Special Issue: Unmanned Marine Systems

Regular Article

A Massively Parallel Implementation of Gaussian Process Regression for Real Time Bathymetric Modeling and Simultaneous Localization and Mapping

Kristopher E. Krasnosky[1] and Christopher Roman[2]
[1]Graduate School of Oceanography, University of Rhode Island, Narragansett, RI 02882
[2]University of Rhode Island, Narragansett, RI 02882

Abstract: A Gaussian process regression (GPR) can be used as a stochastic method for modeling underwater terrain using multibeam sonar data. A GPR model can improve the effective resolution of a terrain model over traditional gridding methods and quantify uncertainty with an estimate of model variance over its entire domain. However, GPR solutions are extremely computationally expensive and generally reserved for post-processing applications. To make GPR viable for real-time applications, we developed massively parallel GPR (MP-GPR) to run on a graphical processing unit (GPU). MP-GPR is first used to process real-time multibeam data when assuming accurate navigation from a high precision position, heading, and attitude source. In underwater environments, however, we are denied the luxury of high precision position sensors and typically rely on dead reckoning. Therefore, MP-GPR was used as a terrain model for a featureless, Rao-Blackwellized particle filter based, bathymetric particle-filter simultaneous localization and mapping (BPSLAM) algorithm. Our GPU-based extension of BPSLAM (GP-BPSLAM) estimates many possible vehicle trajectories and MP-GPR predicts a possible map for each. By comparing the recent multibeam observations against the model for each possible trajectory, unlikely trajectories can be identified and removed. GP-BPSLAM is able to process data in real time and generate a navigation solution that is more accurate than simple dead reckoning.

Keywords: Gaussian process regression, SLAM, navigation, perception, GPS-denied operation, underwater robotics

1. Introduction

Modern autonomous vehicles have advanced perceptual sensors and sufficient processing power to produce bathymetric maps in real time. A continuously updating, online bathymetric map allows such a vehicle to plan its trajectory on the fly to optimize mapping coverage (Galceran et al.,

Received: 29 January 2021; revised: 3 December 2021; accepted: 4 December 2021; published: 28 May 2022.

Correspondence: Kristopher E. Krasnosky, Graduate School of Oceanography, University of Rhode Island, Narragansett, RI 02882., Email: kkrasnosky@uri.edu

DOI: https://doi.org/10.55417/fr.2022031

2015; Shi and Zhou, 2020) or quality (Galceran et al., 2013; Pairet et al., 2018). If computing the bathymetric model is efficient enough, it can also be used to assist in vehicle navigation refinement using simultaneous localization and mapping (SLAM) strategies. Finally, a high fidelity map, especially one with confidence intervals, allows for safe navigation and path planning close to terrain.

Bathymetric maps are typically produced by surveying an area in an organized pattern with a multibeam sonar system aboard a surface ship or underwater vehicle. Those sonar data are converted from raw range and angle measurements to sonar-relative 3D points, and ultimately to a set of geolocated latitude, longitude, and depth points. Lastly, the collection of points is typically processed using a gridding method or surface model to produce a coherent view of the bathymetry. If the survey vehicle's position is known with adequate accuracy, it is possible to create a map directly from the recorded data. If the vehicle's position estimate is too inaccurate and known to degrade over time, SLAM techniques can be employed.

Gridding methods, which divide the survey area into a set of (x, y) bins, can be used in real-time mapping techniques when a vehicle's navigation solution is sufficiently precise. Points that fall in each bin are averaged to produce a single depth estimate and a depth variance to convey the uncertainty. Although gridding is simple to implement and quick to compute, there is an inherent trade-off between resolution and completeness. To obtain high resolution maps, a small cell size must be used. A small cell size will, however, include fewer points to average, which will result in noisy depth estimates with poor per-cell statistics. Some cells within the map may not contain any points at all, producing gaps in the model. Such gaps can be filled in by various non–statistically rigorous methods such as direct interpolation or weighted averaging. Using larger grid cells will result in fewer gaps but the level of detail in the final map will be reduced. When the underlying point density is not uniform, picking a single appropriate cell size for the entire survey area is difficult.

Various methods of gridding, meshing (Scheidegger et al., 2005), and smoothing have been proposed to generate a surface from 3D point cloud data. Many of these methods are geared toward imaging and 3D modeling. Although these fixes produce reasonable looking results, they do not model bathymetry or terrain in a statistically rigorous way. Several stochastic models exist for processing bathymetry such as CUBE (Calder, 2003), Kriging (Cressie, 1990), and Gaussian process regression (GPR) (Vasudevan et al., 2011). These methods seek to predict a continuous surface similar to spline or meshing methods. Stochastic models, however, have mathematically rigorous roots and provide uncertainty metrics to quantify the uncertainty in the depth estimates over the domain of the model.

GPRs have been effectively used to model terrestrial terrain (Lang et al., 2007; Vasudevan et al., 2009) and underwater bathymetry (Barkby et al., 2012). Figure 1 shows a comparison between a standard gridded data product using shipboard sonar data from E/V *Nautilus* and the GPR solution described here. The GPR result shows detail that was smoothed away in the gridded model. The same map overlaid with the GPR confidence estimation (Figure 2) shows the utility of the accompanying error estimate. The well covered flat top of the seamount has low overall uncertainty. Areas of steep terrain sloping away from the vessel's path will have lower point density and more variability in the range data, which results in the higher uncertainty estimate.

Terrain models such as grids or GPRs have also been used to enforce self-consistency during mapping in featureless SLAM algorithms. Featureless SLAM seeks to constrain a vehicle's navigation trajectory estimate by matching observations directly against a representation of the terrain itself. In its prototypical 2D implementation for land-based applications, the SLAM problem consists of a vehicle with a forward looking LIDAR sensor and an odometry sensor (Balasuriya et al., 2016; Kohlbrecher et al., 2011). In this manner, individual LIDAR observations can be continuously compared with the world model and may provide a high degree of positional information. SLAM techniques for down-looking multibeam survey data, commonly called push-broom data collection, pose unique challenges. Consecutive sonar pings in push-broom surveys have little to no overlap between and a single sonar ping offers little constraining information relative to a previously generated representation of the seafloor.

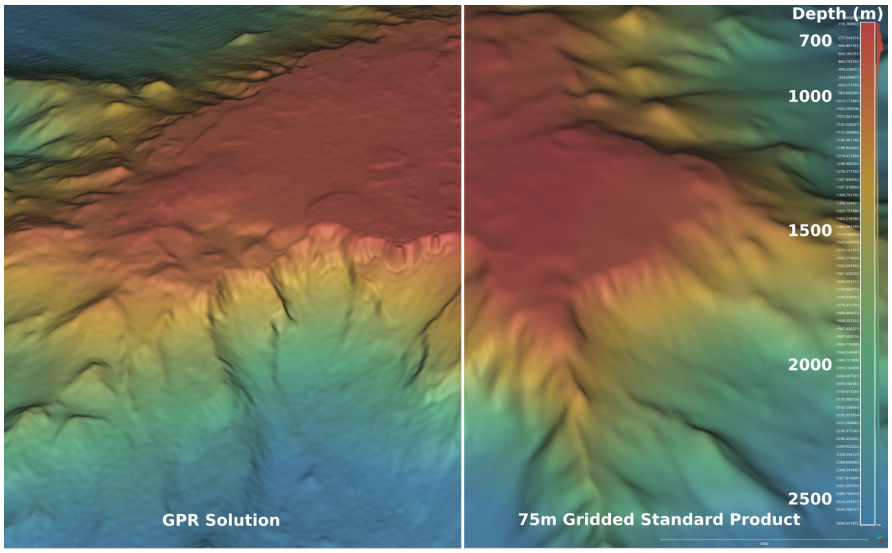


Figure 1. A comparison of "Seamount 6" in the Papahānaumokuākea Marine National Monument surveyed by E/V *Nautilus* in 2018 with an EM302 sonar and Seapath 330+ inertial measurement unit. The image on the right shows *Nautilus*'s standard 75-m gridded product. The image on the left shows the author's massively parallel GPR solution applied to the same data set.

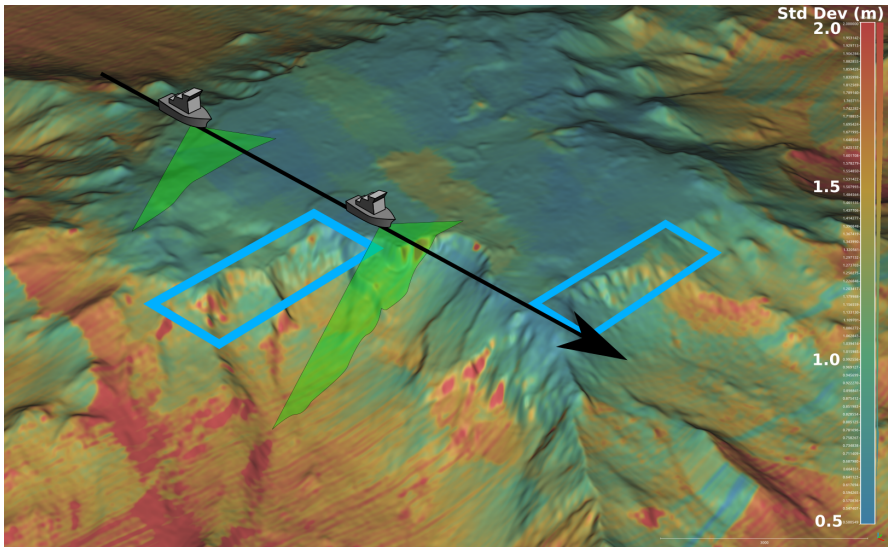


Figure 2. An example survey track on a GPR uncertainty map. Blue boxes represent areas that have limited observability from the ship's survey lines and subsequently lower quality bottom return data (higher uncertainty).

Roman and Singh (2005) proposed a solution to the push-broom SLAM problem using submaps generated from vehicle dead reckoning over a short time window. This approach assumes that the errors in the accrued odometry are small relative to the sonar resolution over a short enough time window and that the submap can be considered internally accurate. Submaps can then be compared to each other when they overlap to help constrain the vehicle position. This method was shown to produce results that are more accurate and self-consistent than simple dead reckoning alone or with long base line (LBL) aided navigation. The original submap method in Roman and Singh (2005) was formulated using an extended Kalman filter (EKF) and had computational limitations for large

numbers of submaps. The concept has since been refined several times to utilize more efficient factor graph frameworks (Bichucher et al., 2015; Torroba et al., 2019; VanMiddlesworth et al., 2015; Vaughn, 2015).

The initial formulation of submap SLAM used a simple gridder to model the terrain and estimate the alignment between submaps with simple 2D correlation followed by iterative closest point (ICP) matching. This two-step process increases robustness of the solution since ICP is prone to finding local minima. Improved registration techniques using probabilistic sampling, variants of point-to-plane, and various error metrics (Massot-Campos et al., 2016; Palomer et al., 2016; Torroba et al., 2018) have also been used to improve robustness for bathymetric data. GPR has also been shown to be an effective tool in submap SLAM as a terrain model (Bore et al., 2018).

