Appendix B

Simulating boundaries for multipartite entangled states

In this appendix, I describe our numerical method of generating the boundaries $\Delta_b^{(3,2,1)}$ for biseparable states containing at most tripartite entanglement, bipartite entanglement and for fully separable states, as discussed in chapters 7–9.

B.1 Numerical GPU computing with NVIDIA's CUDA

Thermal (simulated) annealing[331–334] is one of the most successful *sequential* methods for global optimization of some function $f(\vec{\lambda})$ over parameters $\vec{\lambda}$. The specific problem we have in hand is a minimization problem of finding the boundaries $\Delta_b^{(M)}$ for states $\hat{\rho}^{(M)}$ which contain at most *genuine* M-partite entanglement. Violation of this bound $\Delta < \Delta_b^{(M)}$ *unambiguously* signals the presence of genuine $(M+1)$-partite entanglement. Thus, we have $f = \tilde{\Delta}_b^{(M)}$ and $\vec{\lambda} = \hat{\rho}^{(M)}$, where $\tilde{\Delta}_b^{(M)}$ is evaluated for some non-optimal state $\hat{\rho}^{(M)}$; see the next section for a concrete example of fully separable state ($M = 1$).

The initial attempt to obtain the boundaries $\Delta_b^{(M)}$ in a robust fashion was done by thermal annealing, as shown by Fig. B.1. Interestingly, the inspiration of thermal annealing comes from crystallography, where a sequence of heating and controlled cooling is used to grow the size of the crystal and minimize the defects[334]. For a given step n in the annealing procedure, heating allows thermal excitations[a] (with 'energy' $\tilde{k}_B T_n$) to avoid the state s_n getting stuck at some local minimum (in our case, $\tilde{\Delta}_b^{(M)}$) with transition probability $p_{n,n+1}$ from s_n to s_{n+1} (in our case, we choose a Boltzmann-like factor $p_{n,n+1} \sim \exp\left(-\frac{\tilde{\Delta}_b^{(M)}(s_{n+1})-\tilde{\Delta}_b^{(M)}(s_n)}{\tilde{k}_B T_n}\right)$)[b], while the controlled cooling of $\tilde{k}_B T$ allows to find the configurations (in our case, $s_n = \hat{\rho}_n^{(M)}$ to $s_{n+1} = \hat{\rho}_{n+1}^{(M)}$) with lower internal energy than the initial one (in our case, from $\tilde{\Delta}_b^{(M)}(s_n)$ to $\tilde{\Delta}_b^{(M)}(s_{n+1})$).

By analogy, in each step of the annealing algorithm, we replace the current state s_n by a random neigh-

[a]Here, the tilde '$\sim$' in $\tilde{k}_B$ is used to signify that $\tilde{k}_B$ is not the Boltzmann constant k_B but a scaling factor to make an analogy to energy.

[b]Note that because the transition probability $p_{n,n+1}$ is still higher for states with lower variance than the initial one, the annealing at a given temperature T_n tends to seek for solutions s_{n+1} with lower $\tilde{\Delta}_b^{(M)}(s_{n+1})$ than $\tilde{\Delta}_b^{(M)}(s_n)$.

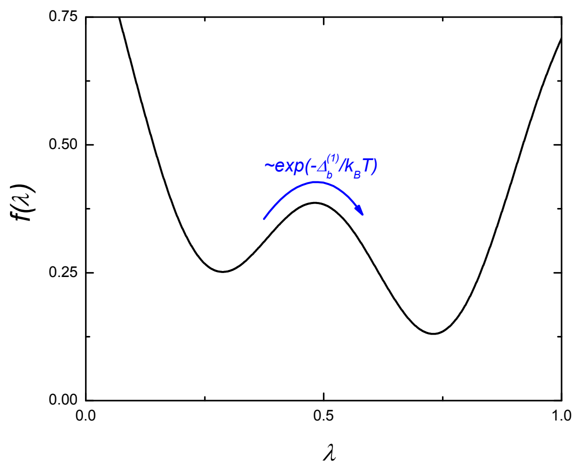

Figure B.1: **Global minimization of a multivariate function $f(\vec{\lambda})$ via simulated annealing.** For simulated annealing, thermal excitations can be used to overcome the local minima on the left with transition probability $p_{n,n+1} = \exp(-\frac{f(\vec{\lambda}_{n+1})-f(\vec{\lambda}_n)}{\tilde{k}_B T_n})$ as the step n progresses. In this scheme, one requires an annealing schedule to reduce the 'temperature' T_n as a function of the step size n, eventually cooling the system $\vec{\lambda}_n$ to the ground state with 'energy' f.

