Laboratoire 3

Intégrales numériques

Ce labo vous fait calculer des intégrales définies et indéfinies par intégration numérique. C'est un exercice mathématique qui vous servira à répétition dans vos cours de physique à venir... ainsi qu'au laboratoire 4. Les développements numérico-théoriques requis se retrouvent à la §2.4 des notes de cours.

3.1 Objectifs

1. Apprendre à effectuer numériquement des intégrales définies avec des intégrants de forme non-triviale,
2. Apprendre à estimer et améliorer la précision des intégrales numériques;
3. Approfondir l'utilisation des fonctions en programmation C;
4. Poursuivre l'apprentissage de PLPLOT.
5. Apprendre une autre façon bizarre de calculer le nombre irrationnel π.

3.2 Rapport de Lab

Il n'y a pas de rapport à remettre pour ce laboratoire. Cependant, vous avez la possibilité de remettre un rapport qui sera corrigé mais ne sera pas comptabilisé dans votre note finale; l'idée ici est de vous permettre de recevoir une première rétroaction pour mesurer jusqu'à quel point le contenu et la présentation de vos rapports répondent aux attentes. Notez également que le prochain laboratoire vous donnera l'occasion de réutiliser plusieurs codes et concepts développés dans le cadre du labo de cette semaine; il sera donc à votre avantage de bien structurer et documenter vos codes C (commentaires, indentations, etc.).

3.3 Intégrales par la méthode du trapèze

On a déjà vu au chapitre 1 une manière bizarroide de calculer la valeur numérique du nombre irrationnel π; en voici une autre:

$$\pi = 4 \int_0^1 \sqrt{1-x^2}\, \mathrm{d}x \ . \tag{3.1}$$

Nous allons l'utiliser dans ce labo pour tester la précision des intégrales numériques par la méthode du trapèze. Nous allons tout d'abord réécrire l'expression ci-dessus sous la forme:

$$\pi = \int_0^1 f(x) \mathrm{d}x \ , \qquad \text{avec} \qquad f(x) = 4\sqrt{1-x^2} \ . \tag{3.2}$$

ce qui semble trivial mais, dans ce qui suit, nous allons toujours coder nos intégrants sous la forme de fonctions C, alors aussi bien l'anticiper dans nos développements "théoriques"!

Il s'agit ici d'une intégrale définie sur un intervalle $[0, 1]$ en x; la première étape sera donc de se définir une maille couvrant uniformément cet intervalle. Toujours en anticipation de ce qui suivra, il sera pratique de le faire en terme d'un nombre de points de maille spécifié dans le code C même; par exemple, la série d'instructions données dans le fragment de code qui suit définit une maille équidistante en x, comprenant N (= 100 ici) points de maille répartis uniformément entre les bornes $x =$ `xi` et `xo` (voir aussi la §2.4.1 des Notes):

```
#include <stdlib.h>
#define N 100                       /* nombre de points de maille */
int main(void)
{
  float x[N] ;                      /* Tableau 1D qui contiendra la maille */
  float xi, xo ;                    /* bouts de la maille */
  ...                               /* autres declarations/instructions */
  xi=0. ;                           /* debut de la maille */
  xo=1. ;                           /* fin de la maille */
  for (k=0 ; k<N ; k++ ) {          /* construction de la maille... */
       x[k]=xi+(xo-xi)*k/(N-1.) ; /* ...un point a la fois */
  }
  ...                               /* la suite du code... */
}
```

Notez que discrétiser ainsi l'intervalle $[0, 1]$ à l'aide de N points de maille, le premier et le dernier correspondant aux bornes de l'intégrale, définit $N - 1$ (et non pas N!) sous-intervalles contigus couvrant l'intervalle total. Attention aux débordements de tableaux! Maintenant, suivez la procédure suivante:

1. Codez l'intégrant de l'éq. (3.1) sous la forme d'une fonction C (appelée `ff`, par exemple) qui accepte un paramètre en argument, soit une valeur de x, et retourne l'évaluation de la fonction $f(x)$ telle que définie ci-dessus.

