# Projet d'Algèbre Linéaire Numérique

François Boulier

Mai 2020

## 1 La fonction logistique

L'importance de la fonction logistique (1) en dynamique de populations a été mise en évidence pour la première fois par Pierre-François Verhulst en 1838 [2].

$$y(x) \quad = \quad \frac{\kappa}{1+e^{\alpha-\rho\,x}}\cdot \tag{1}$$

En dynamique de populations, la variable indépendante $x$ représenterait le temps, la variable dépendante $y(x)$ la population au temps $x$ et les trois lettres grecques $\kappa, \alpha, \rho$ des paramètres. La courbe de la fonction logistique a la forme d'une sigmoïde ayant pour asymptotes horizontales les droites $y = 0$ et $y = \kappa$.

Après les travaux de Verhulst, la fonction logistique a été redécouverte par Pearl en 1920 et a connu un immense succès pour la représentation de données expérimentales dans des domaines très variés. Voir par exemple [1, section 6.2, pages 148 et suivantes]. Une question très naturelle consiste alors, partant de données expérimentales, à déterminer les valeurs des trois paramètres $\kappa, \alpha, \rho$ qui font passer la sigmoïde au plus près de ces données.

On raisonne sur les données suivantes, adaptées de [1, Fig. 7.2, page 229]. À une constante additive près, il s'agit de $m = 10$ mesures de la quantité d'eau présente dans des œufs de *Locustana pardalina* en formation, à une température de 35 degrés.

| nb. jours | $x_i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| qté eau | $y_i$ | .53 | .53 | 1.53 | 2.53 | 12.53 | 21.53 | 24.53 | 28.53 | 28.53 | 30.53 |

(2)

### 1.1 Estimation par moindres carrés linéaires

Introduisons la fonction logit définie par

$$\mathrm{logit}(p) \quad = \quad \ln\left(\frac{p}{1-p}\right)\cdot$$

À partir de (1), on trouve

$$\frac{\kappa}{y} \quad = \quad 1+e^{\alpha-\rho\,x} \quad \text{et donc} \quad \mathrm{logit}\left(\frac{y}{\kappa}\right) \quad = \quad \rho\,x-\alpha\,.$$

Supposons $\kappa$ connu. Il suffit alors d'appliquer, sur les ordonnées $y_i$ des points expérimentaux (tableau (2)), la transformation ci-dessus (logit) puis d'estimer les deux paramètres $\alpha$ et $\rho$ par la méthode des moindres carrés.

Comment déterminer $\kappa$ ? Visuellement, il est souvent facile d'estimer l'asymptote horizontale (d'équation $y = \kappa$) vers laquelle tend la sigmoïde. C'est la méthode préconisée dans les ouvrages anciens [1, page 150].

On remarque que l'emploi de cette méthode implique de choisir $\kappa > y_i$ pour $1 \leq i \leq m$ parce que $\text{logit}(p)$ n'est définie que pour $0 < p < 1$. En prenant $\kappa = 30.54$ on trouve (voir Figure 1) :

$$\alpha = 5.163\,, \quad \rho = 1.188\,.$$

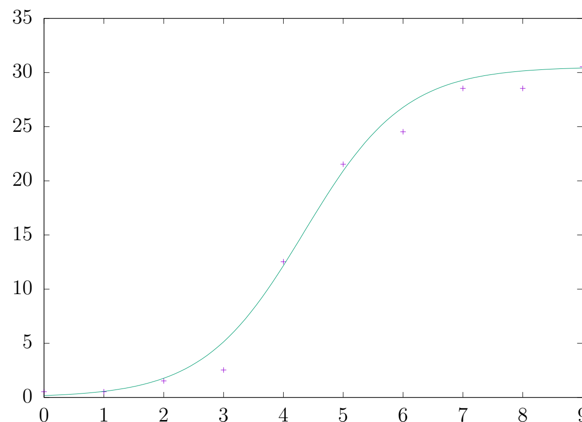


FIGURE 1 – Les données du tableau (2) et la courbe logistique obtenue pour $\kappa, \alpha, \rho = 30.54, 5.163, 1.188$. La somme des carrés des écarts verticaux entre la courbe et les points vaut approximativement 3.968.

