Corrige-TP5-3IMACS

November 29, 2016

1 TP5: Problème de moindres carrés

1.1 Objectifs

Le but de ce TP est d'ajuster des paramètres d'une fonction de manière à ce qu'elle approche un ensemble de données. Prenons l'exemple du nombre total de mégatonnes (Mt) de carbone dûes aux énergies fossiles émises chaque année depuis 1751 (source: http://cdiac.ornl.gov/trends/emis/meth_reg.html). On comprend l'intérêt "vital" d'un modèle capable de "prédire" l'évolution dans le temps d'une telle quantité.

L'émission globale de carbone rejetée dans l'atmosphère est mesurée en différents instants t_n, c'est-à-dire qu'à chaque t_n correspond une mesure m_n, pour $1 \leq n \leq N$. On note $e(x_1, x_2..., x_p, t)$ la fonction choisie comme modèle. Les paramètres x_i sont les paramètres de la fonction à ajuster au mieux : ce sont les inconnues du problème. Pour évaluer ce jeu de coefficients $x = (x_1, x_2, .., x_p)$, on peut minimiser l'erreur quadratique:

$$f(x) = \frac{1}{2} \sum_{n=1}^{N} \left(e(x, t_n) - m_n\right)^2 .$$

On cherche ainsi à trouver le jeu de coefficients x qui minimise f. Sachant que f est positive, on cherche donc le x qui réalise au mieux l'équation $f(x) = 0$ mais on ne cherche pas à résoudre directement $f(x) = 0$, car en général cette équation n'a pas de solution.

1.2 Travail de préparation (à faire chez soi)

Si on pose

$$r_n(x) = e(x, t_n) - m_n,$$

alors chaque r_n est une application de $\mathbb{R}^p$ dans $\mathbb{R}$ et

$$f(x) = \frac{1}{2} \sum_{n=1}^{N} r_n^2$$

Calculer les expressions générales de f, de $\nabla f(x)$ et de la hessienne $H[f](x)$, en fonction des vecteurs suivants :

$$R = \begin{pmatrix} r_1 \\ \vdots \\ r_N \end{pmatrix}, DR_i = \begin{pmatrix} \frac{\partial r_1}{\partial x_i} \\ \vdots \\ \frac{\partial r_N}{\partial x_i} \end{pmatrix}, HR_{ij} = \begin{pmatrix} \frac{\partial^2 r_1}{\partial x_i x_j} \\ \vdots \\ \frac{\partial^2 r_N}{\partial x_i x_j} \end{pmatrix}$$

et vous montrerez que :

$$\begin{cases} f = 0.5 * (R \cdot R) \\ (\nabla f)_i = (DR_i \cdot R) \\ (H[f])_{i,j} = (DR_i \cdot DR_j) + (HR_{ij} \cdot R) \end{cases},$$

où $(U \cdot V)$ est le produit scalaire des vecteurs U et V, donné par la formule :

$$(U \cdot V) = U^T V = \sum_i U_i V_i$$

Donner l'expression explicite de r, DR_i, HR_{ij}, $0 \leq i, j \leq 1$ pour le modèle suivant :

$$e(x_0, x_1, t) = x_0 \exp(x_1 t).$$

Pour simplifier les equations, vous pourrez utiliser le vecteur E suivant

$$E = \begin{pmatrix} \exp(x_1 t_1) \\ \exp(x_1 t_2) \\ \vdots \\ \exp(x_1 t_N) \end{pmatrix}$$

1.3 Travail d'implémentation

1.3.1 Traitement du modèle non linéaire

Pour résoudre l'équation $\nabla f(x) = 0$ associée au modèle, nous utiliserons la méthode de Newton, qui s'écrit dans ce contexte:

$$x^{(k+1)} = x^{(k)} + d^{(k)},$$

où $x^{(k)} = (x_0^{(k)}, x_1^{(k)})$ et $d^{(k)} = (d_0^{(k)}, d_1^{(k)})$ est la solution du système linéaire carré 2 x 2

$$Hf\left(x^{(k)}\right) d^{(k)} = -\nabla f\left(x^{(k)}\right).$$

Vous aurez besoin du fichier TP5-mesures.data (disponible sur la page Moodle du cours) qui contient toutes les données.

