This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

Predicting the interactive rendering time threshold of Gaussian process models with HyperSlice

Thomas Torsney-Weir, Steven Bergner, Derek Bingham, and Torsten Möller, *Senior Member, IEEE*

Abstract—In this paper we present a method for predicting the rendering time to display multi-dimensional data for the analysis of computer simulations using the HyperSlice [36] method with Gaussian process model reconstruction. Our method relies on a theoretical understanding of how the data points are drawn on slices and then fits the formula to a user's machine using practical experiments. We also describe the typical characteristics of data when analyzing deterministic computer simulations as described by the statistics community. We then show the advantage of carefully considering how many data points can be drawn in real time by proposing two approaches of how this predictive formula can be used in a real-world system

Index Terms—Multi-dimensional visualization, surrogate modeling, Gaussian process regression, interactive rendering.

1 Motivation

Many scientific studies investigate the relationship between several explanatory variables (inputs) and one or more system response variables (outputs), thereby leading to multi-dimensional data sets. Such data can arise in exploration of the input-output map for applications ranging from weather, physics and biological processes to image segmentation systems. In these cases, the output is actually a complex object such as a segmented image or 3D+time weather data. A key step towards learning about the mechanisms that are present in a computational model or laws that govern natural phenomena is to study how changes in the input variables affect the output. Visual inspection of individual outputs is suitable in small multiples, but does not scale well with increasing numbers of parameters, because of the large number of runs that are required to adequately represent model behavior in the region of interest. To more comprehensively compare outputs, automation can be taken a step further, for instance, by processing the outputs with feature extractors or fitness functions that are relevant to the driving questions. An interactive, visual investigation of the resulting feature density distribution or fitness landscape then becomes possible [6], [22], [24], but is subject to some fundamental numerical challenges that are topic of this paper.

The general approach to study deterministic computer models is known in the statistics community as *the design and analysis of computer experiments* [27]. This method involves reconstructing a continuous functional representation of the relationships among variables of the system from a discrete set of samples and then investigating the input/output relationship of the function. Numerical methods for this purpose include local derivative computation, global sensitivity indices [26], and response surface exploration [1]. However, these derived computations have to be set up carefully to yield meaningful results.

The most well known example of non-interactive visualizations of the relationships is the scatterplot matrix which works on discrete samples. Another example are continuous plots of "average" behavior over the range of each dimension, as exemplified in Chapman et al. [3]. However, any 2D or 3D view of a multi-dimensional space necessarily requires aggregation of that space. We can only "see" a subsection of the parameter space at one time. Therefore, one must create multiple static views, each looking at the data from a different perspective. The scatterplot matrix, for example, shows a 2D projection of the data for each pair of dimensions.

By allowing for user interaction one is not limited to a predetermined set of views. When the view selection changes then a new view of the data must be built. However, if the visualization system does not respond quickly to the user's interaction then the cognitive connection with the visualization is lost [30] along with the advantage of adding interaction in the first place. Arguments about what exact response time makes a visualization *interactive* vary. However, view updates somewhere between 10fps to 60fps are typically deemed acceptable.

One interactive, multi-dimensional, continuous visualization method is HyperSlice [36], which presents the user with a matrix of 2D slices of a multi-dimensional continuous function around a particular viewpoint in space. HyperSlice allows the user to change the location of the viewpoint around which they are viewing. Given this method, it would be ideal to know if the number of points or the dimensionality of the dataset will overwhelm the graphics capabilities of the user's machine and slow the frame rate. Hence, we need a way to evaluate a priori what the frame rate will be given our data. The main aim of this paper is developing a methodology to estimate the rendering time of a multi-dimensional visualization system in the form of a predictive formula. We can

- *T. Torsney-Weir and T. Möller are with the Faculty of Computing Science, University of Vienna, Vienna, Austria. E-mail: thomas.torsney-weir, torsten.moeller@univie.ac.at*
- *S. Bergner is with FINCAD. E-mail: steven.bergner@gmail.com*
- *D. Bingham is with the Department of Statistics and Actuarial Science, Simon Fraser University, Burnaby, BC, Canada. E-mail: dbingham@stat.sfu.ca*

even invert this formula so that, given a desired frame rate, we can compute the number of points possible to render in the given time. This inversion can be used, for example, to guide the user to sub-sample the dataset when the predicted rendering time will be too slow.

In order to be able to *predict* the rendering time we must come up with a function for the average rendering time based on the size of the $N \times d$ multi-dimensional dataset as well as the search distance, r over all possible view points. The advantage of a predictive formula is that, once fit, one can estimate the rendering time for all unknown values of N and r. In addition, we can use this function to examine the time and accuracy trade-off in terms of point spread versus number of samples.

A proper prediction function should describe the number of pixels that will need to be drawn based on the scene geometry. In order to adapt this function to each user's hardware platform, we require a universal methodology that can be run on each user's environment to make accurate predictions. Combining this strong theoretical foundation with a fitting step makes our method robust to further developments in GPU technology and algorithm development. We can simply recompute the time it takes the GPU to filter and draw the points without having to worry about hardware-specific optimizations.

The contributions of this paper are:

- An evaluation of how to render multi-dimensional slices on the GPU and how we can use that to predict the number of pixels drawn on screen.
- A fitting procedure for predicting rendering times on an individual user's hardware.
- An application of the prediction formula where we show an algorithm for subsampling data until we can render interactively. We also show a UI dialog box for selecting the number of samples based on the predicted rendering time.

2 Related work

The prediction of rendering times is a staple of the 3D rendering community, (e.g., see early work by Funkhouser and Séquin [7]). However, these are in a three-dimensional rendering domain. It has yet to be analyzed in the multi-dimensional domain, which we do here. Another difference with our setting is that we have a known scene geometry that we can take into account. Furthermore, our focus is on the conversion of this high-dimensional data representation to 2D and not global illumination.

One could perform an iterative search method, for example bisection search, on the number of sample points that one could render in interactive time. However, that would need to be performed for every different combination of d, N, and r. This bisection search may be prohibitively expensive if we are determining the number of samples of a complex simulation where each sample takes hours or days to compute.

2.1 Multi-D visualization

Analyzing multi-dimensional data locally is typically done by constraining each dimension within an interval [31]. Arguably, the most popular method to visually inspect multi-dimensional data is a scatter plot or scatter plot matrix (SPloM). Alternatively, one can use parallel coordinates [12] or some type of radial chart [13].

The Prosection Matrix [35] allows the user to explore the density of input parameter settings that match certain performance criteria. The user specifies what constitutes suitable output parameters as well as a "tolerance box" which represents the possible range for input settings. The system then shows a number of 2D density plots — one for each pair of parameters — indicating how many of the performance criteria were in compliance.

Often, the data points represent samples of a continuous function. Hence, it is quite common to reconstruct this continuous function as best as possible. It is imperative to also consider the visual analysis of such continuous multi-dimensional functions. In this regard, HyperSlice [36] plots two-dimensional orthogonal slices of a continuous function around a local viewpoint. This allows the user to visually inspect the behaviour of the function around this point. One advantage of HyperSlice is that it improves the quantitative means for analysis of our multi-dimensional function at least locally by measuring 2D distances. It is difficult to understand distances in multi-dimensional spaces and 2D has been shown to work better for quantitative understanding than 3D [34]. HyperMoVal [24] also relies on the user to define a slab around a particular 2D slice. The sample points within this slab are considered relevant to the view and drawn on screen. The focus of HyperMoVal is visualizing how well a model fits sampled data. Their desire is to show how "close" data points fall to a regression line.

Tuner [33] uses the HyperSlice method to visualize the effects of each parameter around a particular candidate point. Tuner is focused on finding the optimum parameter settings for a computer simulation subject to a number of criteria. The optimum parameter setting must be "high" in the sense of maximizing the objective function but also "stable" in the sense that changes in the parameter settings will not produce large changes in the objective measure. This local sensitivity analysis is visually supported with a HyperSlice view of the high-dimensional parameter space.

2.2 Multi-D interpolation

For estimating a continuous function, a popular technique is kernel regression [32]. In this case, the estimated value at a particular point in the parameter space, x', is computed by averaging over all sample points weighted by a kernel function. Formally this can be expressed as

$$\hat{f}(x) = \sum_{i=1}^{n} \varphi(x_i - x') f'(x_i) \tag{1}$$

for n sample points, x_i, and $\varphi(\cdot)$ is an approximating kernel function. In this case $f'(x_i)$ is the normalized sample value of the function $f(\cdot)$ at location $x_i \in \mathbb{R}^d$ in parameter space. The normalization factor ensures that we can compute the known values of the sample points. This factor is either automatically computed as in the case of Gaussian process regression or explicitly computed from the local neighborhood.

