RENDU D’EXERCICES N°02

Réalisé par:Mr Lougani Faouzi

Module:Algo et calcul scientifique-L3 math-info

(2019/2020)

2. Flottants en C

A. Bien connaître son environnement

Les élements qui semblent pertinents pour caractériser l' environnement de travail pour ce type de travail : un projet en C avec une forte composante de calcul en arithmétique flottante.

-savoir la difference entre les types utilisés float et double c'est a dire avoir une préalable connaissance de IEEE 754 simple précision ainsi que la double précision
-les modes d'arrondi des nombres flottants (infini ,zero ...plus prés)
-une bonne connaissance des fonctions mathematiques de la bibliotheque <math.h> (surtout leurs format d'entrée et sortie)
- maitriser les operations arithmétiques à virgule flottante,(addition ...multiplication …)
- une maitrise du langage C et une connaissance architecurale en plus de la machine

B. Quelques vérifications

1- le code suivant nous explique le principe du gradual underflow en faisant la soustraction du plus petit flottant strictement positif **x** d'un flottant plus grand que lui **y** afin de voir si le le resultant de l'opération n'est pas nul mais avec une valeur absolue inférieure au plus petit nombre à virgule flottante normalisé

```c
#include<stdio.h>

int main(){

        float x,y,z;
                x=1.17549435082e-38; // x=(1.00000000000000000000000 en base bin)*2^-126
                y=1.17549449095e-38; // y=(1.00000000000000000000001 en base bin)*2^-126
                z=y-x;
                printf("%.23e",z); //   z=1.40129846432481707092373e-45
                                   //   z=(0.00000000000000000000001 en base bin)*2^-126

return 0;
}
```

Illustration 1: capture3

on voit que le flottant **z est inférieure** au plus petit nombre à virgule flottante normalisé (1,0000….*2^-126)
donc oui en calcul en float on applique **le gradual underflow**
pour l'autre traitement est possible a faire c'est **la normalisation** c'est a dire vider le resultant a zero et donc **z** sera noté aprés nomalisation z=(1.00000000000000000000 en base bin)*2^**-149**   et non pas 126 comme il etait lors de la représentation **sous-normale**

remarque: pour les valeurs exactes je me suis fais aider d'un IEEE-754 Floating Point Converter en ligne (http://www.binaryconvert.com/convert_float.html)afin d'avoir des les valeurs exactes

```c
#include<stdio.h>

int main(){

        double x,y,z;
                x=2.2250738585072013830e-308; // 1.00000000000000000000000
                y=2.2250738585072018770e-308; // 1.00000000000000000000001
                z=y-x;
                printf("%.23e",z); //   z=4.94065645841246544176569e-324
                                   //     0.0000000000000000000001


return 0;
}
```

Illustration 2: capture4

les resultat sur les doubles:
on remarque que c’est le meme resultat obtenu avec les **flottants** donc **il ya le gradual underflow avec les doubles.**

2-- J'identifie par le calcul le plus petit flottant x > 0 tq. 1 + x != 1 (ou 1 + x > 1)
Aprés une lecture de la partie *Hardware Extended Precision* (**chapitre 8** d’Overton) j’ai conclu qu’il ya 2 cas:

1er cas: séquence d'instructions à virgule flottante utilisant meme precision d’entrée et meme precision de sortie => ne donne pas peut etre le resultat correctement arrondi une fois les calculs effectués

2eme cas:séquence d'instructions à virgule flottante fonctionnant sur des données dans les registres=> resultat cumulé plus précis une fois les calculs effectués=>arrondir le resultat pour s’adapter au format de precision =>le stocker en memoire lorsque la séquence est términée

```c
#include<stdio.h>

int main(){

        float x,y,z;
              x=1; //
              y=1.40129846432e-45; // y=(0.00000000000000000000001 en base bin)*2^-126
              // j ai choisi y comme chiffre sous-normaux (plus petit) pour bien avoir la précision

              z=x+y;
              printf("%e\n",z);
              printf("%f",z)
return 0;
}
```