A faire: Le code suivant charge le fichier TP5-mesures.data et le met dans le tableau data. Affichez le tableau data. La première colonne est l'échelle de temps, et vous devez faire en sorte de changer l'échelle de temps pour que $t_0 = 0$, ce qui correspond à retirer la valeur 1751 à tous les coefficients de la première colonne de data.

```
In [21]: import numpy as np
         from numpy.linalg import *
         import matplotlib.pyplot as plt
         %matplotlib inline

         myfile = open('TP5-mesures.data')
         data = np.loadtxt(myfile)
         data[:,0]-=1751.
```

A faire : Finissez d'implémenter la fonction FoncNewt(x,data), où data est le tableau de taille $[N, 2]$ calculé précédemment. La fonction doit rendre la valeur de f, la valeur de $\nabla f(x)$ et la valeur de la Hessienne $H[f](x)$ quand on choisit :

$$r_n = e(x_0, x_1, t_n) - m_n \quad \text{et} \quad e(x_0, x_1, t) = x_0 \exp(x_1 t).$$

Vérifiez que le programme FoncNewt([11,0.1],data) donne comme réponse :

$$f = 7.017e+24, \nabla f = \begin{pmatrix} 1.276e+24 \\ 3.543e+27 \end{pmatrix} \quad ; \quad H[f] = \begin{pmatrix} 1.160e+23 & 6.443e+26 \\ 6.443e+26 & 1.790e+30 \end{pmatrix}$$

```
In [22]: def FoncNewt(x,data):
             T = data[:,0]
             M = data[:,1]
             E = np.exp(x[1]*T)
             R = x[0]*E-M
             f = 0.5*np.dot(R,R)
             DR = np.zeros((2,len(R)))
             DR[0,:] = E
             DR[1,:] = x[0]*E*T
             Gradf = np.dot(DR,R)
             HR11 = x[0]*T**2*E
             HR01 = T*E
             Hessf = np.zeros((2,2))
             Hessf[0,0] = np.dot(DR[0,:],DR[0,:])
             Hessf[0,1] = np.dot(DR[1,:],DR[0,:])+np.dot(R,HR01)
             Hessf[1,0] = Hessf[0,1]
             Hessf[1,1] = np.dot(DR[1,:],DR[1,:])+np.dot(R,HR11)
             return [f,Gradf,Hessf]

         f,gf,Hf = FoncNewt([11,0.1], data)
         print "f :", f
         print "gf :", gf
         print "Hf :", Hf
```

```
f : 7.01726062278e+24
gf : [  1.27586558e+24   3.54348289e+27]
Hf : [[  1.15987781e+23   6.44269619e+26]
 [  6.44269619e+26   1.79004022e+30]]
```

A faire : Implémentez l'algorithme de Newton. On commencera à $x^{(0)} = (11, 0.1)$ et on arrêtera la méthode quand le nombre d'itérations devient plus grand que nitermax=500 ou que la norme de ∇f devient plus petite que eps=2e-4. Votre algorithme imprimera à chaque itération le numéro de l'itération en cours, la norme du gradient et la valeur de f.

