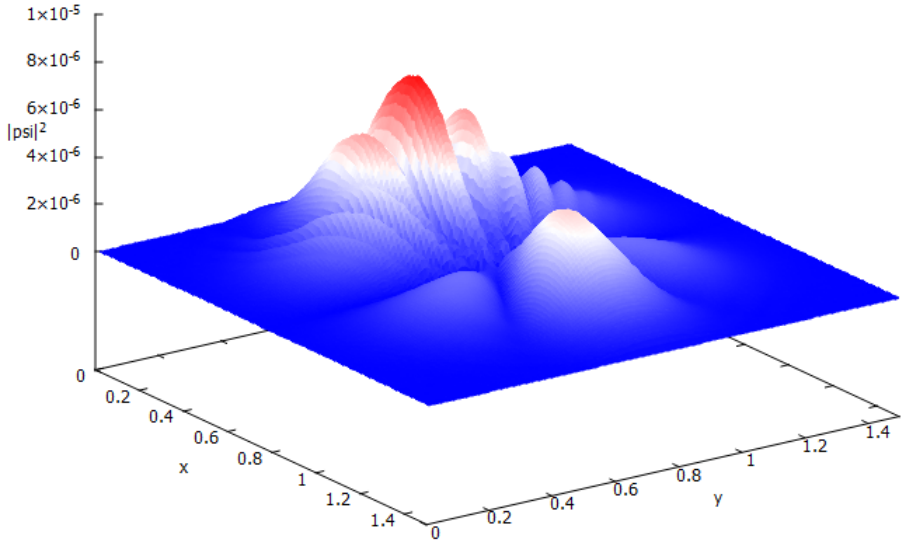
SORBONNE UNIVERSITÉ

Résolution de l'équation de Schrödinger dépendante du temps

EVRARD Antonin

Résumé

Nous résolvons numériquement l'équation de Schrödinger dépendante du temps en langage Fortran avec l'algorithme Runge-Kutta d'ordre 4. Dans un premier temps nous retrouverons les prédictions de la théorie pour des cas analytiquement solubles pour nous assurer autant du bon fonctionnement que de la stabilité du programme puis nous étudierons des cas de diffusions à 2 dimensions.



Sommaire

1 Introduction

L'équation de Schrödinger, conçue en 1925 par Erwin Schrödinger, est un des postulats de la physique quantique qui permet d'étudier l'évolution temporelle d'un système à partir de la connaissance de son Hamiltonien :

$$i\hbar\frac{d}{dt}\left|\Psi(\vec{r},t)\right\rangle = \mathcal{H}(t)\left|\Psi(t)\right\rangle$$

Sa condition de postulat et son utilité la rend omniprésente dans les sciences fondamentales, c'est pourquoi sa résolution représente un centre d'intérêt particulier autant pour les étudiants que pour la communauté scientifique avec au moins 223 articles contenant les mots *Schrödinger*, *equation* et *numerical* ayant été publiés au cours des 12 derniers mois sur Arxiv. Dans le cas d'une particule libre en interraction avec un potentiel $V(\vec{r})$ qui ne dépend pas du temps, l'équation s'écrit :

$$i\hbar\frac{d}{dt}\left|\Psi(\vec{r},t)\right\rangle = \left[-\frac{\hbar^2}{2m}\nabla^2 + V(\vec{r})\right]\left|\Psi(\vec{r},t)\right\rangle$$

Dans toute notre étude nous adopterons le système des unités atomiques, nous aurons donc $m = \hbar = 1$ ce qui permet à la fois d'alléger les notations et le codes mais aussi de mettre à une échelle pertinente l'ensemble des résultats de nos simulations. Nous aurons ainsi :

- Unité de masse : $m_e \sim 9.1 \times 10^{-31}$kg la masse de l'électron
- Unité de longueur : $a_0 = \frac{\hbar}{m_e c\alpha} \sim 0.53$ Å le rayon de Bohr
- Unité d'énergie : $E_H = m_e c^2\alpha^2 \sim 27.2$ eV le hartree
- Unité de temps : $t_0 = \frac{\hbar}{E_H} \sim 2.4 \times 10^{-17}$ s

Nous allons dans la première partie revenir sur les différents choix qui ont été faits au cours de l'élaboration du programme (choix de langage, algorithme, système à étudier ...), dans la deuxième partie nous nous assurerons que les résultats de nos simulations soient cohérents avec les prédictions de la théorie pour quelques cas analytiquement solubles, ensuite nous nous pencherons sur des cas particuliers de diffusions à 2 dimensions.

2 Outils numériques

2.1 Langage informatique

Pour faire suite à l'enseignement de physique numérique MU4PYD12, nous avons décidé d'utiliser le langage Fortran. Nous aurions pu aussi choisir Python, le C/C++, Scilab ou encore Matlab et il aurait été intéressant de faire une comparaison de performance entre ces différents langages de programmation. Le choix n'est cependant pas complètement libre puisque notre projet nous demandera de faire des simulations à haute résolution spatiale et temporelle, le choix d'un langage plus "proche" de la machine et capable de *High Performance Computing* tout en restant flexible dans son l'écriture et dans sa difficulté de programmation, tel Fortran, semble donc plus judicieux.