boring state s_{n+1}, chosen with probability $p_{n,n+1}$ depending on the "thermal" energy $\tilde{k}_B T_n$ that is gradually decreased during the process. Thus, in order to obtain the global minimum $\Delta_b^{(M)}$, the thermal annealing requires a dedicated method to manage the 'cooling' rate for $\tilde{k}_B T_n$, known as the annealing schedule, which decreases the thermal excitations of the parameters in $\hat{\rho}_n^{(M)}$ as n is increased (eventually, reaching to the optimal state, $\lim_{n\to\infty} \hat{\rho}_n^{(M)} = \hat{\rho}_{\text{opt}}^{(M)}$, where $\tilde{\Delta}_b^{(M)}(\hat{\rho}_{\text{opt}}^{(M)}) = \Delta_b^{(M)}$). While I was able to get a numerical agreement between the analytical result and the optimal solution obtained by thermal annealing for balanced verification interferometer, I found it difficult to find an efficient annealing schedule robust to the changes in parameters $\{\alpha, \beta, \eta\}$ describing the interferometer and the constraints $\{p_1, y_c\}$ on the quantum state $\hat{\rho}_n^{(M)}$ being considered, which resulted in either getting stuck in a local minimum (quenching), or taking a vast amount of time to converge on the global minimum. In addition, as thermal annealing is necessarily *sequential* (in that the new solution s_{n+1} depends on the older one s_n), I found it tricky to program the thermal annealing to take advantages of parallel computations.

As an alternative, I decided to implement a more comprehensive and exhaustive random search algorithm, which generated all possible $\hat{\rho}^{(M)}$ for a given set of parameters $\{\alpha, \beta, \eta\}$ and constraints $\{p_1, y_c\}$, and found the minimum variance $\Delta_b^{(M)}$ among them. As each step in the search algorithm is completely random and independent, the optimization problem is inherently parallel (*concurrent*), and importantly very simple to program. In chapter 8, I have used the built-in parallel computing toolbox in Matlab, where I found ~ 4 times the convergence time compared to that of using a single core[c].

Around the same time, I learned that NVIDIA had developed a parallel computing architecture, known as compute unified device architecture (CUDA), which use graphical processing unit (GPU) instead of the

[c]To generate the boundaries in Fig. 8.4 of chapter 8, it took ~ 1 week of running time.

