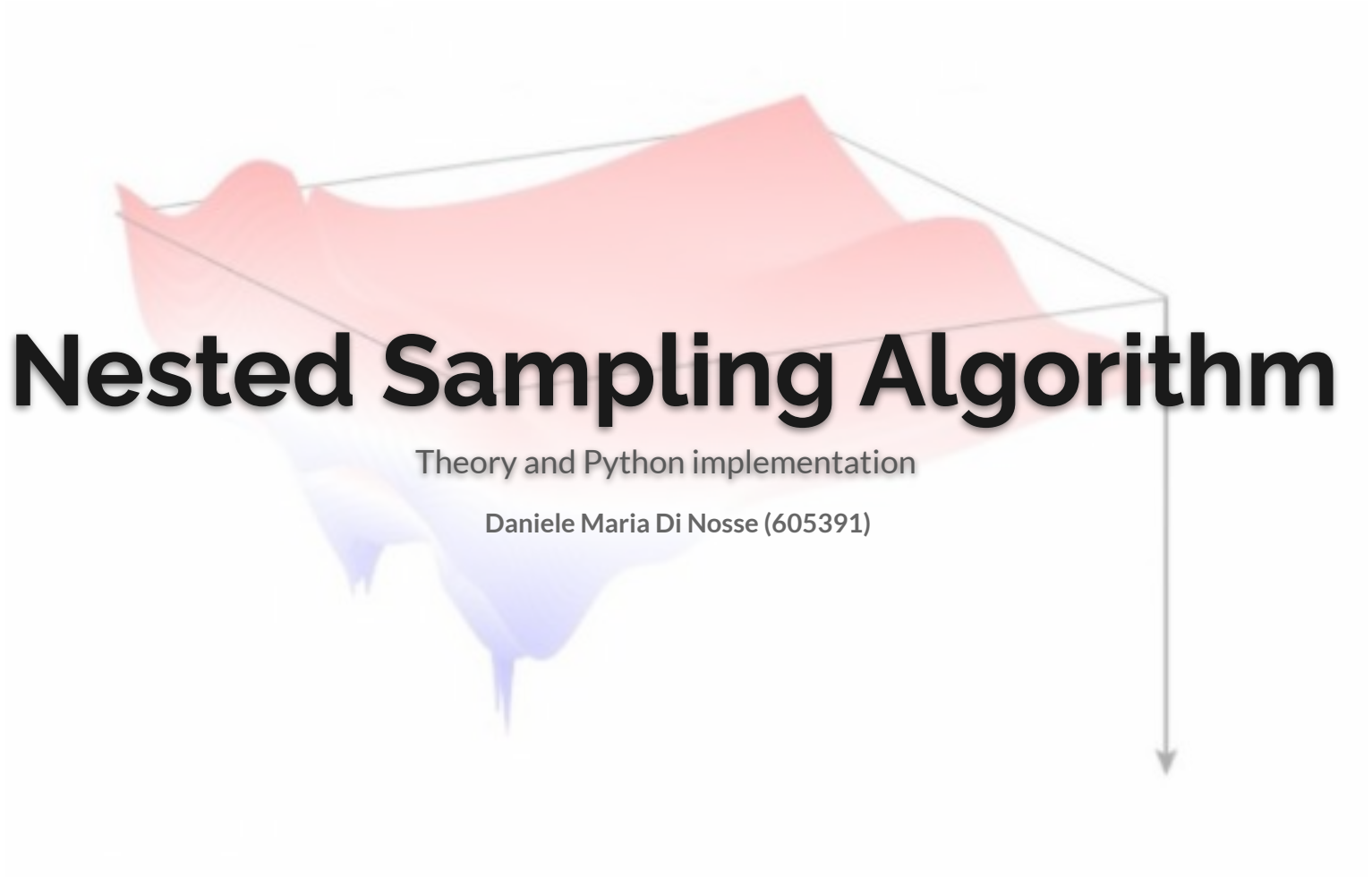


Nested Sampling Algorithm

Theory and Python implementation

Daniele Maria Di Nosse (605391)

