SU(N) Lattice Gauge Theory in 2D Algorithms and Computations

Dibakar Sigdel

November 19, 2014

Contents

Part I

Monte Carlo Simulation of SU(N) Gauge Field in 2D

1 System Configuration

Consider a $L_1 \times L_2$ periodic lattice where one dimension is time and another dimension is space. On that lattice space, we are going to study a guage invariant nonabelian gauge theory. There are $(2L_1L_2)$ no of $SU(N)$ link variables which are denoted by

$$U_\mu(n_1, n_2); \quad 0 \leq n_1 < L_1; 0 \leq n_2 < L_2, \mu = 1, 2. \tag{1}$$

These link variables obey the periodic boundary conditions:

$$U_1(n_1, L_2) = U_1(n_1, 0); U_2(L_1, n_2) = U_2(0, n_2); 0 \leq n_1 < L_1; 0 \leq n_2 < L_2. \tag{2}$$

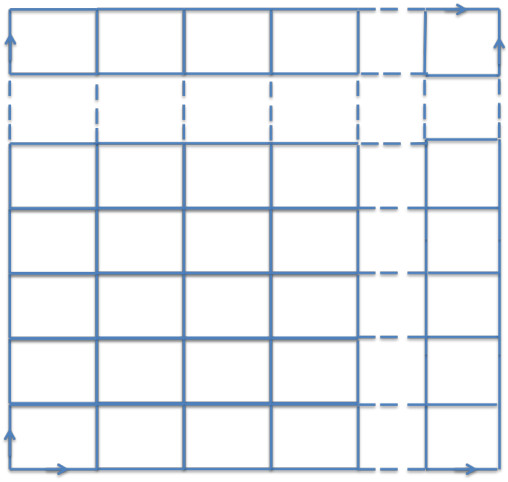


Figure 1: System configuration

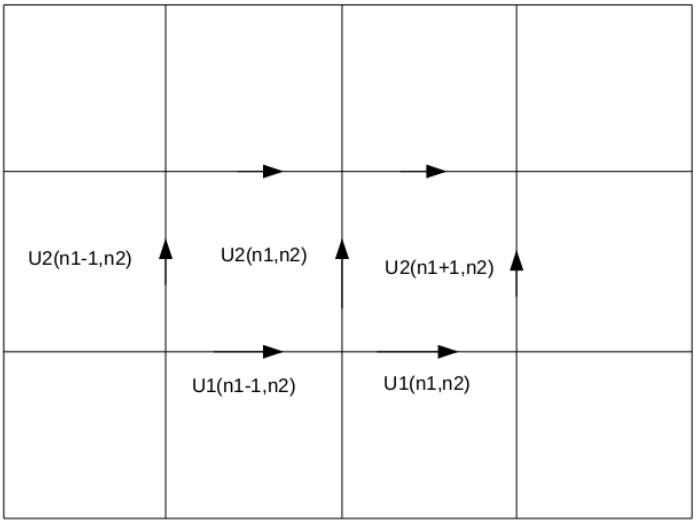


Figure 2: Link notations

Consider a configuration of link variables made up of SU(N) matrices as shown in figure above. Let us update a link variable $U_2(n_1, n_2)$ by multiplying it with another $SU(N)$ matrix V. We are interested to know how the trace of relevant plaquette variables $U_p(n_1 - 1, n_2)$ and $U_p(n_1, n_2)$ change with this modification:

$$U(n_1, n_2) \rightarrow U^{'}(n_1, n_2) = VU(n_1, n_2). \tag{3}$$

Wilson action is given by

$$S = \beta \sum_p Tr(U_p + U_p^{\dagger}) \tag{4}$$

2 Strategy to Update Links

The Plaquette variables $U_p(n_1, n_2)$ and $U_p(n_1 - 1, n_2)$ before a link $U(n_1, n_2)$ is updated are given by

$$U_p(n_1 - 1, n_2) = U^1(n_1 - 1, n_2)U^2(n_1, n_2)U^{1\dagger}(n_1 - 1, n_2 + 1)U^{2\dagger}(n_1 - 1, n_2)$$
$$U_p(n_1, n_2) = U^1(n_1, n_2)U^2(n_1 + 1, n_2)U^{1\dagger}(n_1, n_2 + 1)U^{2\dagger}(n_1, n_2) \quad (5)$$

The local contribution of these two plaquettes to the Wilson action before updating a link is

$$\underbrace{Tr\left(U_p(n_1 - 1, n_2) + U_p^\dagger(n_1 - 1, n_2)\right)}_{I-plaquette} + \underbrace{Tr\left(U_p(n_1, n_2) + U_p^\dagger(n_1, n_2)\right)}_{II-plaquette}$$

$$= Tr[U^1(n_1 - 1, n_2)U^2(n_1, n_2)U^{1\dagger}(n_1 - 1, n_2 + 1)U^{2\dagger}(n_1 - 1, n_2)$$
$$+U^2(n_1 - 1, n_2)U^1(n_1 - 1, n_2 + 1)U^{2\dagger}(n_1, n_2)U^{1\dagger}(n_1 - 1, n_2)]$$
$$+Tr[U^1(n_1, n_2)U^2(n_1 + 1, n_2)U^{1\dagger}(n_1, n_2 + 1)U^{2\dagger}(n_1, n_2)$$
$$+U^2(n_1, n_2)U^1(n_1, n_2 + 1)U^{2\dagger}(n_1 + 1, n_2)U^{1\dagger}(n_1, n_2)]$$

$$= Tr[\underbrace{U^2(n_1, n_2)}_{U_l}(\underbrace{U^{1\dagger}(n_1 - 1, n_2 + 1)U^{2\dagger}(n_1 - 1, n_2)U^1(n_1 - 1, n_2)}_{\Omega_1}$$
$$+\underbrace{U^1(n_1, n_2 + 1)U^{2\dagger}(n_1 + 1, n_2)U^{1\dagger}(n_1, n_2)}_{\Omega_2})]$$
$$+Tr[\underbrace{U^{2\dagger}(n_1, n_2)}_{U_l}(\underbrace{U^{1\dagger}(n_1 - 1, n_2)U^2(n_1 - 1, n_2)U^1(n_1 - 1, n_2 + 1)}_{\Omega_1^\dagger}$$
$$+\underbrace{U^1(n_1, n_2)U^2(n_1 + 1, n_2)U^{1\dagger}(n_1, n_2 + 1)}_{\Omega_2^\dagger})]$$

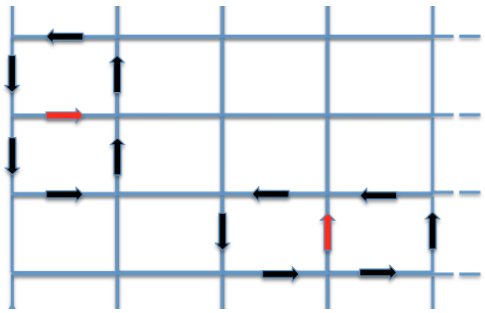


Figure 3: Orientation of links in staple

$$= Tr[U_l \underbrace{(\Omega_1 + \Omega_2)}_{\Sigma}] + Tr[U_l^\dagger \underbrace{(\Omega_1^\dagger + \Omega_2^\dagger)}_{\Sigma^\dagger}]$$

$$= Tr[U_l \Sigma] + Tr[U_l^\dagger \Sigma^\dagger]$$

Here Ω_1 and Ω_2 are product of three gauge link variables that build up the plaquettes together with $(U(n_1, n_2) = U_l)$. The quantity Σ is sum over those two staples. With these variables, contribution from two plaquettes under consideration can be separated from total action as

$$S = \beta Tr[U_l \Sigma + \Sigma^\dagger U_l^\dagger] + \beta \sum_{p \neq I, II} Tr(U_p + U_p^\dagger). \tag{6}$$

When a link U_l is updated, this quantity looks like

$$S_{new} = \beta Tr[V U_l \Sigma + \Sigma^\dagger U_l^\dagger V^\dagger] + \beta \sum_{p \neq I, II} Tr(U_p + U_p^\dagger). \tag{7}$$

Representing product '$U_l\Sigma$' by 'W' matrix one can express it as

$$S_{new} = \beta \left(Tr(VW + W^\dagger V^\dagger) \right) + \beta \sum_{p \neq I, II} Tr(U_p + U_p^\dagger). \tag{8}$$

2.1 Case - I: For SU(2)

One can start with action

$$S = \beta Tr[U_l \Sigma + \Sigma^\dagger U_l^\dagger] + 2\beta \sum_{p \neq I,II} Tr(U_p). \tag{9}$$

For SU(2), the sum of two SU(2) matrices becomes propertional to a SU(2) matrix. Let,

$$\Sigma = cA; c = \sqrt{\det(\Sigma)}. \tag{10}$$

Then S looks like,

$$S = \beta c Tr[U_l A + A^\dagger U_l^\dagger] + 2\beta \sum_{p \neq I,II} Tr(U_p). \tag{11}$$

Since the product $U_l A$ is again a SU(2) matrix,

$$S = 2\beta c Tr[U_l A] + 2\beta \sum_{p \neq I,II} Tr(U_p). \tag{12}$$

When a link U_l is updated, this quantity looks like

$$S_{new} = 2\beta c Tr[V U_l A] + ... \tag{13}$$

Representing product VU_l by $U_l^{'}$ matrix one can express it as

$$S_{new} = 2\beta c Tr(U_l^{'} A) + ... \tag{14}$$

Let

$$X = U_l^{'} A \tag{15}$$

Then

$$\boxed{S_{new} = 2\beta c Tr(X) + ...} \tag{16}$$

One can generate a matrix X using "Heatbath Algorithm". From this X matrix, an updated link can be obtained as

$$U_l^{'} = X A^\dagger = X \Sigma \frac{1}{c}. \tag{17}$$

2.2 Case - II: For SU(N); N > 2

Again one can start with new look of action

$$S_{new} = \beta \left(Tr(VW + W^{\dagger}V^{\dagger}) \right) + \beta \sum_{p \neq I,II} Tr(U_p + U_p^{\dagger}). \tag{18}$$

To construct a SU(N) matrix V, one can take a following SU(2) matrix:

$$SU(2) = \begin{pmatrix} a & b \\ -b^* & a^* \end{pmatrix}; |a|^2 + |b|^2 = 1. \tag{19}$$

By picking some value of integer α and β such that $1 \leq \alpha \leq N, 2 \leq \beta \leq N-1$, matrix element of V can be assigned as

$$\begin{aligned} V_{\alpha\alpha} = a; V_{\alpha\beta} = b; \\ V_{\beta\alpha} = -b^*; V_{\beta\beta} = a^*; \\ V_{kk} = 1; k \neq \alpha, \beta; \\ V_{kl} = 0; k \neq \alpha, \beta; l \neq \alpha, \beta. \end{aligned} \tag{20}$$

For example: For $\alpha = 3, \beta = 5$;

$$V = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & ... \\ 0 & 1 & 0 & 0 & 0 & 0 & ... \\ 0 & 0 & a & 0 & b & 0 & ... \\ 0 & 0 & 0 & 1 & 0 & 0 & ... \\ 0 & 0 & -b^* & 0 & a^* & 0 & ... \\ 0 & 0 & 0 & 0 & 0 & 1 & ... \\ . & . & . & . & . & . & ... \\ . & . & . & . & . & . & ... \\ . & . & . & . & . & . & ... \end{pmatrix}. \tag{21}$$

Diagonal elements of product of V and W is given by:

$$(VW)_{kk} = \sum_{l=1}^{N} V_{kl} W_{lk}. \tag{22}$$

$$\therefore Tr(VW) = \sum_{k} \sum_{l} V_{kl} W_{lk}.$$

(23)

With specific form of matrix V, one can get:

$$\begin{aligned} Tr(VW) &= V_{\alpha\alpha}W_{\alpha\alpha} + V_{\alpha\beta}W_{\beta\alpha} + V_{\beta\alpha}W_{\alpha\beta} + V_{\beta\beta}W_{\beta\beta} + \sum_{k\neq\alpha,\beta} V_{kk}W_{kk} \\ &= aW_{\alpha\alpha} + bW_{\beta\alpha} - b^*W_{\alpha\beta} + a^*W_{\beta\beta} + \sum_{k\neq\alpha,\beta} W_{kk} \end{aligned} \tag{24}$$

Similarly, for Product of matrix $(VW)^\dagger$:

$$\begin{aligned} Tr(W^\dagger V^\dagger) &= W^*_{\alpha\alpha}V^*_{\alpha\alpha} + W^*_{\beta\alpha}V^*_{\alpha\beta} + W^*_{\beta\beta}V^*_{\beta\beta} + \sum_{k\neq\alpha,\beta} W^*_{kk}V^*_{kk} \\ &= W^*_{\alpha\alpha}a^* + W^*_{\beta\alpha}b^* - W^*_{\alpha\beta}b + W^*_{\beta\beta}a + \sum_{k\neq\alpha,\beta} W^*_{kk} \end{aligned} \tag{25}$$

Finally

$$\begin{aligned} Tr(VW) + Tr(W^\dagger V^\dagger) &= a(W_{\alpha\alpha} + W^*_{\beta\beta}) + a^*(W^*_{\alpha\alpha} + W_{\beta\beta}) \\ -b(W^*_{\alpha\beta} - W_{\beta\alpha}) - b^*(W_{\alpha\beta} - W^*_{\beta\alpha}) &+ \sum_{k\neq\alpha,\beta} (W^*_{kk} + W_{kk}) \end{aligned} \tag{26}$$

If one considers a SU(2) matrix $V^{'}$ made up of elements of matrix V as

$$V^{'} = \begin{pmatrix} V_{\alpha\alpha} & V_{\alpha\beta} \\ V_{\beta\alpha} & V_{\beta\beta} \end{pmatrix}; \tag{27}$$

and matrix $W^{'}$ by taking elements from matrix W which combine with elements of V as

$$W^{'} = \begin{pmatrix} W_{\alpha\alpha} & W_{\alpha\beta} \\ W_{\beta\alpha} & W_{\beta\beta} \end{pmatrix}, \tag{28}$$

then action can be conveniently expressed as

$$S = \beta \left[Tr(V^{'}W^{'}) + Tr(W^{'\dagger}V^{'\dagger}) \right] + \sum_{k\neq\alpha,\beta} (W^*_{kk} + W_{kk}). \tag{29}$$

Let

$$X = (W_{\alpha\alpha} + W^*_{\beta\beta})$$

$$Y = (W^*_{\alpha\beta} - W_{\beta\alpha}) \tag{30}$$

Then

$$Tr(V^{'}W^{'}) + Tr(W^{'\dagger}V^{'\dagger}) = aX + a^*X^* - bY - b^*Y^* \tag{31}$$

Pulling out a factor of $K = \sqrt{XX^* + YY^*}$ from LHS of (31) ,

$$Tr(V^{'}W^{'}) + Tr(W^{'\dagger}V^{'\dagger}) = K[a\frac{X}{K} + a^*\frac{X^*}{K} - b\frac{Y}{K} - b^*\frac{Y^*}{K}] \tag{32}$$

