MSD Version 7

Rewriting the MSD software in Assembly and Python

Christopher D'Angelo

January 20, 2026

Center for Nanotechnology Research and Education

University of the District of Columbia

Abstract

This work presents Version 7 of the Molecular Spintronics Device (MSD) simulation software, a complete redesign of the computational engine using Python-driven code generation and x64 assembly. The new framework implements a Modified Classical (MC) Heisenberg model with fully customizable graph-based molecular structures, supporting all existing MSD parameters while enabling long-requested features such as dipolar coupling, layered electrodes, material defects, and non-cubic lattice geometries. Performance-critical components of the Metropolis algorithm—including pseudo-random number generation, random vector sampling, energy-difference evaluation, and Boltzmann acceptance—are re-implemented using AVX2-optimized assembly. A vectorized xoshiro256** PRNG, Marsaglia-based unit-vector generator, and a custom lookup-table-accelerated logarithm routine collectively reduce computational overhead by an order of magnitude. Initial benchmarks show a ~10× speed improvement over the previous C++ implementation. This architecture allows the software to generate configuration-specific native code, eliminating unnecessary branching and memory operations while improving extensibility for future physical models.

Background

The current version of the MSD software, version 6.3.4 published 7-2-2025 (D'Angelo), and its predecessors, have been used in numerous published and unpublished research studies for CNRE (TODO: Source?), and in educational contexts, e.g. MECH 500 Research Methods & Tech Communications (Course description and syllabus). It has been proven reliable, consistent with experimental observations (TODO: Source?), and useful for gaining a theoretical understanding of molecular spintronics.

The earliest versions of the MSD software were developed in 2011. The oldest version still available online, v2.1.3, was developed in 2014. Over the decade and a half that we have been developing this software, there have been numerous new features, and optimizations. In 2011, the first version was a 2D Ising Model which supported 7 (simulation) parameters kT, JL, JR, Jm, JmL, JmR, and B, grouped into 3 subsets: kT, J, and B. The current version models a 3D cross-shaped classical Heisenberg Model with free electrons, fully customizable graph-based molecule configurations, 44 (simulation) parameters grouped into 11 subsets: kT, S, F, J, Je0, Je1, Jee, B, A, b, and D.

There were at least two notable optimizations over the history of the MSD software:

1. Using only localized energy updates, ΔU, in the metropolis algorithm. This led to a roughly 30 times[1] improvement in computational speed.
2. Using a custom sparse array data structure (SparseArray.h) for storing the model's state instead of the C++ std `map`[2]. This led to a roughly 2 times improvement in computational speed.

Motivation

There are at least two features we have wanted to add to the software for a while:

1. Dipolar coupling. This is a long-range effect between nodes who are not just nearest-neighbors. It involves both a parameter coefficient and a distance metric (e.g. Euclidean distance) between nodes.
   We have avoided adding this energy term so far since, due to the current software algorithmics which will be discussed below, even when not being used, just allowing for the possibility of long-range effects would run the likely risk of dramatically increasing computational runtime, since these effects require a polynomial increase in the number of interactions that need to be calculated. For example, in the standard 3D MSD model, each lattice node/vertex has approximately 6 nearest neighbors, and never more than 6. If we allowed an $r = 2$ long-range effect, the number of interactions would increase to a maximum of 25. For $r = 3$, this would increase with $O(n^3)$.

[1] The computational speed improvements would theoretically grow in $O(n^3)$ time for larger and larger systems. The 30 times improvement was observed experimentally on the general system sizes and CPUs used at the time.
[2] C++'s std map, despite the name, is actually an ordered map, and is implemented with a binary search tree. This is most likely what led to the performance hit. No tests were done against the C++ std unordered_map which uses a hash table and would have been more appropriate in this situation.

2. Layered electrodes. Currently the left and right electrodes (i.e. top and bottom electrodes) are modeled as single lattices, usually ferromagnets. But in practice, these electrodes may contain layers or stacks of different materials. (TODO: source? and example?)

Both of these features, along with others, e.g. body-center or face-center cubic crystals, and fixed environmental nodes, can be accomplished with this update.

Independent of any new features, there are also computational inefficiencies in the current algorithms' implementations. The core algorithm, the metropolis algorithm, has four main operations which account for the vast majority[3] of the software's computational runtime.

