montecarlo_garcia

July 22, 2020

1 A Comprehensive Introduction to Monte Carlo Simulations

1.1 Andrew Garcia, 2019 - 2020

Table of Contents

1.1.1 What are Monte Carlo methods?

Monte Carlo (MC) methods are, typically, computational algorithms which can be used to predict the parameters of a certain event or properties correlated thereof, which tend to relate on the likelihood of these event's occurence.

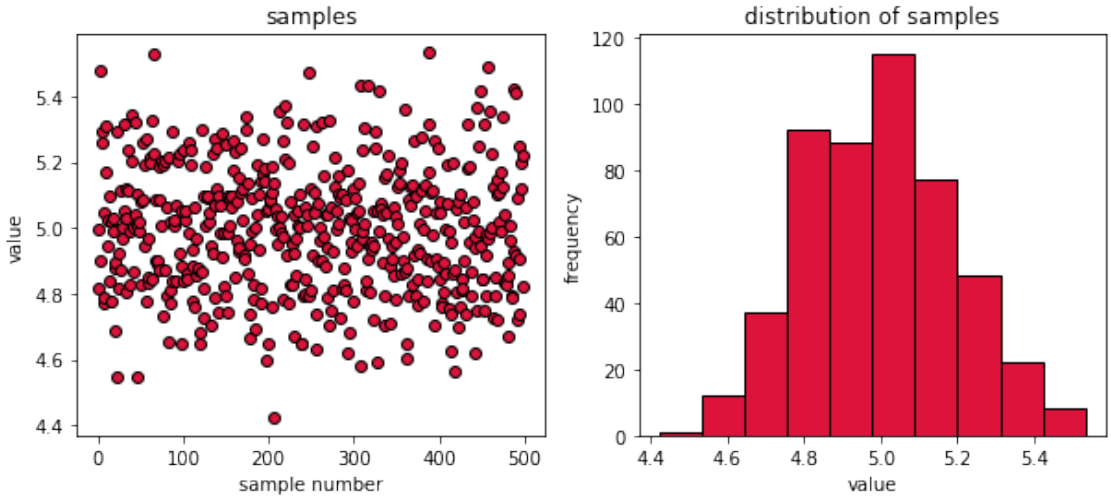
Thus, MC methods can indirectly predict a property of a feature which is correlated to a certain event, as is the popular example for estimating the value of π from the random sampling of points from the area of a circle enclosed by a square (https://academo.org/demos/estimating-pi-monte-carlo/)

1.1.2 Simple Monte Carlo

A standard deviation estimate In the purest sense, MC methods involve sampling from random distributions. We can use python to specify which distribution to draw samples from, though one can also use a specific probability density function (PDFN) were python not able to have the specified PDFN available.

This sampling can be done easily with python's **numpy.random** module. Obviously enough, were we to sample enough points from a random distribution, we would form said distribution:

[55]:
```
import random
import matplotlib.pyplot as plt

sampleno = 500
mean, sd = 5, 0.2
k, x = 0, []
while k < sampleno: x.append(random.gauss(mean, sd)); k+=1

fig = plt.figure()
plt.subplot(1,2,1)
plt.plot(x,'o',markeredgecolor='k',color='crimson'), plt.title('samples'), plt.
↪xlabel('sample number'), plt.ylabel('value')
plt.subplot(1,2,2)
plt.hist(x,color='crimson',edgecolor='k'), plt.title('distribution of␣
↪samples'), plt.xlabel('value'), plt.ylabel('frequency')

fig.set_size_inches(10,4)
plt.show()
```



Now, let's say for simplicity we studying a variable which we know is a linear combination of two dependent variables X1 and X2:

$Y = C_1X_1 + C_2X_2$

Here we assume the coefficients C_i have already been determined. Let's continue with a real world example:

Monte Carlo Simulations for a Candy Manufacturing Process

Modeling the process Let's assume we have a taffy making machine (**US3410230A** https://patents.google.com/patent/US3410230A/en) and want to estimate the *'softness'* of our candy (Y) from the amount of salt added (X_1) and the rotating paddle speed (X_2) . Knowing our machine, the rotating paddle speed is not completely steady; it oscillates exactly $\pm$ 10 speed units from its input. There is also some human error in how much salt is added per batch, and from such we have estimated a standard deviation.

