

# Projet: Pedestrian Dead Reckoning for Android

Navigation à l'estime pour piéton pour Android

Thibaud Michel

Université Joseph Fourier

Laboratoire d'Informatique de Grenoble

29 octobre 2014

## 1 Introduction

Un des problèmes principaux de la localisation sur mobile aujourd'hui est le suivi de la personne là où le signal GPS ne fonctionne pas. C'est notamment le cas à l'intérieur des bâtiments ou des zones couvertes (tunnels, grottes...). Nous aimerions par exemple connaître la position d'une personne à l'intérieur d'un musée pour lui proposer du contenu spécifique, ou encore proposer à un mal voyant un chemin en fonction de l'endroit où il se trouve et de là où il va. Pour répondre à ce problème, nous avons décidé de ne pas ajouter de nouvelles infrastructures dédiées à la localisation comme les iBeacons, émetteur/récepteur radio, WiFi. Ces systèmes là sont utilisés quand la précision est la priorité de l'application. Ce projet s'orientera vers une autre approche qui se nomme Pedestrian Dead Reckoning (ou navigation à l'estime pour piéton en français). C'est une méthode de navigation qui consiste à déduire la position de la personne en fonction de la distance parcourue depuis sa dernière position connue.

## 2 GPX Viewer (6h)

Pour commencer, nous réaliserons un visualiseur de document GPX afin d'afficher nos futures traces réalisées avec le PDR. Le GPX est un format de fichier permettant l'échange de coordonnées géolocalisées sous forme de point de cheminement (waypoint), trace (track) ou itinéraire (route).

### 2.1 GPX Structure (1h)

Le document GPX a [son propre schéma XML](#), mais dans ce projet nous ne nous intéresserons qu'à la partie "Track" du format.

**Question 2.1** Dans un package nommé *gpx*, créez une structure java du document GPX qui correspond aux tags *trk*, *trkseg*, *trkpt* et *ele*.

### 2.2 GPX Parser (2h)

Pour remplir cette structure GPX, vous allez réaliser un parser. Il existe plusieurs façons de parser du XML sous Android, le [XMLPullParser](#) est relativement simple à utiliser, je vous le conseille. Afin de tester votre parser pour la question suivante, vous pourrez faire vos tests grâce aux fichiers *run.gpx* et *bike.gpx*.

**Question 2.2** Réalisez un parser qui prend en entrée un [InputStream](#) et en sortie votre [structure GPX](#).

### 2.3 Trace Viewer (3h)

Le meilleur moyen de visualiser une trace est de l'afficher au dessus d'un fond de carte. Sur Android, Google nous propose un accès facile à son API Maps, je vous conseille de l'utiliser. La première installation est un peu fastidieuse (surtout sous Eclipse) mais l'utilisation de la librairie reste assez facile. Voici le lien vers la [documentation](#) pour la mise en place. Libre à vous d'utiliser un autre système de cartographie si vous préférez ([MapsForge](#), [MapBox](#))

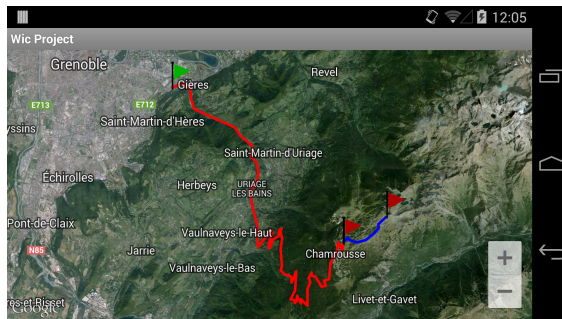


FIGURE 1 – Exemple d'un Viewer GPX

**Question 2.3** Affichez une vue ou un fragment avec un fond de carte centré sur Grenoble.

**Question 2.4** Créez une méthode qui prend en entrée votre [structure GPX](#) et qui affiche les traces en surcouche du fond de carte google grâce aux [polylines](#).

