Computer Science Lab Rotation

Computational Science and Engineering Laboratory: Professor Petros Koumoutsakos

Supervised by: Dr. Panagiotis Angelikopoulos and Dr. Panagiotis Hadjidoukas

Student: Damianos Melidis, Msc in Computational Biology and Bioinformatics
Student id: 12-945-416, dmelidis@student.ethz.ch

February 4, 2014

Abstract

The main goal of this lab rotation was to implement the idea of Netwon update in the Metropolis-Hastings MCMC algorithm [T. Bui-Thanh and O. Ghattas] in the already implemented uncertrainty quantification framework.

1. Introduction

The prominent way to capture an unknown distribution of interest is by the general Metropolis-Hastings algorithm [N. Metropolis et al.] and [W. K. Hastings]. In this algorithm in order to generate samples of a target distribution π(x), we introduce a proposal distribution q(,), q(a,b) for two samples a and b, this distribution is more easy to compute, and stands for the probability moving from a to b. Afterwards we calculate the *acceptance* value for b given a and we select b with probability *min*(1, acceptance value). It can be shown that this procedure creates a Markov process with *such* properties to have π(x) as its' *stationary* distribution. This general idea is easy to implement and it is applicable to most of target and proposal distributions. A simple pseudocode of Metropolis-Hastings is shown below:

Algorithm 1 Metropolis-Hastings Algorithm

```
Choose initial $x_0$
for k = 0, ..., N-1 do
    Propose a new sample y from the proposal $q(x_k, y)$
    Compute $\pi(x)$, $q(x_k, y)$ and $q(y, x_k)$
    Calculate the acceptance probability as $a(x_k, y) = \min(1, \frac{\pi(y)q(y,x_k)}{\pi(x_k)q(x_k,y)})$
    Accept the current sample and update $x_{k+1} = y$ with probability $a(x_k, y)$ or reject and update $x_{k+1} = x_k$
end for
```

It can be shown that this algorithm for a normal proposal distribution proposes a new point y calculated as a scaled value of normal distribution with mean the current value and standard deviation the indentity matrix or:
$y = x_k + \sigma N(0, I_n)$ (1), where I_n is the nxn identity matrix.
We can see that the proposed y is a Euler-Maruyama discretization, using step Δt = σ^2, of the stochastic differential equation:
dx(t) = dW(t) (2), where W(t) stands for the standard Brownian motion in n dimensions.

Besides the Langevin dynamics with stohastic differential equation:
$\mathrm{d}x(t) = \frac{1}{2}\nabla \log(\pi(x))\mathrm{d}t + \mathrm{d}W(t)$ (3)
It is known that if we use as proposal the equation (3) we reach π(x) as stationary distribution. If we use the same technique as in (2) we apply Euler-Maruyama scheme to compute the proposal from:
$y = x_k + \frac{\sigma^2}{2}\nabla log(\pi(x_k)) + \sigma N(0, I_n)$ (4)
But unlike (3), the proposed y by (4) needs Metropolis-Hastings algorithm in order to reach the target π(x). Comparing the proposal from simple Metropolis Hastings (1) and the Langevin diffusion (4), they only differ on the deterministic part equaled to the scaled negative gradient of the logarithm of the target distribution ($\frac{\sigma^2}{2}\nabla log(\pi(x_k))$). Consequently the current point (x_k) plus this term is a move in the *negative* direction of the gradient of log(π(x)). Thus this *drift* term moves us to the neighbor of current x_k, for which the target distribution is maximized (given that σ^2 is sufficiently small) (e.g applying gradient descent). This modification of the general Metropolis-Hastings is called Metropolis-adjusted Langevin algorithm (MALA) [Gareth O. Roberts and Richard L. Tweedie]. Given the stohastic differential equation:
$\mathrm{d}x(t) = b(x)\ \mathrm{d}t + \beta(x)\ \mathrm{d}W(t)$ (5)
where $b_i(x) = \frac{1}{2}\sum_{j=1}^{n} a_{ij}(x)\ \frac{\partial}{\partial x_j}\ \log(\pi(x)) + \sqrt{\delta(x)}\sum_{j=1}^{n}\frac{\partial}{\partial x_j}(a_{ij}(x)\sqrt{\delta(x)})$, $\alpha(x) = \beta(x)\ \beta^T(x)$ and $\delta(x) = \det(\alpha(x))$. We can see that the Langevin dynamics (equation (3)) is an instance of (5). We can prove that this equation has as stationary distribution our target (π(x)) under specific conditions. The equation (5) is called Langevin diffusion on the Riemann manifold with tensor α^{-1}(x). After discretization of the equation (5) we get the proposed sample from the current is given by:
$y = x_k + \frac{\sigma^2}{2}\alpha(x_k)\nabla log(\pi(x_k)) + \sigma\beta(x_k)N(0, I_n) + \sigma^2\sqrt{\delta(x_k)}\sum_{j=1}^{n}\frac{\partial}{\partial x_j}(a_{ij}(x_k)\sqrt{\delta(x_k)})$ (6)
We can easily see that this proposal (6) is computationally expensive as we need to calculate the derivative of the α(x) in each step. As the derivative of α(x) is the manifold's curvature in the neighborhood of the currrent sample, the authors of [T. Bui-Thanh and O. Ghattas] consider as an alternative tensor the Hessian matrix of **f(x) = - log(π(x))**. Consequently the second term of (6) can be seen as a scaled Newton step to minimize the f(x). This Newton update is known to perform better than the gradient descent update, resulting to a better mixing of the Markov chains and a faster exploration of the target π(x). This introductory part is mainly based on [T. Bui-Thanh and O. Ghattas].

