Improving the stochastic template bank algorithm used for detection of compact binary systems by Advanced LIGO

Lucas Shadler* and Kent Blackburn†
(LIGO SURF Program, Summer 2016)
(Dated: July 6, 2016)

Within the next few years, LIGO anticipates between tens and hundreds of gravitational wave (GW) detections. This increase in signal events will inevitably require a more efficient form of analysis. Extending the current method of providing template simulations for analysis with a more intelligent coverage of the parameter space might avoid excessive delays and computational costs in analysis. Exploiting the benefits of parallel computing on powerful multi-core machines offers the potential for dramatic improvements in runtime at no cost of coverage. Thus, current functions contained in the inspiral libraries will be re-factored in the interest of having a parallelized stochastic template bank [4] generation before the start of Advanced LIGO's first observation run.

LIGO Document T1600288

I. MOTIVATION FOR STOCHASTIC TEMPLATE BANKS

Gravitational waves (GW) are ripples in the curvature of spacetime, caused by the accelerating motion of a massive body. Gravitational-wave detectors has made several recent advances that have led to the first detections of gravitational waves [1]. These waves have small amplitude and propagate in two transverse polarizations, denoted "plus" and "cross." Detection and analysis of these signals allow scientists to probe new areas of the universe with precision. Signal events can result from the inspiral, merger, and ringdown of massive binary systems made up of neutron stars and/or black holes. The continued advance of gravitational-wave detection methods bring groups such as Advanced LIGO and Advanced Virgo to anticipate the number of positive signal detection to increase rapidly within the next few observation runs [2].

Using Post-Newtonian (PN) approximations to the inspiraling compact two-body problem, accurate models of the near-final state dynamics can be created, covering a fixed range within the physical parameter space. Hidden behind noise, a positive signal can be extracted through matched filtering [3]. Raw data is filtered through an array of hundreds of thousands of the modeled waveforms, or templates, known as a template bank [4]. The template spans a subset (mass and aligned spin) of the physical parameter space. Adjacent templates are assigned a *minimal match* to the signal to optimize detection chances against the computational cost.

* lxs2208@rit.edu; School of Physics and Astronomy, Rochester Institute of Technology

† kent@ligo.caltech.edu; Division of Physics, Mathematics and Astronomy, California Institute of Technology

Evolving from the original mathematically-ordered lattice [5], the currently implemented method exploits the Metropolis-Hastings algorithm [6] to generate the bank stochastically with a significant reduction of computational cost. This paper will discuss the explored and planned methods to further increase the efficiency of the stochastic template bank algorithm.

II. CONCEPTUAL MODEL

In order to gain a deeper understanding of the underlying algorithm and lay the ground work for optimization, a "conceptual model" of the template bank was produced. The parameter space contained generic "x" and "y" parameters, each with a range of 0 to 1, exclusive. A non-Euclidean metric was defined for the space in order to simulate the computational cost of The algorithm was tasked with placing points randomly within this space such that the distance between any two points, as assigned by the metric, does not exceed a set value. The initial algorithm followed basic Monte Carlo format:

1. Choose random numbers between 0 and 1 in each dimension.
2. For every point that has already been placed, calculate the distance between the new point and that point as given by the metric.
3. If the distance is smaller than a defined minimal distance, the reject. Else accept the point.
4. Repeat the above process until a set number of trial points are rejected consecutively.

The effectiveness of each algorithm is tested on its runtime as well as the ability to cover the space, as the ideal algorithm will fill the entire space with minimized computational cost. The runtime is recorded with built-in modules native to the language. Then, a uniformly spaced grid of points is produced, and tested to see if any would be accepted into the bank. The percent coverage can be defined as one minus the ratio of the number accepted

over the number that would be accepted to an empty space. This method is very simple, and fills the space, but lacks any form of intelligent placement of points, so the number of calculations (and thus the runtime) suffers.

A. Metropolis-Hastings

The need for a more intelligent proposal system lends itself to the implementation of the Metropolis-Hastings algorithm, which uses an effectively biased distribution in choosing the next point. The space was divided up into cells of equal area in both dimensions. For each cell, a rejection probability was defined as

$$p_{reject} = \frac{n_{reject}}{n_{reject} + n_{accept}}, \tag{1}$$

where n_{reject} is the number of points rejected and n_{accept} is the number of points rejected in that cell. Every time a point is generated, a random number is generated and compared to the rejection statistic defined in the cell containing the new point. If the rejection statistic is less than the random value, the the distance calculations will be carried out. Otherwise, the algorithm will jump to a new section. This will avoid running several redundant calculations on a point that will likely be rejected. Since the number of rejections will inevitably and rapidly exceed the number of acceptances, the rejection statistic will approach unity. Thus, the exit case can be defined to leave the algorithm once the p_{reject} exceeds a critical value in each cell.

