

Commande par ordinateur : Estimation récursive de paramètres

Amir Ben Slimane & Salah Bennour

Science Informatique
Polytech Nice-Sophia - France
01-03-2015

Table des matières

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 2 | Environnement de travail | 3 |
| 2.1 | Langage de développement | 3 |
| 2.2 | Bibliothèques | 3 |
| 2.3 | Tests | 3 |
| 2.4 | Dépendances maven | 3 |
| 3 | Mise en place du simulateur | 4 |
| 3.1 | L'objet Mobile | 4 |
| 3.2 | L'observateur | 5 |
| 3.3 | Le simulateur | 5 |
| 3.4 | Les tests | 7 |
| 4 | Estimation de la vitesse du mobile | 8 |
| 4.1 | Méthode par l'inverse | 8 |
| 4.1.1 | Algorithme | 8 |
| 4.1.2 | Implémentation | 8 |
| 4.1.3 | Résultats obtenus | 8 |
| 4.2 | Méthode du gradient conjugué | 10 |
| 4.2.1 | Algorithme | 10 |
| 4.2.2 | Implémentation | 10 |
| 4.2.3 | Résultats obtenus | 11 |
| 4.3 | Méthode des moindres carrés récursifs | 12 |
| 4.3.1 | Algorithme | 12 |
| 4.3.2 | Implémentation | 12 |
| 4.3.3 | Résultats obtenus | 12 |
| 5 | Conclusion | 13 |

1 Introduction

Ce projet consiste à estimer la position et la vitesse initiale d'un objet en mouvement rectiligne uniforme à partir d'angles depuis lesquels un observateur l'observe en gravitant autour. Les angles d'observations se font à chaque intervalle de temps.

Une première version du projet prend en compte que l'observateur connaît exactement sa position à chaque intervalle de temps.

Une deuxième version voit la mesure de sa position entachée de bruit, l'observateur commettra donc des erreurs sur sa position, et donc sur l'estimation de la position et la vitesse initiale du mobile.

Pour se faire, notre simulation se basera sur des méthodes de résolution de systèmes d'équations. Il sera alors possible de faire une estimation assez proche des paramètres initiaux du mobile.

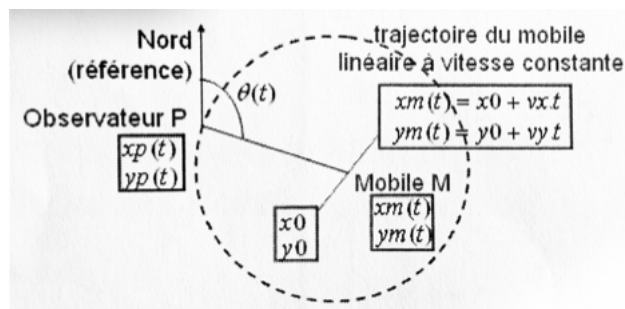


FIGURE 1 – Schéma du problème

2 Environnement de travail

2.1 Langage de développement

Le langage de développement de ce projet est le JAVA. Ce langage répond parfaitement aux besoins, de par son accessibilité que par l'existence de bibliothèques pour afficher une courbe ou bien manipuler des matrices. De plus le JAVA est un langage orienté objets dont les élèves de l'école sont spécialistes.

2.2 Bibliothèques

Pour la manipulation des matrices, la bibliothèque Jblas a répondu parfaitement à nos attentes grâce à son objet DoubleMatrix qui permet de manipuler des matrices de réels.

Concernant l'affichage graphique des trajectoires du mobile et de l'observateur, le projet utilise la bibliothèque JfreeChart qui permet la représentation de courbes sur des axes d'orientation.

2.3 Tests

Pour garantir la veracité et la robustesse de notre modélisation, des tests ont été effectués pour chacun des algorithmes implémentés. Ces tests ont été réalisés à partir de la bibliothèque JUnit.

2.4 Dépendances maven

Le projet est un projet maven, les dépendances citées ci-dessus sont donc directement téléchargés depuis maven.

3 Mise en place du simulateur

Le simulateur du projet permet d'instancier les différents objets pour mener à bien notre estimation des paramètres initiaux du mobile.

