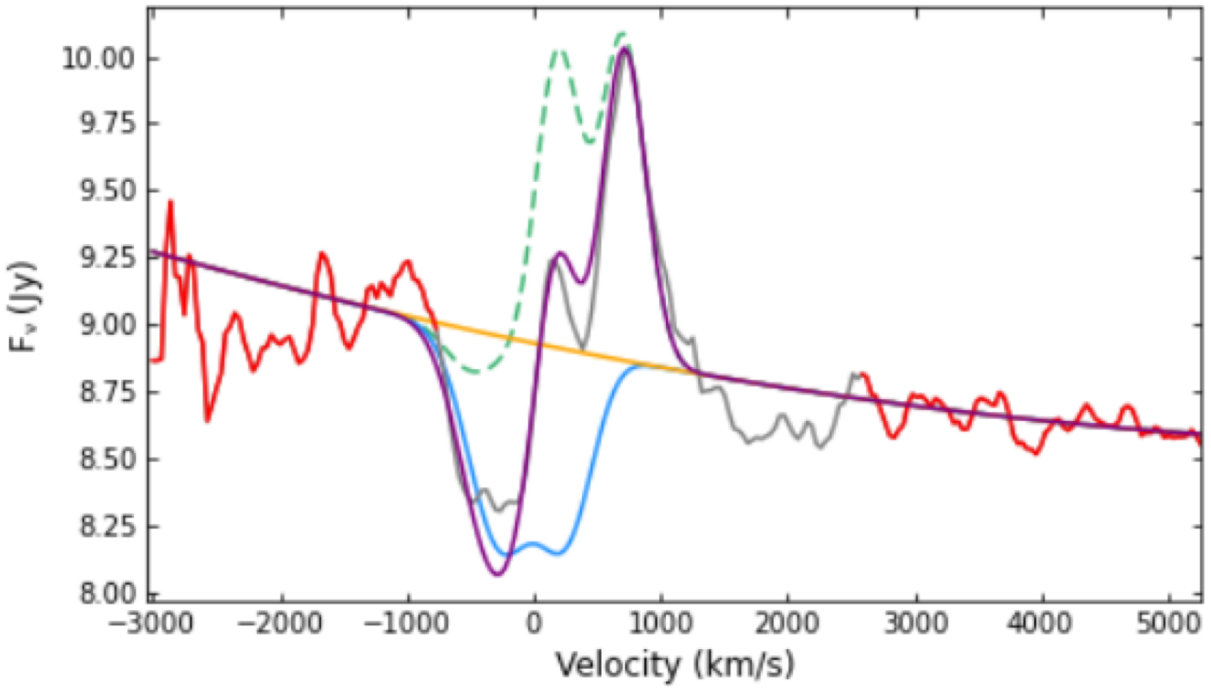
Συνηθισμένο πρόβλημα είναι η ανάγκη μοντελοποίησης δεδομένων, δηλαδή η προσαρμογή της βέλτιστης δυνατής **αναλυτικής συνάρτησης** σε αυτά, επιλέγοντας εκ των προτέρων τη μορφή της συνάρτησης. Π.χ., μπορεί να θέλουμε να κάνουμε fit δεδομένα χρησιμοποιώντας ευθεία γραμμή, Γκαουσιανή, πολυώνυμο, συνάρτηση Planck, συνδυασμό των παραπάνω ή άλλη γνωστή συνάρτηση.

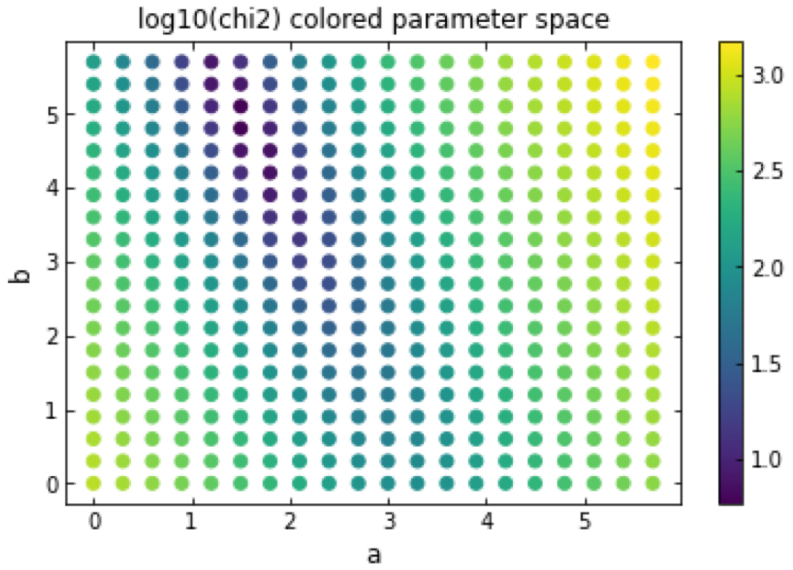
Π.χ., φάσμα με γραμμές εκπομπής + απορρόφησης λόγω διαφορετικού Doppler αερίου, και πολυωνυμικού συνεχούς



Επίλυση χ^2 με υπολογιστικό τρόπο… Υπολογίζοντας το χ^2 όλων των δυνατών λύσεων ή στο χώρο των παραμέτρων. Ξεκινώντας από το γραμμικό πρόβλημα…

```
plt.scatter(grid[:,0],grid[:,1],c=np.log10(chi2))
plt.xlabel('a'),plt.ylabel('b')
plt.title('log10(chi2) colored parameter space')
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7f8fefa0a490>
```




```
print("chi2 range:", min(chi2),max(chi2))
```

```
chi2 range: 5.799052113942225 1494.8165262263683
```

Το χ^2 είναι μέτρηση της ποιότητας της προσαρμογής [measure of the goodness of the fit] με μεγάλο εύρος τιμών, που εξαρτώνται από πολλά πράγματα. Για να ομογενοποιηθεί περισσότερο χρησιμοποιούμε το reduced χ^2 που είναι το χ^2 ανά βαθμό ελευθερίας.

Δηλ. $\chi_v^2 = \chi^2 / v$, κι άρα $\chi_v^2 = \sum_{i=1}^{N} \frac{[f(x_i,\theta) - y_i]^2}{v\sigma_i^2}$

Όπου v= N – m, με N: αριθμό μετρήσεων, m=αριθμό παραμέτρων (2 στην προκειμένη).

Εάν το χρησιμοποιήσουμε με αυτό τον τρόπο

```
print("chi2 range:", min(chi2),max(chi2))
```

```
chi2 range: 1.1150765434260501 343.77586625964375
```

Με αντίστοιχο κώδικα

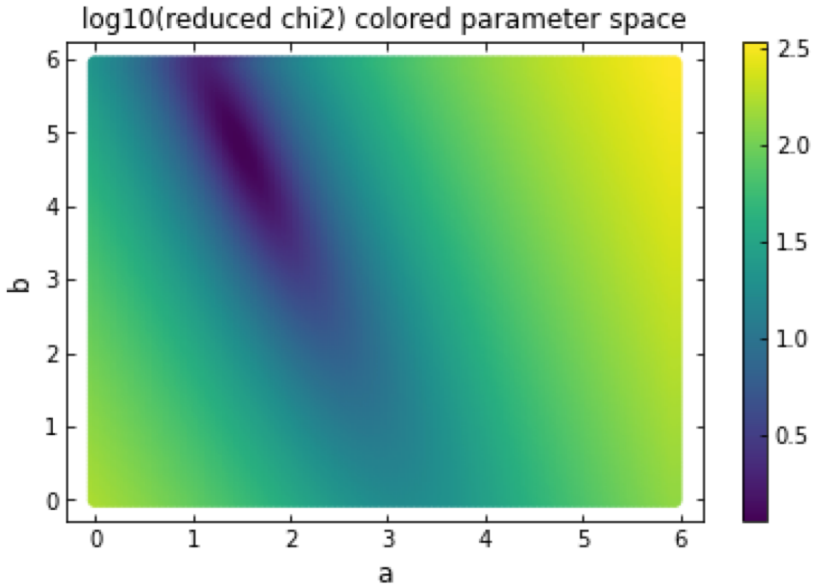
```
# εναλλακτική επίλυση με απλό parameter range sampling και minimization...

def ff(x,a,b): return a*x+b


def simple_chi2_parameter_sampling(ff,x,y,dy,lim1=0,lim2=1,pts=100,use_dof=False): #try also 100

    chi2=np.ones([pts,pts])
    ds=(lim2-lim1)/pts
    parameter_range1=np.arange(pts)*ds+lim1
    parameter_range2=parameter_range1  # same in this example for both parameters
    grid=[]

    for i in range(pts):
        for j in range(pts):
            function_to_minimize=ff(x,parameter_range1[i],parameter_range2[j])
            chi2[i,j] = np.sum( (y - function_to_minimize )**2 / dy**2 )
            if (use_dof): chi2[i,j] /= len(x)-2 # for 2 parameters in this example
            grid.append([parameter_range1[i],parameter_range2[j],chi2[i,j]])

    grid=np.array(grid)

    return grid


grid=simple_chi2_parameter_sampling(ff,x_lin,y_lin,dy_lin,lim2=6,use_dof=True)

chi2=grid[:,2]

minimum=np.where(chi2 == np.amin(chi2))[0]
a=grid[minimum,0][0]
b=grid[minimum,1][0]

print("a=",a,"b=",b)
```

```
a= 1.56 b= 4.8
```

Σημείωση: επαναλαμβάνοντας το προηγούμενο γραμμικό παράδειγμα, αλλά με 1000 σημεία. Αρχίζουμε να θεωρούμε την οπτικοποίηση του χώρου παραμέτρων σαν χάρτη πιθανοτήτων. Ο τρόπος αυτός θα μπορούσε να συνδυαστεί και με **Monte Carlo**, έτσι ώστε η κατασκευή του χώρου των παραμέτρων να μην γίνει γραμμικά και να αποφευχθούν τοπικά ελάχιστα αλλά και για να έχουν μεγαλύτερη ακρίβεια οι υπολογισμοί σε επιλεχθείσες περιοχές.

```
plt.scatter(grid[:,0],grid[:,1],c=np.log10(chi2))
plt.xlabel('a'),plt.ylabel('b')
plt.title('log10(reduced chi2) colored parameter space')
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7f8fe1e1aac0>
```

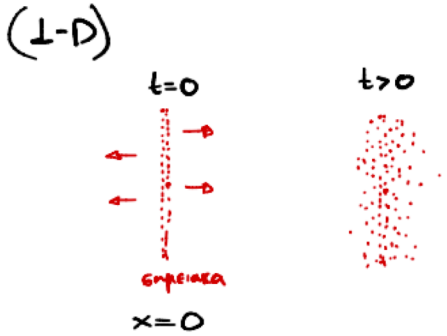


παράδειγμα: τυχαίος περίπατος: από τα πιο κλασσικά προβλήματα για Monte Carlo.