**Question 2.5** (Optionnel) Centrez la carte de manière à afficher toute les traces du document GPX à l'écran. Pour cela vous pourrez utiliser [CameraUpdateFactory.newLatLngBounds\(\)](#). Pour vous aider, créez des méthodes [getLatLngBounds\(\)](#) dans les classes *TrackSegment*, *Track* et *GPS*. Vous pourrez utiliser [LatLngBounds.Builder](#) pour cela.

**Question 2.6** En utilisant les [intent-filter](#), votre visualiseur sera capable de prendre en entrée les fichiers de type GPX.

### 3 Pedestrian Dead Reckoning (8h)

Dans cette partie, nous allons utiliser les différents capteurs du téléphone pour estimer la position de la personne. Le PDR (ou navigation à l'estime) est un système basé sur le positionnement relatif, c'est à dire que la précision de la nouvelle position dépend aussi de la précision de l'ancienne, contrairement au GPS qui fournit des positions (latitude et longitude) qui sont absolues. La navigation à l'estime est simplement basée sur la formule suivante, la seule difficulté est de trouver une valeur pour les différents paramètres.

Nouvelle position = calcul\_nouvelle\_position(Position courante, Taille du pas, Angle du pas)

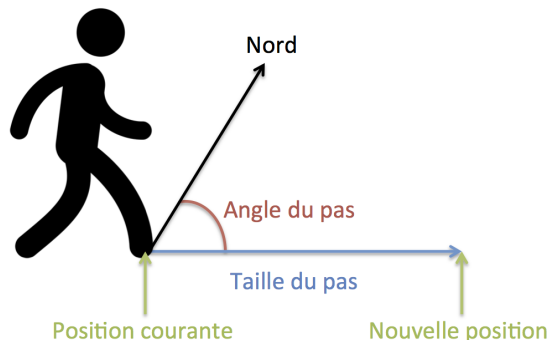


FIGURE 2 – Principe de la navigation à l'estime

**Question 3.1** Créez un package *pdr* qui servira pour cette partie

#### 3.1 Step Detection (3h)

Nous commencerons par faire un système de détection de pas que l'on appelle plus communément un podomètre. Le moyen le plus simple de monitorer le déplacement d'une personne est d'utiliser l'accélération linéaire du téléphone. Un premier problème se pose, selon l'endroit où l'on place le téléphone (main, poche, ...) les données récupérées par les capteurs sont différentes. Il faudrait analyser le modèle de marche d'une personne pour bien comprendre le problème. Pour ceux qui seraient intéressés je vous conseille d'aller lire la [Thèse de Q. Ladetto](#). Pour le projet nous en retiendrons surtout le modèle de marche suivant.

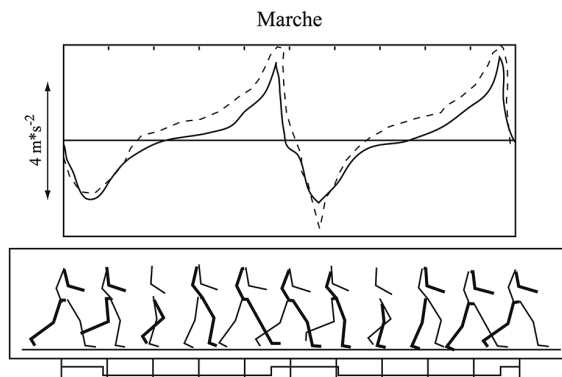


FIGURE 3 – Accélération linéaire en Y du centre de masse (courbe pleine) et la même accélération perçue par le smartphone (courbe discontinue) tenu dans la main

Afin de mieux comprendre les capteurs sous Android, je vous recommande de télécharger l'application [Sensor Kinectics](#) et de regarder ce qu'il se passe au niveau de l'accélération linéaire quand vous marchez. Regardez principalement l'axe Y sur lequel nous allons travailler.

**Question 3.2** Créez une classe `StepDetectionHandler` dont le constructeur prend en paramètre un `SensorManager` et 2 méthodes `start()` et `stop()` qui s'abonneront/désabonneront au "capteur" d'accélération linéaire.