L'estimation ci-dessus n'est pas mauvaise mais n'est pas optimale (en général). En effet, il n'y aucune raison pour que $\kappa$ soit supérieur aux ordonnées de tous les points. C'est la raison pour laquelle les logiciels modernes utilisent des méthodes plus sophistiquées : la méthode de Newton, vue au DM numéro 2 ou la méthode de Gauss-Newton, spécialement adaptée aux problèmes de moindres carrés.

## 1.2 Définitions communes aux deux méthodes

Dans les deux cas, ces méthodes ont besoin d'un bon point de départ pour converger vers le minimum local recherché : on pourra utiliser l'estimation obtenue ci-dessus.

Soient $m$ le nombre de points expérimentaux et $n$ le nombre de paramètres à estimer. Sur notre exemple on a $m = 10$ et $n = 3$. L'idée consiste à construire une suite de vecteurs

$$v_0 = \begin{pmatrix} \kappa_0 \\ \alpha_0 \\ \rho_0 \end{pmatrix}, \quad v_1 = \begin{pmatrix} \kappa_1 \\ \alpha_1 \\ \rho_1 \end{pmatrix}, \quad v_2 = \begin{pmatrix} \kappa_2 \\ \alpha_2 \\ \rho_2 \end{pmatrix}, \quad \ldots$$

de $\mathbb{R}^n$ dont on espère qu'elle converge vers des valeurs qui minimisent la somme des carrés des écarts verticaux entre la courbe et les points. Pour cela, on introduit la fonction $s$ (pour *sigmoïde*)

$$s(\kappa, \alpha, \rho, x) = \frac{\kappa}{1 + e^{\alpha - \rho\, x}}$$

et on considère la fonction $r$ (pour *résidu*) de $\mathbb{R}^n$ dans $\mathbb{R}^m$ définie par

$$r(\kappa, \alpha, \rho) = \begin{pmatrix} s(\kappa, \alpha, \rho, x_1) - y_1 \\ s(\kappa, \alpha, \rho, x_2) - y_2 \\ \vdots \\ s(\kappa, \alpha, \rho, x_m) - y_m \end{pmatrix}.$$

## 1.3 Méthode de Newton

La méthode de Newton consiste à chercher un optimum de la fonction

$$\begin{aligned} f(\kappa, \alpha, \rho) &= \|r(\kappa, \alpha, \rho)\|_2^2 \\ &= \sum_{i=1}^{m} (s(\kappa, \alpha, \rho, x_i) - y_i)^2 . \end{aligned}$$

Il suffit d'appliquer la méthode du DM numéro 2. Une difficulté consiste à calculer la matrice hessienne de $f$. À la fin de ce document, on donne une *worksheet* Maple qui montre comment l'obtenir sans trop de difficultés avec un paquetage de génération de code.

## 1.4 Méthode de Gauss-Newton

Il s'agit d'une variante de la méthode de Newton, spécifique aux problèmes de moindres carrés, qui approxime la matrice hessienne — souvent trop pénible à calculer — en utilisant la jacobienne de $r$.

Notons $J(\kappa, \alpha, \rho)$ la matrice jacobienne de la fonction $r$. Il s'agit de la matrice $m \times n$ définie par :

$$J(\kappa, \alpha, \rho) = \begin{pmatrix} s_{\kappa,1} & s_{\alpha,1} & s_{\rho,1} \\ s_{\kappa,2} & s_{\alpha,2} & s_{\rho,2} \\ \vdots & \vdots & \vdots \\ s_{\kappa,m} & s_{\alpha,m} & s_{\rho,m} \end{pmatrix},$$

où $s_{\kappa,i}$, $s_{\alpha,i}$ et $s_{\rho,i}$ désignent les dérivées partielles de $s$ par rapport à $\kappa$, $\alpha$ et $\rho$, évaluées en $x = x_i$. Sur l'exemple, cela donne :

$$s_{\kappa,i} = \frac{1}{1 + e^{\alpha - \rho\, x_i}}, \quad s_{\alpha,i} = -\kappa\, \frac{e^{\alpha - \rho\, x_i}}{(1 + e^{\alpha - \rho\, x_i})^2}, \quad s_{\rho,i} = \kappa\, x_i\, \frac{e^{\alpha - \rho\, x_i}}{(1 + e^{\alpha - \rho\, x_i})^2}.$$

