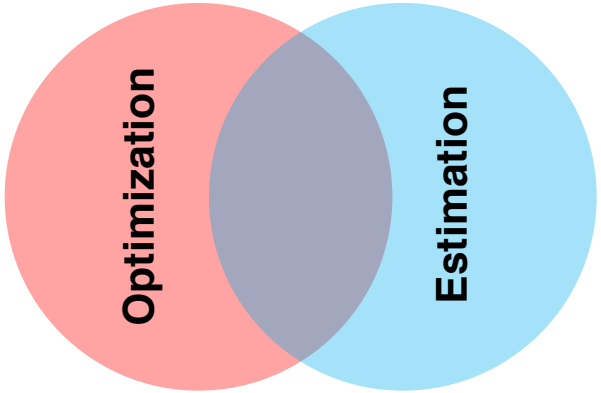
Lecture 14: Parameter Estimation

Parameter Estimation: Using data to estimate the parameters of a model (or distribution) and their uncertainty.

First of two Venn diagrams today:



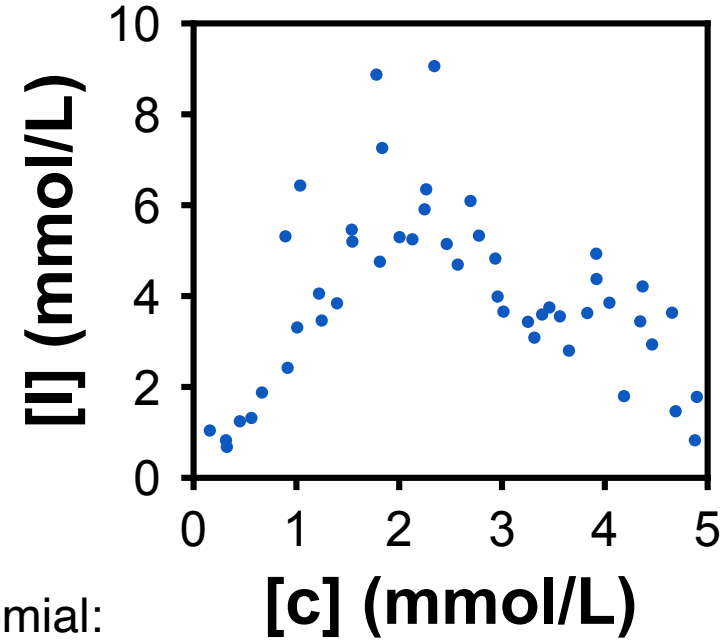
In estimation, we use statistics to estimate model parameters, deliver uncertainty estimates, and update our parameters as new data arrives. In some cases estimation can be formulated as an optimization problem (hence the overlap).

Goals for Today

- Analytic Estimation
- Bootstrap Estimation
- Bayesian Inference (Indirectly, Maximum Likelihood)

Working Example

We have experimental data for the impurity concentration, **[I]**, in our output stream as a function of co-solvent concentration, **[c]**, for a complicated reaction



We want to fit this data to a Legendre polynomial:

$$y_{p,i} = \sum_{i=0}^{5} \beta_i P_i(x_i) = \beta_0 + \beta_1 x_i + \frac{\beta_2}{2}\left(3x_i^2 - 1\right) + \ldots$$

How would we solve this problem using optimization?

$$\mathbf{L}\left[f(\mathbf{x}), \mathbf{y}\right] = \frac{1}{N}\sum_{i}^{N}\left[f(x_i) - y_i\right]^2 +$$

Gradient descent, stochastic gradient descent, adagrad, rmsprop, PSO, GA, …

How would we calculate the uncertainty in parameters?

Working Example – Analytic Estimation

Since the model is linear in our parameters, we can solve for the least-squares solution analytically using linear algebra:

$$\mathbf{A}\boldsymbol{\beta} = \mathbf{y} \rightarrow \begin{bmatrix} P_0(x_1) & P_1(x_1) & \dots & P_5(x_1) \\ P_0(x_2) & P_1(x_2) & \dots & P_5(x_2) \\ \dots & \dots & \dots & \dots \\ P_0(x_N) & P_1(x_N) & \dots & P_5(x_N) \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \dots \\ \beta_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_N \end{bmatrix}$$