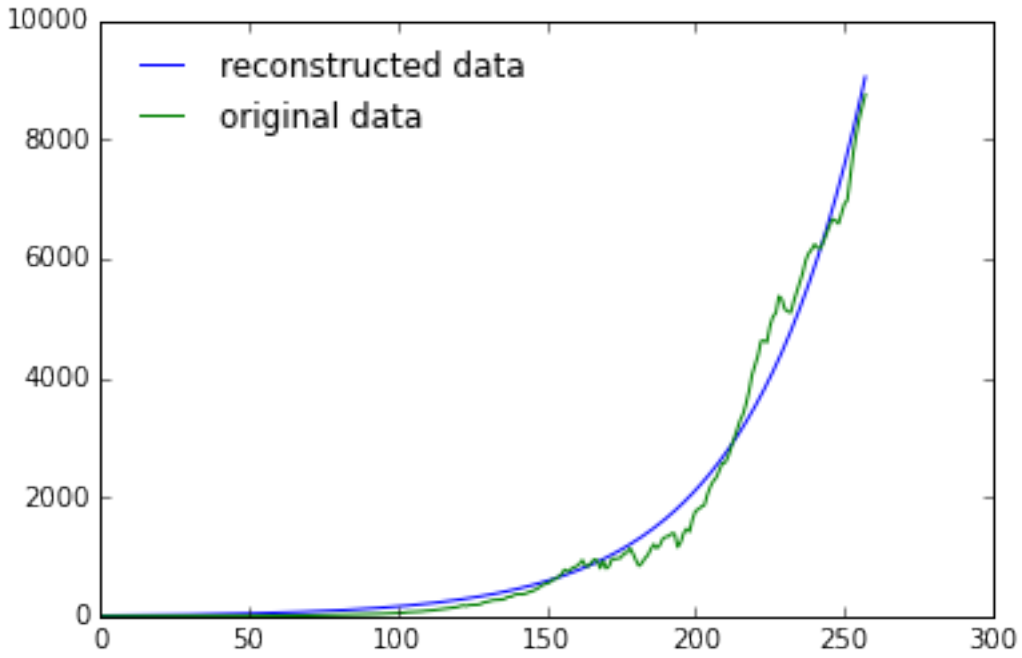
```
In [23]: def Newton_algorithm(xini,data,eps,nitermax):
             x=xini
             niter=0
             err=2*eps
             while (niter<nitermax) & (err>eps) :
                 niter+=1
                 [f,gradf,Hess]=FoncNewt(x,data)
                 print 'iteration =',niter,' f=',f,' grad=',np.linalg.norm(gradf),' x=',x
                 x=x-np.linalg.solve(Hess,gradf)
                 err=np.linalg.norm(gradf)
             return [x,niter]

         [x,niter] = Newton_algorithm([11,0.1], data, 2.e-4, 500)
```

```
iteration = 1  f= 7.01726062278e+24  grad= 3.54348312171e+27  x= [11, 0.1]
iteration = 2  f= 2.58302706533e+24  grad= 1.30382236298e+27  x= [ 11.0043012   0.0980189]
iteration = 3  f= 9.50825950441e+23  grad= 4.79744363352e+26  x= [ 11.00878401   0.09603697]
iteration = 4  f= 3.50013265678e+23  grad= 1.76524483062e+26  x= [ 11.01346022   0.09405419]
iteration = 5  f= 1.28848729926e+23  grad= 6.49536949897e+25  x= [ 11.01834268   0.0920705 ]
iteration = 6  f= 4.74338913886e+22  grad= 2.39004938125e+25  x= [ 11.0234454    0.09008588]
iteration = 7  f= 1.74626935606e+22  grad= 8.79456295098e+24  x= [ 11.0287837    0.08810028]
```

```
iteration = 8  f= 6.42907581598e+21  grad= 3.23613273234e+24  x= [ 11.03437436   0.08611365]
iteration = 9  f= 2.36701907203e+21  grad= 1.19081252974e+24  x= [ 11.0402358    0.08412594]
iteration = 10  f= 8.71509285089e+20  grad= 4.38193397118e+23  x= [ 11.04638828   0.0821371 ]
iteration = 11  f= 3.20893255578e+20  grad= 1.61247881898e+23  x= [ 11.05285416   0.08014706]
iteration = 12  f= 1.18159551787e+20  grad= 5.93373833979e+22  x= [ 11.05965815   0.07815578]
iteration = 13  f= 4.35109159215e+19  grad= 2.18358152839e+22  x= [ 11.06682764   0.07616317]
iteration = 14  f= 1.60232395378e+19  grad= 8.03558652846e+21  x= [ 11.07439305   0.07416916]
iteration = 15  f= 5.90101559881e+18  grad= 2.95715057175e+21  x= [ 11.0823883    0.07217368]
iteration = 16  f= 2.17334748899e+18  grad= 1.08827222238e+21  x= [ 11.09085125   0.07017664]
iteration = 17  f= 8.00495814666e+17  grad= 4.00507332088e+20  x= [ 11.09982435   0.06817794]
iteration = 18  f= 2.94861213235e+17  grad= 1.47398398871e+20  x= [ 11.10935524   0.06617749]
iteration = 19  f= 1.08618906883e+17  grad= 5.42481444514e+19  x= [ 11.11949758   0.06417518]
iteration = 20  f= 4.00148872237e+16  grad= 1.99658175423e+19  x= [ 11.1303119    0.06217089]
iteration = 21  f= 1.47422223849e+16  grad= 7.34850966611e+18  x= [ 11.14186658   0.06016451]