Αφορά σε αλλαγές κατεύθυνσης σε τυχαία γωνία που (συνήθως) οφείλονται σε σκέδαση λόγω κρούσεων ή δυνάμεων με/από σωματίδια.

Ανακαλύφθηκε μέσω της διαρκούς κίνησης κόκκων γύρης στο νερό με μικροσκόπιο, μελετήθηκε από τον Einstein

Συναντιέται σε πολλαπλά θέματα φυσικής και μη, π.χ., κίνηση ατόμων αερίου σε κουτί, διάχυση σταγόνας υγρού σε ένα άλλο (γάλα στον καφέ), διάδοση φωτονίων σε αστέρι.



(τυχαίος περίπατος random walk)

$$s = \begin{cases} +\ell & 50\ \% \\ -\ell & 50\ \% \end{cases}$$

$$\langle s \rangle = (-\ell)(0.5) + (+\ell)(0.5) = 0$$

$$\langle s^2 \rangle = (-\ell)^2(0.5) + (+\ell)^2(0.5) = \ell^2$$

$$\sigma^2 = \langle x^2 \rangle - \langle x \rangle^2 > 0$$

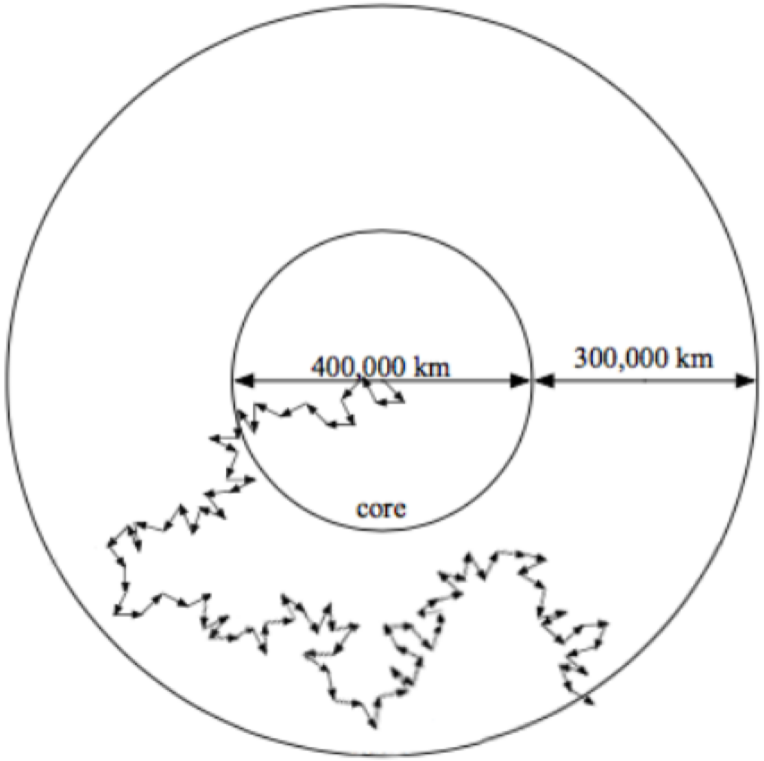
Τυπική απόκλιση σ = 1 βήμα

βασισμένο στις σημειώσεις Κ. Θεοφιλάτου

Τυχαίος περίπατος: διάδοση φωτονίων σε αστέρι

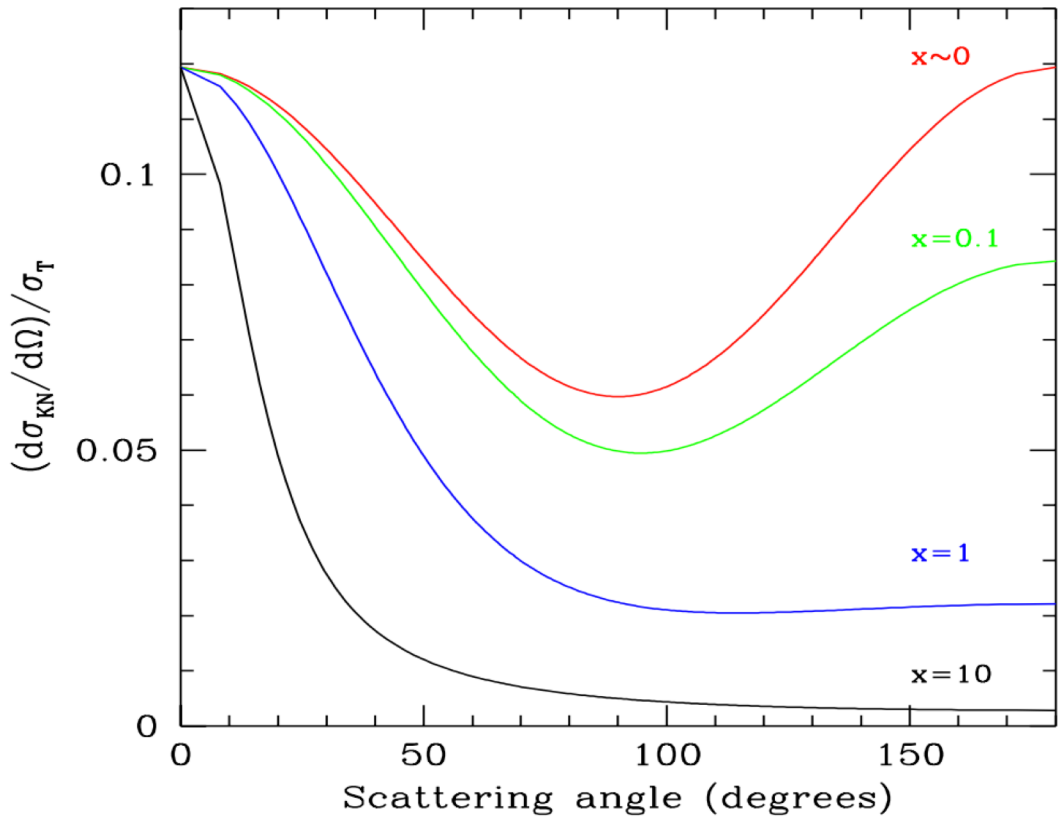
Έστω φωτόνια που **παρήχθησαν στο εσωτερικό αστεριού**.
Πόσο καιρό θα πάρουν να **φτάσουν στην επιφάνειά του** και να ταξιδέψουν προς τα εμάς
- Αν δεν αλληλεπιδρούσαν με την ύλη; (έστω ακτίνα = ηλιακή, 700.000km)
- Αν στο εσωτερικό σκεδάζονται με ελαστική σκέδαση Thompson.

Λάβετε υπ΄όψιν ότι το μήκος ελεύθερης διαδρομής που διανύουν πριν σκεδαστούν είναι



$$l = \frac{1}{n_e \sigma_T} \approx \frac{m_H}{\rho\, \sigma_T} \approx \frac{1.7 \times 10^{-24}\ \mathrm{g}}{1.4\ \mathrm{g\ cm^{-3}} \times 6.7 \times 10^{-25}\ \mathrm{cm^2}} \approx 2\ \mathrm{cm}\ ,$$

Θεωρήστε, για απλότητα, ότι σκεδάσεις γίνονται ομογενώς στις 360°, δηλ τα φωτόνια κινουνται και προς τα εμπρός και προς τα πίσω. Μετά εξετάστε το ενδεχόμενο τα φωτόνια να σκεδάζονται μόνο προς τα εμπρός.



```
import time
start = time.time()

def random_walk(steps=10000000,x0=0,y0=0):   # the photon starts from the center of the star
    mean_free_path=2 #cm
    x,y=x0,y0
    for i in range(steps):
        # for simplicity: photons assumed to be scattered as follows, but it depends on hv
        angle=round(random.random()*360)    # scattering at any angle
        #angle=round(random.random()*180)-90  # forward scattering
        x += mean_free_path*math.cos(np.pi*angle/180.)
        y += mean_free_path*math.sin(np.pi*angle/180.)
    return np.sqrt((x-x0)**2+(y-y0)**2)*1e-5,steps*mean_free_path*1e-5   #in km


def Monte_Carlo_reached_distance(n_iterations=5):
    how_far=[]

    for i in range(n_iterations):
        distance_from_center,overall_distance_walked = random_walk()
        how_far.append(distance_from_center)
        time = overall_distance_walked/2.9979e5
        print('distance from starting point (km):',round(distance_from_center,4), 'in time (sec):',round(time,

    return how_far


res = Monte_Carlo_reached_distance()
#res = Monte_Carlo_reached_distance(10000)
res = np.array(res)

print('Average escape time (in seconds):',np.mean(res[res>7000000]))

end = time.time()
print('Computational time:\t',end-start,'s')
```

```
distance from starting point (km): 0.0755 in time (sec): 0.00066713366
distance from starting point (km): 0.0547 in time (sec): 0.00066713366
distance from starting point (km): 0.0642 in time (sec): 0.00066713366
distance from starting point (km): 0.0421 in time (sec): 0.00066713366
distance from starting point (km): 0.0227 in time (sec): 0.00066713366
Average escape time (in seconds): nan
Computational time:      31.487586975097656 s
```

10 εκατομμύρια βήματα, σκέδαση σε όλες τις γωνίες:
< 0.1 km !!!

```
import time
start = time.time()

def random_walk(steps=10000000,x0=0,y0=0):   # the photon starts from the center of the star
    mean_free_path=2 #cm
    x,y=x0,y0
    for i in range(steps):
        # for simplicity: photons assumed to be scattered as follows, but it depends on hv
        #angle=round(random.random()*360)    # scattering at any angle
        angle=round(random.random()*180)-90  # forward scattering
        x += mean_free_path*math.cos(np.pi*angle/180.)
        y += mean_free_path*math.sin(np.pi*angle/180.)
    return np.sqrt((x-x0)**2+(y-y0)**2)*1e-5,steps*mean_free_path*1e-5   #in km


def Monte_Carlo_reached_distance(n_iterations=5):
    how_far=[]

    for i in range(n_iterations):
        distance_from_center,overall_distance_walked = random_walk()
        how_far.append(distance_from_center)
        time = overall_distance_walked/2.9979e5
        print('distance from starting point (km):',round(distance_from_center,4), 'in time (sec):',round(time,

    return how_far


res = Monte_Carlo_reached_distance()
#res = Monte_Carlo_reached_distance(10000)
res = np.array(res)

print('Average escape time (in seconds):',np.mean(res[res>7000000]))

end = time.time()
print('Computational time:\t',end-start,'s')
```

```
distance from starting point (km): 127.3221 in time (sec): 0.00066713366
distance from starting point (km): 127.3379 in time (sec): 0.00066713366
distance from starting point (km): 127.3042 in time (sec): 0.00066713366
distance from starting point (km): 127.304 in time (sec): 0.00066713366
distance from starting point (km): 127.3553 in time (sec): 0.00066713366
Average escape time (in seconds): nan
Computational time:      35.67810893058777 s
```

10 εκατομμύρια βήματα,
σκέδαση εμπρός: **100 km** !!!

… θα τα δούμε; Ένα τόσο απλό πείραμα, τόσο χρονοβορο!