1. Generating pseudo-random numbers. The current version of the MSD software uses the Mersenne Twister algorithm.
2. Generating pseudo-random vectors. The current version uses a trigonometric approach, generating a spherical form vector with $\theta \sim Uniform[0,2\pi)$ and $\phi \sim \sin^{-1}(Uniform[-1,1])$.
3. Calculating a change in energy, ΔU_i. The current version is fairly efficient, leveraging $\Delta s + \Delta f = \Delta m$ to reduce many calculations. However, there is bounds-checking done within the metropolis algorithm for each node since the degree[4] of each node is not consistent throughout the model.
4. Calculating the Boltzmann distribution probability $p = e^{\frac{-\Delta U}{kT}}$. Specifically calculating e^x, or alternatively, $\ln x$. The current version uses the C std `<cmath>` `exp` function.

Regarding SIMD and AVX instruction, since the current version of MSD is written in C++, none of the above operations are explicitly vectorized. Any vectorization is left up to the C++ compiler. Historically all versions of the MSD software have been compiled for Windows using Visual Studio's `cl`. Due to the iterative nature of the metropolis algorithm, where each iteration is dependent of the results of pervious iterations, it is doubtful any C++ compiler would be able to fully vectorize all of these operations.

[3] Although it is hard to determine the exact percentage of the software's runtime which is taken up by these four operations, even with profiling tools, from experience and observation, I would guess somewhere between 95% and 99.9% of the runtime is just these four operations. This proportion only grows with more iterations.

[4] Degree, in this context, is a graph theory term. The degree of a vertex is the number of vertices connected to that vertex. In the context of the MSD model, the degree is the number of nearest neighbor nodes.

Methodology

MC Heisenberg model

To understand the new version of this software, I am calling the *MC*[5] *Heisenberg model*, or simply the *MC model*, we start with a mathematical description of the model.

An MC Heisenberg model is completely defined by a non-empty graph $(\mathcal{N}, \mathcal{E})$ along with a set of 11 functions, called *parameters*:

$$S, F: \mathcal{N} \to \mathbb{R}_{\geq 0}, \qquad kT: \mathcal{N} \to \mathbb{R}_{>0}, \qquad J_{e_0}: \mathcal{N} \to \mathbb{R}, \qquad \vec{B}, \vec{A}: \mathcal{N} \to \mathbb{R}^3$$

$$J, J_{e_1}, J_{e_e}, b: \mathcal{E} \to \mathbb{R}, \qquad \vec{D}: \mathcal{E} \to \mathbb{R}^3$$

Let $\mathcal{N} \neq \emptyset$ be the set of vertices in a graph. We call these vertices *nodes*. These nodes generally represent atoms in a material (e.g. metals, ferromagnets, antiferromagnets) or molecular subunits in a molecule (e.g. porphyrin, polymers).

Let $\mathcal{E} \subset \{(i, j) | i, j \in \mathcal{N}\}$ be the set of edges in the same graph. We sometimes call these edges *nearest neighbors*. Edges represent short-range energetic interactions between nodes. Self-loops, (i, i), are allowed, although not typical. Note that edges are directional. $(i, j) \neq (j, i)$ unless, and only unless, $i = j$. The direction matters for non-commutative interactions. Currently, the only implemented non-commutative interaction is the Dzyaloshinskii-Moriya interaction, $\vec{D}_{ij}$.

We partition $\mathcal{N}$ into two subsets, the *mutable nodes*, denoted $\mathcal{M}$, and the *immutable nodes*, denoted $\mathcal{I}$. Mutable nodes are subject to the metropolis algorithm, and may change state during execution. $\mathcal{M}$ correspond to the states we are trying to optimize. Immutable nodes will never be selected by the metropolis algorithm. $\mathcal{I}$ states are fixed during execution. All nodes contribute to the internal energy, U, of the system. All nodes are either mutable or immutable and never both.

Let $\mathcal{R} = \{r | r \subset \mathcal{N}\}$ be the set of *regions*. Regions are arbitrary subsets on $\mathcal{N}$ and allow researchers to model the interact between different materials. e.g. In a typical spintronics device, there are two ferromagnets, and a spacer. To model this situation, we may introduce two or three regions. One for the spacer, and one or two for the ferromagnets.