Nonetheless, we have also performed a fit and found the coefficients for X1 and X2 to appropriately correlate them with Y. Thus, our system is defined the following way:

$Y = 0.6X_1 + 1.4X_2$

Y: Softness

X_1: Amount of salt - Gaussian distribution; $\sigma_1 = 3$

X_2: rotating paddle speed - Uniform distribution; $Range$ = [a,b] ; {a,b} = $<X_2>$ $\pm$ 10

With this information we can estimate the statistical distribution of our output (candy softness) for any input value with a simple MC algorithm:

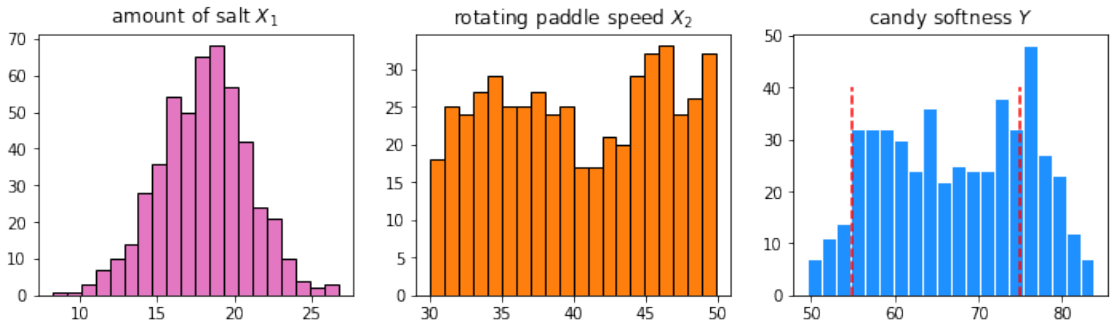
1) A sample from X_1's distribution is chosen
2) A sample from X_2's distribution is chosen
3) These two samples are operated with the function for Y above
4) 1-3 is repeated N times to form a distribution for Y

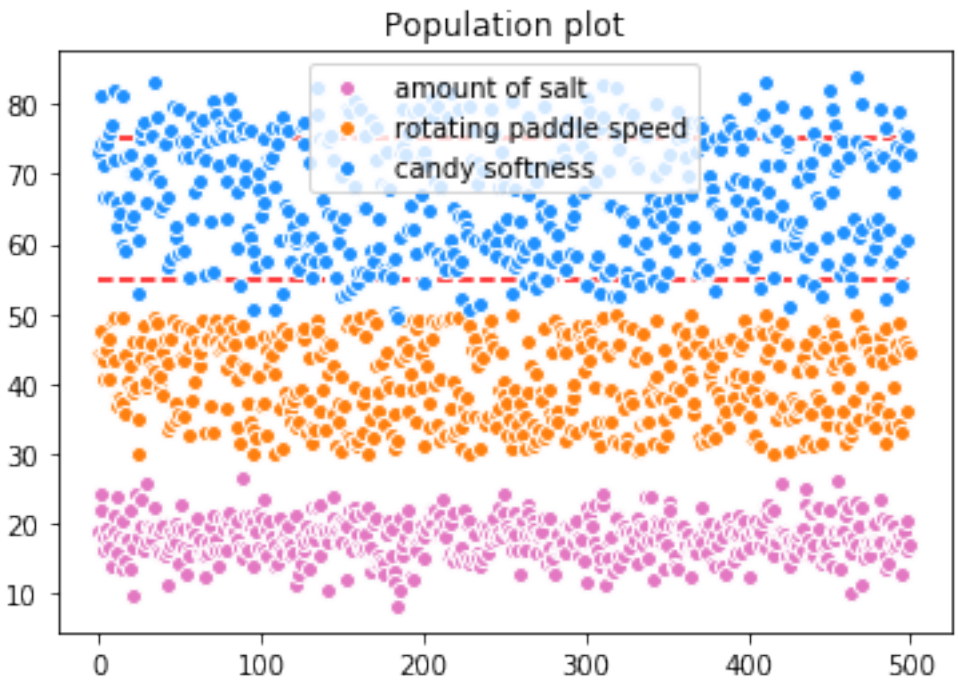
In addition, let's say we have a customer and our customer needs the candy to have softness ratings between 55 and 75. We can place these requirements in our distribution for softness (dashed red lines) and check if our products meet the Six-Sigma requirement i.e. 99.7% of the candy falling within customer specifications.