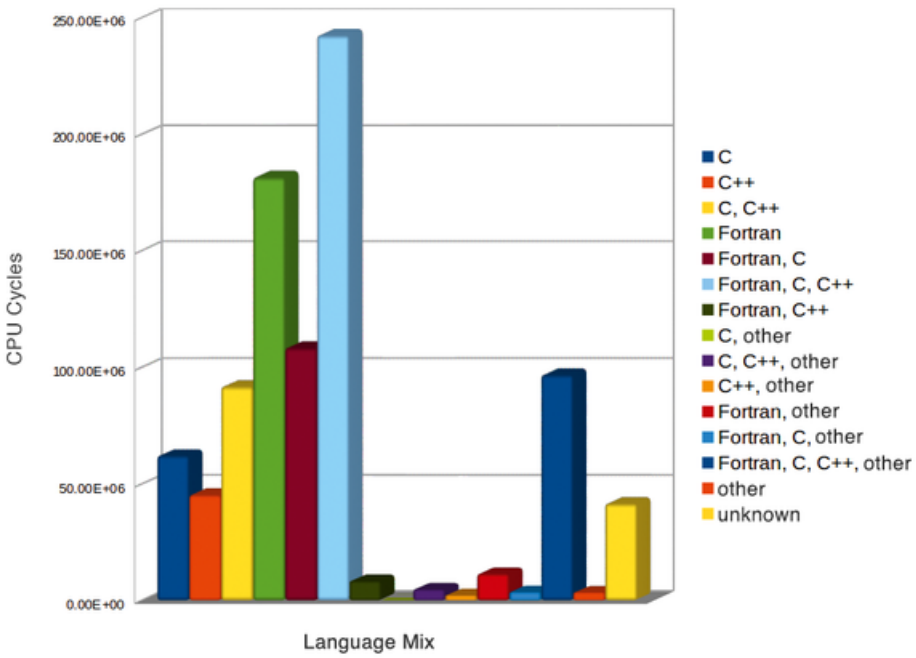


Figure 1: Parts des langages de programmation utilisés dans les projets actifs au SuperMUC en 2017[1]

À noter que nous avançons l'argument de la performance de calcul de Fortran pour justifier le choix du langage de programmation mais il est possible que ce critère de perfomance n'ait qu'un impact secondaire dans nos simulations puisque dans un soucis de temps d'exécution, nous nous limiterons à des petites échelles et à la dimension 2. Cepdendant si nous souhaitions élargir la taille de l'espace dans lequel le système évolue ou passer à la troisième dimension alors ce facteur rentrerait en compte de manière non négligeable.

2.2 Alrgorithme de Runge-Kutta d'ordre 4

Le problème que l'on propose de résoudre est une équation différentielle d'ordre 2 et dépendante du temps. Il n'existe pas, à part pour certains cas particuliers bien définis, de solution générale à cette équation, c'est pourquoi nous utilisons un outil numérique qui, grâce à certaines approximations, nous permet d'approcher une solution pour des systèmes quelconques. Il existe de nombreux algorithmes qui permettent d'approcher des solutions et, comme pour le choix du langage de programmtion, le choix de l'algorithme est soumis à des critères de précisions et de rapidités. On peut citer :

- Euler
- Crank-Nickolson
- Runge-Kutta 2
- Runge-Kutta 4

Les spécificités de chacuns ont déjà été explicité dans le cours de Physique Numérique [2], nous nous passerons d'explication. Nous avons choisi d'utiliser l'algorithme de Runge-Kutta d'ordre 4 (RK4) pour plusieurs raisons. Premièrement pour une raison de facilité, nous avons en effet manipulé RK4 plusieurs fois au cours de notre formation et son caractère vectoriel permet une modification simple pour passer à dimension 2. Deuxièmement pour une question de rapidité, pour des tâches simples il semblerait que RK4 soit le plus rapide des quatres.[3].

Method	Time step	Calculation duration
Explicit Euler	$\Delta t = 10^{-6}$	83 s
TR-BDF2	$\Delta t = 10^{-4}$	45 s
Crank-Nicholson	$\Delta t = 10^{-4}$	22 s
Runge Kutta 2	$\Delta t = 10^{-5}$	6.5 s
Runge Kutta 4	$\Delta t = 10^{-4}$	1.3 s

Figure 2: Temps de calcul de différents algorithmes sur un calcul de propagation unidimensionnel d'un paquet d'onde gaussien.

2.3 Méthode des différences finies

L'utilisation de l'algorithme RK4 implique une discrétisation de l'espace et du temps de notre problème et un choix à faire pour traiter le laplacien de l'équation de Schrödinger. La méthode des différences finies permet d'approximer le laplacien d'une fonction :

$$\nabla^2 f(x) \simeq \frac{f(x+h) - 2f(x) + f(x-h)}{\Delta x^2}$$

Ce qui se traduit pour notre cas :

$$\frac{\psi(x,t+\Delta t)-\psi(x,t)}{\Delta t}=\frac{\psi(x+\Delta x,t)-2\psi(x,t)+\psi(x-\Delta x,t)}{\Delta x^2}$$

Dans le code cette estimation est intégralement traitée par la subroutine :

```
! Finite difference method
subroutine deriv_2_esti(x,w)
  use parametres
  implicit none
  complex(4) , dimension(disc) , intent(in)    :: x
  complex(4) , dimension(disc) , intent(inout) :: w
  integer                                      :: i
  w(1)=0 ; w(disc)=0      ! strict boundary conditions
  do i=2,disc-1,1         ! spatial loop
      w(i)=(x(i+1)-2*x(i)+x(i-1))/(xstep**2)
  enddo
end subroutine deriv_2_esti
```

À noter que cette méthode n'est pas sans contrainte, en effet sa condition de convergeance est :

$$\frac{\Delta t}{\Delta x^2}\leq\frac{1}{2}$$

À partir de maintenant notre discrétisation de l'espace et du temps seront donc correlées.

2.4 Condtions initiales et aux bords

Nous avons choisi d'initialiser notre système comme un paquet d'onde gaussien pour permettre une analogie avec une particule libre et parce que son comportement à travers l'équation de Schrödinger est connu. Nous avons ainsi à 1 dimension :

$$\psi(x,t=0)=A\exp\left(-\frac{(x-x_0)^2}{2\sigma^2}\right)\exp(ikx)$$

Où x_0 est sa position initiale, σ son écart quadratique moyen ou "étalement", k son vecteur d'onde et A le facteur de normalisation de sorte que :

$$\int_0^L \psi^*\psi dx=1$$

Nous avons fixé l'état initial, la propagation et l'estimation de la dérivée seconde de notre système, il ne nous reste plus qu'à fixer les conditions aux bords avant de lancer les premières simulations. Comme il est commenté dans le code de la subroutine de la méthode des

différences finies, nous avons choisi des conditions aux bords fixes, de Neumann, non seulement sur la position mais aussi sur les dérivés secondes (et donc implicitement les dérivées premières) :

$$\begin{cases} \psi(x=0,t)=0 \\ \psi(x=L,t)=0 \end{cases} \quad \text{et} \quad \begin{cases} \psi''(x=0,t)=0 \\ \psi''(x=L,t)=0 \end{cases}$$

Où L est la longueur du système dans lequel le paquet d'onde est confiné. Tout se passe donc comme si notre système évoluait librement dans une boite où il interagissait avec un potentiel $V(\vec{r})$ ou $V(x)$ à 1 dimension, mais dont les mouvements sont limités par un potentiel infini au bords de l'espace. Ce choix permet une simplicité de programmation mais force notre système à évoluer dans un espace suffisemment "grand" pour que le comportement aux bords n'impact pas les résultats de la simulation. Comme on dit souvent en physique "notre système est loin des bords et ne les voit pas".