Let

$$\frac{X}{K} = \bar{X}; \frac{Y}{K} = \bar{Y}; \tag{33}$$

then

$$Tr(V^{'}W^{'}) + Tr(W^{'\dagger}V^{'\dagger}) = K(a\bar{X} + a^*\bar{X}^* - b\bar{Y} - b^*\bar{Y}^*). \tag{34}$$

Let

$$Z = \begin{pmatrix} \bar{X}^* & -\bar{Y}^* \\ \bar{Y} & \bar{X} \end{pmatrix}. \tag{35}$$

Where Z is also a SU(2) matrix with $|\bar{X}|^2 + |\bar{Y}|^2 = 1$. Then we get

$$Tr(V^{'}W^{'}) + Tr(W^{'\dagger}V^{'\dagger}) = KTr(V^{'}Z^{\dagger}). \tag{36}$$

Hence;

$$\boxed{S_{new} = \beta KTr(V^{'}Z^{\dagger}) + ...} \tag{37}$$

Geting Updated Link

The main idea is to generate V which is a SU(N) matricx. For this we need to know $V^{'}$ which is SU(2) matrix. We know new action is

$$S_{new} = KTr(V^{'}Z^{\dagger}) + ... \tag{38}$$

Let

$$V' = V'' Z \tag{39}$$

Such that

$$S_{new} = KTr(V' Z^{\dagger}) + ... = KTr(V'') + ... \tag{40}$$

We generate this V'' with probability

$$P(V'') = e^{\beta KTr(V'')} \tag{41}$$

For this we use Cutz-Kennedy-Pendeleton (CKP) method. Once we get this V'' , we use

$$V' = V'' Z \tag{42}$$

Knowing this V' we can get V which is a SU(N) matrix. This help us to get

$$U'(n_1, n_2) = VU(n_1, n_2) \tag{43}$$

In case of SU(3), there are three different possibilities for V matrix say - R, S and T. Then we update the corresponding link as

$$U'(n_1, n_2) = RSTU(n_1, n_2) \tag{44}$$

$$R = \begin{pmatrix} a & b & 0 \\ -b^* & a^* & 0 \\ 0 & 0 & 1 \end{pmatrix}; S = \begin{pmatrix} c & 0 & d \\ 0 & 1 & 0 \\ -d^* & 0 & c^* \end{pmatrix}; T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & e & f \\ 0 & -f^* & e^* \end{pmatrix} \tag{45}$$

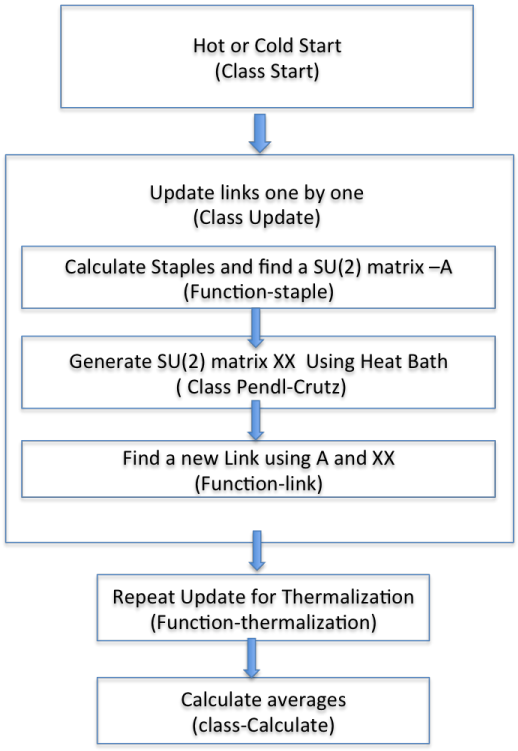
Once one get matrix V, the product VW in (18) must be updated while finding new Z matrix for another version of V matrix in equation (35).

Part II

Python Codes:SU(2)

In order to get the required numerical results, plots etc, there are different programming 'objects' like 'Class' and 'function'. There is a specific class like *Start, Update, Calculate* etc to perform specific part of whloe coding. The function in one class can call another function in another class by using concept of inheritance. The 'Algorithmic Sateps' gives the rough idea about how one class is related to other classes.

3 Algorithmic Steps for SU(2)



4 Hot or Cold Start

There are two starting points to begin the simulation of whole configuration. One is cold start and another is hot start. In cold start, every link is assigned as an identity matrix using function *cold_start* in Class *Start*.

```
def cold_start(self):

                I = np.matrix(np.identity(2))
                UU = [[[I  for x  in  range(self.L)]\
                             for y  in  range(self.L)]\
                               for r  in  range(2)]


                 return UU
```

While in hot start, a function *ransu2* generates a random SU(2) matrix:

$$SU(2) = \begin{bmatrix} \cos\theta e^{i\phi} & \sin\theta e^{i\xi} \\ -\sin\theta e^{-i\xi} & \cos\theta e^{-i\phi} \end{bmatrix}; \phi, \xi \in [-\pi, \pi], \theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]. \qquad (46)$$

For each of the links, angles ϕ and ξ are generated randomly within the range $[-\pi, \pi]$ while angle θ given by: $\theta = \cos^{-1}(2t-1)$. Where t is randomly generated within interval [0,1]. This makes θ in between $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

```
def ransu2(self):

                ai = complex(0,1)
                t = [random.random(),random.random(),\
                       random.random()]
                xi = (math.pi*(2*t[0]-1))
                theta =0.5*(math.acos(2*t[1]-1))
                phi = (math.pi*(2*t[2]-1))
                a = [0.0  for l  in  range(2)]
                a = [math.cos(theta)*(cmath.exp(ai*phi)),\
                      math.sin(theta)*(cmath.exp(ai*xi))]

                SU2 = []
                SU2 = np.matrix([[a[0],a[1]],[-a[1].conjugate(),a[0].conjugate()]])
                 return SU2
```

Function *hot_start* calls function *randsu2* for each of the links in the configuration. It returns a whole configuration with totally randomized SU(2) matrices.

```
def hot_start(self):

                    I = np.matrix(np.identity(2))
                    U = [[[I  for x  in  range(self.L)]
                                for y  in  range(self.L)]
                                      for z  in  range(2)]

                     for i  in  range (2):
                            for j  in  range(self.L):
                                for k  in  range(self.L):
                                       U[i][j][k] = self.ransu2()

                     return U
```

4.1 Codes for Class *Start*

```
class Start(object):

          def __init__(self,L):
                 self.L = L

          def cold_start(self):
                 I = np.matrix(np.identity(2))
                 UU = [[[I  for x  in  range(self.L)]
                             for y  in  range(self.L)]
                               for r  in  range(2)]
                  return UU


          def ransu2(self):
                 ai = complex(0,1)
                 t = [random.random(),random.random(),\
                       random.random()]
                 xi = (math.pi*(2*t[0]-1))
                 theta =0.5*(math.acos(2*t[1]-1))
                 phi = (math.pi*(2*t[2]-1))
                 a = [0.0  for l  in  range(2)]
                 a = [math.cos(theta)*(cmath.exp(ai*phi)),\
                       math.sin(theta)*(cmath.exp(ai*xi))]
                 SU2 = []
                 SU2 = np.matrix([[a[0],a[1]],[-a[1].conjugate(),a[0].conjugate()]])
                  return SU2


          def hot_start(self):
                  I = np.matrix(np.identity(2))
```

```
                U = [[[I  for x  in  range(self.L)]
                            for y  in  range(self.L)]
                                  for z  in  range(2)]
                 for i  in  range (2):
                        for j  in  range(self.L):
                            for k  in  range(self.L):
                                  U[i][j][k] = self.ransu2()
                 return U
```

4.2 Are links in 'Hot-Start' really randomized?

One can check whether these links are really randomized or not just by making a plot of histogram for eigen values of random SU(2) matrices. For sufficiently randomized distribution, eigen value η of SU(2) matrix follows the distribution function: $p(\eta)d\eta = \frac{2}{\pi}\sin^2\eta d\eta$. There is a class *Check_Randomization* which plots the required histogram.

```
class Check_Randomization(object):

          def find_eta(self):
                  ai =  complex(0,1)
                  su2 = self.ransu2()
                  w,v = LA.eig(su2)
                  #from numpy import linalg as LA
                  eta = abs((-ai*cmath.log(w[0])).real)
                   return eta


          def plot_function(self):
                 x = np.arange(0.0,np.pi,0.01)
                 y = (2.0/np.pi)*(np.sin(x))**2
                 plt.figure(11)
                 plt.scatter(x,y)
                 plt.show()
                  return


          def histogram(self,LN,SN):
                  #LN = no of trial matrix(large number)
                  #SN = no of partition for histogram(small number)
                  x = []
                  ct = 0
                   while ct < LN+1:
                     eta = self.find_eta()
                     x.append(eta)
                     ct = ct+1
                  self.plot_function() # plot function
                  num_bins = SN
```

```
                    plt.figure(11)
                    plt.xlabel(' eigen value: eta')
                    plt.ylabel('normalized probability' )
                    plt.hist(x,num_bins, normed= 1.0, facecolor='green',alpha = 0.5)
                     return

Check_Randomization().histogram(50000,50)
```

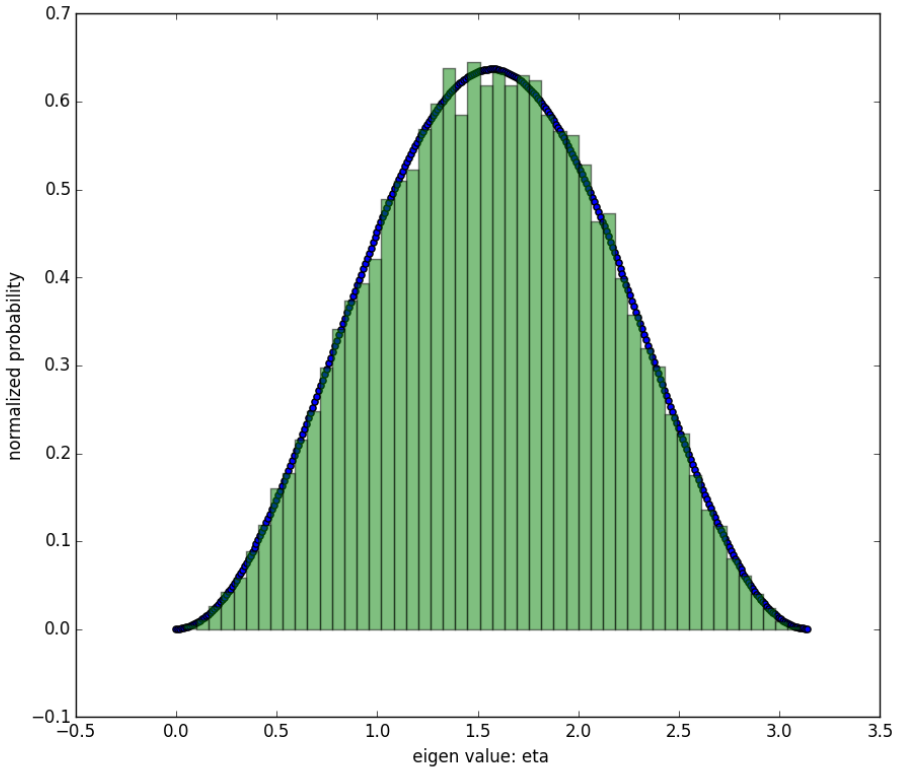


Figure 4: Histogram representing normalized distribution of eigen value η of randomly generated SU(2) matrices. Solid line is plot of normalized distribution function : $p(\eta) = \frac{2}{\pi}\sin^2\eta$.

5 Generating SU(2) Matrices with Heat Bath

To generate SU(2) matrices using Heat Bath algorithm, a Class *Pendl-Crutz* implements the methods according to (I) M. Crutz and (II) Kennedy & Pendelton. In both of these methods the SU(2) matrix of form:

$$SU(2) = \begin{bmatrix} a_0 + ia_3 & a_2 + ia_1 \\ -a_2 + ia_1 & a_0 - ia_3 \end{bmatrix} \tag{47}$$

is used with $a_0 \in [-1, 1]$ generated by using 'Heat Bath Algorithm' and

$$\begin{aligned} a &= \sqrt{1 - a_0^2} \\ a_1 = a\sin\theta\cos\phi; a_2 = a\sin\theta\sin\phi; a_3 &= a\cos\theta \\ \phi \in [-\pi, \pi]; \theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]. \end{aligned} \tag{48}$$

5.1 Kennedy & Pendelton's method

This method includes the following steps to determine a_0:
Step A. Generate two uniformly distributed pseudo-random numbers R and R' in the unit interval.
Step B. Set $X \leftarrow -\frac{(lnR)}{\alpha}, X' \leftarrow -\frac{(lnR')}{\alpha}$;
Step C. Set $C \leftarrow \cos^2(2\pi R'')$ with R'' another random number in [0,1];
Step D. Let $A \leftarrow XC$;
Step E. Let $\delta \leftarrow X' + A$;
Step F. If $R'''^2 > 1 - \frac{1}{2}\delta$ for R''' pseudo random and uniform in (0,1], go back to step A;
Step G. Set $a_0 \leftarrow 1 - \delta$.

```
def pendlgnr(self):
                    count = 0
                    failed = 0

                    while (count < 1):
                       r = [ random.uniform(0.0,1.0),\
                             random.uniform(0.0,1.0),\
```

```
                    random.uniform(0.0,1.0),\
                    random.uniform(0.0,1.0)]
                x  = [-(math.log(r[0])/(self.det*self.alpha)),\
                    -(math.log(r[1])/(self.det*self.alpha))]

                C = (math.cos(2.0*math.pi*r[2]))**2
                A = x[0]*C
                delta = x[1]+A

                 if (r[3]**2) < (1-(0.5*delta)):
                    a0 = (1- delta)
                    count = count+1
                    XX = self.su2(a0)
                     return failed, XX
                 else:
                   count = 0
                   failed = failed+1
```

5.2 Checking Pendelton's Method

One can check the distribution of SU(2) matrices to know whether it is following the distribution of variable a_0 as expressed in the equation:

$$dp(a_0) = C\sqrt{(1-a_0^2)}e^{(\beta k a_0)}da_0. \tag{49}$$

For simplicity, one can set $k = 1$ and C is some normalization constant.