This can all be easily done with python :

[56]:
```
import numpy as np
def distgen(sampleno,dist_type,param1, param2):
    k, x = 0, []
    while k < sampleno: x.append(dist_type(param1,param2)); k+=1
    return np.array(x)

X1 = distgen(500,random.gauss,18, 3)
X2 = distgen(500,random.uniform,30, 50)

C1, C2 = 0.6,1.4
Y = C1*X1 + C2*X2
```

```
fig = plt.figure()
plt.subplot(1, 3, 1), plt.title('amount of salt $X_1$'), plt.
 ↪hist(X1,bins=20,color='C6',edgecolor='k')
plt.subplot(1, 3, 2), plt.title('rotating paddle speed $X_2$'), plt.
 ↪hist(X2,bins=20,color='C1',edgecolor='k')
plt.subplot(1, 3, 3), plt.title('candy softness $Y$'), plt.
 ↪hist(Y,bins=20,color='dodgerblue',edgecolor='w')
'customer specifications'
plt.vlines(55,0,40,linestyle='dashed',color='r'), plt.
 ↪vlines(75,0,40,linestyle='dashed',color='r')
fig.set_size_inches(12,3)
plt.show()

plt.title('Population plot')
plt.plot(X1,'o',color='C6',label='amount of salt',markeredgecolor='w')
plt.plot(X2,'o',color='C1',label='rotating paddle speed',markeredgecolor='w')
plt.plot(Y,'o',color='dodgerblue',label='candy␣
 ↪softness',markeredgecolor='w'),plt.legend()
plt.hlines(55,0,500,linestyle='dashed',color='r'), plt.
 ↪hlines(75,0,500,linestyle='dashed',color='r')
plt.show()
```

