1 Exercices types, flottants

1. Afficher une table de multiplication demandée à l'utilisateur.
2. Déterminer le plus petit entier n tel que `(1.0+2^(-n))-1.0==0` en utilisant le type `float` et `double`. Attention `^` n'est pas la puissance en C, c'est le ou exclusif. Attention, utiliser `1.0f` pour coder 1.0 en flottants. En déduire le nombre de bits de la mantisse dans les 2 cas.
3. Résoudre par dichotomie $\cos(x) = x$ sur l'intervalle $[0, 1]$. Rappel : si f continue change de signe sur $[a, b]$, on peut déterminer une valeur approchée de r tel que $f(r) = 0$ en prenant c le milieu de a et b, si $f(c)f(a) < 0$ on remplace b par c sinon on remplace a par c. Attention, c peut être égal à a ou b avec le type `double`. Utiliser `#include<cmath>` pour la fonction `cos`.
4. Trouver une valeur de h optimale pour déterminer une valeur approchée de $f'(x)$ par les formules

$$\frac{f(x+h)-f(x)}{h}, \quad \frac{f(x+h)-f(x-h)}{2h}$$

   On testera avec f tel que $f(x)$ et $f'(x)$ soient de l'ordre de grandeur de 1, par exemple $f(x) = e^x$ en $x = 1.0$ et avec les types `float` et `double`.
   Proposer et tester des formules pour $f''(x)$.
5. Calculer des sommes partielles de séries à terme positif décroissant en commençant par le terme de plus grand ou de plus petit indice, comparer les résultats en `float` ou `double`, par exemple pour

$$\sum_{k=1}^{N} \frac{1}{k^2}, \quad \sum_{k=1}^{N} \frac{1}{k}, \ldots$$

2 Exercices fonctions/vector

On représentera un vecteur de réels par un `vector<double>`, une matrice par un vecteur de vecteurs de même taille. Pour lire un `vector<double>` depuis un fichier `data` du disque dur contenant le nombre d'éléments suivi par la liste des valeurs, (par exemple faire `emacs data &` puis entrer `3 1.01 2.3 4.7` et sauvegarder), vous pouvez vous inspirer du programme ci-dessous (à adapter avec des templates pour les matrices);

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

istream & operator >> (istream & is,vector<double> & v){
  int n; is >> n;
  v.resize(n);
  for (int i=0;i<n;i++)
    is >> v[i];
  return is;
}

ostream & operator << (ostream & os,const vector<double> & v){
  int n=v.size(); os << n << " ";
  for (int i=0;i<n;++i)
    os << v[i] << " ";
  return os;
}
```

```
int main(){
  ifstream fichier("data");
  vector<double> v; fichier >> v;
  cout << v << endl;
}
```

1. Écrire une fonction min qui calcule le minimum de 2 entiers, puis l'utiliser pour créer une fonction min qui calcule le minimum de 3 entiers.
   Écrire une fonction qui calcule le min d'une liste de réels.
2. Écrire une fonction qui prend en argument deux nombres a, b et remplace a par le plus petit et b par le plus grand.
3. Écrire une fonction prenant en argument un `vector<double>` et calculant les éléments statistiques : moyenne, écart-type, médiane, quartiles. On pourra utiliser `sort` pour trier le vecteur.
4. Écrire une fonction calculant un rationnel proche d'un réel en utilisant la décomposition en fraction continue.
5. Écrire une fonction qui calcule la somme d'une liste de réels.
   Pour améliorer la précision, on pourra tester l'algorithme suivant avec une (grande) liste ou en générant les éléments à sommer au fur et à mesure

   ```
   Somme(l un vector<double>)
     variables locales x,s,c
     s:=0.0; c:=0.0; // la somme et la correction
     pour x élément de l faire
       c += (x-((s+x)-s));
       s += x;
     fin pour;
     renvoyer s+c;
   ```

   Explication : c devrait valoir 0 sans erreurs d'arrondis, avec les erreurs d'arrondis, le premier calcul intervenant dans celui de c est $s + x$ qui contribuera en une erreur opposée à celui du calcul de $s+ = x$ à la ligne suivante, le 2ième calcul effectué $(s + x) - s$ donne une erreur absolue en $\varepsilon|x|$ au pire (car c'est une erreur relative par rapport à $(s + x) - s$), et la 3ième erreur d'arrondi est négligeable (puisque la somme vaut 0). On a donc une erreur absolue sur $s + c$ qui est au premier ordre au pire en $O(\varepsilon \sum |x_i|)$, bien meilleure que la majoration $\varepsilon((n-1)|x_1|+(n-2)|x_2|+...+|x_n||)$ calculée précédemment.
6. Écrire une fonction prenant en argument un `vector<double>` représentant les coefficients d'un polynôme P et un `double` représentant a et calculant $P(a)$, en utilisant la méthode de Horner.
   Bonus : en itérant, déterminer la liste des coefficients de $P(x + a)$.
7. Écrire une fonction calculant une valeur approchée de $\cos(x)$ sur $[-1, 1]$ en utilisant le développement de Taylor en 0 à l'ordre 12.
   Prolonger cette fonction sur $\mathbb{R}$ en utilisant les propriétés de la fonction cosinus.
8. Écrire une fonction récursive puis itérative pour calculer le PGCD de 2 entiers (`int`). Bonus : renvoyer les 3 entiers de l'identité de Bézout.
9. Écrire une fonction récursive calculant $a^n \pmod p$ (si $n = 0$ on renvoie 1, si $n = 1$ on renvoie a, si n est pair on renvoie le carré de $a^{n/2} \pmod p$ etc.). Bonus : le faire en itératif.
10. Écrire une fonction déterminant la position d'un nombre dans une liste triée de nombres par dichotomie. On renverra -1 si le nombre n'est pas trouvé.
11. Écrire une fonction de dichotomie qui prend en argument une fonction `double` donne `double` et les extremités de l'intervalle a, b. Tester.
12. Faire de même par la méthode du point fixe. Pour la méthode de Newton, il faut donner une deuxième fonction (la dérivée) ou en déterminer une estimation numérique, par exemple par la méthode de la sécante. Dans le cas d'un polynôme, on utilisera plutôt Horner deux fois.
13. Implémenter du calcul matriciel.