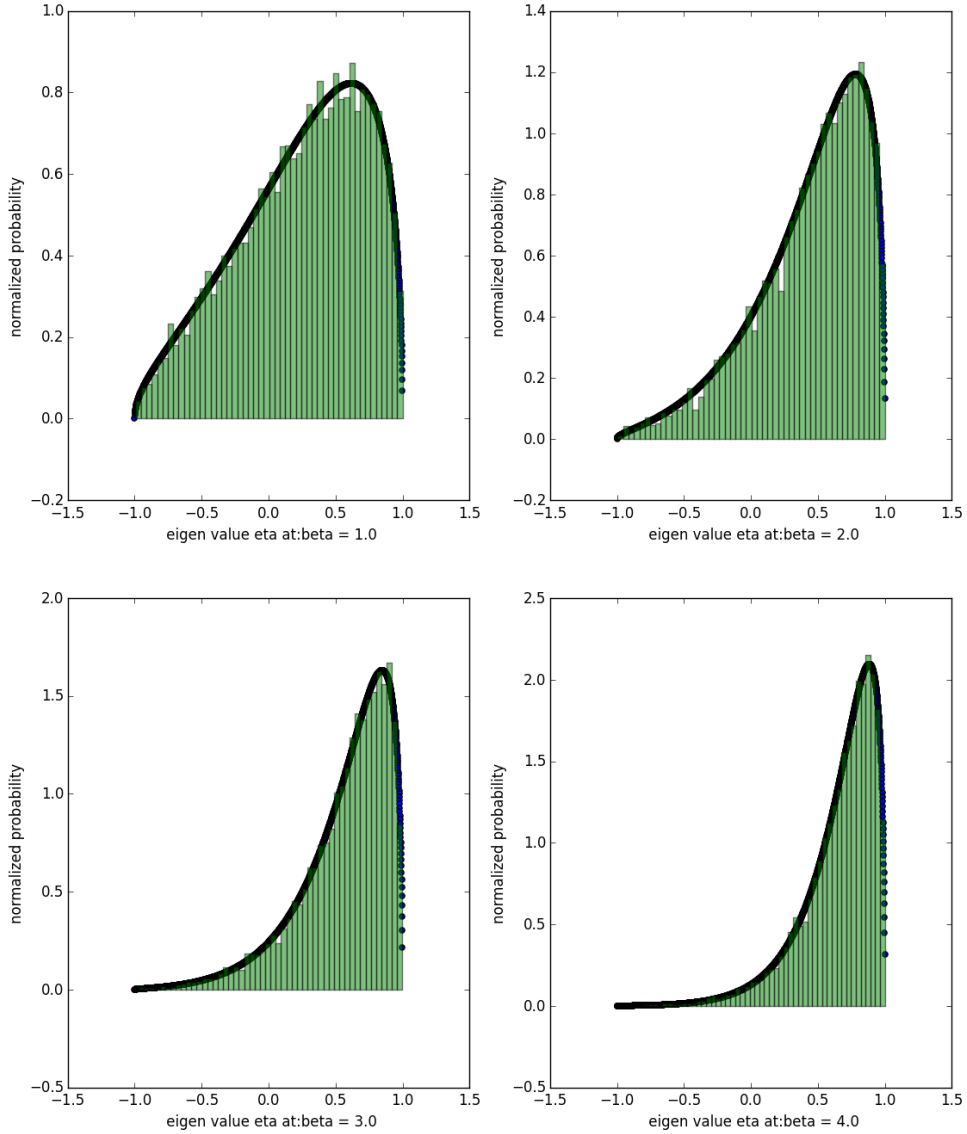


Figure 5: Histograms to represent normalized distribution of a_0 in SU(2) matrices generated by Pendelton's method. Solid lines are normalized distribution function:$p(a_0)da_0 = C\sqrt{(1-a_0^2)}e^{(\beta k a_0)}da_0$.

```
class check_Pendelton_distribution(object):

       def __init__ (self,beta):
                self.beta = beta

       def funct(self,t):
                 ft = (math.sqrt(1 - (t)**2))*(math.exp(self.beta*(t)))
                  return ft

       def nrn(self):
              dx = 0.002
              I = [0.0  for k  in  range(-500,501)]
              x = [k*dx  for k  in  range(-500,501)]
               for k  in  range(-500,500):
                  I[k] = self.funct(x[k])*dx
              II = sum(I)
               return II

       def normalized_funct(self,t):
            nfunct = self.funct(t)/self.nrn()
             return nfunct

       def plot_function(self):
                  x = np.arange(-1.0,1.0,0.001)
                  ct = len(x)
                  y = []
                   for k  in  range(ct):
                      tt = self.normalized_funct(x[k])
                      y.append(tt)
                   return x,y

       def collect_x(self,LN):
                  ll = 0
                  x = []
                   while ll < LN+1:
                      a0 = self.pendelton_generation()
                      x.append(a0)
                      ll= ll+1
                   return x

def subplot_plotter(LN,SN):
                  nt = 1
                   while nt < 4+1:
                      beta = 1.0*nt
                      xx,yy = check_Pendelton_distribution(beta).plot_function()
                      x = check_Pendelton_distribution(beta).collect_x(LN)
                      num_bins = SN
                      st = 'beta = '+  str(beta)
                      plt.figure(22)
                      plt.subplot(2,2,nt)
                      plt.scatter(xx,yy)
                      plt.xlabel(' eigen value eta at:'+ st)
                      plt.ylabel('normalized probability' )
                      plt.hist(x,num_bins, normed= 1.0, facecolor='green',alpha = 0.5)
                      nt = nt+1
subplot_plotter(10000,50)
```

5.3 M. Crutz's Method

In Crutz's method, one can start with the same distribution function for parameter a_0;

$$dp(a_0) = C\sqrt{(1-a_0^2)}e^{(\beta k a_0)}da_0 \tag{50}$$

First one need to make this function bounded between interval $[0, 1]$. To do this, the maxima of this function is determined and whole function is divided by that constant value. Function *optimize* finds the maximum of the function with the help of function *find_max*. Function *find_max* find the maximum value of a function for a given interval on x-axis. Function *optimize* determines the interval where function is maximum and then again divide that interval into sub intervals and finds the subinterval where function is maximum by implementing function *find_max*. One can repeat this process to get more precise value of maxima of a function. Function *nfn* provides a function which is bounded between interval [0,1].

```
def funct(self,a0):
        return (math.sqrt(1.0 - (a0)**2.0))*(math.exp(self.alpha*self.dtm*a0))


def find_max(self,ptn,fpt,dx):
       dt = dx/ float(ptn)
       x = [0.0  for k  in  range(ptn)]
       fx = [0.0  for k  in  range(ptn)]
       x = [(fpt + dt*k)  for k  in  range(ptn)
       for k  in  range(ptn):
           fx[k] = [self.funct(x[k])]
       fmax = max(fx)
       mx = (fpt + (fx.index(fmax))*dt)
       xo = mx - dt
       return xo,fmax


def optimize(self,slice):
       ftp = -1.0
       dx= [0.0  for k  in  range (slice)]
        for k  in  range(slice):
           dx[k] = 2.0/ float(10**(k))
        for k  in  range(slice):
              ftp,fmax = self.find_max(20,ftp,dx[k])
        return fmax
```

```
def nfn(self,a0):
       den = self.optimize(10)
       ft = self.funct(a0)/den[0]
        return ft
```

Once the bounded function *nfn* is obtained, one can generate random point a_0 between interval [-1,1] with a weighting propertional to that bounded function. To do this , first a random number 'r_0' between interval [0,1] is generated and used to get value of a_0 in interval [-1,1]. A second random number r_1 in interval [0,1] is generated and compaired to the value of bounded function *nfn*. One accepts value of a_0 if second random number r_1 is less then functional value of *nfn* at point a_0.

```
def crutzgnr(self):
      count = 0
      failed = 0
      while (count <1):
           r = [random.uniform(0.0,1.0),\
                random.uniform(0.0,1.0)]

           a0 = (2*r[0] - 1)
           ff = self.nfn(a0)

            b0 = r[1]
            if (b0 < ff):
                 count = count +1
                 XX = self.su2(a0)
            return XX
            else:
            count = 0
            failed = failed+1
```

Once a_0 is generated, one need to find a randum point on the surface of a four dimensional sphere with the value of a_0 already known. Such that

$$a_0^2 + a_1^2 + a_2^2 + a_3^2 = 1 \tag{51}$$

i.e.

$$a_1^2 + a_2^2 + a_3^2 = 1 - a_0^2 = a^2 \tag{52}$$

This is equivalent to a three dimensional sphere. One genereates a random point on surface of three dimentional sphere by following steps:

Step I - Get a random no $t_1 \in [-1, 1]$ and get an angle $\theta = \cos^{-1}(t)$.

Step II - Get a randum no $t_2 \in [0, 1]$ and get an angle $\phi = 2\pi t_2$.

Step III - Define:

$$\begin{aligned} a_1 &= a \sin\theta \cos\phi \\ a_2 &= a \sin\theta \sin\phi \\ a_3 &= a \cos\theta \end{aligned} \tag{53}$$

```
def su2(self,a0):

                aa = math.sqrt(1 - (a0**2))

                t = [random.uniform(0.0,1.0),\
                    random.uniform(0.0,1.0)]

                theta = math.acos(2.0*t[0]-1)
                xi = 2.0*math.pi*(t[1])

                a =  [ a0,\
                    aa*math.sin(theta)*math.cos(xi),\
                    aa*math.sin(theta)*math.sin(xi),\
                    aa*math.cos(theta)]

                XX  = np.matrix([[complex(a[0],a[3]),complex(a[2],a[1])] \
                        ,[complex(-a[2],a[1]),complex(a[0],-a[3])]])


                 return XX
```

5.4 Checking Crutz's Method

Similar to the Pendelton's method, One can also chek the distribution of variable a_0 in case of Crutz's method. Eventhough the variable a_0 is generated by two different methods, the distribution is only dependent on value of β. Which is clear from the plot of histograms from two methods for same value of β

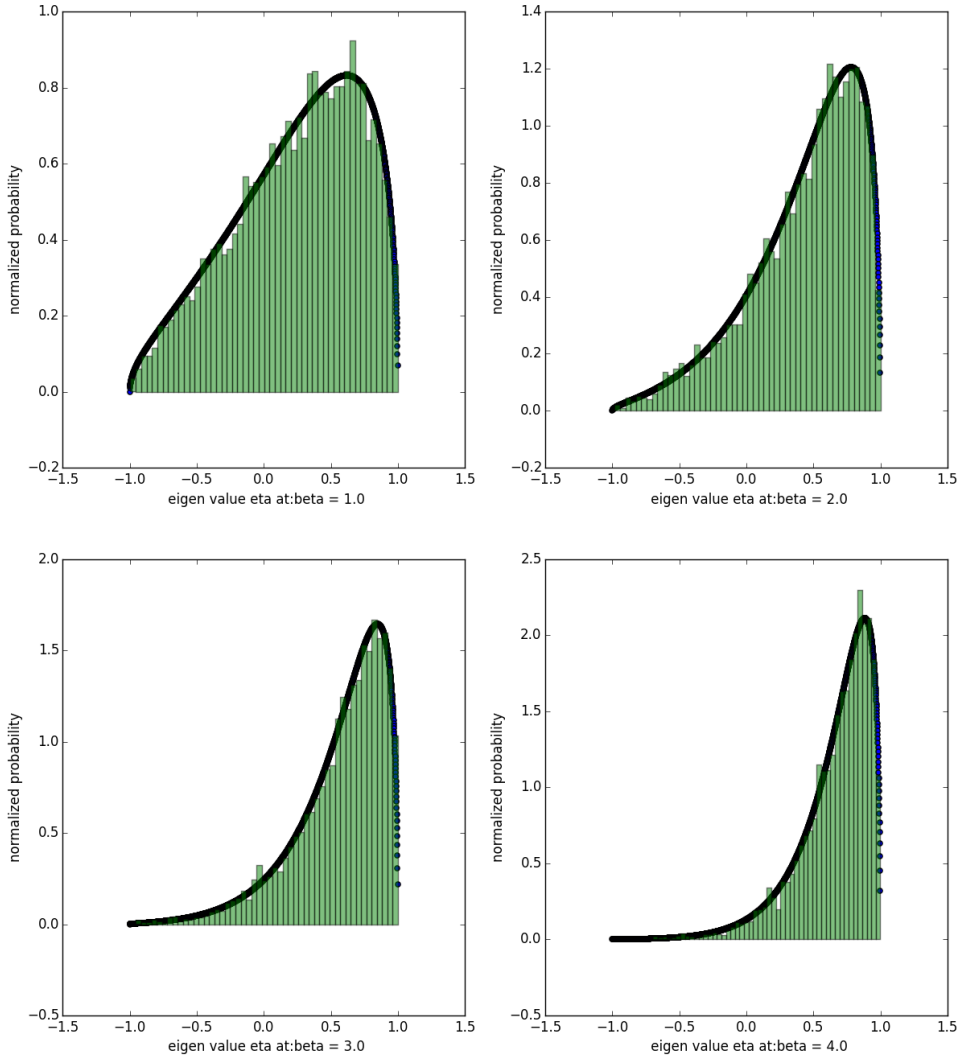


Figure 6: Histograms to represent normalized distribution of a_0 in SU(2) matrices generated by Crutz's method. Solid lines are normalized distribution function:$p(a_0)da_0 = C\sqrt{(1-a_0^2)}e^{(\beta k a_0)}da_0$.

```
class Normalized_Crutz_Distribution(object):

        def __init__ (self,beta):
                self.beta = beta

        def funct(self,t):
                 ft = (math.sqrt(1 - (t)**2))*(math.exp(self.beta*(t)))
                  return ft

        def nrn(self,npt):
               dx = 2.0/(2.0* float(npt))
               I = [0.0  for k  in  range(-npt,npt+1)]
               x = [k*dx  for k  in  range(-npt,npt+1)]
                for k  in  range(-npt,npt):
                   I[k] = self.funct(x[k])*dx
               II = sum(I)
                return II

        def normalized_funct(self,t):
             nfunct = self.funct(t)/self.nrn(50)
              return nfunct


        def plot_function(self):
                   x = np.arange(-1.0,1.0,0.001)
                   ct = len(x)
                   y = []
                    for k  in  range(ct):
                       tt = self.normalized_funct(x[k])
                       y.append(tt)
                    return x,y



class Bounded_Crutz_Distribution(object):

        def __init__ (self,beta):
                self.beta = beta

        def find_max(self,ptn,fpt,dx):
                   dt = dx/ float(ptn)
                   x = [0.0  for k  in  range(ptn)]
                   fx = [0.0  for k  in  range(ptn)]
                   x = [(fpt + dt*k)  for k  in  range(ptn)]
                   NCD = Normalized_Crutz_Distribution(self.beta)
                    for k  in  range(ptn):
                       fx[k] = [NCD.normalized_funct(x[k])]
                   fmax = max(fx)
                   mx = (fpt + (fx.index(fmax))*dt)
                   xo = mx - dt
                    return xo,fmax
```

```
        def optimize(self,slice):
                    ftp = -1.0
                    dx= [0.0  for k  in  range (slice)]
                     for k  in  range(slice):
                        dx[k] = 2.0/ float(10**(k))
                     for k  in  range(slice):
                           ftp,fmax = self.find_max(20,ftp,dx[k])
                     return fmax



        def bounded_funct(self,a0):
                    den = self.optimize(10)
                    NCD = Normalized_Crutz_Distribution(self.beta)
                    ft = NCD.normalized_funct(a0)/den[0]
                     return ft



class Check_Crutz_Distribution(object):

        def __init__ (self,beta):
                  self.beta = beta

        def crutz_generation(self):
                   count = 0
                   failed = 0
                   BCD = Bounded_Crutz_Distribution(self.beta)
                    while (count <1):
                       r = [random.uniform(0.0,1.0),\
                           random.uniform(0.0,1.0)]

                       a0 = (2*r[0] - 1)
                       ff = BCD.bounded_funct(a0)
                       b0 = r[1]
                        if (b0 < ff):
                         count = count +1
                          return a0
                        else:
                        count = 0
                        failed = failed+1


        def collect_x(self,LN):
                    ll = 0
                    x = []
                     while ll < LN+1:
                        a0 = self.crutz_generation()
                        x.append(a0)
                        ll= ll+1
                     return x



def subplot_plotter(LN,SN):
                    nt = 1
```

```
                    while nt < 4+1:
                       beta = 1.0*nt
                       NCD = Normalized_Crutz_Distribution(beta)
                       CCD = Check_Crutz_Distribution(beta)
                       xx,yy = NCD.plot_function()
                       x = CCD.collect_x(LN)
                       num_bins = SN
                       st = 'beta = '+  str(beta)
                       plt.figure(22)
                       plt.subplot(2,2,nt)
                       plt.scatter(xx,yy)
                       plt.xlabel(' eigen value eta at:'+ st)
                       plt.ylabel('normalized probability' )
                       plt.hist(x,num_bins, normed= 1.0, facecolor='green',alpha = 0.5)
                       nt = nt+1


subplot_plotter(5000,50)
```

5.5 Acceptance Ratio

Not all randomly picked a_0 are accepted to construct a random SU(2) matrix. One can display the acceptance ratio in each of the methods. The acceptance ratio is dependent on the value of parameter β which is shown in the table below.