Barkby et al. (2012) proposed a method which uses a Rao-Blackwellized particle filter (Murphy, 1999) to model the uncertainty in the vehicle state known as bathymetric distributed particle SLAM (BPSLAM). In this formulation, each particle preserves its trajectory history and associated sonar returns produce a particle-specific representation of the map. These maps are represented as an x, y grid of the average and standard deviation of the z value. Finally, a particle's most recent observations are compared with its own gridded models to calculate a likelihood. Particles with the highest likelihood have the highest probability of producing child particles during the resampling step.

Barkby et al. also used a Gaussian process regression as the underlying terrain model (Barkby et al., 2012). Much like the gridded model, a GPR can be used to estimate the likelihood of a particle's trajectory map. The GPR method, however, improved the likelihood estimation because of the model uncertainty estimate. The major drawback of using a GPR is the significantly increased compute time, making this method less desirable or unusable for online mapping applications.

GPR has seen uses in other applications of SLAM and, more generally, terrain aided navigation (TAN). Hitchcox and Forbes (2020) use a long length scale GPR on gridded sonar data to model the low frequency signal in a high pass filter. Point matching similar to ICP is then performed only on the high frequency signal, increasing the likelihood of matching the fine scale details in the seabed shape. Peng et al. (2019) conducted simulations that show GPR to be a more effective terrain model than gridding for TAN using multibeam sonar. Ma et al. (2018) use an approximate (sparse pseudo-input Gaussian process) GPR model as a terrain model in a graph-based submap SLAM implementation.

The computational burden of stochastic models is the major drawback over traditional methods. Significant computation demands make stochastic models challenging for online processing on an autonomous vehicle or with very large data sets. SLAM navigation, such as Barkby et al. (2011), cannot be done in real time or used to guide the vehicle survey path. Additionally, without a real-time GPR, information gain path planning methods are not feasible. Typically, GPR performance is improved through various approximate computation methods (Gramacy and Apley, 2015; Qi et al., 2010; Rasmussen and Williams, 2006; Snelson and Ghahramani, 2005). GPR solutions have been implemented on graphical processing units (GPUs) with promising results (Franey et al., 2012; Gramacy et al., 2014) but have not been tailored for situations that require online updates and real-time computation.

Several GPU accelerated libraries like GPFlow and GPTorch exist for computing GPR solutions. However, they were conceived for post-processing applications and lack a few critical features, like real-time recursive online Cholesky updates, that enable our real-time mapping and SLAM solutions. When using these libraries, significant slowdowns may be encountered when applying non-GPU accelerated operations to intermediate products which require expensive GPUs to host machine memory transfers. As such, it was decided to create a solution based directly on the CUDA API with all necessary intermediate steps implemented on the GPU.

Our efforts focus on improving the computational performance of an updateable GPR model by utilizing the large number of parallel processor cores on modern GPUs such that GPRs can be useful for real-time bathymetric mapping tasks. In this paper, we present a novel method of computing complete GPR solutions on large data sets using a GPU known as massively parallel

GPR (MP-GPR). Additionally, we present a graphics processing unit/Gaussian process bathymetric distributed particle SLAM (GP-BPSLAM), which uses the BPSLAM framework with MP-GPR as a terrain model.

Although MP-GPR is a viable terrain model for submap SLAM (Ma et al., 2018), it was decided to focus on BPSLAM for this study. BPSLAM has several advantages over submap SLAM. Particle filter methods like BPSLAM are able to maintain multiple hypotheses. This makes particle filter SLAM more robust to false hypotheses that can cause optimization-based methods like submap SLAM to diverge after incorrect submap registrations. BPSLAM can also be run in real time to aid vehicle navigation. Additionally, submap SLAM is not able to rectify errors in the submaps themselves. In practice though, this is a minor issue when using high quality navigation sensors since the submap size can be reduced until it contains no meaningful errors.

GPRs were shown to be effective in BPSLAM (for post-processing) in Barkby et al. (2011). Additionally, because they are able to support multiple hypotheses, Rao-Blackwellized particle filter SLAM algorithms like BPSLAM are more robust to odometry errors. This further increases the utility of BPSLAM for real-time applications. The hierarchical nature of the BPSLAM particle/trajectory model is also well suited to our recursive GPR model.

BPSLAM's major drawback over submap SLAM is the lack of particle diversity. The navigation solution can only be as good as its best particle trajectory. In its current form, BPSLAM has no way to optimize past trajectories. In theory, a submap SLAM solution can produce a more precise navigation solution.

In Section 2 we introduce the basic GPR formulation and then cover its parallel implementation in Section 3. This method can compute GPRs several orders of magnitude faster than conventional methods, thus making GPR viable for real-time applications on large data sets. Additionally, our GPR computation supports efficient, recursive, online updates of its terrain model. We then demonstrate this in two applications. In the first application, Section 4, we assume a known navigation solution and generate a GPR solution in real time. This was used as a way to test both the memory and computational performance of the GPR model. We also present GP-BPSLAM, which uses the BPSLAM framework with an efficient GPR implementation of the terrain model in Section 5. Sections 6 and 8 then follow with comments on the GP-BPSLAM approach and additional thoughts on further refinements to the method.

2. Gaussian Process Regression

A Gaussian process (GP) is specified by a mean and covariance

$$\begin{aligned} \mu(\boldsymbol{x}) &= E[y(\boldsymbol{x})], \\ k(\boldsymbol{x}, \boldsymbol{x'}) &= E[(y(\boldsymbol{x}) - \mu(\boldsymbol{x}))(y(\boldsymbol{x'}) - \mu(\boldsymbol{x'}))], \end{aligned} \tag{1}$$

where $y(\boldsymbol{x})$ is a real process with a mean $\mu(\boldsymbol{x})$ and a kernel (covariance function) $k(\boldsymbol{x}, \boldsymbol{x'})$. Equation 1 is then rewritten as

$$y(\boldsymbol{x}) \sim \mathcal{GP}(\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'})), \tag{2}$$

where it is assumed that the joint distribution of inputs $\boldsymbol{x}$ and the outputs $y(\boldsymbol{x})$ are normal. Equation 2 can also be expressed as

$$y(\boldsymbol{x}) \sim \mathcal{GP}(\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'})) = \mathcal{N}((\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'})). \tag{3}$$

Finally, to fully define the GP we set the mean function μ to be zero and select the square exponential as our kernel with a characteristic length scale l, process noise σ_f^2, and observations $\boldsymbol{x}_i, \boldsymbol{x}_j$:

$$\begin{aligned} \mu(\boldsymbol{x}) &\equiv 0 \\ k(\boldsymbol{x}_i, \boldsymbol{x}_j) &\equiv \sigma_f^2 \exp\left(-\frac{|\boldsymbol{x}_i - \boldsymbol{x}_j|^2}{2l^2}\right). \end{aligned} \tag{4}$$

These kernel and mean function selections are common, general purpose choices (Rasmussen and Williams, 2006). In practice, however, the covariance function is represented using the exactly sparse approximation

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) \equiv \begin{cases} \sigma_f^2[(\frac{2+cos(2\pi\frac{d}{l})}{3}(1-\frac{d}{l}) + \frac{1}{2\pi}sin(2\pi\frac{d}{l})] & d < l \\ 0 & d \geq l, \end{cases} \tag{5}$$

where d is the distance between the points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ (Melkumyan and Ramos, 2009). This approximation defines the covariance beyond the characteristic length sale to be zero while still closely approximating the square exponential kernel when the points are close. Increasing the number of zero elements in this way greatly increases the performance when using sparse matrix representations.

With the GP fully defined, it is now possible to calculate a realization of the GP as

$$\boldsymbol{Y}_* \sim \mathcal{N}(\boldsymbol{0}, k(\boldsymbol{X}_*, \boldsymbol{X}_*)). \tag{6}$$

Here, $\boldsymbol{Y}_*$ is the output prediction at the inputs specified as $\boldsymbol{X}_*$. In order to make predictions based on training data, the formulation of the GP must be restructured as follows:

$$\begin{aligned} \begin{bmatrix} \boldsymbol{Y} \\ \boldsymbol{Y}_* \end{bmatrix} &\sim \mathcal{N}\left(0, \begin{bmatrix} \boldsymbol{V} & \boldsymbol{K}_*^\intercal \\ \boldsymbol{K}_* & \boldsymbol{K}_{**} \end{bmatrix}\right), \\ \boldsymbol{V} &= k(\boldsymbol{X}, \boldsymbol{X}) + \boldsymbol{W}, \\ \boldsymbol{K}_* &= k(\boldsymbol{X}_*, \boldsymbol{X}), \\ \boldsymbol{K}_{**} &= k(\boldsymbol{X}_*, \boldsymbol{X}_*). \end{aligned} \tag{7}$$

We define our training data set of M three-dimensional points with 2D (horizontal position) input points $\boldsymbol{X} = \{x_i, y_i\}_{i=1}^M$ and 1D (depth) corresponding output points $\boldsymbol{Y} = \{z_i\}_{i=1}^M$. Similarly, we have test points $\boldsymbol{X}_*$ and predicted points $\boldsymbol{Y}_*$. The complete input covariance, $\boldsymbol{V}$, is the sum of the kernel covariance of the input points, $\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X})$, and the sensor noise, represented with the term $\boldsymbol{W} = \text{diag}(\sigma_1^2 \cdots \sigma_M^2)$. The term $\boldsymbol{K}_*$ is the covariance matrix of the prediction locations and the training data and $\boldsymbol{K}_{**}$ is the covariance of the prediction points with themselves. Finally, we can calculate the conditional distribution with mean $\hat{\boldsymbol{Y}}_*$ and covariance $\boldsymbol{\Sigma}_*$ of a predicted data set based on the training data set pairs $\mathbf{X}$ and $\boldsymbol{Y}$as

$$\boldsymbol{Y}_* | \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{X}_* \sim \mathcal{N}(\hat{\boldsymbol{Y}}_*, \boldsymbol{\Sigma}_*), \tag{8}$$

$$\hat{\boldsymbol{Y}}_* = \boldsymbol{K}_* \boldsymbol{V}^{-1} \boldsymbol{Y}, \tag{9}$$

$$\boldsymbol{\Sigma}_* = \boldsymbol{K}_{**} - \boldsymbol{K}_* \boldsymbol{V}^{-1} \boldsymbol{K}_*^\intercal. \tag{10}$$

The main free parameters in the GPR are the characteristic length scale and process noise in the square exponential kernel. These parameters are known as the hyperparameters and will affect how closely the predictions follow the input data. Since the input data for different vehicle configurations and survey parameters will change, it is desirable to have a procedure to determine an appropriate choice of σ_f and l based on the data. The details for an optimization method are given in Appendix A.

In its basic form, a GPR is relatively easy to implement, involving only basic matrix operations like inversion, addition, and multiplication. However, this simple implementation does not scale well with time or memory usage for larger data sets. In the next section, we suggest several ways to both decrease memory usage and restructure the computation in a way conducive to massively parallel GPU computation.

