TP1. Mise en oeuvre d'une version générique de l'algorithme du recuit simulé

Introduction

L'objectif de ce TP est la mise en oeuvre (en C++) d'une version générique de l'algorithme du recuit simulé. La solution retenue doit être indépendante de la nature du problème d'optimisation à résoudre.

Remarque :

Dans cette approche, la généricité est assurée à l'aide de la notion de ***template*** et de la notion de ***pointeur de fonction***.

- **Template**

  La notion de *template* est propre au langage C++. Ce concept est, dans ce contexte, bien plus adéquat que la notion de fonction virtuelle. Cette dernière est, en effet, adaptée à représenter des listes *hétérogènes* (deux éléments consécutifs ne sont pas de même nature) qui n'apparaissent pas dans ce problème.
  A la complexité et au coût en temps de la solution "fonctions virtuelles", nous préférons donc la solution *template* plus simple et plus efficace ici.

- **Pointeur de fonction**

  Cette notion est propre au langage C. Elle permet de manipuler une fonction comme une variable de type simple. Un pointeur de fonction permet aussi de paramétrer une fonction par une fonction. Cette dernière propriété, capitale, a fait du langage C, un *Langage Orienté Objet* avant l'heure !

1. Description de l'algorithme du recuit simulé

Cet algorithme est une méthode très générale d'optimisation. Il a pour objectif de rechercher le minimum d'une fonction *coût*(*s*) où *s* décrit un domaine d'exploration *S* (l'ensemble des solutions possibles). La fonction *s* a *coût*(*s*) est bien sûr à valeurs réelles. La seule hypothèse sur *coût* est que *coût*(*s*) est défini pour tout *s* ∈ *S*. Aucune autre hypothèse n'est exigée de coût (continuité, dérivabilité, etc.) ni de l'espace *S* (fini, discret, etc.).

Analogie avec un système thermodynamique

L'algorithme est basé sur l'analogie entre la notion de coût et de niveau d'énergie d'un système mécanique.
Il exploite le fait qu'un système physique (mécanique, chimique, ...) évolue spontanément vers un niveau d'énergie minimale (entropie croissante). Exemple : la boite d'allumettes que l'on agite.
Plus précisément, il s'inspire directement d'un procédé de cristallographie (le recuit) qui permet d'obtenir un matériau très pur par refroidissement progressif. L'idée générale est de simuler le comportement d'un système mécanique (un ensemble d'atomes par exemple) qui se refroidit. Le système subit des déformations aléatoires (les mouvements des atomes) dont l'amplitude diminue progressivement avec l'abaissement de la température. Le principe thermodynamique d'entropie croissante garantit que le système évolue vers un niveau d'énergie minimale.

La solution *s* est ainsi interprétée comme un système mécanique et *coût*(*s*) comme le niveau d'énergie de *s*.

Idée générale

L'algorithme général consiste donc simplement à suivre les déformations aléatoires que subit *s* lors de l'abaissement progressif de la température *t*. Il est défini par deux boucles imbriquées. La boucle externe contrôle l'abaissement de la température et la boucle interne contrôle les déformations successives de *s*.Toutes les déformations de *s* qui abaissent son coût sont acceptées et celles qui augmentent son coût sont acceptées de façon aléatoire suivant une loi qui favorise les faibles augmentations de coût. La probabilité d'acceptation diminue aussi avec la baisse de la température *t*. L'algorithme est initialisé avec une solution quelconque arbitraire $s_{0,}$ une température initiale $t_{Ininitale}$ et une température finale t_{Finale} telles que $t_{Initiale} \geq t_{Finale} \geq 0$. L'algorithme se poursuit jusqu'à que l'une des deux conditions suivantes soit réalisée :

- la suite *s* soit stationnaire (cela signifie qu'un minimum local de la fonction coût a été atteint)
- $t = t_{Finale}$

L'algorithme est paramétré par les quantités et fonctions suivantes :

$t_{Initiale}$	:	température initiale
t_{Finale}	:	température finale (vérifiant $t_{Initiale} \geq t_{Finale} \geq 0$)
nombreTentativesMax	:	nombre maximal de tentatives de déformations de s par palier de température
nombreSuccèsMax	:	nombre maximal de déformations acceptées par palier de température
s_0	:	solution initiale (quelconque)
s a *coût*(*s*)	:	fonction d'évaluation de la pertinence de la solution, c'est la fonction dont on cherche le minimum.
s a *changementAléatoire*(*s*)	:	fonction qui déforme aléatoirement la solution *s*.
t a *succ*(*t*)	:	fonction d'évolution de la température t. *succ* doit être strictement décroissante.