2. Method

In this section following the introduction's consideration of using the Hessian matrix as tensor for (6) we will try to propose a modification to Metropolis-Hastings algorithm. We use the Hessian matrix because we intend to avoid evaluating the "expensive" last term of (6), we use the assumption of constant curvature of current sample (x_k) and proposed sample (y). This lead us to:
y = x_k + $\frac{\sigma^2}{2}\mathbf{A}\nabla log(\pi(x_k))$ + $\sigma * N(0,\mathbf{A})$ (7), where **A** stands for the inverse of the Hessian for **f**($\pi(x)$) evaluated in current sample (x_k), equivalently A^{-1} = **H** = -$\nabla^2 log(\pi(x_k))$. Now we are able to compute how probable is to move from the current sample to the proposed one ($x_k \rightarrow$ y) quantified by q(x_k,y) and find how probable is the reverse motion (e.g traversing from the proposed sample to the current, y $\rightarrow x_k$) quantified by q(y, x_k). More speficically:
q(x_k,y) = $\frac{\sqrt{det\mathbf{H}}}{\sqrt{(2\pi)^n}}exp\{-\frac{1}{2\sigma^2}\|y - x_k - \frac{\sigma^2}{2}\mathbf{A}\nabla log(\pi(x_k))\|^2_{A^{-1}}\}$ (8.1)
q(y,x_k) = $\frac{\sqrt{det\mathbf{H}}}{\sqrt{(2\pi)^n}}exp\{-\frac{1}{2\sigma^2}\|x_k - y - \frac{\sigma^2}{2}\mathbf{A}\nabla log(\pi(y))\|^2_{A^{-1}}\}$ (8.2)
where the $\|z\|_{A^{-1}}$ is defined as the weigthed norm of z with respect to A^{-1} , *given that* that the inverse of this matrix is semi-positive definite.
Interpreting the equation (7), we see that to propose a new sample y given x_k we calculate two parts the *deterministic* and *stohastic* one. For the deterministic we evaluate a scaled (constant in the second addendum) Netwon step at x_k. Reflecting the last addendum in (4), we calculate a random "movement" from the Netwon step distributed by a normal distribution with mean 0 and covariance matrix A^{-1}(= **H**). The importance of the *free* parameter σ is being discussed in the section 5 of [T. Bui-Thanh and O. Ghattas] and it will be mentioned in "Results" of this report. The resulting modification of Metropolis-Hastings leads to:

Algorithm 2 Scaled stohastic Netwon Algorithm

```
Choose initial $x_0$
for k = 0, ..., N-1 do
  Compute $\nabla log(\pi(x_k))$ and $H(x_k) = -\nabla^2 log(\pi(x_k))$
  Propose a new sample y from the proposal density $q(x_k,)$ as defined in (8.1)
  Compute $\pi(x)$, $q(x_k, y)$ (8.1) and $q(y, x_k)$ (8.2)
  Calculate the acceptance probability as $a(x_k, y) = \min(1, \frac{\pi(y)q(y,x_k)}{\pi(x_k)q(x_k,y)})$
  Accept the current sample and update $x_{k+1} = y$ with probability $a(x_k, y)$ or reject and update $x_{k+1} = x_k$
end for
```

An reader interested on asympotic convergence and optimal scaling analysis (selection of paramter σ) of algorithm 2 is invited to read the section 3 and 4 of [T. Bui-Thanh and O. Ghattas] respectively.

3. Implementation

In the following, we are going to discuss the implementation of Section 2 method using the existing package *uq_framework* developped by my supervisors Dr. P. Angelikopoulos and Dr. P. Hadjidoukas. The package implements an evolutionary multi-chain MCMC in parallel, consequently the new method needs two basic modules, finding the candidate y given x_k applying (7) and evaluating the moving probabilities q(x_k,y) and q(y,x_k) (8.1,8.2). Three practical aspects are resolved. The ensurance that the hessian matrix **H** is semi-positive definite using a modification of [L.R. Schaeffer] for treating small positive eigenvalues or zero eigenvalues. Given this ensurance we exploit the matrix property to use the *Cholesky* decomposition for computing the inverse of the hessian matrix. Besides observing (7) we can see that the deterministic part *does not* change if the proposed point y is not selected. Exploiting this observation we compute the computationally expensive hessian