Multiplication by $\mathbf{A}^{\mathrm{T}}$ solves rank deficiency and yields solution:

$$\mathbf{A}\boldsymbol{\beta} = \mathbf{y} \rightarrow \mathbf{A}^{\mathrm{T}}\mathbf{A}\boldsymbol{\beta} = \mathbf{A}^{\mathrm{T}}\boldsymbol{\beta}$$

Estimating the variance σ^2 of our data as: $$\sigma^2 = \frac{1}{N}\sum_{i}^{N}\left(y_i - \hat{y}(x_i)\right)^2$$

Assuming normally distributed errors, we can calculate the standard errors in the parameters and y predictions as:

$$\mathrm{se}\left(\boldsymbol{\beta}\right) = \mathrm{diag}\left[\left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\right]^{\frac{1}{2}}\sigma \qquad \mathrm{se}\left(\mathbf{y}_i\right)^{*} = \left[\underbrace{\mathbf{P}(x_i)^{\mathrm{T}}}\left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\underbrace{\mathbf{P}(x_i)}\right]^{\frac{1}{2}}\sigma$$

$\mathbf{P}(x_i)$ vector of Legendre polynomials evaluated at x_i

*Likewise, the confidence intervals on the predictions can be obtained by multiplying by the inverse value from the cumulative Student's t-distribution

Working Example - Analytic Estimation

Python Implementation:

```
# Calculate best fit parameters
x_p = (x-2.5)/2.5
A = np.vstack([(x_p*0+1),x_p,0.5*(3.0*x_p**(2.0)-1.0),\
               0.5*(5.0*x_p**(3.0)-3.0*x_p),\
               1.0/8.0*(35.0*x_p**(4.0)-30.0*x_p**(2.0)+3.0),\
               1.0/8.0*(63.0*x_p**(5.0)-70.0*x_p**(3.0)+15.0*x_p)]).T
params = np.linalg.solve(A.T@A,A.T@y)
```

Note: for parameterization x domain must be shifted/scaled to [-1.1] to use Legendre functions

Calculate standard errors:

```
# Calculate variance in residuals
var = np.mean((y-np.polynomial.legendre.legval(x_p,params))**(2.0))

# Calculate se in parameters
ATAi = np.linalg.inv(A.T@A)
var_B = np.diag(ATAi)**(0.5)*var

# Calculate se in predictions
x_plot = np.linspace(min(x),max(x),200)
se = []
for i in x_plot:
    i = (i-2.5)/2.5
    h = np.array([(i*0+1),i,0.5*(3.0*i**(2.0)-1.0),\
                  0.5*(5.0*i**(3.0)-3.0*i),\
                  1.0/8.0*(35.0*i**(4.0)-30.0*i**(2.0)+3.0),\
                  1.0/8.0*(63.0*i**(5.0)-70.0*i**(3.0)+15.0*i)])
    se += [(h@ATAi@h.T)**(0.5)*var**(0.5)]
se = np.array(se)
```

Working Example - Analytic Estimation

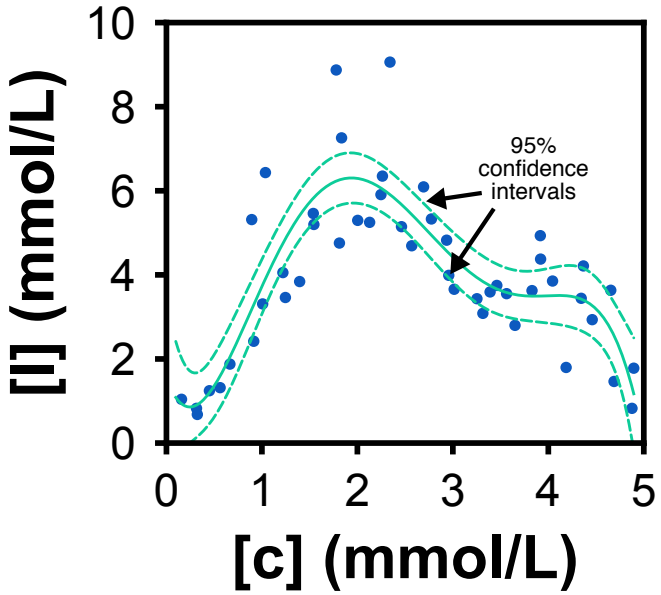
Python Implementation:

```
# Make a plot of the solution with prediction bounds
plt.figure(figsize=(5,5))
plt.scatter(x,y)
plt.plot(x_plot,np.polynomial.legendre.legval((x_plot-2.5)/2.5,params),color=(0.1,0.8,0.2))
plt.plot(x_plot,np.polynomial.legendre.legval((x_plot-2.5)/2.5,params)+se*1.96,\
         linestyle='--',color=(0.1,0.8,0.2)) # scaled by 1.96 for 95% confidence interval
plt.plot(x_plot,np.polynomial.legendre.legval((x_plot-2.5)/2.5,params)-se*1.96,\
         linestyle='--',color=(0.1,0.8,0.2)) # scaled by 1.96 for 95% confidence interval
plt.xlabel("[c] (mmol/L)")
plt.ylabel("[I] (mmol/L)")
plt.ylim([0,10])
plt.xlim([0,5])
plt.show()
```