Nested Sampling Algorithm

❖ Summary

- Introduction
- Outline of the algorithm
- Details
- Python implementation

❖ References

- John Skilling, Nested sampling for general Bayesian computation (2006) https://doi.org/10.1214/06-BA127
- John Skilling, Nested Sampling (2004) https://ui.adsabs.harvard.edu/abs/2004AIPC..735..395S
- Front image from http://helper.ipam.ucla.edu/publications/elws2/elws2_14353.pdf

❖ Code

https://github.com/DanieleMDiNosse/Nested_Sampling

Introduction

- The core of the bayesian probability is of course the Bayes' theorem.
- It has the ability to handle problems that the frequentist approach to probability can not.

$$P(H|DI) = \frac{P(H|I)P(D|HI)}{P(D|I)} = \frac{P(H|I)P(D|HI)}{\sum_i P(H_i|I)P(D|H_iI)}$$

$$Posterior = \frac{Prior \times Likelihood}{Evidence}$$

- It can be used to compute the so-called Odds Ratio between two set of parameters (in general between two models).

$$\frac{P(H_1|DI)}{P(H_2|DI)} = \frac{P(H_1|I)}{P(H_2|I)}\frac{P(D|H_1I)}{P(D|H_2I)} = \mathrm{Pr}\,ior\ Odds \times Bayes\ Factor$$

Introduction

- Without any particular information in favor of one or the other model, Prior Odds are set to 1.
- If $H_1 \equiv H_1(\vec{\theta_1}), H_2 \equiv H_2(\vec{\theta_2})$ Bayes factor reduces to the ratio of the evidence for $\vec{\theta_1}$ and $\vec{\theta_2}$

$$B_{12} = \frac{\int_{\Theta_1} d\vec{\theta_1} P(\vec{\theta_1}|H_1 I) P(D|\vec{\theta_1} H_1 I)}{\int_{\Theta_2} d\vec{\theta_2} P(\vec{\theta_2}|H_2 I) P(D|\vec{\theta_2} H_2 I)}$$

- For the evidence to be used in the Bayes factor, we need to compute

$$Z = \int \int \ldots \int L(\vec{\theta}) \pi(\vec{\theta}) d\vec{\theta}$$

- MCMC algorithms are used to compute the posterior.
- Using such methods to evaluate the evidence requires a lot of extra work (generalization of thermodynamic integration)
- Nested sampling puts the evidence as the prime target and the posterior is just a by-product of it.

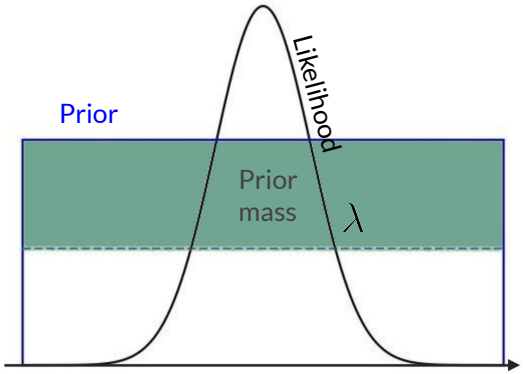
Nested sampling - Outline of the algorithm

- The main idea is to image to change variable

$$\pi(\vec{\theta})d\vec{\theta} = d\xi$$

- Where ξ is called the **prior mass** and represents the cumulative prior over a specific value of the likelihood

$$\xi(\lambda) = \int\int \dots \int_{L(\vec{\theta})>\lambda} \pi(\vec{\theta})d\vec{\theta}$$



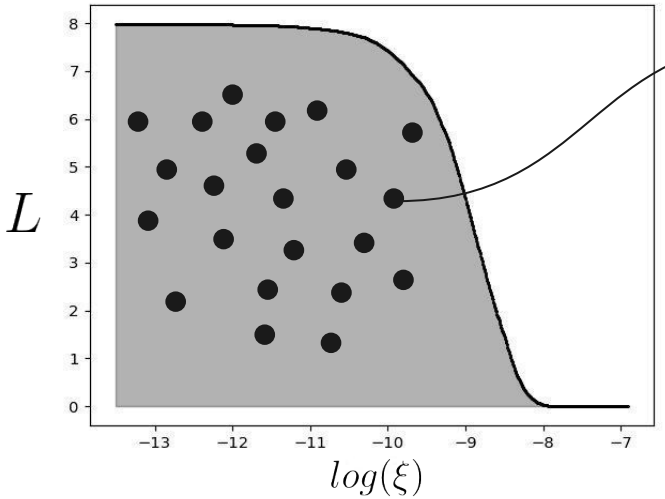
- We constructed so a 1-dim integral over [0,1] from a N-dim one:

$$\int_0^1 L(\xi)d\xi$$

- If each of the N components of $\vec{\theta}$ is stored with an accuracy of 1 part in R, ξ should be stored as 1 part in R^N: information remains $Nlog_2R$

Nested sampling - Outline of the algorithm

- Image now to sort likelihoods in ascending order $L_{min} < L_1 < L_2 < ... < L_{max}$
- At every value of the likelihood we can associate a value of the prior mass, starting from $\xi = 0$ $(L = L_{\min})$ and going up until we reach the maximum of L, where the prior mass is $\xi = 1$



As by-product we have sample from the posterior

$$P(\xi) = \frac{L(\xi)}{Z}$$

- The idea is to approximate the integral as the usual Riemann sum $Z \approx \sum_i L_i \Delta\xi_i$

Nested sampling - Details of the algorithm

- How do we compute $\Delta\xi_i$? We do not know the transformation from $\vec{\theta} \to \xi$!
- Nested Sampling tries to find it statistically.

- NS uses N objects randomly sampled from the prior that each time are updated with an evolving constraint regarding the growing of the likelihood .
- Each time we consider the object with smallest likelihood we have, L^*, remove it from the sample and replace it by a new one satisfying $L(\vec{\theta}_{new}) > L^*$
- In terms of the prior mass, this object are such that $\xi_{i+1} < \xi_i$, so we are shrinking the prior mass domain.

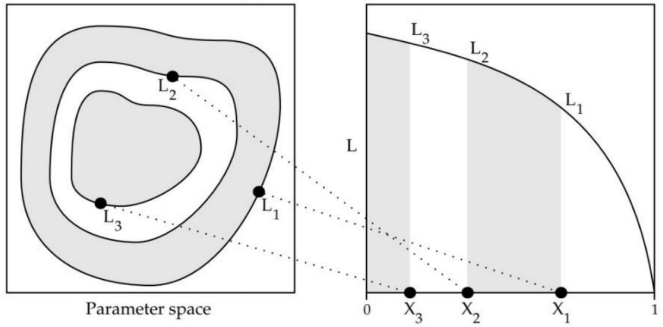
- Reason as follows:
  - At a certain iteration we have an upper bound ξ^* for the prior mass.
  - Choosing the worst L implies choosing a certain ξ
  - ξ is the largest number out of N uniformly distributed in $[0, \xi^*]$.
  - The shrinkage ratio $t = \xi/\xi^* \in [0,1]$ is distributed as $p(t) = nt^{n-1}$ with mean and standard deviation such that

