FAMOSO - A Fast Monte Carlo Solver*

C. del-Castillo-Negrete†

Yale University

New Haven, CT, 06511

Abstract

Particle-based Monte Carlo (MC) simulations are a powerful technique to solve a wide range of problems in physics, chemistry, and engineering. These methods seek to compute the particle probability density function of a system by numerically solving a very large set of stochastic differential equations determining the evolution of the particles. Despite their usefulness, MC methods face two potential limitations. First, noise is introduced because Monte Carlo simulations work with a limited statistical sampling of the exact distribution of particles. To reduce noise due to sampling, a very large number of particles must be used and this can render the method computationally very expensive. Second, MC methods usually exhibit slow convergence towards a steady state solution. To address these limitations, in this paper we propose, implement, and test a novel Projective Monte Carlo (PMC) algorithm to accelerate and denoise particle based simulations. The algorithm is based on tensor product decomposition (TPD) techniques combined with projective integration to accelerate convergence. By recognizing that MC simulation data can be considered a video describing the evolution of the particle system in time, the PMC algorithm constructs an optimal basis that captures the major variability of the simulation data via a TPD that is traditionally used to compress video data. Using this TPD, the PMC algorithm projects an estimate of the distribution function at a later time to accelerate convergence. This paper aslo presents an implementation of the PMC algorithm in the library of Fortran modules called FAMOSO (**FA**st **MO**nte-carlo **SO**lver). FAMOSO was designed to be flexible and adaptable to any generic Monte Carlo particle simulation problem. The results of applying the routines in FAMOSO to a collisional transport problem in plasmas physics is also presented to demonstrate how the algorithms in FAMOSO perform.

---

* Undergraduate Thesis Presented to Yale Computer Science Department May 2015

†Electronic address: carlos.del-castillo-negrete@yale.edu,cdelcastillo21@gmail.com

I. INTRODUCTION

Particle-based Monte Carlo simulations are well-known powerful techniques to solve a wide range of problems in physics, chemistry, and engineering. These systems usually consist of a very large number of particles (e.g., a gas or a plasma) subject to external forces (e.g., electric fields) and mutual interactions (e.g., collisions). The ultimate goal of these methods is to compute the single-particle probability density function that determines the number of particles in a given region of the phase space at a given time. From the physics standpoint, all the macroscopic behavior of the system can be determined from the behavior of this density function and its statistical moments.

The Monte Carlo (MC) particle model is based on the numerical solution of a very large system of stochastic (Langevin) differential equations [4], one for each particle degree-of-freedom. This method faces two potential limitations. First, noise is introduced because the particle model works with a limited statistical sampling of the exact distribution. To reduce noise due to sampling a very large number of particles must be used and this can render the method computationally very expensive. A second related limitation of MC methods is the slow convergence in time towards the desired solution. In particular, reaching an statistical equilibrium state in a MC simulation usually requires very long (compared to the typical collision time scale) integration times. To address these limitations, in this paper we propose, implement, and test a novel Projective Monte Carlo (PMC) algorithm to accelerate and denoise collisional transport calculations. The algorithm is based on tensor product decomposition (TPD) techniques combined with projective integration to accelerate convergence. An implementation of the PMC algorithm in the FAMOSO (**FA**st **MO**nte-carlo **SO**lver) Fortran95 code is also presented.

Previous work on this area include Ref. [1] where TPD methods were used to passive noise reduction in MC simulation. On the other hand, projective methods has been used in different settings to accelerate computations see for example Ref. [2] and references therein. Going beyond Ref. [1] where denosing was used *passively* as an a *post-processing* tool to clean the noise after the simulation was done, in this paper we use TPD methods in *active denoising* that is to clean the noise during the calculation. Most importantly, going beyond previous work on projective methods, here we use for the first time TPD techniques to construct the coarse grained degrees of freedom needed for the projective step.

The rest of this paper is laid out as follows. Section II presents the proposed PMC algorithm. Section III describe the FAMOSO code and the routines implementing the PMC algorithm. Section IV presents an application of FAMOSO to simulate collisional transport in plasmas. Section V contains concluding remarks.

II. PROJECTIVE MONTE CARLO (PMC) ALGORITHM

This section presents an overview of the steps involved in the proposed projective Monte Carlo (PMC) algorithm. As illustrated in Fig.(1), the goal of this algorithm is to accelerate the convergence towards the final equilibrium state via repeated iterations of MC integrations steps followed by projective leaps.

A. Particle Data Initialization

The starting point of this method is an initial particle probability density function, f_0, on a grid. To begin, the PMC algorithm initializes particle data with coordinates using the initial configuration f_0. This is done by a statistical sampling of the initial distribution function . Note that the number of particles N_p used in the sampling directly influences the amount of statistical error introduced in the Monte Carlo method.

