_exacte(x[i], T) << " " << u_calcule[i][Nt-1] <<
            "\n";
    }
    file.close();
}
```

La méthode exportData est également a déclarer dans le fichier TransportDiffusion.h.
Voici ce qu'il faut rajouter dans le main.cpp :

```cpp
td.exportData("C:/Users/Data.dat");
```

Enfin, après avoir ouvert gnuplot voici les instructions à mettre :

```
set title "Comparaison solution exacte / calculee"
set xlabel "x"
set ylabel "u(x, T)"
set grid
plot "C:/Users/Data.dat" using 1:2 with lines lw 2 lc rgb "blue" title "exacte",
     \"C:/Data.dat" using 1:3 with lines lw 2 lc rgb "red" title "calculee"
```

Voici le graphe obtenu :

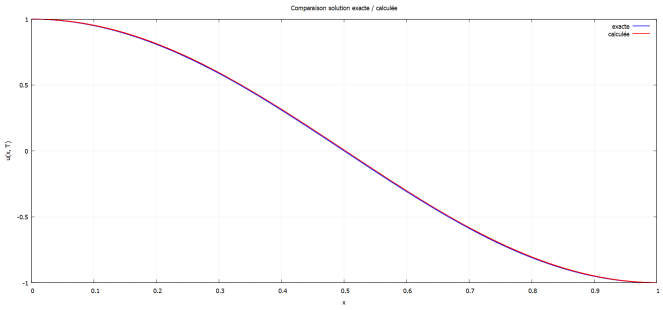

FIGURE 1 – Graphe Cas Test 1

4.2 Cas Test 2

La solution analytique considérée pour le deuxième cas test est

$$u(x,t) = \sin(\pi x)\, \mathrm{e}^{-t}.$$

Le domaine spatial est $\partial\Omega = \{0,1\}$. La condition initiale pour $t = 0$ est $u_i^0 = \sin(\pi x)$, et les conditions aux limites sont nulles, c'est-à-dire $u_L^n = 0$ à gauche et $u_R^n = 0$ à droite.
Le terme source associé à cette solution est

$$f(x,t) = -\cos(\pi x)\, \mathrm{e}^{-t} + D\pi^2 \sin(\pi x)\, \mathrm{e}^{-t} + C\pi \cos(\pi x)\, \mathrm{e}^{-t}.$$

Déclaration de la solution exacte et du terme source :

```
#include <cstdio>
#include <vector>

// Solution exacte
double TransportDiffusion::u_exacte(double x, double t) {
    return std::sin(M_PI*x)*std::exp(-t);
}

// Terme source
double TransportDiffusion::f(double x, double t) {
    return -std::sin(M_PI*x)*std::exp(-t)+D*(M_PI*M_PI)*std::sin(M_PI*x)*std::
        exp(-t)
    +C*M_PI*std::cos(M_PI*x)*std::exp(-t);
}
```

Les conditions limites :

```
u_calcule[0][n+1] = 0;
u_calcule[Nx-1][n+1] = 0;
```

En exécutant notre programme, on obtient les erreurs suivantes : l'erreur L^2 vaut 0.00543136 et l'erreur L^∞ vaut 0.00769624. De la même manière que pour le premier cas test, on affiche le graphe qui compare la solution exacte et celle calculée à l'aide de gnuplot. On obtient :

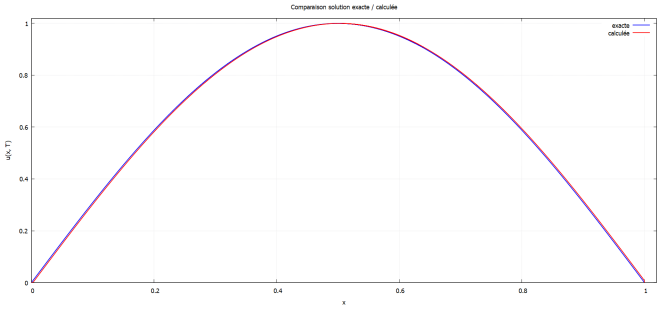

FIGURE 2 – Graphe Cas Test 2

4.3 Cas Test 3

La solution analytique considérée pour le troisième cas test est

$$u(x,t) = \big(\sin(\pi x) + \cos(2\pi x)\big)\, \mathrm{e}^{-\pi^2 t}.$$

Le domaine spatial est $\partial\Omega = \{0, 1\}$. La condition initiale pour $t = 0$ est $u_i^0 = \sin(\pi x) + \cos(2\pi x)$, et les conditions aux limites sont $u_L^n = \mathrm{e}^{-\pi^2 t}$ et $u_R^n = \mathrm{e}^{-\pi^2 t}$.
Le terme source associé à cette solution est

$$f(x,t) = -\pi^2\big(\sin(\pi x)+\cos(2\pi x)\big)\,\mathrm{e}^{-\pi^2 t} + D\pi^2\big(\sin(\pi x)+4\cos(2\pi x)\big)\,\mathrm{e}^{-\pi^2 t} + C\big(\pi\cos(\pi x) - 2\pi\sin(2\pi x)\big)\,\mathrm{e}^{-\pi^2 t}.$$

Déclaration de la solution exacte et du terme source :