    $$logt = \frac{-1 \pm 1}{n}$$

  - After k iteration

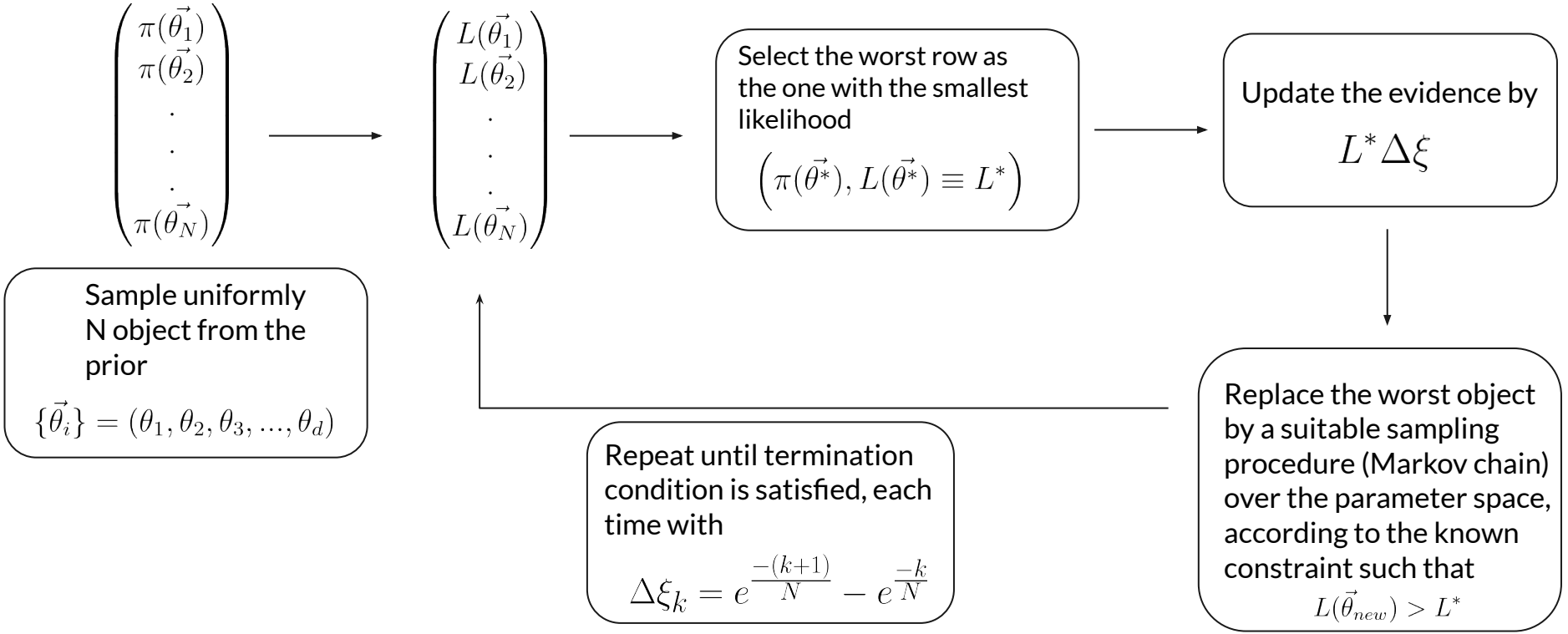
    $$\xi_k = \prod_{j=1}^{k} t_j \quad \text{with} \quad logt \approx \frac{-1}{n}$$

  - Now we are able to compute the sum.



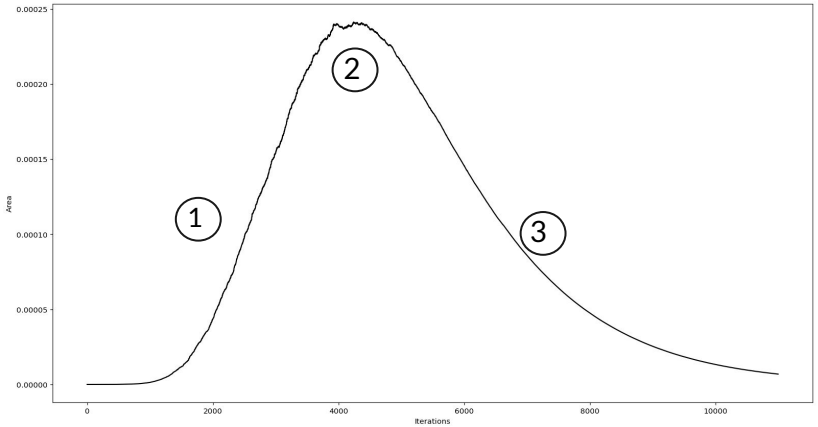
Nested sampling - Details of the algorithm

- Overview of the algorithm steps.



