Logiciels cartographiques et récepteurs GPS*

Notes d'usage

Bernard Perrot

(*)Récepteurs GNSS (GPS, Galileo, Glonas, Beidou,...)

v0.250205

Copyright © 2023+ Bernard Perrot

PUBLIÉ PAR : (*AUTOPUBLICATION*)

Licence : *(à venir...)*

Version : v0.250205

Table des matières

I Généralités et prérequis

I.1 À propos des dénivelés 5

À propos des dénivelés 5

I.1.1 Introduction 5

I.1.2 Le calcul du dénivelé cumulé 5

Index 14

Généralités et prérequis

I.1 À propos des dénivelés 5
À propos des dénivelés 5
I.1.1 Introduction 5
I.1.2 Le calcul du dénivelé cumulé 5

I.1 - À propos des dénivelés

I.1.1 Introduction

L'objet de cette note [1] est de présenter quelques méthodes (algorithmes) pouvant être utilisés pour effectuer les calculs de dénivelés positifs et négatifs indiqués dans les publications de la *FFRP*, afin de parvenir à une définition stricte et documentée de celle qui sera la référence pour les publications à venir. Cette définition pourra également être proposée aux partenaires afin que toutes les publications et applications numériques soient cohérente entre elles.

Il faut indiquer en préambule une difficulté majeure : contrairement à par exemple la distance totale d'un parcours qui est facilement appréhendable et définissable, il n'existe pas de définition précise de ce qu'est le « dénivelé cumulé » d'un parcours. En conséquence, le choix de la méthode et son paramétrage constitue en soi une définition sous forme de fait technique établi.

Si l'une des méthodes exposées ici est retenue, il conviendra de la documenter très précisément, la calibrer (fixer les paramétrages) et définir un jeu de tests (quelques traces typiques sur des terrains variés) afin que tous les acteurs puissent vérifier la conformité de leurs implémentations avant mise en production. C'est cet ensemble (algorithme, paramétrage, jeu de test de référence) qui constituera « notre » définition de la notion de dénivelé cumulé.

I.1.2 Le calcul du dénivelé cumulé

Il y a la méthode « brute », c'est à dire que l'on additionne toutes les variations positives (d'une part, et négatives d'autre part), le résultat donne le dénivelé que je qualifie de dénivelé « brut ». Celui-ci est proche du dénivelé « réel » en montagne, mais pour des randonnées typiques en région accidentées d'altitudes basses ou moyennes, on constate le plus souvent des valeurs qui peuvent sembler véritablement trop importantes.

Même avec des relevés d'altitudes identiques, il est fréquent (très fréquent...) d'obtenir des valeurs de dénivelés cumulés différentes, parfois de beaucoup (un

1. Le format de présentation de cette note provient du fait qu'elle a été rédigée à partir d'une annexe d'un ouvrage plus général (en cours de rédaction depuis deux ans...), en l'adaptant au propos de nos réunions. Cela explique en particulier la numérotation des paragraphes qui semble provenir de nulle part...

facteur deux n'est pas rare). La cause est que les logiciels utilisent des méthodes (algorithmes) différentes pour calculer ces dénivelés, dont certaines parfaitement inadéquates.

La plupart des logiciels calculent alors un dénivelé cumulé auquel est appliqué une méthode de filtrage (compensation) pour donner une valeur plus proche de la "réalité" (étant entendu qu'il n'existe pas de définition précise de cette réalité, sauf à monter une pente parfaitement régulière exactement du bas en haut). C'est de cette méthode de filtrage (et son paramétrage quand le logiciel permet de le faire à son goût) que proviennent les différences constatées d'un logiciel à l'autre.

Il existe plusieurs méthodes "simples" (car plus facile à mettre en œuvre), telles que l'élimination des variations faibles en dessous d'un certain seuil, lissage de courbe par calcul de moyenne, etc, mais qui ne sont pas satisfaisantes. Il est préférable d'opter pour un algorithme plus complexe, mais qui donne des résultats très satisfaisant. On peut simplifier les courbes d'altitudes avec la méthode de *Ramer-Douglas-Peucker* par exemple [2]. Ces méthodes ont l'avantage d'éliminer les petits écarts, sans émousser/écrêter les variations plus importantes ni les "pics" de montée/descente. Ces méthodes efficaces de part l'algorithme utilisé ne peuvent pas être appliquée en temps réel, elles nécessitent de disposer de la totalité de l'enregistrement pour être effectuée.

