Tool for Rapid Analysis of Monte Carlo Simulations

Carolina Restrepo* and Kurt E. McCall†
NASA Johnson Space Center, Houston, TX.
and John E. Hurtado‡
Texas A&M University, College Station, TX.

Designing a spacecraft, or any other complex engineering system, requires extensive simulation and analysis work. Oftentimes, the large amounts of simulation data generated are very difficult and time consuming to analyze, with the added risk of overlooking potentially critical problems in the design. The authors have developed a generic data analysis tool that can quickly sort through large data sets and point an analyst to the areas in the data set that cause specific types of failures. The first version of this tool was a serial code and the current version is a parallel code, which has greatly increased the analysis capabilities. This paper describes the new implementation of this analysis tool on a graphical processing unit, and presents analysis results for NASA's Orion Monte Carlo data to demonstrate its capabilities.

I. Introduction

A Monte Carlo approach that varies a wide range of physical parameters is typically used to generate thousands of simulated scenarios of flight design. The results provide a realistic representation of a vehicle's performance and eventually help with flight certification. NASA's Orion vehicle is a current example of the importance and benefits of the Monte Carlo design approach, because several of its performance requirements are written in terms of Monte Carlo statistics.[1]

Identifying variables that can cause system failures is crucial. It is very difficult and time consuming to identify single variables, and even more so, combinations of variables that can cause problems in a design. Engineers spend much time and effort sorting through large data sets to find the causes of current problems, and often must solely rely on their system expertise. The main disadvantage to this approach, however, is not the time that it takes, but the fact that a manual search for problems does not guarantee that an analyst can find all of them.

To the authors' knowledge, there were no other general methods that could be used to identify individual variables, or their critical interactions in a reliable and timely manner for a flight dynamics problem before the development of this analysis tool in 2010.[2] The current version of this tool is now named TRAM, which stands for Tool for Rapid Analysis of Monte Carlo simulations. There are two main features that make this tool applicable to almost any Monte Carlo data set. The first is that the analyst using this tool is not required to write any problem-specific analysis scripts to run it. The second is that the analyst does not need to provide multiple Monte Carlo data sets. A single statistically significant set is enough for TRAM to run an analysis. Of course, if the single data set provided does not contain enough simulation runs or enough dispersions, the results will be only as informative as the data set provided.

The previous paper written on this topic[2] had the objective of explaining the algorithms and prove that TRAM worked well with a simple problem that had an analytical solution. The goal of this paper is to demonstrate TRAM's new analysis capabilities with a much larger and complex data set that has been used in recent design and analysis work for the Orion vehicle. Additionally, this paper shows computation times for the parallel version of TRAM, which has been programmed on a graphical processing unit (GPU). The paper is organized as follows. Section II is a brief description of the algorithms that TRAM uses. Section

*Aerospace Engineer, Integrated GN&C Analysis Branch, AIAA member, *carolina.i.restrepo@nasa.gov*
†Aerospace Engineer, Integrated GN&C Analysis Branch, *kurt.e.mccall@nasa.gov*
‡Associate Professor, Aerospace Engineering Department, AIAA member, *jehurtado@tamu.edu*

III explains the GPU code implementation. Section IV provides the background on the Orion simulation cases used as examples, and presents TRAM's analysis results for a specific system failure. Finally, Section V summarizes the benefits and capabilities of TRAM.

II. TRAM: Tool for Rapid Analysis of Monte Carlo Simulations

TRAM is a tool that can automatically rank individual design variables and combinations of variables according to how useful they are in differentiating the simulation runs that meet requirements from those that do not. To produce these rankings, the separability of good data points from bad data points is used to find regions in the input and output parameter spaces that highlight the differences between the successful and failed simulation runs in a given Monte Carlo set. The authors strived to develop this tool from the perspective of an aerospace engineer that has a solid flight dynamics background but who is not necessarily an expert in the fields of statistics of pattern recognition. The constraints listed below were published previously,[2] but are listed here once again because they were paramount in the selection of the algorithms for TRAM.