Table 1: Acceptance ratios

β	Crutz's Method	Pendelton's Method
16.0	0.06	0.97
5.0	0.19	0.92
2.0	0.42	0.75
1.0	0.60	0.52
0.5	0.73	0.28
0.25	0.77	0.12
0.10	0.78	0.03

Depending on these acceptance ratios one can set the *flip* parameter equal to 1.0. Which means when value of β is higher then 1.0, the Pendelton's method will be used otherwise Crutz's method will be selected.

```
                    if beta > flip:#flip = 1.0
```

```
                    XX = Pendel_Crutz(beta,det).pendlgnr()
                  else:
                    XX = Pendl_Crutz(beta,det).crutzgnr()
```

5.6 Codes for Class *Pendl-Crutz*

```
class Pendl-Crutz(object):


          def __init__(self,alpha,det):
                 self.alpha = alpha
                 self.det = k


          def su2(self,a0):
                 aa = math.sqrt(1 - (a0**2))
                 t = [random.uniform(0.0,1.0),\
                     random.uniform(0.0,1.0)]
                 theta = math.acos(2.0*t[0]-1)
                 xi = 2.0*math.pi*(t[1])
                 a =  [ a0,\
                 aa*math.sin(theta)*math.cos(xi),\
                 aa*math.sin(theta)*math.sin(xi),\
                 aa*math.cos(theta)]
                 XX  = np.matrix([[complex(a[0],a[3]),complex(a[2],a[1])] \
                        ,[complex(-a[2],a[1]),complex(a[0],-a[3])]])
                  return XX


          def pendlgnr(self):
                   count = 0
                   while (count < 1):
                     r = [ random.uniform(0.0,1.0),\
                     random.uniform(0.0,1.0),\
                     random.uniform(0.0,1.0),\
                     random.uniform(0.0,1.0)]
                     x  = [-(math.log(r[0])/(self.det*self.alpha)),\
                          -(math.log(r[1])/(self.det*self.alpha))]
                     C = (math.cos(2.0*math.pi*r[2]))**2
                     A = x[0]*C
                     delta = x[1]+A
                      if (r[3]**2) < (1-(0.5*delta)):
                        a0 = (1- delta)
                        count = count+1
                        XX = self.su2(a0)
                         return XX
                      else:
                        count = 0


          def funct(self,a0):
```

```
            return (math.sqrt(1.0 - (a0)**2.0))*(math.exp(self.alpha*self.k*a0))


    def find_max(self,ptn,fpt,dx):
           dt = dx/ float(ptn)
           x = [0.0  for k  in  range(ptn)]
           fx = [0.0  for k  in  range(ptn)]

           x = [(fpt + dt*k)  for k  in  range(ptn)]

            for k  in  range(ptn):
              fx[k] = [self.funct(x[k])]
           fmax = max(fx)
           mx = (fpt + (fx.index(fmax))*dt)
           xo = mx - dt
            return xo,fmax


    def optimize(self,slice):
           ftp = -1.0
           dx= [0.0  for k  in  range (slice)]
            for k  in  range(slice):
              dx[k] = 2.0/ float(10**(k))
            for k  in  range(slice):
                  ftp,fmax = self.find_max(20,ftp,dx[k])
            return fmax


    def nfn(self,a0):
           den = self.optimize(10)
           ft = self.funct(a0)/den[0]
            return ft


    def  crutzgnr(self):
          count = 0
          failed = 0
           while (count <1):
              r = [random.uniform(0.0,1.0),\
                  random.uniform(0.0,1.0)]
              a0 = (2*r[0] - 1)
              ff = self.nfn(a0)
              b0 = r[1]
               if (b0 < ff):
                count = count +1
                XX = self.su2(a0)
                 return XX
               else:
               count = 0
               failed = failed+1
```

6 Updating Links

To update links in a configuration one by one, one can call functions from class *Update*. The function *Thermalization*, function in class *Calculate-average* etc uses these steps to generate average values of *plaquette, wilson loops etc* in each cycle of simulation.

This class includes the functions which receive old link variables and returns newly generated link. It includes functions *staple* and *link* which works for specific link $LK(r,t,s)$; where value of r determines whether it is horizontal or vertical link at coordinate (t,s). Function *staple* constructs a list D which temporarily stores links as the elements for associated staple.

```
            if r ==0:
              Q = [1,0,1,1,0,1]
            elif r==1:
              Q = [0,1,0,0,1,0]
          #LK = np.matrix(self.UU[r][t][s])
           D = [ np.matrix(self.UU[Q[0]][(s+1)%self.L][(t-1) +
                (self.L*fun(t))]).getH(),\
                 np.matrix(self.UU[Q[1]][(t-1) + (self.L*fun(t))][s]).getH(),\
                 np.matrix(self.UU[Q[2]][s][(t-1) + (self.L*fun(t))]),\
                 np.matrix(self.UU[Q[3]][(s+1)%self.L][t]),\
                 np.matrix(self.UU[Q[4]][(t+1)%self.L][s]).getH(),\
                 np.matrix(self.UU[Q[5]][s][t]).getH()]
```

Boundary conditions:

$$\begin{aligned} U(r,L,t) &= U(r,0,t) \forall 0 \leq t < L \\ U(r,s,L) &= U(r,s,0) \forall 0 \leq s < L \end{aligned} \tag{54}$$

for a 2D lattice is imposed by using a modulo function *mod(n,m)* as '$n\%m$' and a jump function *fun*.

```
def fun(s):
    if s ==0:
     fn = 1
    else:
     fn = 0
    return fn
```

Those elements of D are used to calculate the staple 'W' and to find it's

determinant which is used to construct a unitary matrix 'A' by dividing W by it's determinant.

```
W = np.dot(D[0],np.dot(D[1],D[2])) \
  + np.dot(D[3],np.dot(D[4],D[5]))
dtm = math.sqrt(linalg.dtm(W).real)
A = W/(dtm)
```

Fortunately, the sum of two SU(2) matrix is a U(2) matrix. We make it SU(2) just by dividing each elements by it's determinant.

The function *link* returns the newly generated link. It uses two methods for Heat bath viz: Crutz and Pendelton. There is a *flip* parameter which sets the flip boundary for those two methods. For example: if $flip = 1.0$, then for $\beta < 1.0$ it uses crut'z method otherwise it uses pendelton's method.

```
def link(self,r,t,s,beta,flip):
dt,A = self.staple(r,t,s)
AD = A.getH()
 if beta > flip:
    XX = Pendel_Crutz(beta,dtm).pendlgnr()
 else:
    XX = Pendel_Crutz(beta,dtm).crutzgnr()
NU = np.dot(XX,AD)
self.U[r][t][s] = NU
 return self.U
```

Complex conjugate of a matrix A is multiplied with newly generated matrix XX to get a new link NU. Function *link* returns this as a new link .

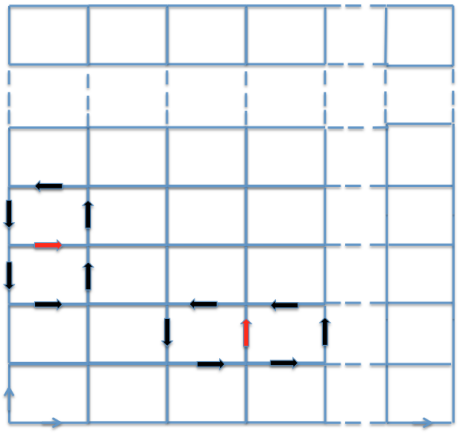


Figure 7: **Mostly Mistaken Point:** *The direction of links while calculating staple is very specific. Red arrows at the centre are links going to be updated, while the direction shown by black arrows are the directions of links to be included in staple. Horizontal or verticle link is recognized by the value of parameter 'r' of function 'staple' of class 'Calculate'.*

6.1 Codes for Class *Update*

```
class Update(object):

        def __init__(self,UU,L,N):
                self.UU = UU
                self.L = L
                self.N = N


        def staple(self,r,t,s):
                if r ==0:
                   Q = [1,0,1,1,0,1]
                elif r==1:
                   Q = [0,1,0,0,1,0]
               #LK = np.matrix(self.UU[r][t][s])
               D = [ np.matrix(self.UU[Q[0]][(s+1)%self.L][(t-1) +
                    (self.L*fun(t))]).getH(),\
                     np.matrix(self.UU[Q[1]][(t-1) + (self.L*fun(t))][s]).getH(),\
                     np.matrix(self.UU[Q[2]][s][(t-1) + (self.L*fun(t))]),\
                     np.matrix(self.UU[Q[3]][(s+1)%self.L][t]),\
                     np.matrix(self.UU[Q[4]][(t+1)%self.L][s]).getH(),\
                     np.matrix(self.UU[Q[5]][s][t]).getH()]


               W = np.dot(D[0],np.dot(D[1],D[2])) \
                + np.dot(D[3],np.dot(D[4],D[5]))
               det = math.sqrt(linalg.det(W).real)
               A = W/(det)
                return det,A


        def link(self,r,t,s,beta,flip):
               det,A = self.staple(r,t,s)
               AD = A.getH()
```

```
                if beta > flip:
                   XX = Pendel_Crutz(beta,det).pendlgnr()
                else:
                   XX = Pendl_Crutz(beta,det).crutzgnr()
               NU = np.dot(XX,AD)
               self.U[r][t][s] = NU
                return self.U
```

7 Thermalization

Function *thermalization* calls the function *cold_start* or *hot_start* from class *Start* to get starting configuration. It further calls the function *link* from class *Update* and updates all links one by one in one complete cycle. It is always important to make sure that once a perticular link is updated, the status of whole configuration should be renewed. To do this one should instantly replace old link by new one before going to update next link in a sequence.

There is a *flip* parameter in argument of function *link* which sets the boundary for crutz or pendeltons method. The acceptance ratio can be calculated simultaneously along with average plaquette value.

```
def thermalization(l,titr,alpha,flip):

          ll = 1
          U = Start(l).cold_start()
           while (ll < titr+1):
                 totfailed = 0
                  for s  in  range(l):
                      for t  in  range(l):
                          for r  in  range(2):
                               failed, U = Update(U,l).link(r,s,t,alpha,flip)
                 avp = Calculate(U,l,N).avplqt()
                 plt.figure(100)
                 plt.scatter(0.0,0.0)
                 plt.scatter(0.0,1,0)
                 plt.scatter(ll,avp)
                 plt.show()
                 ll = ll+1

           return
```

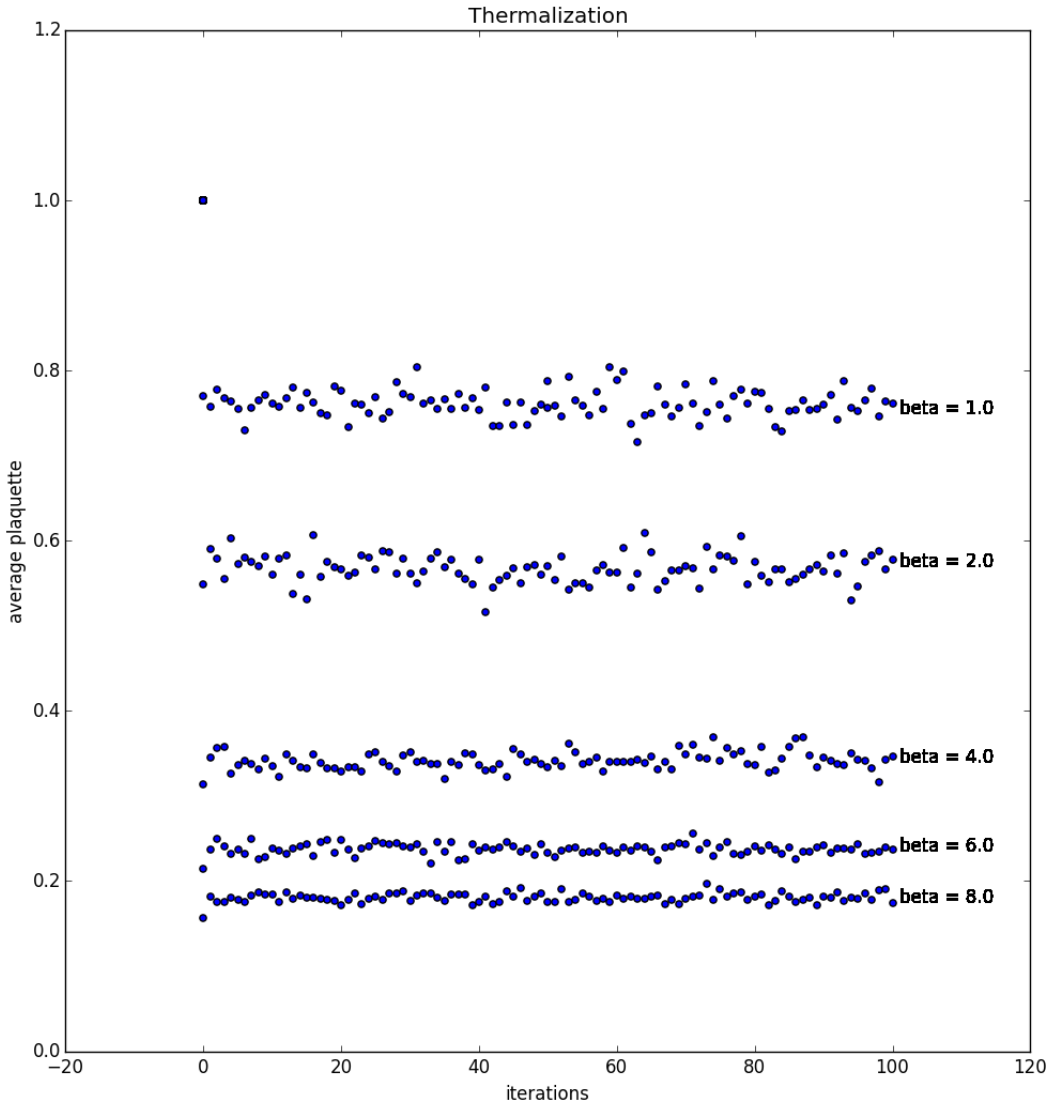


Figure 8: Plot of average plaquette $P =< 1 - (\frac{1}{2})Tr(U_p) >$ w.r.t. iterations at different value of β. Where U_p is a plaquette variable.