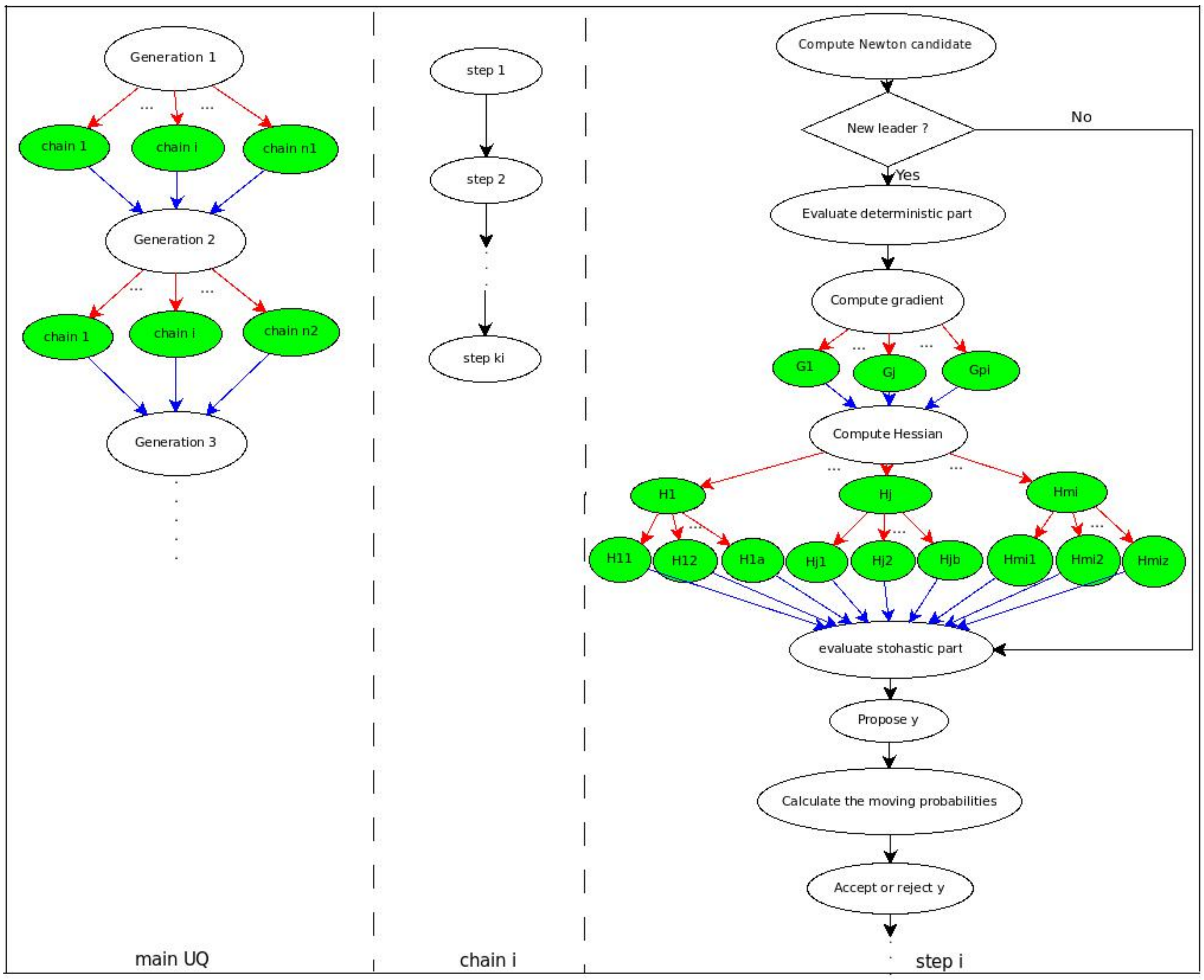


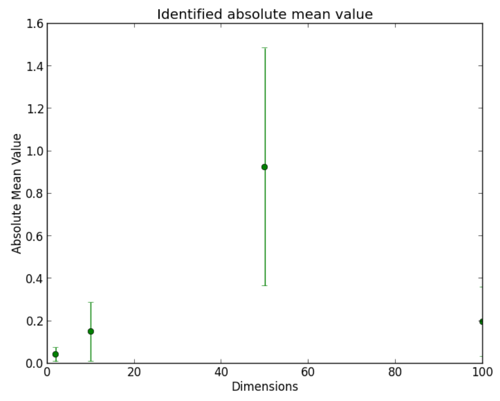
Figure 1: Task graph for method in Section 2.

matrix and gradient of $\mathbf{f}(\pi(x_k))$ at the current point, only if the previously proposed sample is selected.
The parallelization of the method is derived by the *torc* library which helps programmer to write parallel code(*MPI* or *OpenMP*) in heterogeneous clusters [P. E. Hadjidoukas, E. Lappas and V. V.Dimakopoulos]. In order to compensate the time consuming computation of the gradient and the hessian matrix, we use a specific library for parallel computation ([P. E. Hadjidoukas et al.]) of these quantities. In the figure 1 we illustrate the task graph for the proposed method, where from left to right we observe for a given generation how threads are spawn and work for each chain (red arrows (torc_create()) and blue arrows (torc_wait_all())). Afterwards a node process all steps for a specific chain and this in turn needs the proposed algorithm where *other* threads participate to the parallel computation of the gradient and Hessian matrix, where again red and blue arrows show the start and stop (join) of nodes used by *pndl* library [P. E. Hadjidoukas, E. Lappas and V. V.Dimakopoulos]. For more implementation information please read the Appendix section were the actual code is given.

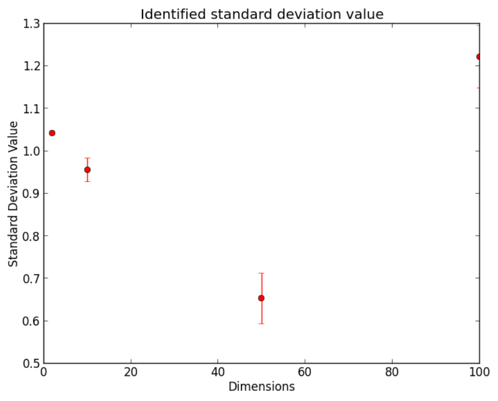
4. Experiments

In this section we will show experiments that are conducted to show the performance of the method. As in the section 5 of [T. Bui-Thanh and O. Ghattas], we consider two types of unknown distribution to approximate, the gaussian and the no gaussian distribution. Before setting up the experiments we have to consider a valuable value for σ (remember shows how much you follow the

Newton step to propose a new sample). The authors of [T. Bui-Thanh and O. Ghattas] state that for gaussian target we have to use $\sigma = l^2 * n^{-1/3}$ (from now on *optimal sigma*), where the l is set in order to have $\sigma^2 = 1$. In other words by looking in (7) we want to suggest that following the *half* of the Newton step we can guarantee that the acceptance ratio is sufficient to have valid results. For gaussian target we assume the normal distribution with $\mu = 0.0$ and covariance matrix equal to identity matrix (e.g $\text{cov}(x_i,x_i) = 1.0$) and we vary the dimensionality of the distribution in the set [2,10,50,100] (Figure 2). For dimensions 2 and 10 we used the *optimal sigma* and 5000 samples. For larger dimensions it is not clear if the *optimal sigma* is best choice as for 50 dimensions the algorithm has high acceptance ratio with smaller σ value ($\sigma = 0.2$). However this not the case for 100 dimensions, where if we a bigger value than the optimal one (intuitively follow more the Newton step) we can achieve 0.32% acceptance for merely 10000 samples. Verifying the algorithm for no gaussian target we choose the *Himmel-Blau* function to be identified. The authors of [T. Bui-Thanh and O. Ghattas] two values for σ (O($\frac{1}{n}$) or O($\frac{1}{n^2}$)), but testing these values for 3000 samples we did not achieve prominent results. Thus we did cross validation for sigma in log range of [0.1,1.0] and we found that the optimal σ is 0.03. For this value we vary the number of samples ([256,512,1024,2048]) trying to identify again the target (Figure 4). To compare these results with a baseline we run the previous implemented evolutionary MCMC (uq_framework) with 30.000 samples to identify the Himmel-Blau (Figure 3). Surprisingly even with the lowest number of samples the new method can capture the target function.



(a) Absolute Mean Value



(b) Standard Deviation Value

Figure 2: Identification of gaussian distribution varying its' dimensions.

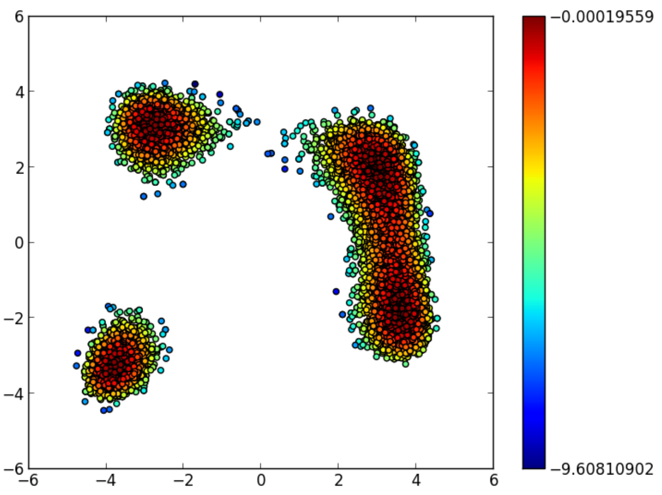
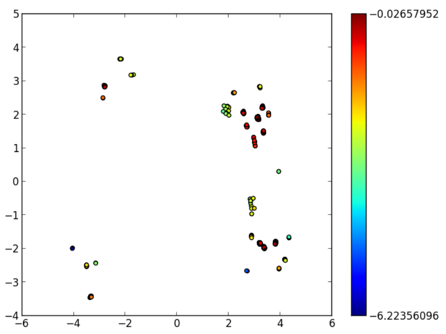
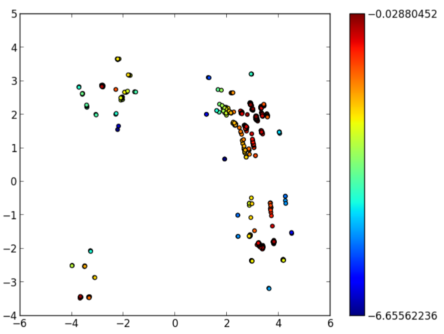