B. Numerical Integration and Data Recording

After the particle data initialization, the PMC algorithm performs a direct Monte-Carlo integration. As mentioned earlier, Monte-Carlo simulations model the microscopic dynamics of each particle according to a set of stochastic differential equations. The general form of the stochastic differential equations that a Monte-Carlo routine uses to update each particles position at each time step is:

$$d\mathbf{X} = \mathbf{A}dt + \mathcal{B}d\mathbf{W}\,, \tag{1}$$

where $d\mathbf{X}$ is an $N-$dimensional vector denoting the change in the particles coordinates, $\mathbf{A}$ is $N-$dimensional vectors that corresponds to the deterministic forces acting on the particle, $\mathcal{B}$ is an $N \times N$ matrix containing the $\mathbf{X}$-dependent particle collision frequencies, and $d\mathbf{W}$ is an $N-$dimensional vector of independent Wiener processes. Further details on

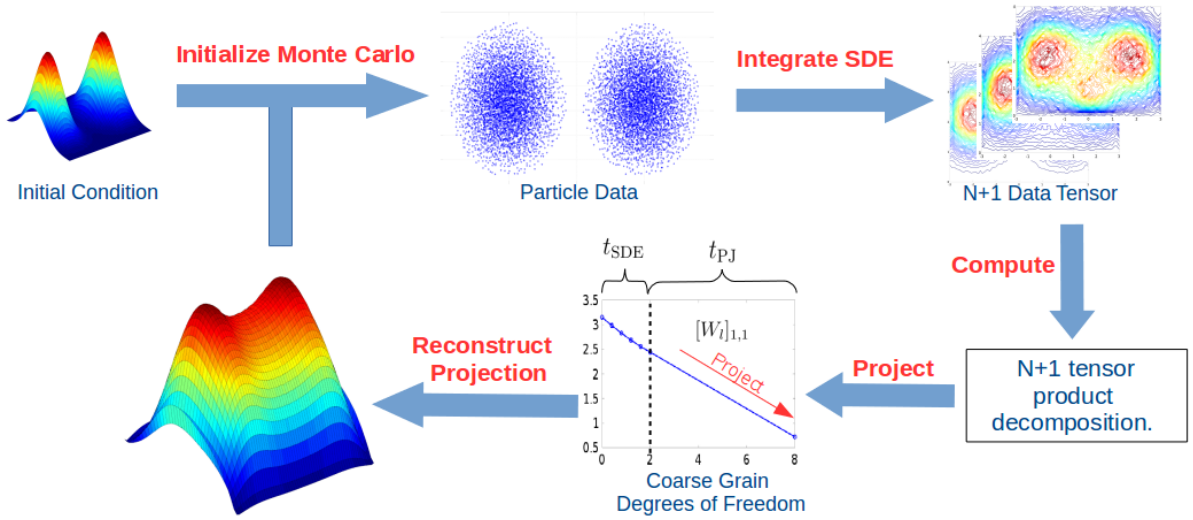

FIG. 1: Schematic diagram of the projective Monte Carlo method.

the formulation of these stochastic differential equations can be found in Ref. [4]. Using Eq.(1), the PMC integrates the coordinate of each particle from $t_0 = 0$ to some final time $t = t_{\mathrm{SDE}}$. Since the stochastic differential equations model the specific dynamics of each particle as an independent random process, this task is computationally simple yet can take a very long time because of the potentially large number of particle positions that need to be updated at each time step.

During the MC integration phase of the PMC algorithm, histogram-based estimations of the particle distribution function are computed at a given number of intermediate times and are collected into a $(N+1)$-dimensional tensor, where N is the number of dimensions of the system (i.e. the number particle degrees-of-fredom). Given an $N_{x_1} \times N_{x_2} \times ... \times N_{x_n}$ spatial grid with nodes located at $(X^{(1)}_{i_1}, X^{(2)}_{i_2}, ..., X^{(n)}_{i_n})$ where $i_j = 1, 2, ...N_{x_j}$, the histogram estimation of the particle distribution function at a given time is the N order tensor F giving the fraction of particles with coordinates $(x_1, x_2, ..., x_n)$ such that $X^{(1)}_{i_1} \leqslant x_1 \leq X^{(1)}_{(i_1+1)}$, $X^{(2)}_{i_2} \leqslant x_2 \leq X^{(2)}_{i_2+1}$, ..., and $X^{(j)}_{i_n} \leqslant x_j \leq X^{(j)}_{(i_n+1)}$. Therefore collecting a series of these N order tensors give us an $(N+1)$-dimensional tensor F_l (l here denoting the time dimension) that contains data describing the dynamics of the distribution function over the time interval $[t_0, t_{SDE}]$.

C. N+1 Tensor Product Decomposition

Following the MC direct integration, we construct an optimal basis that captures the main dynamics of the distribution function recorded in the data tensor F_l. Considering the data set as a $(N+1)$-dimensional tensor, the $+1$ dimension being time, the PMC method seeks via tensor decomposition a time independent basis that captures the main variability in the whole collection of $N-$dimensional tensors. Mathematically, the PMC algorithm seeks a decomposition that minimizes the reconstruction error

$$e(r_1, r_2, ..., r_n) = \sum_{l=1}^{N} ||F_l - UW_l V^{\mathrm{T}}|| \tag{2}$$

where $r_1, r_2, ..., r_n$ are the rank truncations used in the decomposition, U and V are $N-$dimensional tensors that describe the optimal basis, W_l is the compressed form of each F_l, and the sum is done over the index l - time. Note that the key idea to finding a decomposition that minimizes the error en Eq.(3) as opposed to one that minimizes the error,

$$e(r_1, r_2, ..., r_n) = ||F - UWV^{\mathrm{T}}||, \tag{3}$$

is that the error that is being minimized is over *a collection of tensors*. The difference between these two reconstruction errors is analogous to the difference between compressing a video as opposed to compressing a single image. A "video" in this context is a high order image (the extra dimension being time) that can be compressed using the minimization problem posed in Eq.(3). However, as first noted in Ref. [3] the benefits of posing the minimization problem in Eq.(2) is that the minimization problem inherently favours one dimension - time. Since the video data is inherently correlated along the time dimension, minimizing over the error in Eq.(2) is a better form of compression because it inherently incorporates correlations that already exist in the data due to high spatio-temporal correlation that exists in video data.