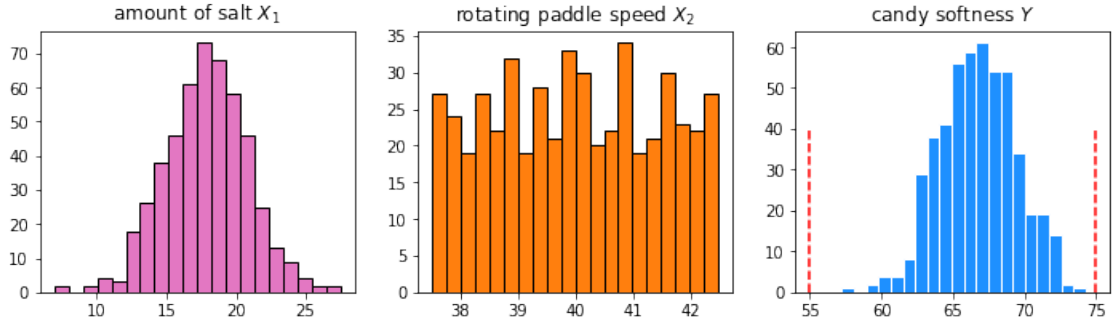


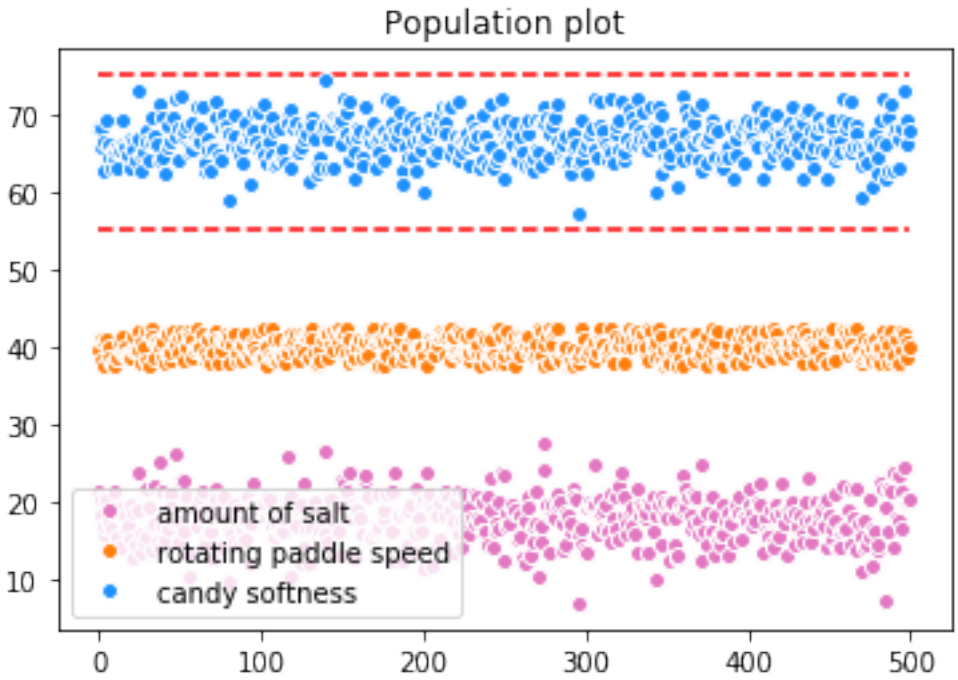


Making the process Six Sigma Through a brute trial and error optimization procedure we find that we need to replace our calender (a taffy pulling machine) with a new one which oscillates $\pm$ 2.5 speed units from 40. We make the modification of variable X_2 and thus see the Candy softness distribution now meets our customer specifications

[60]:
```
X1 = distgen(500,random.gauss,18, 3)
X2 = distgen(500,random.uniform,37.5, 42.5)

C1, C2 = 0.6,1.4
Y = C1*X1 + C2*X2

fig = plt.figure()
plt.subplot(1, 3, 1), plt.title('amount of salt $X_1$'), plt.
 ↪hist(X1,bins=20,color='C6',edgecolor='k')
plt.subplot(1, 3, 2), plt.title('rotating paddle speed $X_2$'), plt.
 ↪hist(X2,bins=20,color='C1',edgecolor='k')
plt.subplot(1, 3, 3), plt.title('candy softness $Y$'), plt.
 ↪hist(Y,bins=20,color='dodgerblue',edgecolor='w')
'customer specifications'
plt.vlines(55,0,40,linestyle='dashed',color='r'), plt.
 ↪vlines(75,0,40,linestyle='dashed',color='r')
fig.set_size_inches(12,3)
plt.show()
```

```
plt.title('Population plot')
plt.plot(X1,'o',color='C6',label='amount of salt',markeredgecolor='w')
plt.plot(X2,'o',color='C1',label='rotating paddle speed',markeredgecolor='w')
plt.plot(Y,'o',color='dodgerblue',label='candy␣
 ↪softness',markeredgecolor='w'),plt.legend()
plt.hlines(55,0,500,linestyle='dashed',color='r'), plt.
 ↪hlines(75,0,500,linestyle='dashed',color='r')
plt.show()
```





A Six-Sigma process can also be simplified as one where the customer requirements are within 3 standard deviations from the mean in both directions.