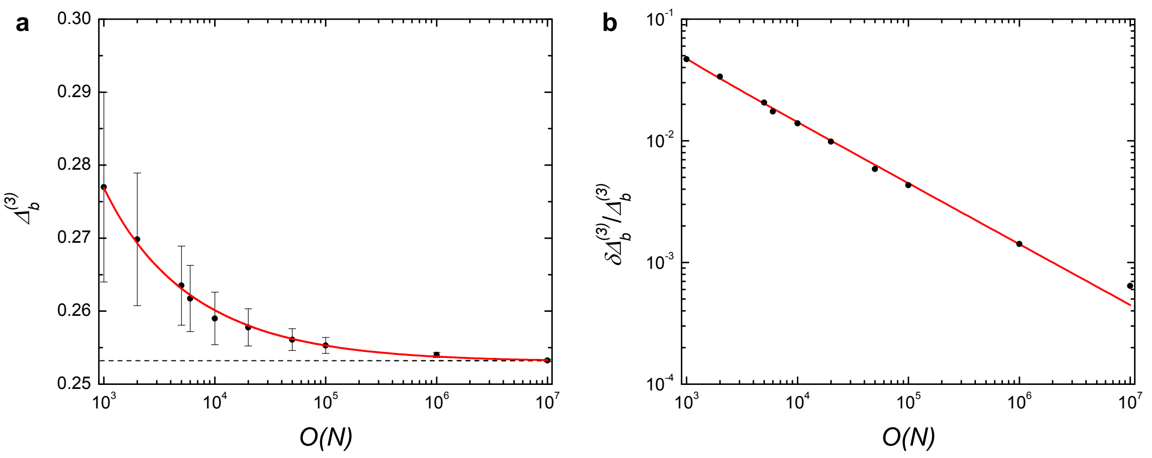

Figure B.2: **Accuracy and precision of the random search algorithm.** **a**, We show the convergence behavior (*accuracy*) of the Monte-Carlo method to obtain $\Delta_b^{(3)}$ by randomly searching for quantum states $\hat{\rho}^{(M)}$ (data points) to the analytical solution of $\Delta_b^{(3)}$ (dashed line) for balanced verification interferometer (section 7.4.3) as a function of numbers of quantum states $\mathcal{O}(N)$. **b**, We show the reduction in the fractional uncertainty $\frac{\delta\Delta_b^{(3)}}{\Delta_b^{(3)}}$ (*precision*) for numerically obtaining the solutions $\Delta_b^{(3)}$ (data points) as a function of numbers of quantum states $\mathcal{O}(N)$.

traditional CPU. From a computing perspective, GPUs are heavily multithreaded many-core chips, comprised of hundreds of cores, with each core capable of running multiple concurrent threads[d]. Since the introduction of CUDA, the so-called general purpose GPU (GPGPU) computing has become a major trend in scientific computing, thanks to the prospects of building a ‘personal’ supercomputers (exceeding 1 Teraflops) to solve certain (massively parallel) scientific problems. Important features of CUDA include shared memory, which can greatly improve the performance of bandwidth-limited applications; double precision floating point arithmetic; and an arbitrary load/store memory model, which enables many new algorithms which were previously difficult or impossible to implement on the GPU.

As of 2011, these GPU accelerations are now built into Mathematica 8 and Matlab 2011a. But at the time of the experiment in chapter 9, these features were not readily available. Thus, we implemented the GPU computing for the Monte-Carlo simulation with an open source development project for Matlab, called “GPUmat”. As described in section B.2, even with my limited experience on parallel computing models, I found that the GPU-accelerated code (GPUmat) employed in chapter 9 out-performed a similar parallel code using 4 CPU cores with Intel Xeon processor in chapter 8 by a factor of ~ 12. The enhancement in GPGPU computing due to increased data parallelism allowed us to perform error analysis of the boundaries $\Delta_b^{(1,2,3)}$ in Figs. 9.2–9.3 arising from the systematic uncertainties of the verification interferometers (section 9.13.3). For reference, in Fig. B.2, I show the convergence behavior of the numerical random search algorithm (data points) for $\Delta_b^{(3)}$ to the analytically predicted value of $\Delta_b^{(3)}$ (dashed line) for balanced verification interferometers (see chapter 7) as a function of $\mathcal{O}(N)$ numbers of quantum states with $\{p_1, y_c\} = \{0.1, 0.035\}$. In

[d]In my case, I just used NVIDIA GeForce GT 330M, comprised of 48 parallel processing units, each running at 1.26 GHz with multi-threading, yielding theoretically a total processing power ~ 180 Gigaflops.

section B.2, I will describe our method of the Monte-Carlo simulation using GPUmat.

B.2 Monte-Carlo simulation for realistic verification interferometers

To generate the boundaries, we need to consider all possible (mixed) states containing bipartite and tripartite entanglement, as well as fully separable states for a given p_1. I refer to chapter 7 for a more detailed theoretical formalism. Here, we will review the case for a fully separable state. A pure separable state $|\psi\rangle_{\text{sep}}^{(a)}$ is given by the form:

$$|\psi\rangle_{\text{sep}}^{(a)} = \prod_k |\varphi\rangle_k^{(a)}, \tag{B.1}$$

where $|\varphi\rangle_k^{(a)} = \frac{1}{\sqrt{1+|\epsilon_k^{(a)}|^2}}(|0\rangle + e^{i\phi_k^{(a)}}\epsilon_k^{(a)}|1\rangle)$. Based on Eq. B.1, we can further consider a mixed state of the form:

$$\hat{\rho}_{\text{sep}}^{(ab)} = |A_{ab}|^2 |\psi\rangle_{\text{sep}}^{(a)}\langle\psi| + (1-|A_{ab}|^2)|\psi\rangle_{\text{sep}}^{(b)}\langle\psi|, \tag{B.2}$$