Plaçons-nous à une itération $\ell$ et supposons $v_\ell$ déjà calculé. La méthode de Gauss-Newton consiste à construire le vecteur $r(\kappa_\ell, \alpha_\ell, \rho_\ell)$ et la matrice $J(\kappa_\ell, \alpha_\ell, \rho_\ell)$ — notés ci-dessous $r$ et $J$ pour simplifier — à résoudre le système d'équations linéaires

$$(J^T\,J)\,w \;=\; -J^T\,r \tag{3}$$

où $w$ est un vecteur de $n$ inconnues, puis à prendre

$$v_{\ell+1} \;=\; v_\ell + w\,.$$

La méthode de Gauss-Newton effectue autant de résolutions du système (3) qu'elle effectue d'itérations. Le système (3) est un système d'équations normales, comme dans la méthode historique de résolution des moindres carrés linéaires. À la fin de ce document, on trouvera une *worksheet* Maple qui applique un paquetage de génération de code à la matrice $J^T\,J$.

## 1.5 Indications

En appliquant la méthode de Gauss-Newton au problème posé et en prenant pour $v_0$ l'estimation fournie en section 1.1, on trouve l'amélioration suivante (voir Figure 2) :

$$\kappa = 29.16\,, \quad \alpha = 5.8\,, \quad \rho = 1.3454\,.$$

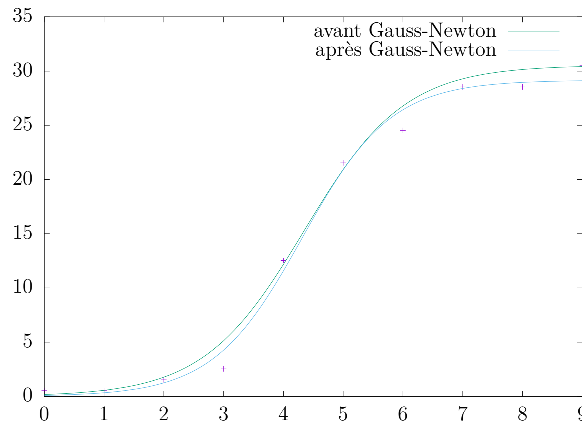


FIGURE 2 – Les données du tableau (2) et la courbe logistique obtenue pour $\kappa$, $\alpha$, $\rho$ = $29.16\,, 5.8\,, 1.3454$. L'asymptote horizontale est légèrement en-dessous du point d'ordonnée maximale. La somme des carrés des écarts verticaux entre la courbe et les points vaut approximativement 3.24.

## 1.6 Travail à faire

**Question 1.** Écrire des codes Python qui calculent les paramètres optimaux par les méthodes présentées dans ce document. Utiliser les fonctions de `scipy.linalg` à la façon vue en TD. Votre code doit être commenté.

**Question 2.** Écrire une fonction Python paramétrée par $\kappa, \alpha, \rho$ qui retourne la matrice hessienne (donner cette fonction dans le rapport).

**Question 3.** Comparer les comportements et les résultats obtenus par la méthode de Newton et la méthode de Gauss-Newton. Trouve-t-on les mêmes valeurs ?

**Question 4.** Adapter l'une des deux méthodes pour estimer un quatrième paramètre ($\lambda$) en même temps que les trois autres, à partir du modèle

$$y(x) \quad = \quad \frac{\kappa}{1 + e^{\alpha - \rho\, x}} + \lambda$$

et appliquer votre méthode sur les données réelles [1, page 229] :

| nb. jours $x_i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| % eau $y_i$ | 51 | 51 | 52 | 53 | 63 | 72 | 75 | 79 | 79 | 81 |

(4)

## 1.7 Consignes

Le projet est à réaliser en binôme. Le rendu est constitué de un ou plusieurs codes Python et d'un court rapport au format PDF que vous placerez dans le dépôt git d'un des membres du binôme.