2. Codez la méthode du trapèze pour la solution de l'intégrale donnée par l'éq. (3.1), en utilisant évidemment votre fonction C pour calculer l'intégrant à chaque point de maille (les f_k dans l'éq. (2.40) des notes de cours). N'oubliez pas l'idée de la variable d'accumulation, tel qu'introduite dans le fragment de code suivant l'éq. (2.40) des Notes.

3. Évaluez votre intégrales pour différents nombres de points de maille dans l'intervalle $[0, 1]$; par exemple, $N = 3, 10, 30, 100, 300, 1000, 3000, 10000$.

4. Pour chaque valeur calculée de l'intégrale ($I(N)$, disons), calculez l'erreur par rapport à la solution attendue; on définira ici cette erreur comme:

$$\varepsilon(N) = |I(N) - \pi|$$

5. Définissez deux tableaux 1D, le premier contenant le logarithme en base 10 (avec la fonction C `log10`) des valeurs de N associées à vos différents calculs de l'intégrale (`logmaille` disons), et le second la valeur du logarithme en base 10 de l'erreur $\varepsilon(N)$ y étant associée (e.g., `logerreur`). Si vous avez suivi à la lettre les instructions au point 3 ci-dessus, ces tableaux contiendront chacun 8 éléments... numérotés "à l'interne" de 0 à 7, ne pas l'oublier. Encore une fois, attention aux débordements de tableaux!

3.4 Graphique de convergence

Il s'agit maintenant de tracer un **graphique de convergence**; ceci consiste essentiellement à porter en graphique $\log(\varepsilon(N))$ versus $\log(N)$ sur un graphique, et offre une excellente occasion d'élargir vos capacités graphiques avec PLPLOT. Les étapes sont les suivantes

1. Vous devez, comme auparavant (viz. Fig. 2.1 du labo 2), commencer par initialiser PLPLOT (appel à `plinit`), établir l'étendue des axes (appel à `plenv`), et ajouter des annotations si désiré (appel à `pllab`). Attention, assurez vous de ne *pas* poser `xmin`= 0 ou `ymin`= 0 (ou encore pire, des valeurs négatives!) dans l'appel à `plenv`, parce que sur un graphique log-log ce serait vraiment pas beau... Et n'oubliez pas l'appel à `plend` à la toute fin du code.

2. Tracez votre courbe de convergence, $\log(\varepsilon)$ versus $\log(N)$, par un appel approprié à `plline`. Assurez vous que les deux tableaux fournis en entrée à `plline` on bien été déclarés `double`. Je sais, c'est agaçant, mais c'est comme ça...

3. Répétez l'étape précédente, mais cette fois à utilisez la valeurs 30 (plutôt que 1) pour le dernier paramètre dans l'appel à `plenv`. C'est joli, non ? Notez bien que ce paramètre contrôle uniquement le traçage des axes; le logarithme des quantités à porter en graphique doit déjà avoir été pris avant l'appel à `pljoin` ou `plline`.

4. À l'aide de votre graphique de convergence, vérifiez que l'erreur diminue bien selon $1/N^2$, comme on s'y attendrait d'une méthode $O(h^2)$. Si vous manquez d'inspiration ici, aller en chercher sur la Fig. 2.5 des notes de cours.

3.5 Exercice de programmation semi-avancée

Passons maintenant à un exercice de programmation qui devrait finir de vous familiariser avec la définition et l'usage des fonctions en C. L'idée est de vous définir une fonction, s'appelant `integreTrapeze1D`, qui accepte en argument le tableau 1D `x` contenant les points de maille, et un entier `n` donnant leur nombre. Un seul appel à cette fonction doit vous renvoyer la valeur de l'intégrale définie. Examinez bien le canevas présenté ci-dessous et remarquez bien comment doit être passé (et déclaré) un tableau en argument à une fonction. Notez également que la taille de ce tableau doit avoir été définie à la compilation (ici via une instruction `DEFINE`). NOTEZ BIEN les "`[]`" suivant le nom du tableau dans la déclaration de la fonction dans `main`, ainsi que dans sa définition suivant `main`, MAIS PAS EN APPEL dans `main`. Notez également que le nombre de point de maille `n` donné en appel à `integreTrapeze1D` ne doit pas nécessairement être égal à `NMAX`, mais ne peux pas être plus grand sinon un débordement de tableau causera une erreur à l'exécution. Il serait prudent d'ajouter un test de sécurité dans `main` pour empêcher un tel dérapage prévisible...

