# A short report on PYTHON implementation of subset simulation

**Technical Report** · November 2017

DOI: 10.13140/RG.2.2.12605.56803



**1 author:**

Sundar VS
Callaway Golf Company
**35** PUBLICATIONS **373** CITATIONS

SEE PROFILE

# A short report on PYTHON implementation of subset simulation

V. S. Sundar
Post-doctoral researcher
Center for Multimodal Imaging and Genetics
Department of Radiology
University of California San Diego
svelkur@ucsd.edu
https://sites.google.com/site/sundarvelkur
November 1, 2017

Since Python is freely accessible to almost all the researchers worldwide, I thought of sharing my Python implementation of the subset simulation method [1] in this report. The code is an exact replica of my Matlab code [2]. The Markov Chains (MC) in each conditional region can be propagated in two way - (a) one chain at a time (b) all the chains take a step at once. To clarify, consider 5 MCs each required to transition through 10 states in the first intermediate conditional region. Strategy (a): MC 1 first propagates through 10 states before MC 2 takes its first step. Strategy (b): MC 1 to 5 each take a single step at once. The performance of both the strategies is more or less the same. A highly nonlinear two-dimensional problem [3] is chosen for the purpose of illustration.

## Numerical example

$$G\left(\boldsymbol{U}\right) = 4 - \frac{U_1}{4} + \sin(5U_1) - U_2; U_1, U_2 \sim N(0,1) \tag{1}$$

## Brute force Monte Carlo simulation

$$\hat{P}_F = \frac{1}{N}\sum_{k=1}^{N} I\left[G\left(\boldsymbol{u}^k\right) \leq 0\right] \tag{2}$$

where $I[\bullet]$ is the indicator function. Python implementation of MCS is given below:

```
import random
import numpy as np
import scipy
import math

# Performance function
def pfn(x):
    G=4-x[0]/4+math.sin(5*x[0])-x[1]
    return G

n=2                                     # Dimension of the problem
N=10000                                 # Number of samples
G=np.zeros((N))
for i in range (0,N):
    ux=np.random.normal(0,1,size=(n,1)) # Generating random numbers
    G[i]=pfn(ux)                        # Performance function evaluations

pf_mcs=(G < 0).sum()/N                  # Estimate of the probability of failure
pf_mcs
```

## Subset simulation

$$\bar{P}_F = P\left(F_1\right)\prod_{k=1}^{M-1} P\left(F_{i+1}\middle|F_i\right) \tag{3}$$

Here $M$ is the number of intermediate subsets, and the conditional probabilities, $P\left(F_{i+1}|F_i\right)$, are obtained using the Markov Chain Monte Carlo simulation with the Modified Metropolis-Hastings (MMH) algorithm [1].

### Importing the required packages

```
import random
import numpy as np
import time
import scipy
import math
from operator import itemgetter
from scipy.stats import multivariate_normal
```

### Defining the performance function

```
def pfn(x):
    G=4-x[0]/4+math.sin(5*x[0])-x[1]
    return G
```

### Defining the parameters

```
n=2                             # Dimension of the problem
N_ss=500                        # Number of samples per subset
pp=0.1                          # Intermediate failure probability
```

### Determining subset properties

```
nf_ini=int(N_ss*pp)             # Initial failure points
nf_ite=int(1/pp)                # Number of samples generated from each Markov Chain
```

### Estimating $P(F_1)$

Brute force Monte Carlo simulation is used in obtaining this quantity.

$$P(F_1) = \frac{1}{N_1} \sum_{k=1}^{N_1} I\left[G\left(\boldsymbol{u}^k\right) \leq 0\right] \tag{4}$$

```
ufailure=np.random.normal(0,1,size=(n,N_ss))
gfailure=np.zeros((N_ss))
for i in range (0,N_ss):
    gfailure[i]=pfn(ufailure[:,i])
pf1=nf_ini/N_ss                 # Initial pf, P(F1)
```

### Choosing the initial state for the Markov Chain

```
index, g = zip(*sorted(enumerate(gfailure), key=itemgetter(1))) # Sorting the g-values
g1=g[nf_ini-1]                  # First intermediate threshold level, g1
if g1<0:                        # Check if failure region has been reached
    g1=0
    pf1=(gfailure<0).sum()/N_ss
x1=ufailure[:,index[0:nf_ini]]  # Sort the samples according to ascending g-vales
gx1=g[0:nf_ini]                 # Choose the samples whose g-value is less than g1
sno=1                           # Counter for the number of subsets
gcollect[0]=gfinal              # This vector will store intermediate threshold values
gfinal=g1
```