[5] MC stands for Modified Classical in reference to the classical Heisenberg model, but alternatively MC could stand for Monte Carlo since the core algorithm, the metropolis algorithm from statistical physics, is a Monte Carlo algorithm.

We also introduce the concept of an *edge-region* (abbreviated as `eregion` in the code). An edge-region, $(\mathcal{r}_0, \mathcal{r}_1) = \{(i,j) | i \in \mathcal{r}_0 \wedge j \in \mathcal{r}_1\}$. Like edges, edge-regions are directional. Generally, $(\mathcal{r}_0, \mathcal{r}_1) \neq (\mathcal{r}_1, \mathcal{r}_0)$.

For each node $i \in \mathcal{N}$, define *states* $\vec{s}_i, \vec{f}_i, \vec{m}_i = \vec{s}_i + \vec{f}_i \in \mathbb{R}^3$ and parameters $S_i = |\vec{s}_i|, F_i = |\vec{f}_i| \in \mathbb{R}_{\geq 0}$. $\vec{s}_i$ is called the *spin* of a node, i. It represents the aggregate magnetization due to unpaired localized spins at each node. e.g. The atoms in ionically neutral iron magnets, $Fe_{(s)}$, have a valence configuration of $3d^6 4s^2$ leading to 4 unpaired electrons. Each electron has a ½ spin magnitude leading to a net per atom magnetization of 2. So, if modeling $Fe_{(s)}$, $S_i = |\vec{s}_i| = 2$ for all i. In practice, $S = |\vec{s}_i| = 1$ is often used as a reference, and all other quantities scaled against it. $\vec{f}_i$ is called the *spin fluctuation*, or simply *flux*, of a node, i. It represents the aggregate magnetization around each node due to unpaired delocalized electron bands. $\vec{m}_i = \vec{s}_i + \vec{f}_i$ represents the *magnetization* of a node, i.

For all (mutable) nodes $i \in \mathcal{M}$, define $kT_i \in \mathbb{R}_{>0}$ called *temperature*. kT_i represents the external thermal energy applied to the system at each node.

There are a number of other optional *parameters* representing terms in the Hamiltonian (i.e. internal energy), $U = U_J + U_B + U_A + U_b + U_D + U_{J_{e_0}} + U_{J_{e_1}} + U_{J_{e_e}}$.

$$U_J = -\sum_{(i,j)\in\mathcal{E}_J} J_{ij}\vec{s}_i \cdot \vec{s}_j = \left\langle \begin{bmatrix} J_{ij} \\ \vdots \end{bmatrix}, \begin{bmatrix} \vec{s}_i \cdot \vec{s}_j \\ \vdots \end{bmatrix} \right\rangle_F \tag{1}$$

$J_{ij} \in \mathbb{R}$ is called the *Heisenberg exchange coupling* parameter. It's a coefficient representing a material property of a metal and can be defined for some or all edges, $(i,j) \in \mathcal{E}_J \subset \mathcal{E}$.

- $J > 0$ for ferromagnets
- $J < 0$ for antiferromagnets
- $J \approx 0$ for paramagnets
- $|J|$ models the strength (i.e. hardness) of a ferromagnet or antiferromagnet.

$$U_B = -\sum_{i\in\mathcal{N}_B} \vec{B}_i \cdot \vec{m}_i = \left\langle \begin{bmatrix} B_{i_x} & B_{i_y} & B_{i_z} \\ \vdots & \vdots & \vdots \end{bmatrix}, \begin{bmatrix} m_{i_x} & m_{i_y} & m_{i_z} \\ \vdots & \vdots & \vdots \end{bmatrix} \right\rangle_F \tag{2}$$

$\vec{B}_i \in \mathbb{R}^3$ is the external applied magnetic field at node i. It can be defined for some or all nodes, $i \in \mathcal{N}_B \subset \mathcal{N}$.

$$U_A = -\sum_{i\in\mathcal{N}_A} \vec{A}_i \cdot \vec{m}_i^{\odot 2} = \left\langle \begin{bmatrix} A_{i_x} & A_{i_y} & A_{i_z} \\ \vdots & \vdots & \vdots \end{bmatrix}, \begin{bmatrix} m_{i_x}^2 & m_{i_y}^2 & m_{i_z}^2 \\ \vdots & \vdots & \vdots \end{bmatrix} \right\rangle \tag{3}$$