Votre code Python doit utiliser `numpy`, les fonctions de `scipy.linalg` correspondant aux algorithmes étudiés en cours et `matplotlib` pour les graphiques. Voir les corrections de TD présentes sur le site du cours. Il devrait donc commencer par

```
import math
import numpy as np
import scipy.linalg as nla
import matplotlib.pyplot as plt
```

Le but de votre rapport est de permettre de noter votre travail. Ne rappelez pas l'énoncé. Répondez aux questions. Expliquez clairement ce que vous avez fait. Incorporez des graphiques ou des simulations numériques obtenus à partir de vos codes. Chaque graphique ou simulation doit être accompagné d'une légende qui explique en quoi ce résultat montre que vos résultats sont corrects. Enfin, donnez les instructions permettant de reproduire vos graphiques ou vos simulations à partir de vos codes.

# Références


[1] L. C. Birch and H. G. Andrewartha. *The Distribution and Abundance of Animals.* The University of Chicago Press, 1954.

[2] Pierre-François Verhulst. Notice sur la loi que la population suit dans son accroissement. *Correspondance mathématique et physique*, 10 :113–121, 1838.

# Génération de code pour Gauss-Newton

```
> restart;
  with (LinearAlgebra):
> s := (kappa,alpha,rho,x) -> kappa/(1+exp(alpha-rho*x));
```

$$s := (\kappa, \alpha, \rho, x) \mapsto \frac{\kappa}{1 + e^{\alpha - \rho \cdot x}} \tag{1}$$

Le vecteur r est de dimension m. On triche : on ne met qu'une seule coordonnée, qui dépend de (x[i],y[i])

```
> r := < s(kappa,alpha,rho,x[i])-y[i] >;
```

$$r := \left[ \frac{\kappa}{1 + e^{-\rho\, x_i + \alpha}} - y_i \right] \tag{2}$$

```
> J := VectorCalculus:-Jacobian (r,[kappa,alpha,rho]);
```

$$J := \left[ \frac{1}{1 + e^{-\rho\, x_i + \alpha}} \quad -\frac{\kappa\, e^{-\rho\, x_i + \alpha}}{\left(1 + e^{-\rho\, x_i + \alpha}\right)^2} \quad \frac{\kappa\, x_i\, e^{-\rho\, x_i + \alpha}}{\left(1 + e^{-\rho\, x_i + \alpha}\right)^2} \right] \tag{3}$$

Chaque coordonnée du vecteur suivant est en réalité une somme pour i variant de 1 à m

```
> m_JT_r := - Transpose(J) . r;
```

$$m_JT_r := \left[ \begin{array}{c} -\dfrac{\dfrac{\kappa}{1 + e^{-\rho\, x_i + \alpha}} - y_i}{1 + e^{-\rho\, x_i + \alpha}} \\[2ex] \dfrac{\kappa\, e^{-\rho\, x_i + \alpha} \left( \dfrac{\kappa}{1 + e^{-\rho\, x_i + \alpha}} - y_i \right)}{\left(1 + e^{-\rho\, x_i + \alpha}\right)^2} \\[2ex] -\dfrac{\kappa\, x_i\, e^{-\rho\, x_i + \alpha} \left( \dfrac{\kappa}{1 + e^{-\rho\, x_i + \alpha}} - y_i \right)}{\left(1 + e^{-\rho\, x_i + \alpha}\right)^2} \end{array} \right] \tag{4}$$

Voici du code Python optimisé pour construire m_JT_r. Ce code doit être réinterprété pour tenir compte du fait que les coordonnées de m_JT_r sont en fait des sommes ! La dernière ligne doit donc être interprétée comme la contribution du ième terme de la somme à la matrice. De même, le décalage d'indice n'a pas nécessairement de raison d'être. À voir ...

```
> CodeGeneration:-Python (m_JT_r, optimize, resultname='b');
t1 = math.exp(-rho * x[i - 1] + alpha)
t2 = 1 + t1
t2 = 0.1e1 / t2
t3 = kappa * t2 - y[i - 1]
t1 = kappa * t2 ** 2 * t1 * t3
b = numpy.mat([-t2 * t3,t1,-t1 * x[i - 1]])
```