Τελικά, (πότε) περιμένουμε νε δούμε απάντηση;

Για απλούστευση, σκεφτείτε το πρόβλημα σε 1D, όπως πριν. Για κάθε ένα από τα N τυχαία βήματα, η μετακίνηση l είναι +1 βήμα ή –1 βήμα, με τυπική απόκλιση (σφάλμα) ±1 βήμα ανά κίνηση. Τοτε το συνολικό σφάλμα, sqrt(N)l, είναι κι η μέγιστη δυνατή απόσταση που μπορεί να διανυθεί εφόσον η μέση τιμή είναι 0 κι η απομάκρυνση είναι λόγω διασποράς.

Κι εφόσον η μέγιστη απόσταση θέλουμε να είναι η ακτίνα του αστεριού

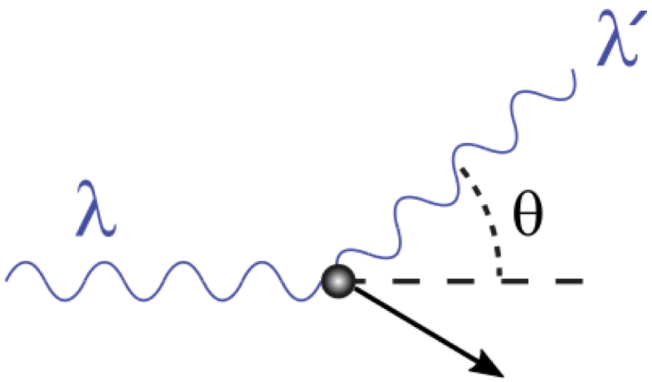
$$\sqrt{N}\,l = R_{star} \rightarrow \mathrm{N} = [\, R_{star} / l \sim 10^{10} \,]^2$$

Άρα πρέπει να καταφύγουμε σε πολύ μεγάλο αριθμό βημάτων,
10^{20} για να δούμε φωτόνιο να δραπετεύει (σε 1D).

Σε υπερυπολογιστές, βρίσκουμε την απάντηση να είναι ίση με **10.000– 10^7 χρόνια,** ανάλογα με γωνία σκεδασης, συχνότητα, αλλά και πυκνότητα αστεριού…

Άλλη εκδοχή

Εύκολη εξάρτηση από τη γωνία και χρήση probability density function ανά γωνία, αν τα φωτόνια έχουν υψηλές ενέργειες, π.χ., ακτίνες γ. Τότε στο εσωτερικό σκεδάζονται με ηλεκτρόνια βάσει φαινομένου Compton, που τους αλλάζει την ενέργεια συναρτήσει της γωνίας με τρόπο που σχετίζεται με το μήκος κύματος:



Built-in συναρτήσεις της Python για parametric fitting με χ^2 και σχετικά με αυτές,

Π.χ., η curve_fit στο πρόβλημα της ευθείας γραμμής

```
# εναλλακτική επίλυση με Python
#scipy.optimize.curve_fit

from scipy.optimize import curve_fit

popt = curve_fit(ff, x_lin, y_lin, sigma=dy_lin, p0=[1,1])[0]

print(popt[0],popt[1])
```

```
1.5341736729495603 4.8609456134567
```

```
help(curve_fit) # μοντέλο chi2, Levenberg-Marquant
```

```
        Method to use for optimization. See `least_squares` for more details.
        Default is 'lm' for unconstrained problems
```

Χρειάζεται διττή implementation: και να καλύπτει το χώρο των παραμέτρων εντός ορίων όπως δείξαμε πριν, αλλά να μπορεί να διαχειρίζεται αναλυτικά, με γρήγορο τρόπο κι εναλλασσόμενο βήμα την περίπτωση που δεν δοθούν όρια p ∈ $[-\infty, \infty]$ -> αλγόριθμος Levenberg-Marquant

→ απαιτεί ύπαρξη **παραγώγου** f ως προς παραμέτρους.

Built-in συναρτήσεις της Python για parametric fitting με χ^2 και σχετικά με αυτές,

Π.χ., το πακέτο lmfit στο πρόβλημα του Γκαουσιανού fit φάσματος

```
# άλλη εναλλακτική επίλυση με Python, στο πρώτο πρόβλημα Γκαουσιανής σε φάσμα.
from lmfit import Model
lmod = Model(Gauss_cont)  #δημιουργεί μια κλάση μοντέλο που ονομάζει και ξεχωρίζει τις παραμέτρους

lmod.set_param_hint('I0',value = max(I)*0.9,min = 0 ,max = max(I)*1.1)
lmod.set_param_hint('x0',value = 400 ,min = 200,max = 600)
lmod.set_param_hint('sigma',value = 100,min = 50, max = 300)
lmod.set_param_hint('a',value = 0 ,min = 0 ,max = 0.1)
lmod.set_param_hint('b',value = 0 ,min = -0.1,max = 0.1)

lfit = lmod.fit(I, x = x)
g_model_params=lfit.params
g_model=lfit.eval(x = x)

print(g_model_params['I0'].value,g_model_params['x0'].value,g_model_params['sigma'].value)
```

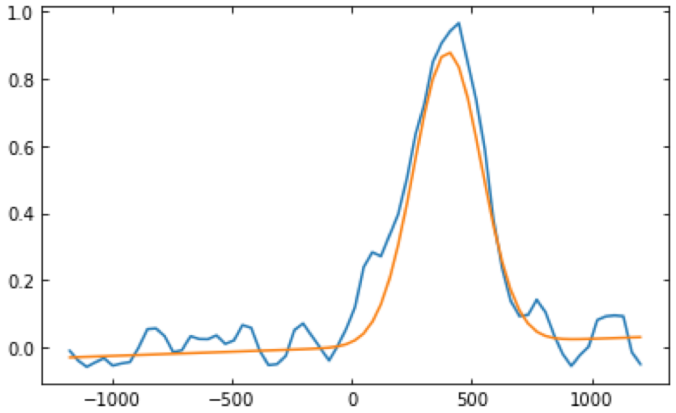
```
0.870172914716734 399.99999469979855 100.00002556585531
```

Built-in συναρτήσεις της Python για parametric fitting με χ^2 και σχετικά με αυτές,

Π.χ., το πακέτο lmfit στο πρόβλημα του Γκαουσιανού fit φάσματος

```
plt.plot(x,I)
plt.plot(x,g_model)
# εξηγηση του σκεπτικού στις παραπάνω δυο με Levenberg-Marquant
```

```
[<matplotlib.lines.Line2D at 0x7f8fe3bdb280>]
```



Non-Linear Least-Squares Minimization and Curve-Fitting for Python

Lmfit provides a high-level interface to non-linear optimization and curve fitting problems for Python. It builds on and extends many of the optimization methods of scipy.optimize. Initially inspired by (and named for) extending the Levenberg-Marquardt method from scipy.optimize.leastsq, lmfit now provides a number of useful enhancements to optimization and data fitting problems, including:

Μέθοδος Levenberg-Marquant

Επαναληπτική μέθοδος: έστω διάνυσμα παραμέτρων p

με αρχική τιμή $p^T = [1, 1, 1, \ldots, 1]$ εαν δεν υπάρχει άλλη γνώση

Για μεταβολή δ στο ελάχιστο: $f(x_i, p+\delta) \approx f(x_i, p) + J_i \delta$

και $\chi^2 = \sum_{i=1}^{N} [y_i - f(x_i, p) - J_i \delta]^2$, όπου $J_i = \frac{\partial f(x_i, p)}{\partial p}$

$\Rightarrow \chi^2 = |Y - F(p) - J\delta|^2 \leftrightarrow$ σε μορφή πινάκων, με χρήση ταυτοτήτων

Σκοπός μας είναι να βρούμε τη μεταβολή (διάνυσμα) δ που θα επιλεχθεί ώστε να οδηγήσει στο ελάχιστο χ^2.

$$\Rightarrow \chi^2 = [Y - F(p) - J\delta]^T [Y - F(p) - J\delta]$$
$$= ([Y - F(p)]^T - [J\delta]^T)([Y - F(p)] - J\delta)$$
$$= [Y - F(p)]^T [Y - F(p)] - [Y - F(p)]^T J\delta - [J\delta]^T [Y - F(p)] + \delta^T J^T J \delta$$

Μέθοδος Levenberg-Marquant

υπενθύμιση

$$(AB)^T = B^T A^T$$

$$\Rightarrow \chi^2 = [y - F(p)]^T [y - F(p)] - 2 [y - F(p)]^T J \delta + \delta^T J^T J \delta$$

Για ελαχιστοποίηση $\frac{\partial \chi^2}{\partial \delta} = 0$ άρα

$$2 [y - F(p)]^T J = \delta^T J^T J$$ και παίρνοντας ανάστροφο

$$2 J^T [y - F(p)] = (J^T J) \delta .$$

Συνεισφορά Levenberg: η χρήση ενός όρου "damping", λ
(όρος απόσβεσης: ταλάντωση, γραμμές φάσματα)

$$(J^T J + \lambda I) \delta = 2 J^T [y - F(p)]$$

διαφορετικός ∀ βήμα ώστε ο αλγόριθμος να προσεγγίσει γρήγορα τη λύση

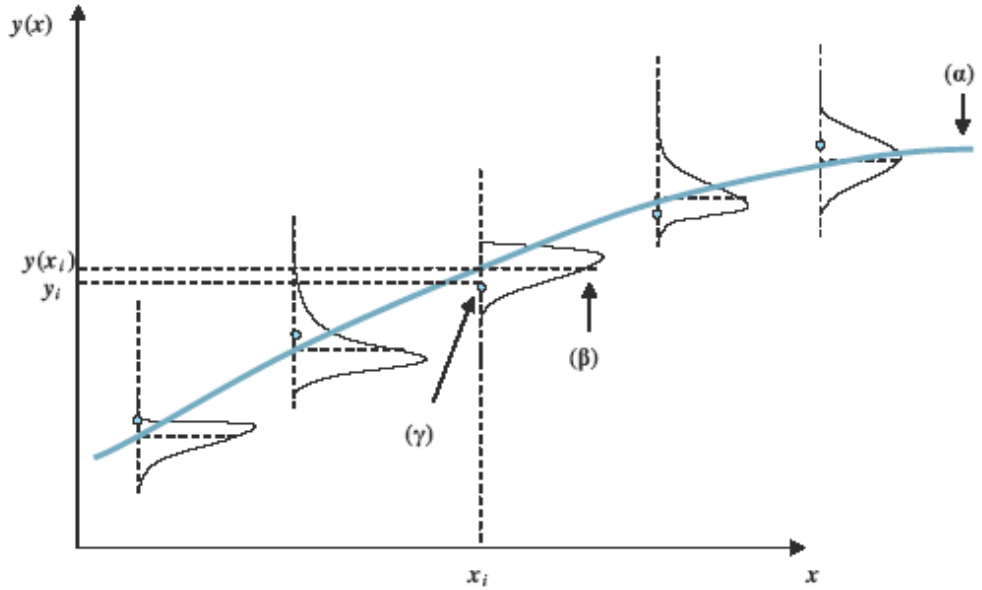
αποδοτική κίνηση στο χώρο παραμέτρων. Εξαρτάται από $\frac{\partial f}{\partial p}$

Ξανα-ανακαλύφθηκε από Marquant.