Nested sampling - Termination

- How do we decide when terminate the algorithm?
- The usual behaviour of the areas element is the following



1. Increase of L is dominant on the decrease of the widths
2. Balance
3. Decrease of the width take over the increase of L

- We want to stop the algorithm when $L(\vec{\theta}^*)\Delta\xi$ becomes so small that it does not contribute significantly to the evidence.
- Qualitatively this happens when the number of iterations exceeds significantly nH.
- More quantitatively this condition will occur when

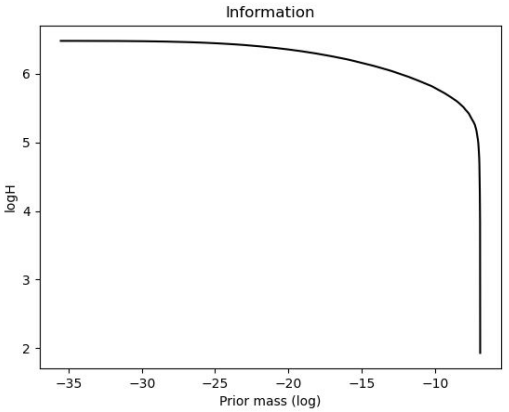
$$max\{L(\vec{\theta})\}_i \, \xi_i < f Z_i$$

Nested sampling - Error

- The peak is to be found in the region $\xi \approx e^{-H}$ where $H = \int P(\xi) log\left[P(\xi)\right] d\xi \approx \sum_k \frac{\Delta\xi_k L_k}{Z} log\left[\frac{L_k}{Z}\right]$ is the information. It is a measure of the prior-to-posterior shrinkage in logarithmic form.

- Remember that

$$logt = \frac{-1 \pm 1}{n} \quad \text{and} \quad \xi_k = \prod_{j=1}^{k} t_j \longrightarrow \log \xi_k = \frac{-k \pm \sqrt{k}}{n}$$



- The nH iterations required induces an error over the evidence such that

$$\log Z \approx \sum_k \log\left(L_k \xi_k\right) \pm \sqrt{\frac{H}{n}}$$

- More accurate estimate of the error can be done using the distribution of t $p(t) = nt^{n-1}$

$$t \implies \xi \implies \Delta\xi \implies Z \equiv Z(t)$$

- We can sample several t from the distribution, obtain the values of the evidence and compute mean and standard deviation from them

Nested sampling - Python implementation

- Compute numerically

$$\int_{-A}^{A} \left(\frac{1}{\sqrt{2\pi\boldsymbol{\Sigma}}} \right)^{d} e^{-\frac{1}{2}(\vec{x}-\vec{\mu})^{t}\Sigma^{-1}(\vec{x}-\vec{\mu})} d\vec{x}$$

where

$$\Sigma = \mathbb{I} \quad ; \quad \vec{\mu} = 0$$
$$d \in [1, 2, ..., 50] \quad ; \quad A = 5$$

```python
def log_likelihood(x, dim, init):
    ''' Return the logarithm of a N-dimensional gaussian likelihood.
    It is set in such a way that the integral of the product with the
    prior over the parameter space is 1.

    Parameters
    ----------
    x : numpy.array
        If init is set to True, x should be a MxN matrix whose M rows (number of points)
        are random N-dimensional vectors. In this case it is used to initialize the
        likelihood of the live points. If init is set to False, x should be just a
        random N dimensional vector.
    dim : int
        Dimension of the parameter space.
    init: bool
        You can choose to use the funcion to initialize the likelihood of the live
        points (True) or to generate just a new likelihood value (False)

    Returns
    --------
    Likelihood : list or float
        Likelihood values or singole likelihood value
    '''

    if init:
        likelihood = [- 0.5*dim*np.log(2*np.pi) - 0.5*v.T.dot(v) for v in x]
    else:
        likelihood = - 0.5*dim*np.log(2*np.pi) - 0.5*x.T.dot(x)

    return likelihood
```

Live Points

$$\begin{pmatrix} \theta_1^1, \theta_2^1, ..., \theta_d^1, L(\vec{\theta_1}) \\ \theta_1^2, \theta_2^2, ..., \theta_d^2, L(\vec{\theta_2}) \\ . \\ . \\ . \\ \theta_1^N, \theta_2^N, ..., \theta_d^N, L(\vec{\theta_N}) \end{pmatrix}$$

Nested sampling - Python implementation

- The idea is to tune the average jump of the points in two ways:

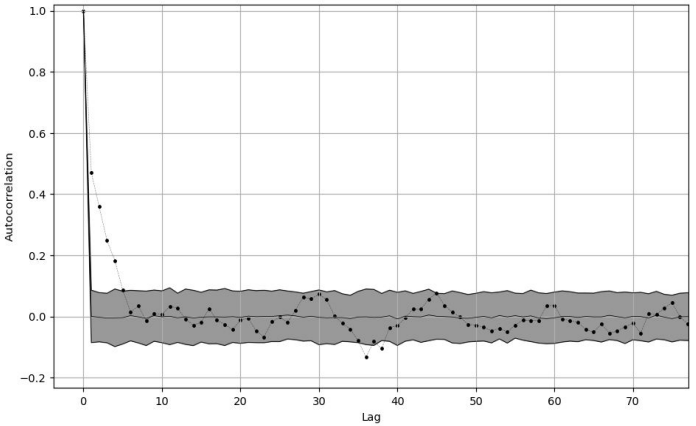
  ➔ as a function of the dimension of the parameter space for the normal proposal

  ➔ keeping the acceptance ratio around 50% in the uniform case. If there are more accepted points than rejected we can enlarge the jump, otherwise we have to shrink it

```python
def proposal(x, dim, boundary, std, distribution):
    ''' Sample a new object from the prior subject to the constrain L(x_new) > Lworst_old
    '''

    logLmin = x[dim]
    boundary_point = x[:dim]

    start = time.time()
    accepted = 0
    rejected = 0
    n = 0
    k_n = 1/(2*np.log(dim+1))

    while True:
        new_line = np.zeros(dim+1, dtype=np.float64)
        for i in range(len(new_line[:dim])):

            if distribution == 'uniform':
                new_line[:dim][i] = boundary_point[i] + np.random.uniform(-std, std)
                while np.abs(new_line[:dim][i]) > boundary:
                    new_line[:dim][i] = boundary_point[i] + np.random.uniform(-std, std)

            if distribution == 'normal':
                new_line[:dim][i] = np.random.normal(boundary_point[i], k_n*std)
                while np.abs(new_line[:dim][i]) > boundary:
                    new_line[:dim][i] = np.random.normal(boundary_point[i], k_n*std)

        new_line[dim] = log_likelihood(new_line[:dim], dim, init=False)[0]

        if new_line[dim] < logLmin:
            rejected += 1
        if new_line[dim] > logLmin:
            n += 1
            accepted += 1
            boundary_point[:dim] = new_line[:dim]
            if n > 10:
                end = time.time()
                t = end - start
                break
        if distribution == 'uniform':
            if accepted != 0 and rejected != 0:
                if accepted > rejected: std *= np.exp(1.0/accepted)
                if accepted < rejected: std /= np.exp(1.0/rejected)

    return new_line, t, accepted, rejected
```

Nested sampling - Python implementation

- New object must be independent from the starting point of the Markov chain




```python
def proposal(x, dim, logLmin, boundary_point, boundary, std, distribution):
    ''' Sample a new object from the prior subject to the constrain L(x_new) > Lworst_old
    '''
    start = time.time()
    accepted = 0
    rejected = 0
    n = 0
    k_n = 1/(2*np.log(dim+1))

    while True:
        new_line = np.zeros(dim+1, dtype=np.float64)
        for i in range(len(new_line[:dim])):

            if distribution == 'uniform':
                new_line[:dim][i] = boundary_point[i] + np.random.uniform(-std, std)
                while np.abs(new_line[:dim][i]) > boundary:
                    new_line[:dim][i] = boundary_point[i] + np.random.uniform(-std, std)

            if distribution == 'normal':
                new_line[:dim][i] = np.random.normal(boundary_point[i], k_n*std)
                while np.abs(new_line[:dim][i]) > boundary:
                    new_line[:dim][i] = np.random.normal(boundary_point[i], k_n*std)

        new_line[dim] = log_likelihood(new_line[:dim], dim, init=False)[0]

        if new_line[dim] < logLmin:
            rejected += 1
        if new_line[dim] > logLmin:
            n += 1
            accepted += 1
            boundary_point[:dim] = new_line[:dim]
            if n > 10:
                end = time.time()
                t = end - start
                break
        if distribution == 'uniform':
            if accepted != 0 and rejected != 0:
                if accepted > rejected: std *= np.exp(1.0/accepted)
                if accepted < rejected: std /= np.exp(1.0/rejected)

    return new_line, t, accepted, rejected
```