1. Algorithm Constraints
   (a) Algorithms are for post-processing data only and cannot rely on iteratively running several Monte Carlo sets.
   (b) Algorithms must make no assumptions about input probability density functions.
   (c) Algorithms must compare all types of parameters at once regardless of their units or relative magnitudes.
   (d) Algorithms must filter out variable correlations that obviously do not affect a particular failure.
2. Usability Constraints
   (a) The tool must be generic enough to address problems for any flight vehicle design.
   (b) The tool must be specific enough to capture subtleties buried in large data sets while ignoring obvious variable correlations that are not informative and do not affect system failures.
   (c) The tool must not require an analyst to modify existing code or write new pieces of problem-specific code.
   (d) The tool must be flexible enough to allow a system expert to introduce additional variables and performance metrics to the analysis.
   (e) The tool results must be tractable enough that an aerospace engineer can trust and understand without being an expert in the fields of statistics or pattern recognition.
   (f) The tool must be consistent enough to yield the same results each time it is used on a given data set (this implies that random algorithms are not appropriate).

Based on these constraints, the authors selected two well-known pattern recognition methods, kernel density estimation (KDE) and k-nearest neighbors (KNN), and combined them into a stand-alone analysis tool that can identify both single variables and combinations of variables that affect a system failure specified by an analyst. A detailed description of how TRAM uses these two methods is published in references.[2,3] Here, consider a simplified flowchart of TRAM's layout as shown in Figure 1. The two main codes are located on a GPU, and the graphical user interface (GUI) is a MATLAB program that allows the user to interface with the GPU in a seamless way, and subsequently plot and save the data in simple format.

III. GPU Implementation

CUDA, which stands for Compute Unified Device Architecture, is Nvidia's parallel computing platform and programming model for general-purpose computing on GPUs. The GPU is usually referred to as the device, which acts as a coprocessor for the computer that houses it, which is referred to as the host. A CUDA kernel is the program that runs on the GPU, while a thread is an instance of a kernel; typically the GPU runs thousands of threads during each kernel invocation. Threads are grouped into 1-, 2- or 3-dimensional