À noter qu'il y a une différence d'approche selon qu'il s'agisse de calculer le dénivelé cumulé d'un enregistrement réalisé avec un GPS, ou bien d'un parcours élaboré sur carte. Et dans le cas d'un parcours élaboré sur carte, celui-ci n'est pour autant pas « parfait », soit qu'il soit tracé « à main levée », soit qu'il utilise une base de données vectorielles des voies qui n'est pas forcément précise. En conséquence, dans tous les cas, il faudra appliquer une fonction de filtrage au procédé de calcul afin de diminuer l'influence du « bruit » affectant le « signal » altitude.

I.1.2.1 Méthode brute

Dans la méthode que j'appellerais « brute », on additionne toutes les variations positives (d'une part, et négatives d'autre part) : le résultat donne le dénivelé que je qualifie de dénivelé « brut ». Celui-ci est proche du dénivelé « réel » en montagne, mais pour des randonnées typiques en région accidentées d'altitudes basses ou moyennes, on constate le plus souvent des valeurs qui peuvent sembler véritablement trop importantes.

Voici l'algorithme de base pour calculer le dénivelé positif [3] d'une trace (ici, un tableau d'altitudes seules pour simplifier le code).

2. ou bien aussi celle de *Visvalingam*, mais plus complexe à mettre en œuvre sans apporter de différence significative par rapport à l'algorithme RDP.
3. seul le dénivelé positif est traité dans les descriptions d'algorithmes afin de ne pas alourdir le code, le traitement du dénivelé négatif est trivial à ajouter en usage réel.

En pseudo-code [4] :

```
fonction calculDenivelePositif(listeAltitudes)
// Initialiser le dénivelé positif à 0
denivelePositif <- 0

// Parcourir la liste des altitudes à partir du deuxième élément
pour i de 1 à taille(listeAltitudes) - 1 faire
 // Calculer la différence entre l'altitude actuelle et la précédente
 delta <- listeAltitudes[i] - listeAltitudes[i - 1]

 // Si la différence est positive, l'ajouter au dénivelé positif
 si delta > 0 alors
  denivelePositif <- denivelePositif + delta
 fin si
fin pour

// Retourner le dénivelé positif total
retourner denivelePositif
fin fonction
```

Cette méthode « brute » ne donne pas satisfaction : sur une trace enregistrée, elle accumule le bruit du signal, et sur une altimétrie issue d'un MNT, elle prend en compte les très petites variations ce qui a pour effet d'augmenter artificiellement les dénivelés. Elle peut cependant être utile et suffisante si on utilise un MNT avec un maillage très large, tels que la BD-Alti à 25m de l'IGN, ou bien les MNT générés à partir des données SRTM1 (une seconde d'arc) ou SRTM3 (trois secondes d'arc).

À noter en ce qui concerne les applications de la *FFRP*, et sauf mauvaise interprétation de ma part, c'est cette méthode (algorithme) qui est utilisé pour calculer les dénivelés dans le *PubliWeb*, ce qui expliquerait qu'ils apparaissent surévalués.

Pour améliorer cette méthode, il faut au minimum ajouter la notion de « seuil », permettant de ne cumuler que si celui-ci (exprimé en mètres) dépasse une valeur minimale en dessous de laquelle les variations sont ignorées.

En pseudo-code :

```
fonction calculDenivelePositif(listeAltitudes, seuil)
// Initialiser les variables
denivelePositif <- 0
deltaCumule <- 0 // Cumul des petites variations en dessous du seuil
enMontee <- faux // Indique si on est dans une phase de montée
```