General closed form optimal solutions for the minimization problems posed in Eq.(2) and Eq.(3) do not exist for the general $N-$dimensional tensor decomposition problem. When $N = 2$, Eq.(3) has the well-known closed form solution known as the Singular Value Decomposition of the matrix F, and for $N > 2$ decompositions known as higher order SVDs have been studied but are not proven for optimality [6]. The general-$N+1$ minimization problem

in Eq.(2) seems to be a much less studied problem in the Machine Learning literature. To the best of our knowledge it has only been addressed in detail for the $N+1=2+1$ case in Ref. [3] that presented an iterative algorithm for minimizing the error in Eq.(2) and to construct a decomposition of the form:

$$f_{lij}{}^{(r_1,r_2)} = \sum_{k_1=1}^{r_1}\sum_{k_2=1}^{r_2}[W_l]_{(k_1,k_2)}u_i{}^{(k_1)}v_j{}^{(k_2)} \tag{4}$$

where $u_i{}^{(k)}$ and $v_j{}^{(k)}$ are time l-independent orthonormal vectors, and all the time dependence is incorporated in the family of $r_1 \times r_2$ weight matrices W_l. Given the lack of a general algorithm to solve the $(N+1)$ tensor decomposition problem, the current PMC algorithm implemented in FAMOSO is restricted to two dimensional particle simulations where the (2+1)-tensor decomposition is computed using the GLRAM (Generalized Low Rank Approximation of Matrices) iterative algorithm proposed in Ref. [3].

D. Projecting Coarse Grained Degrees of Freedom

The key to the projective step is to use the set W_l of $r_1 \times r_2$ matrices to define a new set of "coarse grained" degrees of freedom of the system. Trends in the simulation data, and hence in the dynamics of the system, are represented most effectively by these degrees of freedom because they were constructed to capture maximum variability in the data set according to the reconstruction error in Eq.(2). Therefore the PMC method can project the current trends exhibited by the W_l coefficients to predict the distribution function of the system in the future. There are several options for this projection scheme. For illustration purposes the one currently implemented in FAMOSO is a simple linear extrapolation of the form

$$W_{k_1,k_2}(t+\Delta t) = W_{k_1,k_2}(t) + s\Delta t \tag{5}$$

applied to each coarse grained degree of freedom. In Eq.(5) Δt refers to the length of the projection and s is the slope of the line joining the last two values of W_{k_1,k_2} in the collection of matrices W_l. However note that by construction some elements W_{k_1,k_2} might exhibit small fluctuations, relatively to the other entries. A blind linear extrapolation may project these fluctuating degrees of freedom and amplify what is noise in the data set. Therefore before projecting the coarse grained degrees of freedom, a filtering process is done on the coefficient

matrices W_l based on an energy threshold defined as:

$$\frac{||W_{filtered}||^2}{||W||^2} \leqslant E_{th}. \tag{6}$$

$W_{filtered}$ is constructed by starting with an empty matrix and adding on the original elements of the original W matrix in decreasing order until the ratio of the norms exceeds E_{th}. Therefore the E_{th} parameter of the PMC algorithm controls how many of the coarse grained degrees of freedom are actually projected.

E. Reconstructing Projection and Re-initialize Particle Data

Having projected along the coarse grained degrees of freedom, the PMC algorithm reconstructs the resulting particle distribution function using:

$$f_{ij}{}^{(r_1,r_2)}(t = t_{SDE} + t_{PJ}) = \sum_{k_1=1}^{r_1} \sum_{k_2=1}^{r_2} [W_{projected}]_{(k_1,k_2)} u_i{}^{(k_1)} v_j{}^{(k_2)} \tag{7}$$

where $W_{projected}$ is the weight matrix obtained from linear extrapolation. The goal of this projective step is to predict the future state of the distribution function of the system and thus to accelerate convergence to the final equilibrium state. In order to determine if the equilibrium state has been reached, the PMC method re-initializes the standard Monte Carlo method by sampling the projected distribution function and generating a new set of particle coordinates. By then restarting the standard Monte Carlo solver, it can be determined if a steady state solution has been reached by analysing the net change in the estimations of the distribution function that are recorded in the $(N + 1)$ data tensor. If the data still exhibits significant variability, the PMC method restarts by performing another projective step. Otherwise the PMC algorithm has converged to a steady state and has found a numerical result for the final distribution function of the system as desired.

III. FAMOSO - FAST MONTE-CARLO SOLVER

The main contribution of this project is the development of a set of Fortran95 modules called FAMOSO (a **FA**st **MO**nte-Carlo **SO**lver). The design paradigm of FAMOSO is to build easy to use data structures and functions to make the implemented algorithms user-friendly. In particular, an effort was made to remove the routines implemented in FAMOSO

from any dependencies on the specific particle simulation problem in question. As a result, FAMOSO may be used as a Monte Carlo solver in generic particle dynamics problems formulated in terms of stochastic differential equation of the form in Eq. (1).

FAMOSO is centered around two main modules - montecarlo.mod and projective_montecarlo.mod - that contain routines implementing the standard and projective Monte-Carlo simulation routines, and several other supporting modules. What follows is a brief discussion of the main components of FAMOSO.