Illustration 3: capture 5

le code suivant(capture 5) en utilisant *float* aprés execution il donne 1+x=1 malgrés on a choisi le plus petit flottant >0 donc on deduit que le type **float n’utilise pas les registres en précision étendue**

```c
#include<stdio.h>

int main(){

        double x,y,z;
              x=1.0; //
              y=4.940656458412e-324;
        // meme pour le double j ai choisi y comme chiffre sous-normaux (plus petit)
        // pour bien avoir la précision

              z=x+y;
              printf("%.24e\n",z);
              printf("%f",z);
return 0;
}
```

Illustration 4: capture6

le code suivant(capture 5) en utilisant *double* aprés execution il donne 1+x=1 malgrés on a choisi le plus petit flottant >0 donc on deduit que le type ***double* n'utilise pas les registres en précision étendue.**

Remarque : Aprés une petite recherche j'ai trouvé que GCC implemente `long double` avec des nombres à virgule flottante 80 bits (long double utilise les registres en précision etendue)

4-Dans Overton complet, ch 10. : résoudre l'ex. 10.3

cas 1 la division:
le code ci aprés me permet durant le traitement de voir quand mon flottant s'annule (%f) ,la précision (%e),et l'iteration correspondante i

```c
#include<stdio.h>
int main(){

     int i;
     float x=1.0/10.0;
        printf("x=%e\n",x);
        for(i=0;i<150;i++)
        {
                x=x/2.0;
                printf("x=%f   x= %e   i=%d \n",x,x,i);

        }

return 0;
}
```

Illustration 5: capture7

une simple execution (la capture suivante-capture 8) nous indique que notre **float** s'annule a partir de i=17
par contre la precision est toujours superieure a la normale ,je continue ma boucle et voir quand la precision sera inferieure a la normale .

Illustration 6: capture8

La capture aprés (capture 9) nous montre que la précision est inferieure a la normale a partir de i=39