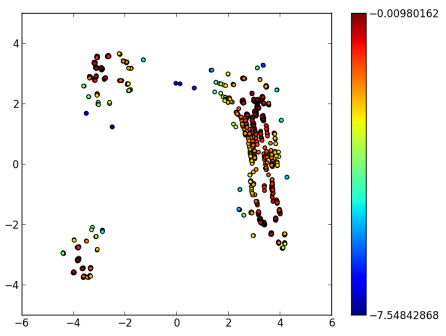
Figure 3: Identification of Himmel-Blau function from evolutionary MCMC (baseline).



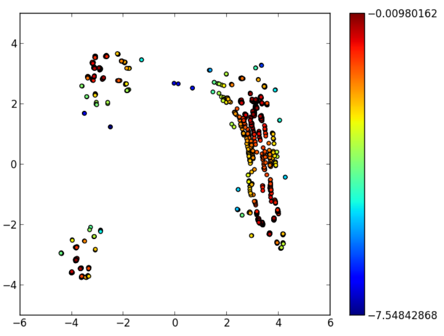
(a) 256 Samples



(b) 512 Samples



(c) 1024 Samples



(d) 2048 Samples

Figure 4: Identification of Himmel-Blau function for increasing number of samples.

5. Appendix

In the following the actual code is given:

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~MCMC with NEWTON UPDATE~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//damianos
//==============================================================================//
//======================dot_product=============================================//
```

```
//==========================================================//
/**
 *
 * input(1): transpose of first vector x^T
 * input(2): second vector (column vector) y
 *
 * output(1): x^T * y
 *
 * remarks: computes the dot product of input vectors
 */
double compute_dot_product(double row_vector[PROBDIM], double vector[PROBDIM]){

    int row;
    double sum = 0.0;

    for(row=0; row<PROBDIM; row++){
      sum += row_vector[row] * vector[row];
    }

    return sum;
}
//==========================================================//
//==========================================================//

//damianos
//==========================================================//
//===============compute_mat_product_vect==================//
/**
 *
 * input(1): matrix A
 * input(2): vector x
 * input(3): calculated A * x
 *
 * output: none
 *
 * remarks: compute the product of the given matrix with the given vector, which is multiplied by the
     given coefficient
 *     coef * (A * x)
 */
//==========================================================//
void compute_mat_product_vect(double mat[PROBDIM][PROBDIM], double vect[PROBDIM], double res_vect[PROBDIM
     ], double coef)
{

    int row, column;
    double current_dot_product;

    for(row=0; row<PROBDIM; row++){
  current_dot_product = 0.0;
  for(column=0; column<PROBDIM; column++){
    current_dot_product += mat[row][column] * vect[column]; //row
  }

  res_vect[row] = coef * current_dot_product;
    }
}
//=====================================================//
//=====================================================//

//damianos
//=====================================================//
//============compute_moving_probab====================//
//=====================================================//
/**
 * input(1): candidate point
 * input(2): current leader point
 * input(3): gradient on leader
 * input(4): hessian on leader
 * input(5): inverse of current hessian
 * input(3): random sigma (needed for stohastic Newton)
 * input(4): annealing coefficient
 * input(5): calculated probability q(xk,y) (equation 2.3)
 * input(6): calculated probability q(y,xk) (equation 2.4)
 *
 * output(): none
 *
 * remarks: function to calculate the moving probability q(xk -> y) and q(y -> xk) equation 2.3 and 2.4 of
      page 4
 * from publication "A scaled Stohastic Newton algorithm for MCMC"
 */
void compute_moving_probab(double candidate[PROBDIM], double leader[PROBDIM], double current_gradient[
    PROBDIM], double current_hessian[PROBDIM][PROBDIM], double inv_current_hessian[PROBDIM][PROBDIM],
     double rand_sigma, double anneal_coef, double *q_xk_y, double *q_y_xk)
{
    int row, column;
    double expont_xk_y[PROBDIM], expont_y_xk[PROBDIM];
    double hessian_product_exp_xk_y[PROBDIM], hessian_product_exp_y_xk[PROBDIM];
    double weig_norm_exp_xk_y, weig_norm_exp_y_xk;
    double inv_hessian_product_cur_gradient[PROBDIM] = {0.0}; //vector computed by inv_hessian *
     cur_gradient
    double inv_hessian_product_cand_gradient[PROBDIM] = {0.0}; // vector computed by inv_hessian *
     cand_gradient
    double coef_expont = -1.0 / (2.0 * pow(rand_sigma,2.0));
    double sum_xk_y, sum_y_xk;
    double gradient_on_candidate[PROBDIM] = {0.0};            // 1st derivatives on the candidate point

    //compute inv_hessian * grad(log(target(leader) ))
    compute_mat_product_vect(inv_current_hessian, current_gradient, inv_hessian_product_cur_gradient,
     anneal_coef); //anneal

    //compute the gradient on candidate point
    compute_grad(candidate, gradient_on_candidate);
    //compute inv_hessian * grad(log(target(cand_point)))
    compute_mat_product_vect(inv_current_hessian,  gradient_on_candidate,
     inv_hessian_product_cand_gradient, anneal_coef);

    // compute the two exponents in 2.3 and 2.4
    //expont_xk_y ->  candidate - leader - (sigma^2 /2) * inv(hessian(log(target_function))) * grad(log(
     target_function(xk)))
    //expont_y_xk -> leader - candidate - (sigma^2 /2) * inv(hessian(log(target_function))) * grad(log(
     target_function(y)))

    for(row=0;row < PROBDIM; ++row){
      expont_xk_y[row] = candidate[row] - leader[row] - ( (pow(rand_sigma,2)/2) * (anneal_coef) *
     inv_hessian_product_cur_gradient[row] ); // pow( , 2)
      expont_y_xk[row] = leader[row] - candidate[row] - ((pow(rand_sigma,2)/2) * (anneal_coef) *
     inv_hessian_product_cand_gradient[row]); // pow(, 2)
    }