8 Calculation of Average Values

To calculate average after the configuration is properly thermalized, there is a class *Calculate* with functions *plqt*, *avplqt*, *wloop11* etc.

List D in function *plqt* stores 4 links associated for particular plaquette at coordinate (s, t).

```
def plqt(self,s,t):
                D  = [ np.matrix(self.UU[0][s][t]),\
                       np.matrix(self.UU[1][(t+1)%self.L][s]),\
                       np.matrix(self.UU[0][(s+1)%self.L][t]).getH(),\
                       np.matrix(self.UU[1][t][s]).getH()]
                 return D
```

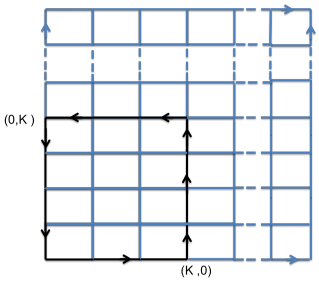
To calculate average plaquette in each cycle of simulation, each plaquette is determined with the help of *plqt* and average is calculated in function *avplqt*.

```
def avplqt(self):
                sum_trp = 0.0
                 for s  in  range(self.L):
                    for t  in  range(self.L):
                       D = self.plqt(s,t)
                       UP = np.dot(D[0],np.dot(D[1],np.dot(D[2],D[3])))
                       trup = (1.0 - ((1.0/ float(2))*np.trace(UP).real))
                       sum_trp = sum_trp + (trup/ float(self.L*self.L))
                 return sum_trp
```

Single plaquette wilson loop at coordinate (s,t) can be calculated by using function *wloop11*.

```
def wloop11(self,s,t):
                D = self.plqt(s,t)
                UP = np.dot(D[0],np.dot(D[1],np.dot(D[2],D[3])))
                wtr =  (1.0/ float(2))*np.trace(UP).real
                 return wtr
```

For larger Wilson loop one can use function *wilsonlp*.




```
def wilsonlp(self,K):
                I = np.matrix(np.identity(2))
                WKK = np.matrix(np.identity(2))
                PD = [[I  for k  in  range(K)] for p  in  range(4)]
                DD = [I  for k  in  range(4)]
                 for s  in  range(K):
                    PD[0][s] = np.matrix(self.UU[0][0][s])
                 for s  in  range(K):
                    PD[1][s] = np.matrix(self.UU[1][K][s])
                 for s  in  range(K):
                    t = K-s-1
                    PD[2][s] = np.matrix(self.UU[0][K][t]).getH()
                 for s  in  range(K):
                    x = K-s-1
                    PD[3][s] = np.matrix(self.UU[1][0][x]).getH()
                 for r  in  range(4):
                     for k  in  range(K):
                        DD[r] = np.dot(DD[r],PD[r][k])
                WKK = np.dot(DD[0],np.dot(DD[1],np.dot(DD[2],DD[3])))
                wilp = (1.0/ float(self.N))*np.trace(WKK).real
                 return wilp
```

8.1 Codes for Class *Calculate*

```
class Calculate(object):
          def __init__(self,UU,L):
                 self.UU = UU
                 self.L = L


          def plqt(self,s,t):
                 D  = [ np.matrix(self.UU[0][s][t]),\
                       np.matrix(self.UU[1][(t+1)%self.L][s]),\
                       np.matrix(self.UU[0][(s+1)%self.L][t]).getH(),\
                       np.matrix(self.UU[1][t][s]).getH()]
                  return D
```

```
    def avplqt(self):
            sum_trp = 0.0
             for s  in  range(self.L):
                 for t  in  range(self.L):
                    D = self.plqt(s,t)
                    UP = np.dot(D[0],np.dot(D[1],np.dot(D[2],D[3])))
                    trup = (1.0 - ((1.0/ float(self.N))*np.trace(UP).real))
                    sum_trp = sum_trp + (trup/ float(self.L*self.L))
             return sum_trp


    def wloop11(self,s,t):
            D = self.plqt(s,t)
            UP = np.dot(D[0],np.dot(D[1],np.dot(D[2],D[3])))
            wtr = (1.0/ float(2))*np.trace(UP).real
             return wtr

    def wilsonlp(self,K):
            I = np.matrix(np.identity(2))
            WKK = np.matrix(np.identity(2))
            PD = [[I  for k  in  range(K)] for p  in  range(4)]
            DD = [I  for k  in  range(4)]
             for s  in  range(K):
                PD[0][s] = np.matrix(self.UU[0][0][s])
             for s  in  range(K):
                PD[1][s] = np.matrix(self.UU[1][K][s])
             for s  in  range(K):
                t = K-s-1
                PD[2][s] = np.matrix(self.UU[0][K][t]).getH()
             for s  in  range(K):
                x = K-s-1
                PD[3][s] = np.matrix(self.UU[1][0][x]).getH()
             for r  in  range(4):
                 for k  in  range(K):
                    DD[r] = np.dot(DD[r],PD[r][k])
            WKK = np.dot(DD[0],np.dot(DD[1],np.dot(DD[2],DD[3])))
            wilp = (1.0/ float(2))*np.trace(WKK).real
             return wilp
```

9 Plotting with Errorbar

There is a function *export_wstor* which , after sufficient thermalization, calls the function *wloop11* from class *Calculate* and stores the value of single plaquette Wilson loop in every cycle of simulation in the list *wstor*.

```
def Export_Wstor(l,ct,titr,sitr,alpha,flip):
       # titr = total no of iterations
       # sitr = start counting iterations for average value
       citr = titr-sitr
```

```
        wstor = [0.0  for k  in  range (citr)]
        ll = 0
        U = Start(l).cold_start()
         while (ll < titr+1):
               print ct,ll
                for s  in  range(l):
                    for t  in  range(l):
                        for r  in  range(2):
                          U = Update(U,l).link(r,s,t,beta,flip)
                if ll > sitr:
                      wtr = Calculate(U,l).wloop11(15,15)
                      rt = ll- sitr-1
                      wstor[rt] = wtr
               ll = ll+1
         return wstor
```

To get a plot of average value with error bar, one need to find mean value of data along with standard deviation. There is a function *Mean_error* which calculates mean and error from a list of data. The list *wstor* is supplied to this function to get required mean and error from it.

```
def Mean_Error(storw):
        nt = len(storw)
        ver = [0.0  for k  in  range(nt)]
        mn = 0.0
         for k  in  range (nt):
            mn = mn+storw[k]/ float(nt)
         for l  in  range (nt):
            ver[l] = (storw[l]-mn)**2
        er = math.sqrt(sum(ver)/nt**2)
         return mn, er
```

One may need to get the mean value and error of obtained data at different value of β to get the plot of wilson loop versus β.

```
def Errorbar(l,titr,tdt,flip):
        # tdt = total no of data points(value of alpha) taken for plot
        plot_dir = "/Users/dibakarsigdel/Dropbox/Plots/"
        data_dir = "/Users/dibakarsigdel/Dropbox/Data/"
        ct =1
        ax = [0.0  for k  in  range (tdt)]
        ay = [0.0  for k  in  range (tdt)]
        y_error = [0.0  for k  in  range (tdt)]
        st =  str(2)
         while (ct < tdt+1):
            beta = ct*1.0
            ax[ct-1] = beta
            wstor = export_wstor(l,ct,titr,beta,flip)
            ay[ct-1],y_error[ct-1] = Mean_Error(wstor)
            plt.figure(102)
```

```
            plt.xlabel('lambda')
            plt.ylabel('W11')
            plt.grid()
            plt.errorbar(ax,ay, yerr = y_error, fmt='8')
            plt.savefig(plot_dir + 'plotsu'+st+'.png')
            plt.show()
            wnr('su'+st+'.dat',[ax,ay,y_error]).writer()
            wnr(data_dir +'su'+st+'.dat',[ax,ay,y_error]).writer()
            ct = ct+1
        return ax,ay,y_error
```

9.1 Calculation-I: Average Plaquette w.r.t. β

One can measure the average value of plaquette as $P =< 1-(\frac{1}{2})Tr(U_p) >$.

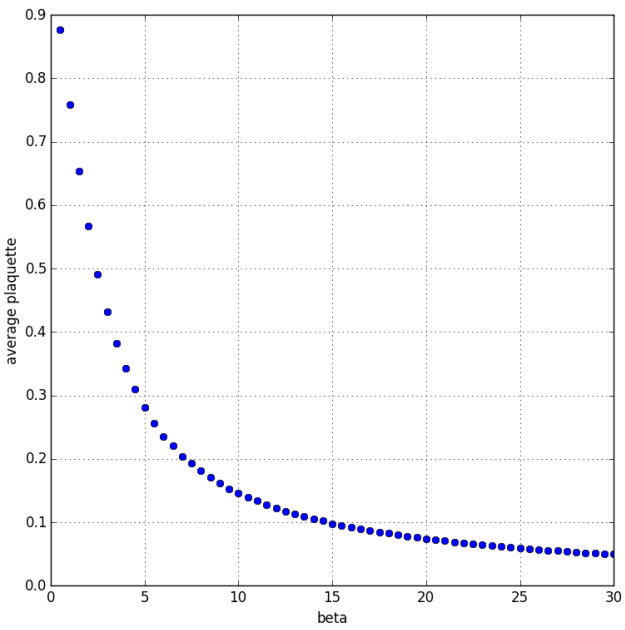


Figure 9: Plot of average plaquette as $P =< 1-(\frac{1}{2})Tr(U_p) >$ w.r.t. β for SU(2). Where U_p is a plaquette operator.

9.2 Calculation-II: Single Plaquette Wilson Loop

For square lattice of length L, the expectation value of K_1 by K_2 wilson loop is given by:

$$< \frac{\chi_r(W(K_1K_2))}{d_r} >= \frac{\sum_{r'} \left[\frac{\tilde{\beta}_{r'}(\beta)}{d_{r'}\tilde{\beta}_0(\beta)}\right]^{(L_1L_2-K_1K_2)} \sum_{r''} \frac{n(r,r';r'')d_{r''}}{d_r d_{r'}} \left[\frac{\tilde{\beta}_{r''}(\beta)}{d_{r''}\tilde{\beta}_0(\beta)}\right]^{(K_1K_2)}}{\sum_{r'} \left[\frac{\tilde{\beta}_{r'}(\beta)}{d_{r'}\tilde{\beta}_0(\beta)}\right]^{L_2L_2}} \quad (55)$$

For Wilson action one can use:

$$\tilde{\beta}_{r'}(\beta) = I_{2j}(\beta) - I_{2j+2}(\beta) = \frac{2(2j+1)}{\beta} I_{2j+1}(\beta) \quad (56)$$

with

$$\tilde{\beta}_o = \frac{2I_1(\beta)}{\beta} \quad (57)$$

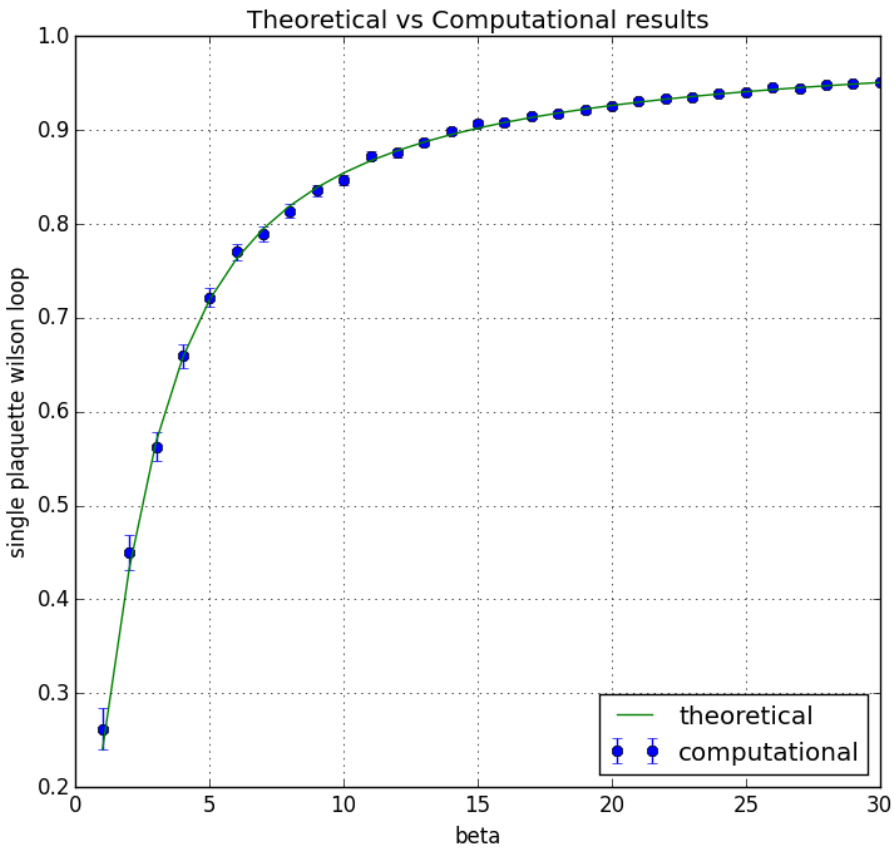


Figure 10: Plot of single plaquette Wilson loop w.r.t β for SU(2). Solid line represents the curve explained in equation (55).

In order to make a plot of single plaquette wilson loop by using above expression, there is a class *Wplot*. To calculate CG cofficients $n(r, r'; r'')$ we calculate the integral in function *cgcoff*.

$$n(r, r'; r'') = \frac{2}{\pi} \int \sin^2 \eta d\eta \frac{\sin((2r+1)\eta)}{\sin \eta} \frac{\sin((2r^{'}+1)\eta)}{\sin \eta} \frac{\sin((2r^{''}+1)\eta)}{\sin \eta} \quad (58)$$

with $r = \frac{1}{2}$ for fundamental representation of SU(2).