iteration = 22  f= 5.43152909389e+15  grad= 2.70470910356e+18  x= [ 11.15423893   0.05815592]
iteration = 23  f= 2.0011719549e+15  grad= 9.95517438239e+17  x= [ 11.16751632   0.05614499]
iteration = 24  f= 7.3726373616e+14  grad= 3.6642061461e+17  x= [ 11.18179715   0.05413163]
iteration = 25  f= 2.71574947705e+14  grad= 1.3486696823e+17  x= [ 11.19719168   0.05211579]
iteration = 26  f= 1.00000495219e+14  grad= 4.96376886564e+16  x= [ 11.21382218   0.05009749]
iteration = 27  f= 3.67975587284e+13  grad= 1.82672270299e+16  x= [ 11.23182198   0.04807689]
iteration = 28  f= 1.35238875072e+13  grad= 6.72118135474e+15  x= [ 11.25133243   0.04605441]
iteration = 29  f= 4.95962120539e+12  grad= 2.47204779005e+15  x= [ 11.27249645   0.04403091]
iteration = 30  f= 1.8121222903e+12  grad= 9.08617726274e+14  x= [ 11.29544645   0.042008  ]
iteration = 31  f= 657949619976.0  grad= 3.3358422475e+14  x= [ 11.32028346   0.03998858]
iteration = 32  f= 236360321988.0  grad= 1.22226092654e+14  x= [ 11.3470426   0.0379777]
iteration = 33  f= 83395735987.5  grad= 4.46291285937e+13  x= [ 11.37563693   0.03598404]
iteration = 34  f= 28540043247.8  grad= 1.61966364107e+13  x= [ 11.40576641   0.03402242]
iteration = 35  f= 9268986418.35  grad= 5.81351970443e+12  x= [ 11.43676424   0.03211802]
iteration = 36  f= 2747573252.51  grad= 2.04347246588e+12  x= [ 11.4673178   0.0303135]
iteration = 37  f= 692294499.113  grad= 688322476364.0  x= [ 11.49489825   0.02868039]
iteration = 38  f= 131763441.132  grad= 210628636445.0  x= [ 11.51437111   0.02733337]
iteration = 39  f= 20396799.4511  grad= 50538368717.2  x= [ 11.51326881   0.02642709]
iteration = 40  f= 9700003.28002  grad= 6228234607.61  x= [ 11.43974146   0.02606276]
iteration = 41  f= 9801734.43961  grad= 2645817818.03  x= [ 10.11492786   0.02648161]
iteration = 42  f= 9550552.28362  grad= 2334214946.14  x= [ 11.28202377   0.02603078]
iteration = 43  f= 9442750.83232  grad= 901414483.243  x= [ 12.08202609   0.02576086]
iteration = 44  f= 9414695.99011  grad= 444405720.082  x= [ 12.67430538   0.02556622]
iteration = 45  f= 9411596.47819  grad= 47481720.9684  x= [ 12.87586516   0.02550464]
iteration = 46  f= 9411539.51257  grad= 1343853.81013  x= [ 12.91000926   0.02549408]
iteration = 47  f= 9411539.48714  grad= 486.115201221  x= [ 12.91066211   0.02549388]
iteration = 48  f= 9411539.48714  grad= 0.000220417989318  x= [ 12.91066244   0.02549388]
iteration = 49  f= 9411539.48714  grad= 6.73532520862e-05  x= [ 12.91066244   0.02549388]
```