3 Vérification du programme

3.1 Libre propagation

Le système le plus simple que l'on puisse simuler pour tester le bon fonctionnement du programme est la propagation libre du paquet d'onde gaussien avec potentiel nul $V(x)=0$:

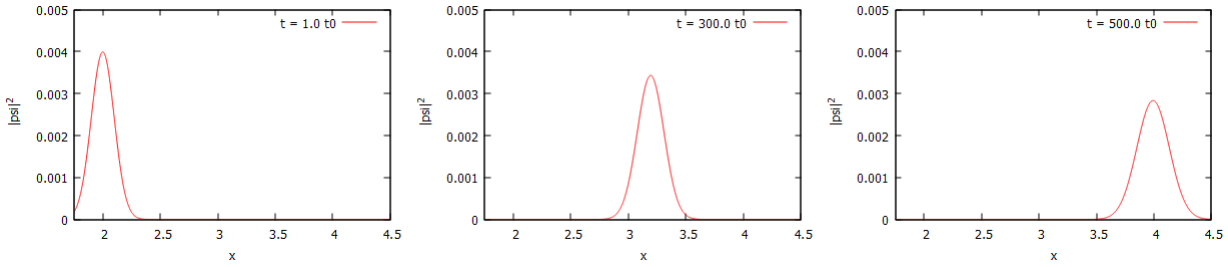


Figure 3: Libre propagation d'un paquet d'onde gaussien à 1 dimension .

On observe conformément aux prédictions de la théorie que le paquet d'onde "s'applatie" au cours de sa propagation perdant en amplitude maximale et gagnant en largeur. En effet en représentation impulsion notre système est également décrit par une gaussienne et les composantes d'impulsions supérieurs et inférieures à p_0 se déplacent respectivement plus vite et moins vite que les autres, amenant à un étalement du système.
À noter quand dans le système des unités atomiques l'impulsion $k = \frac{p}{\hbar} \Leftrightarrow p = k$ où k est le vecteur d'onde du système. De plus le paquet d'onde évolue sur une longueur $L = 5$ allant de $x = 0$ jusque $x = 5$ donc le potentiel infini des bords n'interagit pas ici.

3.2 Potentiel carré

La suite logique est de simuler l'interaction du paquet d'onde avec un potentiel carré défini comme :

$$V(x) : \begin{cases} \forall x \in [x_{p_0} - L_p/2, x_{p_0} + L_p/2] : V(x) = V_0 \\ \forall x \notin [x_{p}0 - L_p/2, x_{p_0} + L_p/2] : V(x) = 0 \end{cases}$$

Où x_{p_0} est la position du centre du potentiel, L_p la largeur et V_0 la hauteur du potentiel.

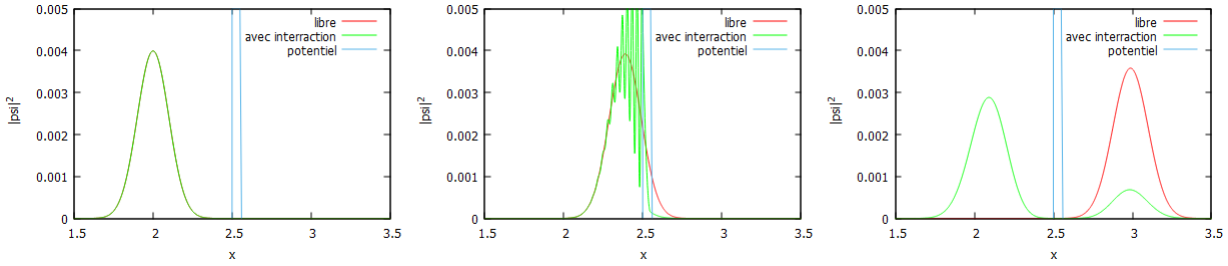


Figure 4: Comparaison entre libre propagation et interaction avec un potentiel carré.

On peut observer des interférences entre l'onde incidente et l'onde réfléchie par la barrière de potentiel ainsi qu'un dédoublement du paquet d'onde en une onde transmise et une onde réfléchie, toutes deux adoptant un comportant gaussien. Ces résultats sont satisfaisants au moins d'un point de vue qualitatif. Malheureusement il ne suffit pas de résultats qualitaviement corrects si l'on veut pouvoir utiliser le programme de manière prédictive, il faut aussi s'assurer que le caractère quantitatif des résultats adhère également à la théorie. Pour ce faire l'observable la plus facilement accessible d'un point de vue de la programmation et de l'analyse théorique est le coefficient de transmission. Nous ferons donc appel à une fonction pour calculer la norme d'un vecteur de dimension quelconque :

```
real function norm(x)
  implicit none
  complex  :: x(:)
  real :: sum
  integer :: i , N0
  sum=0
  N0=size(x)
  do i=1,N0,1
    sum=sum+cabs(x(i))**2
  enddo
  norm=sqrt(sum)
end function norm
```

Et une autre fonction pour calculer le rapport des normes de part et d'autre de la barrière de potentiel :

```
real function Tcoef(x)
  implicit none
  complex :: x(:)
  integer :: i
  i=int(potential_position/xstep+potential_length)
  Tcoef= norm(x(i:disc))/norm(x(:))
end function Tcoef
```

À noter que ces deux fonctions sont contenues dans un module de définition de fonction qui lui même utilise le module de paramètres, donc “potential position”, “xstep”, “potential length” et “disc” sont définis pour ces fonctions. Nous allons nous intéresser à deux cas concrets de la barrière de potentiel finie, nous allons suivre les valeurs du coefficient de transmission en fonction de la largeur de la barrière dans le cas où $E > V_0$ et où $E < V_0$:

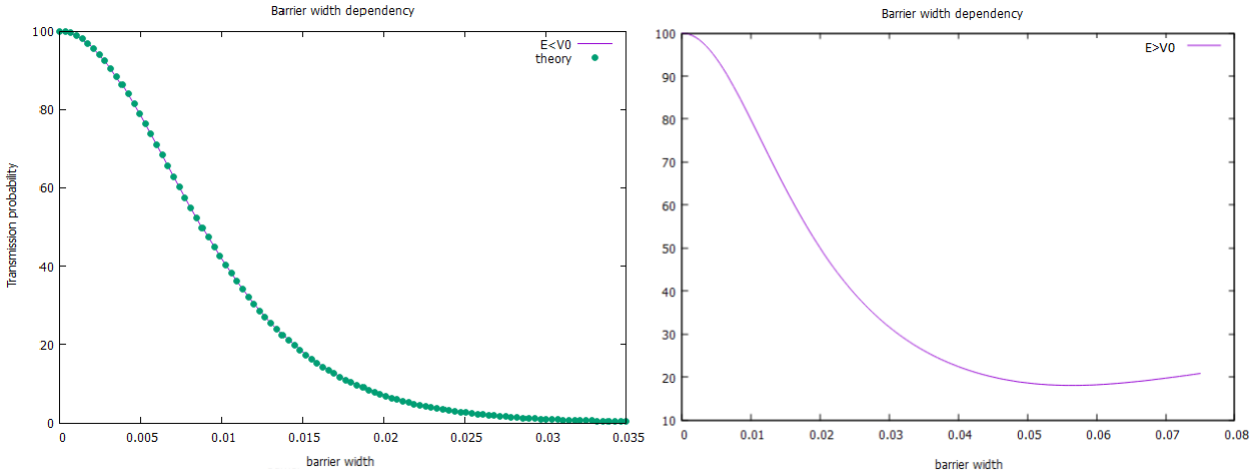


Figure 5: Probabilité de transmission en fonction de la largeur de la barrière. (à gauche $E < V_0$, à droite $E > V_0$)

On observe déjà que les résultats sont au moins qualitativement cohérents, la probabilité de transmission décroit en fonction de la largeur de la barrière. Pour le cas $E < V_0$ la probabilité tend, comme attendu, vers 0 . On a tracé les valeurs prédites par la théorie[4] par :

$$T(a) = \frac{4E(V_0 - E)}{4E(V_0 - E) + V_0^2 \sinh^2(\sqrt{2m(V_0 - E)}a/\hbar)}$$

Pour le cas $E < V_0$, nous avions $V_0 = 2E$ et $E = p^2/2m = k^2/2$ avec $k = 100$. On observe donc un accord parfait entre la simulation et la théorie. Pour le cas $E > V_0$ avons $k = 60$ et on peut observer un phénomène de résonance, pour des valeurs multiple de $\frac{\pi}{k}$, nous n'avons pas simulé plus de points, faute de temps, pour confirmer l'allure du graphique attendu[4] par :

$$T(a) = \frac{4E(E - V_0)}{4E(E - V_0) + V_0^2 \sin^2(\sqrt{2m(V_0 - E)}a/\hbar)}$$

Nous n'avons malheureusement pas conservé les données ni les paramètres autres que $k = 60$ pour pouvoir superposer sur le même graphique la théorie et les résultats, cependant on peut observer la première résonance pour $x = \frac{\pi}{60} \sim 0.05$.

3.3 Conservation de la norme

Pour finaliser cette partie de vérification de la précision du programme on peut aussi se pencher sur la conservation de la norme de notre système au cours du temps. On fait pour cela également appel à la fonction *norm(x)* :

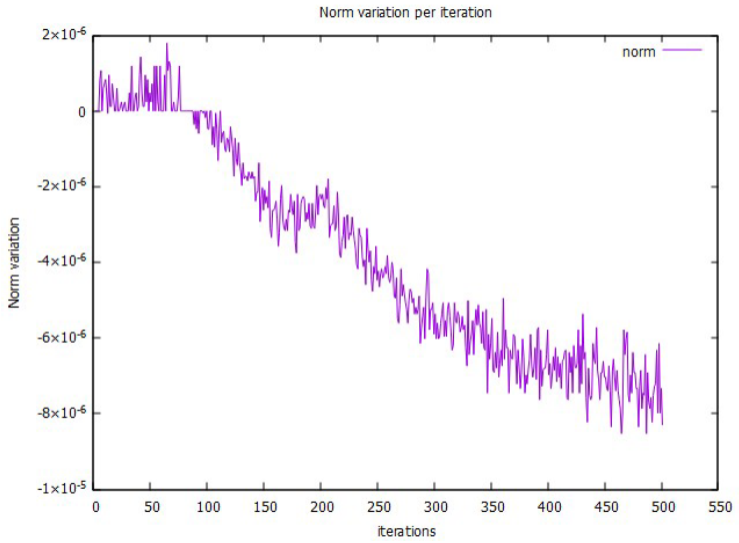


Figure 6: Variation de la norme au cours du temps (chaque itération équivaut à une variation Δt du temps)

Évidemment la norme n'est pas conservée comme attendu par n'importe quelle méthode d'approximation numérique cependant l'ordre de grandeur des variations, lui, est négligeable. À noter que ce sont les interactions avec le potentiel qui génèrent le plus de perte de norme.

On a donc fixer $\Delta x \sim 0.001$ et $\Delta t \sim 10^{-7}$ pour le reste de l'étude.

Maintenant que l'on s'est assuré de la solidité du programme et de sa cohérence avec la théorie sur des cas simples à une dimension, on peut pousser l'étude au-delà de ce qui est analytiquement soluble. Nous avons fait le choix d'étudier la propagation du paquet d'onde à deux dimensions.