3. Massively Parallel Implementation

To implement the massively parallel formulation of the GPR, we need to restructure the standard implementation. In general, when utilizing GPU computation, it is desirable to work on chunks of data that can be solved in parallel as opposed to individual data points. Furthermore, it is desirable

to reduce costly memory transfers between the host machine and GPU. Our implementation requires no such memory transfer for large intermediate calculations such as the $\boldsymbol{V}$ or $\boldsymbol{K}$ matrices. Only the $2 \times M$ input data are transferred to GPU memory and the M element $\boldsymbol{\Sigma}_*$ and $\hat{\boldsymbol{Y}}_*$ vectors are transferred to CPU memory.

In practice, we compute the solution listed in Equation 8 in four major steps:

- Spatially organize observations into "blocks" of n points.
- Transfer blocks to GPU memory and compute their covariance.
- Update the Cholesky factor for the entire covariance matrix with the new covariance block.
- Compute the expected value, $\hat{\boldsymbol{Y}}_*$ (Equation 9), and covariance, $\boldsymbol{\Sigma}_*$(Equation 10), of the solution and transfer them to CPU memory.

3.1. Input Data Organization

The data input assumes that each sequential multibeam sonar ping returns a point cloud of globally referenced $[x, y, z]$ points. These sonar points are divided into memory "blocks," each containing an equal number of n points. These blocks should be assembled such that their points are closely located in the same spatial region. Closely (spatially) packed points minimize the zero values in their covariance matrix $k(block_1, block_1)$ (Section 3.2).

3.2. GPU Memory Transfer and Covariance Computation

Traditionally, covariance matrices are computed by solving Equation 4 for every element of the covariance matrix, requiring n^2 processes, where n is the number of input points. In a massively parallel implementation, it is possible to compute the covariance for thousands of matrix cells simultaneously. In this case the computation requires $n^2/threads$ operations, where the number of concurrent threads can be greater than 4000 on a modern GPU. Figure 3 provides an illustration. Roughly speaking, we can compute a 4000-element covariance matrix on a GPU in comparable time to computing the variance of a single matrix cell on the CPU.

To generate the covariance matrix $K(\boldsymbol{X}, \boldsymbol{X}')$ (Equation 4), we specify the training and prediction points $\boldsymbol{X}$ and $\boldsymbol{X'}$. Since we want to calculate the covariance of 2D points, these data are represented as a $2 \times n$ matrix. Matrices $\boldsymbol{X}$ and $\boldsymbol{X'}$ are typically generated on the host machine which means they

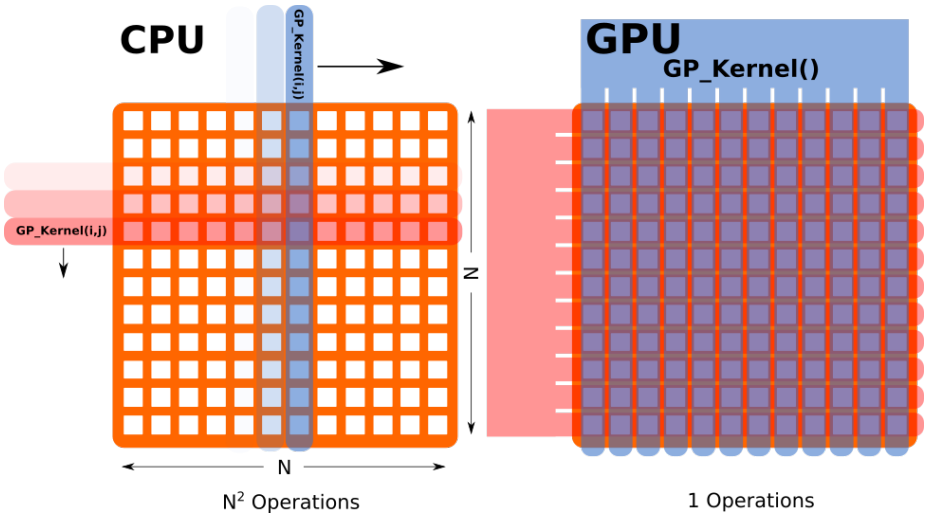


Figure 3. A visual comparison of computing the covariance matrix $K(\boldsymbol{X}, \boldsymbol{X}')$ using the CPU and the GPU. The CPU needs to evaluate each (i, j) cell sequentially whereas the GPU evaluates all of the cells simultaneously.

must be transferred to GPU memory before the covariance matrix can be computed (NVIDIA et al., 2020). Once the data are on GPU memory, n^2 cores are allocated to a GPU function known as a CUDA kernel (not to be confused with a GPR kernel). This CUDA kernel will compute Equation 5 for every cell in the covariance matrix in parallel.

3.3. Compute the Inverse of $\boldsymbol{V}$

Direct matrix inversion is computationally expensive. Fortunately, the matrix $\boldsymbol{V}$ is positive definite and symmetric, which means the inverse problem $\vec{b} = \boldsymbol{V}\vec{x}$ can be solved through Cholesky decomposition rather than direct inversion. Cholesky decomposition allows us to represent $\boldsymbol{V}$ as

$$\boldsymbol{V} = \boldsymbol{L}\boldsymbol{L}^\intercal, \tag{11}$$

where $\boldsymbol{L}$ is the lower triangular Cholesky factor. Using this decomposition, we can restructure the inverse problem $\vec{b} = \boldsymbol{V}\vec{x}$ as $\vec{b} = \boldsymbol{L}\boldsymbol{L}^\intercal\vec{x}$. To solve the system $\boldsymbol{V}\vec{x} = \vec{b}$ using the Cholesky factor, we can simply compute

$$\vec{c} = \vec{b}/\boldsymbol{L},$$
$$\vec{x} = \vec{c}/\boldsymbol{L}^\intercal.$$

Because $\boldsymbol{L}$ is lower triangular the above steps can be achieved by back and forward substitution (Rasmussen and Williams, 2006). We define $\vec{x} = CholeskySolve(\boldsymbol{L}, \vec{b})$ as the solution to $\vec{b} = \boldsymbol{L}\boldsymbol{L}^\intercal\vec{x}$.

The Cholesky factor $\boldsymbol{L}$ can be quite large to represent as a dense matrix. Sparse matrix representations allow for much lower memory usage but are less conducive to massively parallel computation due to their poor random access. Therefore, a hybrid method known as block compressed sparse row (BSR) (NVIDIA et al., 2020), which represents a matrix as a sparse supermatrix of dense submatrices, is used. In this format an $M \times M$ matrix can be stored as an $\frac{M}{n} \times \frac{M}{n}$ matrix of $n \times n$ dense blocks where n is the number of observations in each block (as described in Section 4.1). The BSR format allows us to reap the benefit of dense matrices for parallelization while gaining most of the memory compression of a purely sparse implementation.

The BSR representation of $\boldsymbol{L}$ means we can implement a recursive computation to update the Cholesky factor when new data become available (Figure 4). A detailed explanation of this recursive Cholesky update is available in Appendix A.

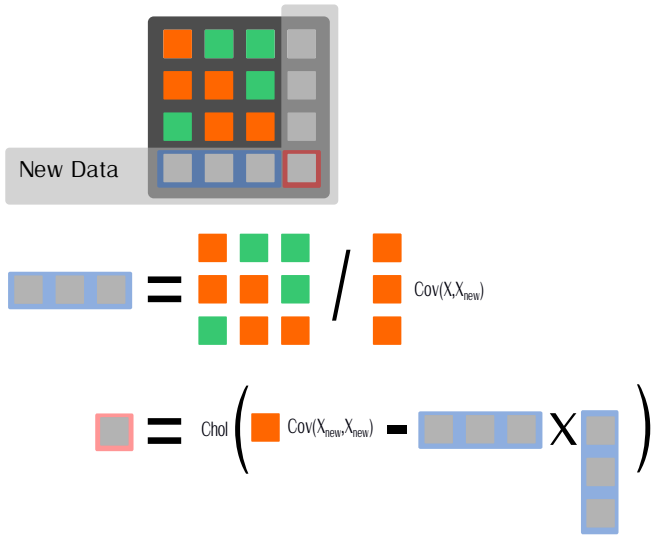


Figure 4. A visual representation of updating the Cholesky factor. A: New data require a new lower row in the Cholesky factor. B: The blue elements of the row are determined by back substitution. C: The red elements of the new row are solved for by dense Cholesky decomposition, multiplication, and subtraction.

3.4. Computing the Expected Value and Covariance

The expected value and variance of the solution can be calculated by writing Equations 9 and 10 in terms of the Cholesky factor $\boldsymbol{L}$ as

$$\hat{\boldsymbol{Y}}_* = \boldsymbol{K}_* CholeskySolve(\boldsymbol{L}, \boldsymbol{Y}), \tag{12}$$

$$\boldsymbol{\Sigma}_* = \boldsymbol{K}_{**} - \boldsymbol{K}_* CholeskySolve(\boldsymbol{L}, \boldsymbol{K}_*). \tag{13}$$

Once the blocks are added to a GPR solution space a prediction for a set of desired locations can be made. In the case of online GPR mapping, an intermediate prediction can be made after any Cholesky update step. This prediction can be updated at any time by simply recomputing the mean and variance with relatively low computational cost. The $\boldsymbol{\Sigma}_*$ and $\hat{\boldsymbol{Y}}_*$ vectors can be transferred from GPU to host memory when they are needed with little cost to performance.

The GPR solution described here has been used as a basis for two application areas, real-time mapping and particle filter SLAM. Most of the differences between the applications are related to the organization of the data. Additionally, the forking nature of our particle filter SLAM solution will require updating several branching Cholesky factors in parallel.

4. Application: Real-Time Mapping

Here, we examine a typical surface vessel mapping configuration with a known GPS-based navigation solution and multibeam sonar data. Such a configuration allows for evaluation of the real-world performance of our GP-GPR implementation using streaming real-world data independent of the navigation solution. The shallow water test data presented here were collected with the Pontoon of Science (POS) in January 2020 in St. Mary's, Georgia. The details of the platform, sonar, and the raw data are available as a data paper (Krasnosky et al., 2021).

4.1. Data Structure

It is necessary to divide up the entire problem into smaller parts to limit memory usage (Section 3.1). For this application, blocks of data are divided up into a data structures known as tiles, where each tile represents a complete GPR solution area. Each tile's "prediction region" represents the area that we want to predict with that tile. Tiles also include a "training region" that extends beyond the prediction region and overlaps neighboring tiles by at least one characteristic length scale on all sides (Figure 5) to enforce smoothness between tiles. Any block of input points that falls within the training region of a tile will be used to train that tile's associated GPR. To calculate a GPR over an arbitrarily large area multiple tiles with adjacent prediction regions can be defined. Because tiles have overlapping training regions, individual blocks of input data can be associated with multiple tiles. Tiles reference the same block in memory without duplication to avoid unnecessary data redundancy.