A. Data Representations - mc_reps.mod

The mc_reps module contains the data type definitions of the different data representations used in the FAMOSO solver routines. As described earlier, the two types of data representations that can describe the state of a particle simulation at a given time are (1) a particle distribution function f defined over a grid and (2) the coordinates $\{p_i\}$ of the N_p particles in the simulation. In FAMOSO these two basic data are represented in the particle_rep and stat_rep data types respectively as follows:

```
! Defines Particle Representation of Simulation Data
type particle_rep
   integer :: np                                  ! # of particles
   integer :: nd                                  ! # of dimensions
   real(8), dimension(:,:), allocatable :: p_i ! Coordinates Array
end type particle_rep

! Defines Statistical Representation of Simulation Data
type stat_rep
   type(grid) :: gd                                ! Grid over which f_yx is defined
   real(8), dimension(:,:), allocatable :: f_yx ! Distribution function as matrix
end type stat_rep
```

Listing 1: Simulation Data Representations

While the standard Monte Carlo solver strictly speaking deals only with particle data representations, the data compression and projection algorithms deal with matrices and hence statistical representations of the system. Accordingly, FAMOSO contains the routines mc_f_to_xy() and mc_xy_to_f() in the mc_reps module to convert between the two data representations. These functions have the following calling conventions:

```
! Constructs a monte-carlo particle representation p_i of np particles
    corresponding to a statistical representation stored in s_rep.
function mc_f_to_xy(s_rep, np)
   type(particle_rep) :: mc_f_to_xy
   type(stat_rep) :: s_rep
   integer :: np
        ....
end function mc_f_to_xy

! Constructs a statistical representation consisting of a distribution function
    f_yx defined over a given grid gd corresponding to the particle data p_rep.
function mc_xy_to_f (p_rep, gd)
   type(stat_rep) :: mc_xy_to_f, s_rep
   type(particle_rep), intent(IN) :: p_rep
   type(grid), intent(IN) :: gd
   ...
end function mc_xy_to_f
```

Listing 2: Simulation Data Conversion Routines

All the data types have been generalized to be able in principle to store an arbitrary number of dimensions. However, the current version of FAMOSO supports only two-dimensional simulation problems. As mentioned before, this limitation is primarily due to the lack of an algorithm to compute the general $(N+1)$ TPD.

B. Standard MC Solver - montecarlo.mod

The montecarlo module is centred around the mc_problem data type that defines the parameters of a Monte Carlo simulation. This data type has the following form and corresponding constructing function:

```
type mc_problem
   type(grid) :: gd                        ! Spatial Grid
   procedure(func_x_i), pointer, nopass :: f_init  ! Analytical Init Dist.
   procedure(stoch_diff_dx), pointer, nopass :: dX ! SDE equations
   real(8) :: t_i, t_f, delta_t            ! Time step and init/final time.
   type(stat_rep) :: s_rep_init            ! Last state of prev. simulation
   type(mc_sim_data) :: mc_data            ! Simulation data
end type mc_problem

! Constructor function
function mc_prob_construct(t_i, t_f, delta_t, gd, f_init, dX)
```

Listing 3: Standad Monte Carlo Problem Structure

The mc_problem structure contains two function pointers: f_init, which is a pointer to a function that describes the initial particle distribution function of the system, and dX that points to a function that defines the stochastic differential equations in Eq.(1). These functions have the form:

```
! Function prototype defining general function f(x_1,x_2,...,x_nd)
function func_x_i (nd, args)
   real(8) :: func_x_i                ! Result of f(arg(1),arg(2),...)
   integer, intent(IN) :: nd             ! # of dimensions
   real(8), dimension(nd), intent(IN) :: args ! Argument list
end function func_x_i

! Function prototype for general stochastic differentila equation
function stoch_diff_dx (nd, delta_t, x_i, d_eta)
   integer, intent(IN) :: nd               ! # of dimensions
   real(8), dimension(nd) :: stoch_diff_dx    ! result vecor dX
   real(8), intent(IN) :: delta_t          ! time step
   real(8), dimension(nd), intent(IN) :: x_i, d_eta ! Position and random #s
end function stoch_diff_dx
```

Listing 4: Initial Distribution and Stochastic Differential Equation Function Prototypes[caption

Since these two functions are dependent upon the particular dynamics of the Monte Carlo simulation, they are left up to the user to define in an external function and link via the function pointers in the mc_problem structure. An example of how this can be done follows:

```
program test_mc
   ! Initialize parameters
   ...

    ! Linking procedure pointers with external functions. These can be in
    ! the same file or in separate files.
    procedure(func_x_i), pointer :: f_init=>f_init_example
    procedure(stoch_diff_dx), pointer :: f_dX=>dX_example

    ! Construct montecarlo problem structure within defined parameters,
       functions and initial grid
    mc_prob = mc_prob_construct(t_init, t_final, delta_t, init_gd, f_init, f_dX)

    ! Perform some computation
    ...
end program test_mc
```

Listing 5: Linking externally defined functions to mc_prob structure.