Often the squared exponential kernel is chosen for $\varphi(\cdot)$. This function has one or more bandwidth parameters which control the amount of smoothing between each sample point. The amount of smoothing also affects the distance at which a data point will have an effect on our regression function. While the bandwidth can be set manually, it can also be set by examining the spatial variation in $f(x)$. The Gaussian process model (GP) [25], uses statistical

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

variation to fit the kernel bandwidth appropriately. In the squared exponential case,

$$\varphi(x_i - x') = \sum_{j=1}^{d} e^{\theta_j (x_{i,j} - x'_j)^2}, \quad (2)$$

where j denotes the particular value for a certain dimension so $x_{i,j}$ is the value of the jth dimension of sample i. The value x_j is the jth dimension of the prediction point. Therefore, we have a separate parameter, θ_j for regression for each dimension. Another approach, exemplified by Hong et al. [10], is to set the kernel bandwidth to take into account the Voronoi cells around each data point. In either case, we recognize that setting the bandwidth is data-dependent. Therefore, we test rendering performance for a number of different kernel sizes.

3 Problem Description

One method of studying the input/output relationship of computer simulations is known as the *black-box model*. The black-box in this case refers to the simulation code itself. Under this analysis method one does not make any assumptions as to the inner workings of the simulator. Instead, we model the simulator as an unknown continuous function that takes a number of numerical inputs and produces a number of numerical outputs. We know the domain of the inputs. What we want to study is how varying the inputs affects the output.

While we don't know anything about how the simulation works, we can sample it by selecting a particular input setting through the simulation and recording the outputs. We can then use a proper continuous reconstruction method built from a number of samples in order to estimate the *response surface*. This fitted continuous function is known as an *emulator* in the statistics literature. We can then study the input/output relationship using the emulator instead.

3.1 1D analysis example

A common choice in the statistics community for this emulator is known as the Gaussian process model [25]. One advantage of the Gaussian process model is that the form is very well known and easy to analyze. It also allows us to measure the uncertainty of the estimation. In order to illustrate the advantages we will go through a 1D example here using a known function $f(x) = \sin(x) + \cos(2x)$ as a stand-in for some simulation code.

We begin by taking a number of discrete samples of the function. Ideally we take as many as possible but this may be limited in terms of time or budget. Since we do not know anything about the behavior of the function in the region we are sampling, we sample in some uniform random fashion. The function as well as the sample locations are shown in Fig. 1.

We would prefer to take as many samples as possible in order to learn as much about the various peaks and valleys. The interpolation method we choose depends on how we expect the values to change between the sample points. If we expect linear behavior then fitting a piecewise linear function would be ideal. If we expect more complex behavior then we should fit higher-order functions. We show 3 different fitting methods for our function in Fig. 2: piecewise linear (blue), cubic spline (green), and Gaussian process model (red) along with the true function (black). In this case the cubic spline and Gaussian process model interpolations are very close to the true function, but the true function normally would not be known beforehand.

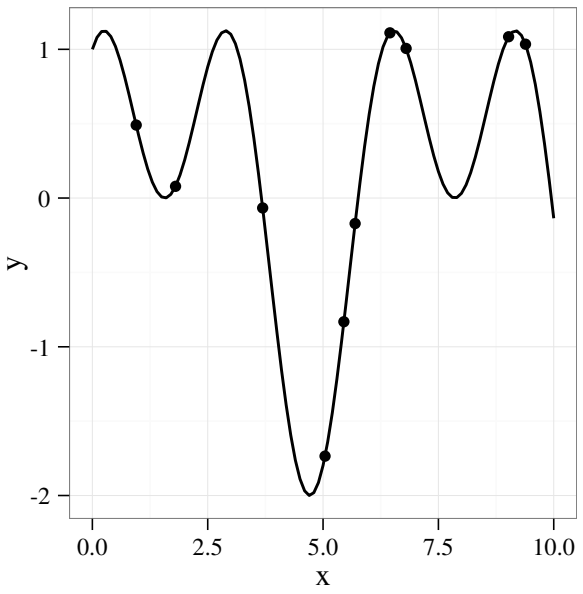


Fig. 1: An example of taking 10 uniformly distributed samples of the function $f(x) = sin(x) + cos(2x)$. The dots show the sampling locations.

The basic assumption of the Gaussian process model, however, is that the function behavior between the sample points is random in the sense it could take any path as long as it intersects the points and the correlation function we select gives the general form of the function between the points. However, the distribution of possible paths follows a multi-variate Gaussian distribution through the sample points we've selected. The mean of this distribution is the most likely path, which is typically what is visualized. By modeling the behavior this way we also get a model for the uncertainty at any point in the domain. This uncertainty grows in proportion to the distance to the sample points. In Fig. 3 we show the Gaussian process estimation of the above function given the sample points along with the standard error of the estimation. The error grows very quickly when extrapolating, for example when $x < 2$. This is because we are moving away from all sample points. This is why in real applications it is important to sample near the edge of the domain.

Parameterizing a Gaussian process model means correctly parameterizing the correlation functions to the data samples. If the function varies a lot between the sample points then we would expect low correlation between the sample points, while if the function is relatively stable between the sample points then we would expect high correlation between the points. In the spatial sense, this high and low correlation can be seen as the amount of influence the value of a particular sample point has on the value at another location a particular distance away.

3.2 Applications in multi-D

Gaussian process regression is very common in the statistics community to analyze simulations among other types of data. Here we list a number of examples where Gaussian process regression along with a uniformly distributed experimental design is used in order to run an analysis. Using uniform sampling with a Gaussian

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

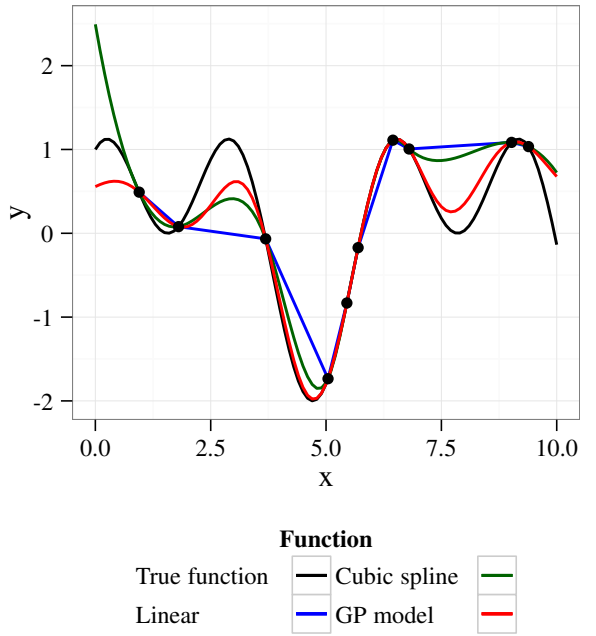


Fig. 2: Here we show different interpolation methods of the function $f(x) = sin(x) + cos(2x)$ using the same 10 uniformly distributed sample points. We show the true function as well as piecewise linear, cubic spline, and Gaussian process interpolations.

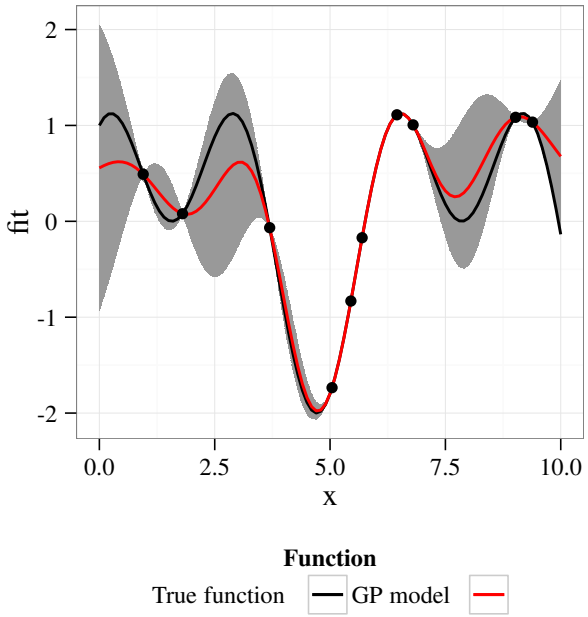


Fig. 3: The Gaussian process interpolation of the function $f(x) = sin(x) + cos(2x)$ from 10 uniformly distributed samples. The standard error of estimation from the model is shown in gray. Note that the standard error increases with the distance from the sample points.

process model has been applied in an optimization scenario, as with Couckuyt et al. [4]. They used a sparse initial sampling and then built a GP model to emulate microwave filter and textile antenna simulations. They then incrementally ran additional samples of the simulation in order to find optimal parameter settings. This process of finding additional sample locations can be quite expensive computationally. Hutter et al. [11] develop a method to find new sample locations when under a time budget. They then applied this method to find optimal parameters for a search algorithm for a propositional satisfyability solver. This method was also used to perform a sensitivity analysis of an arctic sea ice prediction model [3]. They decomposed the Gaussian process model to find "average" behavior due to each model parameter. Linkletter et al. [15] used Gaussian process models to measure the sensitivity of parameters to a cylinder deformation simulation to reduce the parameter space from 14 input factors to seven. Kaufman et al. [14] use compactly supported correlation functions to build Gaussian process models efficiently on very large data sets. This was applied to cosmological data. Hensman et al. [9] used an approach to train a Gaussian process model on $700,000$ data points in an 8-dimensional space to build a model to predict flight delays using a lower rank covariance matrix. Shepherd and Owenius [29] used Gaussian process models as a classification tool in order to classify voxels in dPET images in order to find tumor sites.