4.2. The Real-Time Algorithm

The real-time algorithm is divided into two main subroutines. The first divides up the data into blocks and associates the blocks with the tiles. Any tiles linked to new data in this process are added to the end of a tile queue. A second concurrent process continuously computes the GPR for the prediction region of the tile at the front of the queue as in Sections 3.3 and 3.4. If the same tile is at the front of the queue for two consecutive updates, its Cholesky factor is simply updated with any new data blocks instead of being recomputed. If a different tile is at the front of the queue, the Cholesky factor of the previously predicted tile is cleared to save memory.

4.3. GPR Performance

We have restructured the problem to make large GPRs computationally tractable. The Cholesky factor is represented using BSR and the problem space has been divided into multiple smaller GPRs

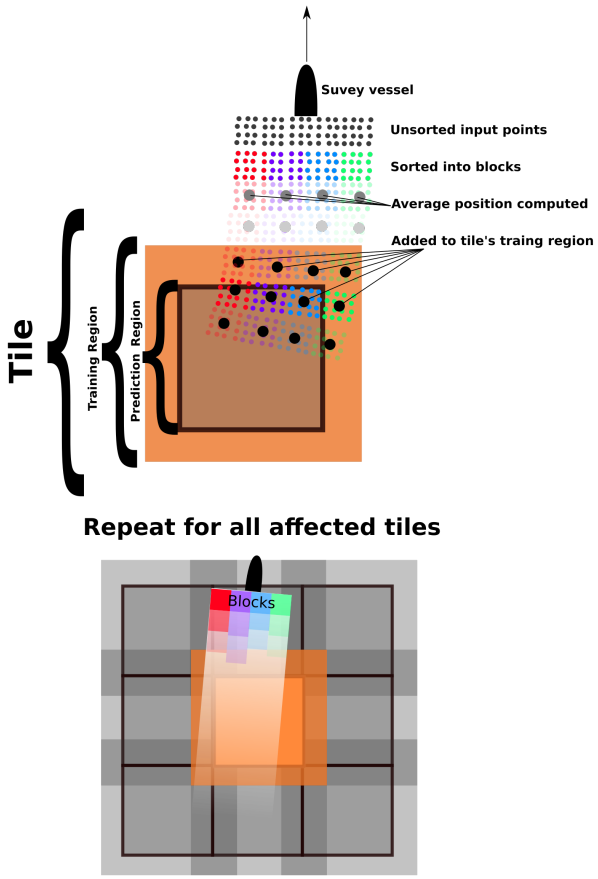


Figure 5. A schematic of the input data tiling scheme used. The orange area represents the area of the tile we are examining; the grey represents neighboring tiles. The survey area is sectioned into tiles that overlap on their borders. Data within each tile, including the overlapping regions, are used to train a GPR. The GPR solutions will be computed over smaller nonoverlapping prediction regions. The individual input points are grouped into blocks and associated with tiles that contain the block's average center location.

represented as tiles with training margins. Varying the block size and changing the tile and training margin size have implications on performance.

Varying the size of the entire training region (prediction region + training margin) has implications on both memory and compute speed. A large tile will have a relatively small training margin, reducing the amount of redundant computation (of the overlapping training margin) when moving to adjacent tiles. However, large tiles will encompass more points. This will require more memory to store their Cholesky factor and take more time to compute per tile. In practice, the tile size is selected to use 25–50% of the GPU's memory on average. This ensures that almost all tiles will fit into memory. If a tile does not fit in memory, approximate methods of computation must be used. The approximate method used for this project is the subset of data approximation (Rasmussen and Williams, 2006) for its ease of implementation and compatibility with the Cholesky decomposition method of computing GPRs.

There is also a trade-off between memory usage and compute time when using the BSR format to solve the GPR. Larger data blocks allow the GPU to perform more operations in parallel, thus reducing the overall compute time. Increasing the block size will decrease compute time until the GPU saturates (runs out of unused threads). In terms of memory, the BSR format represents a matrix by dividing it up into an array of dense blocks. If a block contains all zeros, it is not stored to save memory. If any nonzero elements fall in a block, that entire block must be stored regardless

Algorithm 1. Real-time process to compute a GPR from streaming data.

```
Add Data Subroutine:
Input: 3D point cloud as training data, Length Scale, Tile prediction dimension
Output: the tile_queue
while running do
    generate blocks from incoming data (sec. 3.1)
    if a block fills all n points then
        add tiles to the back of the tile_queue;
    end
end
Compute GPR Subroutine:
Input: tile_queue  length scale, process noise
Output: a point cloud representing the GPR prediction for each tile
while running do
    if queue has tiles then
        if working_tile ≠ tile_queue.front() then
            clear working_tile's Cholesky factor
        end
    end
    set working_tile = tile_queue.front()
    forall new_block in working_tile do
        do online Cholesky update for new_block (Section 3.3)
    end
    use the updated Cholesky factor to make a prediction (Section 3.4)
    remove tile from queue
end
Main Process:
Define: tiles size, block size, training regions size, hyperparameters (l,sigma)
begin Add Data Subroutine
begin Compute GPR Subroutine
```

of the number of zeros it contains. Therefore, increasing the block size will increase the number of zeros that must be stored in dense blocks and increase the overall memory requirements.

To illustrate the effect of block size, an experiment was conducted on a short section of a survey with a single overlapping "Williamson turn" as shown in Figure 6. This test has examples of overlapping and nonoverlapping data, and is representative of a larger survey. For each block size iteration, the survey is divided spatially into the same tiles while the incoming data points are organized into blocks and added to their respective tiles as described in Section 4.1.

Initially, as the block size increases, the compute times decrease drastically as the GPU utilization increases. However, around block sizes of 400–800 points, the GPU begins to saturate and only moderate gains are made. This trend is visible in Figure 7. At small block sizes, the computation is similar to the conventional solution described in Wilson and Williams (2018). Theoretically, at a block size of 1, the solution is computationally equivalent to the conventional GPR solution with a sparse matrix representation on a CPU. Block sizes of 32 and below become intractable. Although single core performance is significantly better on a GPU this trend shows that CPU-only computation is not feasible.

Figure 8 shows a comparison of memory usage per tile vs block size. With our method, memory use increases roughly linearly with block size and ranges from about 100 MB to 1 GB. (Most current mid- to high-end GPUs have at least 8 GB of memory.) In general, it is best to select the smallest possible block size that saturates GPU compute time. In this case, a block size of 400–800 is appropriate depending on memory limitations. It is worth noting that for ever larger block sizes the memory required will continue to grow. When the block size equals n, the BSR format requires the same memory as dense representation. Lower memory usage per tile allows for larger tile sizes,

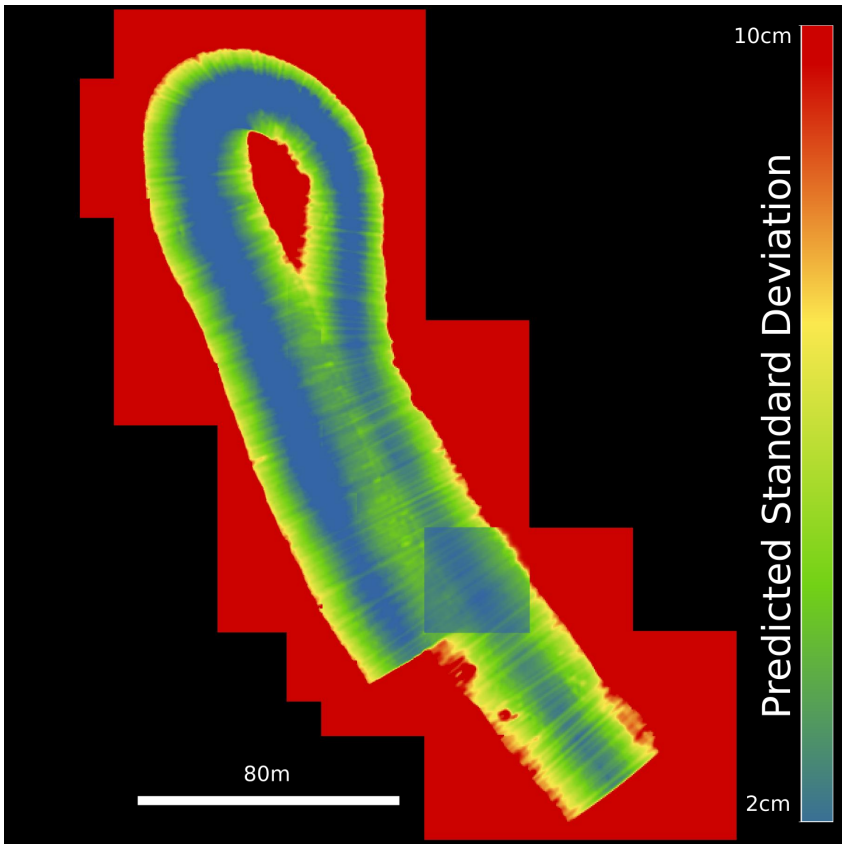


Figure 6. The GPR uncertainty map generated from the performance testing data set. The color scale represents the standard deviation of the solution predicted by the GPR. The highlighted tile corresponds to the memory block diagrams

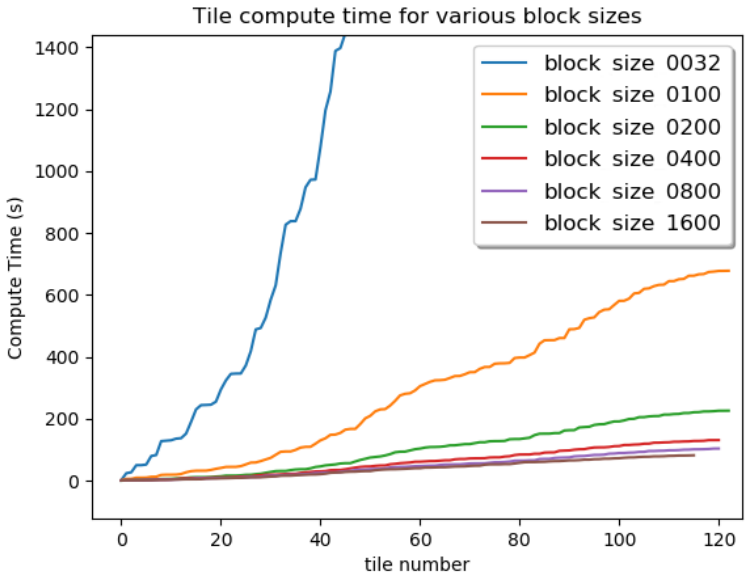


Figure 7. A comparison of varying block size on compute performance. The horizontal axis represents the number of tiles that have been processed so far and the vertical axis represents cumulative compute time (computed using NVIDIA RTX 2080 TI and Intel Core i7-9700k at standard clock speeds).