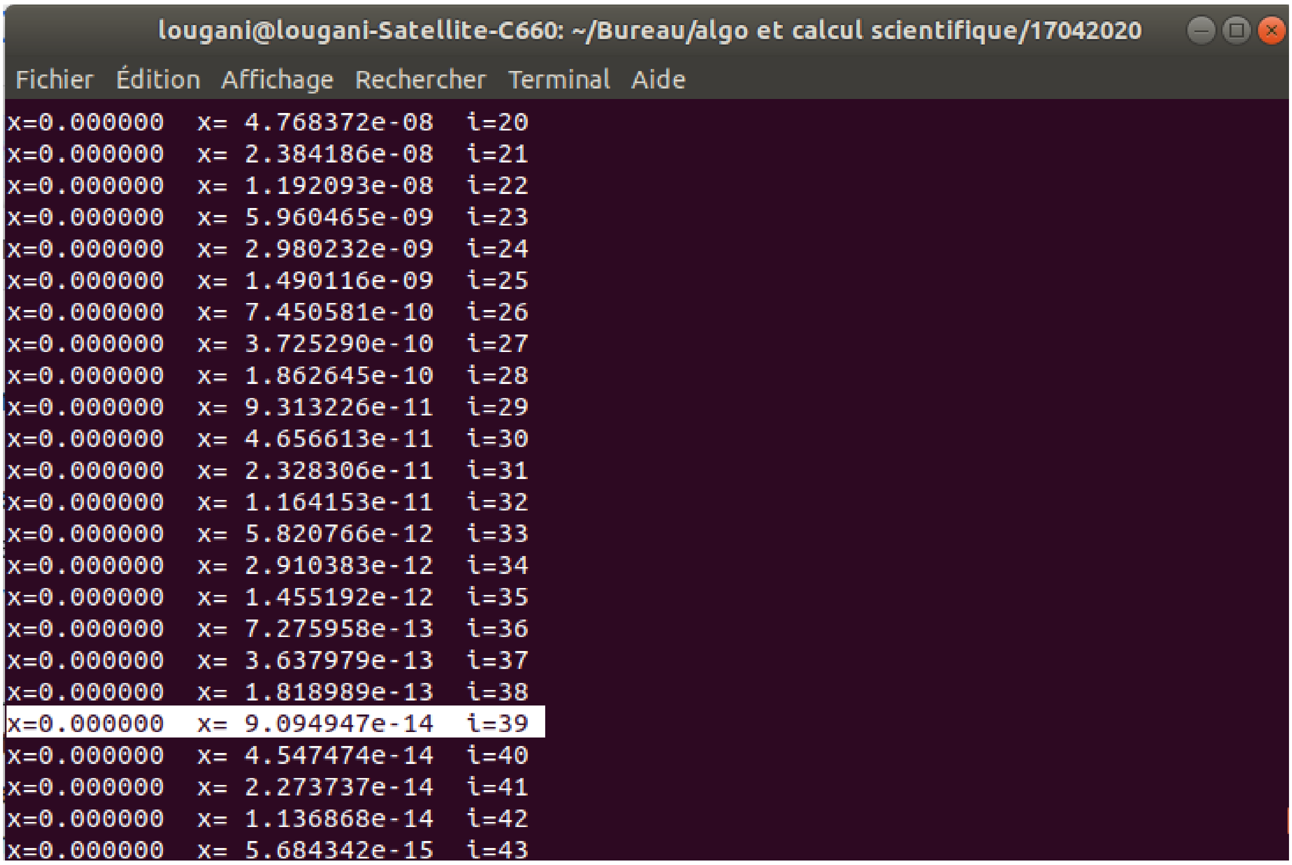

Illustration 7: capture9

la precision sera predue a i=146 comme nous le montre ci-joint la capture 10

```
lougani@lougani-Satellite-C660: ~/Bureau/algo et calcul scientifique/17042020
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
x=0.000000   x= 2.938733e-40   i=127
x=0.000000   x= 1.469374e-40   i=128
x=0.000000   x= 7.346868e-41   i=129
x=0.000000   x= 3.673364e-41   i=130
x=0.000000   x= 1.836682e-41   i=131
x=0.000000   x= 9.184110e-42   i=132
x=0.000000   x= 4.592055e-42   i=133
x=0.000000   x= 2.295327e-42   i=134
x=0.000000   x= 1.147663e-42   i=135
x=0.000000   x= 5.745324e-43   i=136
x=0.000000   x= 2.872662e-43   i=137
x=0.000000   x= 1.429324e-43   i=138
x=0.000000   x= 7.146622e-44   i=139
x=0.000000   x= 3.643376e-44   i=140
x=0.000000   x= 1.821688e-44   i=141
x=0.000000   x= 8.407791e-45   i=142
x=0.000000   x= 4.203895e-45   i=143
x=0.000000   x= 2.802597e-45   i=144
x=0.000000   x= 1.401298e-45   i=145
x=0.000000   x= 0.000000e+00   i=146
x=0.000000   x= 0.000000e+00   i=147
x=0.000000   x= 0.000000e+00   i=148
x=0.000000   x= 0.000000e+00   i=149
lougani@lougani-Satellite-C660:~/Bureau/algo et calcul scientifique/17042020$
```

Illustration 8: capture10

donc pour la multiplication on va choisir le plus petit x or celui en position i=145 (c'est a dire avant que la précision soit perdue) et faire la demarche inversse a fin de voir si on va avoir le x initial (x=1.000000e-01) et c'est ce que le code ci aprés (capture 11) est entrain de nous expliquer .

```c
#include<stdio.h>
int main(){

     int i;
     float x=1.401298e-45;
        printf("x a l'etat initiale =%e\n",x);
        for(i=0;i<146;i++)
        {
                x=x*2.0;
                printf("x=%f   x= %e   i=%d \n",x,x,i);

        }

return 0;
}
```