The primary solver routine of the montecarlo module is mc_solve(). This module implements a direct Monte Carlo algorithm and returns the final distribution function in a stat_rep structure. The montecarlo module also contains an mc_solve_and_record() routine that performs the exact same function as mc_solve() except that it records a specified number of histogram estimations of the particle distribution function of the Monte Carlo simulation as described in step **C** of the PMC algorithm. The data that the mc_solve_and_record() routine records is stored in an mc_sim_data structure that is part of the mc_problem structure. These functions and the mc_sim_data structure have the form:

```
1  type mc_sim_data
     integer :: N                                ! # data points
3    type(stat_rep), dimension(:), allocatable :: s_rep_i  ! Array of dist. funcs.
     real(8), dimension(:), allocatable :: t_i        ! Time each s_rep_i taken at
5    real(8), dimension(:), allocatable :: cpu_t_i    ! CPU runtime for iteration i
   end type mc_sim_data
7
   ! Returns final distribution function of simulating Monte-Carlo
9  ! problem contained in mc_prob with np particles.
   function mc_solve(mc_prob, np)
11     type(stat_rep) :: mc_solve
       type(mc_problem), intent(IN) :: mc_prob
13     integer, intent(IN) :: np
       ...
15 end function mc_solve

17 ! Runs Monte-Carlo simulation problem contained in mc_prob with np
   ! particles. Records N snapshots of particle distribution function
19 ! at equally spaced intervals in the simulation and stores in mc_prob%mc_data
   subroutine mc_solve_and_record(mc_prob, np, N)
21     type(mc_problem), intent(INOUT) :: mc_prob ! Parameters of simulation
       integer, intent(IN) :: np, N         ! # particles, # data points, resp.
23     ...
   end subroutine mc_solve_and_record
```

Listing 6: Standard Monte Carlo solver routines

The arrays t_i and cpu_t_i (lines 4-5) of the mc_sim_data structure contain the time at which each histogram estimation was constructed in the simulation and the amount of computer time spent integrating the stochastic differential equations to reach that particular state from the last histogram estimation constructed.

C. N+1 Tensor Product Decomposition - glram.mod

Currently FAMOSO has one algorithm for $(N+1)$ TPDs. As mentioned before, this algorithm is limited to the $N = 2$ case and is explained in more detail in Re. [3]. In FAMOSO this algorithm is incorporated in the glram module (after the name of the original algorithm - Generalized Low-Rank Approximation of Matrices [3]). The data structure glram_reps contains all the components related to a (2+1) TPD - the matrix sizes, the transformation matrices, and the weight matrices. The two main functions that are called in the PMC algorithm are glram_construct() and glram_reconstruct. These components of the glram module have the form:

```
! A full data set of N rXc matrices A_i can be
! compressed into this data structure using GLRAM decomposition
type glram_rep
    integer :: r, c, N, l1, l2              ! Matrix sizes
    real(8), dimension(:,:), allocatable :: LT, RT  ! Transformation matrices
    real(8), dimension(:,:,:), allocatable :: M_i  ! Weight matrices
end type glram_rep

! Returns glram_rep structure containing the compressed representation
! of the N, rXc matrices stored in A_i. Compressed matrices are l1xl2
function glram_construct( A_i, r, c, N, l1, l2)
    type(glram_rep) :: glram_construct
    type(glram_rep) :: g_rep
    integer, intent(IN) :: r, c, N, l1, l2
    real(8), dimension(r,c,N), intent(IN) :: A_i
...
end function glram_construct

! Reconstructs the given weight matrix M given transformations in g_rep
function glram_reconstruct(M, g_rep)
    real(8), dimension(:,:), allocatable :: glram_reconstruct ! Reconstructed
       Matrix
    real(8), dimension(:,:), intent(IN) :: M        ! Coefficient Matrix
    type(glram_rep) :: g_rep                      ! TPD decomposition
...
end function glram_reconstruct
```

Listing 7: glram type and corresponding TPD routines

glram_construct() contains the actual algorithm to compute the TPD. The algorithm itself reduces essentially to computing matrix multiplications and SVD decompositions. FAMOSO uses the extensively used LAPACK routines to perform these matrix computations. For more

information about the LAPACK and BLAS libraries see Ref. [5]. glram_reconstruct is used to reconstruct a given weight matrix given a glram_rep structure that contains the basis vectors of the TPD. This routine is primarily used in the reconstruction of the projected distribution function.

D. PMC Algorithm - projective_montecarlo.mod

The projective_montecarlo module is centered around the pjmc_problem data type that defines the parameters and dynamics of a projective Monte Carlo simulation problem. This data type has the following form and corresponding constructing function:

```
type :: pjmc_problem
   ! Same parameters as mc_prob
   ...
   ! PMC Algorithm Parameters
   integer :: num_iter      ! Num iterations
   real(8) :: pj_ratio      ! Ratio of time spent projecting
   real(8) :: pj_aff        ! 1/(# of data points in sim_data)
   real(8) :: e_thresh      ! Projection threshold
   type(glram_rep) :: g_rep   ! N+1 tensor decomposition of data.
end type pjmc_problem

! Returns pjmc_problem structure with given parameters
function pjmc_prob_construct(t_i, t_f, delta_t, num_iter, &
        &gd, f_init, dX, pj_ratio, pj_aff, e_thresh)
```

Listing 8: PMC problem data type and constructor function.