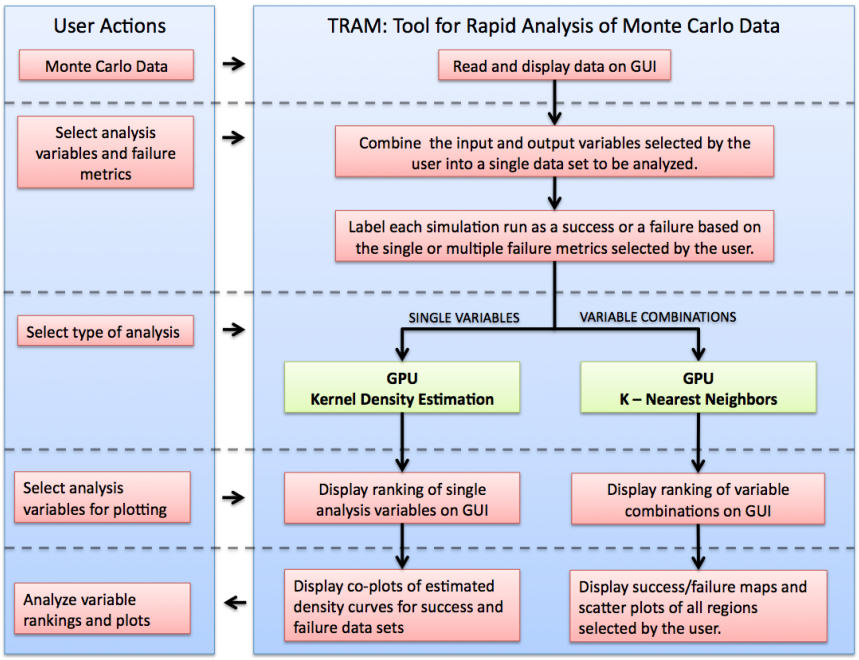


Figure 1: TRAM

blocks; threads within the same block can interact with each other using shared memory. In most cases, different blocks are executed independently of each other, with no inter-block communication. The blocks are organized into a 1-, 2- or 3-dimensional grid, and the number of blocks can greatly exceed the number of Streaming Multiprocessors (SMs) on the GPU. These algorithms were implemented under CUDA 4.2 in the Linux environment. They were tested on Nvidia GTX GeForce 580 cards, which are of the Fermi class, but have not been tested on pre-Fermi cards. The host CPU for all tests was a Intel Xeon E5620 @ 2.40GHz. A server program was created which could accommodate multiple users, providing efficient access to a limited number of GPUs. The server program contained both the GPU and serial implementations of the analysis algorithms, the latter being used for verification of the devices results.

A. Kernel Density Estimation Algorithm Implementation

The GPU implementation of the KDE algorithm was divided into two parts: KDE for original variables and KDE for compound variables. An original variable is supplied by the Monte Carlo simulation, and a compound variable is formed from the difference or the quotient of two original variables.[2] In each type of analysis the highest-ranked variables are those for which there is substantial difference between the success and failure densities. These two types of analysis are performed separately as determined by the user, but the ranked results can be combined.

The CUDA kernels were designed for problems in which the number of runs N is large and the number of variables V is substantially smaller (a typical problem might have 3000 runs but only 400 variables). Each type of analysis is composed of multiple kernels that run in sequence. For the KDE-original variable analysis, the runtime is linear in V. Since in the KDE-compound variable analysis every pair of original variables is analyzed, the runtime is quadratic in V. For brevity, we have omitted the performance data for the compound variable version.

Both KDE implementations use 1-dimensional CUDA blocks. Here we must distinguish a CUDA kernel from a KDE kernel, the latter being a Gaussian probability density that is associated with one sample point. Each thread is responsible for one Monte Carlo sample, which is read from device global memory. An estimated density, or the portion of it that is computed by each block, is stored in shared memory. The program iterates over the points of the density, and each thread in parallel adds the contribution of its samples probability kernel to the overall density. At each iteration, distinct threads visit distinct density

points to avoid shared memory conflicts. After this kernel program finishes, a second kernel program performs a reduction to sum up the portions of the densities computed by each block, creating the final densities. Other kernels rank the densities and sort the results.

B. K-Nearest Neighbors Algorithm Implementation

This type of analysis is concerned with planar scatter plots (or graphs) of the Monte Carlo data, where each axis of the plane corresponds to either an original or a compound variable. Graphs for which there is separation between the passing and failing data points are ranked highest. Each graph point is classified as either an overall success or an overall failure depending on the class of the majority of data points that fall within its local neighborhood.

Like the KDE analysis, the KNN implementation was divided into two parts, one for original variables, and one for original plus compound variables. The part for original variables only considers graphs with exactly one variable on each axis, while the other one considers graphs with original or compound variables on each axis.

The latter algorithm is an exhaustive search of all graphs formed by all combinations of original and compound variables. Let N be the number of Monte Carlo runs, and V be the number of original variables. If V_{knn} is the number of original variables selected for this analysis, with $V_{knn} \leq V$, the number of compound variables that need to be generated is quadratic in V_{knn}. The graphs are formed from all pair-wise combinations of the original and compound variables, and so the total number of graphs that must be evaluated is quadratic in V_{knn}. Because of this, a small to medium sized V_{knn} is usually specified by the user. For brevity, we do not include a table of performance data for the original plus compound variable analysis here.

The design of either nearest-neighbor kernel follows.[4] Graphs are evaluated one at a time and the layout of the grid is 2-dimensional. Each point on the Y-axis of the grid corresponds to a sample, and each point on the X-axis of the grid corresponds to a point on the graph. There is a thread for each (x, y) pair, and each thread is responsible for computing the distance between a single sample and a single point on the graph. In that way, distances for all possible pairings of samples and graph points are computed by the grid. The first kernel computes all distances, and then counts the number of successes and failures in the neighborhood of each graph point on a per-block basis. A second kernel takes these partial counts and sums them up to form the unique success and failure counts for the neighborhood of each point of the graph. Several other kernels compute the ranking of the graphs and sort the results.