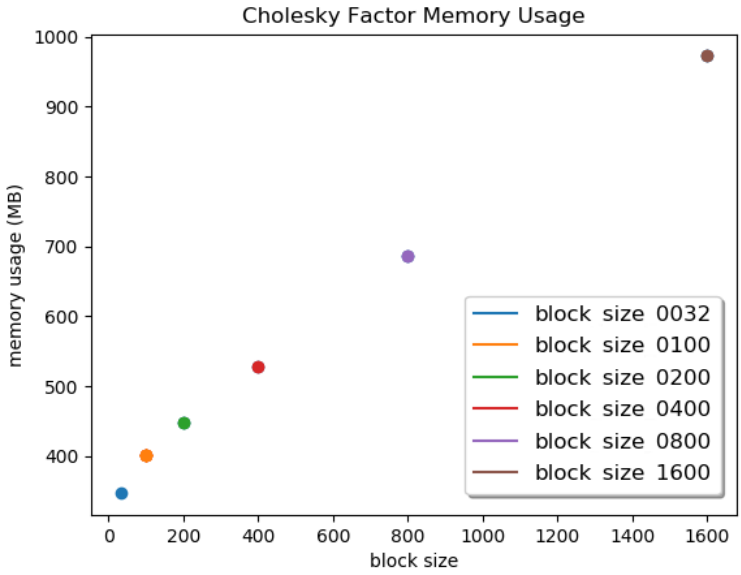


Figure 8. A comparison of memory usage vs block size while computing a single tile of a GPR solution.

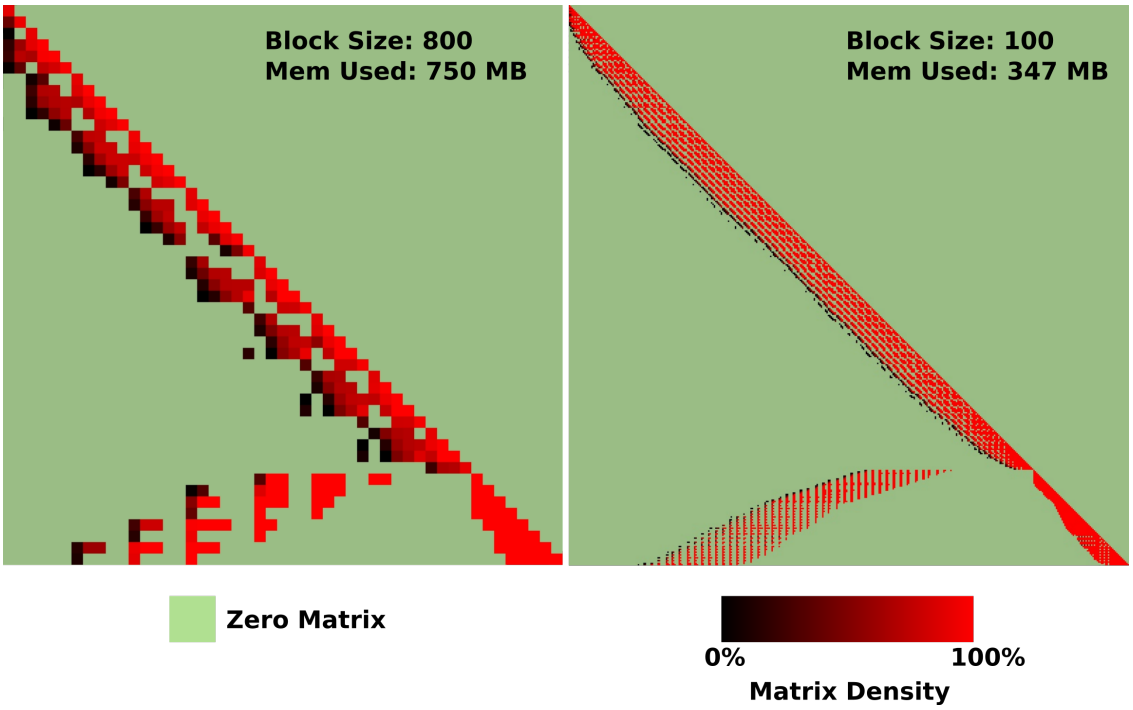


Figure 9. A comparison of the sparsity pattern using block sizes of 800 and 100. Green squares represent zero matrices and require no memory. Black-red squares represent dense matrices. The more red a square is, the fewer zero elements it has.

which allow for less redundant computing in the training regions and therefore less overall compute time. It is possible to further increase performance by enlarging tile sizes to fill available memory.

The detailed memory usage can be seen by comparing the sparsity pattern of the Cholesky factor. Figure 9 shows the sparsity pattern for the block sizes of 800 and 100. Each sparsity pattern is to scale and represents the same solution (with some minor difference due to differences in the tiling).

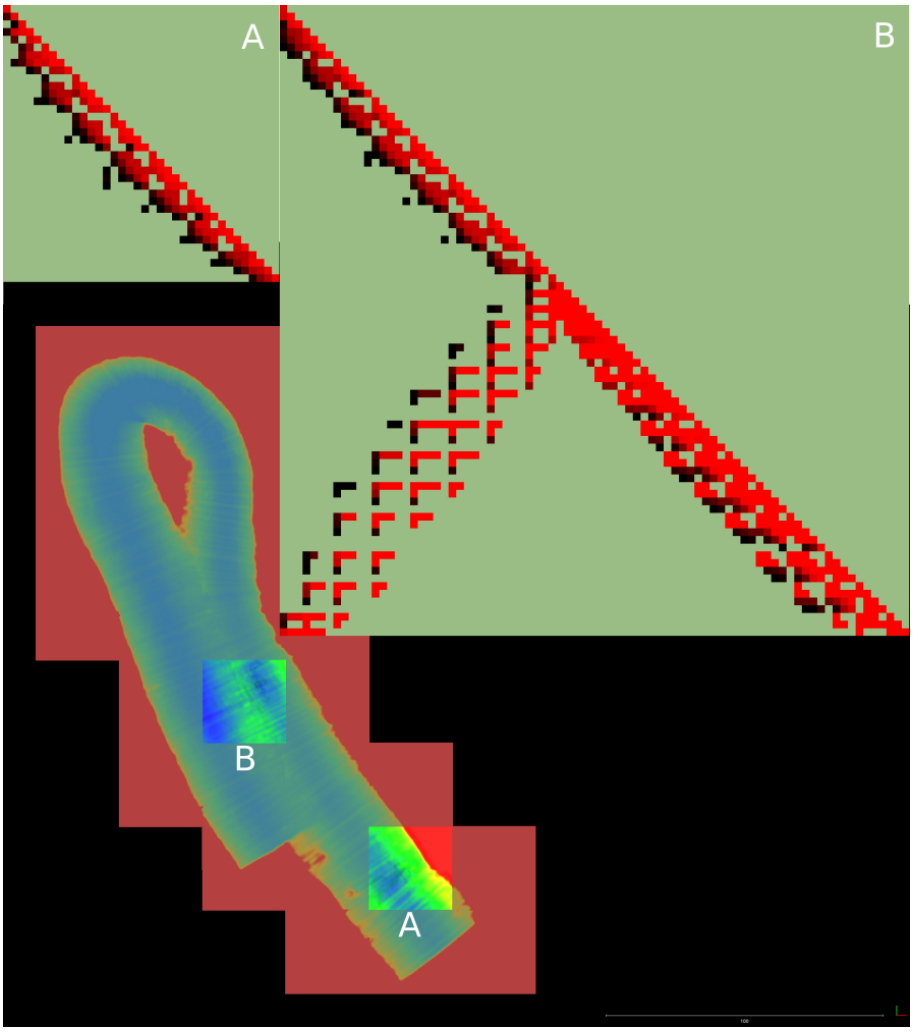


Figure 10. Some example Cholesky factors from the "Williamson turn" survey segment. Label A shows a tile from newly explored terrain resulting in a mostly diagonal Cholesky factor. Label B demonstrates a tile with significant overlap with previous data, resulting in additional off-diagonal elements in the Cholesky factor.

Although the size 800 case has fewer total blocks, the blocks are larger and not as densely packed. Block size of 100 requires many more blocks but they are smaller and on average more densely packed, resulting in overall lower memory usage.

The pattern of the survey itself will affect the performance of the GPR computation. Tiles with few correlated blocks will have a relatively sparse Cholesky factor. A single nonoverlapping survey line in unexplored terrain will produce a Cholesky factor that is only populated along the diagonal and relatively sparse (Figure 10). In areas with partially overlapping sonar swaths, off-diagonal elements will appear in the Cholesky factor and require more compute time and memory. Crossing survey lines, or areas with high block density, will have very dense Cholesky factors requiring the largest compute time and GPU memory. In general, our GPR solution will keep up with the data acquisition in most areas of a typical survey comprised of parallel track lines with partially overlapping data coverage. The prediction portion of the algorithm will fall behind in areas of very high block density, such as near crossing lines. In this situation tiles are being added to the prediction queue faster than they can be processed. The solutions will catch back up to real time when returning to areas with average block density and fewer affected tiles.

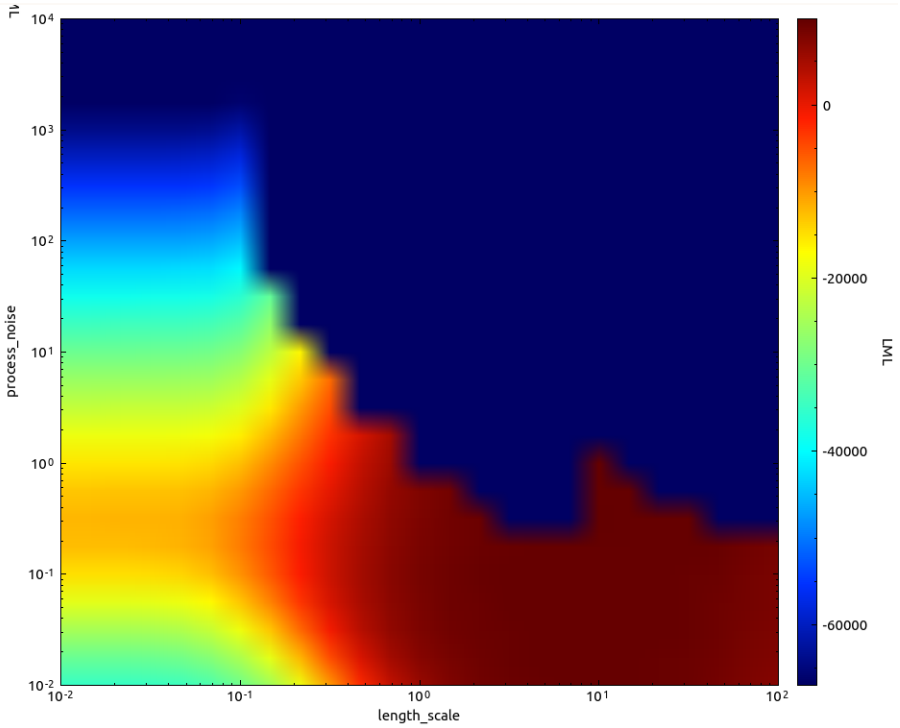


Figure 11. A plot showing LML varying with the length scale and process noise hyperparameters. This plot was generated from a small 8×8 m^2 subset of a survey. This plot has a peak near length scale of 12 m with a process noise of 0.2.