which corresponds to the mixing of two pure separable states $|\psi\rangle_{\text{sep}}^{(a)}$ and $|\psi\rangle_{\text{sep}}^{(b)}$ with a ratio of $|A_{ab}|^2$: $(1-|A_{ab}|^2)$. We then calculate the various expectation values of $\hat{M}_k = |\Pi_k\rangle\langle\Pi_i|$ (see section 8.11 for the definition of the projectors $\hat{M}_k$ in the presence of imbalances $\{\alpha,\beta\}$ and losses $\{\eta\}$) as well as the variance $\tilde{\Delta}_b^{(1)} = \sum_k \text{Tr}(\hat{\rho}_{\text{sep}}^{(ab)}\delta M_k^2)$ for the randomly generated state $\hat{\rho}_{\text{sep}}^{(ab)}$ (specified by the random values A_{ab}, $\{\epsilon_1^{(a,b)}, \epsilon_2^{(a,b)}, \epsilon_3^{(a,b)}, \epsilon_4^{(a,b)}\}$, and $\{\phi_1^{(a,b)}, \phi_2^{(a,b)}, \phi_3^{(a,b)}, \phi_4^{(a,b)}\}$). The minimum value of $\tilde{\Delta}_b^{(1)}$ for all possible A_{ab}, $\{\epsilon_1^{(a,b)}, \epsilon_2^{(a,b)}, \epsilon_3^{(a,b)}, \epsilon_4^{(a,b)}\}$, and $\{\phi_1^{(a,b)}, \phi_2^{(a,b)}, \phi_3^{(a,b)}, \phi_4^{(a,b)}\}$ constrained by $\{p_1, y_c\}$ is what we denote by $\Delta_b^{(1)} = \min(\tilde{\Delta}_b^{(1)})$.

Similarly, we can construct $\Delta_b^{(2,3)} = \min(\tilde{\Delta}_b^{(2,3)})$ for biseparable states containing at most two-mode and three-mode entanglement. The only critical difference between the case of fully separable states and the case of (biseparable) entangled states is that we could also mix entangled states of different partitions for the latter case[e]. Realistically, the minimum uncertainties $\Delta_b^{(3,2,1)}$ for a given y_c will depend on the parameters $\{\alpha,\beta,\eta\}$ of the verification interferometer because of losses and imbalances in the verification setup. Due to the systematic uncertainties $\{\delta\alpha,\delta\beta,\delta\eta\}$ in our determinations of $\{\alpha,\beta,\eta\}$, $\Delta_b^{(3,2,1)}$ are convolved with the normal distributions of $\{\alpha,\beta,\eta\}$. This is a highly parallel computing problem of high concurrency, whereby we search for the minimum $\Delta_b^{(3,2,1)}$ for random quantum states (defined by *independent* variables A_{ab}, $\{\epsilon_1^{(a,b)}, \epsilon_2^{(a,b)}, \epsilon_3^{(a,b)}, \epsilon_4^{(a,b)}\}$, and $\{\phi_1^{(a,b)}, \phi_2^{(a,b)}, \phi_3^{(a,b)}, \phi_4^{(a,b)}\}$) convolved *independently* by the normal distributions of the parameters $\{\alpha,\beta,\eta\}$ which modify $\hat{M}_k$ (section 8.11).

In the matlab m-file, '`scan_batch.m`', we first run the 'GPUstart' command (see the open source CUDA project, 'GPUmat' for porting CUDA into Matlab) to initiate GPUmat package, and to check if the CUDA SDK is installed and if the GPU processors are CUDA compatible. The batch file then takes several inputs: constraints such as p_1 and y_c, numbers of states to generate and store in the GPU (or CPU) memory,

[e]e.g., (i) mixed state of a fully separable pure state and a biseparable entangled state containing two-mode entanglement, (ii) mixed state of bipartite entangled state for $(a|bcd)$ and $(abc|d)$, and etc.