For a simple sanity check, we can fit the blue distribution Y to a probability density function. Assuming the distribution of Y is Gaussian, the center and spread parameters are simply normal mean and standard deviation, respectively, and can be calculated easily:

[61]:
```
print('mean',np.mean(Y))
print('sdev',np.std(Y))
```

```
mean 66.70661620772505
sdev 2.6668248652260163
```

With this information we can then the assessment of whether it meets six-sigma:

[62]:
```
print('customer specifications: \nlowest value (LB): 55 softness \nhighest␣
 ↪value (HB): 75 softness\n')
mean, thrsigma = np.mean(Y), 3*np.std(Y)
print('process output information')
print('mean - 3 sigma: {} (> 55 LB)'.format(mean-thrsigma))
print('mean + 3 sigma: {} (< 75 HB)'.format(mean+thrsigma))
print('\nmeets six-sigma')
```

```
customer specifications:
lowest value (LB): 55 softness
highest value (HB): 75 softness

process output information
mean - 3 sigma: 58.706141612047 (> 55 LB)
mean + 3 sigma: 74.7070908034031 (< 75 HB)

meets six-sigma
```

If the Y distribution is not Gaussian, we can fit it to a probability density function (PDFN) and get the parameters thereof (e.g. mean, spread, kurtosis, etc..). We can load **frame_pdsfit.py** from my **/statistics** repository to do so (see ***/statistics/pdsfit_tutorial*** to learn more) :

[63]:
```
import sys
sys.path.append('/home/andrew/scripts/statistics')

from frame_pdsfit import *
make(Y,'Candy softness (Y)',['gauss','lognorm','beta'])
```

```
Candy softness (Y)
normal_mean 66.70661620772505
normal_sdev 2.6668248652260163

Candy softness (Y)
lognorm_s/sigma 0.013250838697935016
lognorm_loc -134.95680885905722
lognorm_scale/median/exp_mean 201.63448570503533
```

```
Candy softness (Y)
beta_a 25.486248995269236
beta_b 17.780618123652882
beta_c 45.457724362768154
beta_d 36.07362791870497
```

[63]:
```
(['normal_mean',
  'normal_sdev',
  'lognorm_s/sigma',
  'lognorm_loc',
  'lognorm_scale/median/exp_mean',
  'beta_a',
  'beta_b',
  'beta_c',
  'beta_d'],
 [66.70661620772505,
  2.6668248652260163,
  0.013250838697935016,
  -134.95680885905722,
  201.63448570503533,
  25.486248995269236,
  17.780618123652882,
  45.457724362768154,
  36.07362791870497])
```

One can see the mean and standard deviation obtained from numpy are the same as those obtained from a Gaussian fit of the generated Y data. You may also notice that this distribution (histogram) is the same as the one obtained from **distgen()** (because it's the same data!)

1.1.3 Metropolis Monte Carlo

A subset of Markov Chain Monte Carlo (MCMC) techniques, this method was [for the most part] developed by Nicholas Metropolis in the 1940s for the Manhattan Project to calculate neutron transport rates in various materials and thus predict the explosive behavior of various fission weapons being designed at the time.

It is a powerful statistical analysis algorithm which can make good estimates on the equilibrium properties of a physical system by generating random configurations sampled from the system's corresponding statistical mechanics distribution. Though time cannot be accounted in a completely deterministic way, one can use MMC to study the evolution of a particular system.

The probability of transition Let's define a system with two energies, a ground state E_g and an excited state E_e, where $E_e > E_g$.

Assuming we place a particle in the ground state, we let the system have 4 probabilties:

$P(g)$ = probability of finding a particle in the ground state

$P(e)$ = probability of finding a particle in the excited state

$P_g(e)$ = probability of transition to the excited state

$P_e(g)$ = probability of transition to the ground state

The probability of finding the particle in either of these states ($P(e)$ or $P(g)$) can be represented by defining the aforementioned probability using the canonical ensemble:

$P_g = Q^{-1}\ e^{E_g/k_BT}$

and thus, we vanish the partition function by taking the ratio thereof:

$P_e/P_g = e^{\Delta E/k_BT}$

We can apply Bayes theorem and, consequently, the principle of microscopic reversibility to derive an expression for the probability of transition to the excited state:

$P_g(e)\ P(e) = P_e(g)\ P(g)$

Then substituting into the ratio of probabilities,

$P_g(e)\ /\ P_e(g) = e^{\Delta E/k_BT}$

Defining the system to ALWAYS go to ground state from excited state ($P_e(g) = 1$):

