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: September 19, 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. The Metropolis-Hastings algorithm generates intelligent proposal templates to increase the likelyhood of placement in the bank of templates. 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 [1] 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 [2]. 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 [3].

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 [4]. Raw data is filtered through an array of hundreds of thousands of the modeled waveforms, or templates, known as a template bank [1]. 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.

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

* 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

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.

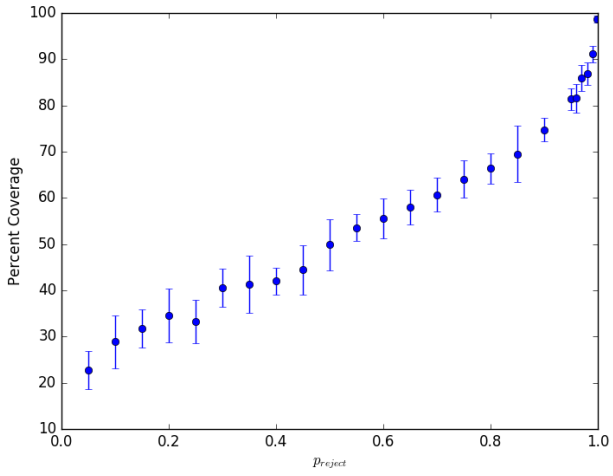

FIG. 1. Plot of percent coverage versus the rejection statistic critical value. It is roughly linear for most values but asymptotic as p_{reject} approaches unity.

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

Allowing this jumping process clearly offers improvements to runtime. However, since cells that 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.

B. Parallel Computing

With access to very large, multi-threaded machines, it becomes clear that parallel programming will 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 is 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 several orders of magnitude 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.

The efficiency of this method can be improved further still. It is foreseeable that as some spaces fill very quickly with valid templates, such cells would be likely to complete early and simply wait for all threads to meet the exit condition before returning to the program. To mitigate this potential misuse of resources, a linked list was created in C to hold each cell containing templates. When the program begins to execute in parallel, every thread will run on the entire parameter space. When a cell meets its exit condition, that cell will simply be removed from the linked list. When the linked list is empty, all threads will join and return to the program. Thus all threads will run computations until the bank has been completely filled.

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 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. Output will now be transferred to the C code, as there are already implementations to generate the necessary data files. 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 $\tau_0 - \tau_3$, or chirp time space, as the templates in this space have relatively uniform density, and thus will effectively benefit from divided computation. Figure 3 shows a sample template bank ranging from 10 to 25 solar masses individually. This sample bank includes the effective spin $\chi_{eff} = (s_1 m_1 + s_2 m_2)/(m_1 + m_2)$ represented as the color gradient. The boundary conditions in chirp time space space are very difficult to constrain, causing previous implementations of the stochastic algorithm to generate non-real values for τ_0 and τ_3.

The previously referenced paper on hexagonal template placement [5] derives characteristic curves that constrain the chirp time space given allowed values for the total mass $M = m_1 + m_2$ and $\eta = m_1 m_2 / M^2 = q/(q+1)^2$ where $q = m_1/m_2$ is the mass ratio. Manipulating these equations yields six curves that constrain the parameter space, given by

$$\tau_3 < \frac{A_3}{A_0^{2/5}} \frac{\tau_0^{2/5}}{\eta_{min}^{3/5}} \tag{2}$$

$$\tau_3 > \frac{A_3}{A_0^{2/5}} \frac{\tau_0^{2/5}}{\eta_{max}^{3/5}} \tag{3}$$

$$\tau_3 > \frac{A_3}{A_0} \tau_0 M_{min} \tag{4}$$

$$\tau_3 < \frac{A_3}{A_0} \tau_0 M_{max} \tag{5}$$

$$\tau_3 < \frac{A_3}{A_0} \tau_0 x^3 \Big|_{m_e = m_{1,min}} \tag{6}$$

$$\tau_3 < \frac{A_3}{A_0} \tau_0 x^3 \Big|_{m_e = m_{1,max}} \tag{7}$$

In this system, A_0 and A_3 represent numerical constants related to the low-cutoff frequency f_L and x represents the cubic solution to the equation

$$x^3 + \frac{A_0}{\tau_0 / m_e} x - m_e = 0 \tag{8}$$

where m_e are the extreme values of m_1. Taking $u = -A_0/(\tau_0/m_e)$ and $v = -m_e$, the solution is

$$\begin{aligned} x = &\left(-\frac{q}{2} - \frac{1}{2} \sqrt{\frac{27u^2 + 4v^3}{27}} \right)^{1/3} \\ &+ \left(-\frac{q}{2} + \frac{1}{2} \sqrt{\frac{27u^2 + 4v^3}{27}} \right)^{1/3} \end{aligned} \tag{9}$$

Figure 7 plots all six constraint curves in the chirp time space. With these constrains on τ_3, new proposals can thus be generated efficiently by selecting a random τ_0 value and generating a τ_3 that meets each of these conditions. To avoid the computational cost of this method, critical points in $m_1 - m_2$ space are recorded in $\tau_0 - \tau_3$.