As one can see, there are a wide variety of application domains. However, all these analyses share commonalities. We show the summary information of these studies in Table 1. The number of inputs is typically on the order of 5–15 inputs. Each of these may correspond to a known factor that can vary in the real world like wind speed or the velocity of a particle, or an unknown fixed-quantity in the real world like Planck's constant or the gravitational constant. The simulation code typically creates a complex object like a 3D+time model of the world or a segmented image. On these outputs scientists define a number of feature extractors or objective functions which reduce the complex output to a set of numerical attributes [28]. Therefore, for each simulation run we get a vector of scalar input factors and the corresponding vector of scalar outputs. Each scalar output can be considered in a separate analysis so in this paper we assume that there is only one scalar output for each input configuration.

The practical number of simulation samples range from a few hundred to hundreds of thousands. This is due to either time or monetary constraints. Running more simulations than this simply takes too long or costs too much. Based on these data we test our timing function on dimensions 3–8 and run up to $1,000,000$ sample points.

3.3 Pipeline description

Despite the wide variety of application areas, all the simulation studies mentioned follow a standard procedure for analyzing these simulations. They start with a uniform sampling of the parameter space. This is usually done with a space-filling design like a Latin hypercube [18], Halton sequence [8], or Centroidal Voronoi tesselation [5]. Without any knowledge of the relative importance of the dimensions they are sampled equally.

The simulation is run using each sample and the outputs are recorded. If the output is a complex object then feature extractors

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

TABLE 1: A summary of the literature described in Sec. 3.2. We show the domain of application of each paper, their analysis goal, as well as the number of samples and number of input parameters (dimensions) of the simulation used to train the GP model.

Reference	Application	Dimensions	Samples	Goal
Couckuyt et al. [4]	Microwave filter	5	51	Optimization
Couckuyt et al. [4]	Textile antenna	2	14	Optimization
Hutter et al. [11]	Propositional logic satisfyability	4	25	Optimization
Chapman et al. [3]	Sea ice prediction	13	157	Sensitivity analysis
Linkletter et al. [15]	Cylinder deformation model	14	118	Sensitivity analysis
Kaufman et al. [14]	Cosmological data	4	20,000	Prediction
Hensman et al. [9]	Flight delays	8	700,000	Prediction
Shepherd and Owenius [29]	dPET data in radiation oncology	4	6 patient images	Classification

are run on the outputs to generate scalar results. We then build an emulator of this process. Prediction using this emulator must be fast as we will want to evaluate it at many points. Often, a Gaussian process model [25] is selected for the emulator.

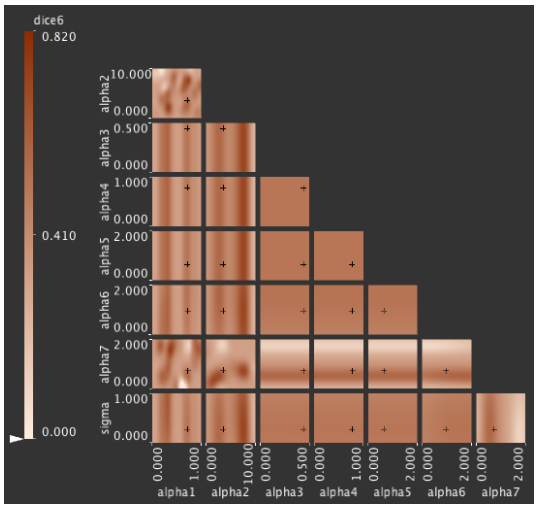


Fig. 4: Screenshot of Tuner [33] demonstrating the HyperSlice [36] method for rendering an 8-dimensional parameter space using squared exponential kernel reconstruction on an image segmentation dataset. Here we show the view using the conditional mean of the Gaussian process model.

With this emulator the user can now analyze the input/output relationship of the simulation. This can be done in a variety of ways, either by looking at static plots [3] or by interactively exploring the response surface [33]. We show an example of the HyperSlice technique for interactively exploring the response surface as implemented in Tuner [33] in Fig. 4. In this case we show the conditional mean of the Gaussian process model. Tuner can also show the estimation variability if desired. Interactive exploration of the Gaussian process model is a relatively new technique as it is more complex to implement and the limits in terms of the number of points and how the size of the kernels affects interactivity is not yet understood. The rest of this paper will discuss how to address both these questions. We present a rendering algorithm which is similar to splatting [20] to render the Gaussian process prediction function using HyperSlice. We also develop a method to determine when the interactivity of the rendering will fail taking into account the geometric interpretation of the Gaussian process model as well as the performance of an individual user's machine.

4 Requirements

In order to provide background for our predictive function we first need to consider what the multi-dimensional "scene" we are trying to render looks like. The fundamental data type we are given are sample points of the simulation. This section describes how these sample points are transformed into higher-order geometric primitives and then what happens when slicing them in order to be drawn on screen.

4.1 Scene geometry

Here we will describe the spatial interpretation of the Gaussian process model so that one can build an intuition for the geometric portion of the prediction formula. This spatial interpretation is very close to the form used for the splatting algorithm.

The spatial interpretation of the Gaussian process model using squared exponential correlation is a set of multi-dimensional ellipsoids, one for each sample point. One may be tempted to think this is due to uncertainty at the sample points but this is not the case here as the outputs of a computer simulation are considered exact. The ellipsoids are due to giving the correlation functions compact support. In order to see why this is the case we first begin by looking at the formula for the "best linear unbiased prediction," at an arbitrary location in the parameter space, x', which is,

$$\hat{y}(x') = \mu + \vec{r}(x')R^{-1}(\vec{Y} - \mu)', \tag{3}$$

here $\vec{r}$ is a vector of functions, one per sample point and each element of r, r_i, is the correlation between sample point x_i and x'. $\vec{Y}$ is a vector of the sampled outputs. μ is the estimated process mean. R is the $N \times N$ correlation matrix between the sample points using the same correlation function. We also note that neither R^{-1} nor $(Y - \mu)$ depend on x' so we let the vector, $\vec{\Upsilon} = R^{-1}(\vec{Y} - \mu)'$. Then, we can write Eq. 3 in a linear form,

$$\begin{aligned}\hat{y}(x') &= \mu + r(x')\vec{\Upsilon} \\ &= \mu + \sum_{i=1}^{N} r_i(x')\Upsilon_i.\end{aligned}$$

The value Υ_i is $f'(x_i)$ and $\hat{y}$ is $\hat{f}$ from Eq. 1 which is normalized by R^{-1} from Eq. 3.

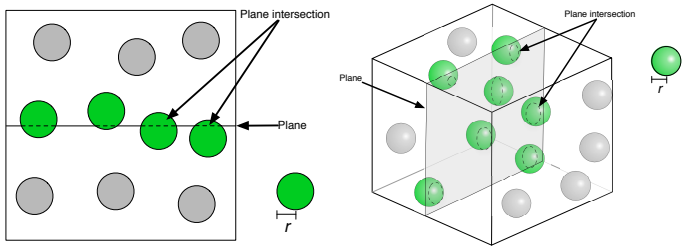
A common choice for $r(\cdot)$ is the squared exponential correlation function which has infinite support and is strictly positive meaning that it is defined everywhere in the domain and always returns some positive value however small. The squared exponential correlation sets a different falloff parameter for each dimension. For visualization purposes these small values don't contribute a perceivable effect. Therefore, we set a lower bound on the correlation value which we denote, $\varepsilon = 1 \times 10^{-9}$, essentially

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

giving the correlation function compact support which can also be done directly as in Kaufman et al. [14]. The squared exponential correlation function is radial meaning the correlation amount between points decreases as the distance increases. This ε value essentially creates a d-dimensional ellipsoid region around each sample point with principal axis lengths related to the correlation falloff parameter. The sample point will only influence predictions within this region.