numbers of states to wait until writing onto a temporary folder in a hard drive, and the parameters (α, β, η) describing the verification interferometer. Then, it loads and computes the functions, 'onemodescanner', 'twomodescanner', and 'threemodescanner', which are, respectively, defined in the codes '`onemodescanner.m`', '`twomodescanner.m`', and '`threemodescanner.m`'. Here, I list the Matlab code for the batch file.

```
%Scanner batch file for generating W-state bounds (K. S. Choi)
GPUstart
GPUmatSystemCheck
%This code assumes that that the GPU is compatible with NVIDIA?s CUDA SDK.

%Define phase uncertainty given by \Δ \phi=2\pi/phase_unc
%N_tot=Total number of states per each step of yc
%N_buff=Total number of states to store in the Ram before storing into hard
%drive
%yc_steps=number of equal steps for yc
phase_unc=1000;
p1=0.3;
N_tot=100000;
N_buff=50000;
yc_steps=20;

%Define the loss and imbalance parameters for the verification
%interferometer
betain=0.497;
betaout=0.484;
alphaAB=1-0.490;
alphaCD=0.487;
etaAout=0.948;
etaBin=0.958;
etaCin=0.899;
etaDout=0.932;
etaA=0.7*0.770;
etaB=0.7*0.776;
etaC=0.7*0.747;
etaD=0.7*0.718;
%err=fractional systematic error in the parameters above, following a
%normal distribution.
%stat_it=number of loss and imbalance parameters to
%generate per degenerate states (y_c) to create per each

err=0.05;
stat_it=250;
%load function XXXmodescanner() which initiate the Monte-Carlo sim and
```

```
%export into files
onemodescanner(phase_unc,p1,betain,betaout,alphaAB,alphaCD,etaAout,...
    etaBin,etaCin,etaDout,etaA,etaB,etaC,etaD,N_tot,N_buff,yc_steps,...
    2.19,100,err,stat_it);
twomodescanner(phase_unc,p1,betain,betaout,alphaAB,alphaCD,etaAout,...
    etaBin,etaCin,etaDout,etaA,etaB,etaC,etaD,N_tot,N_buff,yc_steps,...
    1.3175,1.4,err,stat_it);
threemodescanner(phase_unc,p1,betain,betaout,alphaAB,alphaCD,etaAout,...
    etaBin,etaCin,etaDout,etaA,etaB,etaC,etaD,N_tot,N_buff,yc_steps,...
    0.912,1,err,stat_it);
%end
%matlabpool close;
%check parity
 load -ascii 'one_mode_bound.mat'
 load -ascii 'two_mode_bound.mat'
 load -ascii 'three_mode_bound.mat'

plot(one_mode_bound(:,1),one_mode_bound(:,2),'r',one_mode_bound(:,1),...
    one_mode_bound(:,3),'--r',one_mode_bound(:,1),one_mode_bound(:,4),...
    '--r',two_mode_bound(:,1),two_mode_bound(:,2),'b',...
    two_mode_bound(:,1),two_mode_bound(:,3),'--b',two_mode_bound(:,1),...
    two_mode_bound(:,4),'--b',three_mode_bound(:,1),...
    three_mode_bound(:,2),'g',three_mode_bound(:,1),...
    three_mode_bound(:,3),'--g',three_mode_bound(:,1),...
    three_mode_bound(:,4),'--g')
xlabel('\it{y_{c}}')
ylabel('\Delta_{b}')
title('\Delta_{b} vs. \it{y_{c}}')
grid on
```

The actual computations for obtaining the minimum variances $\Delta_b^{(3,2,1)}$ for a given set of $\{y_c\}$ are run by the m-files, '`onemodescanner.m`', '`twomodescanner.m`', and '`threemodescanner.m`'. Here, the variance $\tilde{\Delta}_b^{(3,2,1)}$ for the given quantum state is computed by Matlab functions 'variance3m', 'variance2m', 'variance1m'. To avoid redundancy, here I only list the Matlab m-code for '`onemodescanner.m`', which generates the lower bound of $\Delta_b^{(1)}$ for fully separable states[f]. Note that 'GPUsingle' is used to load the parameters into the GPU memory, and we used standard GPU functions such as 'GPUmin' and 'GPUmax', as well as custom GPU functions such as 'variance1m'. Direct coding with CUDA C language would have required substantially more effort than to simply write the code with GPUmat, especially in terms of managements of