**Question 3.3** Dans la méthode `SensorListener.onSensorChanged()` extraire l'accélération linéaire en Y. Fixez un seuil au dessus duquel on sera quasiment certain qu'une personne est entrain de faire un pas, vous pourrez vous aider de l'application Kinectics précédemment téléchargée. Détectez le moment où l'accélération linéaire passe sous ce seuil, on considérera qu'à ce moment là, la personne a fait un pas.

**Question 3.4** (Optionnel) La valeur de l'accélération linéaire en Y peut-être affinée en établissant une moyenne sur les 5 dernières valeurs par exemple.

**Question 3.5** Toujours dans la classe `StepDetectionHandler`, créez un listener `StepDetectionListener` avec la méthode : `void newStep(float stepSize)`. Implémentez le système d'abonnement associé au listener. On considérera que la taille du pas est de 0.8m.

## 3.2 Device Attitude (1h)

Nous allons maintenant essayer de déterminer l'angle du pas effectué. Cet angle va être calculé par rapport au Nord magnétique de la terre, c'est à dire que si le téléphone est orienté en direction du Nord magnétique, la valeur retournée est  $0^\circ$ , à l'Est  $90^\circ$ ... Idéalement il faudrait utiliser le [Magnétomètre](#) du téléphone, cependant il n'est pas fiable à l'intérieur des bâtiments à cause des variations magnétiques produites par les différentes machines. On pourrait aussi utiliser le [Gyroscope](#) qui permet à l'aide d'une intégration de connaître la rotation du téléphone sur un temps donné, malheureusement cette valeur est relative à la position de départ du téléphone. De plus, elle n'est pas fiable dans le temps (drift). La meilleure solution est de fusionner les capteurs, plusieurs algorithmes existent déjà et sont assez complexe. Afin de ne pas les recréer, Android nous propose d'utiliser [des capteurs composites](#).

**Question 3.6** Créez une classe `DeviceAttitudeHandle` de la même manière dont vous avez utilisé l'accéléromètre dans la partie précédente (toujours avec les méthodes `start()` et `stop()`). Utilisez cette fois-ci le capteur [Rotation Vector](#).

Le problème est que les valeurs retournées par `event.values` dans `SensorListener.onSensorChanged()` ne sont pas exploitables directement pour ce que nous voulons faire. Il va falloir qu'on change de repère et qu'on récupère le vecteur de rotation comme sur la figure ci-dessous.

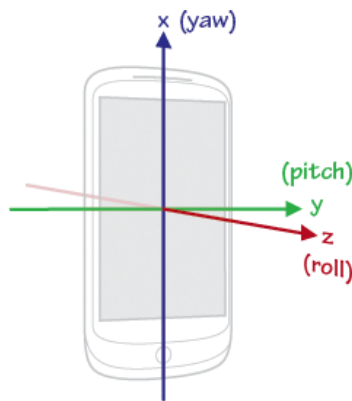


FIGURE 4 – Orientation du Smartphone sur les axes yaw, pith et roll

Voici une petite explication sur ces 3 valeurs, les unités sont en radians :

- **Yaw**  $[-\frac{\pi}{2}; -\frac{\pi}{2}[$  correspond aussi à ce qu'on appelle l'azimuth ou le bearing. C'est la détermination de l'angle que fait, dans le plan horizontal, la ligne du téléphone vers le nord géographique. 0 radian correspond au Nord,  $\frac{\pi}{4}$  radian à l'Est... C'est cette valeur que l'on utilisera pour le PDR.
- **Pitch**  $[-\frac{\pi}{4}; -\frac{\pi}{4}[$ . On l'utilise surtout en réalité augmentée pour afficher des objets selon leur hauteur ou altitude.
- **Roll**  $[-\frac{\pi}{2}; -\frac{\pi}{2}[$  correspond à la rotation du téléphone quand l'écran est toujours face à vous.