The resulting least-squares model with 95% confidence bounds:

β = [3.85, 0.04, -3.30, 1.21, 0.25, -1.97]
std = [0.16, 0.27, 0.35, 0.42, 0.48, 0.55]

Note: higher uncertainties in higher order polynomials.



Bootstrap Analysis

What if we don't want to assume Gaussian statistics? What if we had a non-linear model? What if we forgot how to do error propagation with Taylor exp.?!

Let's say that instead of the N samples in this example, we had 10*N samples. With this extra data, how might we estimate the error in a model with just N samples?

How about:

1) Split the data into 10 sets of N,
2) fit 10 separate models,
3) calculate the standard error across the models?

This would be great (if we had 10*N samples instead of just N). But the idea behind bootstrapping is that we *can* simulate having 10*N samples, if we have a cheap estimate for our sample's parent population. With such an estimate, we can even simulate having 1000*N samples.

Can you think of a distribution that involves no approximations and has the same expected mean and variance as the sample distribution?

The sample distribution (with replacement) is our best guess without approximation of the population distribution

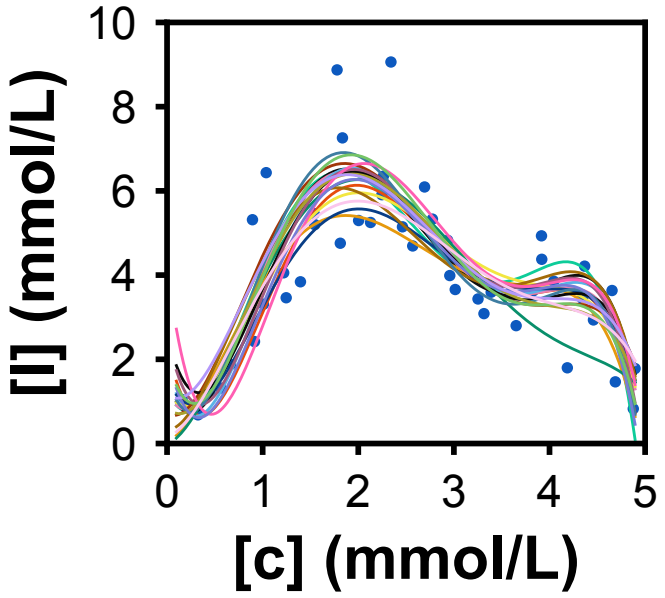
Working Example - Bootstrap

Python Implementation:

```
# bootstrap function
def boot(data,samples=1000):
    N = len(data[0])
    for i in range(samples):
        inds = np.random.randint(0,N,size=N)
        yield [ j[inds] for j in data ]

# Train 10000 models using bootstrap resampling
p = [] # holds the parameters for each model
y_p = [] # holds the predictions for each model
for i,j in boot([x_p,y],samples=10000):
    p += [np.polynomial.legendre.legfit(i,j,5)]
    y_p += [np.polynomial.legendre.legval((x_plot-2.5)/2.5,p[-1])] # x rescaled for [-1,1] domain
```

For each resampled dataset we fit a model. 20 examples are shown in the plot to the right



Working Example - Bootstrap

Python Implementation:

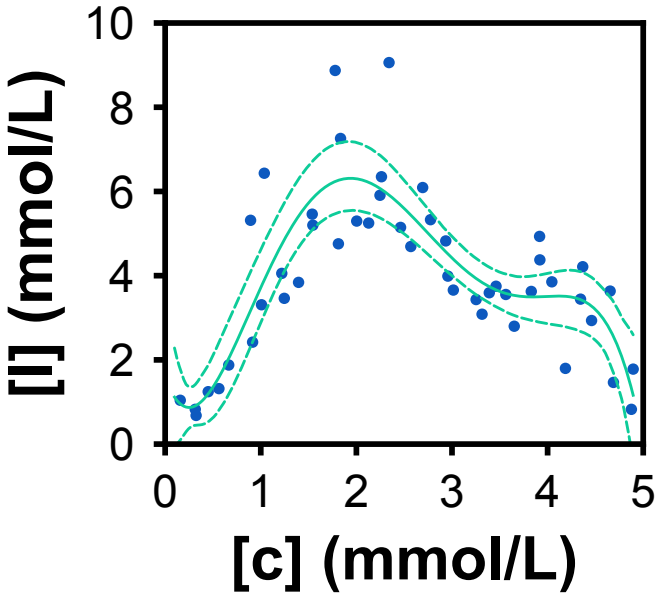
```
# bootstrap function
def boot(data,samples=1000):
    N = len(data[0])
    for i in range(samples):
        inds = np.random.randint(0,N,size=N)
        yield [ j[inds] for j in data ]

# Train 10000 models using bootstrap resampling
p = [] # holds the parameters for each model
y_p = [] # holds the predictions for each model
for i,j in boot([x_p,y],samples=10000):
    p += [np.polynomial.legendre.legfit(i,j,5)]
    y_p += [np.polynomial.legendre.legval((x_plot-2.5)/2.5,p[-1])] # x rescaled for [-1,1] domain
```

Resampling 10,000 times we can estimate the 95% confidence bounds by reporting the 250th and 9,750th largest values at each x (i.e., 2.5%*10,000)

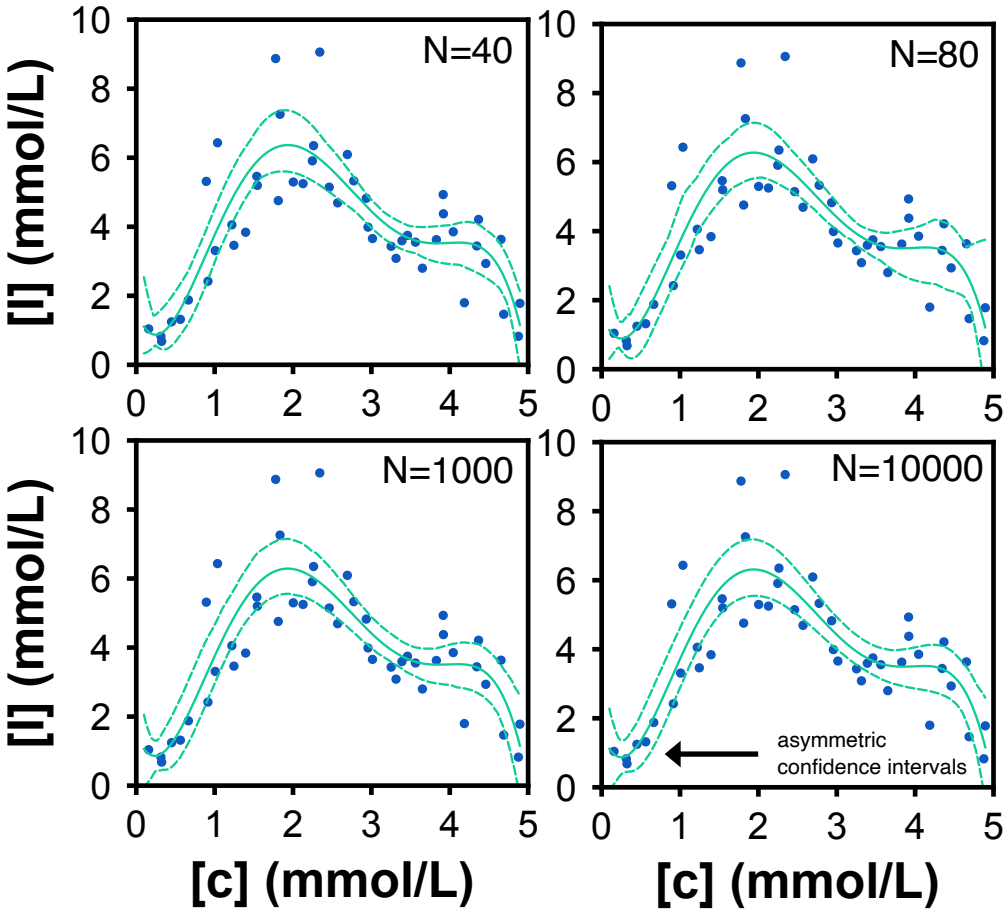
Calculating the standard deviation across fits yields parameter uncertainties:

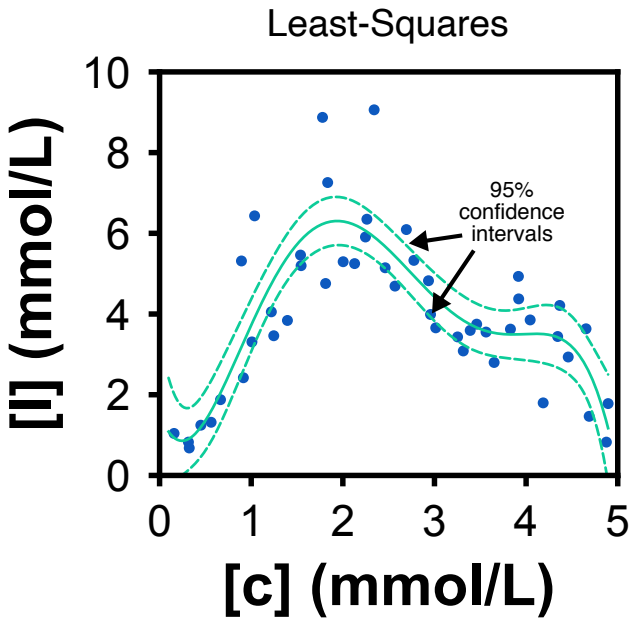
β = [3.85, 0.04, -3.29, 1.19, 0.26, -2.00]
std = [0.16, 0.27, 0.35, 0.45, 0.55, 0.55]



Working Example - Bootstrap

Example of error estimate convergence for N_{boot} = 40,80,1000, and 10000:





A suitable choice of N_{boot} is problem specific (depends on the behavior of the underlying population, something impressively large is usually used)

Bayesian Inference

What if we had a starting assumption for what the parameters should be?
What is the optimal way of updating our assumptions in light of new data?

Bayesian inference is the natural framework for answering these questions in a systematic and logically consistent way.

Logical proof of Bayes Theorem:

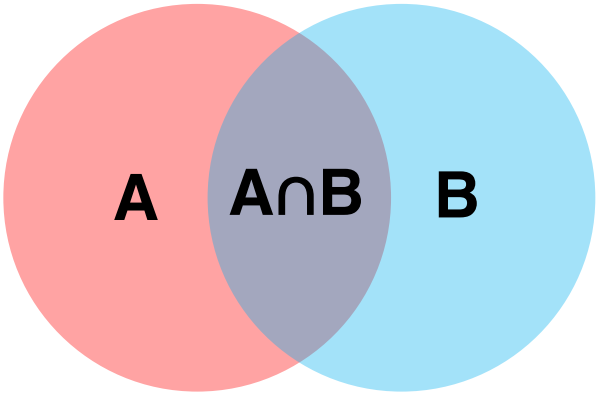
$$P(A \cap B) = P(A|B)P(B)$$

$$P(B \cap A) = P(B|A)P(A)$$

Since, $P(A \cap B) = P(B \cap A)$

Bayes Theorem $$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Second of two Venn diagrams today:



Bayesian Inference

How is this useful to parameter estimation? First, we can redefine **A** as **β**, our model parameters, and **B** as **D**, our observed data:

Bayes Theorem (β,D)

$$\mathbf{P(\beta|D) = \frac{P(D|\beta)P(\beta)}{P(D)}}$$

P(D|β) = "*likelihood* of data given parameters"
P(β) = "*prior* marginal probability of parameter"
P(β|D) = "*posterior* probability of parameters given data"
P(D) = "*evidence*" or simply "marginal probability of data"

- We can get a little further by recognizing that each P in the theorem can be a distribution. So the numerator on the rhs is the product of our prior expectations for **β** times the likelihood of **D** given **β**.
- This makes sense: a stronger starting assumption for **P(β)** (narrow distribution) will have a large effect on the product. A flat distribution for **P(β)** (reflects no prior knowledge) will have no effect on the product.
- **P**(**D**|**β**) is the likelihood of our data given a set of parameters. This can be calculated assuming a distribution (e.g., normal distribution with a hypothetical variance), our model, and the data.
- **P**(**D**) acts as a normalization constant. It doesn't actually affect the shape of the posterior, but it can formally be rewritten as int(**P(D|β)P(β)**d**β**).