9.3 Codes for Class *Wplot*

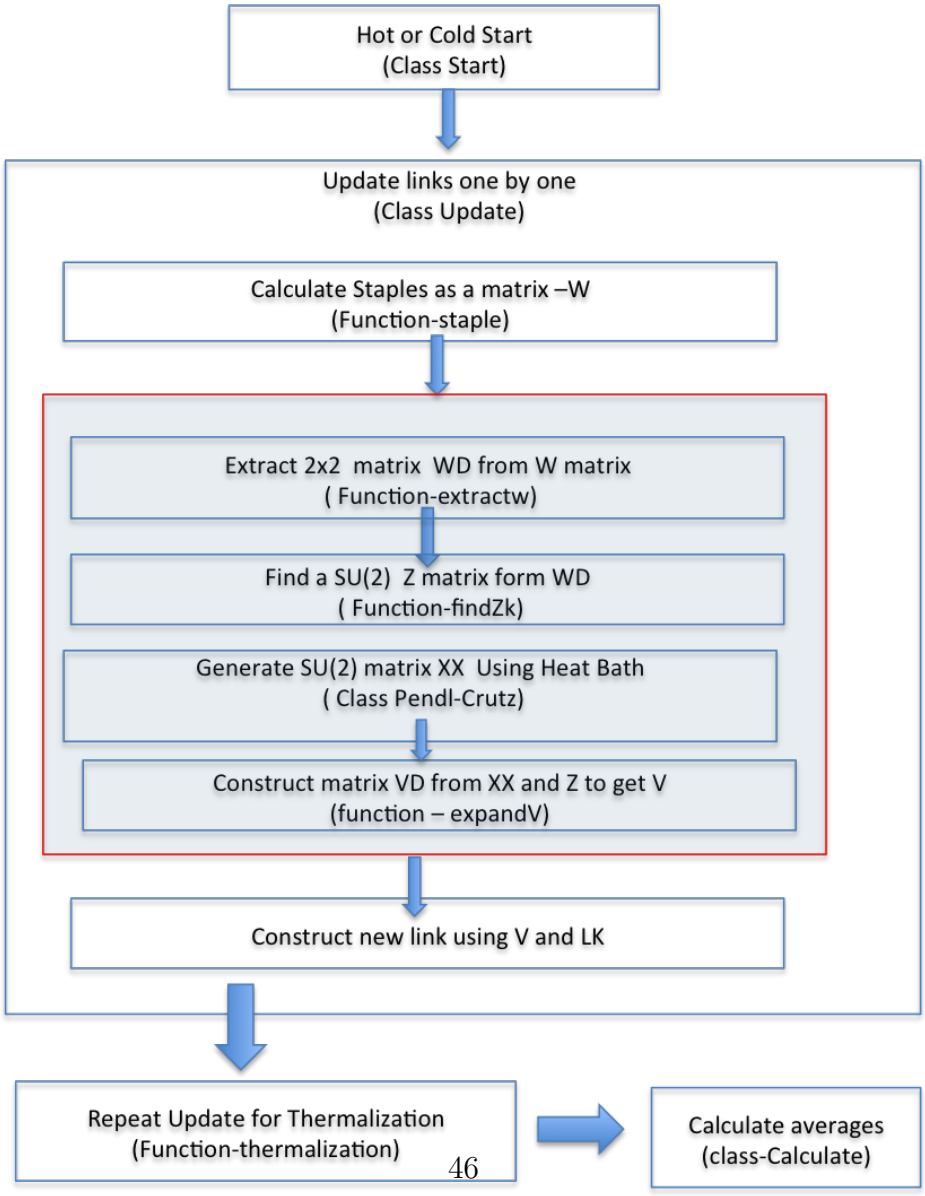
```
class Wplot(object):

  def __init__(self,L,K):
      self.L = L
      self.K = K


  def cgcoff(self):

          deta = math.pi/ float(100)
          II_sum = []
          II_int = []

          II_int = (2.0/math.pi)*[[[(math.sin(deta*(dx+1))**2) \
                *(math.sin((2)*deta*(dx+1))/math.sin(deta*(dx+1))) \
                 *(math.sin((ri+1)*deta*(dx+1))/math.sin(deta*(dx+1))) \
                    *(math.sin((rii+1)*deta*(dx+1))/math.sin(deta*(dx+1)))*deta \
                      for dx  in  range(100)]\
                       for rii  in  range(100)] \
                          for ri  in  range(100)]


          II_sum = [[sum(II_int[ri][rii]) \
                for rii  in  range(100)] \
                  for ri  in  range(100)]


           for ri  in  range(100):
              for rii  in  range(100):
                  if II_sum[ri][rii] > 0.5:
                       II_sum[ri][rii] = 1.0
                  else:
                       II_sum[ri][rii] = 0.0


           return II_sum


  def wclc(self,DD,alpha):

              xx = ((self.L*self.L)-(self.K*self.K))
              dr = [r+1  for r  in  range(200)]
              beta_r = [0.0  for r  in  range(200)]
              #from scipy.special import iv
              beta_r = [((r+1)*iv(r+1,alpha))/(iv(1,alpha))
                       for r  in  range(200)]

              factor = [0.0  for rii  in  range(200)]
              factor = [(beta_r[rii]/ float(dr[rii]))  for rii  in  range(200)]


              multiplier = [[0.0  for rii  in  range(100)] \
                           for ri  in  range(100)]
```

```
            multiplier = [[((DD[ri][rii])*dr[rii] \
                        *(1/ float(2.0*dr[ri]))*(factor[rii])**(self.K*self.K))\
                          for rii  in  range(100)] \
                             for ri  in  range(100)]



            sum_multiplier = [0.0  for ri  in  range(100)]
            sum_multiplier = [ sum(multiplier[ri]) \
                           for ri  in  range(100)]



            numerator = [0.0]
            numerator = [(((factor[ri])**xx) \
                        *sum_multiplier[ri]) \
                         for ri  in  range(100)]



            sum_numerator = sum(numerator)
            denominator = []
            denominator = [ (factor[ri])**(self.L*self.L)  for ri  in  range(100)]

            sum_denominator = sum(denominator)

            final_value = []
            final_value = (sum_numerator/sum_denominator)

             return final_value
```

Part III

Python Codes:SU(N),N>2

10 Algorithmic Steps for SU(N),N>2



11 Hot or Cold Start

For lattice configuration of SU(N) matrices, there is a class *Start* which has argumrnt $N > 2$. To get random matrix for Hot start, we use two functions *su2tosun* and *sun_gnr*. Function *SU2* in class *Start* generates the random SU(2) matrix whose elements are later pluged to specific position of a N by N matrix to construct a SU(N) by using function *su2tosun*.

```
def su2tosun(self,s,t):

                  SUN = CI(self.N)
                  SU2 = self.SU2()
                  SUN[s,s] = SU2[0,0]
                  SUN[s,t] = SU2[0,1]
                  SUN[t,s] = SU2[1,0]
                  SUN[t,t] = SU2[1,1]
                   return SUN
```

Function *sun_gnr* collects such partially filled SU(N) and multiplies to get a random SU(N) matrix.

```
          def sun_gnr(self):
                 SUNM = CI(self.N)
                 s = 1
                  while s < self.N:
                    t = s+1
                     while t < self.N+1:
                        SUN = self.su2tosun(s-1,t-1)
                        SUNM = np.dot(SUNM,SUN)
                        t = t+1
                    s = s+1
                    ZSUN = SUNM
                     return ZSUN
```

11.1 Codes for Class *Start*

```
class Start(object):
          def __init__(self,L,N):
                 self.L = L
                 self.N = N

          def cold_start(self):
              I = np.matrix(np.identity(self.N))
              UU = [[[I  for x  in  range(self.L)]
                           for y  in  range(self.L)]
                               for r  in  range(2)]
               return UU



          def SU2(self):
                ai = complex(0,1)
                r = [random.random(),random.random(),\
                     random.random()]
                xi = (math.pi*(2*r[0]-1))
                theta =0.5*(math.acos(2*r[1]-1))
                phi = (math.pi*(2*r[2]-1))
                a = [0.0  for l  in  range(2)]
                a = [math.cos(theta)*(cmath.exp(ai*phi)),\
                math.sin(theta)*(cmath.exp(ai*xi))]
                su2 = []
                su2 = np.matrix([[a[0],a[1]],\
                    [-a[1].conjugate(),a[0].conjugate()]])
                 return su2


          def su2tosun(self,s,t):

                 SUN = CI(self.N)
                 SU2 = self.SU2()
                 SUN[s,s] = SU2[0,0]
                 SUN[s,t] = SU2[0,1]
                 SUN[t,s] = SU2[1,0]
                 SUN[t,t] = SU2[1,1]
                  return SUN


          def sun_gnr(self):
                 SUNM = CI(self.N)
                 s = 1
                  while s < self.N:
                     t = s+1
                      while t < self.N+1:
                         SUN = self.su2tosun(s-1,t-1)
                         SUNM = np.dot(SUNM,SUN)
                         t = t+1
                     s = s+1
                     ZSUN = SUNM
                      return ZSUN
```

```
        def hot_start(self):
               I = np.matrix(np.identity(self.N))
               UU = [[[I  for x  in  range(self.L)] for y  in  range(self.L)] for z  in
                   range(2)]

                for i  in  range (2):
                    for j  in  range(self.L):
                        for k  in  range(self.L):
                           SUN = self.sun_gnr()
                           UU[i][j][k] = SUN
                return UU
```

12 Updating Links

Updating links in case of SU(N) matrix with $N > 2$ is very different compaired to SU(2). After finding Staple W in function *staple* in Class *Update*, it is multiplied with associated link LK to get matrix WW.

```
def staple(self,r,t,s):

                 if r ==0:
                    Q = [1,0,1,1,0,1]
                 elif r==1:
                    Q = [0,1,0,0,1,0]

                #LK = np.matrix(self.UU[r][t][s])
                D = [ (self.U[Q[0]][(s+1)% self.L][(t-1) + (self.L*fun(t))]).getH(),\
                     (self.U[Q[1]][(t-1) + (self.L*fun(t))][s]).getH(),\
                     (self.U[Q[2]][s][(t-1) + (self.L*fun(t))]),\
                     (self.U[Q[3]][(s+1)%self.L][t]),\
                     (self.U[Q[4]][(t+1)%self.L][s]).getH(),\
                     (self.U[Q[5]][s][t]).getH()]

                W = np.dot(D[0],np.dot(D[1],D[2])) \
                   + np.dot(D[3],np.dot(D[4],D[5]))

                LK = self.U[r][t][s]
                WW = np.dot(LK,W)
                 return WW
```

This WW matrix is used to extract WD matrix in function *extractw* in class *Selector*.

```
 def extractw(self,W,ct):
                WD = CI(2)
```

```
                P,Q = self.select()
                s = P[ct]
                t = Q[ct]
                WD[0,0] = W[s,s]
                WD[0,1] = W[s,t]
                WD[1,0] = W[t,s]
                WD[1,1] = W[t,t]
                 return WD
```

WD matrix is further used to obtain Z matrix.

```
def findZk(self,W,ct):
                Nn = Selector(self.N)
                WD = Nn.extractw(W,ct)
                X = WD[0,0] + (WD[1,1]).conjugate()
                Y = (WD[0,1]).conjugate() - WD[1,0]
                k = cmath.sqrt(abs(X)**2 + abs(Y)**2).real
                x = X/k
                y = Y/k
                Z = np.matrix([[(x).conjugate(), - (y).conjugate()] ,[y,x]])
                 return k,Z
```

A new SU(2) matrix XX is generated by using class *Crutz-Pendl* as described in case of SU(2). Using those two SU(2) matrices: XX and Z, a VD matrix is created.

```
def link(self,r,t,s,alpha,flip):
                LK =  self.U[r][t][s]
                W =  self.staple(r,t,s)
                Nn = Selector(self.N)

                V = [CI(self.N)  for lt  in  range(Nn.count())]
                ct = 0
                 while ct < Nn.count():
                   k,Z = self.findZk(W,ct)
                    if alpha > flip :
                      failed, XX = Pendl-Crutz(alpha,k).pendlgnr()
                    else:
                      failed, XX = Pendl-Crutz(alpha,k).crutzgnr()
                   VD = np.dot(XX,Z)
                   V[ct] = Nn.expandv(VD,ct)
                   W = np.dot(V[ct],W)# <----*** Internal update!
                   ct = ct+1

                NU = CI(self.N)

                 for q  in  range(Nn.count()):
                   NU = np.dot(NU,V[q])
                NNU = np.dot(NU,LK)

                self.U[r][t][s] = NNU
```

```
                return failed, self.U
```

This VD matrix is further pluged to a N by N matrix to get a new matrix V by using function *expandv*.

Mostly Mistaken Point: *In each of the internal loops for getting matrix V from VD, associated W matrix should be kept updated instanlty.*

```
def expandv(self,VD,ct):
                V = CI(self.N)
                P,Q = self.select()
                s = P[ct]
                t = Q[ct]
                V[s,s] = VD[0,0]
                V[s,t] = VD[0,1]
                V[t,s] = VD[1,0]
                V[t,t] = VD[1,1]
                 return V
```

This process is repeated untill all possible matrices are extracted. The number of total extractions and the element to be extracted is found by function *count* and *select* in class *Selector*. All V matrices are collected and multiplied together to get new link.

```
def count(self):
                ct = 0
                 for k  in  range(self.N):
                    ct = ct + k
                 return ct


          def select(self):
                ct = 0
                s = 0
                P = [0  for k  in  range(self.count())]
                Q = [0  for k  in  range(self.count())]
                 while s < self.N-1:
                    t = s+1
                     while t < self.N:
                        P[ct] = s
                        Q[ct] = t
                        ct = ct+1
                        t = t+1
                    s = s+1
                 return P,Q
```

12.1 Codes for Class *Update*

```
class Update(object):

          def __init__(self,U,L,N):
                  self.U = U
                  self.L = L
                  self.N = N



          def staple(self,r,t,s):

                   if r ==0:
                     Q = [1,0,1,1,0,1]
                   elif r==1:
                     Q = [0,1,0,0,1,0]

                  #LK = np.matrix(self.UU[r][t][s])
                  D = [ (self.U[Q[0]][(s+1)% self.L][(t-1) + (self.L*fun(t))]).getH(),\
                       (self.U[Q[1]][(t-1) + (self.L*fun(t))][s]).getH(),\
                       (self.U[Q[2]][s][(t-1) + (self.L*fun(t))]),\
                       (self.U[Q[3]][(s+1)%self.L][t]),\
                       (self.U[Q[4]][(t+1)%self.L][s]).getH(),\
                       (self.U[Q[5]][s][t]).getH()]

                  W = np.dot(D[0],np.dot(D[1],D[2])) \
                      + np.dot(D[3],np.dot(D[4],D[5]))

                  LK = self.U[r][t][s]
                  WW = np.dot(LK,W)
                   return WW


          def findZk(self,W,ct):
                 Nn = Selector(self.N)
                 WD = Nn.extractw(W,ct)
                 X = WD[0,0] + (WD[1,1]).conjugate()
                 Y = (WD[0,1]).conjugate() - WD[1,0]
                 k = cmath.sqrt(abs(X)**2 + abs(Y)**2).real
                 x = X/k
                 y = Y/k
                 Z = np.matrix([[(x).conjugate(), - (y).conjugate()] ,[y,x]])
                  return k,Z



          def link(self,r,t,s,alpha,flip):
                 LK =  self.U[r][t][s]
                 W =  self.staple(r,t,s)
                 Nn = Selector(self.N)

                 V = [CI(self.N)  for lt  in  range(Nn.count())]
                 ct = 0
                  while ct < Nn.count():
                     k,Z = self.findZk(W,ct)
```

```
                    if alpha > flip :
                      failed, XX = PendlCrutz(alpha,k).pendlgnr()
                    else:
                      failed, XX = PendlCrutz(alpha,k).crutzgnr()
                   VD = np.dot(XX,Z)
                   V[ct] = Nn.expandv(VD,ct)
                   W = np.dot(V[ct],W)
                   ct = ct+1

               NU = CI(self.N)

                for q  in  range(Nn.count()):
                  NU = np.dot(NU,V[q])
               NNU = np.dot(NU,LK)

               self.U[r][t][s] = NNU

                return failed, self.U
```

12.2 Codes for Class *Selector*

```
class Selector(object):

          def __init__(self,N):
                 self.N = N

          def count(self):
                 ct = 0
                  for k  in  range(self.N):
                     ct = ct + k
                  return ct


          def  select(self):
                 ct = 0
                 s = 0
                 P = [0  for k  in  range(self.count())]
                 Q = [0  for k  in  range(self.count())]
                  while s < self.N-1:
                     t = s+1
                      while t < self.N:
                         P[ct] = s
                         Q[ct] = t
                         ct = ct+1
                         t = t+1
                     s = s+1
                  return P,Q


          def extractw(self,W,ct):
                 WD = CI(2)
                 P,Q = self.select()
```

```
                s = P[ct]
                t = Q[ct]
                WD[0,0] = W[s,s]
                WD[0,1] = W[s,t]
                WD[1,0] = W[t,s]
                WD[1,1] = W[t,t]
                 return WD



        def expandv(self,VD,ct):
                V = CI(self.N)
                P,Q = self.select()
                s = P[ct]
                t = Q[ct]
                V[s,s] = VD[0,0]
                V[s,t] = VD[0,1]
                V[t,s] = VD[1,0]
                V[t,t] = VD[1,1]
                 return V

         return
```

13 Thermalization

Thermalization plot for different values of N greater then 2 in SU(N) can be obtained just by changing the value of argument 'N' of function *Thermalization*.