L'algorithme gère bien sûr une variable s_{Best} qui représente à tout moment la meilleure solution trouvée.

Algorithme :

$t \neg t_{Initiale}$
$s \neg s_0$
$s_{Best} \neg s_0$

Tant que $t > t_{Finale}$ **faire**
    $nombreTentatives \neg 0$
    $nombreSuccès \neg 0$
    Tant que $nombreTentatives < nombreTentativesMax$ et
     $nombreSuccès < nombreSuccèsMax$ **faire**
        $nombreTentatives \neg nombreTentatives + 1$
        $s_{Précédente} \neg s$
        $s \neg changementAléatoire(s)$
        Si $coût(s) < coût(s_{Précédente})$ **alors**
            $nombreSuccès \neg nombreSuccès + 1$
            Si coût(s) < coût(s_{Best}) **alors**
                $s_{Best} \neg s$
            Fsi
        Sinon
            v $\neg$ tirage aléatoire dans [0,1]
            $\Delta_{coût} \neg coût(s) - coût(s_{Précédente})$ // $coût(s) - coût(s_{Précédente}) \geq 0$
            Si v < $e^{-\frac{\Delta coût}{t}}$ **alors**
                $nombreSuccès \neg nombreSuccès + 1$
            Sinon
                $s \neg s_{Précédente}$
            Fsi
        Fsi
    Ftq
    Si $nombreSuccès = 0$ **alors** on arrête tout, la solution est s_{Best} /* la suite est stationnaire*/
    Fsi
    $t \neg succ(t)$
Ftq

Critère de Métropolis

Le test v < $e^{-\frac{\Delta coût}{t}}$ est appelé critère de Métropolis. La quantité $e^{-\frac{\Delta coût}{t}}$ est toujours comprise entre 0 et 1. Elle s'approche de 0 si $\Delta_{coût}$ est grand ou si t est proche de 0. $t_{Initiale}$ et t_{Finale} doivent être choisies de telle sorte que la quantité $e^{-\frac{\Delta coût}{t}}$ soit proche de 1 au début d'exécution et proche de 0 en fin d'exécution.
Le recuit simulé est un algorithme **heuristique** car il ne fournit aucune garantie de résultat. Il est aussi dit **stochastique** (non déterministe) car il utilise des opérateurs aléatoires.

Exercice 1

Ecrire en C++ la fonction *recuitSimule*(...) qui recherche à l'aide de l'algorithme précedent, le minimum d'une fonction *coût*. L'espace d'exploration, noté *S*, est quelconque.

Utiliser les templates de fonctions et les pointeurs de fonctions pour obtenir la version la plus générale possible.

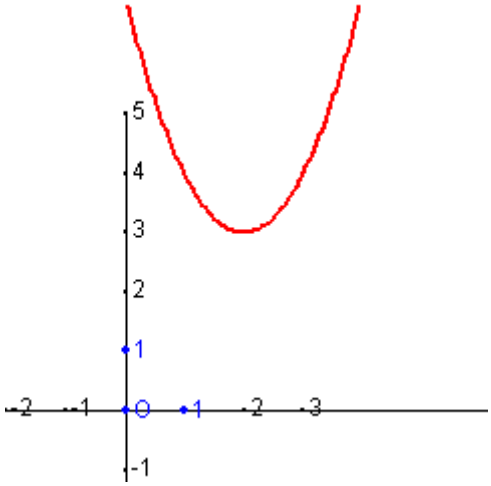
Attention ! Avec une solution template, il n'y a pas de fichier .cpp. Tout le code est écrit dans le fichier *recuitSimule.h*

Exercice 2. 1ère application en dimension 1

Appliquer la fonction *recuitSimule*() à la recherche du minimum de la fonction à une variable réelle :

$x \rightarrow f_1(x) = (x-2)^2 + 3$

La courbe de f_1 est une parabole dont l'allure est donnée par la figure suivante :



f_1 admet donc un minimum de 3 pour x = 2.