The pjmc_problem structure contains all the elements of a mc_problem structure along with the parameters of the PMC algorithm. The main routine of the projective_montecarlo module is pjmc_solve(). A simplified version of this routine follows:

```
subroutine pjmc_solve(pjmc_prob, np, r1, r2, max_gd)
...
type(pjmc_problem), intent(INOUT) :: pjmc_prob
integer, intent(IN) :: np, r1, r2
type(grid), optional, intent(IN) :: max_gd
...
! Set parameters
time_interval = (pjmc_prob%t_f-pjmc_prob%t_i)/real(pjmc_prob%num_iter)
n_data = int(1.0/pjmc_prob%pj_aff)
t_i_mic = pjmc_prob%t_i
t_f_mic = pjmc_prob%t_i + time_interval*(1.0-pjmc_prob%pj_ratio)
```

```
12 t_proj = pjmc_prob%t_i + time_interval
   ...
14 ! Main loop of algorithm
   do i=1,num_iter
16     ! Integrate MC. Record snapshots for projection
       pjmc_prob%t_i = t_i_mic
18     pjmc_prob%t_f = t_f_mic
      pjmc_microscopic_integrate(pjmc_prob, np, n_data, cpu_proj_time)
20
       ! Check if projection step necessary
22     if (t_proj > t_f_mic) then
           ! If it is, calculate projection using glram decomposition
24         pjmc_prob%s_rep_init = pjmc_project_glram(pjmc_prob%mc_data,&
                        &t_proj,n_data,r1,r2,pjmc_prob%e_thresh)
26     else
           ! If not, just copy last stored distribution function for
28         ! re-initialization next iteration -> no projection done.
           pjmc_prob%s_rep_init =
              mc_copy_srep(pjmc_prob%mc_data%s_rep_i(pjmc_prob%mc_data%N))
30     end if

32     ! Set new time interval for next iteration
       t_i_mic = t_i_mic + time_interval
34     t_f_mic = t_f_mic + time_interval
       t_proj = t_proj + time_interval
36 end do
   end subroutine pjmc_solve()
```

Listing 9: Implementation of PMC algorithm

The PMC algorithm operates through successive iterations of direct Monte Carlo integrations and projections (see Fig. 1). The subroutine pjmc_microscopic_integrate() (line 19 above) is identical to the mc_solve_and_record() subroutine from the montecarlo module and serves the purpose of iterating the standard Monte Carlo and recording simulation data. The function pjmc_project_glram() (line 24) uses the stored simulation data of a given iteration to perform the projective step. This function performs the steps of computing the $(N+1)$ TPD, filtering the weight matrices, projecting, and reconstructing the projected distribution function:

```
1 function pjmc_project_glram( mc_data, t_project, N, l1, l2, e_thresh)
      type(stat_rep) :: pjmc_project_glram, s_rep_proj
3     type(mc_sim_data), intent(INOUT) :: mc_data
      real(8), intent(IN) :: t_project
5     integer(4), intent(IN) :: N, l1, l2
```

```
    real(8), intent(IN) :: e_thresh

   ! Compute glram compressed rep of n_data most recent data points
    ! stored in the mc_data structure. Use r1*r2 coefficient matrices
   g_rep = pjmc_compute_glram(mc_data, l1, l2, N)


    do i=1,g_rep%N
        call mc_e_thresh(g_rep%M_i(:,:,i), l1, l2, e_thresh)
    end do

    ! Project each coefficient according to linear model
    do j=1,l2
        do i=1,l1
            M_proj(i,j) = linear_project_simple( N, t_i, g_rep%M_i(i,j,:),
                t_project)
        end do
    end do

    ! Reconstruct projected matrix
    s_rep_proj%f_yx = glram_reconstruct(M_proj, g_rep)


    ! Normalize
    s_rep_proj%f_yx(:,:) = s_rep_proj%f_yx(:,:)/mc_srep_norm(s_rep_proj)

    pjmc_project_glram = s_rep_proj
end function pjmc_project_glram
```

Listing 10: Implementation of projection routine

Note that although the mc_sim_data structure stores the simulation data from the whole simulation, only the most recent data points from the past MC integration are used to compute the $(N+1)$ TPD. pjmc_compute_glram() (line 10) is a wrapper function for the glram_construct() routine from the glram module and is responsible for extracting the proper information from the mc_sim_data structure to use in the glram TPD. Right after computing the TPD, the weight matrices (stored in g_rep%M_i) are filtered according to the e_threh parameter and Eq.(6) (lines 13-15). Next the function linear_project_simple() applies the linear extrapolation described in Eq.(5) to find the projected weight matrix (lines 17-21). The projected weight matrix is then reconstructed using the glram_reconstruct() function. Finally, the distribution function is normalized and returned as the future projected distribution function. This is stored in the s_rep_init value of the pjmc_problem structure which

tells the routine pjmc_microscopic_integrate() to restart the simulation from this distribution function on the next iteration of the PMC algorithm and not the initial distribution function from the f_init function pointer. On a final note, it must be said that no explicit convergence criteria has been set in the current implementation of the pjmc_solve() routine.

IV. CODE EXAMPLE - COLLISIONAL TRANSPORT PROBLEM

In this section we present numerical results obtained using FAMOSO to solve a collisional transport relaxation problem of interest to plasma physics. Our focus here is not the physics of the problem but the computational performance of the code with respect to the parameters of the projective Monte Carlo algorithm.