A faire : Visualisez dans un même graphique les données et votre solution.

```
In [24]: e = x[0]*np.exp(x[1]*data[:,0])
         plt.clf()
         plt.plot(data[:,0], e, label="reconstructed data")
         plt.plot(data[:,0], data[:,1], label="original data")
         plt.legend(loc='upper left', frameon=False)
         plt.show()
```

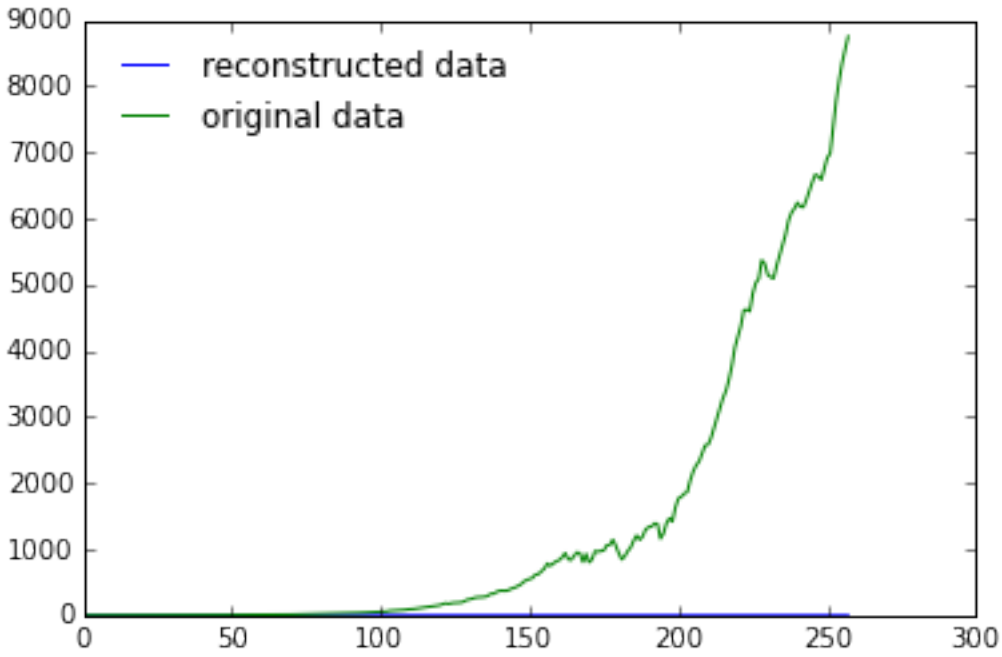


A faire : Etudiez l'effet des conditions initiales $x^{(0)}$ sur la solution obtenue.

```
In [25]: [x_tilde,niter] = Newton_algorithm([7,0],data,2.e-4,500)
         e_tilde = x[0]*np.exp(x_tilde[1]*data[:,0])
         r_tilde = e_tilde-data[:,1]
         f_tilde = 0.5*np.dot(r_tilde,r_tilde)
         plt.clf()
         plt.plot(data[:,0], e_tilde, label="reconstructed data")
         plt.plot(data[:,0], data[:,1], label="original data")
         plt.legend(loc='upper left', frameon=False)
         plt.show()
         print "L'algorithme ne converge pas vers la bonne solution."
```

```
iteration = 1  f= 829910473.0  grad= 539400811.3  x= [7, 0]
iteration = 2  f= 831414934.872  grad= 199880720.314  x= [  7.13316895e+00  -4.48964361e-03]
iteration = 3  f= 831981278.173  grad= 74242998.8581  x= [ 7.30455668 -0.00907126]
iteration = 4  f= 832196025.135  grad= 27656127.98  x= [ 7.5279235  -0.01377665]
iteration = 5  f= 832278191.322  grad= 10341420.0465  x= [ 7.81964971 -0.01864615]
iteration = 6  f= 832310012.044  grad= 3886958.58096  x= [ 8.19284121 -0.02373009]
iteration = 7  f= 832322544.358  grad= 1471221.78378  x= [ 8.63507062 -0.02908214]
iteration = 8  f= 832327598.161  grad= 561966.757401  x= [ 9.05021537 -0.03473385]
iteration = 9  f= 832329698.041  grad= 216912.088446  x= [ 9.18983023 -0.04066186]
iteration = 10  f= 832330594.528  grad= 84567.9144757  x= [ 8.7919804  -0.04686384]
iteration = 11  f= 832330987.965  grad= 33420.557301  x= [ 7.97647302 -0.05361572]
iteration = 12  f= 832331172.161  grad= 13564.6589172  x= [ 7.13956081 -0.06160454]
iteration = 13  f= 832331268.51  grad= 5732.36212022  x= [ 6.49305943 -0.07196537]
iteration = 14  f= 832331326.177  grad= 2526.99298649  x= [ 6.03452778 -0.08667827]