This implementation deviated from the original algorithm[2] in that all samples that fell within the local neighborhood of each graph point were counted. This eliminated the need for the neighbors to be sorted according to their distance from the graph point. Previously,[2] only a fixed number of the very nearest samples were used in determining the class of the graph point.

C. Experiments

Wall-clock run times in seconds for the KDE analysis are given in Table 1. The implementations compared are written in C++ (serial) and CUDA-C++. The serial KDE code has been optimized. The GPU implementations may be optimized in the future.

For the KDE experiments, the parameters being varied are N and V. The number of points on the abscissa of each density is 128.

Table 1: KDE Speedup

	$N = 2000$	$N = 4000$	$N = 8000$	$N = 16000$
$V = 400$	C++: 18.24 CUDA: 0.177 Speedup: 103X	C++: 35.75 CUDA: 0.365 Speedup: 98X	C++: 70.26 CUDA: 0.673 Speedup: 104X	C++: 136.55 CUDA: 1.35 Speedup: 101X
$V = 800$	C++: 36.58 CUDA: 0.357 Speedup: 102X	C++: 71.67 CUDA: 0.731 Speedup: 98X	C++: 140.53 CUDA: 1.365 Speedup: 103X	C++: 272.72 CUDA: 2.6 Speedup: 104X

For the KNN-original analysis, the wall clock times are in seconds, and the varied parameters are V and V_{knn}. The serial version of the KNN code has not been optimized. The number of points on each axis of a graph is 32, for a total of 1024 points. Table 2 shows these results.

Table 2: KNN Speedup

	$N = 2000$	$N = 4000$	$N = 8000$	$N = 16000$
$V_{knn} = 40$	C++: 40.20 CUDA: 0.95 Speedup: 42X	C++: 80.30 CUDA: 4.77 Speedup: 16X	C++: 160.54 CUDA: 6.33 Speedup: 25X	C++: 321.55 CUDA: 6.52 Speedup: 49X
$V_{knn} = 80$	C++: 162.70 CUDA: 3.43 Speedup: 47X	C++: 325.28 CUDA: 6.72 Speedup: 48X	C++: 650.11 CUDA: 13.55 Speedup: 48X	C++: 1301.19 CUDA: 25.97 Speedup: 50X
$V_{knn} = 120$	C++: 367.34 CUDA: 7.61 Speedup: 48X	C++: 735.21 CUDA: 15.77 Speedup: 47X	C++: 1471.6 CUDA: 29.34 Speedup: 50X	C++: 2891.48 CUDA: 57.51 Speedup: 50X

Except for the V = 40 case, the average speedup is about 48X. The erratically varying speedups for the V = 40 cases are not unusual for CUDA code, as run times are highly dependent on kernel parameters. For some of the kernels comprising the KNN analysis, our host code dynamically selects these parameters for each problem.

IV. Example

This current example demonstrates the use of the tool in analyzing a fully integrated spacecraft. All dispersed variables are treated the same way, meaning there is no need to categorize them into mass properties, aerodynamics, environment, etc. Monte Carlo simulation data from NASA's Orion vehicle is used here to show TRAM's ability to find the significant design parameters out of the hundreds that should be analyzed in more detail.

The Orion vehicle is required to provide full abort coverage throughout the ascent phase of flight.[5,6] This abort requirement has been a major design driver for the launch abort system, the rocket, and the vehicle itself. As shown in Figure 2 , the launch abort system (LAS) has three sets of motors up stream of the