Μέθοδοι μέγιστης πιθανοφάνειας

Οι πιθανοκρατικές μέθοδοι βασίζονται στην παρατήρηση ότι η ελαχιστοποίηση του χ2 αντιστοιχεί σε μεγιστοποίηση πιθανότητας.

Έστω ότι φιτάρουμε σετ δεδομένων, για **το καθένα** από τα οποία η μέση τιμή αλλά και η διακύμανση γύρω από αυτή ακολουθούν **κανονική κατανομή**. Αυτό μπορεί να συμβαίνει πχ, γιατί τα σημεία προέκυψαν μετά από πολλές μετρήσεις για το καθένα (βλ. θεώρημα κεντρικού ορίου).



* See Data Analysis – Siegmund Brandt,
Data analysis recipes: fitting a model to data: Hogg, Bovy, Lang 2010

Μέθοδοι μέγιστης πιθανοφάνειας

Οι πιθανοκρατικές μέθοδοι βασίζονται στην παρατήρηση ότι η ελαχιστοποίηση του χ2 αντιστοιχεί σε μεγιστοποίηση πιθανότητας.

Conditional probability: πιθανότητα να βρεθεί το y_i δεδομένου του x_i ∀ σημείο και

$$P(y_i|x_i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(y_i - f(x_i))^2}{2\sigma_i^2}}$$

$$\mathcal{L} \equiv \text{likelihood} = \prod_{i=1}^{N} P(y_i|x_i) \quad \text{για όλο το dataset. Τότε}$$

$$\ln \mathcal{L} = \ln \left[\left(\frac{1}{\sqrt{2\pi}} \right)^N \prod_{i=1}^{N} \frac{1}{\sigma_i} e^{-\frac{(y_i - f(x_i))^2}{2\sigma_i^2}} \right]$$

$$= N \ln \frac{1}{\sqrt{2\pi}} - \ln \prod_{i=1}^{N} \sigma_i + \ln e^{-\sum_{i=1}^{N} \frac{(y_i - f(x_i))^2}{2\sigma_i^2}}$$

$$= C - \sum_{i=1}^{N} \frac{(y_i - f(x_i))^2}{2\sigma^2} = C - \frac{1}{2}\chi^2$$

$\Rightarrow$ likelyhood maximation = χ^2 minimization

Μέθοδοι μέγιστης πιθανοφάνειας

Βασικά πλεονεκτήματα των πιθανοκρατικών μεθόδων σε σχέση με το απλό χ^2

1. Η δυνατότητα χρήσης «prior knowledge» ή «updated knowledge» καθώς εξελίσσεται το πείραμα. Δηλαδή, μπορούμε να δίνουμε **βάρη** (όχι μόνο όρια) βάσει γνωστών ήδη πληροφοριών ώστε να κατευθύνουμε τον αλγόριθμο προς τη σωστή κατεύθυνση.

   Αν δεν γίνει χρήση πρωτύτερης γνώσης, το αποτέλεσμα ίδιο με χ^2

2. Χάρη στις πιθανότητες, μπορούμε όχι μόνο να δούμε ποια είναι η δεύτερη (και τρίτη...) κλπ καλύτερη λύση αλλά και πόσο πιθανό είναι αν αλλάξουμε μια μεταβλητή με καποιο τρόπο να βρούμε το σωστό αποτέλεσμα.

   Αντιθέτως, οι αλγόριθμοι χ^2 εγγενώς επιστρέφουν μόνο την καλύτερη λύση.

3. Μπορούμε να συγκρίνουμε ακόμα και 2 μοντέλα μεταξύ τους

   → **Bayesian statistics**

' Bayes describes the conditional probability of an event based on data as well as prior information or beliefs about the event or conditions related to the event (1763). Laplace developed the Bayesian interpretation of probability '

Στο πιθανοτικό framework, θέλουμε να μεγιστοποιήσουμε

το $\prod_{i=1}^{N} P(\alpha, \beta \mid y_i)$, όπου α και β οι παράμετροι του μοντέλου,

δηλ την πιθανότητα των α και β δεδομένων των μετρήσεων y_i:

$$\prod_{i=1}^{N} P(\alpha,\beta \mid y_i) = \prod_{i=1}^{N} \frac{P(y_i \mid \alpha,\beta) \cdot P(\alpha,\beta)}{P(y_i)}$$

από θεώρημα Bayes. Γενική μορφή για μοντέλο M, data D:

Likelihood: πιθανότητα συμβατότητας D με κάθε M

prior probability: πιθανότητα μοντέλου, συμπεριλαμβάνει προϋπάρχουσα γνώση

$$P(M \mid D) = \frac{P(D \mid M) \cdot P(M)}{P(D)}$$

posterior probability: πιθανότητα μοντέλου για τις δεδομένες μετρήσεις

Evidence: πιθανότητα μετρήσεων ανεξάρτητη μοντέλου ↔ κοινός παράγοντας κανονικοποίησης (παραλείπεται)

Απόδειξη θεωρήματος για δύο γεγονότα E_1, E_2:

Ορισμός conditional probability*: $P(E_1|E_2) = \frac{P(E_1 \cap E_2)}{P(E_2)}$

και $P(E_2|E_1) = \frac{P(E_2 \cap E_1)}{P(E_1)}$

Εφόσον η joint probability

$P(E_1 \cap E_2) = P(E_2 \cap E_1)$ τότε προκύπτει

$P(E_1|E_2) = \frac{P(E_2|E_1) \cdot P(E_1)}{P(E_2)}$

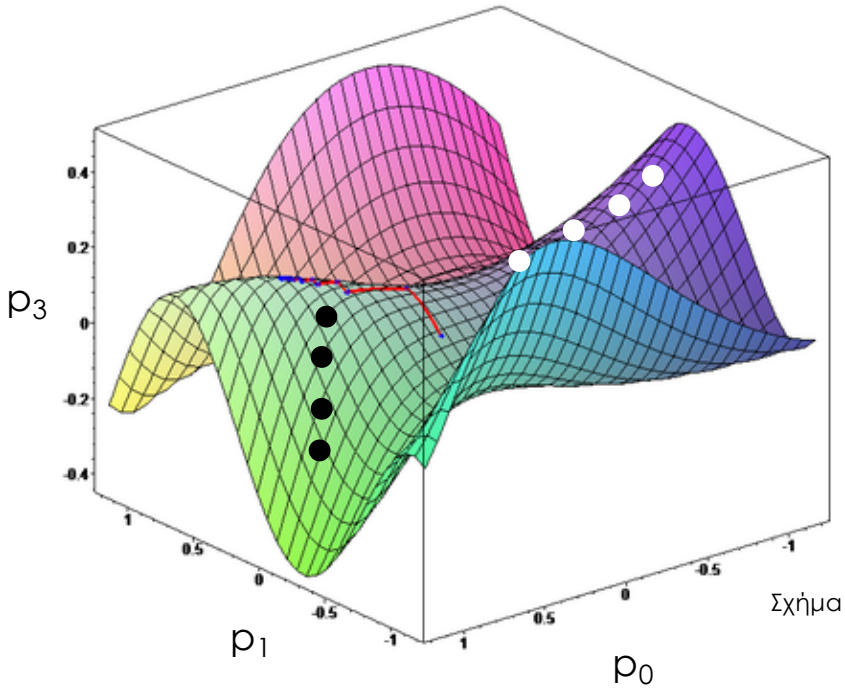
* ποσοστό κοινής πιθανότητας που αναστοιχεί σε ένα γεγονός

Έχοντας τις prior, evidence, και μπορώντας να υπολογίσουμε την likelihood για κάθε μοντέλο, βρίσκουμε την **posterior κι άρα τις βέλτιστες τιμές των παραμέτρων.**

Πακέτο emcee python. Συνδυάζει

- **Θεώρηση Bayesian** (μέσω ελαχιστοποίησης lnP)
- **Monte Carlo** για επιλογή και μεταβολή σημείων στο χώρο παραμέτρων σε κάθε χρόνο, με αλγόριθμο **τύπου Metropolis** για την επιλογή εύρους βήματος.
- Χρήση **Ν αριθμού περιπατητών (walkers)** που ακολουθούν τυχαίας κατεύθυνσης που δείχνουν την κλίση του χώρου
- **αλυσίδα βημάτων** που εξαρτάται μόνο από την τρέχουσα θέση σημείων – όχι τη παρελθοντική. Αλλά δεδομένου ότι τα βήματα σώζονται, η αλυσίδα μπορεί να ακολουθηθεί κι ανάποδα - Markov chain

Π.χ., κίνηση 2 walkers, 4 βημάτων, σε χώρο παραμέτρων



Σχήμα wikpedia

Metropolis (- Hastings) algorithm :

- Θέλουμε να βρούμε σε δείγμα επιδυτικής κατανομής πιθανότητας P με διάφορη PDF $f \sim P$

- Διαλέγουμε συνάρτηση βήματος $g(x)$, επίσης κανονικοποιημένη σαν PDF - flat, random, Gaussian για ψεσανίμηση εντός αυτών ...

- for $i = 1, \ldots,$ number_of_steps :

  α) βρίσκουμε νέο δείγμα x_i ως $g(x_{i-1})$

  β) ελέγχουμε αν $a = \frac{f(x_i)}{f(x_{i-1})}$

  γ) παράγουμε τυχαίο αριθμό $u \in [0,1]$ - από uniform

  δ) Αν $u < a$, κρατάμε νέα δέση. (accept / reject)

Αν $\frac{f(x_i)}{f(x_{i-1})} > 1$, πάντα $u < a \Rightarrow$ λύσεις προς τη σωστή κατεύθυνση επιλέγονται πάντα

Αν $a < 1$ και $u < a$: πάλι κρατάμε τη λύση, ώστε λύσεις προς άλλες κατευθύνσεις να επιτρέπονται τυχαία κι η επίλυση να μην "κολλάει" σε τοπικές λύσεις.

Αν $u > a$: $x_i = x_{i-1}$ (δεν προχωράει σε νέα δέση)

Likelihood maximization (via lnP minimization) with a Metropolis, Monte Carlo Markov chain (MCMC) algorithm

An example of efficient sampling of the parameter grid is the so-called Metropolis(-Hastings) algorithm. The Metropolis algorithm works by iteratively generating a random sample of values x (with a Monte Carlo algorithm) with a probability distribution that resembles more and more a desired probability distribution P(x) with every (time) step. It operates as follows.

• The algorithm chooses an arbitrary or a user-defined set of points for the initial state. Using a proposed probability density function g for the jump $g(x_{i+1}|x_i)$, it provides a new state. The function g can be random, fixed, Gaussian (if desired to make a step of a certain length near the old value), or any other function.