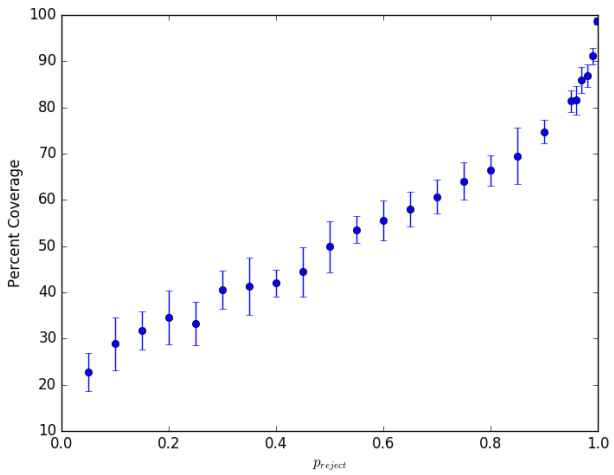

FIG. 1. Plot of percent coverage versus the rejection statistic critical value. It is roughly linear for most values but asymptotic as p_{reject} approaches unity.

Allowing this jumping process clearly offers improvements to runtime. However, since cells that had randomly had a large number of rejections towards the beginning of the process will continue to be passed, there is a potential loss in coverage due to jumping. This suggested that limiting the number of allowed jumps before overriding the jump test and forcing the point to be tested could mitigate the loss in coverage. Table I is a recording of the impact of limiting the number of consecutive jumps on runtime and coverage percentage. There is a clear trade-off between coverage and runtime as this value is increased, and illustrates a potential danger of Figure 1 shows the Percent coverage of the space as a function of p_{reject}, averaged over ten trials for each value.

TABLE I. Coverage and Runtime vs Number of Jumps

Jumps	Coverage (%)	Runtime (s)
2	100	700
10	100	200
50	99.75	80
75	99.75	73
200	98.75	52
∞	96.7	40

B. Parallel Computing

With access to very large, multi-threaded machines, it became clear that parallel programming would result in the greatest runtime improvement. Several outlets were explored, including CUDA, OpenMP, and Julia. CUDA is a gpu-parallel-computing framework, and has been set aside for now in the interest of potential future exploration. Julia is a new and up-and-coming numerical computing language developed at MIT [7] for the scientific community. It functions with Python libraries, offers friendly syntax, uses an in-house compiler and takes advantage of open-source C and Fortran numerical libraries in order to offer performance that can almost rival that of low level languages. However, the parallel computing offered natively by Julia functions by *message passing*, where each new process is remotely called by the main "parent" process. In order to improve the runtime of the stochastic template bank generation, the parallel computing must exist in the paradigm where each thread acts independently from each other, manipulating it's own data exclusively.

Eliminating several options, coupled with the fact that a great deal of the framework for LIGO is implemented in C, OpenMP stood out as a stellar option for parallel computing. Short for Open-source Multi-Processing, OpenMP offers simple compiler flags and directives to run any number of processes (defaulted to the number of Central Processing Unit (CPU) hyper-threads) simultaneously on a machine. Now that most modern computers have several CPU cores contained on their machines, running processes simultaneously can offer several runtime improvements, dividing the computational cost between threads.

Code was written in C using the OpenMP framework.

Once again the parameter space was separated into several equal-area cells. The program then split up the computation by assigning a thread to each section However, for simplicity, the original algorithm was tested, ignoring the Metropolis-Hastings algorithm. The result was an decrease in runtime by a factor of between 150 and 600 with no loss of coverage. Although it wasn't flushed out in this prototype, it is foreseeable that dividing each cell further and exploiting the Metropolis-Hastings algorithm with each thread should generate even greater improvements to efficiency. This parallel method will be implemented completely for the stochastic bank generation.

III. LSC ALGORITHM LIBRARY IMPLEMENTATION

With the desire to implement the OpenMP framework, the stochastic bank generation must be written in C. In order to exploit the benefits of parallel computing, several structures required by the algorithm, which are currently implemented in Python, will have to be re-factored into C files that can be included. Several functions are already implemented in C, including mathematical constants, and the waveform generation given mass and spin parameters. However, unit conversions, random proposal generation, and minimal match calculations require implementation. Input and output can still be handled by the previous used Python code, as the necessary information of the parameter space can be transferred via a plain text file. After all sections are fully implemented, the newly generated C functions can be encapsulated for succinct python implementation using SWIG.

A. Random Proposal Generation

The proposal generation produces random values of masses and spins in the parameter space defined physically and also by command line arguments. New proposals are generated in the $\tau_0 - \tau_3$ space, as the templates in this space have relatively uniform density, and thus will effectively benefit from divided computation. Figure 2 shows a sample template bank ranging from 10 to 25 solar masses individually. The boundary conditions in this space are very difficult to constrain, causing previous implementations of the stochastic algorithm to generate non-real values for τ_0 and τ_3.