Nested sampling - Python implementation

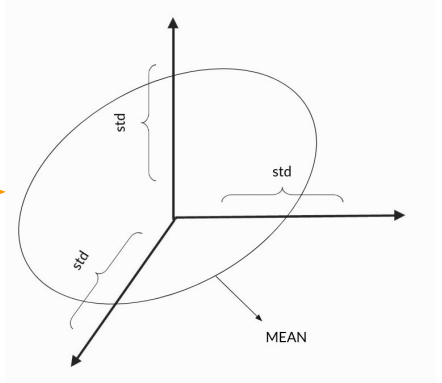
```python
def autocorrelation(x, max_lag, bootstrap=False):
    '''Simple implementation for the autocorrelation function.
        Autocorrelation function'''

    x = np.array(x)
    x_mean = np.mean(x)
    auto_corr = []
    for d in range(max_lag):
        ac = 0
        for i in range(len(x)-d):
            ac += (x[i] - x_mean) * (x[i+d] - x_mean)
        ac = ac / np.sqrt(np.sum((x - x_mean)**2) * np.sum((x - x_mean)**2))
        auto_corr.append(ac)
    auto_corr = np.array(auto_corr)
    plt.figure()
    plt.plot(auto_corr, 'k--', linewidth=0.4, alpha=0.5)
    plt.scatter(np.arange(len(auto_corr)), auto_corr, s=5, color='black')
    plt.grid()
    plt.xlabel('Lag')
    plt.ylabel('Autocorrelation')


    if bootstrap:
        auto_corr_bootstrap = []
        for i in range(200):
            xs = x
            np.random.shuffle(xs)
            xs_mean = np.mean(xs)
            auto_corr_bootstrap_i = []
            for d in range(max_lag):
                ac = 0
                for i in range(len(x)-d):
                    ac += (x[i] - x_mean) * (x[i+d] - x_mean)
                ac = ac / np.sqrt(np.sum((x - x_mean)**2) * np.sum((x - x_mean)**2))
                auto_corr_bootstrap_i.append(ac)
            auto_corr_bootstrap.append(auto_corr_bootstrap_i)
        meanac = np.mean(np.array(auto_corr_bootstrap), axis=0)
        stdac = np.std(np.array(auto_corr_bootstrap), axis=0)
        plt.plot(meanac - 2*stdac, 'black', lw=0.5)
        plt.plot(meanac, 'black', lw=0.5)
        plt.plot(meanac + 2*stdac, 'black', lw=0.5)
        plt.fill_between(np.arange(0, max_lag), meanac + 2*stdac, meanac - 2*stdac, color='black',
        alpha=0.4)
    plt.show()

    return auto_corr
```

The random shuffling of time series ensures all the temporal relations to be lost

Nested sampling - Python implementation

```python
def nested_samplig(live_points, dim, boundary, proposal_distribution, verbose=False):
    '''Nested Sampling by Skilling (2004)
        '''

    N = live_points.shape[0]
    f = np.log(0.01)
    area = []; Zlog = [[],[]]; logL_worst = []; T = []; prior_mass = []; logH_list = []

    logZ = -np.inf
    logH = -np.inf
    parameters = np.random.uniform(-boundary, boundary, size=(N, dim))
    live_points[:, :dim] = parameters
    live_points[:, dim] = log_likelihood(parameters, dim, init=True)
    logwidth = np.log(1.0 - np.exp(-1.0/N))

    steps = 0
    rejected = 0
    accepted = 0
    while True:
        steps += 1
        prior_mass.append(logwidth)

        Lw_idx = np.argmin(live_points[:, dim])
        logLw = live_points[Lw_idx, dim]
        logL_worst.append(logLw)

        logZnew = np.logaddexp(logZ, logwidth+logLw)
        logZ = logZnew

        logH = np.logaddexp(logH, logwidth + logLw - logZ + np.log(logLw - logZ))
        logH_list.append(logH)
        error = np.sqrt(np.exp(logH)/N)
        Zlog[0].append(logZ + error)
        Zlog[1].append(logZ - error)

        survivors = np.delete(live_points, Lw_idx, axis=0)
        std = np.mean(np.std(survivors[:dim], axis = 0))
        boundary_point = live_points[Lw_idx,:dim]
```