**Question 3.7** Le but de ce projet n'est pas de rentrer dans les détails. Aidez vous de [ce post](#) afin de passer des valeurs brutes du capteur *Rotation Vector* au vecteur défini par le schéma précédent. Vérifiez les valeurs de yaw.

**Question 3.8** Créez un getter pour le vecteur défini par le schéma.

### 3.3 Step Positioning (1h)

Maintenant que nous avons la taille et l'angle du pas il ne nous reste plus qu'à calculer la nouvelle position en fonction de la position courante. C'est à dire **calcul\_nouvelle\_position** dans la formule suivante :

Nouvelle position = calcul\_nouvelle\_position(Position courante, Taille du pas, Angle du pas)

Si nous nous plaçons dans un repère orthonormé il est facile de calculer la nouvelle position.

Nouvelle position x = Position courante x + Taille du pas \* cos(Angle du pas)

Nouvelle position y = Position courante y + Taille du pas \* sin(Angle du pas)

Cependant, ce n'est pas nouveau mais la terre n'est pas plate et le système de latitude/longitude que l'on utilise peut être assimilé à une sphère ou encore une ellipsoïde.

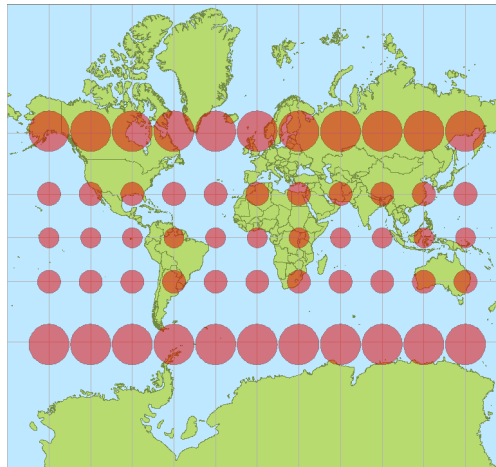


FIGURE 5 – Distortion de la projection de Mercator. Tous les cercles rouges représentent la même aire à la surface de la terre.

**Question 3.9** Créez une classe *StepPositioningHandler* avec un attribut *mCurrentLocation* ainsi que les getter et setter associés.

**Question 3.10** Créez une méthode *computeNextStep(float stepSize, float bearing)* qui prend en entrée la taille et l'angle du pas puis qui retourne la nouvelle position. [Ce document](#) vous aidera à calculer le point de destination en utilisant un arc du cercle.

### 3.4 Google Map Tracer (1h)

Afin de représenter la trace de la personne à l'écran en temps réel, nous allons ré-utiliser GoogleMap comme à la partie précédente. Cette fois-ci nous n'allons pas utiliser le format GPX mais simplement créer une [polyline](#) qu'on mettra à jour à chaque nouvelle position.

**Question 3.11** Créez une classe *GoogleMapTracer* dans le package *viewer* dont le constructeur prend en paramètre *GoogleMap*.

**Question 3.12** Créez une méthode *newSegment()* qui commencera une nouvelle polyline et une méthode *newPoint(LatLng point)* qui affichera un point supplémentaire à la polyline courante.

**Question 3.13** (Optionnel) Ajoutez un marker au début et à la fin d'une polyline.

### 3.5 PDR Application (2h)

Vous avez tout en main pour faire l'application qui visualise le déplacement d'un piéton. Il ne reste plus qu'à réutiliser tout ce qui a été fait précédemment.

**Question 3.14** Commencez par créer une nouvelle Activity. On y affichera la carte GoogleMap en plein écran comme on l'a fait pour le [GPX Viewer](#).

Depuis le début il manque une donnée important que l'on n'a pas définie. Nous n'avons pas de point de départ et c'est normal, le système PDR est relatif. Afin de rendre l'application flexible et éviter d'entrer manuellement une valeur latitude/longitude, nous allons réutiliser la carte précédemment créée pour choisir un point de départ.

**Question 3.15** Abonnez vous au listener de *GoogleMap.setOnMapClickListener()*. A chaque nouveau clic sur la carte vous commencerez un nouveau segment.