$\vec{A}_i \in \mathbb{R}^3$ is called the *Anisotropy* parameters. It's a vector representing the (dis)favored magnetization direction in the material at node, i. It can be defined for some or all nodes, $i \in \mathcal{N}_A \subset \mathcal{N}$. e.g. If $\vec{A}_i = [A_x, 0,0]$, then

- If $A_x > 0$, then the material has normal anisotropy at i. $\vec{m}_i$ will prefer to align either parallel or antiparallel to $\vec{A}_i$.
- If $A_x < 0$, then the material has planar anisotropy at i. $\vec{m}_i$ will prefer to align perpendicularly to $\vec{A}_i$.
- If $A_x = 0$, then the material is isotropic.
- $|\vec{A}|$ models the strength of the anisotropy.

$$U_b = -\sum_{(i,j)\in\mathcal{E}_b} b_{ij}(\vec{s}_i \cdot \vec{s}_j)^2 = \left\langle \begin{bmatrix} b_{ij} \\ \vdots \end{bmatrix}, \begin{bmatrix} (\vec{s}_i \cdot \vec{s}_j)^2 \\ \vdots \end{bmatrix} \right\rangle_F \tag{4}$$

$b_{ij} \in \mathbb{R}$ is called the *biquadratic coupling* parameter. It's a coefficient representing higher order effects usually modeling indirect exchange or superexchange. It's important for splitting degenerate ground states in frustrated magnets. It can be defined for some or all edges, $(i,j) \in \mathcal{E}_b \subset \mathcal{E}$.

$$U_D = -\sum_{(i,j)\in\mathcal{E}_D} \vec{D}_{ij} \cdot (\vec{s}_i \times \vec{s}_j) = \left\langle \begin{bmatrix} D_{i_x} & D_{i_y} & D_{i_z} \\ \vdots & \vdots & \vdots \end{bmatrix}, \begin{bmatrix} (\vec{s}_i \times \vec{s}_j)_x & (\vec{s}_i \times \vec{s}_j)_y & (\vec{s}_i \times \vec{s}_j)_z \\ \vdots & \vdots & \vdots \end{bmatrix} \right\rangle_F \tag{5}$$

$\vec{D}_{ij} \in \mathbb{R}^3$ represents the *Dzyaloshinskii-Moriya interaction*, abbreviated *DMI*, responsible for stable magnetic vortices called skyrmions found in some materials. It can be defined for some or all edges, $(i,j) \in \mathcal{E}_D \subset \mathcal{E}$.

$$U_{J_{e_0}} = -\sum_{i\in\mathcal{N}_{J_{e_0}}} J_{e_{0_i}} \vec{s}_i \cdot \vec{f}_i = \left\langle \begin{bmatrix} J_{e_{0_i}} \\ \vdots \end{bmatrix}, \begin{bmatrix} \vec{s}_i \cdot \vec{f}_i \\ \vdots \end{bmatrix} \right\rangle_F \tag{6}$$

$$U_{J_{e_1}} = -\sum_{(i,j)\in\mathcal{E}_{J_{e_1}}} J_{e_{1_i}} (\vec{s}_i \cdot \vec{f}_j + \vec{s}_i \cdot \vec{f}_j) = \left\langle \begin{bmatrix} J_{e_{1_i}} \\ \vdots \end{bmatrix}, \begin{bmatrix} \vec{s}_i \cdot \vec{f}_j + \vec{s}_i \cdot \vec{f}_j \\ \vdots \end{bmatrix} \right\rangle_F \tag{7}$$

$$U_{J_{ee}} = -\sum_{(i,j)\in\mathcal{E}_{J_{ee}}} J_{ee_i} \vec{f}_i \cdot \vec{f}_j = \left\langle \begin{bmatrix} J_{ee_i} \\ \vdots \end{bmatrix}, \begin{bmatrix} \vec{f}_i \cdot \vec{f}_j \\ \vdots \end{bmatrix} \right\rangle_F \tag{8}$$

$J_{e_0}, J_{e_1}, J_{ee} \in \mathbb{R}$ represent exchange coupling interactions involving delocalized electron band, $\vec{f}_i$. J_{e_0}, the distance-0 interaction, can be defined on some or all nodes, $i \in \mathcal{E}_{J_{e_0}}$. J_{e_1} and J_{ee}, the distance-1 interactions, can be defined on some of all edges, $(i,j) \in \mathcal{E}_{J_{e_1}}, \mathcal{E}_{J_{ee}} \subset \mathcal{E}$ (respectively).