Illustration 9: capture11

Une execution du code précedent nous indique qu'on a pas obtenu le meme resultant et c'est ce que montre la capture ci-joint

```
lougani@lougani-Satellite-C660: ~/Bureau/algo et calcul scientifique/17042020
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
x=0.000000   x= 2.980232e-08   i=123
x=0.000000   x= 5.960464e-08   i=124
x=0.000000   x= 1.192093e-07   i=125
x=0.000000   x= 2.384186e-07   i=126
x=0.000000   x= 4.768372e-07   i=127
x=0.000001   x= 9.536743e-07   i=128
x=0.000002   x= 1.907349e-06   i=129
x=0.000004   x= 3.814697e-06   i=130
x=0.000008   x= 7.629395e-06   i=131
x=0.000015   x= 1.525879e-05   i=132
x=0.000031   x= 3.051758e-05   i=133
x=0.000061   x= 6.103516e-05   i=134
x=0.000122   x= 1.220703e-04   i=135
x=0.000244   x= 2.441406e-04   i=136
x=0.000488   x= 4.882812e-04   i=137
x=0.000977   x= 9.765625e-04   i=138
x=0.001953   x= 1.953125e-03   i=139
x=0.003906   x= 3.906250e-03   i=140
x=0.007812   x= 7.812500e-03   i=141
x=0.015625   x= 1.562500e-02   i=142
x=0.031250   x= 3.125000e-02   i=143
x=0.062500   x= 6.250000e-02   i=144
x=0.125000   x= 1.250000e-01   i=145
lougani@lougani-Satellite-C660:~/Bureau/algo et calcul scientifique/17042020$
```

Explication :

Les nombres à virgule flottante sont représentés dans le matériel comme des fractions de base 2 donc les nombres décimaux à virgule flottante que vous entrez ne sont qu'approximés par les nombres binaires à virgule flottante réellement stockés dans la machine
quel que soit le nombre de chiffres de base 2 que vous souhaitez utiliser, la valeur décimale 0,1 ne peut pas être représentée exactement comme une fraction de base 2. Dans la base 2, 1/10 est la fraction répétée à l'infini
comme suit:
0.0001100110011001100110011001100110011001100110011...
de ce fait si on fais la division de cette valeur sur 2 et l'inversse on aura jamais le meme resultat (valeur approximé)

3.Cancellation

1-La reprise en C + gnuplot le travail exposé dans le chapitre et qui conduit à la figure 11.1
Remarque: j'ai stocké mes points dans un fichier «donnees.txt» apartir de ce dernier j ai fais mon plot qui donne la figure suivante qui est identique a la 11.1