Exercice 3. 2ème application en dimension 1

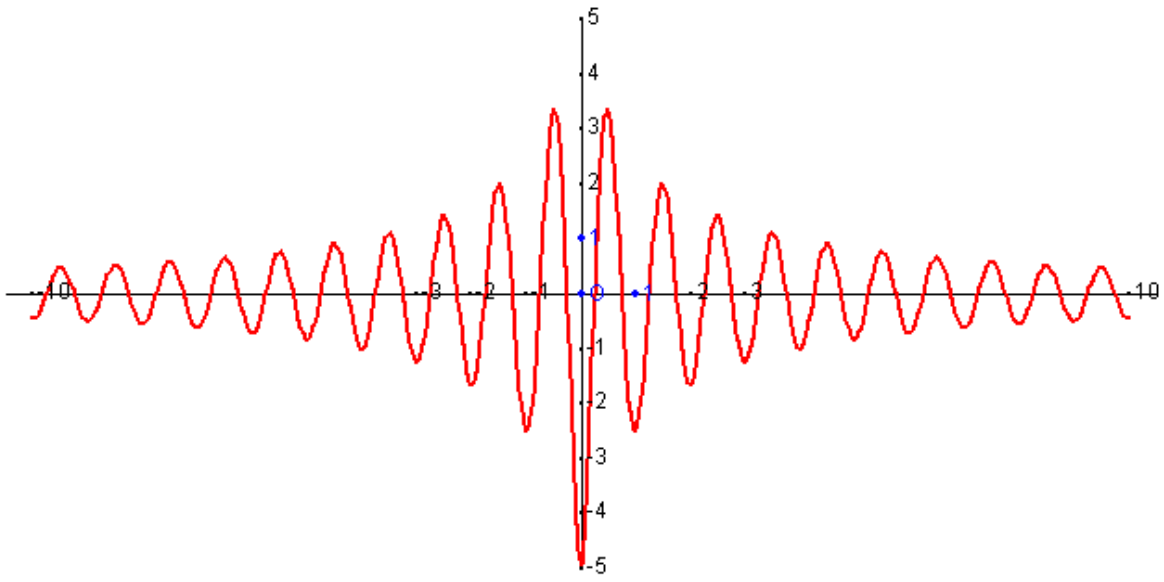
Appliquer la fonction *recuitSimule*() à la recherche du minimum de la fonction à une variable réelle :

$x \rightarrow f_2(x) = -5\frac{\cos(\omega x)}{1+|x|}$ avec $\omega = 2\pi$

La courbe de f_2 est une sinusoïde amortie dont l'allure est donnée par la figure suivante :

fonction sinusoide amortie d'équation t |--->A*cos(wt)/(k+|t|)

A = -5, w = 6.28319, k = 1



Exercice 4. 1ère application en dimension 2

Appliquer la fonction *recuitSimule*() à la recherche du minimum de la fonction à deux variables réelles :

(x,y) a $f_1(x,y) = (x-2)^2 + (y-2)^2 + 3$

La surface engendrée par $f_1$est un paraboloïde. f_1 admet un minimum global de valeur 3 pour $(x,y) = (2,2)$

Exercice 5. 2ème application en dimension 2

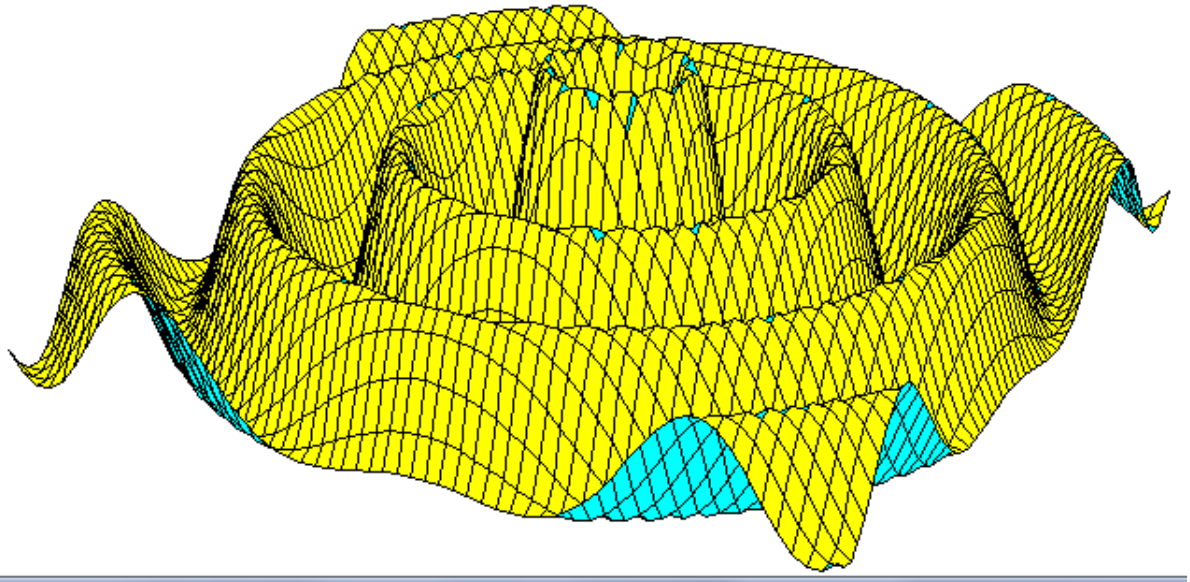
Appliquer la fonction *recuitSimule*() à la recherche du minimum de la fonction à deux variables réelles :