Using the critical values bounding the parameter space, regions in τ_0 are 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_3 random variate. This improves on previous stochastic bank generations since the bounds of τ_0 will be generated only once.

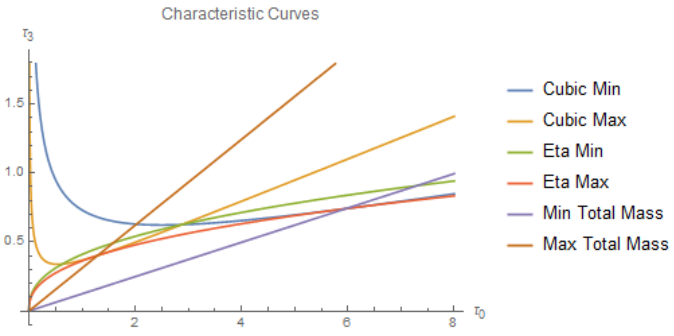

FIG. 2. Plot of the constraining curves given by Eqn. 7. These curves represent a space with the two masses ranging between 10 and 25 solar masses, and the mass ratio ranging from 1.75 to 2.5.

B. Brent's Algorithm

It was quickly discovered that the extreme values in the mass parameter space were insufficient in giving a tight coverage of chirp time parameter space. In particular, the upper bound in τ_0 often extended past the maximum physically allowed value. Careful analysis indicated that the upper bound always coincides with the intersection between the $m_{1,min}$ cubic curve and the η_{max} curve, expressed as a single equation as

$$\frac{A_3}{A_0}\tau_0 x^3 - \frac{A_3}{A_0}\tau_0 M_{max} = 0 \tag{10}$$

As there are no closed-form analytical solutions to this equation currently, a numerical approach was taken. For speed of convergence, the Van Wijngaarden-Dekker-Brent Method[8] is used. Often referenced as "Brent's Method", this algorithm uses careful book-keeping to alternate between several different numerical methods as they become more appropriate to reach a convergence. A implementation for C was taken from *Numerical Recipes in C*[9]. Combining superlinear methods of approximation with bijection algorithms, this algorithm converges on any bracketed root. The bounds were chosen as small increment larger than the original lower and upper bounds. This avoids the potential of the algorithm converging on a root in the lower bound and gives the potential for it to converge on the true upper bound while avoiding the risk of floating point precision errors when bracketing the upper bound. The lower bound can be calculated mirroring this method in the negative direction. This provides careful and tight bounding of the τ_0 space, allowing for consistent and efficient proposal generation.

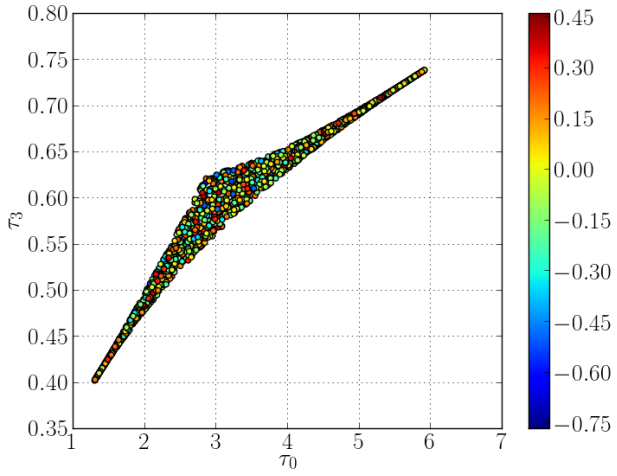

FIG. 3. Sample stochastic template bank in $\tau_0 - \tau_3$ space. The space has nearly uniform template placement but nonlinear boundaries.

C. Minimal Match Calculation

Once physically allowed and well-defined proposals are made, the proposals will are converted to waveforms. Each method of template generation requires a different set of initial parameters and will produce waveforms of varying accuracy. Each waveform simulation has already been implemented in C and accessed through functions contained within the *LALSimulation* library and yields the "plus" and "cross" polarization strains individually as 16-bit complex series of a given frequency or time step, dependent on the desired domain. Once the waveforms are created, the minimal match between each template must be evaluated. This is defined as the inner product between two templates that is normalized by the power spectral density, and the user specifies a required minimal overlap (typically 97%). Since this computation is an integrative method, it can also be run in parallel for an increase in speed, calling multiple threads and dividing 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. Current methods require that the coarse overlap calculation yield a value 5% lower than the given *minimal match* parameter. A finer integration can be run on any that meet the condition on the coarse evaluation.

D. Challenges in the Parallel Method

As the LSC implementation of the parallel method matured, several dependencies of the project presented problems outside of the scope of the summer project. Producing parallel code that accesses the same information, for example leads to "Race condition" cases of undefined behavior. When two or more threads try to access the same memory, one may have removed that cell while the other cell(s) are attempting to access a template in that location, causing an unanticipated abort. The Metropolis-Hastings algorithm will by definition increase the chance that threads will attempt to access the same cells. The most obvious solution would be to lock the memory each cell as it's being removed and providing a check for the nullity of the memory location. However, no locking mechanism provided by openMP will provide this characteristic reliably. Another method is to decrease the grid size. Smaller cells equate to a larger quantity of cells, thus decreasing the likelyhood that two threads will attempt to access the same data. However, by nature of the dynamic grid, threads will eventually compete to modify a small amount of cells, so the same issue is inevitable. Since the threads are seeded randomly by C defined *time* function, it is unknown before calling the process if the system will run to completion without error.