    /*                                                  expont_q_xk_y
    //                                      /                 \
    //compute the exp ( (-1/2*(sigma^2)) * ||y - xk - ((sigma^2)/2)* A * grad ||H )
    //                          |                \     /                  \        /
    //                     coef_expont        xk_y           inv_hessian_product_cur_gradient
    */

    // || exponent_xk_y ||Hessian -> transpose(exponent_xk_y) * Hessian * exponent_xk_y

    //so first coef_expont compute Hessian * exponent_xk_y
    compute_mat_product_vect(current_hessian, expont_xk_y, hessian_product_exp_xk_y, coef_expont *
     anneal_coef);
    compute_mat_product_vect(current_hessian, expont_y_xk, hessian_product_exp_y_xk, coef_expont *
     anneal_coef);

    //then compute transpose(exponent) * hessian_product_exponent
    weig_norm_exp_xk_y = compute_dot_product(expont_xk_y, hessian_product_exp_xk_y);
    weig_norm_exp_y_xk = compute_dot_product(expont_y_xk, hessian_product_exp_y_xk);

    //apply exp to get practical q_xk_y and q_y_xk
    *q_xk_y = exp(weig_norm_exp_xk_y);
    *q_y_xk = exp(weig_norm_exp_y_xk);
}
//=====================================================//
//=====================================================//


//damianos
//=====================================================//
//==================Force_Pos_Def======================//
//=====================================================//
/**
 *
 * input(1): non positive definite matrix
 * input(2): resulted forced positive definite matix (output)
 *
 * remarks: Force matrix to be positive definite following
 *  publication "Efficient stohastic generation of multi-site synthetic precipitation data"
 *  by F.P.Brissette et al.
 * "Modification of a negative eigenvalues to create a positive
 * definite matrices and approximation for standard errors of correlation estimates by L.R. Schaeffer"
 */
void force_pos_def(gsl_matrix *non_pos_def_mat, gsl_matrix *forced_pos_def_mat){//, double
     forced_pos_def_mat[PROBDIM][PROBDIM]){

    int row, column, num_neg_eig_val, exist_pos_eig_values, exist_zero_eig_value;//, first_neg_value
    int diag_idx;
    double sum_diag, scaling_factor, current_value;
    double current_eig_value, sum_neg_eig_values, min_pos_eig_value; //min_pos_eig_value,
    double normalization_factor, small_pos_eig_value;

    double eps;
    eps = pow(10.0,-6);     //set a value to place instead of zero or negative eigen values
    double zero;
    zero = pow(10.0, -10); //set a value to look for zero

    gsl_matrix *working_mat, *temp_mat, *eig_vectors, *diag_mat, *intermediate_mat;//, *pos_def_mat;
    gsl_vector *eig_vec, *eig_values;
    gsl_eigen_symmv_workspace *work_v;

    working_mat = gsl_matrix_alloc(PROBDIM, PROBDIM);
    temp_mat = gsl_matrix_alloc(PROBDIM, PROBDIM);
    diag_mat = gsl_matrix_alloc(PROBDIM, PROBDIM);
    intermediate_mat = gsl_matrix_alloc(PROBDIM, PROBDIM);

    eig_vectors = gsl_matrix_alloc(PROBDIM, PROBDIM);
    eig_values = gsl_vector_alloc(PROBDIM);

    //copy input non positive matrix twice
    gsl_matrix_memcpy(working_mat,non_pos_def_mat);
    gsl_matrix_memcpy(temp_mat, non_pos_def_mat);

    //Get the eigenystem of the input matrix
    //allocate the space for the eigen values and copy temp_mat to gsl matrix
    work_v = gsl_eigen_symmv_alloc(PROBDIM);

    gsl_eigen_symmv(temp_mat, eig_values, eig_vectors, work_v); //get eigenvalues and eigen vectors


    min_pos_eig_value = DBL_MAX;//1000000000.0;//-1.0;//set the minimum positive eigen value to a big
     starting value
    exist_pos_eig_values = 0; //set the flag for existence of positive eigen values to FALSE
    sum_neg_eig_values = 0.0;
    exist_zero_eig_value = 0; //set the flag for zero eigen value existence to FALSE
    //first_neg_value = 1;
    /**
     * loop through the eigen values to find:
     * the minimum positive value
     * the sum of negative values
     */
    if(gsl_vector_isneg(eig_values)){ //if all eigen values are negative

      min_pos_eig_value = -gsl_vector_max(eig_values); //if all values are negative assumed that the min
     positive value is the maximum of the negative values
      for(column=0; column<PROBDIM; column++){//sum of negative values equals the sum of all vector
     elements
  sum_neg_eig_values += gsl_vector_get(eig_values, column);
      }
      exist_pos_eig_values = 1;
    }
    else{
      for(column=0; column < PROBDIM; column++){
  current_eig_value = gsl_vector_get(eig_values, column);
  if(current_eig_value > zero){ // if the current eigen value is positive, try to update the minimum
     positive eigen value
    if(current_eig_value <= min_pos_eig_value){
      min_pos_eig_value = current_eig_value;
    }
    exist_pos_eig_values = 1; //you found positive eigen value, set the flag to TRUE
  }
  else if (current_eig_value < 0.0){
    sum_neg_eig_values += current_eig_value;
  }
  else{
    //handling zero eigen value
    printf("Warning: Zero eigen value\n");
    exist_zero_eig_value = 1; //as you find zero eigen value, set flag to true
  }
      }
    }

    if (sum_neg_eig_values == 0.0 && exist_zero_eig_value){//let's check if only zero eigen values hinder
     matrix to be positive
      sum_neg_eig_values = 0.01; //set this value for correcting zero eigen values

      if (min_pos_eig_value == 1000000.0){

        printf("Warning: Correction of all zero eigen values \n");
  min_pos_eig_value = 101.0;
  sum_neg_eig_values = 1.0;
  exist_pos_eig_values = 1; //modifying flag as we recover the situtation of all zero eigen values
      }
    }

    assert(sum_neg_eig_values != 0.0); //test that the sum of negative values is negative(!)
    assert(exist_pos_eig_values && min_pos_eig_value != 1000000.0); //assert that the starting value for
     minimum is enough big

    sum_neg_eig_values = 2.0 * sum_neg_eig_values;
    //follow the publication on remarks to force the matrix to be positive definite
    normalization_factor = ( pow((sum_neg_eig_values),2) * 100.0) + 1.0; //compute the squared sum of
     negative values 2.0 * sum_neg
    //update the negative eigen values to create new small positive ones
    for(column=0; column< PROBDIM; column++){
      current_eig_value = gsl_vector_get(eig_values, column);

      if (current_eig_value <= zero){ //<= for zero or negative eigen value
  small_pos_eig_value = min_pos_eig_value * ( (pow(sum_neg_eig_values - current_eig_value,2)) /
     normalization_factor );

  while (small_pos_eig_value < eps){ //iterative find the closeste value to given eps
    small_pos_eig_value *= 10.0;
  }
  gsl_vector_set(eig_values, column, small_pos_eig_value); //update the eigen value
      }
    }