[f]Note that $\Delta_b^{(1)}$ for a given set of parameters of the quantum state (A_{ab}, $\{\epsilon_1^{(a,b)},\epsilon_2^{(a,b)},\epsilon_3^{(a,b)},\epsilon_4^{(a,b)}\}$, and $\{\phi_1^{(a,b)},\phi_2^{(a,b)},\phi_3^{(a,b)},\phi_4^{(a,b)}\}$), and the verification interferometer ($\{\alpha,\beta,\eta\}$ and $\{\beta_1,\beta_2,\beta_3\}$) is obtained through a compiled GPU function, 'variance1m.mex'. This function internally takes these GPU variables describing the quantum state and the verification interferometer, and then calculates $\tilde{\Delta}_b^{(1)}$ with single-precision. The GPU function 'variance1m' is a machine-specific. In practice, we analytically calculated the function $\tilde{\Delta}_b^{(1)}$ with Quantum Mathematica and converted to an m-file. This m-file is later precompiled in Matlab similar to the command 'emlmex' (for running embedded Matlab mex) before running the batch file in order to speed up the computation.

the data transfers between the CPU and GPU, concurrency, and their memory usages.

```
function onemodescanner(sigma,p1,betain,betaout,alphaAB,alphaCD,...
    etaAout,etaBin,etaCin,etaDout,etaA,etaB,etaC,etaD,Nloss,kj,resolve,...
    ycend,GPUmaxx,err,stat_it)
%GPUcompileStart(?onemodescanner?, ?-f?,
%sigma,p1,betain,betaout,alphaAB,alphaCD,etaAout,etaBin,etaCin,etaDout,
%etaA,etaB,etaC,etaD,Nloss,kj,resolve,ycend,GPUmaxx,err,stat_it)
%Here, we use GPUmat for GPU acceleration
iter=GPUsingle([0,0,0,0;0,0,0,1;0,0,1,0;0,0,1,1;0,1,0,0;0,1,0,1;0,1,1,0;0, ...
  1,1,1;1,0,0,0;1,0,0,1;1,0,1,0;1,0,1,1;1,1,0,0;1,1,0,1;1,1,1, ...
  0;1,1,1,1]);
time0=cputime;

% qc2s=(p1/GPUmaxx*rand(1,Nloss));
% qc4s=(p1/GPUmaxx*rand(1,Nloss));
et=GPUsingle(10^-4:ycend/resolve:ycend);
p2s=GPUsingle(3/8*et*p1^2);
prog_wait=waitbar(0,'1-mode boundary is being scanned. Please Wait ...');
betain_list=GPUsingle(random('norm',betain,betain*err,1,stat_it));
betaout_list=GPUsingle(random('norm',betaout,betaout*err,1,stat_it));
alphaAB_list=GPUsingle(random('norm',alphaAB,alphaAB*err,1,stat_it));
alphaCD_list=GPUsingle(random('norm',alphaCD,alphaCD*err,1,stat_it));
etaAout_list=GPUsingle(random('norm',etaAout,etaAout*0,1,stat_it));
etaBin_list=GPUsingle(random('norm',etaBin,etaBin*0,1,stat_it));
etaCin_list=GPUsingle(random('norm',etaCin,etaCin*0,1,stat_it));
etaDout_list=GPUsingle(random('norm',etaDout,etaDout*0,1,stat_it));
etaA_list=GPUsingle(random('norm',etaA,etaA*err,1,stat_it));
etaB_list=GPUsingle(random('norm',etaB,etaB*err,1,stat_it));
etaC_list=GPUsingle(random('norm',etaC,etaC*err,1,stat_it));
etaD_list=GPUsingle(random('norm',etaD,etaD*err,1,stat_it));
%std(random('norm',1,0.5,1,100))
%mean(random('norm',1,0.5,1,100))
init=20;
% GPUmaxten=GPUmaxx*((init-1)/ycend*(ycend-et)+1);
GPUmaxten=GPUsingle(GPUmaxx*((et./ycend).^(-1/init)-1)+1);
    clear et;
    %varri=zeros(1,Nloss);
    varianceout=GPUsingle(zeros(length(p2s),4));
    varianceout2=GPUsingle(zeros(length(p2s),stat_it));
    GPUfor iqq=GPUsingle(1:length(p2s))

        p2=p2s(iqq);
        varritemp=GPUsingle(zeros(1,floor(Nloss/kj+1)));
        let=1;

        vari_list=GPUsingle(zeros(1,stat_it));
        qc2s=GPUsingle(p1/GPUmaxten(iqq)*rand(1,Nloss));
        qc4s=GPUsingle(p1/GPUmaxten(iqq)*rand(1,Nloss));
        GPUfor stat_itr=GPUsingle(1:stat_it)
            waitbar((iqq+stat_itr/stat_it-1)/length(p2s));
            betain=GPUsingle(betain_list(stat_itr));
            betaout=GPUsingle(betaout_list(stat_itr));