Pour cela, il se charge de créer un objet Mobile et un objet Observateur héritant d'un même objet Point. Ces deux objets représentent respectivement le mobile en mouvement rectiligne uniforme et l'observateur qui gravite autour de ce mobile.

La classe Resolution contient des algorithmes permettant de résoudre un système d'équations sous la forme d'une équation matricielle.

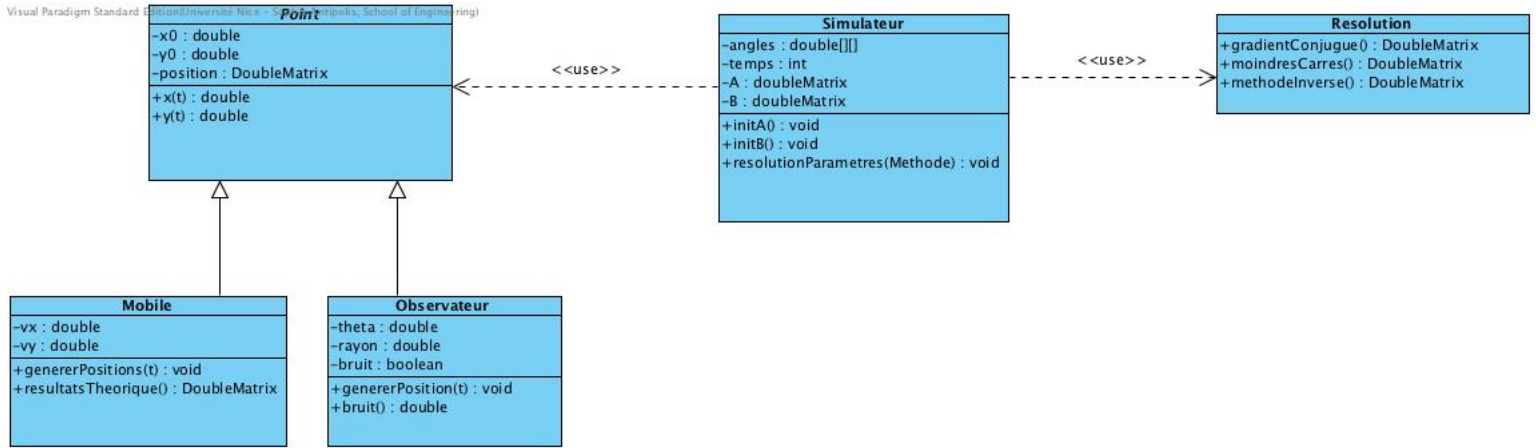


FIGURE 2 – Diagramme de classe

3.1 L'objet Mobile

L'objet Mobile est initialisé avec une matrice de taille 2x1 dont les deux valeurs correspondent aux coordonnées initiales du mobile, respectivement en abscisse et en ordonnée. Un algorithme génère ses positions sur le nombre de périodes souhaitées, la matrice devient alors de taille 2xn.

Ainsi pour tout t , on a :

$$x_t = x_0 + v_x \times t \quad (1)$$

$$y_t = y_0 + v_y \times t \quad (2)$$

- x_t et y_t : la position du mobile à un instant t .
- x_0 et y_0 : la position initiale.
- v_x et v_y : la vitesse

3.2 L'observateur

Observateur et Mobile héritant de la même classe, le concept de génération des positions reste le même. La génération des positions dépend de la présence ou non du bruit.

Ainsi pour tout t , on a :

$$x_t = x_c + R \times \cos(v \times t) \quad (3)$$

$$y_t = y_c + R \times \sin(v \times t) \quad (4)$$

- x_t et y_t : la position du mobile à un instant t .
- x_c et y_c : la position du cercle.
- R : le rayon du cercle.
- v : la vitesse circulaire.

3.3 Le simulateur

Le simulateur a pour tâche de générer les déplacements de l'observateur et du mobile au cours du temps t . En fonction de ces positions, il calcule alors l'angle qu'il y a entre ces deux objets pour chaque intervalle t . Lors de la prise en compte du bruit, la position de l'observateur n'étant pas exacte, les mesures d'angles se retrouvent être imprécises.

$$angle_t = atan2(y_{mobile}(t) - y_{observateur}(t), x_{mobile}(t) - x_{observateur}(t)) \quad (5)$$

Une fois toutes les générations des positions de l'observateur et du mobile faites, on génère les matrices A et B. Pour cela nous utilisons l'erreur de prédiction qui est nulle si les mesures sont parfaites. A chaque intervalle de temps, on doit donc avoir une prédiction nulle.