Avoiding this errors forced the need for "brute force" methods of comparing all constraining curves to the new template to determine its validity. To avoid the computational cost of this method, critical points in $m_1 - m_2$ space will be recorded in $\tau_0 - \tau_3$. Using the critical values bounding the parameter space, regions in τ_0 can be defined where fewer constraint curves need to be examined. Once a τ_0 is randomly chosen, the fewest amount of curves possible will be tested to produce a valid τ_0 random variate. This will greatly reduce the cost of computing the bounds of random proposal generation.

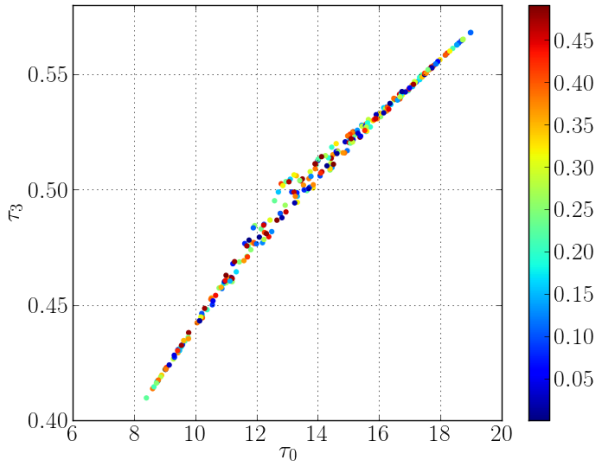

FIG. 2. Sample stochastic template bank in $\tau_0 - \tau_3$ space. The space has nearly uniform template placement but nonlinear boundaries.

B. Minimal Match Calculation

Once physically allowed and well-defined proposals are made, the proposals will be converted to waveforms. Each method of template generation requires a different set of initial parameters and will produce waveforms of varying accuracy. They will require a fair amount of analysis in order to smoothly implement their functions into the stochastic bank generation. Once the waveforms are created, the minimal match between each template must be tested. Since this an integration method, it can also be run in parallel for an increase in speed, calling multiple threads and splitting up the integration process. Excessive calculations can be avoided by immediately rejecting any templates that clearly don't belong in the template bank after a coarse integration is run. A finer integration can be run on any that meet the condition on the coarse evaluation.

IV. CONCLUSIONS

As it stands currently, the framework fully implements a stochastic bank that can generate a template bank with a *minimal match* of 97%. This process, depending on the method of waveform generation and the extent of the parameter space, can take as long as two weeks to generate. With an increase in signal events, there will not be enough computational power available to dedicate so much time to each template bank generation.

Using the new techniques of parallel computing to divide the computational requirement among CPU processes, the potential for an improvement in runtime by several orders of magnitude has been shown. It has also been demonstrated that the algorithm can be developed within the currently standing libraries, both taking advantage of previously developed code and creating new functions as needed. While the algorithm alone will reduce CPU time, the clock time required to generate the template bank will be greatly improved by the parallel computing, as well as the algorithm. The code is anticipated to run in its entirety within a few hours. Reducing the runtime by this margin will open up the powerful computational tools to other processes, allowing more events to be digested by the LIGO Data Grid.

[1] Observation of gravitational waves from a binary black hole merger. *Phys. Rev. Lett.*, 116:061102, Feb 2016.

[2] Krzysztof Belczynski, Serena Repetto, Daniel E. Holz, Richard O'Shaughnessy, Tomasz Bulik, Emanuele Berti, Christopher Fryer, and Michal Dominik. Compact Binary Merger Rates: Comparison with LIGO/Virgo Upper Limits. *Astrophys. J.*, 819(2):108, 2016.

[3] Benjamin J. Owen and B. S. Sathyaprakash. Matched filtering of gravitational waves from inspiraling compact binaries: Computational cost and template placement. *Phys. Rev.*, D60:022002, 1999.

[4] S. Babak, R. Balasubramanian, D. Churches, T. Cokelaer, and B. S. Sathyaprakash. A template bank to search for gravitational waves from inspiralling compact binaries: I. Physical models. *Classical and Quantum Gravity*, 23:5477–5504, September 2006.

[5] T. Cokelaer. Gravitational waves from inspiralling compact binaries: Hexagonal template placement and its efficiency in detecting physical signals. *Physical Review D - Particles, Fields, Gravitation and Cosmology*, 76(10), 2007.

[6] Edward Greenberg Siddhartha Chib. Understanding the metropolis-hastings algorithm. *The American Statistician*, 49(4):327–335, 1995.

[7] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A Fresh Approach to Numerical Computing. *ArXiv e-prints*, November 2014.