$P_g(e) = e^{\Delta E/k_BT}$

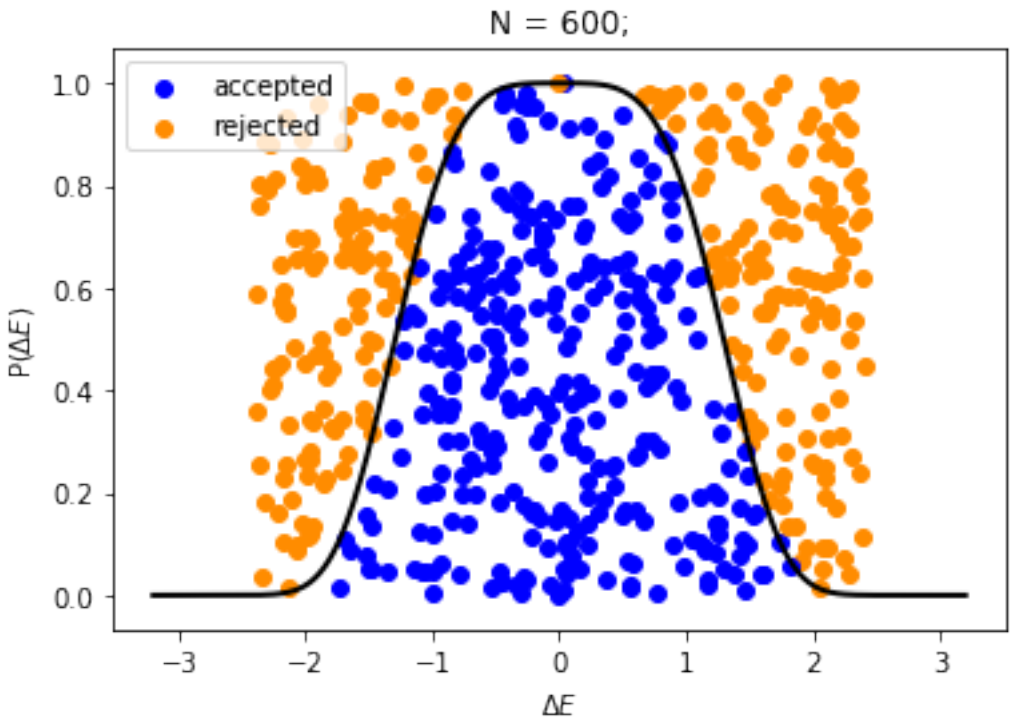
The rejection sampling criterion A Metropolis Monte Carlo simulation evaluates whether or not the transition occurs by checking if said probability is higher than a random number between 0 and 1:

- if $e^{\Delta E/k_BT}$ > random.uniform(0,1) accept transition
- else reject

The rationale behind this criterion lies in an infinite sampling of random particles from 0-1 below the new distribution (ratio of two distributions) approaching the area thereof, i.e. its cumulative density function. I simulate this graphically using the script below:

[5]:
```
import random as ran
import numpy as np
import matplotlib.pyplot as plt

#function for a Gaussian
def pdf(x,sigma):
    return (1/np.sqrt(2*np.pi*sigma**2))*np.exp((-x**2)/(2*sigma)**2)

#E as a function of x
def Ei(x):
    return (1/np.pi)*(x)**2

def accrej(n):

    #plot a Gaussian with the following parameters:
    sigma, x_e = 0.5, np.linspace(-4,4,500)
```