4. Dans cette note, je tente de décrire les algorithmes en « pseudo-code », dont je ne suis pas spécialiste (il n'y a pas de syntaxe standardisée de pseudo-code...). Le but est avant tout de montrer que les implémentations sont simples et nécessitent relativement peu de développement. Il n'est pas impossible qu'une erreur subsiste dans l'expression de ces pseudo-codes qui par nature ne peuvent pas être testés. J'aurais été plus à l'aise avec un langage de programmation réel, mais qui lui ne serait pas lisible et compréhensible par tout le monde sauf à connaitre ce langage. Et cela n'aurait de toute façon pas d'utilité immédiate, les différentes applications fédérales n'étant pas programmées avec les même outils et langages.

```
// Parcourir la liste des altitudes à partir du deuxième élément
pour i de 1 à taille(listeAltitudes) - 1 faire
 // Calculer la différence entre l'altitude actuelle et la précédente
 delta <- listeAltitudes[i] - listeAltitudes[i - 1]

 si delta > 0 alors
 // Si la pente est positive
 si non enMontee alors
  // Si on n'est pas encore en montée
  deltaCumule <- deltaCumule + delta
  si deltaCumule > seuil alors
   // Si le cumul dépasse le seuil, on commence à accumuler
   denivelePositif <- denivelePositif + deltaCumule
   deltaCumule <- 0 // Réinitialiser le cumul
   enMontee <- vrai
  fin si
 sinon
  // Si on est déjà en montée, on accumule directement
  denivelePositif <- denivelePositif + delta
 fin si
 sinon
  // Si la pente est négative ou stable
  deltaCumule <- 0 // Réinitialiser le cumul
  enMontee <- faux // Sortir de la montée
 fin si
fin pour

// Retourner le dénivelé positif total
retourner denivelePositif
fin fonction
```

Le principe est le suivant : on ne décide d'accumuler les variations positives que si celle entre deux points successifs est supérieure au seuil prédéfini. Si cette variation est inférieure, on retient le sens de la pente, afin de retenir le cumul éventuel si on dépasse le seuil en restant en pente montante. Si la pente change de sens avant que ce seuil soit atteint, la variation est ignorée.

Cet algorithme permet d'éliminer les petites variations. En l'absence de l'utilisation d'un algorithme plus sophistiqué, la méthode peut être relativement satisfaisante s'il s'agit de traiter une altimétrie issue d'un MNT tel que le RGE-Alti de l'IGN.

À signaler qu'une erreur souvent rencontrée dans des implémentations consiste à ne pas cumuler les variations inférieures au seuil tant que la pente reste de même signe : cela a pour effet de minimiser le dénivelé d'une valeur égale à la valeur du seuil multipliée par le nombre de changement de sens significatif de la pente.

Cette méthode « brute » est la base pour les algorithmes plus sophistiqués qui vont suivre : ceux-ci vont consister à traiter l'altimétrie, et après traitement, c'est cette méthode « brute » qui sera utilisée pour le calcul final des dénivelés.

Utilisée seule, la méthode « brute » est utilisable en temps réel dans les enregistreurs étant donné qu'elle ne nécessite que la connaissance du point courant et du point précédent.

I.1.2.2 Moyenne mobile

L'algorithme utilisant la « moyenne mobile » (ou « moyenne glissante ») est très fréquemment utilisé par les applications de traitement de traces GPS. Parmi les raisons de ce « succès », il y a le fait que c'est une méthode qui peut traiter les données en temps réel, et peut donc être implémentée dans les assistants sportifs (montres, ...) et récepteurs GNSS pour afficher des statistiques en cours d'activité. Elle est également facile à coder et rapide sans consommer trop de ressources processeur (et donc de batterie) dans les matériels.

C'est un algorithme très utilisé dans le domaine des statistiques pour analyser des séries de données ordonnées. Physiquement, en traitement du signal par exemple, il se présente comme un filtre passe-bas. Il permet donc de lisser une courbe bruitée.

L'algorithme général est de la forme suivante (pseudo-code) :

```
fonction lisserAltitudesParMoyenneGlissante(listeAltitudes, tailleFenetre)
// Initialiser la liste des altitudes lissées
altitudesLisse <- listeVide()
demiFenetre <- partieEntiere(tailleFenetre / 2)

// Parcourir chaque élément de la liste des altitudes
pour i de 0 à taille(listeAltitudes) - 1 faire
 somme <- 0
 count <- 0

 // Calculer la somme des altitudes dans la fenêtre centrée sur ‘i‘
 pour j de i - demiFenetre à i + demiFenetre faire
  si j >= 0 et j < taille(listeAltitudes) alors
   somme <- somme + listeAltitudes[j]
   count <- count + 1
  fin si
 fin pour

 // Calculer la moyenne pour le point ‘i‘ et l’ajouter à la liste lissée
 ajouter (somme / count) à altitudesLisse
fin pour

// Retourner la liste des altitudes lissées
retourner altitudesLisse
fin fonction
```

Chaque point est recalculé en faisant la moyenne avec ses voisins. La largeur de la « fenêtre » détermine le nombre de voisins pris en compte. Après avoir obtenu un nouvel ensemble de données altimétriques lissées, il suffit d'appliquer l'algorithme de calcul brut, avec ou sans seuil (si l'altimétrie de départ est celle d'un MNT, le

seuil peut ne pas être utile, si ce sont des données d'enregistrement sur le terrain, un seuil léger peut se révéler pertinent).