4.2 HyperSlice effect on scene geometry

The last step is using some method to examine this multi-dimensional scene on a computer screen. Our chosen display method is the HyperSlice [36] technique to which will draw 2-dimensional slices through these multi-dimensional ellipsoids on screen. HyperSlice relies on the user selecting a *viewpoint* which determines the location in the multi-dimensional parameter space where to position of the slices. In Fig. 5 we show the representation for the HyperSlice technique in both 2 and 3 dimensions. In 2D the slicing plane (what one sees) is a line. In 3D it is a 2D plane slicing through the space. In higher dimensions it is also a 2D slicing plane.



(a) 2D view (b) 3D view

Fig. 5: Diagrams showing how the HyperSlice [36] method "slices" through the kernels of the Gaussian process model in both 2D (a) and 3D (b). The slicing plane is denoted as "Plane" in the figure. This is the plane the user views. Only the points, denoted in green, fall within a distance r of the slicing plane will influence the rendered image. All other points can be filtered out as they do not affect the image. We then compute the influence of the unfiltered (green) points on the slicing plane.

Only the points within range r of the slicing plane have an effect on the final image on the slice. The conceptual algorithm works by first filtering out all points that do not fall within range, r, of the slicing plane. Then, for all points within this range, we determine where the drawing plane intersects the ellipsoid which determines its impact on the drawing plane.

4.3 Algorithm

The GPU has vertex and fragment processing stages. This is analogous to the filtering and rendering stages of our rendering algorithm. We present the full schematic of the rendering algorithm in Alg. 1.

The distance to the slice is a $d-2$ dimensional distance since the remaining 2 dimensions are projected on to the slice directly. We compute the projection of the point onto the current 2D slice Alg. 1 (line 3). We then compute the distance of the sample point to the slice in line 4. Because we are only interested in the distance to the slice itself, and not a particular point on that slice, we don't include the two dimensions of the slice in the distance computation. If this distance is smaller than the size of the reconstruction kernel, we would need to render a 2D slice through this reconstruction kernel. We use a template exponential distribution which is then drawn into a quad. This is a speed-up often used in GPU-based splatting algorithms as well [20]. However, since these splat-like slices have different distances from each subplot they affect the subplot by different amounts depending on the distance. Therefore, we need to scale the intensity value of the texture by its distance to the slice (line 6).

Input: viewpoint $\vec{v}$, maximum distance r
Output: $\binom{d}{2}$ slices through an N-point, d-dimensional data set

1 **forall the** $\binom{d}{2}$ *subplots* S_i **do** *filtering*
2 **forall the** N *points* p *in the vertex buffer* **do**
3 $p_{2D} \leftarrow$ the 2D projection of p onto the slice S_i;
4 dist $\leftarrow$ the distance of p to the slice S_i;
5 **if** dist $< r$ **then** *rendering*
6 tex $\leftarrow$ the Gaussian splat scaled by dist;
7 $\hat{p}_{2D} \leftarrow$ transform p_{2D} into screen coordinates;
8 $w \leftarrow$ compute the splat width $(\sqrt{r^2 - \text{dist}^2})$;
9 send 2D quads $(\hat{p}_{2D}(x) \pm w, \hat{p}_{2D}(y) \pm w)$ to the fragment shader to be shaded with tex;
10 **end**
11 **end**
12 **end**

Algorithm 1: Algorithm for rendering multi-dimensional data using HyperSlice and Gaussian process regression

In our vertex shader implementation we filter the points as well as compute the sliced splat size. This splat size is used to generate a quad that we send to the fragment shader. We use the fragment shader to compute the final pixel color values within the slice. For practical visualizations this final pixel color value should be passed through a colormap. Therefore, in such practical applications, we recommend to render the pixel values to a floating point texture. Then this floating point texture can be rendered to the screen with a colormap shader program to convert the floating point values to color values.

One of the main bottlenecks in the rendering pipeline is transferring vertex data from CPU memory to GPU memory. This is due to the slower speed of the bus compared to the GPU. We simply do not want the GPU waiting for pixel data. The best way to address this, as specified by Xue and Crawfis [37], is to store vertex data in display lists on the GPU. Then, before calling a draw command, we only need to update the small amount of viewpoint information to render the group of slices to the GPU. Hence, we store all N d-dimensional data points on the GPU. Memory of current GPUs is large enough that we can easily store millions of points in a number of dimensions directly on the card.

5 Derivation of Scene Geometry

We now turn to a formulation for the expected total running time to draw N points in d dimensions within a slice distance of r. Our complexity analysis is based on the fact that our rendering algorithm can be decomposed into a pipeline with filtering and drawing steps. We also assume, as exemplified in Sec. 3.2, that our points are uniformly distributed in our data space. Our mathematical derivation also assumes that the ellipsoids generated

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

by the Gaussian process model are, in fact, hyperballs. While this may seem like an over-simplification, the principle axes of the ellipsoids are axis-aligned with respect to the parameter space Sec. 4.1. An ellipse is simply "stretching" the parameter space by a fixed amount in each direction.

The two stages of rendering mean that the measured time is the time to run the filtering stage plus the time to run the drawing stage. However, because of the pipeline setup of the GPU, a low number of fragments can be drawn "for free" on the spare compute capacity of the card not being used for filtering. Once this spare capacity is exhausted, the rendering time will dominate the total drawing time. Therefore, the total drawing time, t_{total}, is the time to filter the points plus the time to render the points on screen but only after a certain number of fragments are drawn. We represent this breaking point with $I(\text{frags} > a)$ which is an idicator function that returns 0 when the number of fragments is less than the breakpoint, a, and 1 otherwise.

$$t_{\text{total}} = t_{\text{filter}} + I(\text{frags} > a) * t_{\text{render}} \tag{4}$$

5.1 Filtering

In the filtering step (lines 4 and 5 of Alg. 1) we must take each data point and compute its distance to each plot in order to determine if it is worth the effort to actually draw the quad. For each sample point and for each slice, we compute the distance from the sample point to the slice. If the distance is less than r we must draw it.

Since we have subplots for each pair of dimensions there are $\binom{d}{2}$ subplots in total. Filtering is a constant time per point but is architecture-dependent. We denote this time, t_f, and the total filtering time, t_{filter}, is

$$t_{\text{filter}} = t_f \binom{d}{2} N. \tag{5}$$

5.2 Rendering

During the rendering step, we only need to process a fraction of the points N, that are visible. We call this fraction N'. The rest of the points are discarded in the filtering code on the GPU. In the case of HyperSlice, the rendering time is significant. Besides having to determine the size of the quad to be rendered in lines 7 and 8 of Alg. 1, the actual rendering time depends on the number of pixels covered by the quad since each pixel requires a constant time to draw. Because of this, our formulation for the rendering time must include the quad size for each point rendered, q_i, and the time to render each pixel in a quad, t_{H},

$$t_{\text{render}} = t_{\text{H}} \sum_{i=1}^{N'} q_i. \tag{6}$$

5.3 Expected total time

Eq. 4 gives us the total running time for a particular configuration of N points in d dimensions and for a particular viewpoint $\vec{v}$. However, we are interested in how well the rendering algorithm performs under many different configurations of points and viewpoints. The worst case performance is when we need to draw the full kernel. In other words, when the kernels are not cut off by the edges of the parameter space and the view point is in the center. However, we would like to know how the rendering will perform as the user views a set of different plots. Hence, a much more useful measurement is the *average* time to draw the view over all possible point configurations and all possible viewpoints. In order to compute this, the expected rendering time, $E[t_{\text{total}}]$, is an average over all point configurations and viewpoints in the unit cube $[0,1]^d$:

$$E[t_{\text{total}}] = t_f \binom{d}{2} N + I(\text{frags} > a) t_{\text{H}} E[N'] E[q], \tag{7}$$

Here, the first term represents the time to filter all N points over the $\binom{d}{2}$ plots and the second term is the time to draw any points that pass the filter. $I(\text{frags} > a)$ is an indicator function that returns 1 if the total number of fragments produced over all points is greater than some threshold, a.