4.4. Log Marginal Likelihood Analysis

For these data, the GPR hyperparameters were determined by plotting the log marginal likelihood (LML) vs length scale and process noise (Figure 11). An 8×8 m^2 subset of the larger data set was selected to generate this plot to reduce memory requirements and computation time. Details of the LML calculation are given in Appendix A. Although this method is not time efficient, it allows us to study the likelihood trends. In this instance we find a peak near a length scale of 12 m with a process noise of 0.2. The large blue area in the top right of the plot represents an area where the GPR did not converge because the covariance matrix K was not sufficiently positive definite. Since our peak was surrounded by nonconvergent hyperparameter values we risk nonconvergence when using data dissimilar to what was used in this calculation. To prevent this nonconvergence, the process noise was relaxed to a value around 0.1. In practice, this will have little effect on our model but will significantly increase the numerical stability.

5. Application: GP-BPSLAM

Graphics processing unit/Gaussian process bathymetric distributed particle SLAM is a reformulation of BPSLAM (Barkby et al., 2011) that frames the problem in a way conducive to massively parallel computation on a GPU. We assume that the vehicle's depth, velocity, and attitude states can be tracked by a single shared EKF that is common to all particles. In practice, this EKF's odometry solution can be computed by an external process and is largely independent of the particle structure (Moore and Stouch, 2014). Once the raw odometry is available, the variance of x and y velocities is randomly sampled using the EKF's variance estimate to produce a plausible velocity error vector for each particle. The odometry solution evolves for the lifespan of the particle by integrating the raw odometry velocities and adding the sampled velocity error vector. This

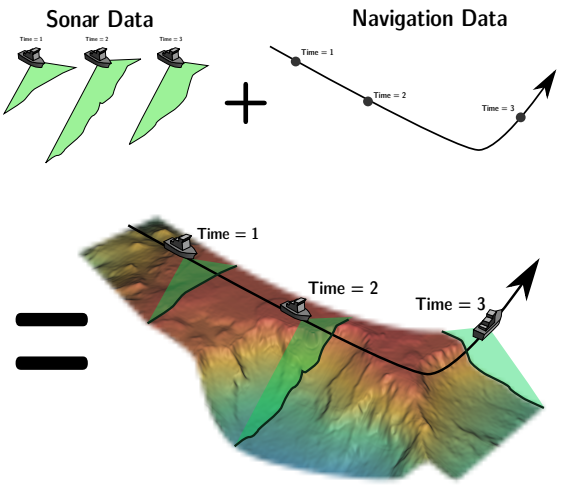


Figure 12. A schematic of how a single trajectory is associated with raw sonar returns to produce a trajectory map.

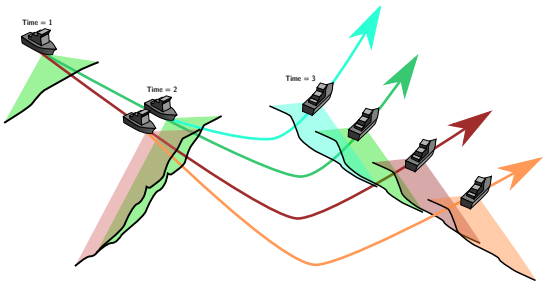


Figure 13. A graphical representation of the branching trajectory map structure. Notice how one set of multibeam data can generate many map permutations according to each particle trajectory.

complete six-degree-of-freedom odometry solution (x, y, z, roll, pitch, yaw) is stored as that particle's navigation hypothesis.

The sonar returns can be placed into particle-specific maps using each particle's navigation hypothesis (Figure 12). Each hypothetical map will include current and prior data potentially inherited from parent particle trajectories. A GPR is then used to quantify the likelihood of each particle's map (Section 5.1). Particles have a probability of being resampled proportional to their normalized map likelihood (Section 5.2). The particle trajectory and likelihood computation is then repeated for all child particles. This has the effect of producing branching particle trajectories with associated maps stemming from the location where the filter was initialized. This structure is known as a "trajectory map" and is detailed in Figure 13.

5.1. Computing Point Likelihood

Once a GPR is computed for a given particle's trajectory, a particle weight must be computed. With a GPR formulation, it is relatively easy to compute the probability that an observation matches the estimate, $z_{est} - z_{obs} = 0$. Since z_{est} and z_{obs} are both modeled as Gaussian the likelihood can be computed as

$$pointLikelihood = p(z_{est} - z_{obs} = 0) = \frac{\exp -\frac{1}{2}\frac{(\mu_{z_{est}} - \mu_{z_{obs}})^2}{\sigma^2_{z_{est}} + \sigma^2_{z_{obs}}}}{\sqrt{2\pi(\sigma^2_{z_{est}} + \sigma^2_{z_{obs}})}}. \tag{14}$$

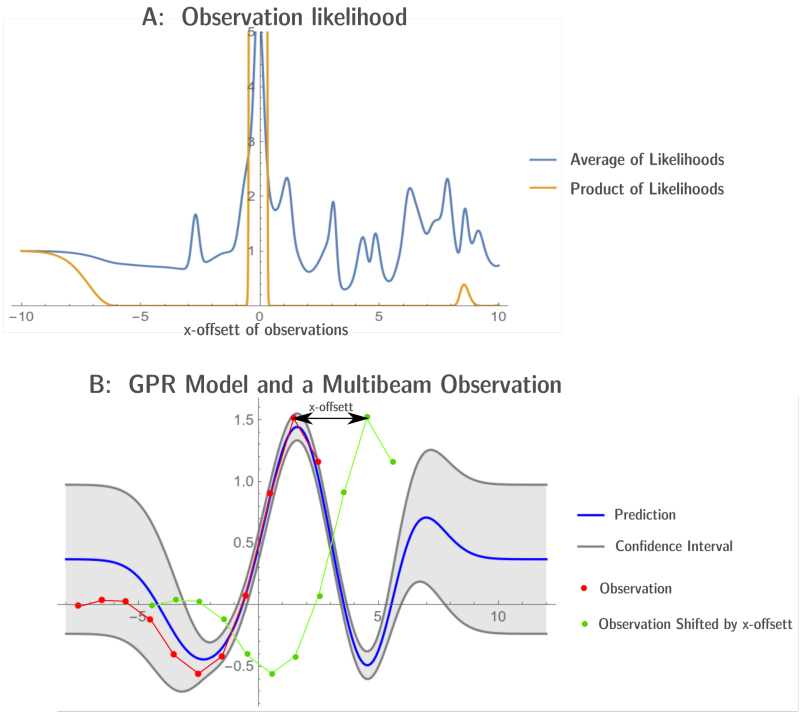


Figure 14. Illustration of the likelihood calculation comparing a set of new observations to an existing model. (a) The likelihood, calculated as function of shifting the observations relative to the model, showing a peak around zero offset. (b) The sample points shown matching the model best with no offset applied.

5.2. Weighting and Resampling

To compute the particle weight from the set of point likelihoods, Barkby suggests using a subset of data points overlapping past swaths and computing the joint likelihood (product of likelihoods). However, when using the joint likelihood extra care must be taken to remove outliers from the testing points. A single outlying point with a likelihood near zero can cause the joint likelihood for the entire set to be essentially zero. Therefore, in this method we use the mean of the point likelihoods to weigh particles. Since the GPR is a continuous function with known uncertainty, we need not filter out nonoverlapping points. Figure 14 demonstrates that including nonoverlapping points in likelihood computation is not detrimental to the computation. This example observation to compare with the GPR model was created (pictured in red). This observation was then translated along the x axis and its weight recomputed for each step using both the product of and sum of likelihoods methods. The results of each are plotted as a function of x offset in Figure 14(a).

Using this method, a set of observations with little to no overlap with previously collected data will approach a reasonably stable nonzero weight value. We call this the background weight, which can be thought of as having no information to judge the observations one way or the other. Observations with good alignment will have greater than the background weight. Those with poor alignment will have less than the background weight.

The set of particle weights is normalized from zero to one where the least likely particle has a weight of zero and the most likely a weight of one. The weight is used as the probability that a particle will be resampled. In this way, the particles with the highest probability of self-constancy have the highest probability of surviving the resampling step and spawning new particles. To save computation power, resampling is only performed on a set of particles if the combined training region for those particles overlaps a previous trajectory. This way if any particle is affected by a nearby (within one characteristic length scale) trajectory map, all of the particles will undergo resampling.

5.3. Implementation, Optimization, and Parallelization

Particle filters in general are conducive to parallelization because of the independent nature of each leaf particle. To exploit this structure, a framework can be developed to balance tasks best suited for CPU and GPU parallelization. Through the development process, it became clear that the computation time to produce the GPR maps was going to be the limiting factor. It was therefore decided to keep all particle filter–specific tasks on the CPU to allow the GPU to focus on the GPR computation.

Figure 15 shows a high level overview of the GP-BPSLAM algorithm and Figure 16 shows the core data structure of GP-BPSLAM particles. Each step of the algorithm and the relevant data structures are explained in this section.

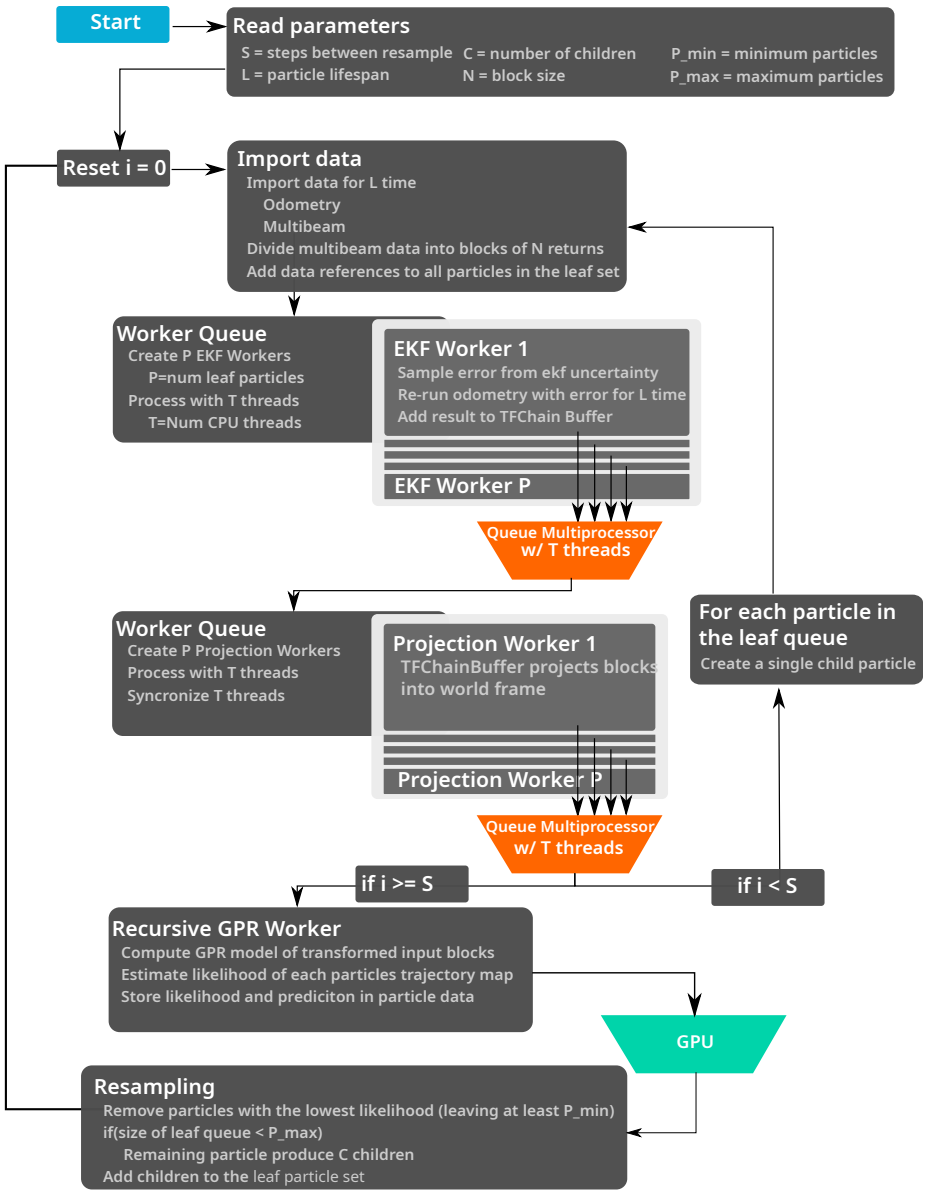


Figure 15. A high level flowchart of the GP-BPSLAM algorithm describing the loop that occurs for each leaf particle set's lifespan.