    //follows a Eigen_Vectors * diag(Eigen_Values) * (Eigen_Vectors)^T
    gsl_matrix_set_identity(diag_mat); //construct a diagonal matrix with eigen values in the first
     diagonal
    //hard copying the eigen values to the identity matrix to construct a diag[eig1 eig2 .. eign]
    for(row=0; row<PROBDIM; row++){
      gsl_matrix_set(diag_mat, row, row, gsl_vector_get(eig_values, row));
    }

    //now compute intermediate matrix = [v1 v2 .. vn] * diag[eig1 eig2 .. eign]
    gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, eig_vectors, diag_mat, 0.0, intermediate_mat);

    //finally compute pos definite matrix = [v1 v2 .. vn]  * diag[eig1 eig2 .. eign] * [v1 v2 .. vn]^T
    gsl_blas_dgemm(CblasNoTrans,CblasTrans, 1.0, intermediate_mat, eig_vectors, 0.0, forced_pos_def_mat);
     //pos_def_mat

    //free the workspace
    gsl_eigen_symmv_free(work_v);
    //free the matrices
    gsl_matrix_free(working_mat);
    gsl_matrix_free(temp_mat);
    gsl_matrix_free(diag_mat);
    gsl_matrix_free(intermediate_mat);
    gsl_matrix_free(eig_vectors);
    //free the vector
    gsl_vector_free(eig_values);
}
//==================================================//
//==================================================//
```

```
//damianos
//==============================================================//
//======================force_symm==========================//
//==============================================================//
/**
 *
 * input(1): matrix to be symmetric (input-output)
 *
 *output(0): none
 *
 * remarks: copies the upper triangular matrix to each lower forcing the matrix to be symmetric
 * the result is saved to the input matrix  [in-place]
 */
void force_symm(double input_mat[PROBDIM][PROBDIM]){//gsl_matrix *input_mat){

  int diag_coord, moving_coord;
  double value_to_copy;

  for(diag_coord=0; diag_coord<PROBDIM; diag_coord++){ //loop through the non diagonal elements and copy each row to its' corresponding column
      for(moving_coord=diag_coord+1; moving_coord<PROBDIM; moving_coord++){
  value_to_copy = input_mat[diag_coord][moving_coord];//gsl_matrix_get(input_mat, diag_coord, moving_coord);
  input_mat[moving_coord][diag_coord] = value_to_copy;//gsl_matrix_set(input_mat, moving_coord, diag_coord, value_to_copy);
      }
  }

}
//==============================================================//
//==============================================================//


//damianos
//==============================================================//
//==============check_mat_pos_def====================//
//==============================================================//
/**
 *
 * input(1): matrix to check if positive definite
 *
 * output(1): 1 or 0 if matrix is non negative or negative respectively
 *
 * remarks: check if input matrix is non negative
 */
int check_mat_pos_def(gsl_matrix *mat_to_check){
  //printf("Check pos def START\n");
  gsl_eigen_symm_workspace *work;
  gsl_vector *eig_values;
  int row,column, is_pos_def; //flag to show the matrix is positive definite

  //Get the eigen vales of hessian and check if there are positive
  work = gsl_eigen_symm_alloc(PROBDIM);
  eig_values = gsl_vector_alloc(PROBDIM);
  gsl_eigen_symm(mat_to_check, eig_values, work);//get the eigen values, the hessian_mat is going to be destroyed
  is_pos_def = gsl_vector_ispos(eig_values);

  gsl_vector_free(eig_values);
  gsl_eigen_symm_free(work);

  return is_pos_def;
}
//=========================================================//
//=========================================================//

//damianos
//=========================================================//
//==============check_mat_symmetry====================//
//=========================================================//
/**
 *
 * input(1): matrix to check for symmetry
 *
 * output(1): 1 or 0 if symmetric or non symmetric respectively
 *
 * remarks: checks if the input matrix is symmetric
 */
int check_mat_symmetry(gsl_matrix *mat_to_check){

  gsl_matrix *cur_hes_trans = gsl_matrix_alloc(PROBDIM, PROBDIM);

  gsl_matrix_transpose_memcpy(cur_hes_trans, mat_to_check);// take the transpose of the matrix

  int matrix_equal = gsl_matrix_equal(mat_to_check, cur_hes_trans);//if A = A^T => A symmetric

  gsl_matrix_free(cur_hes_trans);

  return matrix_equal;
}
//=========================================================//
//=========================================================//


//damianos
//=========================================================//
//===================inv_matrix======================//
//=========================================================//
/**
 * input(1): coefficient to multiply the matrix
 * input(2): hessian matrix
 * input(3): matrix to store the inverse of current hessian
 * output(): none
 *
 * remarks: compute the (inverse of the NEGATIVE hessian matrix)
 */
void inv_matrix(double coef, double current_hessian[PROBDIM][PROBDIM] ,double inv_current_hessian[PROBDIM
    ][PROBDIM])
{
    double current_hessian_copy[PROBDIM][PROBDIM]= {{0.0}};
    int row, column, s;
    int iter_forcing_pos_def;
    int is_pos_def; //flag to show if we need to force the hessian matrix to be semi positive definite
    int is_symm_mat; //flag to show if the matrix is symmetric

    gsl_matrix *hessian_mat = gsl_matrix_alloc(PROBDIM, PROBDIM);
    gsl_matrix *pos_def_mat = gsl_matrix_alloc(PROBDIM, PROBDIM);
    gsl_matrix *hessian_mat_cpy = gsl_matrix_alloc(PROBDIM,PROBDIM);

    for(row=0; row<PROBDIM; row++){
      for(column=0; column<PROBDIM; column++){
  current_hessian[row][column] = -1.0 * current_hessian[row][column];
      }
    }

    for(row=0; row<PROBDIM; row++){
      for(column=0; column<PROBDIM; column++){
  gsl_matrix_set(hessian_mat, row, column, coef * current_hessian[row][column]);
  current_hessian_copy[row][column] = coef * current_hessian[row][column];
      }
    }
    gsl_set_error_handler_off();

    gsl_matrix_memcpy(pos_def_mat, hessian_mat); //copy the current hessian to pos_def_mat
    gsl_matrix_memcpy(hessian_mat_cpy, hessian_mat);

    is_pos_def = check_mat_pos_def(hessian_mat_cpy);//check if the hessian is non negative definite,
     hessian_mat_cpy will be destroyed

    if(!is_pos_def){//if the matrix is negative force to be positive
      force_pos_def(hessian_mat, pos_def_mat);
    }