The total amount of rendering we need to do is the number of points passing the filtering stage times the size of these points. The quantity t_{H} is the time to draw a single fragment using the HyperSlice method, $E[N']$ is the expected number of points within a distance of r from all 2D slices of the subplot matrix and $E[q]$ is the expected size of a quad drawn.

$E[N']$ is based on the the total number of sample points we need to process times the expected percentage of points that will be within range of the slices. There are N points to process for each of the $\binom{d}{2}$ subplots. For a single 2D slice, we denote the expected percentage of points within a distance r in d dimensions $\widehat{N}'(r,d)$. The percentage of points can be expanded into

$$\begin{aligned} E[N'] &= \binom{d}{2} N \cdot \widehat{N}'(r,d) \\ &= \binom{d}{2} N \cdot \sum_{i=0}^{d-2} (-1)^i \binom{d-2}{i} \frac{\pi^{d-2-i} r^{d-2+i}}{\Gamma\left(\frac{d-2+i}{2}+1\right)}. \end{aligned} \tag{8}$$

$\widehat{N}'(r,d)$ is a summation of higher and higher dimensional spheres as they are sliced by 2-dimensional planes. For the full derivation, please see Sec. A of the appendix.

For the HyperSlice technique the quantity $E[q]$ depends on the size of the spherical reconstruction kernel which depends on r and d. We denote this quantity $\widehat{Q}(r,d)$. This represents the expected number of fragments produced on a particular slice when the sample point is within a distance r of the slice. As described in Sec. B of the appendix, $E[q]$ expands into,

$$\begin{aligned} E[q] &= \widehat{Q}(r,d) \\ &= \frac{1}{\widehat{N}'(r,d)} \sum_{i=0}^{d-2} (-1)^i \binom{d-2}{i} [\text{corner}(d,i,r) \\ &\qquad - \text{side}(d,i,r) \\ &\qquad + \text{center}(d,i,r)] \\ &= \frac{1}{\widehat{N}'(r,d)} \sum_{i=0}^{d-2} (-1)^i \binom{d-2}{i} \left[\frac{4\pi^{(d-i)/2-1} r^{d+i}}{\Gamma((d+i)/2+1)} \right. \\ &\qquad - \frac{3\pi^{(d-i-1)/2} r^{d+i+1}}{\Gamma((d+i+3)/2)} \\ &\qquad \left. + \frac{2\pi^{(d-i)/2-1} r^{d+i+2}}{\Gamma((d+i)/2+2)} \right]. \end{aligned} \tag{9}$$

Here the $\text{corner}(d,i,r)$, $\text{side}(d,i,r)$, and $\text{center}(d,i,r)$ functions correspond to derivations 1, 2, and 3 listed in Sec. B.3.2 respectively. While the current formula appears quite complex, it is fast and easy to evaluate on a computer. In fact without this formula the computations would be intractable. There might exist a more

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

comprehensible formula, however, this is beyond the scope of this paper and is a subject for future work.

These formulas take into account the size of the kernel even if part of it is clipped by the edge of the parameter space. This is very important for larger kernels in higher dimensions. There, the volume of a kernel is very small but the radius may be very large and therefore the kernel is *always* clipped by the edges of the parameter space. These corner and edge terms will dominate in higher dimensional cases or for large values of r. In lower dimensional cases, or smaller values of r the center term will contribute more to the final rendering time.

6 Fitting

With a proper mathematical formulation of the scene geometry in terms of how many pixels are produced on screen, we now describe the second missing piece of the prediction. Namely, a procedure for tuning this formula to a particular user's implementation and hardware configuration. For the purposes of these timings we set our correlation cutoff value (see Sec. 4.1), ε, to 1×10^{-9}. This value allows us to link the kernel radius, r, with the kernel bandwidth parameter in the GP model. The values t_f and t_{H} in Eq. 7 are dependent on a particular hardware. Hence, we engage in an empirical stage to determine these values for a particular rendering environment.

The architecture of each GPU is different and efficiencies on one architecture may not carry over to another. Therefore, it is impossible to argue from first-principles how to derive t_f and t_{H} for a particular GPU. We fit these parameters by doing a regression analysis on empirical results obtained from examining the time to render a frame for various values of d, N, and r. While the values we derive are specific to a particular architecture, the method we present is applicable elsewhere.

In order to fit Eq. 7 we note that the first term represents the number of points we need to check to be filtered. This is constant with respect to the kernel size, r. The second term, the drawing time, increases with respect to r. The filtering time dominates for small values of r while the drawing time dominates for larger values of r. Therefore, there will be a point in terms of number of fragments drawn, that we designate a, at which point the dominant term will change from the first to the second. In order to fit this behaviour we used a segmented regression model which changes behaviour according to the value of a $\{0,1\}$ indicator function $I(\text{frags} < a)$ where frags is the total number of fragments drawn. This function returns 0 if frags $< a$. Therefore, we can form the regression formula as:

$$t_{\text{render}} = N\binom{d}{2}t_f + I(\text{frags} > a)t_{\mathrm{H}}\text{frags}. \tag{10}$$

This formula contains three parameters to be estimated: t_f (the time to filter one sample point), t_{H} (the time to render one fragment), and a (the crossover point).

6.1 Sampling

For each dimension, we would like to ensure that we have a good coverage of the number of fragments we are drawing. Hence, we must choose different values of r for each d. In order to obtain a sensible range of values for $\widehat{Q}(r,d)$ we begin by using dimension 3 as a baseline and vary radii from 0 to 0.5 in that dimension to come up with a reasonable range of $\widehat{Q}(r,d)$. Given this desired range of fragments, for the remaining dimensions we can numerically invert Eq. 9, given d and hence, obtain a range of radii r.

The last remaining issue is that for each setting of d, N, and r we must generate enough iterations such that the average number of points affecting the slices, N', converges to the theoretical expected value, $E[N']$. This is the expected number of points we will need to draw over all possible uniform distributions of points and viewpoints. In our case we found that 20 viewpoint changes, redrawing the screen for a fixed viewpoint 5 times, and 3 sample point distributions for a total of 300 timing measures resulted in good convergence.

We compute the number of fragments drawn on screen internally by the application. We can do this because we know the position of the focus point, locations of all the sample points, and the kernel information. The number of fragments in our calculations is the percentage the quad takes up on the subplot. In other words, if a quad takes up an entire subplot then it has area 1. This measurement serves as a proxy for the number of fragments generated in the GPU. OpenGL offers a query object, GL_SAMPLES_PASSED, that should return the number of fragments needed to draw the screen. This is very convenient and would correctly account for the rasterization method used. However, in our experiments this query would return several different values for the exact same scene, which does not make sense. Due to this inconsistency, we went with an internal calculation.

We can record the rendering time with either the CPU timer or the GPU timer. The CPU timer better represents the user's perception of how long it takes to draw the screen since it includes the time necessary to transfer data back and forth to the GPU. However, this timer is much noisier than the GPU timer. In our tests we found that on average the CPU timing differed from the GPU timing by a constant amount. Therefore, we used the GPU timer for our timing. The GPU timer is still quite noisy however and in order to smooth out this noise we redrew each screen five times for each change of viewpoint and then averaged the times.

6.2 Final model

If we then apply these estimates of filter and draw time to Eq. 7, the full form of our prediction model, conditional on the dimension, d, is

$$\begin{aligned} t_{total}(d,N,r) &= t_f(d)\binom{d}{2}N + I(\text{frags} > a)t_{\mathrm{H}}(d)E[N']E[q] \\ &= t_f(d)\binom{d}{2}N + I(\text{frags} > a)t_{\mathrm{H}}(d)\binom{d}{2}N\widehat{N'}(r,d)\widehat{Q}(r,d) \\ t_{total}(d,N,r) &= \binom{d}{2}N \\ &\quad \left[t_f(d) + I(\widehat{N'}(r,d)\widehat{Q}(r,d) > a(d))t_{\mathrm{H}}(d)\widehat{N'}(r,d)\widehat{Q}(r,d)\right], \end{aligned} \tag{11}$$

where $t_f(d)$, $t_{\mathrm{H}}(d)$, and $a(d)$ are the parameters we fit and $\widehat{N'}(r,d)$ and $\widehat{Q}(r,d)$ are from Eq. 8 and Eq. 9 respectively. The parameters are conditional on the dimension, d, because we fit each dimension separately so we have a different value of t_f, t_{H}, and a for each dimension. We fit these parameters using segmented regression as described in Sec. 7.1.