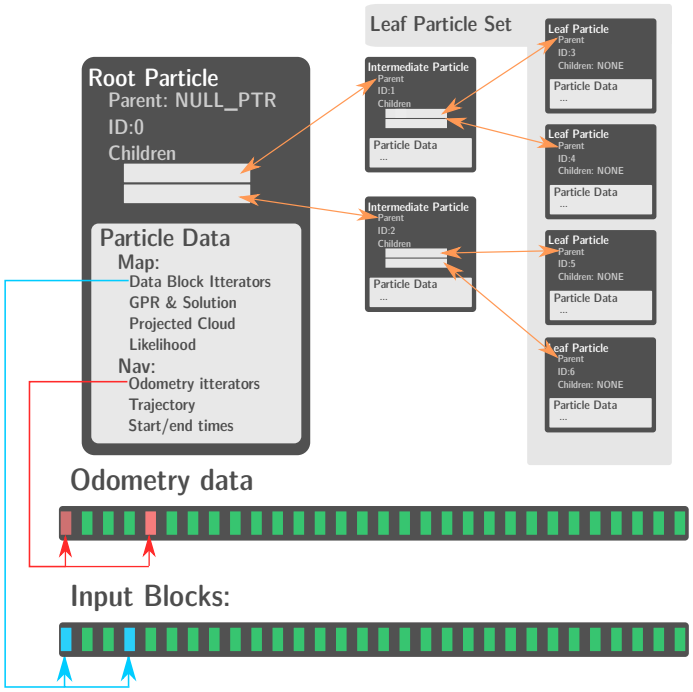


Figure 16. A diagram showing the basic structure of our particle tree. Each particle has a reference to a parent particle and a set of pointers to its children. Each particle has associated data which include iterator references to the input odometry and block arrays of sonar data. All other particle data are stored by the particle itself.

5.4. Particle Tree Structure

Particles are represented as templated container C++ objects, which allows the user to separate node and tree functionality from the stored data. Each particle stores a pointer to a parent particle, pointers to each child particle, a unique ID, and a generic type particle data.

The particle object has member functions to

- add child particles,
- iterate through child particles,
- get a pointer to its parent particle,
- check if the particle is the root particle,
- check if the particle is a leaf particle, and
- remove the particle from a tree.

By using these functions it is possible to form and maintain a tree representing forking particle trajectories. When a leaf particle is removed because of low likelihood, all parent particles with no remaining children are deleted recursively to save memory.

GP-BPSLAM also defines a particle trajectory data structure used by each particle that contains "map" and "nav" data, and provides member functions to compare itself with other particles. Nav data represent information pertaining to the trajectory hypothesis for that particle. It includes the trajectory start time, end time, a reference to the relevant input odometry, and the hypothetical trajectory generated from integrating velocity data and added noise. Since the input odometry is not unique to any one particle, only iterators to the beginning and end of the associated odometry are stored to save memory. Map data contain iterators to the front and back of the relevant blocked sonar data. The map data also store the results of the particle's associated GPR if computed. Optionally, a point cloud of all of the associated projected points for a particle can be generated to

visually see the map of a particle up to the current time. A graphical representation of the entire particle tree is shown in Figure 16.

5.5. TF Chain Buffer

The ROS TF library (Quigley et al., 2009) is used to perform all point cloud coordinate transformations. In its original form, the TF library is not able to handle the forking trajectory nature of our particle filter. Therefore, a *TFChainBuffer* extension was made to store data for a particle's lifespan and reference its parent particle. Using this method, no TF buffer data are repeated for the entire particle tree. Finally, all necessary members of the ROS TF buffer are overridden so the standard ROS TF interface is preserved for compatibility.

5.6. Data Import and Initial Processing

When new sonar data become available, they must be divided up into "blocks" of equal size (Section 3.1). To maximize compression in the Cholesky factor, the blocks should consist of tightly packed (in x, y) input points. At this step, the points are not transformed to a global frame, as in our the real-time mapping application. Instead, they are contained in a *SensorFrameBlock* data structure that may contain several smaller point clouds with different time stamps. A particle trajectory can then be applied as necessary to create a cumulative point cloud in the common map reference frame using the *TFChainBuffer*.

5.7. Asynchronous Workers

To maximize asynchronicity and maximize CPU utilization, a task handling system was developed. It consists of an abstract worker object and a worker queue. This abstract worker is inherited by child workers designed to complete a simple task and store all of the required input and output data. The workers can then be added to a thread-safe worker queue where the next unprocessed worker is claimed by a compute thread and processed. Additionally, every worker has a queue command. The default command is "run" which simply processes the worker. It is also possible to issue a "wait" command, which instructs the T threads processing the queue to wait for other threads to complete, and a "sync" command that synchronizes and stops all threads.

EKF Worker

The first type of worker for the particle filter is the EKF worker. This worker is responsible for adding noise to the selected states (usually x and y) and computing a hypothetical trajectory for the set particle's lifespan. The worker looks at the odometry of the particle's (or its parents') last position, randomly samples a velocity error using the EKF's uncertainty estimate, and integrates the velocity plus noise to compute the position for the particle's life.

Because all particles share a single EKF the odometry is generated by another process. In this implementation the ROS Robot Localization package (Moore and Stouch, 2014) is used to generate the odometry data. For each iteration of the GP-BPSLAM filter P EKF workers will be created, where P is the number of leaf particles.

Projection Worker

Once the EKF workers complete, the leaf particles are handed to the projection workers. These workers use the *TFchainbuffer* to project all the raw sonar sensor frame blocks to the global frame based on the particle's trajectory. Once in the map frame, blocks can be transferred to the GPU for processing by a GPR worker. This process needs to be completed for each leaf particle so P projection workers are added to the worker queue.

GPR Worker

A GPR worker is responsible for managing the GPR compute process on the system GPU (or GPUs). The data blocks associated with a worker's particle are added to the GPR (Section 3.3)

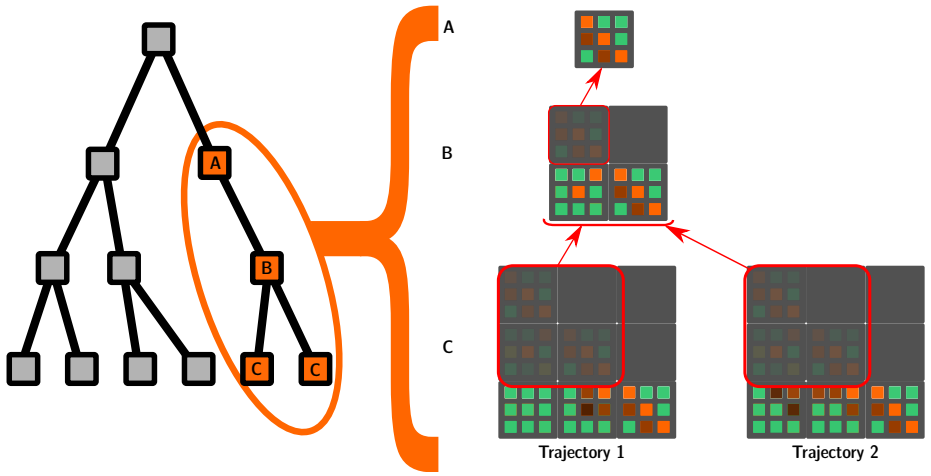


Figure 17. A graphical depiction of the recursive Cholesky factor representing the diverging trajectory maps. Labels A–C represent time steps. A: The initial block Cholesky factor is computed as in Section 3.3. B: The initial Cholesky factor is updated by adding several more blocks for this time step. The blocks in the red square are represented as pointers to those blocks created in step A. C: At this time two new particles have been created, creating additional branching trajectories. The two new Cholesky factors reference the blocks created in previous steps.

recursively updating the GPR's Cholesky factor. Next, the worker computes the likelihood of each of the leaf particle's sonar returns, as described in Section 5.2. P GPR workers are added to a special GPU *WorkerQueue*. The GPU queue is processed by G threads where G is the number of GPUs on the system (usually one). Each thread manages the computation on its respective GPU by performing memory transfers and starting execution.

Recursive GPR Worker

The recursive GPR worker is an extension of the GPR worker that utilizes the BSR format of our Cholesky factor (Section 3). In the particle recursive approach each dense matrix is stored in heap memory and referenced by a shared pointer stored by the supermatrix. This allows a super Cholesky matrix to be copied without copying its composite submatrices. In this way, a supermatrix can reference shared data with another supermatrix. The structure of either supermatrix can be modified or cells added without affecting the other. A submatrix will be deleted only when all references to that submatrix are deleted. We call this structure a "recursive Cholesky factor" (Figure 17).

The recursive GPR worker will work its way up the particle tree, adding training points to the Cholesky factor, until it reaches a leaf particle. At this point the Cholesky factor is used to make a likelihood prediction as in Section 5.1. Once a prediction is made, that Cholesky factor is no longer needed and is deleted to save memory. The next branch is followed to the next leaf particle in the same way. This branching approach can save significant computation time by reusing previously computed Cholesky factor components that were forked from the last computation (Figure 18). In practice, most of the branching in the recursive Cholesky factor occurs near the leaves of the tree, which helps the performance by reusing the Cholesky factors along the limbs. This tree exploration method also means the GPR worker will never use more memory than a single nonrecursive GPR worker.