Independent variable, e.g. $S, F, kT, J, J_{e_0}, J_{e_1}, J_{e_e}, \vec{B}, \vec{A}, b, \vec{D}$, are called parameters. Dependent variables, e.g. $\vec{s}_i, \vec{f}_i, \vec{m}_i, U$, are called states. There are a few more basic states and notations.

$n = |\mathcal{N}| + |\mathcal{E}|$ is the number of nodes and edges in the system. The notation $n_\star = |\star|$ where $\star = \mathcal{N}, \mathcal{E}, \mathcal{r}, (\mathcal{r}_0, \mathcal{r}_1)$ is used to represent the number of nodes or edges in the system, a region, or an edge-region.

$$\vec{s}_{\mathcal{r}} = \sum_{i \in \mathcal{r}} \vec{s}_i, \qquad \vec{f}_{\mathcal{r}} = \sum_{i \in \mathcal{r}} \vec{f}_i, \qquad \vec{m}_{\mathcal{r}} = \sum_{i \in \mathcal{r}} \vec{m}_i$$

are used to represent the aggregate spin, flux, or magnetizations (respectively) of regions of the system, with $\vec{s} = \vec{s}_{\mathcal{N}}, \vec{f} = \vec{f}_{\mathcal{N}}, \vec{m} = \vec{m}_{\mathcal{N}}$ as shorthands for net spin, flux, or magnetization of the entire system (respectively).

In a similar way, $U_{\mathcal{r}}, U_{\mathcal{r}_0 \mathcal{r}_1}, U_i, U_{ij}$ can be used to get region, edge-region, node, or edge specific internal energy (respectively).

Metropolis algorithm

As with all pervious version of the MSD software, we use the metropolis algorithm, a numerical approximation method, for finding the minimal energy state of any particular system. In the new version of the software, we call a particular system a *configuration*, abbreviated as `Config` in the code. Given a configuration, and a number of iterations $\in \mathbb{N}$, the metropolis algorithm proceeds as follows:

1. Pick a random node, i
2. Pick random new states $\vec{s}_i', \vec{f}_i'$
3. Compute $\Delta U = U' - U$ where U' would be the internal energy if $\vec{s}_i \to \vec{s}_i'$ and $\vec{f}_i \to \vec{f}_i'$
4. Generate a pseudo-random $\omega \sim Uniform[0,1)$
5. If $\omega < e^{\frac{-\Delta U}{kT_i}}$, then $\vec{s}_i \to \vec{s}_i'$ and $\vec{f}_i \to \vec{f}_i'$
6. Repeat

As previously mentioned, this algorithm accounts for almost all of the runtime of the software. Of note, points 1,2,4 require a pseudo-random number generator. Point 2 involves an expensive transformation. Point 3 is potentially computationally expensive and memory expensive. Point 5 involves calculating e^x which is computationally expensive. We'll tackle these one at a time in Results.

Implementation

My approach here to implementing our MC Heisenberg model involves three steps:

1. Create a python API which handles the configuration of the system, i.e. $\mathcal{N} = \mathcal{M} + \mathcal{I}$, $\mathcal{E}$, $\mathcal{R}$, parameters, and implementation specific meta-parameters, e.g. PRNG algorithm, and PRNG seed.
2. Given all configuration information, have a python function which generates a corresponding ASM (x64 MASM) file which implements the state, and full metropolis algorithm including the four previously mentioned computationally expensive operations: PRNG, random vectors, $-\Delta U$, and $\ln \omega$ (the inverse of e^x).
3. Have python invoke a locally installed assembler and linker (e.g. Microsoft Visual Studio's `ml64 /c` and `link /DLL`) to assemble, link, and then dynamically load the model as native code specifically optimized for the given configuration. The researcher would then be able to use the model directly through a Python wrapper which depends on this native DLL.

