Projet informatique, programmation orientée objet.

Méthode d'optimisation : *Simulated Annealing* ou le "recuit simulé".

Mode d'emploi.

Thibault Barnouin

23 mars 2020

Table des matières

1 Principe d'utilisation.

Dans ce mode d'emploi, j'explique l'utilisation du code d'optimisation dont la structure est décrite dans la *Rapport de Projet*.

De façon générale, pour optimiser un modèle sur un set de données il faudra le faire de la façon suivante :

1. Importer les données dans un objet de type Data,
2. Créer un objet de type d'un modèle choisi (noté ici ModeleChoisi) basé sur les données précédemment importées,
3. Créer un pointeur de type Model vers le modèle ainsi créé,
4. Créer un objet de type d'une méthode d'optimisation choisie (pour ce projet seul SimulatedAnnealing a été implémenté) basé sur le pointeur précédemment créé,
5. Initialiser, exécuter et finaliser la méthode d'optimisation.

2 Les méthodes spécifiques.

2.1 Méthodes associées aux objets de type Data.

Cet objet dispose de plusieurs constructeurs. Supposons que nous voulons créer un objet X de type Data :

- Data X crée l'ensemble vide nommé *Data_0*;
- Data X(Y) récupère l'ensemble et le nom de l'objet donné Y ;
- Data X("donnéesX") crée l'ensemble vide nommé *donnéesX* ;
- Data X(vector<vector<double» Tab, "donnéesX") crée l'ensemble Tab nommé *donnéesX*.

Il dispose aussi de méthodes de génération de données aléatoires :

- randSet(int N) crée un ensemble de N points uniformément tirés dans le carré [[0 ;1],[0 ;1]] ;
- randSet(int N, double xa, double xb, double ya, double yb) crée un ensemble de N points uniformément tirés dans le carré [[xa ;ya],[xb ;yb]] ;
- randSet(int N, int d, double mu) crée un ensemble de N points associés à un modèle polynomial aléatoire de degré d avec un bruit moyen mu.

X.fromFile("path") importe les données du fichier situé à path (ce fichier doit comporter les données sous forme de 2 colonnes de nombres flottants séparées d'un espace).

On peut récupérer et modifier ses attributs via les accesseurs et mutateurs :

- getData() retourne un ensemble de données, de type vector<vector<double» ;
- setData(vector<vector<double») modifie l'ensemble de données ;
- getName() retourne le nom de l'ensemble, de type string ;
- setName(string) modifie le nom de l'ensemble.

Pour l'export des données :

- displayData() affiche les données via une fenêtre gnuplot ;
- exportData() crée un fichier d'extension *.data* portant le nom de l'ensemble et contenant les points de l'ensemble (sous formes de 2 colonnes de nombres flottants séparées d'un espace).

2.2 Méthodes associées aux objets de type Model.

Cet objet dispose de plusieurs constructeurs. Supposons que nous voulons créer un objet A de type ModeleChoisi *(on verra en fin de cette section les différents modèles implémentés et leurs spécificités)* :

- ModeleChoisi A crée un modèle vide (les ensembles de données expérimentales et obtenues par les modèle sont vides) avec la fonction de coût associée au modèle ;
- ModeleChoisi A(B) récupère les attributs de l'objet donné B ;
- ModeleChoisi A(Data X) crée un modèle basé sur les données expérimentales *X*.

On peut modifier les attributs via les mutateurs :
- setModelData(Data) modifie l'ensemble de données du modèle ;
- setExpData(Data) modifie l'ensemble de données expérimentales ;
- setCost(CostFunction*) modifie la fonction du coût associée au modèle ;
- setParam(vector<double>) modifie les paramètres du modèle.

Pour récupérer les valeurs des attributs via les accesseurs :
- getSol() retourne l'ensemble de données du modèle, de type Data ;
- getParam() retourne les paramètres du modèle, de type vector<double> ;
- getCost() retourne le coût associé au modèle (comparaison des données expérimentales et modélisées), de type double ;
- getNeighbor() retourne les paramètres d'un modèle *voisin* du modèle courant, de type vector<double>.

Pour l'export des données :
- displayModel() affiche les données expérimentales et modélisées via une fenêtre gnuplot,
- exportModel() crée un fichier d'extension *.data* portant le nom du modèle et contenant les points expérimentaux et modélisés (sous formes de 4 colonnes de nombres flottants séparées d'un espace).

Les différents modèles implémentés et leurs spécificités :
- LinearApprox est un modèle linéaire, la fonction de coût associée est $\chi 2$ l'écart quadratique, les paramètres associés sont l'ordonnée à l'origine et la pente, un modèle *voisin* est un modèle pour lequel les paramètres courants ont été légèrement perturbés ;
- PolynomialApprox est un modèle polynomial, la fonction de coût associée est $\chi 2$ l'écart quadratique, les paramètres associés sont les coefficients du polynôme modélisateur, un modèle *voisin* est un modèle pour lequel les paramètres courants ont été légèrement perturbés ;
- TSP est un modèle du voyageur de commerce *(Traveling SalesPerson)*, la fonction de coût associé est la distance totale, les paramètres associés sont l'ordre de visite des villes, il existe plusieurs définitions du modèle *voisin* : la définition retenue modifie l'ordre des villes d'un sous ensemble du chemin parcouru par le modèle courant.