4 Propagation à 2 dimensions

4.1 Libre propagation

Le passage à deux dimensions ne nécessitait que peu de changements dans le code :

```
  subroutine rk4(t,x,dt2,deriv)
    ! 4th order Runge-Kutta. See Numerical Recipes p. 701ff
    use parametres
    implicit none
    real                , intent(in)                  :: t, dt2
    complex(4), dimension(disc,disc), intent(inout) :: x
    real                                              :: ddt
    complex, dimension(disc,disc):: xp, k1, k2, k3, k4
    external :: deriv
    ddt = 0.5*dt2
    call deriv(t,x,k1)      ; xp = x + ddt*k1
    call deriv(+ddt,xp,k2) ; xp = x + ddt*k2
    call deriv(t+ddt,xp,k3); xp = x +  dt*k3
    call deriv(t+dt,xp,k4) ; x = x+dt*(k1+2.0*k2+2.0*k3+k4)/6.0
  end subroutine rk4
```

La subroutine qui calcule le terme laplacien :

```
  ! Second spatial derivative estimation
  subroutine deriv_2_esti(psi,w,a)
    use parametres
    implicit none
    complex(4), dimension(disc,disc), intent(in)     :: psi
    complex(4), dimension(disc,disc), intent(inout)  :: w
    integer                                          :: i,a
    w=0     !boundary condition
    if (a==1) then      ! derivative with respect to x
        do i=2,disc-1,1
            w(i,:)=(psi(i+1,:)-2*psi(i,:)+psi(i-1,:))/(xstep**2)
        enddo
    else if (a==2) then ! derivative with respect to y
        do i=2,disc-1,1
            w(:,i)=(psi(:,i+1)-2*psi(:,i)+psi(:,i-1))/(xstep**2)
        enddo
    endif
end subroutine deriv_2_esti
```

Et enfin la subroutine deriv appelée par RK4 :

```
! RK4 Deriv subroutine
subroutine deriv(t,psi,dx)
  use parametres
  implicit none
  real    , intent(inout)       :: t
  complex , dimension(disc,disc)     :: w_x,w_y
  complex , dimension(disc,disc) , intent(inout) :: psi, dx
  call deriv_2_esti(psi,w_x,1)
  call deriv_2_esti(psi,w_y,2)
  dx=i0*(w_x+w_y-V*psi)
end subroutine deriv
```

On peut noter que le potentiel utilisé V est calculé une seule fois à l'initialisation de la propagation puisqu'on a supposé qu'il était constant dans le temps, si nous voulions avoir un potentiel dépendant du temps nous devrions actualiser la matrice $V(i,j)$ à chaque pas de temps. On retrouve le terme laplacien de dimension deux en coordonnées cartésiennes :

$$\nabla^2 f(x,y) = \frac{\partial^2}{\partial x^2} f(x,y) + \frac{\partial^2}{\partial y^2} f(x,y)$$

Pour des raisons de temps d'exécution nous n'avons pas pu faire beaucoup de simulations ni à trop haute résolution, l'espace étant discrétisé sur une grille $\left(\frac{L}{\Delta x}\right)^2$, nous avons choisi $L_x = L_y = 2$ et $\Delta x = \Delta y = 0.002$, donc notre programme fait évoluer 10^6 points à travers RK4 pendant un temps T_{max} et donc $\frac{T_{max}}{\Delta t} = \frac{0.006}{10^{-7}} = 6 \times 10^4$ fois ! La charge de calcul commence à devenir importante et le temps de cacul non négligeable (environ 30 minutes pour les "faibles" résolutions et plusieurs heures pour les "hautes résolutions")

Pour s'assurer que le passage à la deuxième dimension a bien été fait nous allons vérifier le résultat le plus simple, la libre propagation d'un paquet d'onde. Nous nous attendons à ce que son amplitude diminue avec le temps et que le paquet d'onde "s'applatisse" :

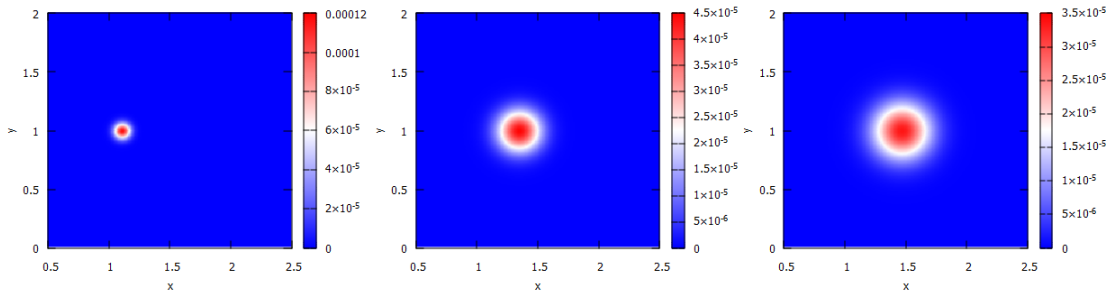


Figure 7: Libre propagation d'un potentiel gaussien à deux dimension.

4.2 Potentiel carré

On peut ensuite retrouver les résultats précédents de la barrière de potentiel :

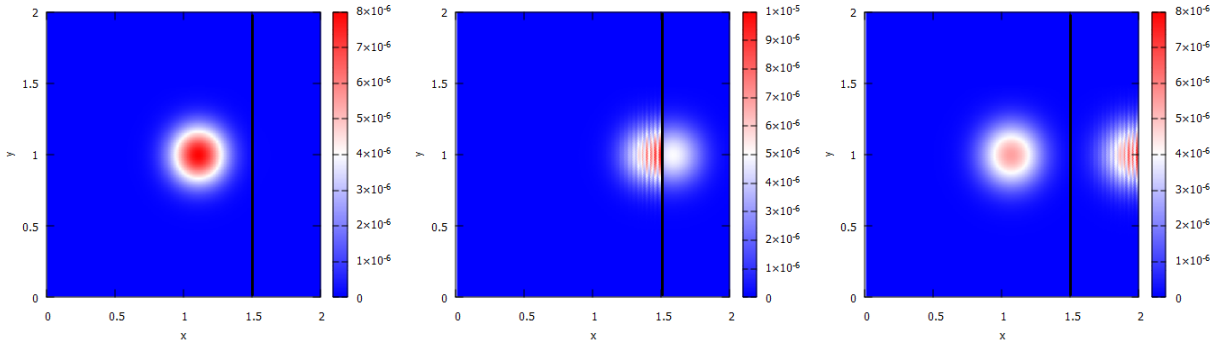


Figure 8: Réflection et transmission d'un paquet d'onde gaussien à deux dimensions.