Due to the lack of time at the culmination of the project, two more features have yet to be implemented in the C source code to provide full functionality of the code. Although the looping minimal match testing has been flushed out, the overlap calculations have not been developed with reproducible results. Continued work will prioritizing develop a fast and effective way of testing two waveform against each other. Finally, for use with other analytic tools, the stochastic bank must be output in XML format, a feature that has not yet been developed. The individual templates have been created in a structure that the XML file will accept, but the XML file generation and dynamic updating has yet to have been implemented.

E. Current Implementation

The current state of the parallel template bank generation code is not ready to be integrated inline with the python code using SWIG wrapped C source files. The discussed lack of reliability and helper function implementation. In order to interact with the C source code, the original sbank code was truncated and modified python script entitled `sbank_parser.py`. This new script takes the normal command line arguments for the sbank generator and outputs relevant parameter information to a plain text file labeled `params.txt`. before the C file is run, it should be compiled to operate with the openMP framework and link with LSC libraries. The command line `gcc -fopenmp -std=c99 proposals.c proposals.h grid.h grid.c -o proposals -ggdb -lm -l lal -l lalsupport -l lalsimulation -l lalinspiral` in the terminal will provide this. This will provide an executable that can be run with `./proposals` to generate the template bank and any resultant output.

F. Results

As a demonstration of the structure and parallelism of the algorithm, the source files have been configured to generate the cell grid based on the input parameters, produce 10 templates in each cell, and dynamically remove each cell when it reaches the set number of templates. The code will output text files containing the mass and spin parameters of the accepted templates so that a simple python script, `sbank_plot.py`, will generate plots in $m_1 - m_2$ and $\tau_0 - \tau_3$ spaces with color maps indicating the effective spin, χ_{eff}, which is a function of the mass and spin of the template.

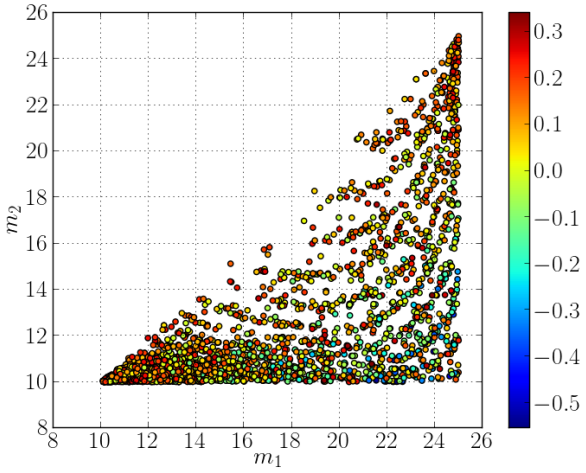

FIG. 4. Generated stochastic template bank in $m1 - m2$ space. The non-uniform density and mass constraints are readily shown.

Figure 3 shows a template bank in $\tau_0 - \tau_3$ with both masses ranging from 10 to 25 solar masses and a low-frequency cutoff $f_L = 20Hz$ using the IMRPhenomD waveform approximant, placing limits on χ_{eff}. Figure 4 shows this same bank plotted in $m_1 - m_2$. It is clear qualitatively that there is a very significant and noticeable difference in runtime with the addition of more threads. Although this has yet to be fully analyzed, the anticipated improvement in speed is feasible within the conditions presented. Looking at the plots, it is clear that templates can be placed throughout the space, and in fact appear to be overfilling in current implementation. Further work with the minimal match calculation will

reduce the amount of templates placed within the parameter space to represent all potential signal events efficiently. This code ran significantly quicker than any method used previously, running in just a few minutes compared to the days that current full implementations would take to fill a template bank.

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 without performance improvements in the algorithms.

Extensive work has been undertaken to ensure that some of the losses in efficiency are eliminated for this new implementation. Although the largest computational cost is the overlap calculations, significant computation time will be saved by the tighter constraints that have been placed on the valid proposal generations.

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 addition of the linked list eliminates the risk of individual threads to finish early on their portion of the space, allowing for a more complete use of the computational power this method provides. Implementing this algorithm in parallel introduces several complications to the original development of the code, which will continue to be worked on over the following months.

In preliminary testing, the code has been observed to run orders of magnitude quicker than current implementations on the LIGO Data Grid, and is anticipated to run in its entirety within a few hours under "worst-case" considerations. Despite these optimistic findings, continued development will be required before the framework can be implemented for an engineering run on the LIGO Data Grid. 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] 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.

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

[3] 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.

[4] 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.

[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.

[8] Richard P. Brent. *Algorithms for minimization without derivatives*. Prentice-Hall series in automatic computation. Englewood Cliffs, N.J. Prentice-Hall, 1973.

[9] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.