L'erreur de prédiction est la suivante :

$$\begin{aligned} \epsilon = & \cos(\theta(t)) \times y_p(t) - \sin(\theta(t)) \times x_p(t) \\ & + \sin(\theta(t)) \times x_0 + \sin(\theta(t)) \times t \times v_x \\ & - \cos(\theta(t)) \times y_0 - \cos(\theta(t)) \times t \times v_y \end{aligned} \quad (6)$$

Avec $\epsilon = 0$ lorsque les mesures sont parfaites.

Les paramètres initiaux du mobile à estimer sont x_0 , y_0 , v_x et v_y . Ce sont les 4 inconnus que le simulateur va chercher à découvrir. Pour ça, il faut donc n équations à un intervalle de temps différent avec $n \geq 4$ car il faut au moins un système de 4 équations pour résoudre un système d'équations à 4 inconnues. Ce qui donne l'équation matricielle suivante $A * X = B$ avec A une matrice de taille $4 \times n$, X et B deux matrices de taille $4 * 1$.

Ce qui donne en développant :

$$\begin{pmatrix} \sin(\theta(1)) & -\cos(\theta(1)) & \sin(\theta(1)) \times 1 & -\cos(\theta(1)) \times 1 \\ \vdots & \vdots & \vdots & \vdots \\ \sin(\theta(T)) & -\cos(\theta(T)) & \sin(\theta(T)) \times T & -\cos(\theta(T)) \times T \end{pmatrix} \times \begin{pmatrix} x_0 \\ y_0 \\ v_x \\ v_y \end{pmatrix} = \quad (7)$$

$$\begin{pmatrix} \sin(\theta(1)) \times x_p(1) - \cos(\theta(1)) \times y_p(1) \\ \vdots \\ \sin(\theta(T)) \times x_p(T) - \cos(\theta(T)) \times y_p(T) \end{pmatrix}$$

S'ajoute ensuite le bruit lorsque la simulation est lancée. Le bruit est ici implémenté par la génération d'un nombre aléatoire variant entre $[- \text{rayon} * 3\% ; \text{rayon} * 3\%]$ dont l'espérance mathématique est de 0. Le bruit est appliqué à la matrice B juste après le calcul de l'angle entre l'observateur et le mobile. On obtient alors pour toute ligne i de la matrice B :

$$\sin(\theta(i)) \times x_p(i) - \cos(\theta(i)) \times y_p(i) + \text{bruit} \quad (8)$$

3.4 Les tests

Afin que la simulation n'ait pas d'erreur, les algorithmes de résolution de systèmes d'équations ont été testés à partir de la bibliothèque JUnit. Avant chaque test, on crée nos matrices A et B à partir d'un exemple proposé par Mr Leroux.

```
@Before // Initialisation des matrices A et B avant chaque test
public void setUp() throws Exception {
    double[][] tabA = new double[4][4], tabB = new double[4][1];
    // Matrice A
    tabA[0] = new double[]{3.044, 0.705, -0.896, 1.324};
    tabA[1] = new double[]{0.705, 1.303, -1.055, 0.934};
    tabA[2] = new double[]{-0.896, -1.055, 1.202, -1.096};
    tabA[3] = new double[]{1.324, 0.934, -1.096, 1.269};

    // Matrice B
    tabB[0][0] = -0.098;
    tabB[1][0] = -0.886;
    tabB[2][0] = 0.567;
    tabB[3][0] = 0.04;

    A = new DoubleMatrix(tabA);
    B = new DoubleMatrix(tabB);
}
```