### Propagating the Markov Chains till failure region is reached – strategy (a)

```
while (gfinal>=0):
    ua=x1
    ga=list(gx1)
    for i in range (0,nf_ini):      # For each Markov Chain
        x_seed=x1[:,i]              # Initial state for the Markov Chain
        g_seed=gx1[i]               # Corresponding g-value
        for j in range(1,nf_ite):   # Till nf_ite number of samples are obtained from each Chain
            ux= (x_seed - 0.5) + np.random.uniform(0,1,size=(1,n)) # Uniform proposal density function
            m1=multivariate_normal.pdf(ux[0])
            m2=multivariate_normal.pdf(x_seed)
            alpha=np.minimum(np.ones((1,n)),m1/m2)
            r=np.random.uniform(0,1,size=(1,n))
            for ii in range (0,n):  # Modified Metropolis-Hastings algorithm acceptance criterion
                if alpha[0][ii]<r[0][ii]:
                    ux[0][ii]=x_seed[ii]
            check=np.sum(alpha[0]>r[0])
```

```
            if check>0: # Performance function is evaluated only when an unique sample is obtained
                gx=pfn(ux[0])
                if gx<g1:
                    x_seed=ux[0]
                    g_seed=gx
            ua=np.append(ua,np.transpose([x_seed]),axis=1)
            ga.append(g_seed)
    ufailure=ua
    gfailure=ga
    index, g = zip(*sorted(enumerate(gfailure), key=itemgetter(1))) # Sorting the g-values
    x1=ufailure[:,index[0:nf_ini]] # Sort the samples according to ascending g-vales
    gx1=g[0:nf_ini]
    g1=g[nf_ini-1] # Threshold level for the sno-th subset
    gfinal=g1
    gcollect[sno]=gfinal
    sno+=1
```

### Propagating the Markov Chains till failure region is reached – strategy (b)

```
while (gfinal>=0):
    ua=x1
    ga=list(gx1)
    for j in range (1,nf_ite):          # Till nf_ite number of samples are obtained from each Chain
        for i in range(0,nf_ini):       # For each Markov Chain
            x_seed=ua[:,(j-1)*nf_ini+i] # Initial state for the Markov Chain
            g_seed=ga[(j-1)*nf_ini+i]   # Corresponding g-value
            ux= (x_seed - 0.5) + np.random.uniform(0,1,size=(1,n))
            m1=multivariate_normal.pdf(ux[0])
            m2=multivariate_normal.pdf(x_seed)
            alpha=np.minimum(np.ones((1,n)),m1/m2)
            r=np.random.uniform(0,1,size=(1,n))
            for ii in range (0,n):  # Modified Metropolis-Hastings algorithm acceptance criterion
                if alpha[0][ii]<r[0][ii]:
                    ux[0][ii]=x_seed[ii]
            check=np.sum(alpha[0]>r[0])
            if check>0: # Performance function is evaluated only when an unique sample is obtained
                gx=pfn(ux[0])
                if gx<g1:
                    x_seed=ux[0]
                    g_seed=gx
            ua=np.append(ua,np.transpose([x_seed]),axis=1)
            ga.append(g_seed)
    ufailure=ua
    gfailure=ga
    index, g = zip(*sorted(enumerate(gfailure), key=itemgetter(1)))
    x1=ufailure[:,index[0:nf_ini]]
    gx1=g[0:nf_ini]
    g1=g[nf_ini-1]
    gfinal=g1
    gcollect[sno,kkk]=gfinal
    sno+=1
```

### Determining the $\bar{P}_F$

```
if sno>1:
    pf_final_subset=sum(gi < 0 for gi in gfailure)/(N_ss)
    pf_subset=pf1*pf1**(sno-2)*pf_final_subset
    pf_ss=pf_subset # Estimate of the probability of failure
else :
    pf_ss=pf1
```

## References


[1] S.-K. Au, J. L. Beck, Estimation of small failure probabilities in high dimensions by subset simulation, Probabilistic Engineering Mechanics 16 (4) (2001) 263–277.

[2] V. S. Sundar, A short report on matlab implementation of subset simulation, ResearchGate.
URL http://dx.doi.org/10.13140/RG.2.1.3041.7444

[3] V. S. Sundar, M. D. Shields, Surrogate-enhanced stochastic search algorithms to identify implicitly defined functions for reliability analysis, Structural Safety 62 (2016) 1–11.