• The algorithm checks whether the new state has a distribution that resembles more the desired probability distribution P. For this purpose it uses a discretized form of the function P, so another PDF, i.e., a histogram depicting the relative likelihood of discrete values. Note that the Metropolis algorithm asks for the shape of this PDF to be provided - not its normalization, which can occasionally be hard to find. Then it computes the ratio α = PDF (x_{i+1}) / PDF(x_i) = P(x_{i+1})/P(x_i).

• The algorithm then draws a random number u from a uniform distribution in the range [0,1]: if $u \lesssim \alpha$ then it accepts the new position, else it stays in place. If the new position is more likely than the old position, then α is by default >1 and u always satisfies the criterion. Otherwise, the algoritmh proceeds by chance based on whether u satisfies the criterion. This allows for some randomness to help the solver visit low probability density regions. The algorithm, thus, proceeds by randomly moving in the sample space, sometimes accepting the moves and sometimes remaining in place.

The Metropolis is a so-called Markov Chain Monte Carlo chain, because it satisfies the criterium that Markov set for the description the evolution of a stochastic model. Markov's criterium is memoryless, meaning that any state depends ONLY on its immediate prior state, no past ones. A random walk of a photon in the Sun is such a process, as the next location only depends on the present location (plus the random step). A random draw of a card from a deck is non Markovian, as the card to be drawn explicitely depends on what is left on the deck from previous drawings. A chain is the representation of model that describes how one event evolved to the other (a series of lines connecting the different states). The Markov chain typically assumes discretized time steps and time reversability so the outome does not depend on the direction of the transitions.

Fit **ευθείας γραμμής με Bayesian** + Monte Carlo (emcee)

Πρώτα να κάνουμε fit ευθεία γραμμή στα παραπάνω linear data (x_lin, y_lin), με Bayesian τρόπο. Για να κανταλάβουμε την "ισχυ" των priors επί του αποτελέσματος, θα κάνουμε την εξής τροποποίηση στο πρόβλημα: θα φιτάρουμε χωρίς να έχουμε γνώση των μεμονωνμένων σφαλμάτων (dy_lin) αλλά μόνο ενός ενδεικτικού μέσου σφάλματος (1.5) και μετά, μέσω της prior, θα κατευθύνουμε τη λύση στις αναμενόμενες για τα σφάλματα τιμές των παραμέτρων. Για το λόγο αυτό, θα μοντελοποιήσουμε την prior probability σαν μια στενή Gaussian γύρω από τις αναμενόμενες τιμές (με σ_{theor} =0.1).

Υπενθύμιση ότι από χ^2 minimization η λύση (slope and intercept) της y= ax+b είναι:

a=2.04, b=3.10 # without errors and a=1.53, b=4.86 # with errors

Define the probabilities

Likelihood

Υποθέτοντας ότι τα πειραματικά σημεία που τραβηχτηκαν γύρω από το εκάστοτε μοντέλο αποκλίνουν από αυτό με Γκαουσιανό τρόπο, τότε για N=7 points με κοινό σφάλμα

$$\mathcal{L}(y_{data}|y_{model}) = \prod_{i=1}^{N} \frac{e^{-\frac{(y_i - y_{model(x_i,a,b)})^2}{2\sigma^2}})}{\sqrt{2\pi\sigma^2}} = \frac{e^{-\sum_{i=1}^{N}\frac{(y_i - y_{model(x_i,a,b)})^2}{2\sigma^2}}}{(2\pi\sigma^2)^{N/2}}$$

Priors

Θα δούμε 2 επιλογές:

- α and β uniformly distributed - flat prior = καθόλου prior
- α and β spread around the correct values in a Gaussian manner, so

$$Prior(\alpha,\beta) = Prior(\alpha)\,Prior(\beta) = \frac{1}{\sqrt{2\pi\sigma_{theor}^2}} e^{-\frac{(a-a_0)^2}{2\sigma_{theor}^2}} \frac{1}{\sqrt{2\pi\sigma_{theor}^2}} e^{-\frac{(b-b_0)^2}{2\sigma_{theor}^2}}$$

Χτίζουμε τις πιθανότητες

```python
# emcee requires log-posterior

def log_prior(params):
    alpha, beta=params
    sigma_dev=0.1
    return np.log( 1/2*np.pi*sigma_dev**2) - (alpha-1.53)**2/(2*sigma_dev**2) - (beta-4.86)**2/(2*sigma_dev**2)

# gaussian likelihood for each data point given a model, sum over all data points
def log_likelihood(params, x, y):
    alpha, beta = params
    sigma_typical = 1.5
    y_model = alpha* x + beta
    N=len(x)
    # note that the result should be the same with or without the second term of the sum = constant
    return -0.5 * np.sum((y-y_model)**2 / sigma_typical**2)  - (N/2)*np.log(2*np.pi* sigma_typical** 2)

def log_posterior(params, x, y):
    #return log_likelihood(params, x, y)   # compare to χ2 without errors
    return log_prior(params) + log_likelihood(params, x, y)  # compare to χ2 with errors
```

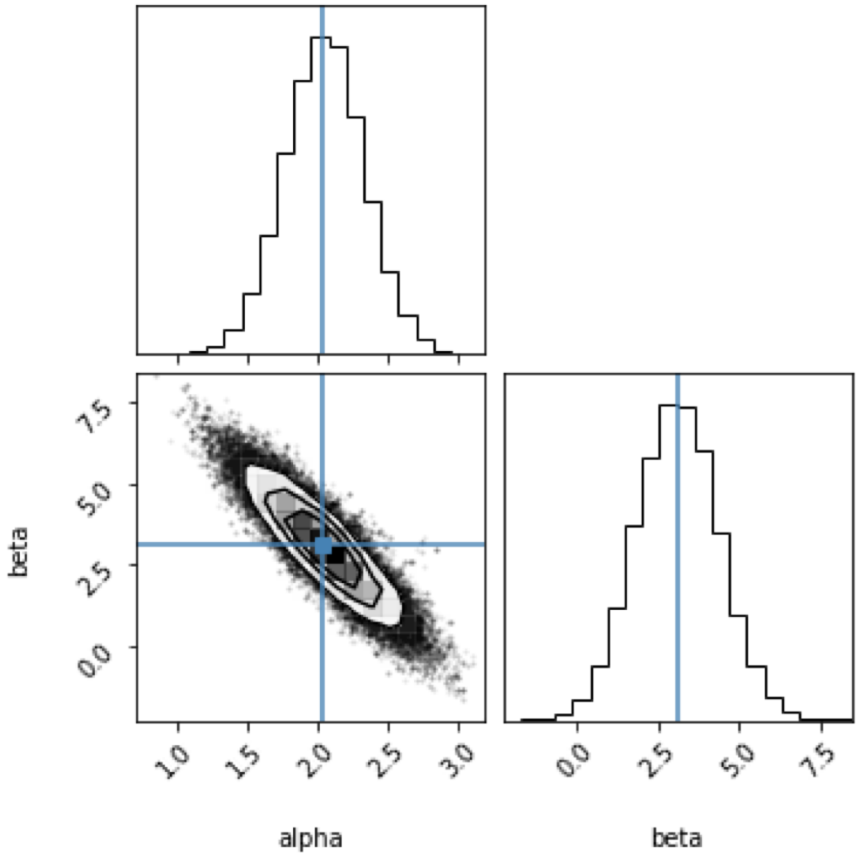
Εκτελούμε τον περίπατο στο χώρο των παραμέτρων

```
# emcee combines MCMC chains (walkers) of a number of steps.

ndim = 2  # number of parameters in the model
nwalkers = 50  # number of MCMC walkers
nburn = 20 # "burn-in" period to stabilise chains
nsteps = 2000  # number of MCMC steps to take


starting_guesses = np.random.random((nwalkers, ndim))*6 # random positioning of 50 walkers in parameter space
#print(starting_guesses)

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_posterior, args=[x_data, y_data])
sampler.run_mcmc(starting_guesses, nsteps)

#Αφαιρώ τα πρωτα nburn βήματα από καθε walker
#επίσης μπορώ τα βήματα των walkers μαζί για να χαμηλώσω μια διάσταση τον πίνακα
#emcee_trace =  sampler.chain[:, nburn:, :].reshape(-1, ndim)
emcee_trace = sampler.get_chain(discard=nburn,flat=True)
```

```
labels = ['alpha','beta']

#previous_solution=[1.53,4.86] # with errors
previous_solution=[2.04,3.10] # without errors


fig = corner.corner(emcee_trace, labels=labels
    ,truths=[previous_solution[0], previous_solution[1]]);

print( '\n Mean alpha bayesian =  ', emcee_trace[:,0].mean(),
     '\n χ^2 alpha  = ', previous_solution[0],
     '\n Mean beta bayesian =  ', emcee_trace[:,1].mean(),
     '\n χ^2 beta = ', previous_solution[1])
```

Χωρίς βάρη στην prior

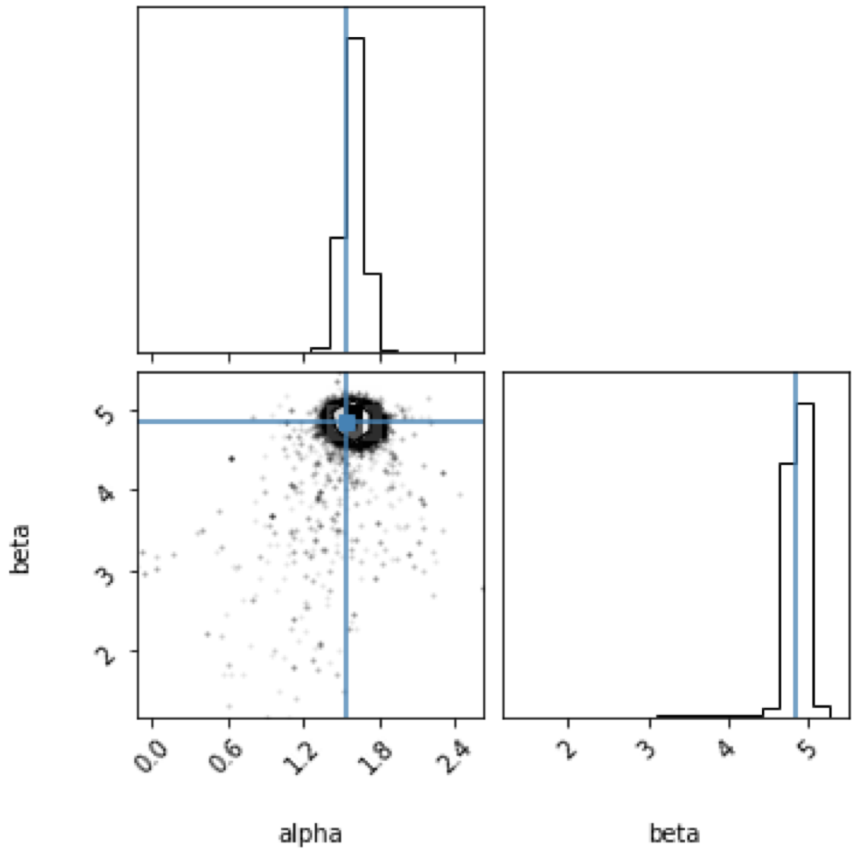
```
Mean alpha bayesian =   2.0436413093307277
χ^2 alpha  =  2.04
Mean beta bayesian =   3.0778930524593844
χ^2 beta =  3.1
```



```
labels = ['alpha','beta']

previous_solution=[1.53,4.86] # with errors
#previous_solution=[2.04,3.10] # without errors


fig = corner.corner(emcee_trace, labels=labels
    ,truths=[previous_solution[0], previous_solution[1]]);

print( '\n Mean alpha bayesian =  ', emcee_trace[:,0].mean(),
     '\n χ^2 alpha  = ', previous_solution[0],
     '\n Mean beta bayesian =  ', emcee_trace[:,1].mean(),
     '\n χ^2 beta = ', previous_solution[1])
```