FIGURE 3 – Initialisation des matrices A et B

Une fois nos matrices créées, les différentes méthodes de résolution sont testées.

```
@Test
public void testGradientConjugué() {
    DoubleMatrix res = Resolution.gradientConjugué(A, B); // Résolution de X
    DoubleMatrix resxA = A.mmul(res); //  $AX := A * X$ 

    assertTrue(resxA.equals(B)); // Test si  $AX = B$ 
}

@Test
public void testMoindresCarres() {
    DoubleMatrix res = Resolution.moindresCarres(4, A, B); // Résolution de X
    DoubleMatrix resxA = A.mmul(res); //  $AX := A * X$ 

    for(int i = 0; i < resxA.rows; i++) {
        for(int j = 0; j < resxA.columns; j++)
            // test pour chaque valeur si  $AX(i,j) = B(i,j)$  à 0,01 près
            assertTrue(Math.abs(resxA.get(i, j) - B.get(i, j)) < 0.01);
    }
}

@Test
public void testInverse() {
    DoubleMatrix res = Resolution.methodeInverse(A, B); // Résolution de X
    DoubleMatrix resxA = A.mmul(res); //  $AX := A * X$ 

    assertTrue(resxA.equals(B)); // Test si  $AX = B$ 
}
```

FIGURE 4 – Tests des algorithmes de résolution

4 Estimation de la vitesse du mobile

Cette partie du rapport consiste à présenter les différentes méthodes de résolution de l'équation matricielle décrite dans la description du simulateur.

Pour les exemples, l'objet mobile est initialisé avec les valeurs suivantes :

$$\begin{pmatrix} x_0 \\ y_0 \\ v_x \\ v_y \end{pmatrix} = \begin{pmatrix} 10 \\ 10 \\ 2 \\ 2,5 \end{pmatrix} \quad (9)$$

4.1 Méthode par l'inverse

4.1.1 Algorithme

On peut dès lors utiliser ces deux matrices pour trouver la matrice X en utilisant la formule de la pseudo-inverse.

En effet d'après l'équation obtenue précédemment, la matrice X est la solution de l'équation linéaire :

$$A^T \times A \times X = A^T \times B \quad (10)$$

Qui peut également s'écrire en fonction de la pseudo-inverse :

$$X = (A^T \times A)^{-1} \times A^T \times B \quad (11)$$

4.1.2 Implémentation

```
public static DoubleMatrix methodeInverse(DoubleMatrix A, DoubleMatrix B) {
    DoubleMatrix res = new DoubleMatrix(4, 1); // initialisation d'une matrice de taille 4 x 1

    DoubleMatrix transposeA = A.transpose(); // transposeA := transpose de A;
    DoubleMatrix gamma = transposeA.mmul(A); // gamma := transpose de A * A
    B = transposeA.mmul(B); // B := transpose de A * B

    res = inverse(gamma).mmul(B); // X := inverse de gamma * B

    return res;
}
```

FIGURE 5 – Implémentation de la méthode par l'inverse

4.1.3 Résultats obtenus

| Sans bruit | | Avec bruit | |
|--|--|--|--|
| Estimation | Erreur | Estimation | Erreur |
| $\begin{pmatrix} 10 \\ 10 \\ 2 \\ 2.5 \end{pmatrix}$ | $\begin{pmatrix} 2.61E-12 \\ 2.05E-12 \\ 2.70E-13 \\ 1.42E-13 \end{pmatrix}$ | $\begin{pmatrix} 9.47 \\ 9.54 \\ 2.05 \\ 2.54 \end{pmatrix}$ | $\begin{pmatrix} 0.53 \\ 0.46 \\ 0.05 \\ 0.04 \end{pmatrix}$ |

Les résultats calculés sans le bruit sont excellents. Cependant les résultats avec le bruit faussent légèrement l'estimation mais ils restent relativement bon et assez proche de la réalité.

4.2 Méthode du gradient conjugué

4.2.1 Algorithme

Toujours à partir des matrices A et B, il est possible de retrouver le résultat grâce à la méthode du gradient conjugué.

En effet, la méthode du gradient conjugué nous permet de résoudre une équation du même type que celle qui se présente ici, à savoir $T x = b$.

Pour ce faire, il y'a un algorithme à suivre :

The diagram illustrates the steps of the conjugate gradient algorithm. It is divided into an initialization phase and a main loop.