On retrouve les interférences entre l'onde incidente et l'onde réfléchie par la barrière de potentiel ainsi qu'avec le potentiel infini du bord de la boite en $x = 2$. On peut contrôler l'angle de propagation avec les valeurs initiales de k_x et k_y, par exemple avec un angle d'incidence non nul et donc $k_y \neq 0$:

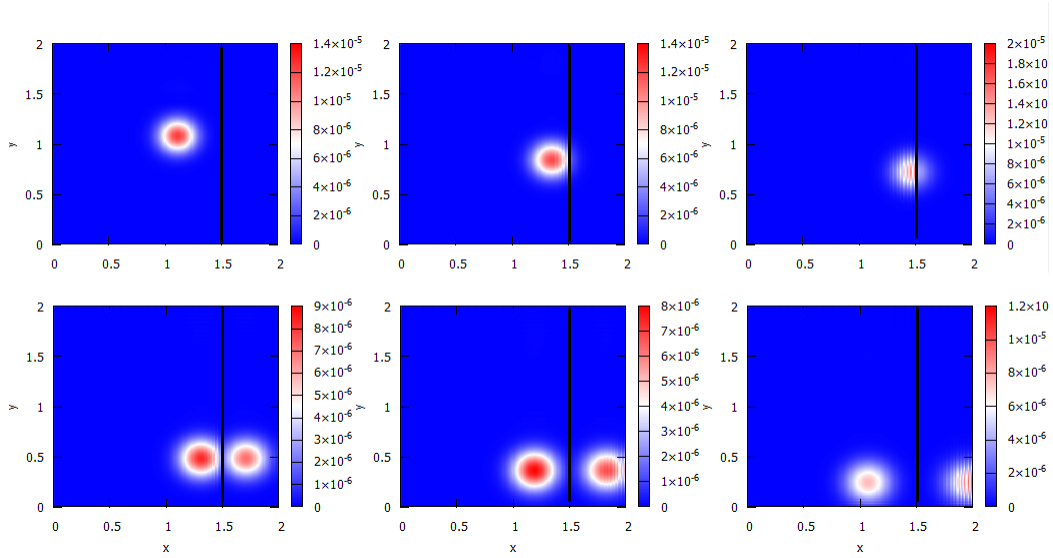


Figure 9: Réflexion et transmission avec une incidence non nulle.

On peut observer que le phénomène d'interférence le long de l'axe (Oy) sur la dimension des x n'est pas impacté par la propagation du paquet d'onde sur les y, les interférences sont similaires à celles d'indicidence normale et complètement parallèles à l'axe (Oy).

4.3 Diffraction par une fente

On peut s'intéresser au phénomène de diffraction et d'interférences où l'on choisi la dimension des trous et de l'espace entre les fentes d'Young en fonction du vecteur d'onde : $a \sim \frac{2\pi}{k} \sim 0.06$ soit environ 30 cases dans notre espace discrétisé.
À partir de maintenant le potentiel n'est plus représenté sur les graphiques pour améliorer la lisibilité aux abords de ce dernier, il est cependant facilement distinguable grâce au contraste de couleurs.

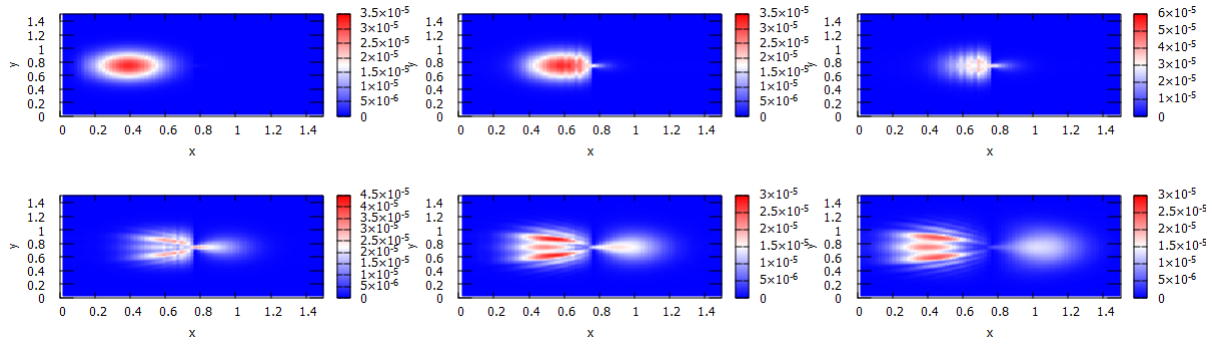


Figure 10: Diffraction avec une taille de fente $a = 0.06 \sim \frac{2\pi}{k}$

On peut observer les interférences de l'onde réfléchie qui ne sont plus contenues que sur l'axe (Oy) maintenant, mais aussi ce qui ressemble au principe de Huygens-Fresnel en électromagnétisme, l'onde incidente est transmise à travers la fente de manière quasi sphérique. En jouant sur la taille de la fente on trouve :

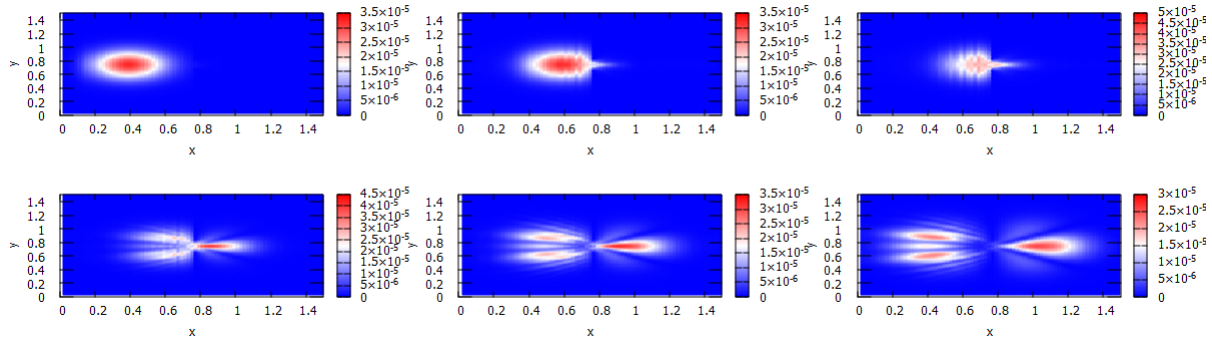


Figure 11: Diffraction avec une taille de fente $a = 0.12 > \frac{2\pi}{k}$

On peut constater la différence majeure avec la simulation précédente est le fait que le lobe centrale de l'onde réfléchie d'avant a été transmis mais il y a également des interférences destructives après la barrière. Les similitudes avec la diffraction d'une onde plane progressive sur une fente prévue par l'électromagnétisme s'arrête donc ici.