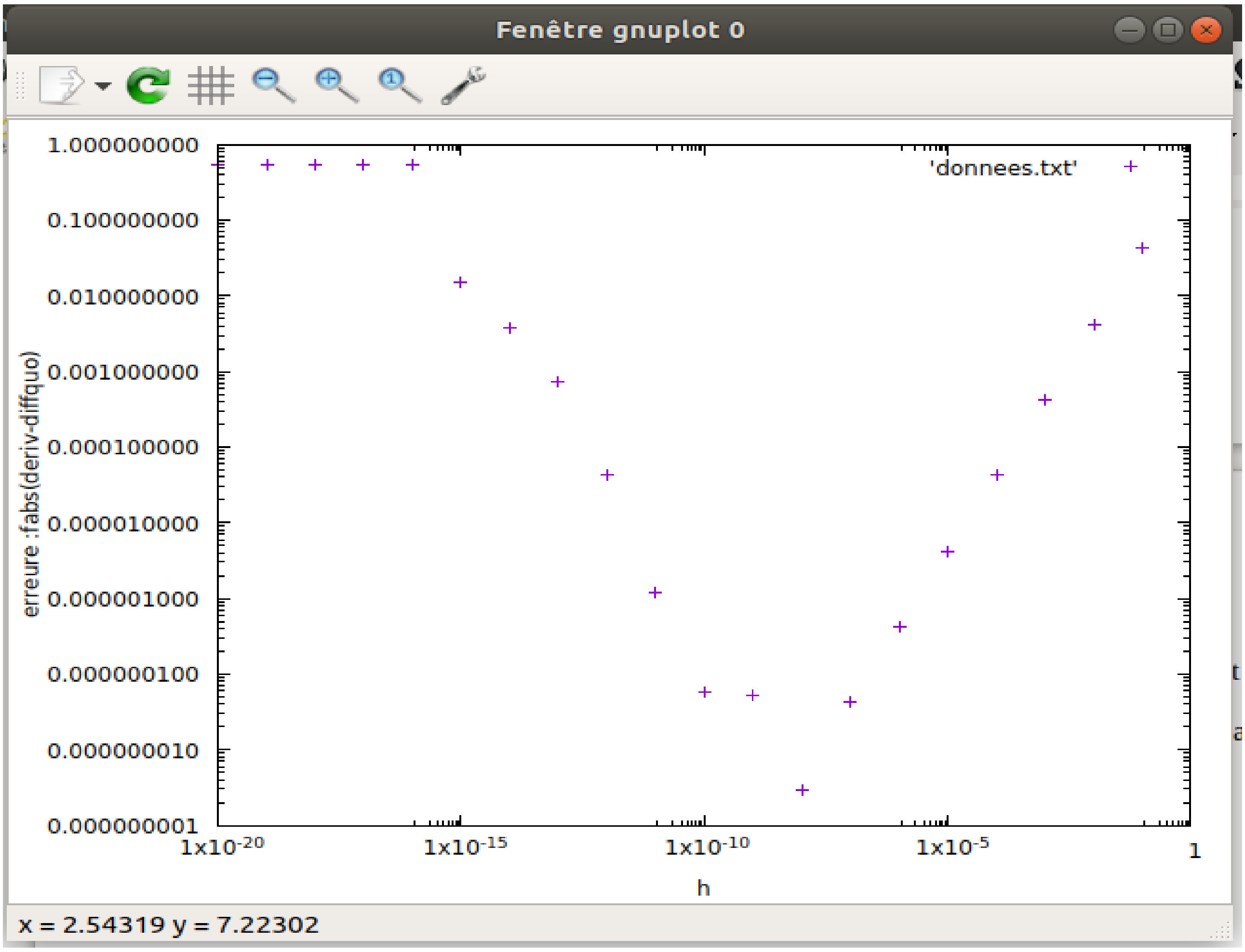

Illustration 10: capture1

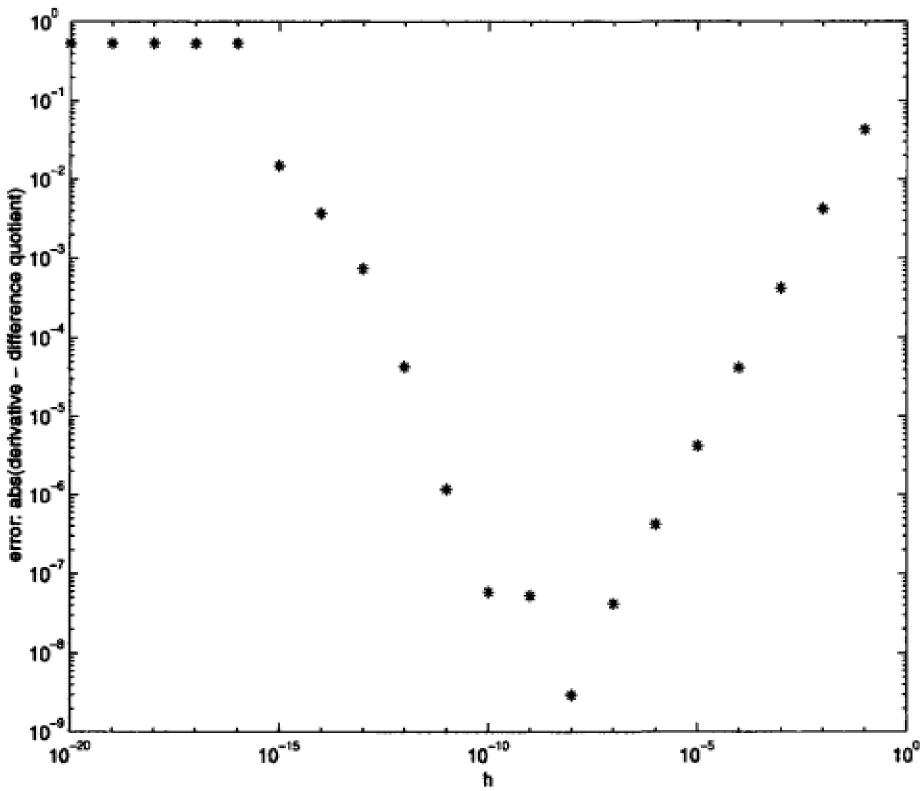

Figure 11.1: Error (Absolute Value of Derivative Minus Difference Quotient) as a Function of h (Log–Log Scale)