The advantage of this approach over the current C++ approach is two-fold.

1. Python is ubiquitous in scientific research, and many more users would be able to develop custom experiments and analysis tools in Python compared to C++. Some of our lab's existing post-processing tools could potentially be incorporated directly into this new software version.
2. Because the code knows about the full configuration at compiler time, it can selectively generate only the relevant code. e.g. If F is undefined, the ASM code will remove all references to flux state vectors in its memory, and optimize calculation to leverage the fact that $\vec{s}_i = \vec{m}_i \; \forall i$.

Results

Pseudo-random numbers

In statistical physics, the efficient generation of statistically robust pseudo-random numbers is of paramount importance. A pseudo-random number generator (PRNG) is an algorithm whose goal is to generate random looking or random acting numbers. There are a few metrics which are important for measuring the efficacy of a PRNG.

1. The period, or the number of iterations before the PRNG starts to repeat its sequence, or a degenerate subsequence. As a bad example or a PRNG period, consider the Collatz conjecture which appears to always end in the degenerate sub-sequence $\{4,2,1,\ldots\}$. Since our metropolis algorithm may run for millions, billions,

or even trillions of iterations, we need a PRNG with a period at least on the order 10^{15}.

2. Statistical uniformity. There are a battery of statistical tests that are canonically performed on new PRNG algorithms to verify no anomalous statistical behavior. We expect the numbers to be indistinguishable statistically from true random numbers.
3. Cryptographic security. This is unimportant for our use case, but when pseudo-random numbers are used in other areas like security or finance (think online gambling), the ability for the PRNG seed to be determined from a subsequence is a vulnerability since if the seed can be reverse engineered, the pseudo-random numbers can be predicted.
4. Speed. Obviously, some algorithms are faster than others for different reasons. Generally, instructions which are parallelizable are preferred, and a small state reduces or removes memory access penalties.

In the current version of MSD in C++ we are using an implementation of Mersenne Twister. This algorithm was commonly used in scientific programming in 2011-2012 when the first versions of the software were written. In 2018, David Blackman and Sebastiano Vigna introduced xoshiro256** and variants, xoshiro256++ and xoshiro256+. Xoshiro256** has since become a standard PRNG used in research and high-performance statistical packages. (TODO: source?)

Xoshiro256** has a period of $2^{256} \approx 10^{77}$. My implementations of both xoshiro256** and xoshiro256++ passed RNG_test using PractRand version 0.94 with no detectable anomalies after 64 terabytes of output, or approximately $8 \cdot 10^{12}$ double precision floating point numbers (not vectorized). It took my computer just over 7 days to generate this many numbers.

The xoshiro256 family of generators have a small internal state of 4 `uint64`, or 32 bytes. x64 processors supporting AVX2 have 16 floating point registers, each capable of holding 4 `uint64`. This means that the state of a single xoshiro256 PRNG can be stored in a single floating point register eliminating any need for loading and storing the PRNG state during execution.

My implementation of xoshiro256** is vectorized, meaning it can generate four PRNGs at once. Specifically, I chose to store 4 parallel PRNGs, one in each channel of an AVX register. This requires a total of 4 dedicated registers to avoid the need to save and load the PRNG state between consecutive iterations of metropolis. I then generate pseudo-random numbers in batches of four, and employ loop-unwinding to improve performance.

Pseudo-random vectors

Previously, I was using a spherical coordinate method for generating uniform unit vectors in $\mathbb{R}^3$. $\theta = \omega,\ \phi = \sin^{-1} 2\omega - 1 \to x = \cos\phi\cos\theta, y = \cos\phi\sin\theta, z = \sin\phi$. The advantage of this method is that it does not require any rejection sampling, and therefore does not require branching. However, the computation of 6 trigonometric functions is very slow.