Cependant, dans le cas qui nous intéresse, cet usage de la moyenne mobile n'est pas correct car il n'a pas de sens physique précis [5] : il ne faut pas utiliser une fenêtre en nombre de points, mais en distance afin que cette fenêtre soit constante et ai un sens relatif comparée à l'altitude.

L'algorithme devient alors [6] (en pseudo-code) :

```
fonction
lisserAltitudesParMoyenneGlissante(listeAltitudes,listeDistances,largeurFenetre)

// Initialiser la liste des altitudes lissées
altitudesLisse <- listeVide()

// Parcourir chaque altitude
pour i de 0 à taille(listeAltitudes) - 1 faire
 sommeAltitudes <- 0
 count <- 0

 // Parcourir tous les points pour vérifier ceux dans la fenêtre centrée sur ‘i‘
 pour j de 0 à taille(listeAltitudes) - 1 faire
  distanceRelative <- valeurAbsolue(listeDistances[j] - listeDistances[i])

  si distanceRelative <= largeurFenetre / 2 alors
   sommeAltitudes <- sommeAltitudes + listeAltitudes[j]
   count <- count + 1
  fin si
 fin pour

 // Calculer la moyenne des altitudes dans la fenêtre
 // et l'ajouter à la liste lissée
 ajouter (sommeAltitudes / count) à altitudesLisse
fin pour

// Retourner la liste des altitudes lissées
retourner altitudesLisse
fin fonction
```

De la même façon qu'il est indiqué précédemment, le tableau des altitudes lissées est utilisé par l'algorithme « brut » pour calculer les dénivelés.

---

5. Il y a exception si les points sont équidistants d'une quantité paramétrable, mais ce cas ne se présente pas sur un tracé vectorisé sur la base *BDrando*.
6. Ici, j'utilise deux tableaux en entrée, un contenant les altitudes des points, et un contenant les distance au précédent. Cela permet une formulation plus lisible de l'algorithme en évitant les calculs de projections à partir de coordonnées géographiques. Dans une implémentation réelle, l'argument d'entrée serait un tableau de coordonnées 3D.

Cette méthode donne des résultats qui peuvent être plus ou moins satisfaisants, la condition primordiale étant de correctement fixer la largeur de la fenêtre.

Il existe d'autre méthodes relevant du même domaine des mathématiques statistiques ou traitement de signal qui seraient envisageables, comme par exemple un filtre de *Kalman*. Mais cela n'apporterait pas d'avantages déterminant au prix d'une mise en œuvre plus complexe.

I.1.2.3 Algorithme de Ramer-Douglas-Peucker

La majorité (malheureusement...) des applications de traitement de données GPS qui simplifient et/ou filtrent les traces utilisent un algorithme de type statistique tel que la moyenne mobile évoquée ci-dessus. Cela a pour effet principal créer une trace qui ne passe plus par les « sommets » du tracé, en particulier, les virages sont coupés. Si un tel algorithme est satisfaisant pour des parcours « monotones », ce n'est pas le cas des parcours effectués par un randonneur, qui ne sont pas facilement modélisables. En montagne par exemple, avec des lacets très courts, le résultat est une déformation importante du parcours (qui à l'extrême peut entrainer un sous-évaluation de la distance).

Il existe un algorithme très performant pour cette tâche de simplification qui est l'algorithme de *Ramer-Douglas-Peucker* (RDP). Cet algorithme est très bien décrit sur la page Wikipédia qui lui est consacrée :
https://fr.wikipedia.org/wiki/Algorithme_de_Douglas-Peucker [i].

Ma proposition serait d'utiliser également cet algorithme pour les calculs de dénivelés. La raison principale est qu'en y réfléchissant bien, qu'est-ce qu'un dénivelé cumulé si ce n'est une distance parcourue dans la direction verticale ? En conséquence, un algorithme performant pour traiter une trace au sol doit l'être pour traiter un « parcours » vertical. La courbe (tableau) à simplifier définie par un ensemble de points avec la distance à l'origine en abscisse et l'altitude en ordonnée.