A. Problem Formulation

The example pertains a systems with a known equilibrium steady state that physically corresponds to a Maxwell-Boltzmann distribution, f_∞, for the plasma particles. That is, f_∞ is the answer towards the computation should converge. To test convergence, the Monte Carlo simulations were initialized using a far from equilibrium state f_0, and the rate of convergence was measured using the L_2 norm of the difference between the computed distribution function at time t, f_t, and the exact asymptotic final equilibrium state f_∞:

$$||f_t - f_\infty||^2 = \sum_{i,j=1} {(f_i - f_\infty)_{i,j}}^2, \tag{8}$$

The main parameters determining the performance of the PMC algorithm are:

1. The ratio of the projection time to the the total time during an iteration of the PMC algorithm,
$$\mathrm{PJ}_{ratio} = \frac{t_{\mathrm{PJ}}}{t_{\mathrm{PJ}} + t_{\mathrm{SDE}}}\,, \tag{9}$$
where t_{SDE} is the integration time of the stochastic differential equation (1) and t_{PJ} is the size of the projective step.

2. The truncations ranks, (r_1, r_2), used in the N+1 tensor product decomposition used to construct the coarse grained degrees of freedom.

3. The filtering threshold, E_{th}, discussed in Eq. (6).

	Standard MC	Projective MC		
PJ Ratio	0.00	0.25	0.50	0.75
SMC Runtime (sec)	2391.0	1806.8	1139.9	713.85
PMC Runtime (sec)	0.00	236.29	142.39	89.03
TOTAL RUNTIME (sec)	2391.0	2043.1	1282.2	802.87
Speedup	**——**	**1.170**	**1.865**	**2.978**

FIG. 2: Runtime table for Projective Monte Carlo method using 3 different values of PJ_{ratio} and with $n_p = 10^6$, $(r_1, r_2) = (5, 5)$, $\mathrm{PJ}_{ratio} = 0.5$, $E_{th} = 0.9$ (standard Monte Carlo shown with $\mathrm{PJ}_{ratio} = 0.00$).

In all the numerical results presented, the parameters of the direct Monte Carlo integration $\Delta t = 0.005$, $t_0 = 0.0$, $t_f = 40.0$, and $N_p = 10^6$, were kept constant.

B. Dependence of rate of convergence on PJ_{ratio}:

PJ_{ratio} determines for the most part how quickly the PMC routine runs. From the results in the table in Fig. (2) it is clear that, since the projective step of the PMC algorithm is essentially computationally free, the possibilities of accelerating standard Monte Carlo simulations using FAMOUS is tremendous. Furthermore, when looking at the error plot in Fig.(3) it is evident that the PMC routine rapidly converges to the same solution as the standard MC. It is important to note that although the PMC algorithm in some cases (green and blue line) performed "unfavorable" projective steps, it still ended up converging to the same asymptotic state of the standard MC, which is what matters when solving for the steady state equilibrium problem.

C. Dependence of rate of convergence on (r_1, r_2)

The rank truncation choice determines how much of the dynamic particle data information the coarse grained degrees capture - using too high of a rank decomposition may be unnecessary since there may be no higher modes in the data, but using too low of rank decomposition clearly can damage the quality of the projection if these higher modes exist

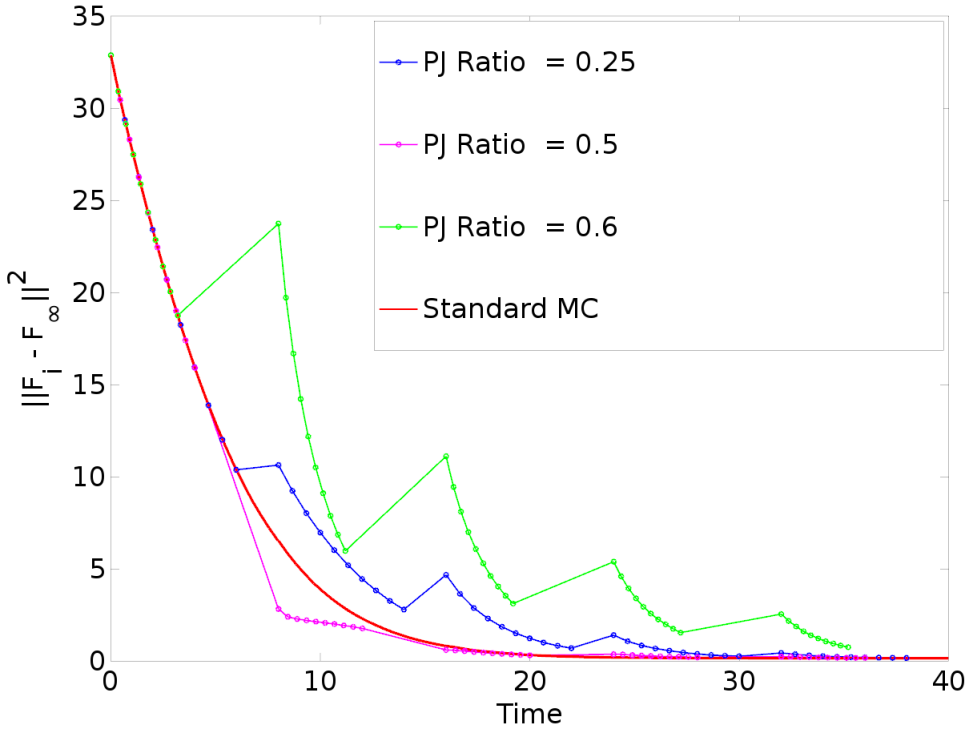

FIG. 3: Error plot for Projective Monte Carlo method using 3 different values of PJ_{ratio} and with $n_p = 10^6$, $(r_1, r_2) = (5, 5)$, $\mathrm{PJ}_{ratio} = 0.5$, $E_{th} = 0.9$ (standard Monte Carlo shown in red). Note that all cases converge to the same final value regardless of the projection ratio used. However the quality of the projective step in pushing the solution towards convergence is highly dependent on the other parameters of the projection.

and the tensor decomposition misses them. This is evident from Fig.(4, right), where the rank $(3, 3)$ projection (green line) is the only case that fails to make a positive projective step.