            alphaAB=GPUsingle(alphaAB_list(stat_itr));
            alphaCD=GPUsingle(alphaCD_list(stat_itr));
            etaAout=GPUsingle(etaAout_list(stat_itr));
            etaBin=GPUsingle(etaBin_list(stat_itr));
            etaCin=GPUsingle(etaCin_list(stat_itr));
            etaDout=GPUsingle(etaDout_list(stat_itr));
            etaA=GPUsingle(etaA_list(stat_itr));
            etaB=GPUsingle(etaB_list(stat_itr));
            etaC=GPUsingle(etaC_list(stat_itr));
            etaD=GPUsingle(etaD_list(stat_itr));
            count=1;
            GPUfor itr = GPUsingle(1:2^4 )
                phi1s=GPUsingle(random('unif',-pi/sigma+...
                    pi*iter(itr,1),pi/sigma+...
                    pi*iter(itr,1),1,Nloss));
                phi2s=GPUsingle(random('unif',-pi/sigma+...
                    pi*iter(itr,2),pi/sigma+...
                    pi*iter(itr,2),1,Nloss));
                phi3s=GPUsingle(random('unif',-pi/sigma+...
                    pi*iter(itr,3),pi/sigma+...
                    pi*iter(itr,3),1,Nloss));
                phi4s=GPUsingle(random('unif',-pi/sigma+...
                    pi*iter(itr,4),pi/sigma+...
                    pi*iter(itr,4),1,Nloss));
                GPUfor cc=GPUsingle(1:kj:Nloss)
                    t0=GPUsingle(let);
                    i1=cc:GPUmin(cc+kj,Nloss);
                    qc2=qc2s(i1);
                    qc4=qc4s(i1);
                    phi1=phi1s(i1);
                    phi2=phi2s(i1);
                    phi3=phi3s(i1);
                    phi4=phi4s(i1);
                    hq1=GPUsingle((-1/2).*((-1)+p1+p2).^(-1).*(1+...
                        qc2).^(-1).*(1+qc4).^(-1).*(p1+(-1).*qc2+ ...
                    2.*p1.*qc2+p2.*qc2+(-1).*qc2.^2+p1.*qc2.^2+...
```

```
                        p2.*qc2.^2+(-1).*qc4+2.*p1.* ...
                        qc4+p2.*qc4+(-2).*qc2.*qc4+3.*p1.*qc2.*qc4+...
                        2.*p2.*qc2.*qc4+(-1).* ...
                        qc2.^2.*qc4+p1.*qc2.^2.*qc4+p2.*qc2.^2.*qc4+...
                        (-1).*qc4.^2+p1.*qc4.^2+p2.* ...
                        qc4.^2+(-1).*qc2.*qc4.^2+p1.*qc2.*qc4.^2+...
                        p2.*qc2.*qc4.^2+((1+qc2).*(1+ ...
                        qc4).*((-4).*((-1)+p1+p2).*((-1).*qc2.^2+...
                        (-1).*qc2.*qc4+(-1).*qc2.^2.* ...
                        qc4+(-1).*qc4.^2+(-1).*qc2.*qc4.^2+p2.*(1+...
                        qc2).*(1+qc4).*((-1)+qc2+qc4)+ ...
                        p1.*(1+qc2).*(1+qc4).*(qc2+qc4))+(1+qc2).*(1+...
                        qc4).*(((-1)+p2).*(qc2+qc4) ...
                        +p1.*(1+qc2+qc4)).^2)).^(1/2));
                        hq3=(-1/2).*((-1)+p1+p2).^(-1).*(1+...
                            qc2).^(-1).*(1+qc4).^(-1).*(p1+(-1).*qc2+ ...
                        2.*p1.*qc2+p2.*qc2+(-1).*qc2.^2+p1.*qc2.^2+...
                        p2.*qc2.^2+(-1).*qc4+2.*p1.* ...
                        qc4+p2.*qc4+(-2).*qc2.*qc4+3.*p1.*qc2.*qc4+...
                        2.*p2.*qc2.*qc4+(-1).* ...
                        qc2.^2.*qc4+p1.*qc2.^2.*qc4+p2.*qc2.^2.*qc4+...
                        (-1).*qc4.^2+p1.*qc4.^2+p2.* ...
                        qc4.^2+(-1).*qc2.*qc4.^2+p1.*qc2.*qc4.^2+...
                        p2.*qc2.*qc4.^2+(-1).*((1+qc2) ...
                        .*(1+qc4).*((-4).*((-1)+p1+p2).*((-1).*qc2.^2+...
                        (-1).*qc2.*qc4+(-1).* ...