(x,y) a $f_2(x,y) = -2\dfrac{\cos(\omega r)}{1+r}$ avec $r = (x^2+y^2)^{1/2}$ et $\omega = 2\pi$

f_2 admet un minimum global de valeur -2 pour $(x,y) = (0,0)$. L'allure de la surface engendrée par f_2 est donnée par la figure suivante :

sinusoide spatiale vue de la position : 5, 2.5, 2.5

modélisation d'une sinusoide radiale à l'aide d'un B-rep à faces planes



Annexe

1. Classe Vecteur2D

```
/*
 * Vecteurs2D.h
 *
 *  Created on: 1 juil. 2011
 *      Author: Dominique
 */

#ifndef VECTEUR2D_H_
#define VECTEUR2D_H_

#include <iostream>
#include <string>
#include <cmath>

using namespace std;

class Vecteur2D
{
public:
double x,y;

inline explicit Vecteur2D(const double & x = 0, const double & y = 0);

inline const Vecteur2D operator + (const Vecteur2D & u) const;
inline const Vecteur2D operator * (const double & a) const;
inline const Vecteur2D operator - () const;
```

```
/**
 * produit scalaire
 * */
inline double operator * (const Vecteur2D & u) const;



inline const Vecteur2D & operator +=(const Vecteur2D & u);
inline const Vecteur2D & operator *=(const double & a);

operator string() const;

//------------------ Vecteur2D ---------------------------------
};

inline const Vecteur2D operator *(const double & a, const Vecteur2D & u) { return u*a;}

/**
 * calcule |u|_sup, c-à-d max(|x|,|y|)
 * */
inline double normeSup(const Vecteur2D& u);

ostream & operator << (ostream & os, const Vecteur2D & u);

void fusionne(const double x[], const double y[], int m, Vecteur2D v[]);

//----------------------------- implémentation des fonctions inline ---------------------------------

inline  Vecteur2D::
Vecteur2D(const double & x, const double & y): x(x),y(y){}

inline const Vecteur2D Vecteur2D::operator + (const Vecteur2D & u) const
{
return Vecteur2D( x+u.x, y+u.y);
}
inline const Vecteur2D Vecteur2D::operator * (const double & a) const
{
return Vecteur2D( x*a, y*a);
}
inline const Vecteur2D Vecteur2D::operator - () const
{
return Vecteur2D(-x,-y);
}
/**
 * produit scalaire
 * */
inline double Vecteur2D::operator * (const Vecteur2D & u) const
{
return x*u.x + y*u.y;
}

inline const Vecteur2D & Vecteur2D::operator +=(const Vecteur2D & u)
{
x+=u.x;
y+=u.y;
return *this;
}

inline const Vecteur2D & Vecteur2D::operator *=(const double & a)
{
x*=a;
y*=a;
return *this;
```

```
}

/**
 * calcule |u|_sup, c-à-d max(|x|,|y|)
 * */
inline double normeSup(const Vecteur2D& u)
{
return max(abs(u.x),abs(u.y));
}


#endif /* VECTEUR2D_H_ */
```

---------- Vecteur2D.cpp ----------

```
#include <sstream>
#include "Vecteur2D.h"

Vecteur2D::operator string() const
{
ostringstream oss;

oss <<"( "<< x <<", "<<y<<")";

return oss.str();
}


ostream & operator << (ostream & os, const Vecteur2D & u)
{
os << (string) u;
return os;
}


void fusionne(const double x[], const double y[], int m, Vecteur2D v[])
{
int i;
for (i = 0; i <= m; ++i)
  v[i] = Vecteur2D(x[i],y[i]);
}
```