iteration = 15  f= 832331365.158  grad= 1140.83685029  x= [ 5.70748264 -0.10930215]
iteration = 16  f= 832331392.952  grad= 516.299966087  x= [ 5.47651394 -0.14525864]
iteration = 17  f= 832331412.753  grad= 233.944267827  x= [ 5.30532801 -0.20208566]
```

```
iteration = 18  f= 832331426.878  grad= 105.905515086  x= [ 5.12897262 -0.29169147]
iteration = 19  f= 832331436.826  grad= 47.3766277807  x= [ 4.87550214 -0.43159127]
iteration = 20  f= 832331443.536  grad= 20.8632851444  x= [ 4.51206504 -0.64341878]
iteration = 21  f= 832331447.876  grad= 9.0330866766  x= [ 4.06973766 -0.95512596]
iteration = 22  f= 832331450.569  grad= 3.80907138575  x= [ 3.62529421 -1.4020359 ]
iteration = 23  f= 832331452.125  grad= 1.55033139847  x= [ 3.2762002  -2.01581433]
iteration = 24  f= 832331452.922  grad= 0.610469448116  x= [ 3.08277794 -2.79993   ]
iteration = 25  f= 832331453.276  grad= 0.234660871181  x= [ 3.01589895 -3.7090979 ]
iteration = 26  f= 832331453.416  grad= 0.0883417631144  x= [ 3.00220661 -4.67816332]
iteration = 27  f= 832331453.469  grad= 0.0327959575928  x= [ 3.00028161 -5.66790784]
iteration = 28  f= 832331453.489  grad= 0.0121040505582  x= [ 3.0000369  -6.66432478]
iteration = 29  f= 832331453.496  grad= 0.00445801536821  x= [ 3.00000493 -7.66303206]
iteration = 30  f= 832331453.498  grad= 0.00164070838148  x= [ 3.00000066 -8.66255988]
iteration = 31  f= 832331453.499  grad= 0.000603676837362  x= [ 3.00000009 -9.66238663]
iteration = 32  f= 832331453.5  grad= 0.000222093000016  x= [  3.00000001 -10.66232295]
iteration = 33  f= 832331453.5  grad= 8.17051671786e-05  x= [  3.         -11.66229953]
```



L'algorithme ne converge pas vers la bonne solution.

1.3.2 Modèle linéarisé

Nous avons implémenté un algorithme qui recherche les valeurs (x_0, x_1) telles que :

$$r_n = e(x_0, x_1, t_n) - m_n,$$

soit le plus proche possible de 0 pour tous les n, en minimisant la somme des r_n^2.
On peut changer l'algorithme en essayant par exemple de minimiser la somme des $\tilde{r}_n^2$ avec $\tilde{r}_n$ défini par :

$$\tilde{r}_n = \ln(e(x_0, x_1, t_n)) - \ln(m_n).$$

Effectivement, r_n est nul si et seulement si $\tilde{r}_n$ est nul.