7 Timing Results

We now present the results of running our timing experiments. Our test machine is a Macbook Pro with Retina display with a 2.6GHz

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

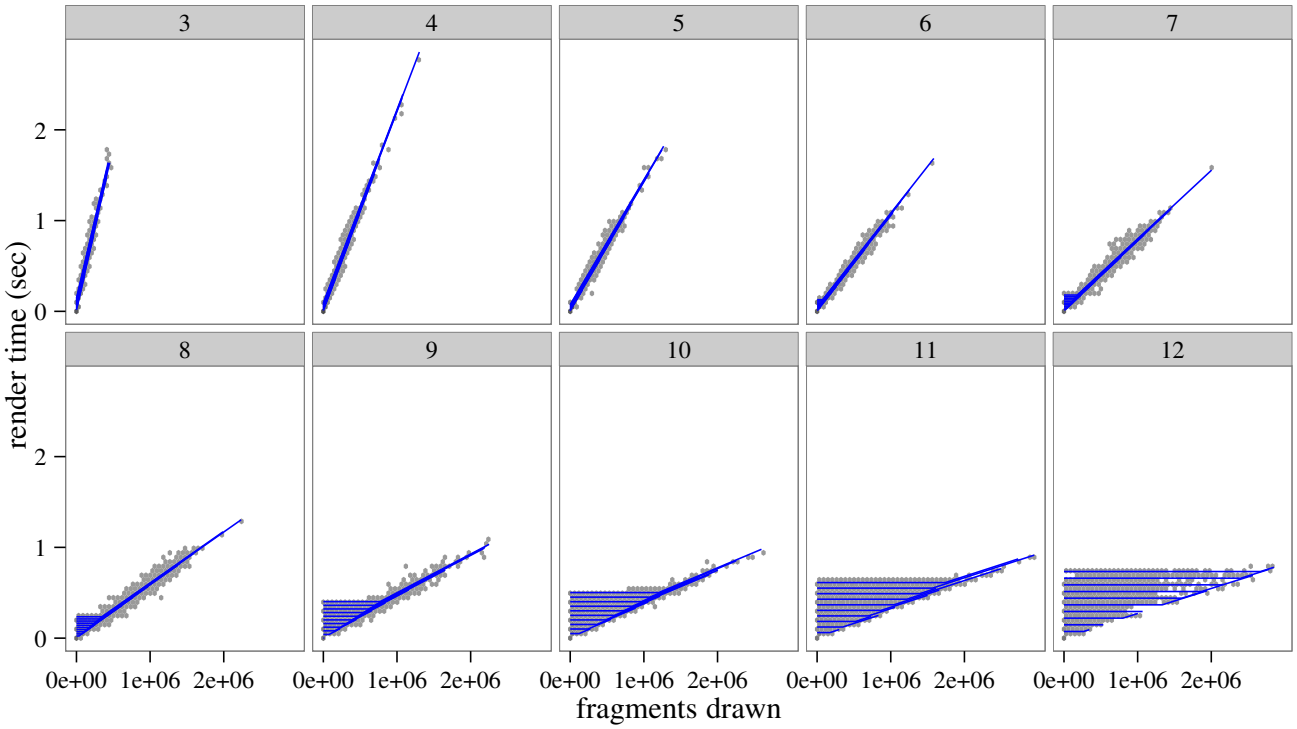


Fig. 6: Scatterplots of the time to render using the HyperSlice method. Each dimension is analyzed separately. The x-axis is the number of fragments drawn on screen and the y-axis is the number of seconds recorded by the GPU timer for the frame to draw. The blue line is the predicted rendering time using our fitted formula.

Intel Core i7, 16GB of RAM, and an NVIDIA GeForce GT 650M graphics card with 1GB of graphics memory. In order to produce consistent results we disabled the GPU power management extension. With it enabled the system varies the clock speed of the GPU while the experiments are running, producing inconsistent results.

7.1 Data fitting

We plot the rendering time as a function of number of fragments drawn in Fig. 6 for different values of N, d, and r. We treat each dimension separately as our estimation procedure is volume-based and volumes are not readily comparable between dimensions. In particular, the units of volume depend on the dimensionality and the relationship between radius and volume, for example, a 3-dimensional ball is very different from a 5-dimensional ball. Furthermore, the dimensionality of the data is usually given and not variable while one can vary the number of sample points. The x-axis in the figure is the actual number of fragments drawn on screen and the y-axis is the time, in seconds, to draw the frame. As predicted by Eq. 7, the rendering time remains constant while the GPU is primarily filtering points and then increases linearly with the number of fragments once the drawing stage dominates.

To fit these data we first computed the fragments and rendering time per sample by dividing the recorded fragments and time by $N\binom{d}{2}$ since our prediction function, Eq. 10, is linear in $N\binom{d}{2}$. We also filtered out any experiments where the rendering time was greater than 1 second since this would extend the sampling time and we are primarily concerned with finding interactive times. We then fit a basic linear model and a 2-segment regression model using the `segmented` package [21] in R. If the slope of the two segments did not differ significantly then we simply used the linear model and set the break-point to 0. For these dimensions the rendering time always dominates. We found that if the ratio between slopes of the 2-segment regression was greater than 10 then we got a better fit with the 2-segment regression than with a single linear fit.

The blue line shown in Fig. 6 is the predicted rendering time versus the number of fragments drawn. Here the blue line goes directly through the cloud of timing points. The multiple horizontal lines within each dimension correspond to the different values of N we used in our experiments. We can also see as the dimensionality increases, the filtering time begins to dominate. This is because for each subplot of the HyperSlice we must filter all N points and the number of subplots increases as $O(d^2)$. We can also see how the slope of the rendering time line (t_{H} in Eq. 10 and Table 2) decreases as the dimensions increase. This is because the screen size for running the experiments is fixed so as the number of dimensions increase the area of each individual plot becomes smaller since we have $\binom{d}{2}$ HyperSlices. Therefore, in higher dimensions we have fewer pixels to process.

The table of parameters by dimension for our reference system is listed in Table 2. Here the relationship between the number of dimensions and the fitted parameters is more apparent. For lower numbers of dimensions (3–5), it is difficult to directly measure the filtering time (t_f) as the rendering time always dominates. In this case t_f is just the y-intercept of the fit line for the rendering time. For higher values of d ($d > 5$), we can directly measure the filtering time. The reason t_f increases between dimensions 8 and 9 is because in the filtering code we parallelize the distance

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

computations in OpenGL using the `vec4` type for every group of dimensions. So, an additional group of `vec4s` is required for dimensions 9–12 and computing distances with these additional `vec4s` takes slightly more time.

TABLE 2: Table showing the calibrated parameters, a, t_f, and t_H, as a result of running a segmented regression fit using the data we gathered from our timing experiments and Eq. 11. The factors t_f and t_H are in nanoseconds and a is defined in terms of fragments per sample.

d	a	t_f (ns)	t_H (ns)
3	0.000	48.100	3470
4	0.000	15.600	2130
5	0.000	7.430	1380
6	0.00504	8.550	1030
7	0.00859	8.560	755
8	0.0123	8.400	569
9	0.0214	11.200	427
10	0.0267	11.200	342
11	0.0316	11.100	263
12	0.0404	11.100	255

7.2 Accuracy

As with the filtered scatterplot, we compared our predicted running time against new timing data using the same experimental conditions. We do this in order to test new values of r. We then compared the predicted rendering time against the empirically recorded ones. Fig. 7 shows the absolute and relative errors for prediction using the HyperSlice method and squared exponential kernel regression.

Many of the largest relative errors occur for the smallest total rendering times so *any* miscalculation will result in a large relative error. The actual difference, is shown as a histogram in Fig. 7a. Each sub-plot is a separate dimension. We show the difference between the predicted and measured rendering times, in seconds, on the x-axis. Every dimension has a strong spike at 0 indicating that most of our predictions are off by a very small amount with only a few being very inaccurate. We also show the relative error as a hexagonal-binned plot in Fig. 7b. A hexagonal-binned plot [2] is a 2d density plot using a hexagonal grid as the binning primitive. The x-axis is the percent error between the predicted time and the measured time relative to the recorded time and the y-axis is the measured time to render the frame. Many of the rendering times are very small so any error in the prediction results in a very high relative error.

In order to check the predictive ability of our procedure we also performed a 5-fold cross validation. For each dimension, we split the data so that 20% is used for building the model and the remaining 80% used for testing. We then use the testing set to compute the difference between the predicted and expected rendering times. We then repeat this procedure four more times using the next 20% partition for training. We then compute the root-mean squared error and maximum absolute error between the prediction and the recorded values. The results are shown in Table 3. While the relative errors may seem high these occur when we are trying to predict very small times so any error will be high on a relative basis. We also show the Nash-Sutcliffe efficiency [23] for each dimension, which is the ratio of variance explained by our model to the total variance. This ranges from $-\infty$ to 1 where values close to 1 mean that the model explains most of the total variance. A value over 0 is considered an acceptable level of performance [19]. All of the values in Table 3 are very close to one so our model contains a great deal of information from the data.