```cpp
#include <cstdio>
#include <vector>

// Solution exacte
double TransportDiffusion::u_exacte(double x, double t) {
    return (std::sin(M_PI * x) + std::cos(2*M_PI*x)) * std::exp(-(M_PI*M_PI)*t);
}

// Terme source
double TransportDiffusion::f(double x, double t) {
    return -(M_PI*M_PI)*(std::sin(M_PI*x)+std::cos(2*M_PI*x))*std::exp(-(M_PI*
        M_PI)*t)+
    D*(M_PI*M_PI)*(std::sin(M_PI*x)+4*std::cos(2*M_PI*x))*std::exp(-(M_PI*M_PI)*
        t) +
    C*((M_PI * std::cos(M_PI * x) -2*M_PI*std::sin(2*M_PI*x))*std::exp(-(M_PI*
        M_PI)*t));
}
```

Les conditions limites :

```cpp
u_calcule[0][n+1]=std::exp(-(M_PI*M_PI) * t[n+1]);
u_calcule[Nx-1][n+1]=std::exp(-(M_PI*M_PI) * t[n+1]);
```

En exécutant notre programme, on obtient les erreurs suivantes : l'erreur L^2 vaut 0.00688547 et l'erreur L^∞ vaut 0.0102145. De la même manière que pour le premier cas test, on affiche le graphe qui compare la solution exacte et celle calculée à l'aide de gnuplot. On obtient :

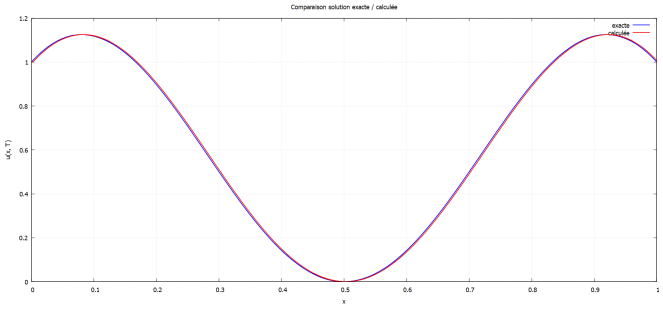

FIGURE 3 – Graphe Cas Test 3

5 Tests unitaires

Nous avons fait des tests unitaires pour vérifier que les principales fonctions de notre fichier TransportDiffusion.cpp se comportent correctement. Pour cela, nous avons créé une fonction runAllTests() qui regroupe nos différents tests unitaires. Nous avons alors créés trois blocs : le premier effectue un test sur la solution exacte, le second vérifie le terme source et enfin le dernier teste la partie résolution numérique avec notre fonction calculer() et nos deux fonctions d'erreur (L2 et Linf).

```cpp
void runAllTests() {
    // Test 1: Solution exacte
    {
        TransportDiffusion td(1.0, 1.0, 1.0, 100, 100);
        assert(std::abs(td.u_exacte(0.0, 0.0)) < 1e-12);
        assert(std::abs(td.u_exacte(1.0, 0.0)) < 1e-12);
        assert(std::abs(td.u_exacte(0.5, 0.0) - std::sin(M_PI*0.5)) < 1e-12);
    }
    std::cout << "Test de la solution exacte reussi" << std::endl;

    // Test 2: Terme source
    {
        TransportDiffusion td(1.0, 1.0, 1.0, 100, 100);
        double f_val = td.f(0.5, 1.0);
        assert(std::isfinite(f_val));
    }
    std::cout << "Test du terme source reussi" << std::endl;

    // Test 3: Calcul
    {
        TransportDiffusion td(1.0, 1.0, 1.0, 100, 100);
        td.calculer();
        assert(std::isfinite(td.erreur_L2()));
        assert(std::isfinite(td.erreur_Linf()));
    }
    std::cout << "Tests calcul et erreurs reussis" << std::endl;
}
```

6 Conclusion

Dans ce travail, nous avons mis en œuvre le principe d'encapsulation en développant des classes en C++. La gestion des objets a été assurée par l'utilisation de constructeurs et de destructeurs. Un schéma numérique explicite a ensuite été implémenté pour résoudre le problème étudié. Les résultats obtenus ont été comparés aux solutions exactes afin d'évaluer la précision de la méthode, et les erreurs ont été calculées en utilisant les normes L^2 et L^∞.