Instead, the new version uses Marsaglia's method:

1. Generate $u, v \sim Uniform(-1,1)$
2. $s = u^2 + v^2 = |[u \quad v]|^2$
3. If $s \geq 1$, reject and try again. The ensures $[u \quad v]$ is uniform in the unit ball in $\mathbb{R}^2$.
4. $x = 2u\sqrt{1-s}, y = 2v\sqrt{1-s}, z = 1 - 2s$

This method does have a probability, $\left(1 - \frac{\pi}{4}\right)^n \approx 21.5\%, 4.6\%, 1.0\%, \ldots$, of needing to regenerate n new pseudo-random pairs u, v, which would be somewhat expensive. However, it bypasses all trigonometric functions. Moreover, the only computationally expensive operation is $\sqrt{\quad}$ which is implemented on the hardware level in x64 with the `(v)sqrtpd` instruction. And since my PRNG is vectorized to generate 4 doubles at once, we get 2 pairs per generation. If the first fails, we can use try the second before needing the expensive regeneration.

Natural logarithm, ln

First we note that the Boltzmann distribution can be rewritten.

$$\omega < e^{\frac{-\Delta U}{kT_i}} \quad \rightarrow \quad \ln\omega < \frac{-\Delta U}{kT_i} \quad \rightarrow \quad kT_i \ln\omega < -\Delta U$$

This has the primary benefit of changing a costly floating point division instruction into an easier multiplication.

Computing ω can be vectorized, so my goal is to write an efficient vectorized $\ln$ routine, and then compute 4 $\ln\omega$ values only once at the top of each unwound loop. This reduces the number of $\ln$ calls needed.

My method for implementing $\ln$ is fairly canonical. First, we notice the following identity:

$$\ln(m \cdot 2^x) = \ln m + x \ln 2$$

Since all non-zero real numbers can be written in (binary) scientific notation with mantissa $m \in [1, 2)$ and exponent x, we can reduce solving $\ln\omega$ to calculating $\ln m$ with $m \in [1,2)$.

Next, I generated a 2^M element lookup table (e.g. $M = 10$) with precomputed values of $\ln m$. At the low level, IEEE-754 double precision floating point numbers are represented with the bit pattern

```
sxxx xxxx xxxx mmmm, mmmm mmmm mmmm mmmm, mmmm mmmm mmmm mmmm, mmmm mmmm mmmm mmmm
```

s is the sign bit. For $m \in [1,2)$, $s = 0$ (i.e. positive). xxx…x is the (unsigned) twos-complement bit pattern for the biased exponent, $x + \underbrace{(2^{10} - 1)}_{bias}$. Finally, mmm…m is the (unsigned) twos-complement notation for the mantissa, $m \in [1,2)$. We consider only the top M bits of m and ignore everything else. We then calculate $\ln m$ using a two-step method:

1. Approximate $\ln m$ using the closest value in the generated lookup table.
2. Correct the approximating using a minimax polynomial approximation precomputed using Remez method.

Once we know $\ln m$, $\ln 2$ is a constant, and x can be determined by reading the xxx…x exponent field and subtracting the bias.

Note: this method ignores denormalized floating point numbers. This means that the calculation will incorrectly calculate $\ln x$ for values of $x \leq 2^{-2^{10}+2} = 2^{-1022}$. I decided not to waste CPU cycles checking for and handling this situation since the probability of generating a denormalized input $\omega \leq 2^{-1022}$ is 2^{-1022} which is astronomically small, and basically impossible. Also, even in these cases, the program would not error catastrophically. The value of $\ln \omega$ would simply be overestimated. But since this value is so small, there is almost no chance that it would be $\geq -\Delta U$ even after accounting for the factor of kT_i. I conclude that not handling this edge case should not result in any calculation errors.

Change in internal energy, $-\Delta U$

Similar to the current C++ implementation, the ΔU calculation leverages the fact that only connected nodes need to be recalculated when a single node update is performed by the metropolis. More formally, if node i is being updated with $\vec{s}_i \to \vec{s}_i', \vec{f}_i \to \vec{f}_i'$, ΔU_{ij} is only a function of i and $j \in \{j | (i,j) \in \mathcal{E} \vee (j,i) \in \mathcal{E}\}$. We call this set of $\{j\}$ the *nearest neighbors* of i. The exact energy equation for $\Delta U_i = \Delta U_{J_i} + \Delta U_{B_i} + \cdots$ can be derived and have been published in previous works on this model. (TODO: reference?) I list them here for reference:

$$\Delta U_J = -\sum_j J_{ij}\, \Delta\vec{s}_i \cdot \vec{s}_j, \qquad \Delta U_B = -\vec{B}_i \cdot \Delta\vec{s}_i, \qquad \Delta U_A = -\vec{A}_i \cdot \left(\vec{m}'^{\odot 2} - \vec{m}^{\odot 2}\right),$$