2.3 Méthodes associées aux objets de type OptimisationMethod.

Cet objet dispose de plusieurs constructeurs. Supposons que nous voulons créer une méthode d'optimisation M de type MethodeChoisie *(la seule méthode implémentée pour ce projet est celle dite du "recuit simulé" (Simulated Annealing) que l'on précisera par la suite).* :
- MethodeChoisie M est un constructeur spécifique à la méthode qui initialise des paramètres par défaut pour un modèle vide ;
- MethodeChoisie M(Model* A) construit la méthode et initialise des paramètres par défaut pour optimiser le modèle pointé par A.

Pour utiliser la méthode :
- initialise() initialise les paramètres à partir des valeurs entrées dans le ficher *header* correspondant à la méthode choisie ;
- execute() exécute la méthode d'optimisation ;
- finalise() exporte et affiche le modèle optimisé.

La seule méthode implémentée pour ce projet informatique est la méthode dite du "recuit simulé" *(Simulated Annealing)*. On l'appelle par SimulatedAnnealing, ses paramètres sont : l'amplitude des perturbations, la température initiale et la température finale du modèle, le taux de refroidissement, le nombre d'itération à une température constante et le nombre maximal de rejet de solutions avant de considérer le modèle gelé.

SimulatedAnnealing M(Model* A, double $ampl$, double T_{ini}, double T_{fin}, double τ_{froid}, int N_{isoT}, int m_{gel}) permet de construire la méthode avec les paramètres de notre choix sans passer par la modification des variables dans *SimulatedAnnealingParameters.h* (⚠ dans ce cas ne pas utiliser M.initialise()).

3 Exemples d'utilisation.

3.1 Approximation linéaire d'un modèle bruité

```
#include <stdlib.h>
#include <time.h>
#include "models.h"
#include "methods.h"

int main(){
        srand(time(NULL));

        //Créer un ensemble de données Exp1 nommé "Expérience1"
        Data Exp1("Experience1");
        //Générer 100 point aléatoirement suivant un modèle affine (polynomial de degré 1)
        Exp1.randSet(100,1,1.);
        //Créer un modèle linéaire basé sur les données expérimentales Exp1
        LinearApprox Lin1(Exp1);

        //Créer un pointeur de type Model vers le modèle linéaire
        Model* Mod1 = new LinearApprox(Lin1);

        //Créer la méthode d'optimisation pour le modèle linéaire
        SimulatedAnnealing Opti1(Mod1);
        Opti1.initialise();
        Opti1.execute();
        Opti1.finalise();

        free(Mod1);

        return 0;
}
```

3.2 Approximation polynomiale d'un modèle bruité

```
#include <stdlib.h>
#include <time.h>
#include "models.h"
#include "methods.h"

int main(){
        srand(time(NULL));

        //Créer un ensemble de données Exp2 nommé "Expérience2"
        Data Exp2("Experience2");
        //Générer 100 point aléatoirement suivant un modèle polynomial de degré 2
        Exp2.randSet(100,2,1.);
        //Créer un modèle polynomial de degré 2 basé sur les données expérimentales Exp2
        PolynomialApprox Pol2(Exp2,2);

        //Créer un pointeur de type Model vers le modèle polynomial
        Model* Mod2 = new PolynomialApprox(Pol2);

        //Créer la méthode d'optimisation pour le modèle polynomial
```

```
        SimulatedAnnealing Opti2(Mod2);
        Opti2.initialise();
        Opti2.execute();
        Opti2.finalise();

        free(Mod2);

        return 0;
}
```

3.3 Application au problème du voyageur de commerce

```
#include <stdlib.h>
#include <time.h>
#include "models.h"
#include "methods.h"

int main(){
        srand(time(NULL));

        //Créer un ensemble de données Villes nommé "30Villes"
        Data Villes("30Villes");
        //Générer 30 point distribué uniformément sur le carré [[0;1],[0;1]]
        Villes.randSet(30);
        //Créer un modèle du voyageur de commerce sur la base de la distribution Villes
        TSP TSP30(Villes);

        //Créer un pointeur de type Model vers le modèle du voyageur de commerce
        Model* Mod = new TSP(TSP30);

        //Créer la méthode d'optimisation pour le modèle du voyageur de commerce
        SimulatedAnnealing Opti(Mod);
        Opti.initialise();
        Opti.execute();
        Opti.finalise();

        free(Mod);

        return 0;
}
```

Commentaires supplémentaires.

Le code ainsi écrit permet facilement d'ajouter des modèles ou méthodes d'optimisation.

Pour la génération aléatoire de données, il pourrait être intéressant d'ajouter un méthode qui prend en argument un pointeur vers une fonction-modèle définie par l'utilisateur. De la même façon, un modèle basé sur une fonction définie par l'utilisateur pourrait aussi être pertinent.

On a pu le voir pour la méthode de "recuit simulé", l'efficacité d'optimisation de la méthode dépend très fortement du modèle étudié. Une optimisation des paramètres de la méthode par rapport au modèle serait alors une bonne façon d'optimiser l'optimisation!