**Question 3.16** Abonnez vous au listener de *Step Detection Handler* pour savoir à quel moment l'utilisateur fait un pas. Faites le nécessaire en utilisant *Device Attitude Handler* et *Step Positioning Handler* pour récupérer la nouvelle position et l'afficher avec le *Google Map Tracer*

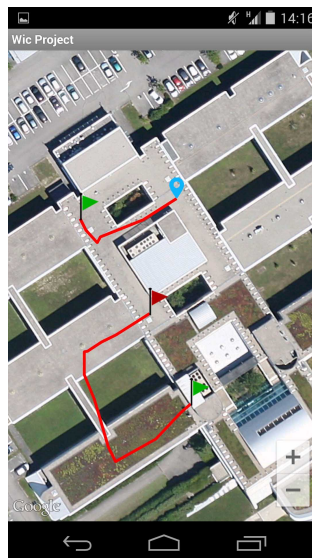


FIGURE 6 – Un exemple de l'Application PDR.

## 4 Extras

Cette partie est facultative mais elle permettra de rendre l'application un peu plus complète. Libre à vous d'implémenter ce que vous voulez. Voici une liste de propositions qui ne doivent pas forcément être traitées dans l'ordre.

### 4.1 Exporter la trace en GPX (1h30)

Vous avez un [Visualiseur de GPX](#) et un [PDR](#) qui affiche sa propre trace. Il serait intéressant de pouvoir sauvegarder les traces du PDR dans un fichier GPX afin de le revoir plus tard. Pour cela on peut utiliser [XMLSerializer](#) ou [DocumentBuilder](#) qui permettent de générer du XML. Voici [un exemple](#) d'utilisation de chacun des sérialiseurs.

**Question 4.1** *Créez une méthode qui prend en entrée une [structure GPX](#) et un [File](#) puis qui stocke les informations dans un fichier.*

### 4.2 Suivre la position de l'utilisateur (15 min)

**Question 4.2** *Faites en sorte que la caméra de la [GoogleMap](#) suive la position de l'utilisateur à chaque nouveau pas.*

### 4.3 Centrer la carte sur la dernière position de l'utilisateur (15 min)

**Question 4.3** *A l'aide du [LocationManager](#), récupérez (si elle existe) la dernière position connue de l'utilisateur. Utilisez là pour centrer la carte à l'ouverture de l'Activity*

### 4.4 Icône de l'utilisateur (15min + 45min)

Pour le moment quand vous avancez dans l'application PDR, vous voyez seulement une trace s'agrandir à l'endroit où se trouve l'utilisateur.

**Question 4.4** *Créez un marker dans [Google Map Tracer](#) qu'on déplacera à la position courante de l'utilisateur.*

Grâce au [Device Attitude Handler](#) nous avons accès à l'azimuth en temps réel du téléphone. On va s'en servir pour afficher un marker (flèche) sur lequel on effectuera une rotation.

**Question 4.5** *Créez un listener dans la classe [Device Attitude Handler](#) pour que le [Google Map Tracer](#) puisse recevoir les notifications.*

**Question 4.6** *Modifiez le marker précédent par une flèche et lui appliquer une rotation quand le téléphone aura changé de direction.*

Pour information, cette dernière partie peut être faite avec l'interface [GoogleMap.OnMyLocationChangeListener](#), mais son implémentation est un peu complexe pour le temps imparti.

## 5 Annexes

### 5.1 Les Listeners

En programmation événementielle, il existe plusieurs Design Pattern différents pour recevoir des informations provenant de d'autres entités, les listeners en font parti. Sur Android vous en avez peut être déjà vu avec les boutons d'une interface graphique, lors d'un clic de l'utilisateur un évènement est renvoyé. L'idée est de s'abonner à une entité puis de recevoir une notification à un moment qui n'était pas prévisible. Nous pouvons nous aussi utiliser ce même principe pour surveiller les capteurs (par exemple). Voici ci-dessous un exemple avec un capteur de température.

```
1 public class TemperatureHandler implements SensorEventListener {
2
3     @Override
4     public final void onSensorChanged(SensorEvent event) {
5
6         float temperature = event.values[0];
7
8         if(temperature > 30 && mTemperatureListener != null) {
9             mTemperatureListener.onHightTemperatureDetected(event.values[0]);
10        }
11    }
12
13    private TemperatureListener mTemperatureListener;
14
15    public void setTemperatureListener(TemperatureListener listener) {
16        mTemperatureListener = listener;
17    }
18
19    public interface TemperatureListener {
20        public void onHightTemperatureDetected(float temperature);
21    }
22 }
23 }
```