2. AlgebreLineaire.h

```
/*
 * AlgebreLineaire.h
 *
 *  Created on: 30 juin 2011
 *      Author: Dominique
 */

#ifndef ALGEBRELINEAIRE_H_
#define ALGEBRELINEAIRE_H_

#include <math.h>
#include <iostream>
using namespace std;

template <class T>
inline const T operator - (const T & u, const T & v)
{
```

```
return u + -v;
}

/*
template <class T>
inline const T operator * (const double & a, const T & u)
{
return u * a;
}
*/

template <class T>
inline const T operator / (const T & u, const double & a)
{
return u * (1/a);
}

template <class T>
inline const T & operator *=(T & u, const T & v)
{
u = u * v;

return u;
}

template <class T>
inline const T & operator -= ( T & u, const T & v)
{
return u += -v;
}

template <class T>
inline const T & operator /= ( T & u, const double & a)
{
return u *= (1/a);
}

template <class T>
inline double norme(const T & u)
{
return sqrt(u*u);
}

template <class T>
inline double norme2(const T & u)
{
return (u*u);
}

/*
template <class T>
inline ostream & operator << (ostream & os, const T & x)
{
os << (string) x;
return os;
}
*/

#endif /* ALGEBRELINEAIRE_H_ */
```

Solution TP1

1. RecuitSimule.h

```
/**

Mise en oeuvre générique de l'algorithme du recuit simulé

*/
#pragma once

#include <float.h>
#include <math.h>
#include <stdlib.h>
#include <string>
#include <sstream>
#include <iostream>

using namespace std;
#define K 1     // constante pour le test de Métropolis, telle que K > 0
/**
réalise un tirage pseudo-aléatoire dans l'intervalle [a,b]

ATTENTION : Il faut appeler srand() au préalable !

*/
inline double tirageAleatoire(const double & a, const double &b)
{
const static int MAX(100);
const static double MAX1(99);

int n = rand() % MAX; // n est tiré au hasard tel que 0 <= n <= MAX-1

double x = n/MAX1; // x est un nombre réel au hasard entre 0 et 1 compris

return a + (b-a)*x;
}


/**

Représente une mise en oeuvre de l'algorithme heuristique "recuit simulé"

le but est de trouver le minimum de la fonction cout()

S est l'ensemble des solutions ou encore le domaine d'exploration

@param tInitiale : température initiale
@param tFinale : température finale telle que tFinale <= tInitiale
@param nombreTentativesMax : nombre maximal de tentatives par palier de température
@param nombreSuccesMax : nombre maximal de succès par palier de température
@param solutionInitiale : solution initiale du problème
@param fonction cout1(...) : permet d'évaluer la qualité d'une solution, plus le coût est faible,
meilleure est la solution
```

```
@param fonction changementAleatoire(...) : construit une nouvelle solution par perturbation aléatoire de la solution courante
@param fonction succ(...) : calcule la température suivante de la température courante. Doit vérifier succ(t) < t, quelque soit t

@return bestSolution : la meilleure solution trouvée
retourne aussi le coût de la meilleure soution grâce au paramètre coutBestSolution

*/
template <class S>
const S recuitSimule1(const double & tInitiale, const double & tFinale, const int nombreTentativesMax, const int nombreSuccesMax, const S & solutionInitiale,
        double (*cout1)(const S & solution), const S (* changementAleatoire)(const S & solution), double (*succ)(const double & temperature), double & coutBestSolution)
{
S solutionCourante, bestSolution;
double t; // température courante
double coutCourant;

for ( t = tInitiale, bestSolution = solutionCourante = solutionInitiale, coutBestSolution = coutCourant = cout1(solutionInitiale); t > tFinale; t = succ(t))
        {
        int nombreTentatives, nombreSucces;
        S solutionPrecedente; double coutPrecedent;

        for (nombreTentatives = nombreSucces = 0; nombreTentatives < nombreTentativesMax && nombreSucces < nombreSuccesMax; ++nombreTentatives)
                {

                solutionPrecedente = solutionCourante; coutPrecedent = coutCourant;
                solutionCourante = changementAleatoire(solutionCourante); coutCourant = cout1(solutionCourante);



                //cout<< "solution courante = " << solutionCourante<<", cout = " << coutCourant << endl;
                if (coutCourant < coutPrecedent)  // la solution courante est meilleure que la solution précédente
                  {
                  ++nombreSucces;
                  if (coutCourant < coutBestSolution)          // la solution courante est meilleure que la meilleure solution trouvée jusqu'à présent
                    {
                          bestSolution = solutionCourante; coutBestSolution = coutCourant;
                          }
                  }
                else     // coûtCourant >= coûtPrécédent. La solution courante n'est pas meilleure que la solution précédente