$$\Delta U_b = -\sum_j b_{ij}\left(\left(\vec{m}_i' \cdot \vec{m}_j\right)^2 - \left(\vec{m}_i \cdot \vec{m}_j\right)^2\right), \qquad \Delta U_D = -\sum_j \vec{D}_{ij} \cdot \left(\Delta\vec{m}_i \times \vec{m}_j\right),$$

$$\Delta U_{J_{e_0}} = -J_{e_{0}i}\left(\vec{s}_i' \cdot \vec{f}_i' - \vec{s}_i \cdot \vec{f}_i\right), \qquad \Delta \mathrm{U}_{\mathrm{J_{e1}}} = -\sum_j J_{e_{1_{ij}}}\left(\Delta\vec{s}_i \cdot \vec{f}_j + \Delta\vec{f}_i \cdot \vec{s}_j\right),$$

$$\Delta U_{J_{ee}} = -\sum_j J_{e e_{ij}}\,\Delta\vec{f}_i \cdot \vec{f}_j$$

What is different, however, is that the current C++ implementation has a single $\Delta U\left(i, \vec{s}_i', \vec{f}_i'\right)$ function which both requires bounds checking for j, and which computes many unnecessary multiplications by 0 for unused parameters. The new ASM version, by contrast, simply skips the code for undefined parameters entirely, and creates a table of ΔU_i function, one for each node i, which explicitly hard code all connected edges. This removes all branching instructions, and replaces them with a single function pointer dereference from a lookup table and indirect `call` instruction.

Moreover, by carefully planning the order in which each parameter is loaded from memory, we can reduce the required AVX register footprint low enough that no local variables are needed. (My current best implementation needs 11 registers[6].) This also removes the standard preamble and epilog from each ΔU_i function call.

Also, when multiple edges share the same parameter definition, e.g. in a region, the parameter values only need to be loaded once per region. This makes sure the ASM code doesn't unnecessarily have more memory loads than the C++ version.

Initial Testing

Based on my initial tests of an iterate simulation on a typical MSD cross-shaped model, the new ASM + Python version of the software runs around 10 times faster than the C++ version. If accurate, this is a dramatic improvement over the current version.

This would mean switching to the new version of the software would not only lead to a massive speed improvement, but would also open the door to long desired features like dipolar coupling, layered electrodes, simulating defects, environmental interactions, non-simple cubic lattice structures, and more.

[6] I suspect by reordering phase 2 into two subphases, I can get this down to 10 floating point registers. 11 is sufficient for the time being. If I got it down to 10 registers, I could keep a constant (1.0, 1.0, 1.0, 1.0) loaded in YMM11 (e.g.), which would save the 9 instructions it takes to reload load it per 4 iterations: 2.25 instructions per iteration.

Appendix

To understand this paper, the reader should have at least a high-level understanding of the following topics. Language and notation common to these fields will be used.

1. Math
   a. Elementary functions $\sqrt{x}, e^x, \ln x$
   b. Trigonometric functions like $\sin x$, $\cos x$, $\tan x$, … , $\sin^{-1} x$, …
   c. Vectors, Matrices, and the Frobenius inner product
   d. Set theory, and Graph theory or Network theory
2. Computer Science:
   a. C++ and the C++ standard library
   b. Randomness and Pseudo-randomness
   c. Interpreters and Compilers
   d. Assemblers, Linkers, and Dynamic Linking
   e. CPU instruction sets, x86, x64
   f. SIMD and AVX/AVX2
3. Physics and Engineering:
   a. Thermodynamics, and Internal energy, U
   b. Statistical Mechanics (i.e. Statistical Physics)
   c. The Ising model, Heisenberg model, and metropolis algorithm
4. Chemistry:
   a. Valence electrons, and Quantum spin
   b. Quantum orbitals, and the Pauli exclusion principle or Hund's rule

References

D'Angelo, C., Molecular spintronics research project, GitHub,
https://github.com/Mathhead200/Molecular-Spintronics-Research-Project

Course description and syllabus, University of the District of Columbia,
https://www.udc.edu/seas/courses#MECH