TABLE 3: Results of the cross-validation procedure. For each dimension we compute the root-mean squared error of prediction as well as the Nash-Sutcliffe efficiency [23], and the relative maximum error. The Nash-Sutcliffe efficiency is the ratio of the variance explained by our prediction model to the total variance. All errors are in terms of seconds.

d	RMSE	Nash-Sutcliffe	Relative max error
3	0.0685	0.931	0.624
4	0.0669	0.922	0.703
5	0.0702	0.888	0.403
6	0.0735	0.848	1.340
7	0.0726	0.801	0.383
8	0.0661	0.791	0.556
9	0.0398	0.920	0.601
10	0.0246	0.977	0.461
11	0.0127	0.996	0.386
12	0.00456	1.000	0.233

8 Application Scenarios

As was mentioned in Sec. 1, there are a number of ways to apply our prediction methodology in a practical visualization system. To this end we show two application scenarios where our method can be used to control the number of samples such that we maintain interactive rates. We show how our method may be used to sample the simulation in order to maintain a desired frame rate and to subsample an existing dataset in order to attain interactive frame rates.

8.1 Constrain sampling

Fig. 8 is a dialog box for the Tuner [33] system. The task is to enter the number of sample points to take from the simulation. The dialog is driven by Eq. 10. When the user changes the number of samples directly (a), the dialog computes the expected frame rate and displays that to the user in (b). As an alternative method, the user may value interactivity highly and consequently selects the number of sample points to take by entering the desired frame rate (b) and letting the system select the number of samples.

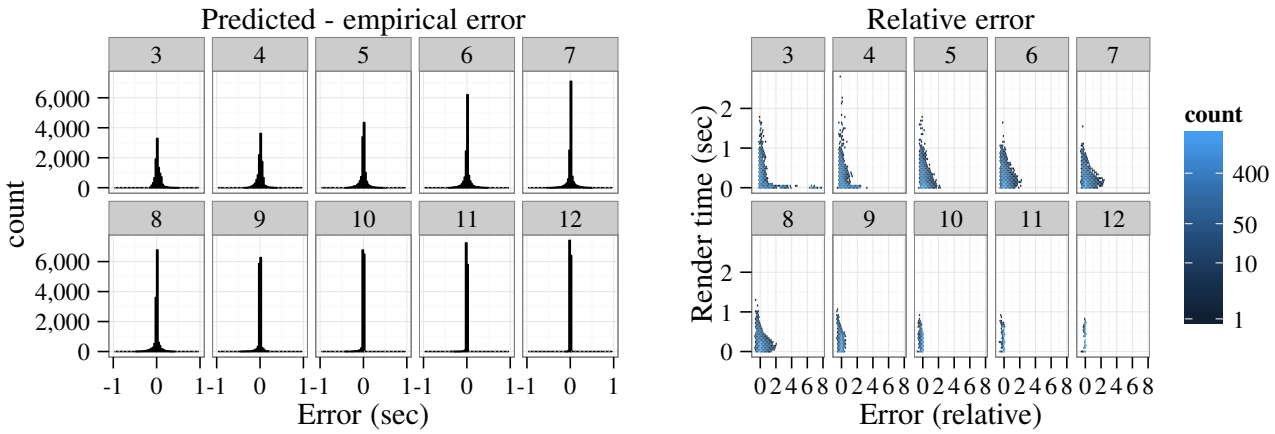
8.2 Subsample points

The goal of this algorithm, presented as Alg. 2 is to reduce the sample size, N, such that the rendering time reaches an acceptable 30fps. We will do this by removing samples from the set. An issue with simply removing points and rebuilding the Gaussian process model is that the bandwidth parameters, θ will change.

In Fig. 9 we show the trade-off between the radius, r, and the number of points, N, that can be drawn in 30fps. When subsampling data we expect that the radius around each sample point will increase as we reduce the number of sample points. The goal of Alg. 2 is to lower the number of sample points until we reach this line.

In this fashion we can have a progressive rendering setup using 2 GP models, a low-resolution model for fast rendering and a high-resolution one for detail views. The system could dynamically switch between these two when interacting.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics



(a) Actual prediction error

(b) Relative prediction error

Fig. 7: Histograms showing actual (a) and relative (b) error rates for the HyperSlice method comparing predictions using Eq. 10 to empirical results. We show (b) as a scatterplot in order to demonstrate that the largest relative errors occur when the drawing times are smallest. Each dimension is treated separately since the units of volume differ for each dimension and we have computed the filtering time, t_f, and drawing time, t_{H}, separately for each dimension.

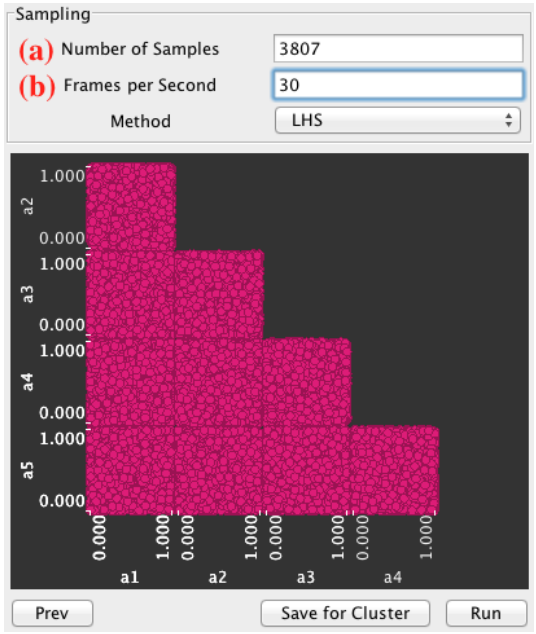


Fig. 8: A prototypical example use case for our prediction formula, Eq. 10. The user is able to either enter the number of sample points directly in field (a) and the system displays the expected fps in (b) or enter the desired fps in (b) first and the system calculates the number of sample points.

Input: Calibrated prediction formula $E[t_{\mathrm{total}}^{\mathrm{H}}](N,d,r)$, calibrated GP model parameters θ

$t_{\mathrm{pred}} \leftarrow E[t_{\mathrm{total}}^{\mathrm{H}}](N,d,r)$;
while $t_{pred} < 30fps$ **do**
  $N_{30\mathrm{fps}} \leftarrow$ Numerically solve $E[t_{\mathrm{total}}^{\mathrm{H}}](N,d,r)$ for an N that will give 30fps rendering times;
  Uniformly remove $N - N_{30\mathrm{fps}}$ sample points;
  $r' \leftarrow$ Rebuild the GP model, thereby recomputing r;
  $t_{\mathrm{pred}} \leftarrow E[t_{\mathrm{total}}^{\mathrm{H}}](N_{30\mathrm{fps}},d,r')$;
end

Algorithm 2: A proposed algorithm for subsampling data in order to achieve interactive rendering times using the Gaussian process model with the HyperSlice rendering technique.

9 Limitations and Future Work

In this paper we have presented the characteristics of data used for analyzing computer simulations under the *design and analysis of computer experiments* framework [27]. We investigate how interactive rendering times may be used in this framework using the HyperSlice [36] rendering technique implemented on the GPU. We then describe a method using both the scene geometry and estimates of the user's machine capabilities in order to make an accurate prediction of the rendering time. We find that timing, especially using wall-clock time, is extremely noisy and makes fitting very difficult. It is much more reliable to use the timer on the GPU itself, however one must take into account the time needed to return back from the GPU which in our experiments is about 1ms.

In the future, we will investigate how to reduce the number of trials needed to properly fit the formula. Currently each dimension takes about 6 hours to complete and requires the machine to be dedicated to running the timing code. Reducing the number of trials will help to alleviate this time consuming task.