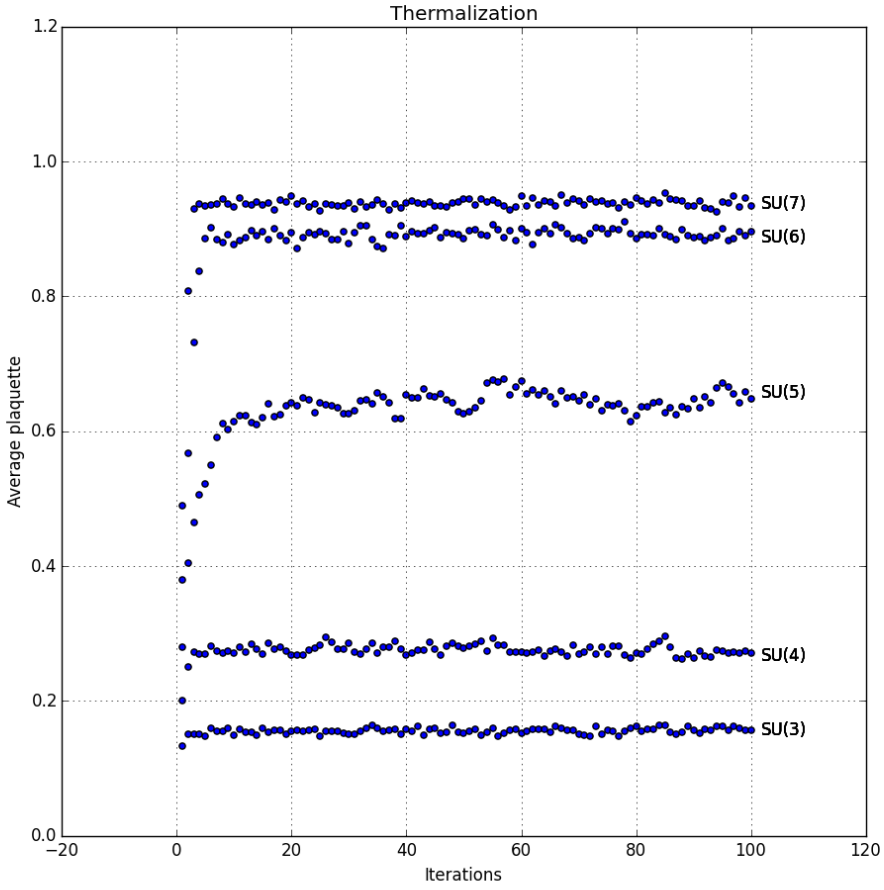


Figure 11: Plot of average plaquette $P =< 1-(\frac{1}{N})Tr(U_p) >$ w.r.t. iterations at different value of N = 3,4,5,6 and 7 for $\beta = 10.0, L = 20$. Where U_p is a plaquette variable.

```
def thermalization(N,l,titr,alpha,flip):
          ll = 1
          U = Start(l,N).cold_start()

           while (ll < titr+1):
                  for s  in  range(l):
                      for t  in  range(l):
                          for r  in  range(2):
                               failed, U = Update(U,l,N).link(r,s,t,alpha,flip)
```

```
                avp = Calculate(U,l,N).avplqt()
                print ll, avp
                plt.figure(100)
                plt.scatter(ll,avp)
                plt.show()
                ll = ll+1
            return
```

14 Calculation of Average Values

The main difference in code for class *calculate* for SU(2) to SU(N) is argument'N' which can be any integer greater then 2.

14.1 Codes for Class *Calculate*

```
class Calculate(object):
          def __init__(self,U,L,N):
                self.U = U
                self.L = L
                self.N = N

          def plqt(self,s,t):
                D  =  [(self.U[0][s][t]),\
                      (self.U[1][(t+1)%self.L][s]),\
                      (self.U[0][(s+1)%self.L][t]).getH(),\
                      (self.U[1][t][s]).getH()]
                 return D


          def avplqt(self):
                sum_trp = 0.0
                 for s  in  range(self.L):
                    for t  in  range(self.L):
                       D = self.plqt(s,t)
                       UP = np.dot(D[0],np.dot(D[1],np.dot(D[2],D[3])))
                       trup = (1.0 - ((1.0/ float(self.N))*np.trace(UP).real))
                       sum_trp = sum_trp + (trup/ float(self.L*self.L))
                 return sum_trp


          def wloop11(self,s,t):
                D = self.plqt(s,t)
                UP = np.dot(D[0],np.dot(D[1],np.dot(D[2],D[3])))
                wtr =  (1.0/ float(self.N))*np.trace(UP).real
                 return wtr
```

15 Plotting with Errorbar

According to D.J. Gross and E.Witten, expectation value of wilson loop operator $W(g^2, N)$ in case of SU(N) gauge field in 2D is given by

$$w(\lambda) = \lim_{N\to\infty} w(g^2, N) = \begin{cases} \frac{1}{\lambda} : \lambda \geq 2 \\ 1 - \frac{\lambda}{4} : \lambda \leq 2. \end{cases} \tag{59}$$

Where $\lambda = g^2 N$. One can use above equation to check the correctness of codes for SU(N).

```
def Mean_Error(stor_w11):
       nt = len(stor_w11)
       ver = [0.0  for k  in  range(nt)]
       mw11 = 0.0

        for k  in  range (nt):
           mw11 = mw11+stor_w11[k]/ float(nt)
        for l  in  range (nt):
           ver[l] = (stor_w11[l]-mw11)**2
       s_error = math.sqrt(sum(ver)/nt**2)
        return mw11, s_error


def exportstorw(N,l,titr,alpha,flip):
          sitr = 100
          storw = [0.0  for k  in  range(titr-sitr)]
          ll = 1
          U = Start(l,N).cold_start()
           while (ll < titr+1):

                  for s  in  range(l):
                      for t  in  range(l):
                          for r  in  range(2):
                              failed, U = Update(U,l,N).link(r,s,t,alpha ,flip)
                w11 = Calculate(U,l,N).wloop11(3,3)

                print alpha,w11
                 if ll > sitr:
                    storw[ll-sitr-1] = w11
                ll = ll+1

           return storw


def erorbar(N,l,titr):
          plot_dir = "/Users/dibakarsigdel/Dropbox/Plots/"
          data_dir = "/Users/dibakarsigdel/Dropbox/Data/"
          Nmax = 28
```

```
        Nvalue = 1
        dlamda = 0.25
        x = [0.0  for k  in  range(Nmax)]
        y = [0.0  for k  in  range(Nmax)]
        y_error = [0.0  for k  in  range(Nmax)]
         while Nvalue < Nmax+1:
               lamda = dlamda*Nvalue
               alpha = (2.0*N)/lamda
               x[Nvalue-1] = (2.0*N)/alpha
               storw = exportstorw(N,l,titr,alpha)
               y[Nvalue-1],y_error[Nvalue-1] = Mean_Error(storw)
               #print x[Nvalue-1],y[Nvalue-1]
               plt.figure(104)
               plt.xlabel('lambda')
               plt.ylabel('W11')
               plt.grid()
               plt.errorbar(x,y, yerr = y_error, fmt='8')
               st =  str(N)
               plt.savefig(plot_dir + 'plotsu'+st+'.png')
               plt.show()
               wnr('su'+st+'.dat',[x,y,y_error]).writer()
               wnr(data_dir +'su'+st+'.dat',[x,y,y_error]).writer()
               Nvalue = Nvalue+1

         return
```

15.1 Plot for SU(2)

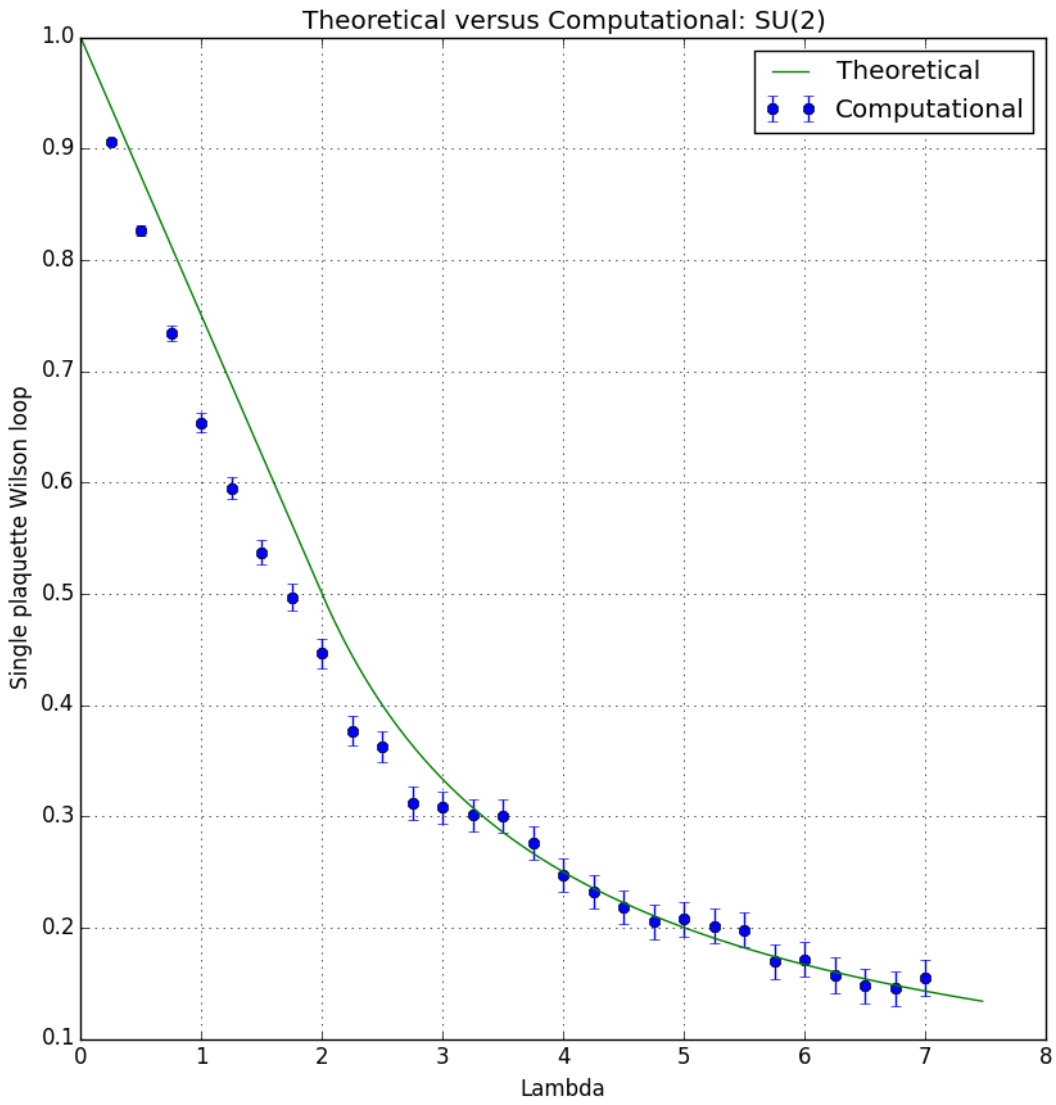


Figure 12: Plot of single plaquette Wilson loop versus lambda for SU(2) with lattice size: L = 30

15.2 Plot for SU(3)

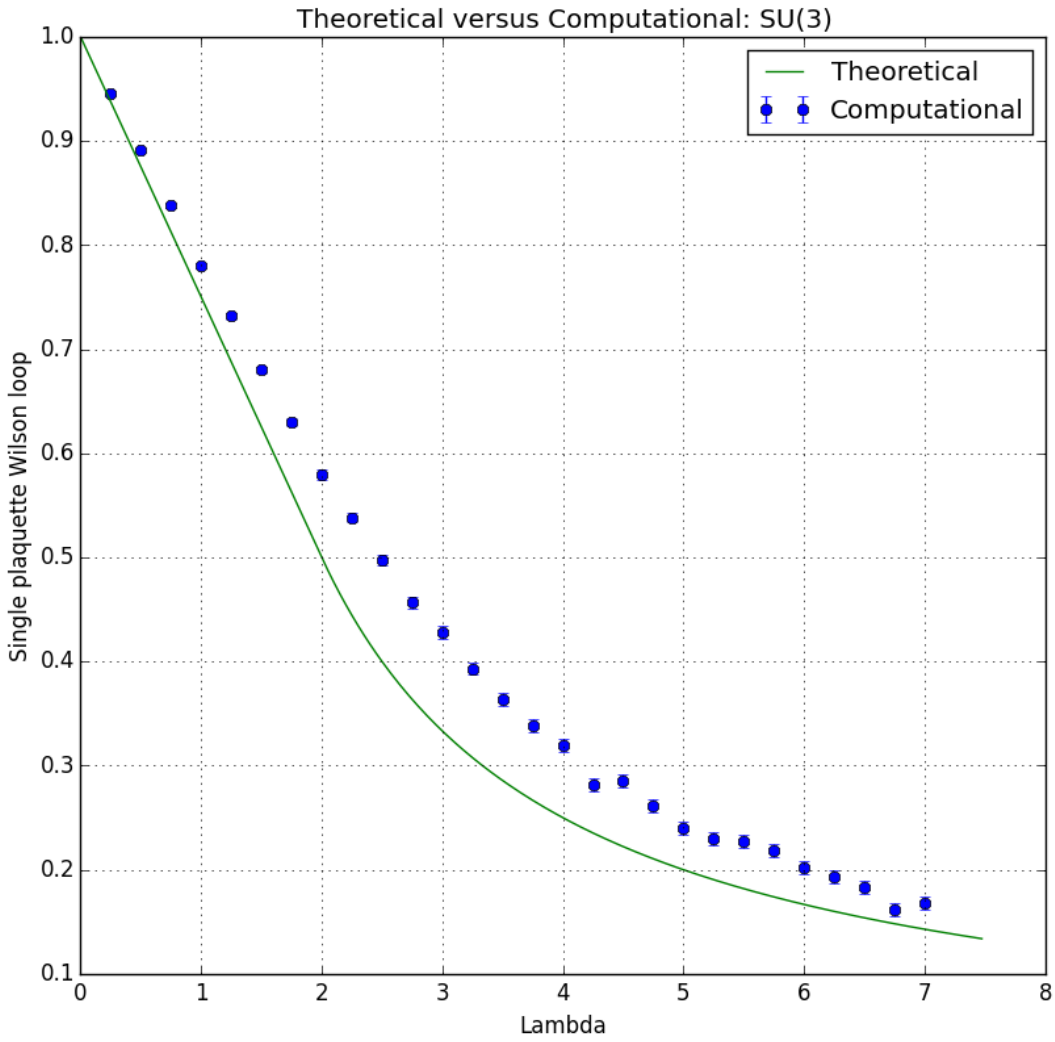


Figure 13: Plot of single plaquette Wilson loop versus lambda for SU(3) with lattice size: L = 30