                  {
                  double v = tirageAleatoire(0,1);
                  double deltaCout = coutCourant - coutPrecedent; // on a deltaCout >= 0
                  double metropolis = exp(-K*deltaCout/t);

                  if (v < metropolis) ++ nombreSucces;                // la solution courante est acceptée bien que moins bonne que la précédente

                  else
                    { solutionCourante = solutionPrecedente; coutCourant = coutPrecedent;}
        // la solution courante est refusée

                  }      // coûtCourant >= coûtPrécédent.
```

```
                }       // for, boucle tentatives d'améliorations

        if (nombreSucces == 0) return bestSolution;         // l'algorithme est stationnaire : il a atteint un minimum, on arrête tout et on retourne la meilleure solution trouvée

        }       // for, boucle température

return bestSolution;
}


/**
pour optimiser le temps de calcul, il vaut mieux économiser les appels à la fct cout(). Aussi il vaut mieux stocker les coûts calculés.

Le plus simple est d'alors associer une solution particulière et son coût. C'est le but de cette classe

*/
template <class S>
class SolutionCout
{
public:
S solution;
double cout; // cout de solution

SolutionCout( const S & solution, double (*cout1)( const S & solution)):solution(solution), cout(cout1(solution)) {}

const SolutionCout<S> change( const S (*changementAleatoire) (const S & solution), double (*cout1) (const S & solution)) const;

operator string() const;
};

template <class S>
ostream & operator << (ostream & os, const SolutionCout<S> & solutionCout)
{
return os << (string)solutionCout;
}

template <class S>
const SolutionCout<S> SolutionCout<S>::change( const S (*changementAleatoire) (const S & solution), double (*cout1) (const S & solution)) const
{
return SolutionCout<S>(changementAleatoire(this->solution), cout1);
}

template <class S>
SolutionCout<S>::operator string() const
{
ostringstream oss;

oss << "( " << solution <<", " << cout << ")";
return oss.str();
}



/**
Dans cette version, une solution et son coût associé sont associés dans un objet SolutionCout

*/
```

```
template <class S>
const SolutionCout<S> recuitSimule(const double & tInitiale, const double & tFinale, const int
nombreTentativesMax, const int nombreSuccesMax, const S & solutionInitiale,
        double (*cout1)(const S & solution), const S (* changementAleatoire)(const S &
solution), double (*succ)(const double & temperature))
{
SolutionCout<S>  solutionCourante(solutionInitiale, cout1);
SolutionCout<S> bestSolution(solutionCourante);
double t; // température courante

for ( t = tInitiale; t > tFinale; t = succ(t))
        {
        int nombreTentatives, nombreSucces;

        for (nombreTentatives = nombreSucces = 0; nombreTentatives <
nombreTentativesMax && nombreSucces < nombreSuccesMax; ++nombreTentatives)
                {

                SolutionCout<S> solutionPrecedente(solutionCourante);
                solutionCourante = solutionCourante.change(changementAleatoire,cout1);



                //cout<< "solution courante = " << solutionCourante << endl;
                if (solutionCourante.cout < solutionPrecedente.cout)  // la solution courante est
meilleure que la solution précédente
                  {
                  ++nombreSucces;
                  if (solutionCourante.cout < bestSolution.cout)      // la solution courante est
meilleure que la meilleure solution trouvée jusqu'à présent
                          bestSolution = solutionCourante;
                  }
                else     // coûtCourant >= coûtPrécédent. La solution courante n'est pas
meilleure que la solution précédente
                  {
                  double v = tirageAleatoire(0,1);
                  double deltaCout = solutionCourante.cout - solutionPrecedente.cout; // on a
deltaCout >= 0
                  double metropolis = exp(-K*deltaCout/t);

                  if (v < metropolis) ++ nombreSucces;                // la solution courante est
acceptée bien que moins bonne que la précédente

                  else
                    solutionCourante = solutionPrecedente;  // la solution courante est refusée

                  }      // coûtCourant >= coûtPrécédent.

                }         // for, boucle tentatives d'améliorations

        if (nombreSucces == 0) return bestSolution;          // l'algorithme est stationnaire : il a
atteint un minimum, on arrête tout et on retourne la meilleure solution trouvée

        }        // for, boucle température

return bestSolution;
}
```