It would also be interesting to extend our approach to the analysis of HyperSlice rendering in general. The basic geometrical operation in our mathematical model is slicing multi-dimensional spheres with 2D planes and estimating the area. Therefore our method should work directly with any radial basis function reconstruction technique like the work by Hong et al. [10] although we have not directly tested this. We would also like to extend our mathematical model to take the shape of the reconstruction primitive into account. This would allow us to analyze the timing of HyperSlice rendering using a much more broad set of recon-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2016.2532333, IEEE Transactions on Visualization and Computer Graphics

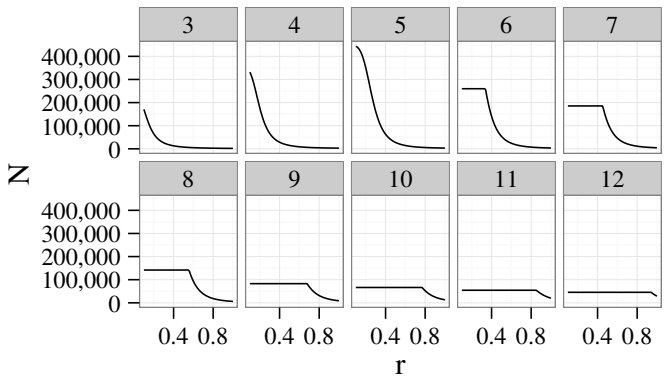


Fig. 9: Average number of points that can be rendered in 30 frames per second for the HyperSlice technique.

struction methods like nearest neighbor or linear regression. The framework can also be used to estimate the rendering time of density estimation by setting $f(x_i) = 1$ in Eq. 1. Repeating the analysis using box primitives would allow us to estimate the time complexity of some of the recent real-time large-data aggregation and visualization methods like imMens [17] and NanoCubes [16]. Both these methods use rectangular binning in their density estimation.

We will also implement our subsampling strategy. There is a lot of overdrawing occurring which does not contribute at all to the final plot as the color channel is effectively maxed out, especially as the value of N and r increases. By detecting when this occurs and only rendering the first few points we could improve rendering efficiency.

APPENDIX

For the derivations mentioned in the paper, please see the appendix in the supplementary material.

REFERENCES

[1] G. E. Box and N. R. Draper. *Response Surfaces, Mixtures, and Ridge Analyses*. Wiley Publishing, 2nd edition, April 2007. (document)

[2] D. B. Carr, R. J. Littlefield, W. L. Nicholson, and J. S. Littlefield. Scatterplot matrix techniques for large N. *Journal of the American Statistical Association*, 82(398):424–436, June 1987. 7.2

[3] W. L. Chapman, W. J. Welch, K. P. Bowman, J. Sacks, and J. E. Walsh. Arctic sea ice variability: Model sensitivities and a multidecadal simulation. *Journal of Geophysical Research: Oceans*, 99(C1):919–935, Jan. 1994. (document), 3.2, 1, 3.3

[4] I. Couckuyt, F. Declercq, T. Dhaene, H. Rogier, and L. Knockaert. Surrogate-based infill optimization applied to electromagnetic problems. *International Journal of RF and Microwave Computer-Aided Engineering*, 20(5):492–501, Sept. 2010. 3.2, 1

[5] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi tessellations: Applications and algorithms. *SIAM Review*, 41(4):637–676, Dec. 1999. 3.3

[6] S. K. Feiner and C. Beshers. Worlds within worlds: Metaphors for exploring N-dimensional virtual worlds. In *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, UIST '90, pages 76–83. ACM, Oct. 1990. (document)

[7] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93*, Annual Conference Series, pages 247–254. ACM, Sept. 1993. 2

[8] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12):701–702, Dec. 1964. 3.3

[9] J. Hensman, N. Fusi, and N. D. Lawrence. Gaussian processes for big data. In *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence*, Corvallis, Oregon, Aug. 2013. AUAI Press. 3.2, 1

[10] W. Hong, N. Neophytou, K. Mueller, and A. Kaufman. Constructing 3D elliptical Gaussians for irregular data. In *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration*, pages 213–225. Springer Berlin Heidelberg, 2006. 2.2, 9

[11] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. Murphy. Time-bounded sequential parameter optimization. In *Proceedings of the 4th International Conference on Learning and Intelligent Optimization*, pages 281–298, Venice, Italy, Jan. 2010. Springer-Verlag. 3.2, 1

[12] A. Inselberg. The plane with parallel coordinates. *The Visual Computer*, 1:69–91, Aug. 1985. 2.1

[13] S. Jayaraman and C. North. A radial focus+context visualization for multi-dimensional functions. In *Proceedings of the Conference on Visualization '02*. IEEE, Oct./Nov. 2002. 2.1

[14] C. G. Kaufman, D. Bingham, S. Habib, K. Heitmann, and J. A. Frieman. Efficient emulators of computer experiments using compactly supported correlation functions, with an application to cosmology. *The Annals of Applied Statistics*, 5(4):2470–2492, Dec. 2011. 3.2, 1, 4.1

[15] C. Linkletter, D. Bingham, N. Hengartner, D. Higdon, and K. Q. Ye. Variable selection for Gaussian process models in computer experiments. *Technometrics*, 48(4):478–490, Nov. 2006. 3.2, 1

[16] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2456–2465, Dec. 2013. 9

[17] Z. Liu, B. Jiang, and J. Heer. imMens : Real-time visual querying of big data. *Computer Graphics Forum*, 32(3):421–430, June 2013. 9

[18] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, May 1979. 3.3

[19] D. N. Moriasi, J. G. Arnold, M. W. V. Liew, R. L. Bingner, R. D. Harmel, and T. L. Veith. Model evaluation guidelines for systematic quantification of accuracy in watershed simulations. *American Society of Agricultural and Biological Engineers*, 50(3):885–900, May/June 2007. 7.2

[20] K. Mueller, T. Möller, J. E. Swan II, R. Crawfis, N. Shareef, and R. Yagel. Splatting errors and antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):178–191, Apr. 1998. 3.3, 4.3

[21] V. M. Muggeo. segmented: An R package to fit regression models with broken-line relationships. *R News*, 8(1):20–25, May 2008. 7.1

[22] T. Mühlbacher and H. Piringer. A partition-based framework for building and validating regression models. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):1962–1971, Dec. 2013. (document)

[23] J. Nash and J. Sutcliffe. River flow forecasting through conceptual models part I: A discussion of principles. *Journal of Hydrology*, 10(3):282–290, Apr. 1970. 7.2, 3

[24] H. Piringer, W. Berger, and J. Krasser. HyperMoVal: Interactive visual validation of regression models for real-time simulation. *Computer Graphics Forum*, 29(3):983–992, Aug. 2010. (document), 2.1

[25] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. The MIT Press, 2006. 2.2, 3.1, 3.3

[26] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola. *Global sensitivity analysis: The primer*. Wiley Publishing, West Sussex, England, Aug. 2008. (document)

[27] T. J. Santner, B. J. Williams, and W. I. Notz. *The Design and Analysis of Computer Experiments*. Springer, 2003. (document), 9

[28] M. Sedlmair, C. Heinzl, S. Bruckner, H. Piringer, and T. Möller. Visual parameter space analysis: A conceptual framework. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2161–2170, Dec. 2014. 3.2

[29] T. Shepherd and R. Owenius. Gaussian process models of dynamic PET for functional volume definition in radiation oncology. *IEEE Transactions on Medical Imaging*, 31(8):1542–1556, Aug. 2012. 3.2, 1

[30] B. Shneiderman. *Designing the User Interface*. Strategies for Effective Human-Computer Interaction. Addison-Wesley Publishing Company, Reading, MA, 1987. (document)

[31] B. Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6):70–77, Nov. 1994. 2.1

[32] J. S. Simonoff. *Smoothing methods in statistics*. Springer Series in Statistics. Springer-Verlang, New York, NY, USA, 1996. 2.2

[33] T. Torsney-Weir, A. Saad, T. Möller, B. Weber, H.-C. Hege, J.-M. Verbavatz, and S. Bergner. Tuner: Principled parameter finding for image segmentation algorithms using visual response surface exploration. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1892–1901, Nov./Dec. 2011. 2.1, 4, 3.3, 8.1

[34] M. Tory, A. E. Kirkpatrick, M. S. Atkins, and T. Möller. Visualization task performance with 2D, 3D, and combination displays. *IEEE Transactions on Visualization and Computer Graphics*, 12(1):2–13, Jan. 2006. 2.1

[35] L. Tweedie and R. Spence. The Prosection Matrix: A tool to support the interactive exploration of statistical models and data. *Computational Statistics*, 13(1):65–76, Mar. 1998. 2.1

[36] J. J. van Wijk and R. van Liere. HyperSlice: Visualization of scalar functions of many variables. In *Proceedings of the 4th Conference on Visualization '93*, pages 119–125. IEEE Computer Society, Oct. 1993. (document), 2.1, 4, 4.2, 5, 9

[37] D. Xue and R. Crawfis. Efficient splatting using modern graphics hardware. *Journal of Graphics, GPU, and Game Tools*, 8(3):1–21, Sept. 2003. 12