Bayesian Inference – Working Example

In our working example we have a six parameter model and data:

$$y_{p,i} = \sum_{i=0}^{5} \beta_i P_i(x_i) = \beta_0 + \beta_1 x_i + \frac{\beta_2}{2}\left(3x_i^2 - 1\right) + \ldots$$

If we assume our errors are due to some Gaussian noise, then our likelihood can be expressed as:

$$P(\mathbf{D}|\boldsymbol{\beta}) = \prod_i^N e^{-\frac{(y_i - y_{p,i})^2}{2\sigma^2}} = e^{\frac{-1}{2\sigma^2}\sum_i^N (y_i - y_{p,i})^2}$$

In general, you can't easily evaluate your posterior, but you can sample it:

Metropolis Sampling Algorithm (normal multivariate, uniform prior):

1) Make an initial guess for $\boldsymbol{\beta_0}$ and calculate $\mathcal{X}^2(\boldsymbol{\beta_0})$ (sum in exponential above)

2) Make a new guess, $\boldsymbol{\beta}_{\text{trial}}$, and calculate the likelihood ratio*: $\frac{P(\mathbf{D}|\boldsymbol{\beta}_{\text{trial}})}{P(\mathbf{D}|\boldsymbol{\beta}_0)} = \frac{e^{\frac{-1}{2\sigma^2}\sum_i^N (y_i - y_{\text{trial},i})^2}}{e^{\frac{-1}{2\sigma^2}\sum_i^N (y_i - y_{0,i})^2}}$

3) Draw a uniform random number r∈[0,1], if r is less than the ratio, then $\boldsymbol{\beta}_0 \rightarrow \boldsymbol{\beta}_{\text{trial}}$

4) Repeat steps 2 and 3 until the distribution of parameters converges.

*log is typically used for numerical stability

Note: you will need to discard early samples since they don't reflect the steady-state posterior distribution.

Note: If you save $\boldsymbol{\beta}_0$ at each step the mean of the posterior reflects the maximum likelihood estimate of the parameters, and the first moment reflects the parameter uncertainty.

Working Example - MCMC

Python Implementation:

```
# argument of odds ratio
def xhi(x,y,p):
return np.sum((y-np.polynomial.legendre.legval((x-2.5)/2.5,p))**(2.0))

# MCMC Metropolis Hastings fit algorithm
def MCMC(params,xhi,N,max=10000,thresh=0.1,step=0.1,var=1):

    # Preinitialize random numbers (use log for numerical stability)
    r = np.log(np.random.uniform(0,1,max))

    # Initialize accepted counter and xhi_0 and xhi_best values
    xhi_0 = xhi(params[-1])

    # Loop with update based on odds ratio
    for i in range(max):

        # Generate trial parameters
        p_t = params[-1] + np.random.normal(loc=0.0,scale=step,size=len(params[-1]))

        # Calculate xhi for trial params
        xhi_t = xhi(p_t) # Calculate the xhi2 for the trial parameters

        # Calculate log odds ratio
        ratio = (-xhi_t+xhi_0)/(2.0*var**(2.0))

        # Update if log(rand) is less than log odds ratio
        if r[i] < ratio:
            params += [p_t]
            xhi_0 = xhi_t
        else:
            params += [params[-1]]
    return params
```

Working Example - MCMC

Python Implementation:

```
xhi_0 = lambda params: xhi(x,y,params)
params = MCMC([np.array([0.0,0.0,0.0,0.0,0.0,0.0])],xhi_0,len(y),max=10000)
p_array = np.vstack(params[-1000:])

# Plot the parameter trajectory
plt.figure(figsize=(5,5))
for i in range(len(params[-1])):
    plt.plot(range(len(params)),[ _[i] for _ in params ],label='p{}'.format(i))
plt.legend()

# Plot the parameter histograms
plt.figure(figsize=(5,5))
for i in range(len(params[-1])):
    plt.hist(p_array[:,i],'auto', density=True, alpha=0.75,label="p{}".format(i))
plt.legend()
plt.xlabel(r"$\beta_i$")
plt.show()
```

β = [3.88, 0.06, -3.21,
1.24, 0.19, -1.98]

std = [0.15, 0.22, 0.21, 0.32,
0.37, 0.43]