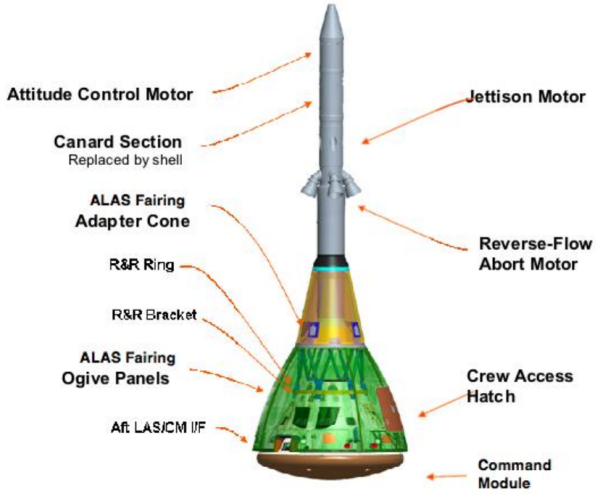


Figure 2: Orion Launch Abort System[5]

vehicle, labeled Command Module in this figure. At the current stage in the design, it is well known that

the motor plumes are major drivers for GN&C algorithm design. One of the challenges has been to design the reorientation maneuver that is performed during an abort trajectory (Figure 3) while keeping the vehicle under control.

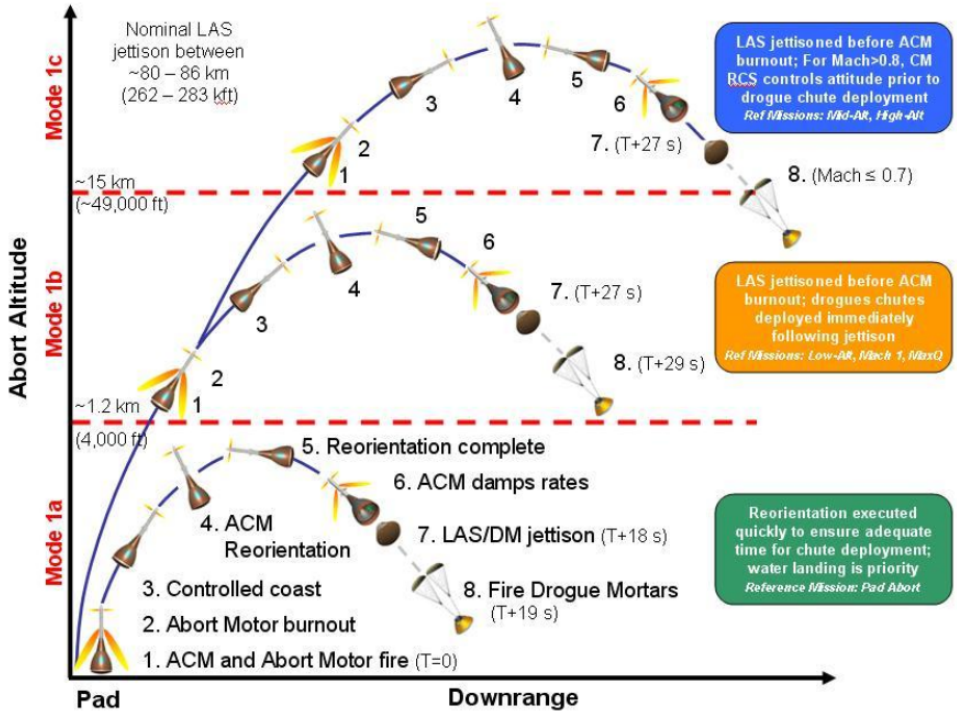


Figure 3: Orion Launch Abort Regimes[5]

NASA flight dynamics engineers have already characterized the impact of certain influential aerodynamic variables on the performance of the vehicle along the abort trajectories through detailed analysis of several Monte Carlo simulation sets. This example shows that TRAM was able to identify and rank the same aerodynamic variables that affect performance during reorientation by analyzing a single Monte Carlo set in just a few seconds.

A. Influential Design Variables

TRAM uses the KDE method to find and rank any individual variables that affect the reorientation maneuver. Figure 4 is a bar graph that displays the relative influence of each dispersed variable on the specific failure metric. Here, the cases that fail to perform a controlled reorientation maneuver are labeled as failures. It is clear that, out of over 400 variables dispersed for an ascent abort Monte Carlo simulation, only a few (six) variables are significant in comparison to the rest.