Initialization



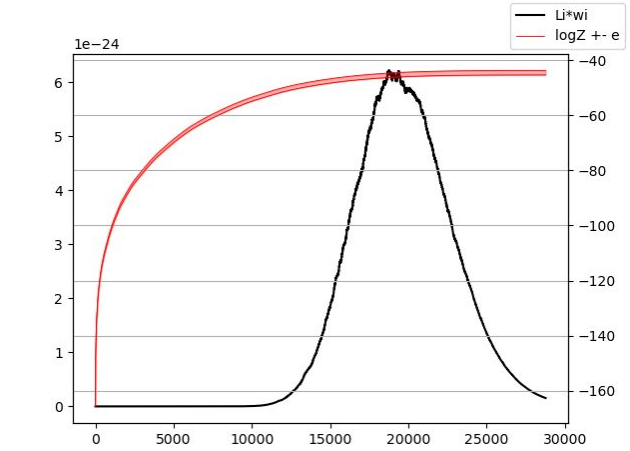

```python
        area.append(logwidth+logLw)

        if verbose:
            print("i:{0} d={1} log(Lw)={2:.2f} term={3:.2f} log(Z)={4:.2f} std={5:.2f}
            e={6:.2f} H={7:.2f} prop={8}".format(steps, dim, logLw,
            (max(live_points[:,dim]) - steps/N - f - logZ), logZ, std, error,
            np.exp(logH), proposal_distribution))

        new_sample, t, acc, rej = proposal(live_points[Lw_idx], dim, boundary, std,
        proposal_distribution)
        accepted += acc
        rejected += rej
        T.append(t)

        live_points[Lw_idx] = new_sample
        logwidth -= 1.0/N

        if max(live_points[:,dim]) - (steps+steps**0.5)/N < f + logZ:
            break

    final_term = np.log(np.exp(live_points[:,dim]).sum()*np.exp(-steps/N)/N)
    logZ = np.logaddexp(logZ, final_term)
    area = np.exp(area)

    return area, Zlog, logL_worst, prior_mass, logZ, T, steps, accepted, rejected,
    logH_list, error
```

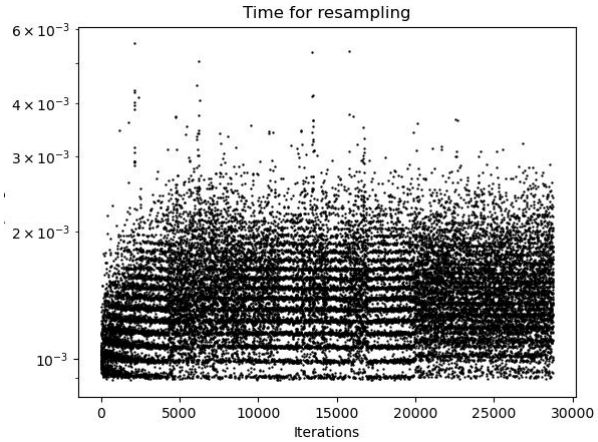
Sample new object

Terminal condition

Adding final term

Nested sampling - Python implementation (results)






```
============ SUMMARY ============

Dimension of the integral = 20

Number of steps required = 28628

Evidence = -44.46 +- 0.81

Theoretical value = -46.05170185988092

Information = 649.47

Maximum of the likelihood = -18.38

Proposal chosen: normal

Last area value = 0.00

Last worst Likelihood = -21.50

Accepted and rejected points: 314908, 169792

Mass prior sum = 1.00

Total time: 56.25 s

================================
```

Nested sampling - Python implementation (results)

- Both in terms of accuracy and computational time, the normal proposal with the implementation used to handle its standard deviation performed better than the uniform one.