D. Dependence of rate of convergence on E_{th}

From Fig.(4, left) it is evident that there is an optimal E_{th} for a given PMC problem, holding all other parameters constant. This suggests that there is an optimal filtering threshold that allows the projective step to extrapolate only those coarse grained degrees of freedom that are not noise and actually represent physically relevant aspects of the dynamics of the system. Therefore the quality of the projective step in the PMC algorithm is highly dependent on E_{th}.

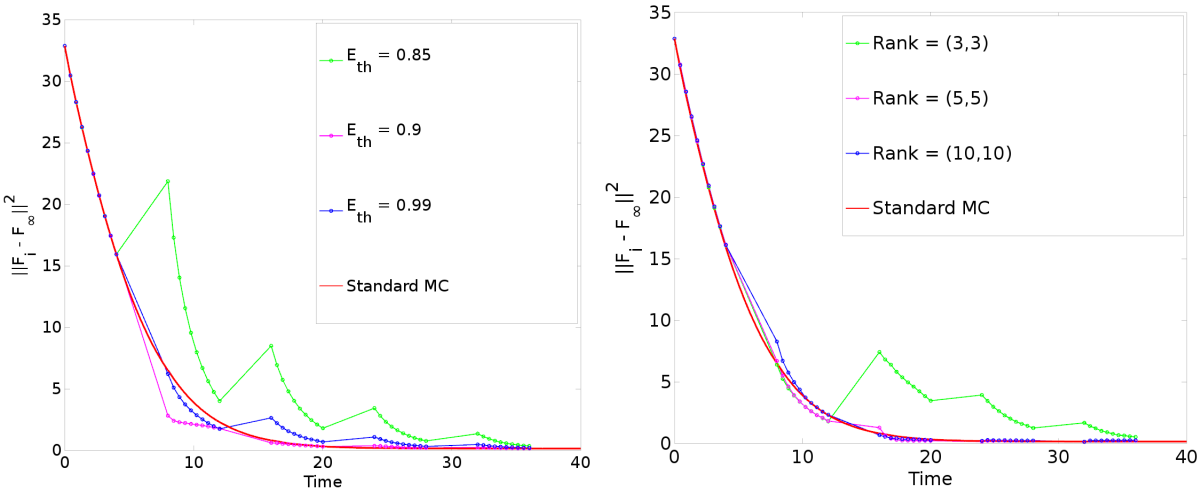

FIG. 4: (LEFT) - Error plot for Projective Monte Carlo method using different values of E_{th} and with $n_p = 10^6$, $(r_1, r_2) = (5, 5)$, and $\text{PJ}_{ratio} = 0.5$ (standard Monte Carlo shown in red). Note the optimal value of E_{th} (magenta line). (RIGHT) - Error plot for Projective Monte Carlo method using 3 different rank truncation values and with t $n_p = 10^5$, $E_{th} = 0.99$, and $\text{PJ}_{ratio} = 0.5$. Note how using too low of a rank truncation may damage the quality of the projective step (green line).

V. CONCLUSIONS

In this paper a novel machine learning based projective Monte Carlo method for solving problems related to particle dynamics was presented. The concept of an $N+1$ tensor product decomposition that seeks to find an optimal, compressed representation of the dynamic state over a preferred dimension (time) was introduced. Recognizing that this compressed representation defines the coarse-grained degrees of freedom that capture the major components of the dynamical system, an algorithm to project these variables to predict the state of the system at a later time was developed. This novel method was incorporated and implemented in the Fortran library called FAMOSO. The major components and routines of FAMOSO were presented and an example of applying FAMOSO to a collisional transport relaxation problem was discussed. The example presented indicates that the PMC algorithm implemented in FAMOSO can be an effective way to accelerate traditional Monte Carlo particle simulations.

Future work will include developing a comprehensive set of heuristics to determine optimal parameters for the PMC method given a particular problem and incorporating these

heuristics into FAMOSO. Another possible area for improvement in FAMOSO could be the development of new algorithms to compute the (N+1) Tensor Product Decomposition for $N > 2$. This will open the possibility of applying FAMOSO to high dimensional particle simulations.

VI. ACKNOWLEDGMENTS

I would like to thank my senior thesis advisor at Yale University, Dr. Sahand Negahban, for his guidance throughout my project. Furthermore, I would like to thank Dr. Diego del-Castillo-Negrete for his help with this project during my visit to Oak Ridge National Laboratory in the Summer of 2014.

---

[1] D. del-Castillo-Negrete, D. A. Spong, and S. P. Hirshman, Phys. Plasmas **15**, 092308 (2008).

[2] I. G. Kevrekidis, C W. Gear and G. Hummer, AIChE Journal **50** 1346 -1355 (2004).

[3] Jieping Ye, Mach. Learn. **61** 167 (2005).

[4] C.W. Gardiner, *Handbook of Stochastic Methods*. (Springer-Verlag, Berlin Heidelberg, 2004).

[5] E. Anderson, et al, *1999 LAPACK Users' Guide* (Third Ed.). Soc. for Industrial and Applied Math., Philadelphia, PA, USA 1999.

[6] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000. SIAM J. Matrix Anal. Appl. **21**, 4 (March 2000), 1253-1278.