Με βάρη στην prior

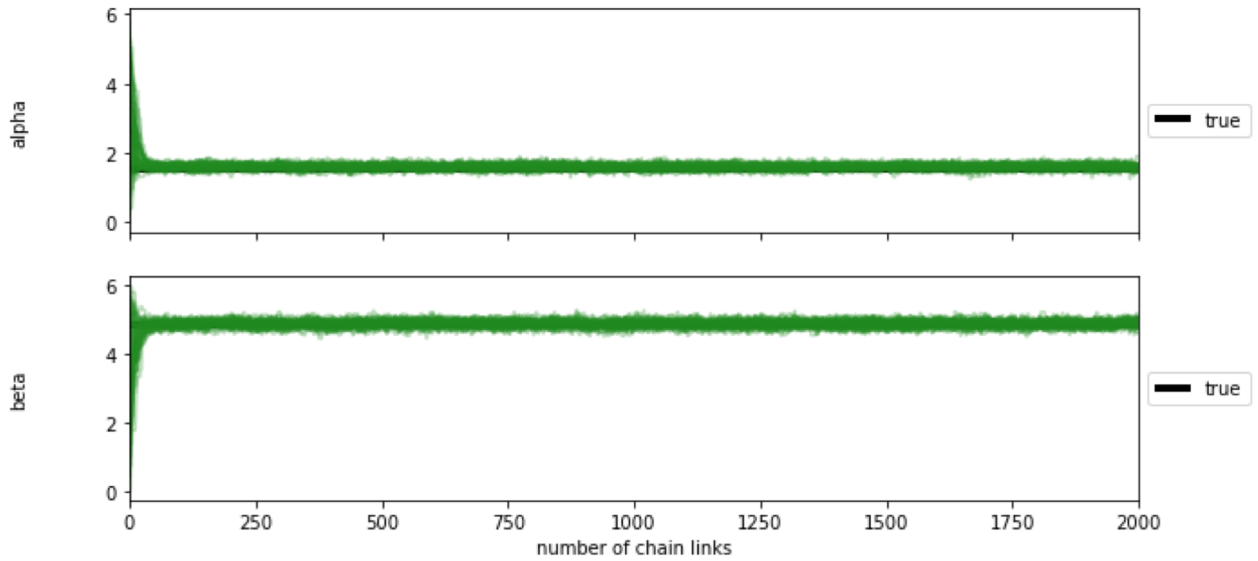
```
Mean alpha bayesian =   1.588738518764105
χ^2 alpha  =  1.53
Mean beta bayesian =   4.851960032274037
χ^2 beta =  4.86
```



Πολύ πιο στενή Κατανομή, όπως αναμενόταν

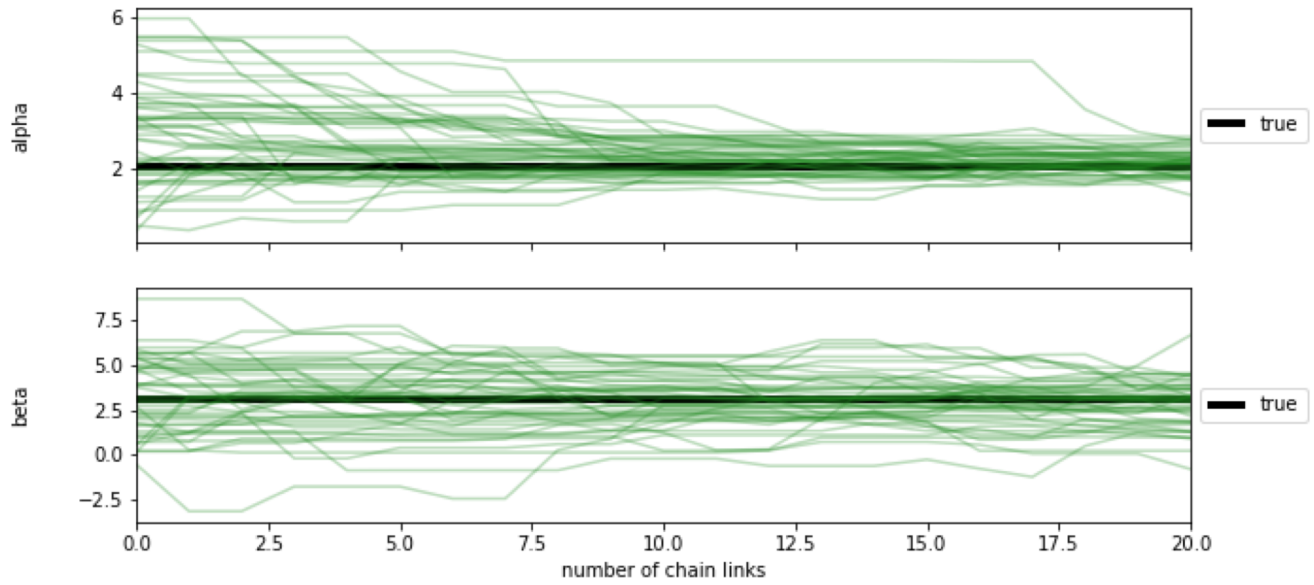
Έλεγχος πως κινούνται οι walkers στο χώρο των παραμέτρων (όλα τα βήματα)

```
fig, axes = plt.subplots(2, figsize=(10, 5), sharex=True)
samples = sampler.get_chain()
labels = [ 'alpha', 'beta']
for i in range(ndim):
    ax = axes[i]
    #ax.plot(samples[:,0, i], 'forestgreen',label='walk') #αν θελω μόνο ένα walker
    ax.plot(samples[:,:, i], 'forestgreen', alpha=0.3)    #βλεπω όλους τους walkers
    ax.set_xlim(0, len(samples))
    #ax.set_xlim(0, nburn)
    ax.set_ylabel(labels[i])
    ax.yaxis.set_label_coords(-0.1, 0.5)
    ax.hlines(y=previous_solution[i], xmin = 0, xmax = len(samples), linewidth=4, color='k',label='true')
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

axes[-1].set_xlabel('number of chain links');
```



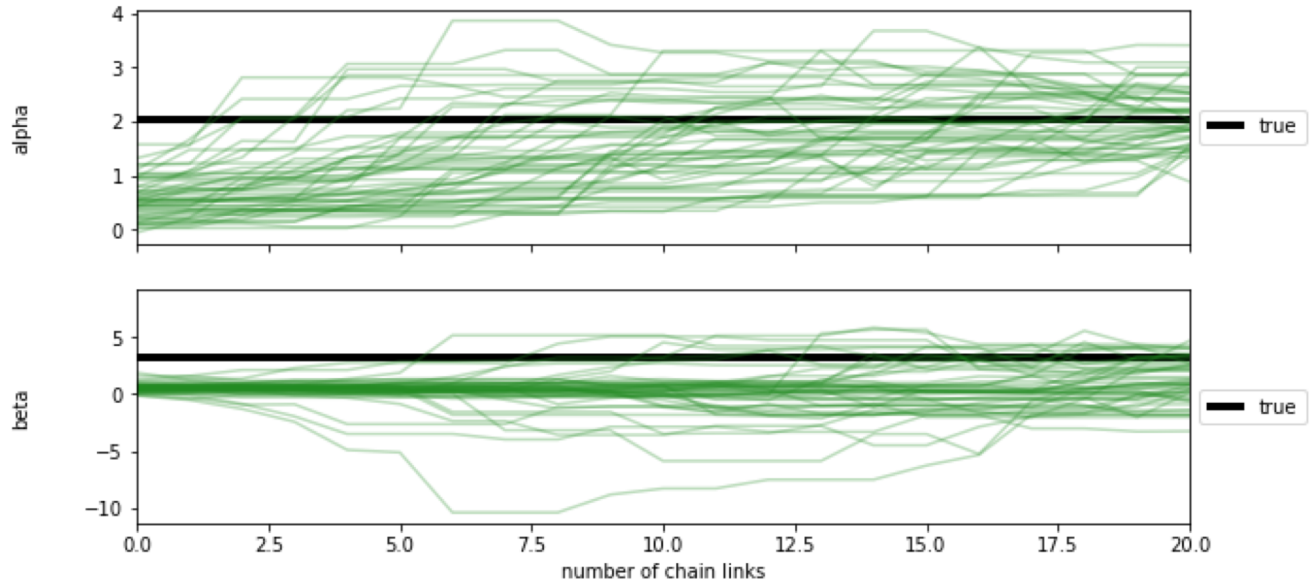
Έλεγχος πως κινούνται οι walkers στο χώρο των παραμέτρων (πρώτα βήματα, πριν τη σύγκλιση)

```
fig, axes = plt.subplots(2, figsize=(10, 5), sharex=True)
samples = sampler.get_chain()
labels = [ 'alpha', 'beta']
for i in range(ndim):
    ax = axes[i]
    #ax.plot(samples[:,0, i], 'forestgreen',label='walk') #αν θελω μόνο ένα walker
    ax.plot(samples[:,:, i], 'forestgreen', alpha=0.3)    #βλεπω όλους τους walkers
    #ax.set_xlim(0, len(samples))
    ax.set_xlim(0, nburn)
    ax.set_ylabel(labels[i])
    ax.yaxis.set_label_coords(-0.1, 0.5)
    ax.hlines(y=previous_solution[i], xmin = 0, xmax = len(samples), linewidth=4, color='k',label='true')
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

axes[-1].set_xlabel('number of chain links');
```



Έλεγχος πως κινούνται οι walkers στο χώρο των παραμέτρων (πρώτα βήματα, ξεκινώντας από το 1)