La matrice J**T . J est de dimension 3 x 3. Chacun de ses éléments est en réalité une somme pour i variant de 1 à m

```
> JT_J := Transpose(J) . J;
```

$$JT_J := \begin{bmatrix} \frac{1}{\left(1+e^{-\rho x_i+\alpha}\right)^2} & -\frac{\kappa e^{-\rho x_i+\alpha}}{\left(1+e^{-\rho x_i+\alpha}\right)^3} & \frac{\kappa x_i e^{-\rho x_i+\alpha}}{\left(1+e^{-\rho x_i+\alpha}\right)^3} \\ -\frac{\kappa e^{-\rho x_i+\alpha}}{\left(1+e^{-\rho x_i+\alpha}\right)^3} & \frac{\kappa^2 \left(e^{-\rho x_i+\alpha}\right)^2}{\left(1+e^{-\rho x_i+\alpha}\right)^4} & -\frac{\kappa^2 \left(e^{-\rho x_i+\alpha}\right)^2 x_i}{\left(1+e^{-\rho x_i+\alpha}\right)^4} \\ \frac{\kappa x_i e^{-\rho x_i+\alpha}}{\left(1+e^{-\rho x_i+\alpha}\right)^3} & -\frac{\kappa^2 \left(e^{-\rho x_i+\alpha}\right)^2 x_i}{\left(1+e^{-\rho x_i+\alpha}\right)^4} & \frac{\kappa^2 x_i^2 \left(e^{-\rho x_i+\alpha}\right)^2}{\left(1+e^{-\rho x_i+\alpha}\right)^4} \end{bmatrix} \tag{5}$$

Et voici le code Python pour la matrice JT_J. Mêmes remarques que ci-dessus

```
> CodeGeneration:-Python(JT_J,optimize,resultname='A');
t1 = math.exp(-rho * x[i - 1] + alpha)
t2 = 1 + t1
t2 = 0.1e1 / t2
t3 = t2 ** 2
t2 = t2 * t3 * kappa
t4 = t2 * t1
t2 = t2 * x[i - 1] * t1
t1 = kappa ** 2 * t3 ** 2 * t1 ** 2
t5 = t1 * x[i - 1]
A = numpy.mat([[t3,-t4,t2],[-t4,t1,-t5],[t2,-t5,t1 * x[i - 1] **
2]])
```

## Génération de code pour la méthode de Newton

```
> restart;
```

La version générique de la fonction f à minimiser (= la norme deux du vecteur r)

```
> f(kappa,alpha,rho) := sum ((s(kappa,alpha,rho,x[i])-y[i])**2, i =
  1..m);
```

$$f(\kappa, \alpha, \rho) := \sum_{i=1}^{m} \left(s(\kappa, \alpha, \rho, x_i) - y_i\right)^2 \tag{6}$$

La première commande calcule la matrice Hessienne, qui est de dimension 3x3. La seconde commande est destinée à simplifier l'affichage. On observe que chaque coordonnée de H est une somme pour i variant de 1 à m

```
> H := VectorCalculus:-Hessian (f(kappa,alpha,rho),[kappa,alpha,
  rho]):
  subs ([`diff`=`Diff`,s(kappa,alpha,rho,x[i])=s], H);
```

(7)

$$\left[\begin{array}{ccc} \sum_{i=1}^{m}\left(2\,\frac{\partial}{\partial\kappa}s^2+2\,(s-y_i)\,\frac{\partial^2}{\partial\kappa^2}s\right) & \sum_{i=1}^{m}\left(2\,\frac{\partial}{\partial\alpha}s\,\frac{\partial}{\partial\kappa}s+2\,(s-y_i)\,\frac{\partial^2}{\partial\alpha\partial\kappa}s\right) & \sum_{i=1}^{m}\left(2\,\frac{\partial}{\partial\rho}s\,\frac{\partial}{\partial\kappa}s+2\,(s-y_i)\,\frac{\partial^2}{\partial\kappa\partial\rho}s\right) \\ \sum_{i=1}^{m}\left(2\,\frac{\partial}{\partial\alpha}s\,\frac{\partial}{\partial\kappa}s+2\,(s-y_i)\,\frac{\partial^2}{\partial\alpha\partial\kappa}s\right) & \sum_{i=1}^{m}\left(2\,\frac{\partial}{\partial\alpha}s^2+2\,(s-y_i)\,\frac{\partial^2}{\partial\alpha^2}s\right) & \sum_{i=1}^{m}\left(2\,\frac{\partial}{\partial\rho}s\,\frac{\partial}{\partial\alpha}s+2\,(s-y_i)\,\frac{\partial^2}{\partial\alpha\partial\rho}s\right) \\ \sum_{i=1}^{m}\left(2\,\frac{\partial}{\partial\rho}s\,\frac{\partial}{\partial\kappa}s+2\,(s-y_i)\,\frac{\partial^2}{\partial\kappa\partial\rho}s\right) & \sum_{i=1}^{m}\left(2\,\frac{\partial}{\partial\rho}s\,\frac{\partial}{\partial\alpha}s+2\,(s-y_i)\,\frac{\partial^2}{\partial\alpha\partial\rho}s\right) & \sum_{i=1}^{m}\left(2\,\frac{\partial}{\partial\rho}s^2+2\,(s-y_i)\,\frac{\partial^2}{\partial\rho^2}s\right) \end{array}\right] \quad (7)$$