Dès que le capteur de température dépasse 30°, l'objet abonné reçoit alors une notification grâce à la méthode **onHightTemperatureDetected()**

```
1 public class TemperatureActivity extends Activity {
2
3     @Override
4     public final void onCreate(Bundle savedInstanceState) {
5         super.onCreate(savedInstanceState);
6         setContentView(R.layout.main);
7
8         TemperatureHandler mTemperatureHandler = new TemperatureHandler();
9         mTemperatureHandler.setListener(mTemperatureListener);
10    }
11
12    private TemperatureListener mTemperatureListener = new TemperatureListener() {
13
14        public void onHightTemperatureDetected(float temperature) {
15            Toast.makeText(getApplicationContext(),
16                "High temperature detected : " + temperature,
17                Toast.LENGTH_SHORT).show();
18        }
19    };
20 }
```



## 5.2 Les Capteurs

La plupart des appareils Android ont des capteurs embarqués qui mesurent les mouvements, l'orientation et l'environnement. Ces capteurs fournissent des données brutes avec une bonne précision. Par exemple, dans un jeu on va utiliser le capteur de gravité et l'accéléromètre pour reconnaître des mouvements complexes de l'utilisateur : inclinaison, secousse, rotation ou encore dans une application de voyage l'application utilisera le capteur magnétique et l'accéléromètre pour les utiliser comme boussole. La plateforme Android supporte 3 catégories de capteurs, ils peuvent être bruts ou **composés** (fusion) :

- **Capteurs de Mouvement** : Accéléromètre, Gravité, Gyroscope...
- **Capteurs d'Environnement** : Température extérieure, Pression, Luminosité...
- **Capteurs de Position** : Magnétique, Proximité, Orientation...

On peut accéder à tous ces capteurs grâce au Sensor Framework d'Android. On s'abonne au capteur que l'on veut utiliser grâce à un système de Listener (**SensorEventListener**). Les données reçues sont au format **SensorEvent**, on a accès à la précision de la donnée, le capteur duquel elle provient, le timestamp de l'évènement et un vecteur contenant les valeurs en fonction du capteur.

Les capteurs sont souvent très pratiques dans les applications mais il ne faut pas oublier qu'ils sont très gourmands en énergie ! Il est alors important de les utiliser dans les bonnes situations et surtout de s'y abonner seulement quand on en a besoin. Voici un exemple :

```
1 public class SensorActivity extends Activity implements SensorEventListener {
2     private SensorManager mSensorManager;
3     private Sensor mLight;
4
5     @Override
6     public final void onCreate(Bundle savedInstanceState) {
7         super.onCreate(savedInstanceState);
8         setContentView(R.layout.main);
9
10        mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
11        mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
12    }
13
14    @Override
15    public final void onAccuracyChanged(Sensor sensor, int accuracy) {
16        // Do something here if sensor accuracy changes.
17    }
18
19    @Override
20    public final void onSensorChanged(SensorEvent event) {
21        // The light sensor returns a single value.
22        // Many sensors return 3 values, one for each axis.
23        float lux = event.values[0];
24        // Do something with this sensor value.
25    }
26
27    @Override
28    protected void onResume() {
29        super.onResume();
30        mSensorManager.registerListener(this, mLight, SensorManager.SENSOR_DELAY_NORMAL);
31    }
32
33    @Override
34    protected void onPause() {
35        super.onPause();
36        mSensorManager.unregisterListener(this);
37    }
38 }
```