The dispersed variables include aerodynamics, mass properties such as mass and inertias of the vehicle and motors, environment properties such as air density and wind magnitude and direction, and dispersed abort initiation conditions. The top variables that correspond to the highest bars in Figure 4 are precisely the aerodynamic variables that are already known to affect tumbling during reorientation. The next highest ranked variables are the LAS moments of inertia followed by the positions of the motor nozzles. Table 3 lists these six influential variables in order. The analysis was done with a *single* Monte Carlo set. Due to ITAR restrictions on the data, the design variables used in this example are referred to by number only.

Before TRAM, the same analysis had to be performed using several different Monte Carlo sets with modified input decks. Several analysts generated all the data and developed different scripts to compare the failure rates between sets with the goal of identifying which parameter dispersions were affecting the number of failures. Table 4 shows the result of this manual analysis process. The first column shows the ranking based on how much the failure rate is reduced when compared to the failure rate for a case with all variables

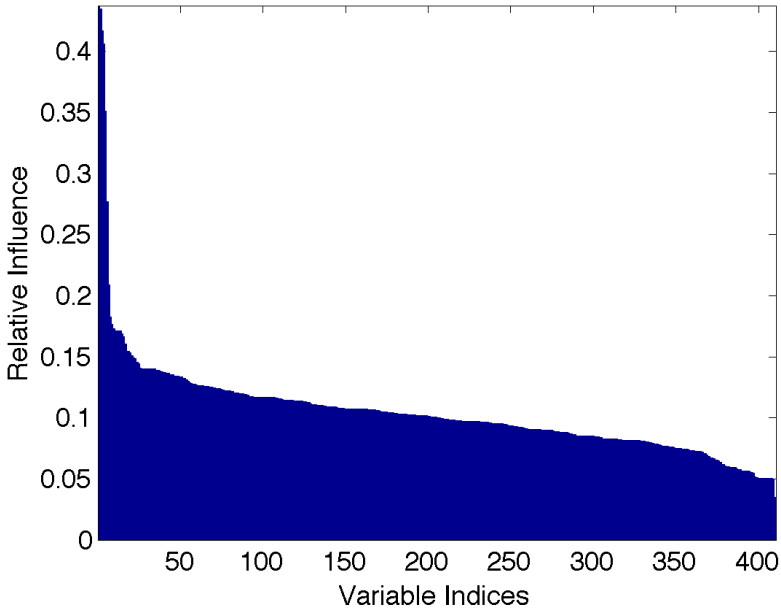


Figure 4: Orion Ascent Abort Performance Relative Effects of Dispersed Variables

Table 3: Ascent Abort Individual Variables

Rank	Variable No.	Type
1	90	aero
2	89	aero
3	92	aero
4	98	aero
5	97	aero
6	100	aero
⋮	⋮	⋮

dispersed. The second column of the table shows the name of the variable that was held constant. The third and fourth columns contain the number of tumbling cases and percentage of tumbling cases, respectively. When comparing the ranking in Table 3 to the ranking in Table 4, it is important to keep in mind that some dispersions may actually improve the results so a "true" ranking may not always be explicit. However, the algorithm can still identify the handful of critical variables out of the hundreds of variables dispersed through the analysis of a single Monte Carlo set and do so without manipulating the simulation. This is a very significant improvement over a manual analysis technique. TRAM saves the analyst the time it takes to plan and run additional Monte Carlo sets, and it saves significant time sorting through large data sets. While this manual analysis typically takes a few days, TRAM can generate the same ranking of influential variables very quickly and the analysts can immediately start focusing on the variables that matter most.