On triche : on supprime la somme dans la définition de f, qui dépend maintenant d'un point (x[i],y[i])

```
> f(kappa,alpha,rho) := (kappa/(1+exp(alpha-rho*x[i]))-y[i])**2;
```

$$f(\kappa,\alpha,\rho) := \left(\frac{\kappa}{1+e^{-\rho\,x_i+\alpha}}-y_i\right)^2 \quad (8)$$

On recalcule la Hessienne (on ne l'affiche pas - elle est horrible)

```
> H := VectorCalculus:-Hessian(f(kappa,alpha,rho),[kappa,alpha,rho]
  ):
```

Pour fixer les idées, voici H[1,2], qui vaut -t8 dans le code Python plus bas. Mêmes remarques que précédemment concernant le code Python.

```
> H[1,2];
```

$$-\frac{2\,\kappa\,e^{-\rho\,x_i+\alpha}}{\left(1+e^{-\rho\,x_i+\alpha}\right)^3}-\frac{2\left(\frac{\kappa}{1+e^{-\rho\,x_i+\alpha}}-y_i\right)e^{-\rho\,x_i+\alpha}}{\left(1+e^{-\rho\,x_i+\alpha}\right)^2} \quad (9)$$

```
> CodeGeneration:-Python(H,optimize,resultname='H');
t1 = math.exp(-rho * x[i - 1] + alpha)
t2 = 1 + t1
t2 = 0.1e1 / t2
t3 = kappa * t2
t4 = t3 - y[i - 1]
t5 = t2 ** 2
t6 = t5 * t1
t3 = t6 * (t3 + t4)
t7 = 2
t8 = t7 * t3
t3 = t7 * t3 * x[i - 1]
t6 = t6 * kappa
t6 = t6 * (t6 - t4)
t1 = 4 * t4 * kappa * t2 * t5 * t1 ** 2
t2 = -t6 * t7 * x[i - 1] - t1 * x[i - 1]
t4 = x[i - 1] ** 2
H = numpy.mat([[t7 * t5,-t8,t3],[-t8,t6 * t7 + t1,t2],[t3,t2,t4
* t6 * t7 + t1 * t4]])
```

On a aussi besoin de - J où J désigne la jacobienne de la fonction f

```
> m_J := VectorCalculus:-Jacobian(< f(kappa,alpha,rho) >,[kappa,
  alpha,rho]);
```

$$m_J := \left[\frac{2\left(\frac{\kappa}{1+e^{-\rho\,x_i+\alpha}}-y_i\right)}{1+e^{-\rho\,x_i+\alpha}},\; -\frac{2\left(\frac{\kappa}{1+e^{-\rho\,x_i+\alpha}}-y_i\right)\kappa\,e^{-\rho\,x_i+\alpha}}{\left(1+e^{-\rho\,x_i+\alpha}\right)^2},\right. \quad (10)$$

$$\left[ \frac{2\left(\frac{\kappa}{1+e^{-\rho x_i+\alpha}} - y_i\right)\kappa\, x_i\, e^{-\rho x_i+\alpha}}{\left(1+e^{-\rho x_i+\alpha}\right)^2} \right]$$

```
> CodeGeneration:-Python(m_J,optimize,resultname='b');
t1 = math.exp(-rho * x[i - 1] + alpha)
t2 = 1 + t1
t2 = 0.1e1 / t2
t3 = 2
t3 = t3 * (kappa * t2 - y[i - 1])
t4 = t3 * kappa * t2 ** 2
b = numpy.mat([[t3 * t2,-t4 * t1,t4 * x[i - 1] * t1]])
```