                        qc2.^2.*qc4+(-1).*qc4.^2+(-1).*qc2.*qc4.^2+...
                        p2.*(1+qc2).*(1+qc4).*((-1)+ ...
                        qc2+qc4)+p1.*(1+qc2).*(1+qc4).*(qc2+qc4))+...
                        (1+qc2).*(1+qc4).*(((-1)+p2).* ...
                        (qc2+qc4)+p1.*(1+qc2+qc4)).^2)).^(1/2)));
                        test=zeros(1,GPUmin(kj,Nloss-cc)+1);
                        GPUfor  kk=GPUsingle(1:GPUmin(kj,Nloss-cc)+1)
                             if GPUabs(imag(hq1(kk)))<0.1 && GPUabs(...
                                     imag(hq3(kk)))...
                                     <0.1 && (0≤(hq1(kk))≤1) && ...
                                     (0≤(hq3(kk))≤1) && (0≤(qc2(kk))≤1)...
                                     && (0≤(qc4(kk))≤1)
                                 test(kk)=1;
                             else
                                 test(kk)=0;
                             end

                        GPUend
                        var_ind=find(test==1);
```

```

                          varri = GPUmax(GPUabs(variance1m(p1,p2,qc2,qc4,...
                              betain,betaout,alphaAB,alphaCD,etaAout,etaBin,...
                              etaCin,etaDout,etaA,etaB,etaC,etaD,phi1,phi2,...
                              phi3,phi4)),0);

                        t1=GPUsingle(let+length(i1));
                        vv=varri(var_ind);
                        if isempty(var_ind)
                            vv=0;
                        end
                        varritemp(count)=GPUmin(vv);
                        count=count+1;
                        clear varri
                    GPUend
                GPUend
                vari_list(stat_itr)=GPUmin(varritemp);
            GPUend
            yc=8/3*(p2*(1-p1-p2))/p1^2;
            varianceout(iqq,1)=yc;
            varianceout2(iqq,1)=yc;
            hist(double(vari_list))
            mean_data=mean(double(vari_list));
            std_data=std (double(vari_list));
            x=mean_data-5*std_data:std_data/5:mean_data+5*std_data;
            y=histc (double(vari_list),x,2);
            f = ezfit (x,y,'gauss2 (x) = norm*(exp (-(x-xc)^2/(2*...(tm1*...
            heaviside (x-xc)+tm2*heaviside (-x+xc))^2)))',[stat_it/10 ...
                std_data std_data mean_data]);
            plot (x,y,'r*');
            showfit (f);
            % dispeqfit (f);
            makevarfit (f);
            tm1 = evalin ('base','tm1');
            tm2 = evalin ('base','tm2');
            xc = evalin ('base','xc');
            Δc=[xc, tm1, tm2];
            disp ('(Δc, +err, -err) = ');
            disp (Δc);
            vari_list=double(vari_list);
            GPUfor lstr=1:stat_it
            varianceout2(double(iqq),lstr)=vari_list(lstr);
            GPUend
             varianceout(double(iqq),2)=xc;
             varianceout(double(iqq),3)=xc+tm1;
```

```
         variancecout(double(iqq),4)=xc-tm2;
        clear let
    GPUend
close(prog_wait)
ploter = plot(varianceout(:,1),varianceout(:,2),varianceout(:,1),...
    varianceout(:,3),'--',varianceout(:,1),varianceout(:,4),'--');
set(gca,'XTick',0:0.25:1.5)
xlabel('y_c')
ylabel('\Delta^{(1)}_b')
title('\Delta^{(1)}_b vs. y_c')
set(ploter,'Color','blue','LineWidth',.5)
time1=cputime-time0;
disp(['elapsed time (s)=',num2str(time1)]);
save one_mode_bound.mat -ascii varianceout;
save one_mode_bound2.mat -ascii varianceout2;

%GPUcompileStop
end

%%
```