L'algorithme RDP est un algorithme récursif qui nécessite l'ensemble des points pour être appliqué. Cela le destine plutôt à des traitements différés, en post-production. Si on voulait l'appliquer en temps-réel, il faudrait à chaque point refaire le calcul complet, calcul de plus en plus long au fur et à mesure que la trace s'allonge. C'est peu envisageable dans des applications embarquées, c'est pourquoi la plupart utilisent alors la moyenne glissante en temps-réel.

Voici une description en pseudo-code (en supposant le tableau déjà constitué à partir des coordonnées 3D des points de trace) :

```
fonction distancePointLigne(point, p1, p2)
 x1 <- p1.distance
 y1 <- p1.altitude
 x2 <- p2.distance
 y2 <- p2.altitude
 x0 <- point.distance
 y0 <- point.altitude
```

i. https://fr.wikipedia.org/wiki/Algorithme_de_Douglas-Peucker

```
 // Calcul de la distance point-segment
 numerateur <- valeurAbsolue((y2 - y1) * x0 - (x2 - x1) * y0 + x2 * y1 - y2 * x1)
 denominateur <- racineCarree((y2 - y1)^2 + (x2 - x1)^2)

 retourner numerateur / denominateur
fin fonction

fonction ramerDouglasPeucker(points, tolerance)
 // Trouver le point le plus éloigné du segment reliant les extrémités
 maxDist <- 0
 indexPlusLoin <- 0

 pour i de 1 à taille(points) - 2 faire
  dist <- distancePointLigne(points[i], points[0], points[taille(points) - 1])
  si dist > maxDist alors
   maxDist <- dist
   indexPlusLoin <- i
  fin si
 fin pour

 // Si la distance maximale est inférieure à la tolérance
 si maxDist < tolerance alors
  // Garder uniquement les extrémités
  retourner [points[0], points[taille(points) - 1]]
 sinon
  // Diviser la courbe en deux segments
  premiereMoitie <-
   ramerDouglasPeucker(sousListe(points, 0, indexPlusLoin + 1), tolerance)
  deuxiemeMoitie <-
   ramerDouglasPeucker(sousListe(points,indexPlusLoin,taille(points)),tolerance)

  // Combiner les deux segments en évitant les doublons
  retourner concatener(premiereMoitie[0 à taille(premiereMoitie)-1],deuxiemeMoitie)
 fin si
fin fonction
```

Contrairement à la méthode par moyenne mobile, le tableau de sortie est de taille inférieure à celui d'entrée. Cela permet toujours de calculer les dénivelés positifs et négatifs, mais ce nouveau tableau ne permet pas de substituer les altitudes dans la trace d'origine (sauf à interpoler, mais cela ne présente pas d'intérêt, le but étant de calculer des dénivelés).

La valeur de la tolérance est à déterminer. D'expérience empirique, trois mètres me semble donner le résultat le plus satisfaisant (filtrage faible à moyen) avec une altimétrie MNT ou barométrique.

Une caractéristique très important de cette méthode est que si on ré-applique l'algorithme sur lui même avec la même contrainte, le résultat est invariant. Ce qui n'est pas les cas avec la moyenne mobile qui ré-appliquée à elle-même tend vers une ligne droite (donc des dénivelés nuls dans le cas d'une boucle !). De ce point de vue, elle est donc mathématiquement (et intellectuellement) plus satisfaisante.

D'autre part, tout comme lorsque l'on utilise cet algorithme RDP pour simplifier une trace au sol avec conservation des sommets des changements de direction, appliqué à l'altimétrie, les points bas et points hauts sont conservés, ce qui n'est pas le cas avec la moyenne glissante.

I.1.2.4 Implémentation

La facilité à coder ces algorithmes est décroissante dans l'ordre de présentation ici. Mais dans tous les cas, l'implémentation est aisée pour un développeur compétent et le code est relativement court (quelques dizaines de lignes de code tout au plus). Concernant le *PubliWeb*, à partir du moment où il est décidé de modifier le code actuel, le choix d'un algorithme plutôt qu'un autre ne devrait pas influer notablement sur la difficulté à développer le code proprement dit. Les phases de tests et validation seront les mêmes quelque soit le choix fait. En conséquence, ce choix ne devrait être dicté que par la qualité, l'adéquation au besoin, et les performances attendues.

Bernard Perrot, le 5 février 2025

Ce livre est une collection de notes et notices personnelles concernant mon usage des récepteurs et logiciels GPS.

Copyright © 2022+ Bernard Perrot
PUBLIÉ PAR : (*AUTOPUBLICATION*)
Licence *(à l'étude...)*