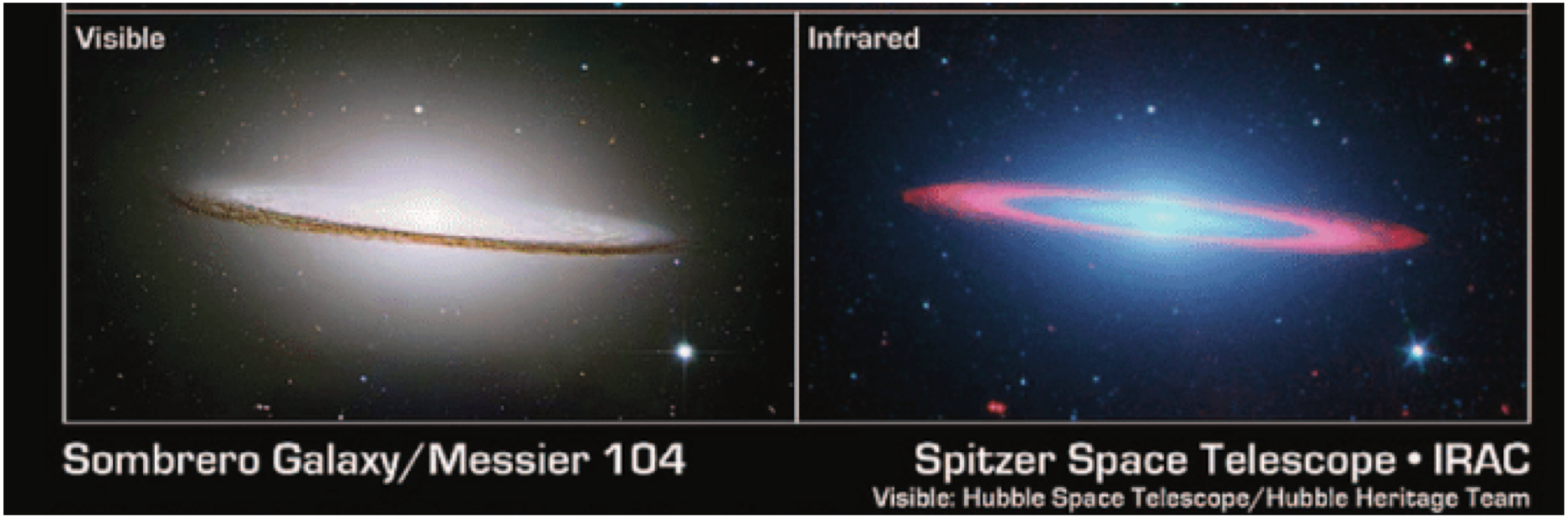
```
fig, axes = plt.subplots(2, figsize=(10, 5), sharex=True)
samples = sampler.get_chain()
labels = [ 'alpha', 'beta']
for i in range(ndim):
    ax = axes[i]
    #ax.plot(samples[:,0, i], 'forestgreen',label='walk') #αν θελω μόνο ένα walker
    ax.plot(samples[:,:, i], 'forestgreen', alpha=0.3)    #βλεπω όλους τους walkers
    #ax.set_xlim(0, len(samples))
    ax.set_xlim(0, nburn)
    ax.set_ylabel(labels[i])
    ax.yaxis.set_label_coords(-0.1, 0.5)
    ax.hlines(y=previous_solution[i], xmin = 0, xmax = len(samples), linewidth=4, color='k',label='true')
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

axes[-1].set_xlabel('number of chain links');
```



Ένα ακόμα πρόβλημα: προσαρμόζοντας **καμπύλη μελανού σώματος (Planck)** σε θερμική εκπομπή.

Θα μπορούσαμε να εξετάσουμε οποιοδήποτε σώμα αρκεί να πάρουμε τη φασματική κατανομή ενέργειάς του. π.χ., σκόνη στην ατμόσφαιρα της Γης που ζεσταθηκε από τον Ήλιο, εξωπλανήτης, ...

Στην προκειμένη θα δούμε **θερμική εκπομπή σκόνης** σε γαλαξία, και θα μετρήσουμε τη μάζα της.



The modelling function: A modified black body

Grey (or modified black) Body emission from dust grains

The monochromatic energy flux from a surface element of a sphere, in a direction normal to this element, can be obtained from Planck's law: $F_\lambda = \pi B_\lambda$ (e.g., Rybicki & Lightman). Let the sphere be a single dust grain. The black body luminosity detected from a heated dust grain is then obtained by integrating the flux from all elements of its surface: $4\pi a_g^2 \pi B_\lambda$, where a_g is its radius. As the grain may be somewhat less efficient in its emission by a pure black body, its emission can instead be described as

$B_\lambda^{modified}(T) = Q_{em}(\lambda) B_\lambda^{Planck}(T)$, where $Q_{em}(\lambda)$ is called the emissivity (Hildebrand 1983). The emissivity encapsulates the deviation in the emission efficiency from that of a pure black body and it is equal to the ratio of the emitting cross section to the geometrical cross section. It depends on λ as:

$Q_{em}(\lambda) = Q_{ref}\left(\frac{\lambda_{ref}}{\lambda}\right)^\beta = \frac{1}{507}\left(\frac{\lambda_{ref}}{\lambda}\right)^2$, (Bianchi et al. 1999,Whittet 1992).

Conservation of energy and thus of luminosity (=E/s), whether measured from the grain surface of from the Earth, implies that $4\pi D^2 F_\lambda = 4\pi a_g^2 \pi B_\lambda$, where D is the distance of the galaxy that hosts the grain. Multiplying with the number of grains N, the flux that we detect from the galaxy is:

$F_\lambda = N\,[a_g^2/D^2]\pi Q_{em}(\lambda) B_\lambda^{Planck}(T)$. So by substituting B^{Planck}, we get

$F_\lambda = N\left(\frac{a_g^2}{D^2}\right)\frac{\pi}{507}\left(\frac{100\mu m}{\lambda}\right)^2 \frac{2hc^2}{\lambda^5}\frac{1}{e^{(hc/\lambda k_B T)}-1}$

for the flux. As the flux connects to the luminosity as $4\pi D^2$ (so times the solid angle of a sphere multiplied by distance squared), the luminosity is:

$L_\lambda = N a_g^2\left(\frac{4\pi^2}{507}\right)\left(\frac{100\mu m}{\lambda}\right)^2 \frac{2hc^2}{\lambda^5}\frac{1}{e^{(hc/\lambda k_B T)}-1}$

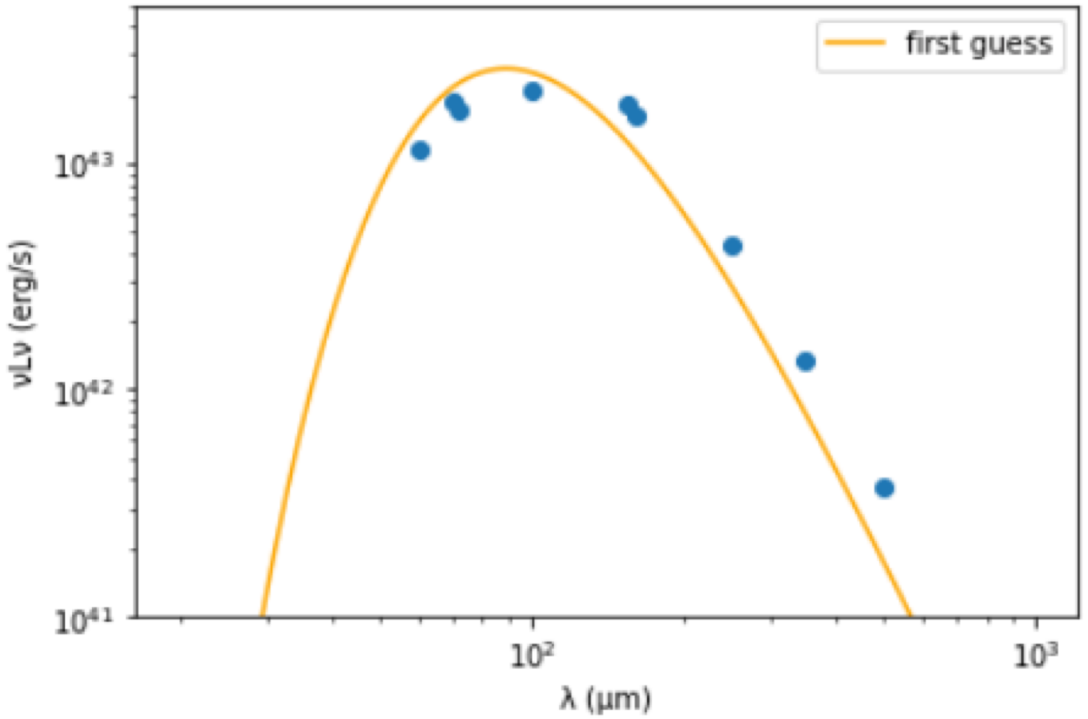
We can replace $N a_g^2$ via the total dust mass, $M = N m_{grain} = N\rho_g \frac{4\pi a_g^3}{3} = N a_g^2 \rho_g \frac{4\pi a_g}{3}$, where for a typical grain the density ρ_g =2.7 gr/cm^3 and a_g =1 μm. Finally, by multiplying both sides with λ, we get

$$\lambda L_\lambda(T, M) = \frac{M}{a_g \rho_g}\frac{6\pi}{507}\left(\frac{100\mu m}{\lambda}\right)^2 \frac{1}{\lambda^4}\frac{hc^2}{e^{(hc/\lambda k_B T)}-1} = 1.8\times 10^{35} M\left(\frac{100\mu m}{\lambda}\right)^2 \frac{1}{\lambda^4}\frac{1}{e^{(14388/\lambda T)}-1}[erg/s],$$

for the modified black body luminosity, with M in solar masses, T in K, and λ in μm. So, we end up with a function of only 2 free parameters: the dust temperature and mass