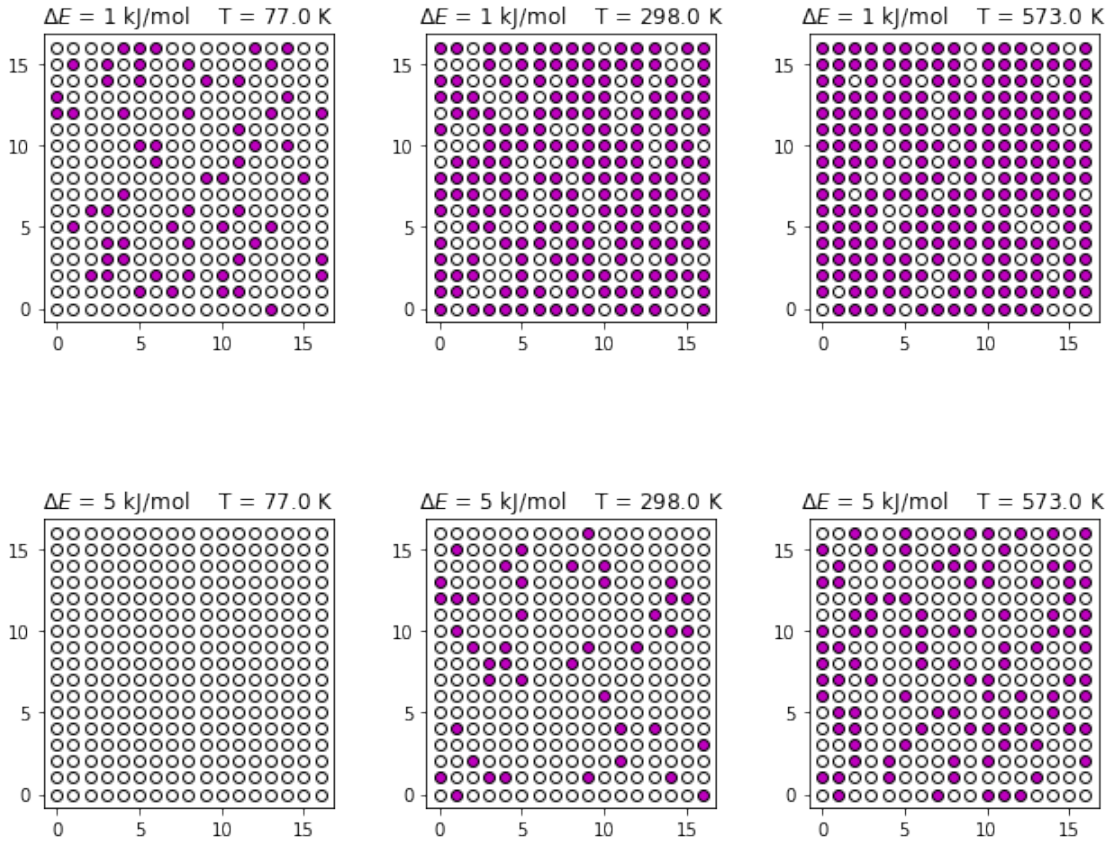
```
    x_g = x_e/5

    P = pdf(Ei(x_e),sigma) / pdf(Ei(x_g),sigma)

    plt.figure()
    plt.ylabel('P($\Delta E$)')
    plt.xlabel('$\Delta E$')
    plt.plot(x_e-x_g,P,linewidth=2,color='k')


    #the rejection sampling algorithm explained graphically
    k=0
    while k < n:
        xs, ys = ran.uniform(-3,3), ran.uniform(0,1)
        xs_g = xs/5
        Pxs = pdf(Ei(xs),sigma) / pdf(Ei(xs_g),sigma)
        [plt.scatter(xs-xs_g,ys,color='b') if Pxs > ys else plt.
↪scatter(xs-xs_g,ys,color='darkorange')]
        plt.title('N = {};'.format(k+1))
        k+=1

    plt.scatter(0,0,color='b', label='accepted'), plt.
↪scatter(0,1,color='darkorange',label='rejected')
    plt.legend()
    plt.show()

accrej(600)
```