A faire : Vérifiez que, quitte à faire le changement d'inconnue $y = (ln(x_0), x_1)$ vous obtenez maintenant un problème de moindres carrés linéaire de la forme :

$$\min_y \|My - s\|^2$$

avec une matrice M et des données s que l'on déterminera.

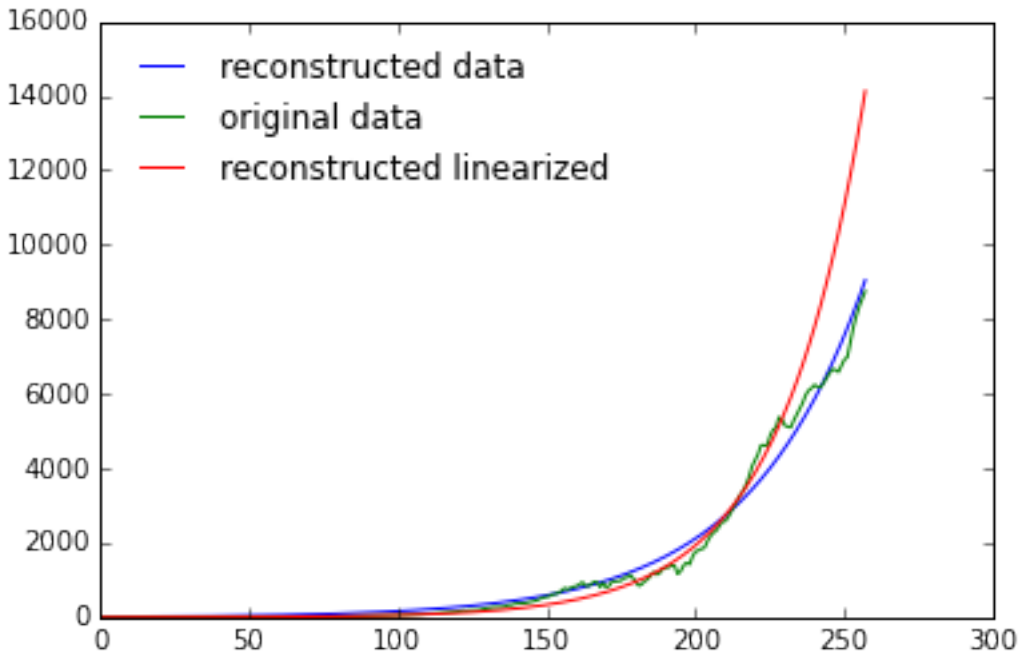
A faire : Créez une matrice M de taille $[N, 2]$, dont la première colonne est remplie avec des 1, et la seconde avec les t_n, $1 \leq n \leq N$. Créez le vecteur s de taille $[N, 1]$ rempli avec les $\ln(m_n)$. Résoudre le système des équations normales associées à la matrice M et au second membre s.

In [26]:
```
M = np.ones(data.shape)
M[:,1] = data[:,0]
s = np.log(data[:,1])
coef = np.linalg.solve(np.dot(M.T,M),np.dot(M.T,s))
xl = np.array([np.exp(coef[0]), coef[1]])
print xl
```

```
[ 1.79919831  0.03489559]
```

A faire : Ajouter à la figure précédente cette nouvelle solution. Comparez les valeurs de f pour chaque solution.

In [27]:
```
e2 = xl[0]*np.exp(xl[1]*data[:,0])
rl = e-data[:,1]
fl = 0.5*np.vdot(rl,rl)
plt.plot(data[:,0], e, label="reconstructed data")
plt.plot(data[:,0], data[:,1], label="original data")
plt.plot(data[:,0], e2, label="reconstructed linearized")
plt.legend(loc='upper left', frameon=False)
plt.show()
```