15.3 Plot for SU(4)

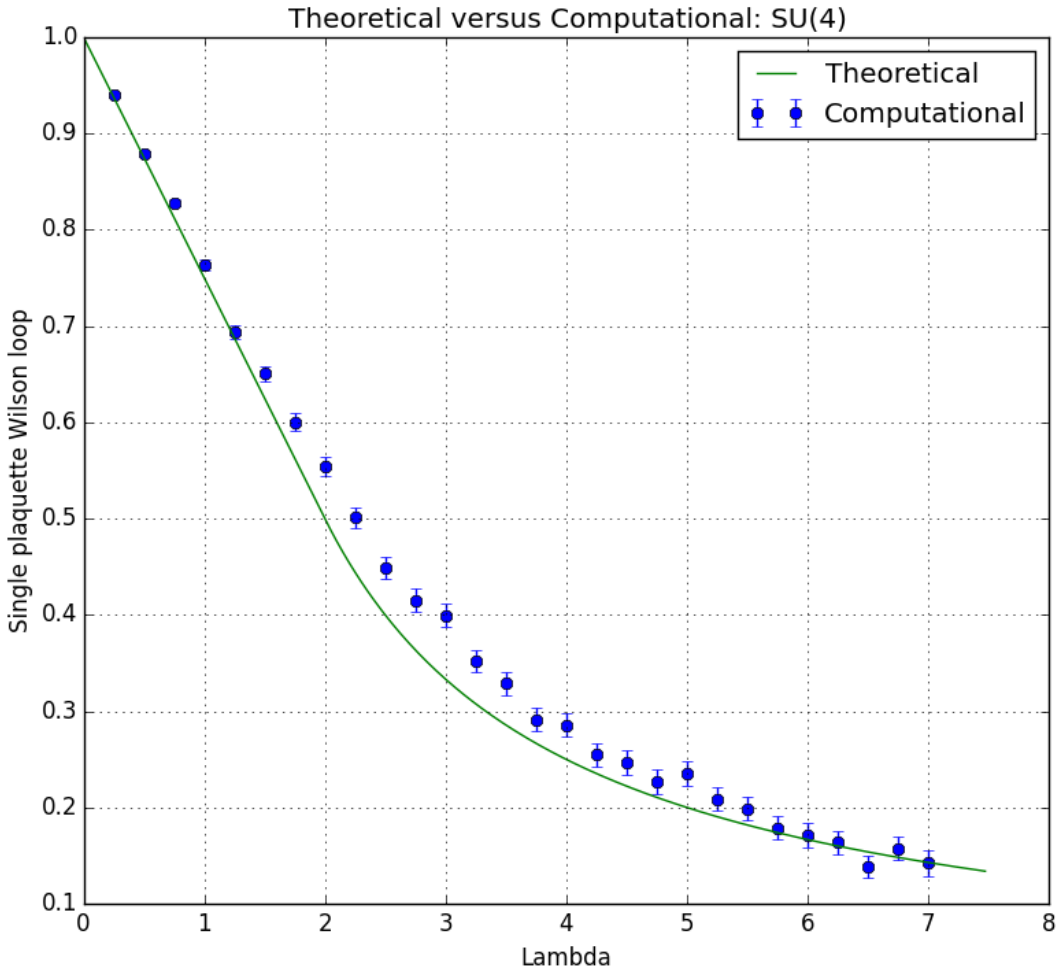


Figure 14: Plot of single plaquette Wilson loop versus lambda for SU(4) with lattice size: L = 30

15.4 Plot for SU(5)

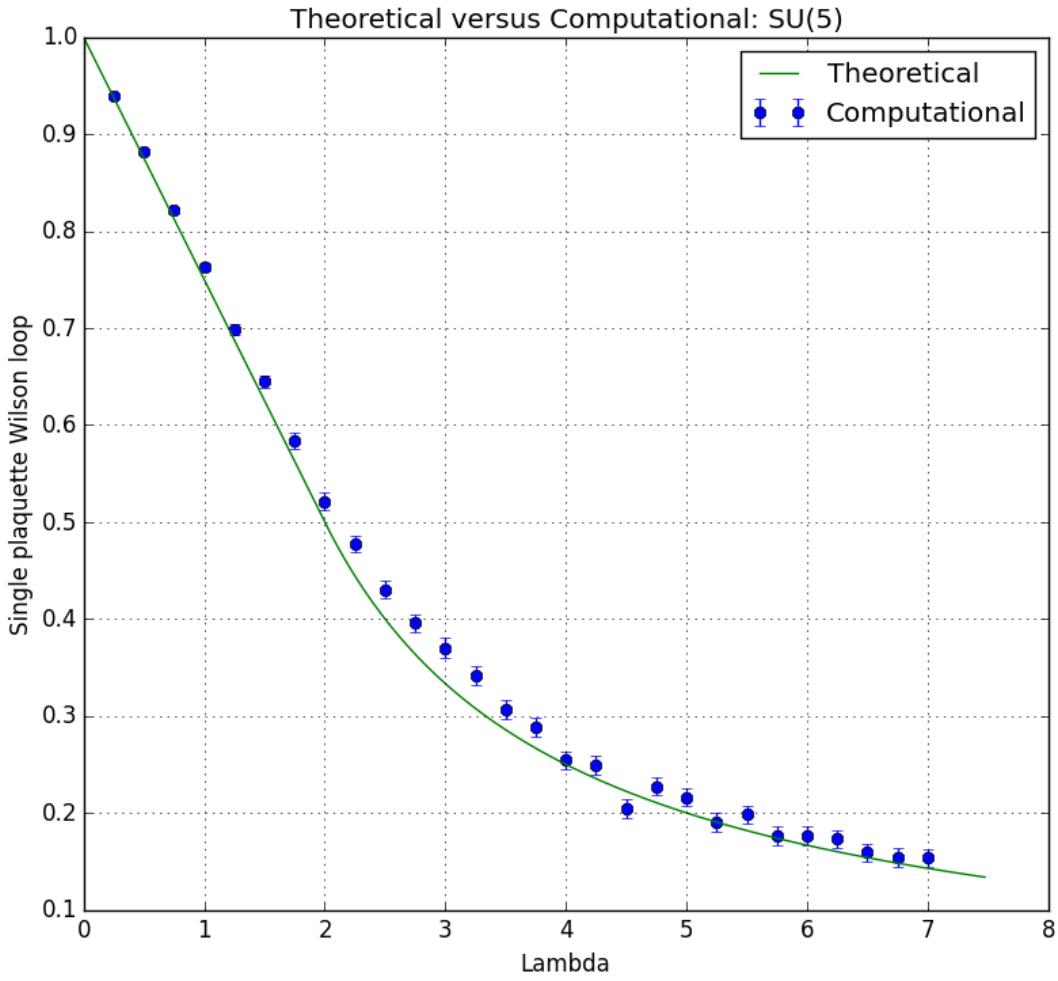


Figure 15: Plot of single plaquette Wilson loop versus lambda for SU(5) with lattice size: L = 30

15.5 Plot for SU(6)

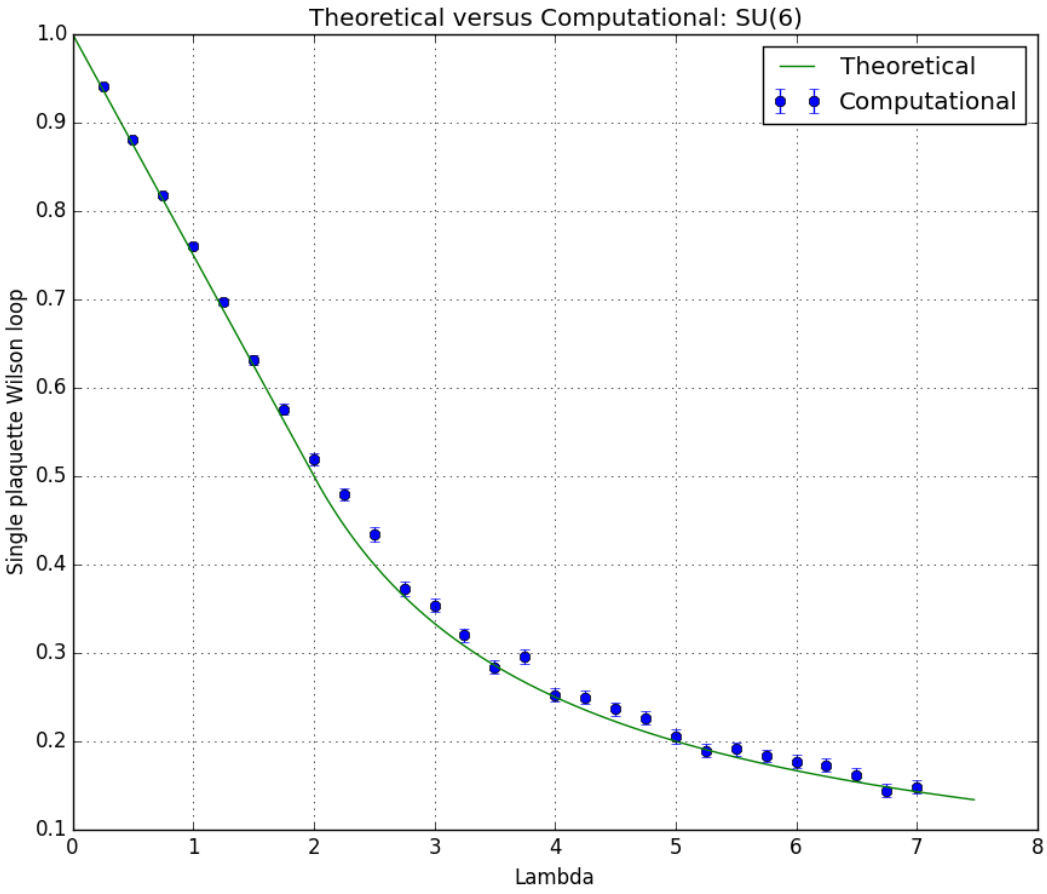


Figure 16: Plot of single plaquette Wilson loop versus lambda for SU(6) with lattice size: L = 30

15.6 Plot for SU(7)

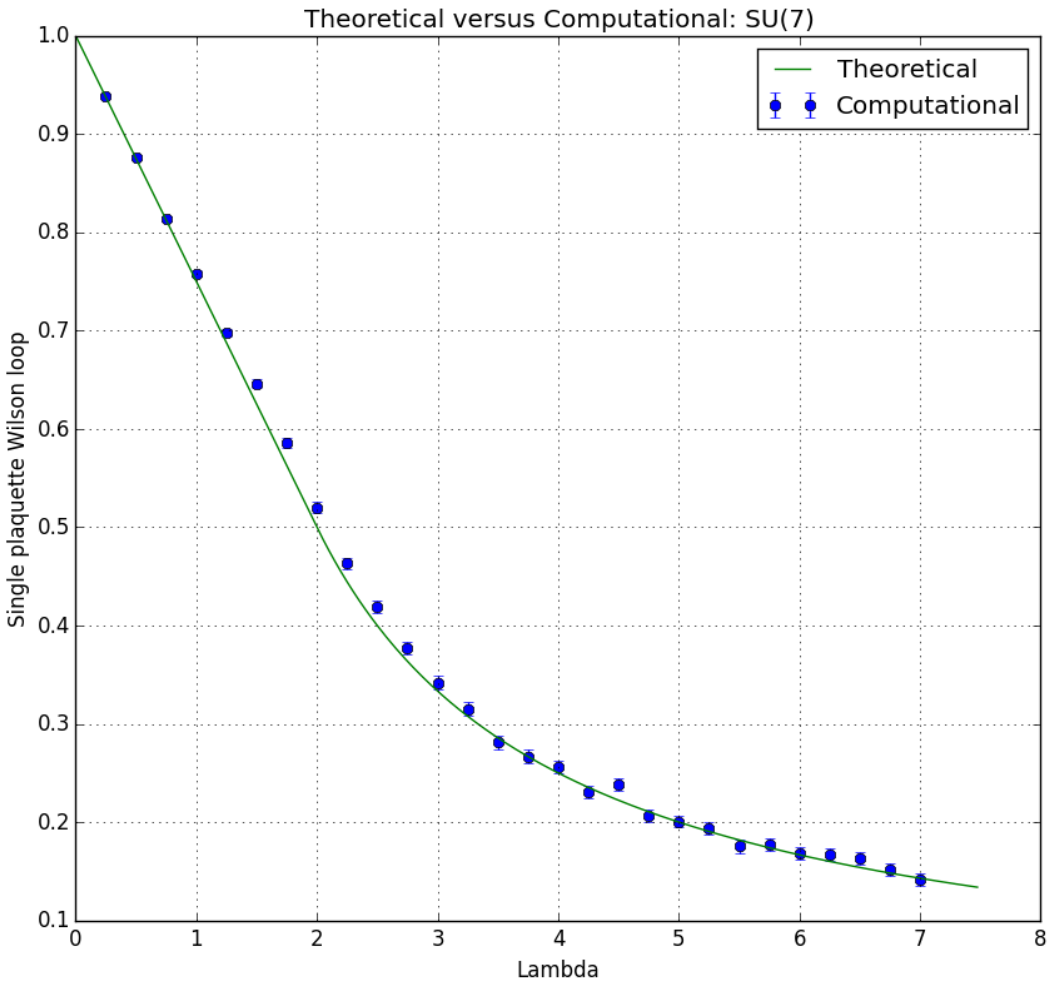


Figure 17: Plot of single plaquette Wilson loop versus lambda for SU(7) with lattice size: L = 30