4.4 Fentes d'Young

Pour finir on peut simuler l'expérience des fentes d'Young avec comme critère d'interférence et de diffraction $a = b = 0.06 \sim \frac{2\pi}{k}$:

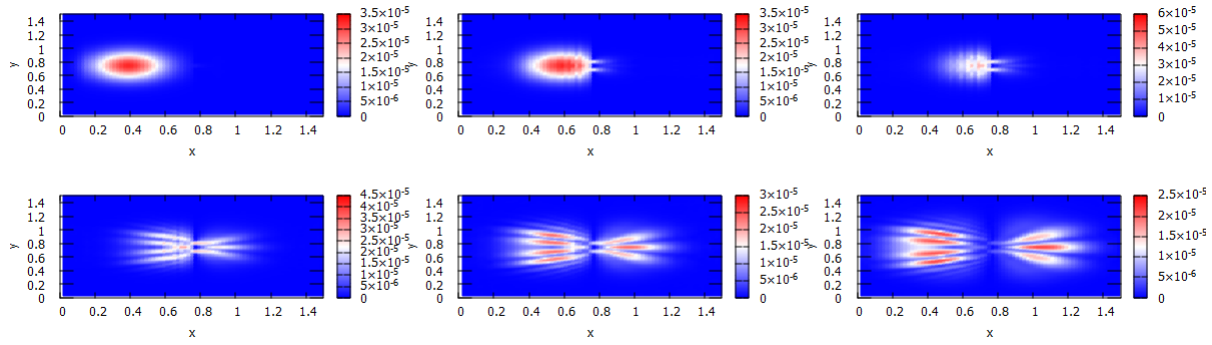


Figure 12: Expérience des fentes d'Young avec $a = b = 0.06 \sim \frac{2\pi}{k}$

On peut observer que le nombre de lobes réfléchis et transmis à augmenté. Nous ne retrouvons pas une figure en sinus cardinal de l'électromagnétisme alors que l'on se trouve bien dans les conditions de Fraunhofer, on a $d \gg \frac{a^2 k}{2\pi} \sim 0.05$

À noter que nous n'avons pas fait le suivi de la norme pour la partie à deux dimensions mais on peut l'estimer grossièrement en supposant que chaque vecteur 1D subit une perte estimée en première partie à $\sim 10^{-5}$ soit $10^{-5} \times \frac{L}{\Delta x} \sim 7.5 \times 10^{-3}$ ce qui serait très satisfaisant.

5 Conclusion

Nous avons retrouvé de manière satisfaisante les prédictions de la théorie et nous avons fait l'étude de la propagation d'un paquet d'onde gaussien à deux dimensions dans des cas typiques de diffusion par un potentiel. Dans un soucis de temps nous avons laissé de côté un certain nombre de considérations théoriques et pratiques, on peut citer de manière non exhaustive : la comparaison avec différents algorithmes, la comparaison avec d'autre langages de programmation, l'optimisation générale du code, le suivi de la norme à deux dimensions, un potentiel de forme quelconque ou dépendant du temps, la représentation impulsion en faisant la transformée de Fourrier de la fonction d'onde, l'application à la radioactivité ou encore la diffusion Coulombienne.

6 Bibliographie

[1] Dr. C Guillen et Dr. R Bader, "HPC Compilers." `https://www.admin-magazine.com/HPC/Articles/Selecting-Compilers-for-a-Supercomputer`, HPC Compilers.

[2] G. F. Simon Huppert, "Équations différentielles ordinaires, équations aux dérivées partielles et leur résolution numérique," 2023-2024.

[3] E. T. Loren Jørgensen, David Lopes Cardozo, "Numerical resolution of the schrödinger equation," 2011.

[4] B. D. e. F. L. Claude Cohen-Tannoudji, "Mécanique quantique," p. 75, 2018.