Initialisation

- x_0 quelconque
- $g_0 = \Gamma x_0 - b$ (gradient de $\frac{1}{2} x^T \Gamma x - x^T b$)
- $h_0 = g_0$ (le gradient conjugué de g_0)

pour $n = 1 \dots L$ (matrice Γ de dimension L)

- $$x_n = x_{n-1} - h_{n-1} \cdot \frac{h_{n-1}^T g_{n-1}}{h_{n-1}^T \Gamma h_{n-1}}$$
- $$g_n = g_{n-1} - \Gamma h_{n-1} \cdot \frac{h_{n-1}^T g_{n-1}}{h_{n-1}^T \Gamma h_{n-1}} (= \Gamma x_n - b)$$
- $$h_n = g_n - h_{n-1} \cdot \frac{h_{n-1}^T g_n}{h_{n-1}^T \Gamma h_{n-1}} \left(= g_n + h_{n-1} \cdot \frac{g_n^T g_n}{g_{n-1}^T g_{n-1}} \right)$$

FIGURE 6 – Gradient conjugué

4.2.2 Implémentation

```
public static DoubleMatrix gradientConjugué(DoubleMatrix A, DoubleMatrix B) {
    DoubleMatrix res = new DoubleMatrix(4, 1); // initialisation d'une matrice de taille 4 x 1

    DoubleMatrix transposeA = A.transpose(); // A := transpose de A

    DoubleMatrix gamma = transposeA.mmul(A); // gamma := transpose de A * A
    B = transposeA.mmul(B); // B := transpose de A * B

    DoubleMatrix g = gamma.mmul(res).sub(B); // g := gamma * X - B

    DoubleMatrix h = g; // h := g

    double temp;
    DoubleMatrix transposeH;
    for (int i = 0; i < res.rows; i++) { // Iteration sur les n lignes de X
        transposeH = h.transpose();
        // temp := (transpose de h * g) / (transpose de h * gamma * h)
        temp = transposeH.mmul(g).get(0, 0)
            / (transposeH.mmul(gamma).mmul(h).get(0, 0));

        res = res.sub(h.mmul(temp)); // X := X - h * temp

        g = g.sub(gamma.mmul(h).mmul(temp)); // g = g - gamma * h * temp

        // h := g - h * (transpose de h * gamma * g) / (transpose de h * gamma * h)
        h = g.sub(h.mmul(
            (transposeH.mmul(gamma).mmul(g)).get(0, 0)
            / (transposeH.mmul(gamma).mmul(h).get(0, 0))));
    }

    return res;
}
```

FIGURE 7 – Implémentation de la méthode du gradient conjugué

4.2.3 Résultats obtenus

| Sans bruit | | Avec bruit | |
|---|--|--|--|
| Estimation | Erreur | Estimation | Erreur |
| $\begin{pmatrix} 10 \\ 10 \\ 2 \\ 2.5 \end{pmatrix}$ pmatrix | $\begin{pmatrix} 2.74E - 12 \\ 1.82E - 11 \\ 5.78E - 11 \\ 2.49E - 10 \end{pmatrix}$ | $\begin{pmatrix} 9.45 \\ 9.55 \\ 2.04 \\ 2.56 \end{pmatrix}$ | $\begin{pmatrix} 0.55 \\ 0.45 \\ 0.04 \\ 0.06 \end{pmatrix}$ |

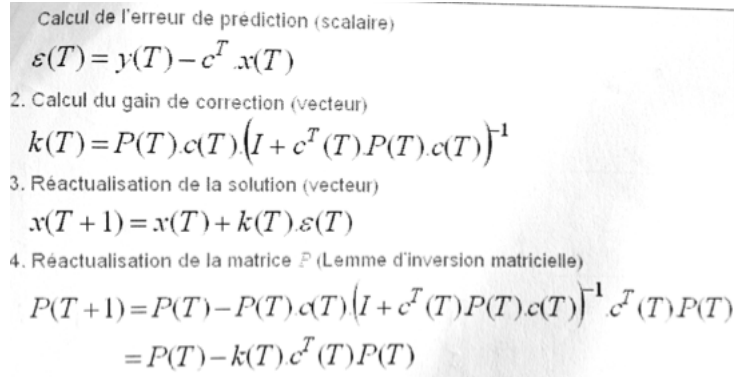
Même constat qu'avec la méthode inverse, les résultats calculés sans le bruit sont excellents et ceux avec le bruit restent relativement bon et assez proche de la réalité.

4.3 Méthode des moindres carrés récursifs

4.3.1 Algorithme

Au lieu de disposer d'un vecteur de données et d'effectuer une estimation globale via les méthodes explicitées ci-dessus, les moindres carrés récursifs visent à réactualiser le vecteur des paramètres à chaque nouvelle mesure tout en conservant le même critère.