2-Resoudre l’exo 11.1

la ligne a remplacer dans le code (de la question precedente ci_joint) est diffquo = (sin(x+h)-sin(x))/h; par diffquo = (sin(x+h)-sin(x-h))/(2*h);vu qu’on est dans le cas des des différences centrées

```
#include<stdio.h>
#include<math.h>
/* Program 5: Difference Quotient*/

int main(){

        FILE *pipe = popen("gnuplot -persist","w");
        fprintf(pipe, "set logscale xy\n");
        fprintf(pipe, "set xrange [1e-20:1e-0]\n");
        fprintf(pipe, "set yrange [1e-9:1e-0]\n");
        fprintf(pipe, "set xlabel 'h'\n");
        fprintf(pipe, "set ylabel 'erreure :fabs(deriv-diffquo)'\n");
        FILE *temp=fopen("donnees.txt","w");
        int n = 1,i;
        double x = 1.0, h = 1.0, deriv=cos(x), diffquo, error;
        printf(" deriv =%13.6e \n", deriv);
        printf(" h      diffquo       abs(deriv - diffquo) \n");
        /* Let h range from 10~{-1} down to 10~{-20} */

        while(n <= 20) {

                h = h/10; /* h = 10~(-n) */
                diffquo = (sin(x+h)-sin(x))/h; /* difference quotient */
                error = fabs(deriv-diffquo);
                fprintf(temp,"%5.1e%13.6e\n",h,error);
                printf("%5.1e  %13.6e %13.6e \n", h, diffquo, error);
                n++;
                       }
               fprintf(pipe, "plot'donnees.txt'\n");
          fclose(temp);


 pclose(pipe);
return 0;
}
```

une fois executé on aura les valeurs dans la capture2 ci_joint aprés , d’aprés le tableau le **h** a choisir est celui qui devient trop petit mais suffisament grand pour qu’il ne pose pas probleme pour l’annulation pour notre cas c’est celui cadré en rouge .

```
lougani@lougani-Satellite-C660:~/Bureau/algo et calcul scientifique/17042020$ gcc c_dif.c -lm
lougani@lougani-Satellite-C660:~/Bureau/algo et calcul scientifique/17042020$ ./a.out
 deriv = 5.403023e-01
 h       diffquo        abs(deriv - diffquo)
1.0e-01   5.394023e-01  9.000537e-04
1.0e-02   5.402933e-01  9.004993e-06
1.0e-03   5.403022e-01  9.005045e-08
1.0e-04   5.403023e-01  9.004295e-10
1.0e-05   5.403023e-01  1.114087e-11
1.0e-06   5.403023e-01  2.771683e-11
1.0e-07   5.403023e-01  1.943278e-10
1.0e-08   5.403023e-01  2.581230e-09
1.0e-09   5.403023e-01  2.969885e-09
1.0e-10   5.403022e-01  5.848104e-08
1.0e-11   5.403011e-01  1.168704e-06
1.0e-12   5.402900e-01  1.227093e-05
1.0e-13   5.401235e-01  1.788044e-04
1.0e-14   5.440093e-01  3.706976e-03
1.0e-15   5.551115e-01  1.480921e-02
1.0e-16   5.551115e-01  1.480921e-02
1.0e-17   0.000000e+00  5.403023e-01
1.0e-18   0.000000e+00  5.403023e-01
1.0e-19   0.000000e+00  5.403023e-01
1.0e-20   0.000000e+00  5.403023e-01
```

Illustration 11: capture2

Donc :ce qui donne les meilleurs résultats c’est bien la methode des <u>differences centrées</u> et non pas <u>celle des diffrences quotients</u>