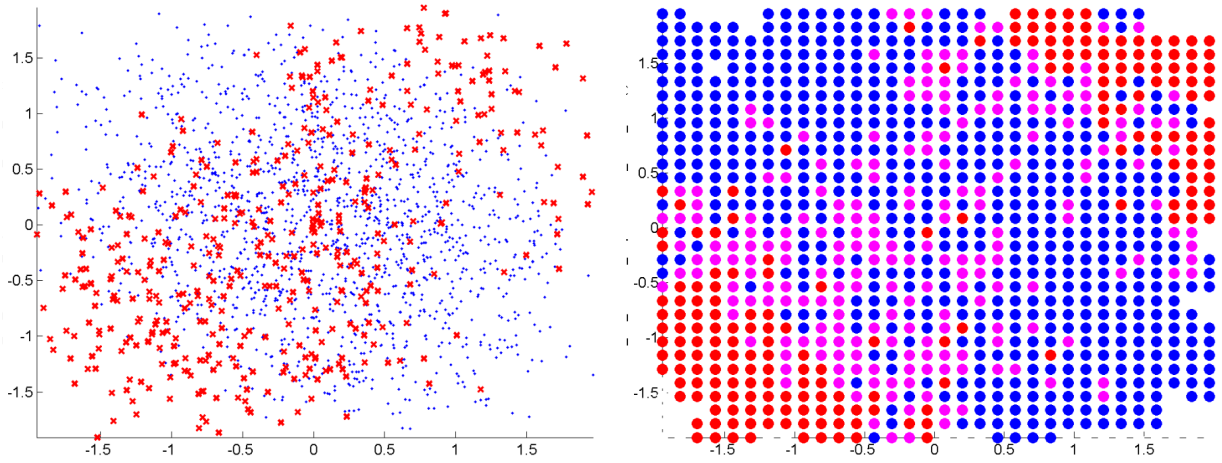
B. Influential Variable Combinations

TRAM can output a very large number of regions, some times making it difficult to display all of them. For this example, it is known that the aerodynamic variables are most influential to the ascent abort performance metrics. Here, TRAM was used to generate thousands of 2-dimensional regions combining 52 aerodynamic parameters. Figure 5 is one of these regions which shows the analyst an important relationship between

Table 4: Ascent Abort Monte Carlo Results

Rank	Fixed Variable	# failures	% failures
	0 variables dispersed	0	0
1	variable 97	237	11.85
2	variable 92	238	11.9
3	variable 89	264	13.2
4	variable 100	301	15
5	variable 98	344	17.2
6	variable 90	367	18.35
	409 variables dispersed	391	19.55

two of these variables. Figure 5a is the actual data and Figure 5b is the region created by TRAM that was used to place this particular combination at the top of the ranking so the analyst can focus on studying this relationship. The plots show that successful aborts tend to have a small x variable when the y variable is large, and a large x variable when y is small. TRAM does not explain the physics of this relationship, but it quickly points the engineer to these figures for further analysis and intuitive interpretation of the problem.



(a) Monte Carlo Data (b) k-NN Mapping

Figure 5: Orion Failure Regions

V. Summary

In general, TRAM is now a practical tool for the analysis of large Monte Carlo data sets. The GPU version has significantly improved computation times making it possible to analyze variable combinations in a timely manner. TRAM is currently a generic tool, and therefore applicable to non-aerospace data sets.

References

[1] "Crew Exploration Vehicle system requirements document," NASA CEV Document: CxP-72000, January 2007.

[2] C. Restrepo and J. E. Hurtado, "Pattern recognition for a flight dynamics monte carlo simulation," in *AIAA Guidance, Navigation and Control Conference and Exhibit*, Portland, Oregon, August 2011, number 2011-6590.

[3] C. Restrepo, "An analysis tool for flight dynamics monte carlo simulations," Ph.D. dissertation, Texas A&M University, August 2011.

[4] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k-nearest neighbors search using GPU," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, Anchorage, AK, June 2008.

[5] Ryan W. Proud, John R. Bendle, Mark B. Tedesco, and Jeremy J. Hart, "Orion guidance and control ascent abort algorithm design and performance results," in *American Astronomical Society*, August 2009.

[6] John Davidson, Jennifer Madsen, Ryan Proud, Deborah Merrit, David Raney, Dean Sparks, Paul Kenyon, Richard Burt, and Mike McFarland, "Crew Exploration Vehicle ascent abort overview," in *AIAA Guidance, Navigation and Control Conference and Exhibit*, August 2007, number 2007-6590 in AIAA.