L'algorithme à suivre est le suivant :



Calcul de l'erreur de prédiction (scalaire)

$$\varepsilon(T) = y(T) - c^T x(T)$$

2. Calcul du gain de correction (vecteur)

$$k(T) = P(T) c(T) \left(I + c^T(T) P(T) c(T) \right)^{-1}$$

3. Réactualisation de la solution (vecteur)

$$x(T+1) = x(T) + k(T) \varepsilon(T)$$

4. Réactualisation de la matrice P (Lemme d'inversion matricielle)

$$P(T+1) = P(T) - P(T) c(T) \left(I + c^T(T) P(T) c(T) \right)^{-1} c^T(T) P(T) \\ = P(T) - k(T) c^T(T) P(T)$$

FIGURE 8 – Formule : moindres carrés récursive

4.3.2 Implémentation

```
public static DoubleMatrix moindresCarres(int n, DoubleMatrix A, DoubleMatrix B) {
    DoubleMatrix res = new DoubleMatrix(4, 1),
    A2 = new DoubleMatrix(4, 1),
    P = new DoubleMatrix(4, 4); // Initialisation de matrices 4 x 1

    for (int i = 0; i < 4; i++)
        P.put(i, i, 999999); // Initialisation de la diagonale avec une grande valeur

    double predictionErreur, temp;
    DoubleMatrix K;
    for (int i = 0; i < n; i++) {
        A2.put(0, 0, A.get(i,0));
        A2.put(1, 0, A.get(i,1));
        A2.put(2, 0, A.get(i,2));
        A2.put(3, 0, A.get(i,3)); // A2 := colonne i de A

        predictionErreur = B.get(i,0) - A2.transpose().mmul(res).get(0,0); // calcul de l'erreur de prediction e

        temp = A2.transpose().mmul(P).mmul(A2).get(0,0); // temp := transpose de A2 * P * A2

        K = P.mmul(A2).mmul(1 / (temp + 1)); // K := P * A2 * (1 / temp + 1)

        res = res.add(K.mmul(predictionErreur)); // X := X + K * e

        P = P.sub(K.mmul(A2.transpose()).mmul(P)); // P := P - K * transpose de A2 * P
    }

    return res;
}
```

FIGURE 9 – Implémentation de la méthode des moindres carrés

4.3.3 Résultats obtenus

Dans le cas d'une mesure sans bruit les résultats sont tout aussi parfaits qu'avec les autres méthodes. Cependant on ressent une erreur plus importante lors de l'insertion du bruit.

| Sans bruit | | Avec bruit | |
|--|--|--|--|
| Estimation | Erreur | Estimation | Erreur |
| $\begin{pmatrix} 10 \\ 10 \\ 2 \\ 2.5 \end{pmatrix}$ | $\begin{pmatrix} 2.62E-5 \\ 2.15E-5 \\ 2.72E-6 \\ 1.91E-6 \end{pmatrix}$ | $\begin{pmatrix} 10.82 \\ 11.52 \\ 1.84 \\ 2.36 \end{pmatrix}$ | $\begin{pmatrix} 0.82 \\ 1.52 \\ 0.16 \\ 0.14 \end{pmatrix}$ |

5 Conclusion

Grâce aux trois méthodes mathématiques présentées ici, il a été possible de déterminer la vitesse et la position initiale du mobile en déplacement. Toutes donnent un résultat plus que correct lorsque la mesure est effectuée sans bruit, tandis que le résultat reste approximatif – mais tout à fait réaliste – dans le cas d’une mesure avec bruit. Il est bon de signaler que dans le cadre de cette simulation, la méthode de résolution par moindres carrés récursifs reste la moins précise.

Dans le cadre du projet fait lors de la première partie de ce cours, il aurait été possible d’intégrer les connaissances acquises ici. En effet, il s’agissait de poser un module sur un des astres du système solaire. La suite de ce projet se penche donc logiquement vers l’ajout d’un module à envoyer depuis l’observateur qui atterit sur le mobile une fois les paramètres initiaux du mobile estimés. Et dans ce cas les bruits risquent vraiment de poser problème...