Quant à la fonction `integreTrapeze1D` même, elle doit incorporer la partie de votre code précédent effectuant le calcul de l'intérale par la méthode du trapèze, avec l'intégrant lui-même défini par une fonction C appropriée, comme notre `ff` d'auparavant. Donc, `main` appelle `integreTrapeze1D` en lui passant la maille en argument, et `integraTrapeze1D` appelle `ff` pour calculer l'intégrant à chaque point de maille, comme l'exige la méthode du trapèze. Ce genre de modularité est la règle dans tout "vrai" code bien structuré[1].

Le fragment de code C ci-dessous donne le canevas global requis. À vous de combler les parties manquantes, soit tous les "..."; Ici on a supposé que la variable `integrale` de type `float` dans la fonction `integreTrapeze1D` contient le résultat du calcul de l'intégrale et que la variable `integrant` de type `float` dans la fonction `ff` contient le résultat du calcul de

[1] En C il serait aussi possible de passer en argument à `integreTrapeze1D` le *nom* de la fonction définissant l'intégrant; la syntaxe est loin d'être transparente, et ce genre de manoeuvre n'est pas permis dans la plupart des langage de programmation; il a donc été jugé préférable de ne pas utiliser ici cette fonctionalité du C. Les enthousiastes peuvent trouver comment faire sur le Web...

l'intégrant en fonction de la valeur de x au point de maille fourni en argument. Notez encore une fois comment les variables sont purement locales aux fonction; du point de vue de `main` et `integreTrapeze1D`, `x` est un tableau 1D de dimension `NMAX`, mais dans la fonction `ff`, `x` est une simple variable de type `float`; Certains jugeraient que l'utilisation d'un nom de variable différent (e.g., `xx`) serait préférable pour éviter la confusion. Quel que soit le nom choisi, l'appel par `integreTrapeze1D` assignera à cette variable locale la valeur correspondant à un élément du tableau `x[NMAX]`, via un appel du genre "`ff( x[k] )`".

```
#include <stdlib.h>
#define NMAX 10000                /* nombre maximal de points de maille */
int main(void)
{
  float integreTrapeze1D( float[], int ) ;
  float x[NMAX] ;                 /* tableau contenant la maille */
  ...                             /* autres declarations/instructions/etc */
  n=100 ;                         /* nombre desire de points de maille */
  xi=0. ;                         /* borne inferieure de l'integrale */
  xo=1. ;                         /* borne superieure de l'integrale */
  ...                             /* calcul de la maille */
  resultat=integreTrapeze1D(x,n) ;
  ...                             /* la suite du code, printf, etc... */
}
float integreTrapeze1D( float x[], int n )
{
  float ff( float ) ;             /* la fonction a integrer */
  ...                             /* declarations des variables locales  */
  ...                             /* integrale par trapeze */
  return integrale ;
}
float ff(float x)                /* L'integrant en fonction de x */
{
  ...                             /* declarations des variables locales  */
  integrant= ... ;                /* calcul de l'integrant a x */
  return integrant ;
}
```

Bref, une fois tout ça codé correctement, vous devriez obtenir le même résultat numérique que précédemment (pour le même maillage et les mêmes bornes d'intégration, évidemment). Vérifiez que c'est bien le cas, et montrez le tout par un(e) démonstrateur(trice); et voilà, le labo 3 est terminé !

Lectures supplémentaires:

Notes de cours: Sections 2.2 et 2.4
Delannoy: Chapitres 7 et 8