    //as the matrix now is semipositive definite apply cholesky decomposition to invert it
    gsl_linalg_cholesky_decomp(pos_def_mat);
    gsl_linalg_cholesky_invert(pos_def_mat);

    //pass the result to inv_current_hessian array in "hard way"
    for(row=0; row<PROBDIM; row++){
      for(column=0; column<PROBDIM; column++){
  inv_current_hessian[row][column] = gsl_matrix_get(pos_def_mat, row, column); //coef *
      }
    }

    //free used matrices
    gsl_matrix_free(hessian_mat);
    gsl_matrix_free(pos_def_mat);
    gsl_matrix_free(hessian_mat_cpy);
}
//==================================================//
//==================================================//


//==================================================//
//==============compute_grad========================//
//==================================================//
/**
 * input(1): vector with point to compute gradient (either leader or candidate)
 *
 * input(2): vector to save the computed gradient
 * output(): none
 *
 * remarks: find the gradient in the given input point of log(target_function)
 */
void compute_grad(double point_to_compute[PROBDIM], double computed_gradient[PROBDIM]){

  int pdim = PROBDIM;
  int i;
  double FEPS = 1e-3;
  int IPRINT = 0;
  int NOC;
  int JOBID = torc_worker_id()+1;
  int IERR;
  int IORD = 2;
  double XL[PROBDIM], XU[PROBDIM], UH[PROBDIM];

  for (i = 0; i < PROBDIM; i++)
  {
  XL[i] = -1e10;
  XU[i] = +1e10;
  UH[i] =  1e-3;
  }

  pndlga_(F, point_to_compute, &pdim, XL, XU, UH, &FEPS, &IORD, &IPRINT, computed_gradient, &NOC,&IERR,&
     JOBID);

}
//==================================================//
//==================================================//

//==================================================//
//==============compute_hessian=====================//
//==================================================//
/**
```

```
 * input(1): vector with leaders value, leader[]
 * input(2): array to save the hessian matrix
 * output(): none
 *
 * remarks: find the hessian in leader's point of log(target_function)
 */
void compute_hessian( double leader[PROBDIM], double current_hessian[PROBDIM][PROBDIM]){
    int pdim = PROBDIM;
    int i;

    double FEPS = 1e-3;
    int IPRINT = 0;
    int NOC;
    int JOBID = torc_worker_id()+1;
    int IERR;
    int IORD = 2;

    double XL[PROBDIM], XU[PROBDIM], UH[PROBDIM];

    for (i = 0; i < PROBDIM; i++)
    {
  XL[i] = -1e10;
  XU[i] = +1e10;
  UH[i] =  1e-3;
    }

    pndlhfa_(F, leader, &pdim, XL, XU, UH, &FEPS, &IORD, &IPRINT, current_hessian, &pdim, &NOC,&IERR,&
     JOBID);
}
//=============================================================//
//=============================================================//

//damianos
//=============================================================//
//=========compute_deterministic_part==================//
//=============================================================//
/**
 * input(1): leader coordinates, leader[]
 * input(2): current gradient
 * input(3): inverse of current hessian
 * input(4): deterministic part of the proposed sample
 * input(5): random sigma
 * input(6): annealing coef
 *
 * output: none
 *
 * remarks: compute the deterministic part of the equation 2.1
 * from publication "A scaled Stohastic Newton algorithm for MCMC"
 */
void compute_deterministic_part(double leader[PROBDIM], double current_gradient[PROBDIM], double
     inv_current_hessian[PROBDIM][PROBDIM], double deterministic_part[PROBDIM], double rand_sigma, double
     anneal_coef)
{
    int row, column;
    double inv_hessian_product_cur_gradient[PROBDIM] = {0.0};
    double current_dot_product;

    //compute the (inv(hessian)) * gradient(log(target_function))
    compute_mat_product_vect(inv_current_hessian, current_gradient, inv_hessian_product_cur_gradient,
     anneal_coef); //the inverse of hessian SHOULD be multiplied by the annealing coefficient

    //now compute the deterministic part of equation 2.1
    for(row=0; row<PROBDIM; row++){
      deterministic_part[row] = leader[row] + (pow(rand_sigma,2) / 2.0) * (anneal_coef) *
     inv_hessian_product_cur_gradient[row];
    }

}
//=============================================================//
//=============================================================//

//damianos
//=============================================================//
//===============newton_compute_candidate===============//
//=============================================================//
/**
 *
 * input(1): candidate vector
 * input(2): leader vector
 * input(3): gradient on the current point (leader)
 * input(4): hessian on current point (leader)
 * input(5): inverse hessian on current point
 * input(6): deterministic part of proposed sample
 * remarks: compute y = x_k + ((sigma^2)/2) * (-inv(hessian)) * gradient(log(target_function)) + sigma N
     (0,A)
 * publication "A scaled Stohastic Newton algorithm for MCMC" equation (2.1) page 4
 */
void newton_compute_candidate(double candidate[PROBDIM], double leader[PROBDIM], double current_gradient[
    PROBDIM], double current_hessian[PROBDIM][PROBDIM], double inv_current_hessian[PROBDIM][PROBDIM],
     double deterministic_part[PROBDIM], double rand_sigma, double anneal_coef, int new_leader_flag)
{
    int i,j, row, column;
    double mean_nrm[PROBDIM] = {0.0};
    double sigma_nrm[PROBDIM*PROBDIM] = {0.0};
    double stohastic_part[PROBDIM] = {0.0};

    //if we have a new leader then compute the gradient, hessian and the inverse of hessian
    //also calculate the deterministic part of equation 2.1
    if (new_leader_flag){
      compute_grad(leader, current_gradient);
      compute_hessian(leader, current_hessian);

      //change hessian matrix for FORTAN to C format, using force_symm()
      force_symm(current_hessian);
      inv_matrix(1.0, current_hessian, inv_current_hessian); //anneal_coef
      compute_deterministic_part(leader, current_gradient, inv_current_hessian, deterministic_part,
     rand_sigma, anneal_coef);
    }