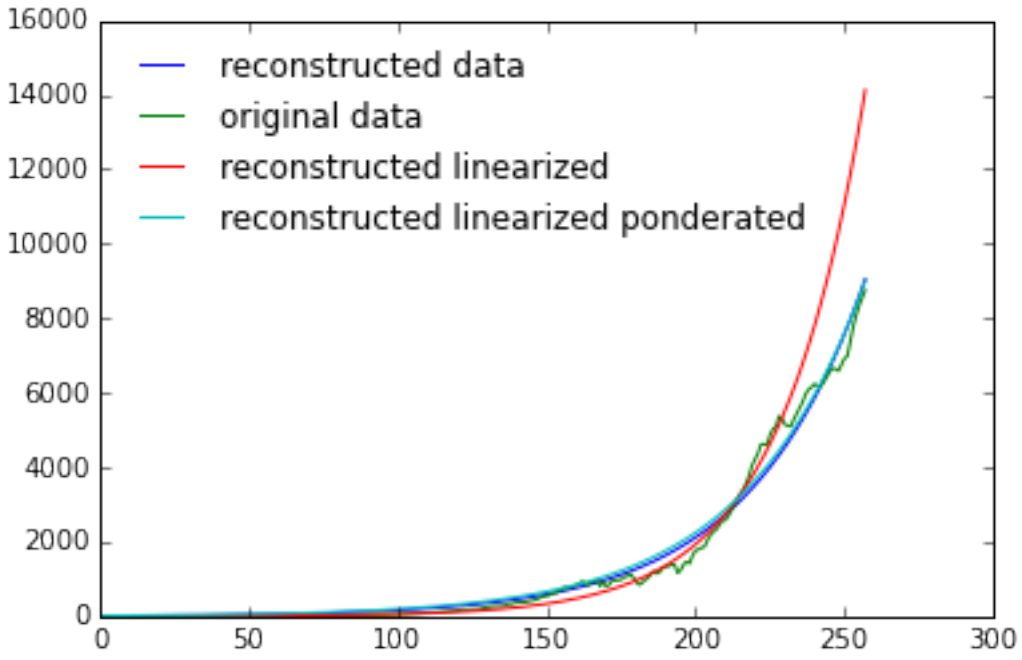


1.3.3 Modèle linéarisé pondéré

La fonction logarithme a tendance à écraser les grandes valeurs. On va pondérer l'approche linéaire en multipliant chaque équation du système surdéterminé $Mx = s$ pas la mesure m_n qui lui est associée, c'est-à-dire que l'on va résoudre le système $\tilde{M}x = \tilde{s}$, où la i^{eme} ligne de $\tilde{M}$ (resp $\tilde{s}$) est juste la i^{eme} ligne de $\tilde{s}$ multipliée par m_i.

A faire : Calculer la nouvelle solution. Comparer graphiquement et numériquement les solutions.

```
In [28]: Mp = np.ones(data.shape)
         sp = s*data[:,1]
         Mp[:,0] = M[:,0]*data[:,1]
         Mp[:,1] = M[:,1]*data[:,1]
         coef = np.linalg.solve(np.dot(Mp.T,Mp),np.dot(Mp.T,sp))
         xp = np.array([np.exp(coef[0]), coef[1]])
         e3 = xp[0]*np.exp(xp[1]*data[:,0])
         rp = e-data[:,1]
         fp = 0.5*np.vdot(rp,rp)
         plt.plot(data[:,0], e, label="reconstructed data")
         plt.plot(data[:,0], data[:,1], label="original data")
         plt.plot(data[:,0], e2, label="reconstructed linearized")
         plt.plot(data[:,0], e3, label="reconstructed linearized ponderated")
         plt.legend(loc='upper left', frameon=False)
         plt.show()
         print x,xl,xp
         print f,fl,fp
```




```
[ 12.91066244   0.02549388] [ 1.79919831  0.03489559] [ 17.1566762    0.02436853]
7.01726062278e+24 9411539.48714 9411539.48714
```

A faire : Une des deux approches linéaires permet-elle d'initialiser la méthode de Newton utilisée pour le traitement initial ?

La deuxième approche car elle est plus proche de la solution de la méthode de Newton