7 Annexe A

Dans un soucis de place nous ne mettons que le dernier code utilisé pour la diffusion à deux dimensions :

```
module parametres
    real    , parameter :: L=1.5 , xstep=0.002 , Tmax = 0.005
    real    , parameter :: dt = 0.00000025 , pi= acos(-1.0)
    real    , parameter :: kx = 100,ky=0 , sigma = 0.1
    real    , parameter :: x0 = 0.3, y0 = L/2
    integer , parameter :: disc = int(L/xstep)+1
    integer , parameter :: steps = Tmax/dt
    integer , parameter :: dimension = 1 , n = 2*dimension
    integer , parameter :: set = steps/100
    complex , parameter :: i0=complex(0,1)
    character(len=20)   :: potential_shape = 'hole'
    real , dimension(disc,disc) :: V
  end module parametres

  module square_well_parameters
    use parametres
    real , parameter :: potential_position = L/2 , V0 = 1E6
    integer :: potential_length = 10, potential_gap = 20
    integer ::  potential_distance = 60

  end module square_well_parameters

  module func
    use parametres
    use square_well_parameters
    contains
```

```
  subroutine write_in_file(i,psi)
    use parametres
    implicit none
    integer , intent(in)                    :: i
    complex(4), dimension(disc,disc)        :: psi
    integer                                 :: j,p
    integer                                 :: file_unit
    character(len=20)                       :: file_name
    file_unit = i+10
    !call wave_function(norm,x,y)
    if (file_unit>=10)then
      open(file_unit,file=file_name(i),status='replace')
      do j=1,disc,1
        if (modulo(j,10)==0)then
          do p=1,disc,1
            if (modulo(p,10)==0) then
              write(file_unit,*)(p-1)*xstep,(j-1) \
                              *xstep,cabs(psi(p,j))
            endif
          end do
          write(file_unit,*)' '
        endif

      enddo
    endif
    close(file_unit)
  end subroutine write_in_file

  ! Clearing data files from previous run
  subroutine clear_file()
    open(9,file='Tcoef',status='replace')
    open(11,file='data\file_1.dat',status='replace')
    close(9)
    close(11)
  end subroutine clear_file
end module func

program main
  use parametres
  use func
  implicit none
  complex(4) , dimension(disc,disc) :: psi
  real                              :: t=0
  call clear_file()
```

```
  call init(psi)
  call propagate(t,psi)
end program main

! Propagation subroutine
subroutine propagate(t,psi)
  use parametres
  use func
  use square_well_parameters
  implicit none
  real                          :: t
  complex(4) , dimension(disc,n) :: x,y
  complex(4), dimension(disc,disc) :: psi
  integer             :: j,i=1
  integer             :: pot_x1 , pot_x2
  external            :: deriv
  i=1
  j=2
  pot_x1 = int(potential_position/xstep)
  pot_x2 = int(potential_position/xstep+potential_length)
  call pot()
  do while (j <= steps)
    call rk4(t,psi,dt,deriv)
    psi(1,:)=0 ; psi(disc,:)=0
    psi(:,1)=0 ; psi(:,disc)=0
    if (modulo(j,set)==0)then
      write(*,*)'Writing␣in␣file␣:',i
      i=i+1
      call write_in_file(i,psi)
      call normal_check(psi)
    endif
    t=t+dt
    j=j+1
  enddo
end subroutine propagate


! Normalizing subroutine
subroutine normalize(psi)
  use parametres
  implicit none
  complex(4), dimension(disc,disc), intent(inout) :: psi
  real :: sum
  integer :: i,j
```

```
  sum=0
  do j=1,disc,1
      do i=1,disc,1
      sum=sum+cabs(psi(i,j))
      enddo
  enddo
  psi=psi/(sum)
end subroutine normalize

subroutine normal_check(psi)
  use parametres
  implicit none
  complex, dimension(disc,disc), intent(in) :: psi
  real    :: sum
  integer :: i,j
  sum=0
  do j=1,disc,1
      do i=1,disc,1
      sum=sum+cabs(psi(i,j))
      enddo
  enddo
  write(*,*)'Norm = ',sum
end subroutine normal_check



 ! Potential defining subroutine
subroutine pot()
  use parametres
  use square_well_parameters
  implicit none
  integer      :: i , j , p, c, a
  if (potential_shape=='square')then
    i = int(potential_position/xstep)
    j = potential_length
    V=0
    V(i:i+j,:)=V0
  else if (potential_shape == 'hole')then
      i = int(potential_position/xstep)
      j = potential_length
      p = potential_gap/2
      c = int(disc/2)
      V=0
      V(i:i+j,:)=V0
```

```
        V(:,c-p:c+p)=0
    else if (potential_shape == 'young')then
        i = int(potential_position/xstep)
        j = potential_length
        p = potential_gap/2
        c = int(disc/2)
        a = potential_distance/2
        V=0
        V(i:i+j,:)=V0
        V(:,c-a-p:c-a+p)=0
        V(:,c+a-p:c+a+p)=0
    endif
end subroutine pot


! RK4 Deriv subroutine
subroutine deriv(t,psi,dx)
    use parametres
    implicit none
    real                              , intent(inout) :: t
    complex , dimension(disc,disc)    :: w_x,w_y
    complex , dimension(disc,disc) , intent(inout) :: psi, dx
    call deriv_2_esti(psi,w_x,1)
    call deriv_2_esti(psi,w_y,2)
    dx=i0*(w_x+w_y-V*psi)
end subroutine deriv


! Second spatial derivative estimation
subroutine deriv_2_esti(psi,w,a)
    use parametres
    implicit none
    complex(4) , dimension(disc,disc) , intent(in)      :: psi
    complex(4) , dimension(disc,disc) , intent(inout)   :: w
    integer                                             :: i,a
    w=0
    if (a==1) then
        do i=2,disc-1,1
            w(i,:)=(psi(i+1,:)-2*psi(i,:)+psi(i-1,:))/(xstep**2)
        enddo
    else if (a==2) then
        do i=2,disc-1,1
            w(:,i)=(psi(:,i+1)-2*psi(:,i)+psi(:,i-1))/(xstep**2)
        enddo
```

```
    endif
end subroutine deriv_2_esti

  ! File name generation function with integer input
  function file_name(i) result(fname)
    implicit none
    character(len=20) :: fname
    integer , intent(in) :: i
    write(fname,'(a,i0,a)')'data\file_',i,'.dat'
  end function file_name

  ! Initialisation subroutine
  subroutine init(psi)
    use parametres
    use func
    implicit none
    complex(4) , dimension(disc)          :: x,y
    complex(4) , dimension(disc,disc)     :: w_x,w_y
    complex(4) , dimension(disc,disc)     :: psi
    integer :: i,j
    do j=1, disc,1
        y(j)=exp(-((j-1)*xstep-y0)**2/(4*sigma**2))\
                                *exp(i0*((j-1)*xstep-y0)*ky)
        do i=1,disc,1
            psi(i,j)=exp(-((i-1)*xstep-x0)**2/(4*sigma**2))\
                        *exp(i0*((i-1)*xstep-x0)*kx)*y(j)
        enddo
    enddo
    call normalize(psi)
    call normal_check(psi)
    call deriv_2_esti(psi,w_x,1)
    call deriv_2_esti(psi,w_y,2)
    call write_in_file(1,psi)
  end subroutine init

  !RK4
  subroutine rk4(t,x,dt2,deriv)
    ! 4th order Runge-Kutta. See Numerical Recipes p. 701ff
    use parametres
    implicit none
    real                , intent(in)                :: t, dt2
    complex(4), dimension(disc,disc), intent(inout) :: x
    real                                            :: ddt
    complex, dimension(disc,disc)     :: xp, k1, k2, k3, k4
```

```
    external :: deriv
    ddt = 0.5*dt2
    call deriv(t,x,k1)       ; xp = x + ddt*k1
    call deriv(+ddt,xp,k2) ;  xp = x + ddt*k2
    call deriv(t+ddt,xp,k3) ; xp = x +  dt*k3
    call deriv(t+dt,xp,k4);x= x +  dt*(k1+2.0*k2+2.0*k3+k4 )/6.0
  end subroutine rk4
```

À noter que certaines lignes étaient trop longues pour le format de page et ont donc été tronquées avec un \ . Les commentaires ont également été retirés pour la même raison. Je peux toutefois fournir sur demande le code utilisé dans chacune des parties de ce rapport.