    //now compute the stohastic part of the equation
    for (i = 0; i < PROBDIM; i++){
  for (j = 0; j < PROBDIM; j++){
      sigma_nrm[i*PROBDIM+j] = anneal_coef * inv_current_hessian[i][j]; // the hessian and its inverse are
       NOT multiplied by
  }
    }
    //draw a sample from multivariate normal with given mean and sigma
    mvnrnd(mean_nrm, (double *)sigma_nrm, stohastic_part, PROBDIM);

    //for each problem dimension compute the candidate point
    for( i=0; i < PROBDIM; i++){
      candidate[i] = deterministic_part[i] + rand_sigma * stohastic_part[i];
    }
}
//===========================================================//
//===========================================================//

//~~~~~~~~~~~~~~~~~~~~~~~MCMC with NEWTON UPDATE~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//

void chaintask(double in_tparam[PROBDIM], int *pdim, int *pnsteps, double *out_tparam, int winfo[4])
{
  int i,step;
//  twork_t *chainwork = (twork_t *) arg;
//  twork_t *work_table, *work;
  int j;
//  int chain_id;
  int dim = *pdim;
  int nsteps = *pnsteps;  //chainwork->nsteps;
  int gen_id = winfo[0];
  int chain_id = winfo[1];

//  psthread_t thr;
  long me = torc_worker_id();//psthread_current_vp();

  //damianos
  double current_gradient[PROBDIM] = {0.0};           // 1st derivatives on the leader point
  double current_hessian[PROBDIM][PROBDIM] = {{0.0}};        // hessian
  double inv_current_hessian[PROBDIM][PROBDIM] = {{0.0}};   // inverse of hessian
  double deterministic_part[PROBDIM] = {0.0}; //deterministic part of equation (2.1)

  double leader[PROBDIM], fleader, fpc_leader;      // fold
  double candidate[PROBDIM], fcandidate, fpc_candidate; // fnew
  double q_xk_y, q_y_xk;
  /*logspace(-2,0,5)
  ans = 0.010000   0.031623   0.100000   0.316228   1.000000
  */
  double rand_sigma = 1.5 * sqrt( 3.68 / cbrt(PROBDIM) );//sqrt( 2.72 / cbrt(PROBDIM) );//0.03;//sqrt( 1.0
     / sqrt(PROBDIM) );//sqrt(1.0/PROBDIM);//sqrt(1.44/cbrt(PROBDIM));//1.0;//sqrt( 2.88 / cbrt(PROBDIM) )
    ; //1.26

  int row,column;

  for (i = 0; i < PROBDIM; i++) leader[i] = in_tparam[i]; //chainwork->in_tparam[i];  // get initial
     leader
  fleader = *out_tparam; //chainwork->out_tparam[0];                 // and its value
  fpc_leader = posterior(leader, PROBDIM, fleader);

  //damianos
  //at first turn the leader is "new" so flag is TRUE
  int leader_changed_flag = 1;
  update_curgen_db(leader, fleader);

  for (step = 0; step < nsteps; step++) {
    //damianos
    double pj = runinfo.p[runinfo.Gen]; // TODO: MPI, access to global memory !!

    //compute the proposed sample by the stohastic Newton algorithm
    newton_compute_candidate(candidate, leader, current_gradient, current_hessian, inv_current_hessian,
     deterministic_part, rand_sigma, pj, leader_changed_flag);
    evaluate_F(leader, &fleader, me, gen_id, chain_id, step, 1);
    evaluate_F(candidate, &fcandidate, me, gen_id, chain_id, step, 1);

     {
    double f = 0.0025;
    double L, P;
    double pj = runinfo.p[runinfo.Gen]; // TODO: MPI, access to global memory !!
    //damianos
    double log_target_product_moving_xk_y;
    double log_target_product_moving_y_xk;
    double L2;

    //damianos
    //CHANGE IT A BIT!
    //compute target_function(leader) and target_function(candidate)

    //compute the probability of moving from the current sample to the proposed and from the proposed to
     the current
    compute_moving_probab(candidate, leader, current_gradient, current_hessian, inv_current_hessian,
     rand_sigma, pj, &q_xk_y, &q_y_xk);

    log_target_product_moving_xk_y =  ( pj *fleader + log(q_xk_y) );
    log_target_product_moving_y_xk =  ( pj *fcandidate + log(q_y_xk) );
    L2 = exp( (log_target_product_moving_y_xk - log_target_product_moving_xk_y) );

```

```
    L = L2;
    if (L > 1) L = 1;
    P = uniformrand(0,1);
    if (P < L) {
      for (i = 0; i < PROBDIM; i++){
        leader[i] = candidate[i]; // new leader!
      }
      fleader = fcandidate;

      torc_update_curgen_db(leader, fleader);
      leader_changed_flag = 1;//we have new leader update the flag in order to update the gradient and the
     hessian matrix

    }
    else {
      torc_update_curgen_db(leader, fleader);
      leader_changed_flag = 0; //as we did not change the leader, we can keep the already calculated
   gradient and hessian
    }

      }
  }

}
```

References

[T. Bui-Thanh and O. Ghattas] T. Bui-Thanh and O.Ghattas A scaled stohastic Newton algorithm for Markov chain Monte Carlo simulations SIAM Journal on Uncertainty Quantification, submitted (2012).

[N. Metropolis et al.] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller Equation of state calculations by fast computing machines The journal of Chemical Physics, 21, pp. 1087-1092, (1953).

[W. K. Hastings] W. K. Hastings Monte Carlo sampling methods using Markov chains and their applications Biometrika, 57, pp.97-109, (1970)

[Gareth O. Roberts and Richard L. Tweedie] Gareth O. Roberts and Richard L. Tweedie Exponential convergence of Langevin distributions and their discrete approximations Bernoulli, 2, pp. 341363 , (1996).

[L.R. Schaeffer] L.R. Schaeffer Modification of negative eigenvalues to create positive definite matrices and approximation of standard errors of correlation estimates (2010).

[P. E. Hadjidoukas, E. Lappas and V. V.Dimakopoulos] P. E. Hadjidoukas, E. Lapppas and V. V.Dimakopoulos A Runtime Library for Platform-Independent Task Parallelism 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing (2012).

[P. E. Hadjidoukas et al.] P. E. Hadjidoukas, C. Voglis, V. V. Dimakopoulos, Isaac E. Lagaris and Dimitris G. Papageorgiou High-Performance Numerical Optimization on Multicore Clusters 17th International Conference, Euro-Par 2011 Parallel Processing