2. fonctionsmathvariees.h

```
/**
```

```
Quelques fonctions mathématiques candidates pour tester les performances de l'algo du
recuit simulé

*/
#pragma once

#include <math.h>
#include "AlgebreLineaire.h"
#include "MesMaths.h"
#include "Vecteur2D.h"

/**
fonction dont la courbe C est une parabole de sommet (2,3). (2,3) est le point de la courbe C
d'ordonnée la plus petite
*/
inline double f1(const double & x)
{
double t = x-2;
return t*t + 3; // f1(x) = (x-2)^2 + 3  donc le minimum est f1(2) = 3
}


/**
fonction dont la courbe C est une sinusoïde amortie de minimum -5 en 0 de pseudo période 1.
*/
inline double f2(const double & t)
{
static double A = 5;
static double w = 2*M_PI;

return -A*cos(w*t)/(1+abs(t));
}

/**
fonction à 2 variables (x,y) dont la surface  S est un paraboloïde de sommet (2,2,3). (2,2,3) est
le point de la surface S de cote la plus petite
*/
inline double f1(const Vecteur2D & u)
{
Vecteur2D t = u-Vecteur2D(2,2);
return t*t + 3; // f1(x,y) = (x-2)^2 + (y-2)^2 +  3  donc le minimum est f12(2,2) = 3
}


/**
fonction à 2 variables dont la surface est une onde plane radiale amortie de maximum 5 en
(0,0) de pseudo période 1.
*/
inline double f2(const Vecteur2D & u)
{
return f2(norme(u));
}
```

3. RecuitSimuleDimension1.h

```
/**

paramètres pour appliquer l'algo du recuit simulé à la recherche de minima de fonctions à une
variable réelle

*/
```

```
#ifndef RECUITSIMULEDIMENSION1_H
#define RECUITSIMULEDIMENSION1_H

//#pragma once
#include "RecuitSimule.h"



inline double succ(const double & temperature)
{
return temperature - 1;
}

inline const double changementAleatoire(const double & solution)
{
return solution + tirageAleatoire(-1, 1);        // doit-il y avoir un rapport entre la largeur du
domaine de recherche et l'amplitude de la perturbation aléatoire ?
}

#endif
```

4. TestRecuitSimuleDimension1.cpp

```
/*
teste l'algorithme du recuit simulé pour rechercher le minimum d'une fonction math réelle à 1
variable

Pour cette application, l'ensemble des solutions, S = R
*/

#include <stdlib.h>     /* pour accéder aux fonctions de génération de nombres pseudo-
aléatoires : srand, rand */
#include <time.h>
#include <iostream>
#include <math.h>
#include "MesMaths.h"
#include "RecuitSimule.h"
#include "fonctionsmathvariees.h"
#include "RecuitSimuleDimension1.h"

using namespace std;

/*
Le recuit simule est appliqué à la recherche des minima des 2 fonctions suivantes
*/

/*
parabole
*/
inline double cout1(const double & solution)
{
return f1(solution);
}

/*
sinusoïde amortie

*/
inline double cout2(const double & solution) // le minimum est -5 atteint en 0
{
return f2(solution);
```

```
}

//int main()
int main4()
{
char ch;

srand (time(NULL));              // initialisation du générateur de nombres pseudo-aléatoires

double  tInitiale = 1000;                // température initiale : affectation telle que
deltaCoutMax/tInitiale  ~= 0
double  tFinale = 0;                     // température finale : affectation arbitraire telle que
température finale <= température initiale
int nombreTentativesMax = 50;        // arbitraire : nombre max de tentatives par palier de
température
int nombreSuccesMax = 25;            // arbitraire : nombre max de succès par palier de
température
//double solutionInitiale = -rand() +   rand(); // tirage d'un nombre réel quelconque
double solutionInitiale = tirageAleatoire(-10,10); // tirage d'un nombre réel quelconque

double meilleureSolution1;
double coutMeilleureSolution1;

//const S recuitSimule(const double & tInitiale, const double & tFinale, const int
nombreTentativesMax, const int nombreSuccesMax, const S & solutionInitiale,
//         double (*cout)(const S & solution), const S (* changementAleatoire)(const S &
solution), double (*succ)(const double & temperature))

cout << "recherche du minimum d'une fonction à une variable par la méthode du recuit
simulé" << endl;
cin >> ch;
meilleureSolution1 = recuitSimule1( tInitiale, tFinale, nombreTentativesMax,
nombreSuccesMax, solutionInitiale,
          cout2, changementAleatoire, succ, coutMeilleureSolution1);

cout << "le minimum de la fonction est trouvé en : " << meilleureSolution1 <<", le minimum
vaut : "<< coutMeilleureSolution1 << endl;
cin >> ch;

SolutionCout<double> meilleureSolution2 = recuitSimule( tInitiale, tFinale,
nombreTentativesMax, nombreSuccesMax, solutionInitiale,
          cout2, changementAleatoire, succ);

cout << "meilleure solution et son coût : " << meilleureSolution2 << endl;

 cin >> ch;
return 0;
}
```