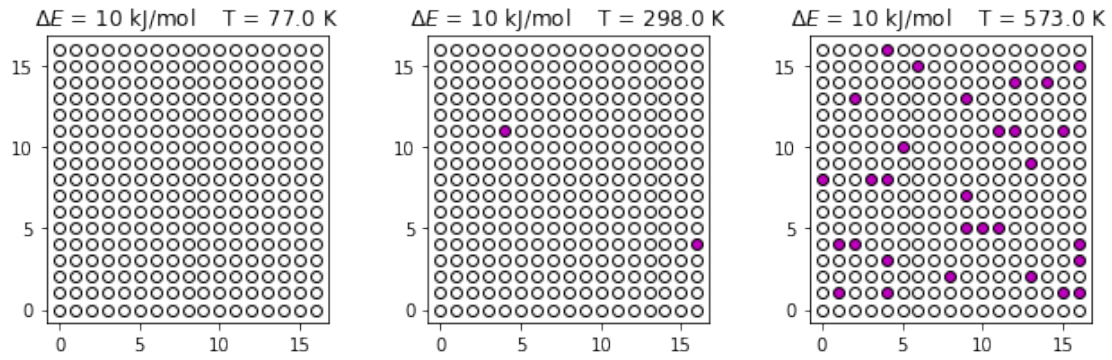


Particle grid of ground and excited states In this example I use Metropolis Monte Carlo to "simulate" a grid of particles in ground and excited states at different temperatures T and energy barriers ΔE. Ground and excited states will be represented as empty and filled circles, respectively.

[3]:
```
import numpy as np
import random as ran
import matplotlib.pyplot as plt

def P(dE,kBT):
    return np.exp(-dE/kBT)

R = 8.314e-3 #kJ mol-1 K-1
def atoms(N,dE,kBT):
    #k = 0
    L = int(np.sqrt(N))
    for i in range(L):
        for j in range(L):
            dE_fluc = ran.gauss(dE,0.001)
            plt.plot(i,j,'o',color='m',markeredgecolor='k') if P(dE_fluc,kBT) >␣
 ↪ran.uniform(0,1) \
            else plt.plot(i,j,'o',color='w',markeredgecolor='k')

    plt.title('$\Delta E$ = {} kJ/mol    T = {} K'.format(np.round(dE,2),kBT/R))
```

```
def atomgrids(N,dE):
    T1,T2,T3 = 77, 298, 573
    T = [T1,T2,T3]
    #plotting
    fig = plt.figure()
    for i in range(3):
        plt.subplot(1, 3, i+1)
        atoms(N,dE,R*T[i])

    fig.set_size_inches(9,3)
    fig.tight_layout()
    plt.show()

atomgrids(300,1)
atomgrids(300,5)
atomgrids(300,10)
```





With the canonical ensemble, kinetics are more favorableat higher T, whereas lower ΔE bring the transition to excited state barrier down and thus make it more probable. Thus, we get more particles in the excited state at high values of T and low ΔE

1.1.4 Kinetic Monte Carlo

(Reading)

This is also a method I know well. However, the explanation is a bit dense so I'd rather refer you to the following sources which I feel explains the method best. The general section is sorted in order of relevance. Happy reading [if interested in KMC]. -Andrew

KMC General

Kratzer, P. (2009) *Monte Carlo and kinetic Monte Carlo methods – a tutorial* (https://pdfs.semanticscholar.org/5a8f/7bb5e83c8f982004d8f75bf68cf0cf7fcce7.pdf)

Arthur F. Voter (2007) *Introduction to the Kinetic Monte Carlo Method* (https://link.springer.com/chapter/10.1007/978-1-4020-5295-8_1)

Kristen A. Fichthorn, and W. H. Weinberg (1991) *Theoretical foundations of dynamical Monte Carlo simulations* (https://doi.org/10.1063/1.461138)

Mie Andersen, Chiara Panosetti and Karsten Reuter (2019) *A Practical Guide to Surface Kinetic Monte Carlo Simulations*(https://doi.org/10.3389/fchem.2019.00202)

Jansen, A.P.J. (2003) *An Introduction to Kinetic Monte Carlo Simulations of Surface Reactions* (https://arxiv.org/abs/cond-mat/0303028)

KMC for chemical reaction networks

Gillespie, D. (1976) *A general method for numerically simulating the stochastic time evolution of coupled chemical reactions Author links open overlay panel* (https://doi.org/10.1016/0021-9991(76)90041-3)

Gillespie, D. (1977) *Exact stochastic simulation of coupled chemical reactions* (https://doi.org/10.1021/j100540a008)