Another example: Modified black body (Planck) fitting to dust emission

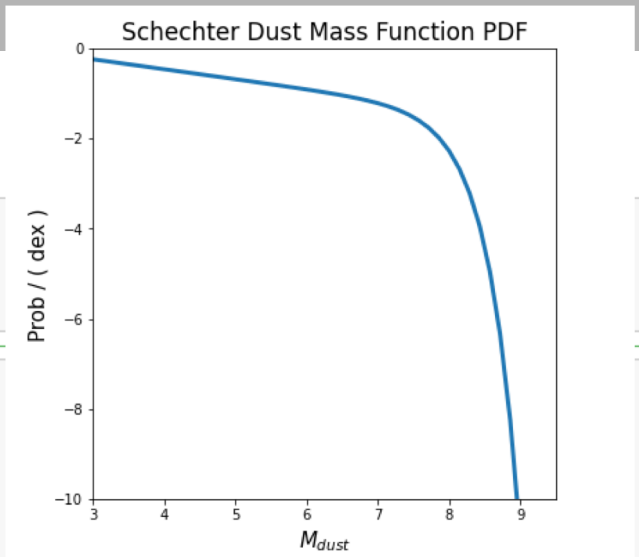
```
: ### Far infrared emission from a galaxy, to be modelled with a modified Planck law

data_array=np.array([
[59.96,1.16e+43,1.14e+41],
[70.04,1.88e+43,1.07e+42],
[71.34,1.73e+43,9.92e+41],
[99.93,2.11e+43,1.14e+42],
[156.1,1.83e+43,1.29e+42],
[160.3,1.63e+43,8.49e+41],
[249.8,4.36e+42,3.36e+41],
[349.8,1.34e+42,1.23e+41],
[499.7,3.71e+41,6.04e+40]
])

x_data,y_data,y_data_err = data_array[:,0], data_array[:,1], data_array[:,2]
```



```
#Modified black body function

def grey_body_luminosity(wavelength,T, logMd): #wavelength in μm
    h,c, kb, Lodot, Modot = 6.626e-27 ,3e10 , 1.38e-16 ,3.9e33 , 2.0e33
    ag = 0.1*10**-4 #cm
    rhog = 2.7 #gr/cm^3

    lambda_cm = wavelength/1e4 #frequencies in Hz
    lambdaB_lambda = (2*h*c**2)*1/(lambda_cm)**4*1/(np.exp(h*c/kb/T/lambda_cm)-1)

    Q = (1./507.)*(100./wavelength)**2
    lambdaL_lambda = (3*np.pi*((10**logMd)*Modot)/ag/rhog) * Q*lambdaB_lambda

    return_value=lambdaL_lambda

    return return_value
```

```
# first, a chi^2 solution
#Calculate the chi2 function FOR A SINGLE MODEL, CONSTRUCTED WITH A GIVEN SET OF PARAMETERS

def calc_chi2(params,model_function, data_x, data_y, data_y_err):
    model_y = model_function(data_x, *params) #parameters must be provided in the correct order
    chi2 = np.sum((model_y - data_y)**2 /(data_y_err)**2)
    return  chi2

#Run Minimizer
sol = minimize(calc_chi2, [30,7], args=(grey_body_luminosity, x_data,y_data,y_data_err),
               bounds  = [ [10, 100] , [4.0, 11.0] ],tol=1e-15)

chi2_values_fit =  sol.x
#T_fit, logMd_fit = np.round(sol.x,4)

print("T=",chi2_values_fit[0],"mass=",10**chi2_values_fit[1])
```

```
T= 24.544995031025476 mass= 5801213.331081659
```




```python
def mass_function(logMd):
    M = 10**logMd
    return 5.36e-2*(M/4.65e7)**(-0.22)*np.exp(-M/4.65e7)
```

```python
## probabilities:
#Calculate the probability FOR A SINGLE MODEL, CONSTRUCTED

def ln_priors_grey_body(params):
    expected_value=40.
    sigma_around_expected_value=1.
    ln_prior_on_T =  -0.5*(params[0]-expected_value)**2/(2*sigma_around_expected_value**2)
                     -0.5*np.log(2*np.pi*sigma_around_expected_value**2 )
    ln_prior_on_M = np.log( mass_function(params[1]) )
    return ln_prior_on_T + ln_prior_on_M

def ln_likelihood_grey_body(params, x, y, dy):
    y_model = grey_body_luminosity(x, params[0], params[1])
    like=-0.5 * np.sum((y-y_model)**2 / dy**2 ) -np.sum(np.log(2*np.pi*dy**2))
    if np.isnan(like):
        return -np.inf
    else:
        return like

def ln_posterior_grey_body(params, x, y,dy):
    return ln_likelihood_grey_body(params, x, y, dy)
    #return ln_priors_grey_body(params) + ln_likelihood_grey_body(params, x, y, dy)

def minus_ln_posterior_grey_body(params, x, y,dy):
    return -ln_likelihood_grey_body(params, x, y, dy)
    #return -(ln_likelihood_grey_body(params, x, y, dy) + ln_priors_grey_body(params)  )
```

```
def minus_ln_posterior_grey_body(params, x, y,dy):
    return -ln_likelihood_grey_body(params, x, y, dy)
    #return -(ln_likelihood_grey_body(params, x, y, dy) + ln_priors_grey_body(params)  )
```

```
: ### Πρώτα να το δούμε χωρίς walkers στο χώρο, μέσω minimization σαν chi^2, αλλά με προσθήκη priors
## Όντως, βάσει των priors, άλλο αποτέλεσμα!!!

sol = minimize(minus_ln_posterior_grey_body, [30,7], args=(x_data,y_data,y_data_err), bounds  = [ [10, 100] ,

prob_values_fit =  sol.x
#T_fit, logMd_fit = np.round(sol.x,4)

print("T=",prob_values_fit[0],"mass=",10**prob_values_fit[1])
```

```
T= 24.54499496926218 mass= 5801213.478752314
```

VS.

```
def minus_ln_posterior_grey_body(params, x, y,dy):
    #return -ln_likelihood_grey_body(params, x, y, dy)
    return -(ln_likelihood_grey_body(params, x, y, dy) + ln_priors_grey_body(params)  )
```

```
### Πρώτα να το δούμε χωρίς walkers στο χώρο, μέσω minimization σαν chi^2, αλλά με προσθήκη priors
## Όντως, βάσει των priors, άλλο αποτέλεσμα!!!

sol = minimize(minus_ln_posterior_grey_body, [30,7], args=(x_data,y_data,y_data_err), bounds  = [ [10, 100] ,

prob_values_fit =  sol.x
#T_fit, logMd_fit = np.round(sol.x,4)

print("T=",prob_values_fit[0],"mass=",10**prob_values_fit[1])
```

```
T= 39.192899459423785 mass= 130944.25488361447
```

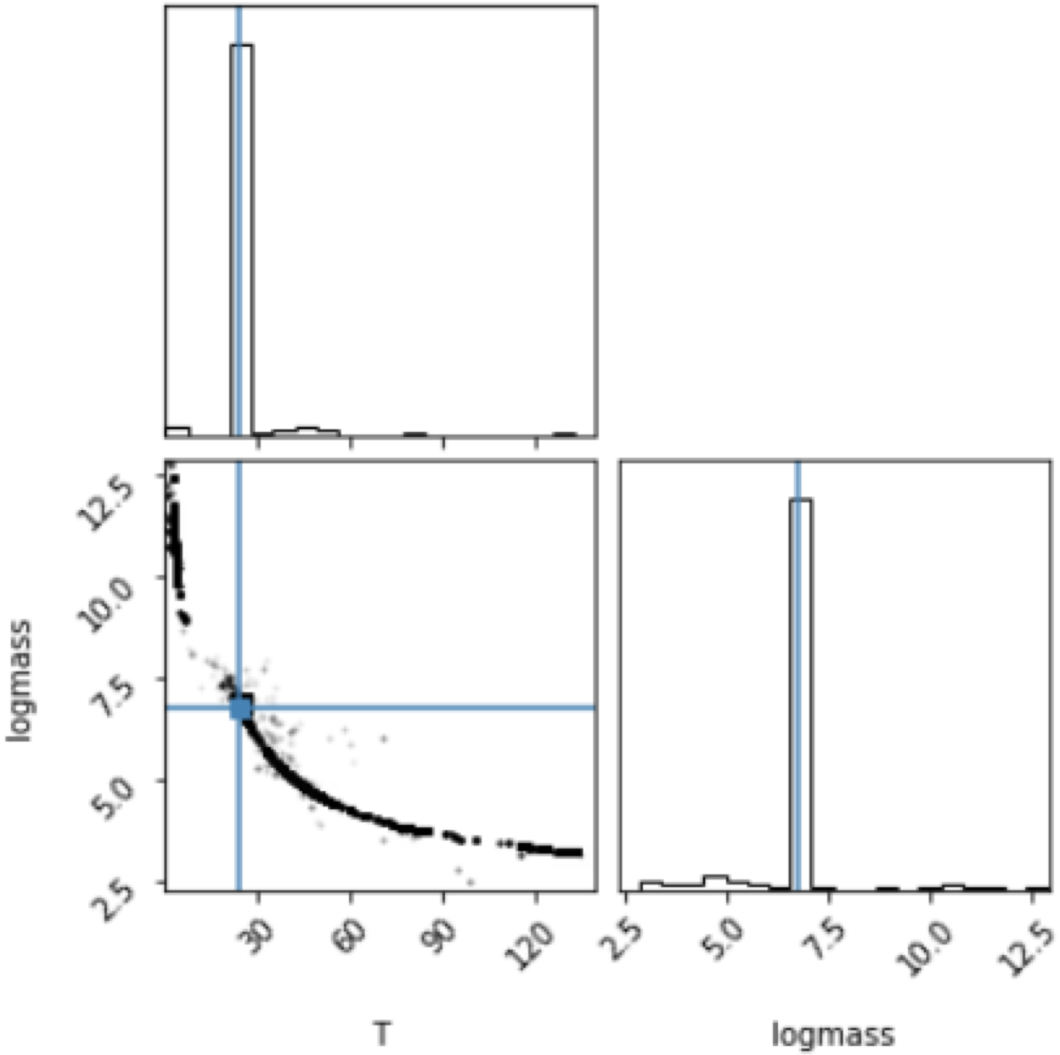
Now with a Markov Chain Monte Carlo + Bayesian

```
# emcee combines MCMC chains (walkers) of a number of steps.

ndim = 2  # number of parameters in the model
nwalkers = 100  # number of MCMC walkers
nburn = 20 # "burn-in" period to stabilise chains
nsteps = 2000  # number of MCMC steps to take


#starting_guesses = np.random.random((nwalkers, ndim)) # random positioning of 50 walkers in parameter space
#starting_guesses[:,0]*=10
#starting_guesses[:,1]=(starting_guesses[:,1]*5+6)
starting_guesses=np.random.normal([30,8],[7,2],size=(nwalkers, ndim))


sampler = emcee.EnsembleSampler(nwalkers, ndim, ln_posterior_grey_body, args=[x_data, y_data,y_data_err])
sampler.run_mcmc(starting_guesses, nsteps)
emcee_trace = sampler.get_chain(discard=nburn,flat=True)

previous_solution=[24,6.8]


fig = corner.corner(emcee_trace, labels=['T','logmass']
    ,truths=[previous_solution[0], previous_solution[1]]);

print( '\n Mean T bayesian =  ', emcee_trace[:,0].mean(),
     '\n Mean mass bayesian =  ', 10.**emcee_trace[:,1].mean())
```

```
WARNING:root:Too few points to create valid contours
```

```
 Mean T bayesian =   28.106281567692122
 Mean mass bayesian =   4314394.427696127
```

```
Mean T bayesian =   28.106281567692122
Mean mass bayesian =   4314394.427696127
```



```
fig, axes = plt.subplots(2, figsize=(10, 5), sharex=True)
samples = sampler.get_chain()
labels = [ 'T', 'logMd']
for i in range(ndim):
    ax = axes[i]
    #ax.plot(samples[:,0, i], 'forestgreen',label='walk') #αν θελω μόνο ένα walker
    ax.plot(samples[:,:, i], 'forestgreen', alpha=0.3)    #βλεπω όλους τους walkers
    ax.set_xlim(0, len(samples))
    #ax.set_xlim(0, nburn)
    ax.set_ylabel(labels[i])
    ax.yaxis.set_label_coords(-0.1, 0.5)
    ax.hlines(y=previous_solution[i], xmin = 0, xmax = len(samples), linewidth=4, color='k',label='true')
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

axes[-1].set_xlabel('number of chain links');
```