5.RecuitSimuleDimension2.h

```
/**
paramètres pour appliquer l'algo de recuit simulé à la recherche de minima de fonctions à 2
variables réelles

*/

#pragma once

#include "RecuitSimuleDimension1.h"

inline const Vecteur2D changementAleatoire(const Vecteur2D & solution)
```

```
{
return Vecteur2D( changementAleatoire( solution.x), changementAleatoire( solution.y));
}
```

6. TestRecuitSimuleDimension2.cpp

```
/*
teste l'algorithme du recuit simulé pour rechercher le minimum d'une fonction math réelle à 2 variables


Pour cette application, l'ensemble des solutions, S = R^2 = {(x,y) avec x dans R et y dans R}
*/

#include <stdlib.h>     /* srand, rand */
#include <time.h>
#include <iostream>
#include <math.h>
#include "MesMaths.h"
#include "AlgebreLineaire.h"
#include "Vecteur2D.h"
#include "RecuitSimule.h"
#include "fonctionsmathvariees.h"
#include "RecuitSimuleDimension2.h"

using namespace std;

/*
Le recuit simule est appliqué à la recherche des minimas  des 2 fonctions suivantes à 2 variables
*/

/**
paraboloïde de sommet (2,2,3)
*/
inline double cout1(const Vecteur2D & solution)
{
return f1(solution);
}

/**
onde circulaire amortie de minimum -5 en (0,0)

*/
inline double cout2(const Vecteur2D & solution) // le minimum est -5 atteint en (0,0)
{
return f2(solution);
}

int main()
//int main5()
{
char ch;
srand (time(NULL));             // initialisation du générateur de nombres pseudo-aléatoires

double  tInitiale = 1000;               // température initiale : affectation telle que deltaCoutMax/tInitiale  ~= 0
```

```
double  tFinale = 0;                    // température finale : affectation arbitraire telle que
température finale <= température initiale
int nombreTentativesMax = 100;      // arbitraire : nombre max de tentatives par palier de
température
int nombreSuccesMax = 50;           // arbitraire : nombre max de succès par palier de
température
//double solutionInitiale = -rand() +   rand(); // tirage d'un nombre réel quelconque
Vecteur2D solutionInitiale ( tirageAleatoire(-10,10), tirageAleatoire(-10,10)); // tirage d'un point
quelconque

Vecteur2D meilleureSolution;
double coutMeilleureSolution;

//const S recuitSimule(const double & tInitiale, const double & tFinale, const int
nombreTentativesMax, const int nombreSuccesMax, const S & solutionInitiale,
//        double (*cout)(const S & solution), const S (* changementAleatoire)(const S &
solution), double (*succ)(const double & temperature))
cout << "recherche du minimum d'une fonction à 2 variables réelles par la méthode du recuit
simulé" << endl;
cin >> ch;
meilleureSolution = recuitSimule1( tInitiale, tFinale, nombreTentativesMax, nombreSuccesMax,
solutionInitiale,
        cout1, changementAleatoire, succ, coutMeilleureSolution);

cout << "le minimum de la fonction est trouvé en : " << meilleureSolution <<", le minimum
vaut : "<< coutMeilleureSolution << endl;
cin >> ch;

SolutionCout<Vecteur2D> meilleureSolution2 = recuitSimule( tInitiale, tFinale,
nombreTentativesMax, nombreSuccesMax, solutionInitiale,
        cout1, changementAleatoire, succ);

cout << "meilleure solution et son coût : " << meilleureSolution2 << endl;


cin >> ch;
return 0;
}
```