5.8. Resampling

Before resampling, the average likelihood computed by the GPR worker is normalized and used as the particle weight to determine the probability that a particle will be resampled. Particles that are allowed to produce child particles maintain their trajectories as new parents. Particles that are not

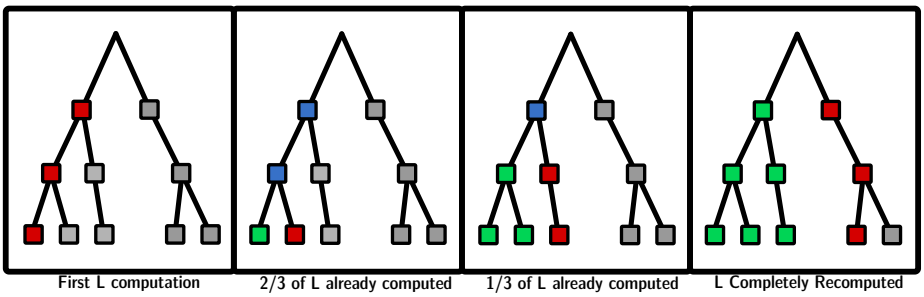


Figure 18. The recursive Cholesky factor tree exploration algorithm. Squares shown in red represent components of the Cholesky factor that must be computed. Blue squares represent Cholesky components that can be reused. Green squares show previously computed Cholesky components that have been cleared from memory.

resampled are deleted from the particle tree. All parent particles with no remaining children are also deleted.

5.9. Data Collection and Results

To generate a testing data set, a surface vehicle, the Pontoon of Science (POS), was once again used. This platform was designed with autonomous underwater vehicle (AUV) navigation sensors and two multibeam sonars. In this way, the POS can be used as an easily deployable analog to an AUV mapping platform while providing a ground truth navigation solution. A complete description of the POS's sensor suite and the raw data are available in Krasnosky et al. (2021). The real time kinematic (RTK) GPS position and heading provide high quality ground truth and accurate nonmagnetic heading that is comparable to or better than most non-fiber-optic gyros. The Doppler velocity log (DVL) and inertial measurement unit (IMU) are similar to sensors on a standard mid-grade AUV.

For this data set, the multibeam uncertainty was not reported directly from the sensor. Therefore, an estimate of 10 cm was determined by computing the vertical standard deviation of points collected on a flat plane and assumed for each beam. A more principled method using a GPR to fit multibeam data in range-angle space before transforming it to Cartesian coordinates was proposed in Barkby et al. (2012). Although this could be used, this method was not implemented to avoid the additional computational load.

Several metrics were computed to analyze performance and are shown in Figures 19 and 20. The average error and variance of the leaf particle's position from the GPS ground truth was computed to assess the spread of the particle cloud. The number of leaf particles over time was recorded along with the size of the Cholesky factor in GB. The compute time of each particle step was recorded and then summed over time to calculate the cumulative compute time. This value can be thought of as the cumulative time to compute the solution up to a given time step. If this value remains below the real-time line the solution was computed in less time than it took to acquire the data.

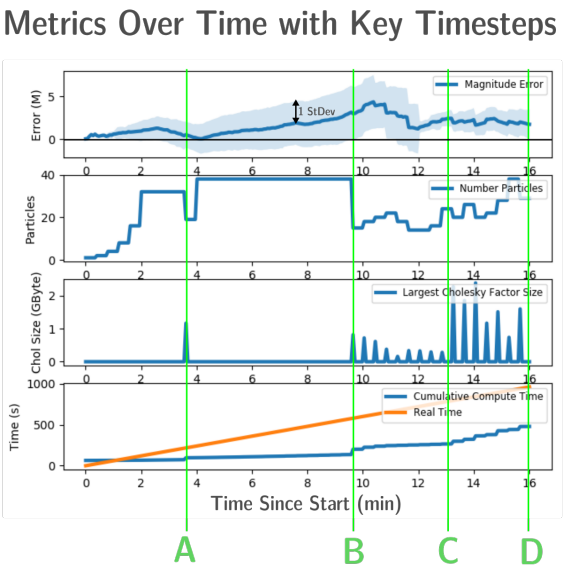
Stepping through a few critical timestamps from the start of a survey until the first loop closure better illustrates the main aspects of the GP-BPSLAM implementation. Figure 19 shows several key time points with the filter set to run with the parameters detailed in Table 1. The POS was traveling at approximately 5 knots with the multibeam sonar pinging at 15–20 Hz (depending on water depth) collecting 256 range returns per ping.

Start to Time A

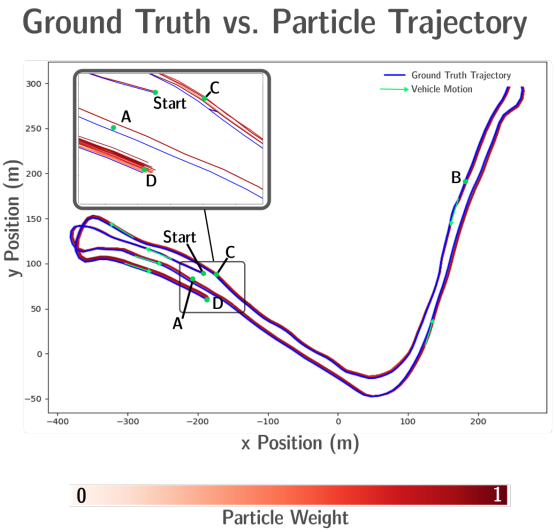
At the start, the filter is accumulating data and has not yet been able to perform a loop closure or resampling step up to time A. The number of particles doubles every time resampling is attempted because no particles are removed and each particle spawns two children. Eventually, the number of particles reaches the user-set limit (32) and particles are no longer allowed to spawn more than one child. The blocks associated with the worker's particle are added to the GPR recursively (Section 3).

Table 1. A summary of the tunable parameters in the GP-BPSLAM filter and the values used for testing.

Parameter	Value Used	Description
Particle Lifespan	4 s	Integration time of each particle's odometry trajectory (Figure 15)
Steps Between Resample	6	How many particle lifespans between resampling steps (Figure 15)
Min Model Particle Age	120 s	How old a particle map must be before it can be used in a loop closure (Figure 15)
Number of Children	2	How many children a particle can produce if resampled (Figure 15)
Max Particles	32	If the number of particles in the leaf queue is less than this "number of children," particles will be created for each particle in the queue (Figure 15)
Min Particles	8	The fewest particles that can exist after removing the least likely particles just *before* resampling (Figure 15)
Block Size	800	The number sonar points per block (Section 5.1)
GPR Length Scale	4 m	The GPR length scale hyperparameter (Section 5.1)
GPR Process Noise	0.1	The GPR process noise hyperparameter (Section 5.1)



(a) Particle metrics over time with key time steps labeled.



(b) Particle trajectories and ground truth trajectory with key time steps labeled.

Figure 19. A set of graphs showing the state of the GP-BPSLAM filter over time. A: The first resampling step. B: Continuous loop closures begin as vehicle doubles back along its prior path. C: Loop closures performed at the beginning of the trajectory. D: Reaching steady state to continue surveying.

To avoid premature loop closures, the map data must be older than a user-defined age (120 s) to be considered. At time A, sufficiently old data are encountered and the first resampling step can be completed. This results in a drop in particle count and a small drop in particle cloud variance.

Time A to Time B

The filter is exploring new terrain during this period and relying entirely on odometry. The average error of the particle cloud relative to the true GPS track grows steadily due to sensor biases, likely in the DVL, or a heading offset. The velocity noise added by our filter causes the size of the particle cloud to grow. Over this period, the error remains within one standard deviation of the true position. This implies the filter is adding enough noise to the particle cloud to properly represent the error in the navigation solution.

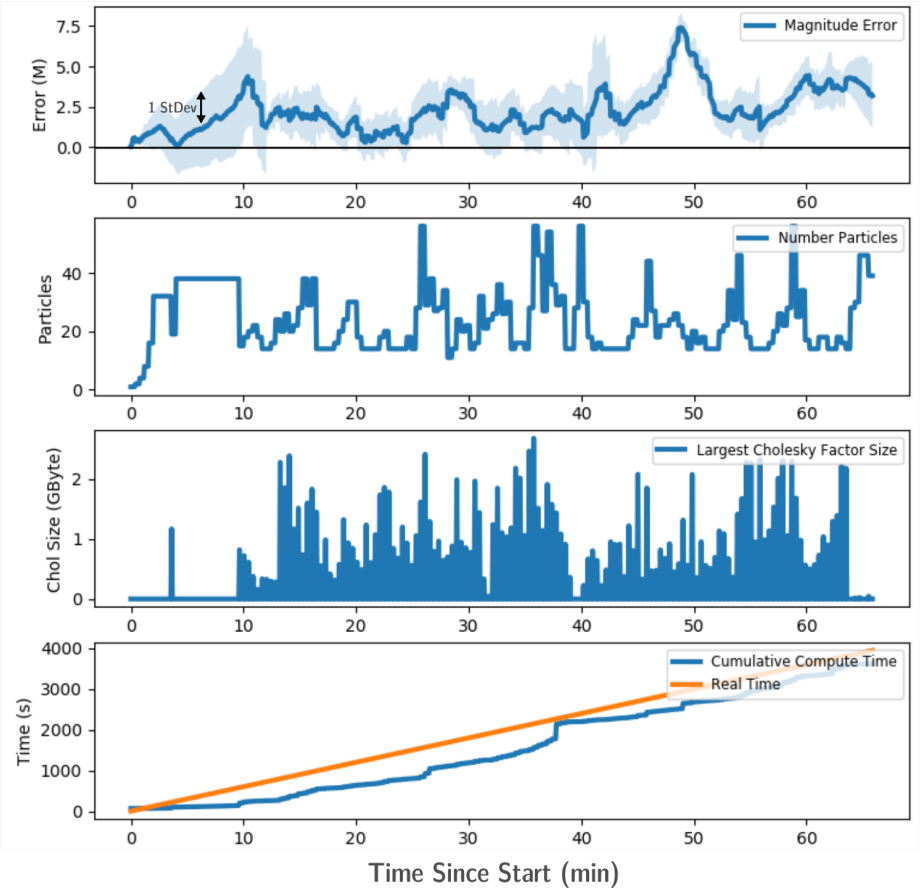


Figure 20. A set of graphs showing the state of the GP-BPSLAM filter for the entire run time.

Time B to Time C

At time B, the multibeam swath overlaps a previous trajectory and the filter performs a loop closure and resampling with the prior trajectory map. Over this entire period the variance of the particle cloud decreases along with the absolute error. Many outlying particles and trajectories are removed during successive resampling steps.

The blocks associated with the worker's particle are added to the GPR recursively and this time period is insightful for filter performance. We can see that the first resampling takes significant time relative to the rest of this period with each successive resampling taking less time. There are two reasons for this. First, simply dropping the number of particles means there are fewer leaf particle trajectories to compute. After the first drop, however, the particle cloud remains relatively constant. At that point, performance improvements are largely due to the recursive GPR worker. As time progresses, more parent trajectories are removed, meaning that leaf particles will have more similar Cholesky factors and fewer will need to be recomputed. In other words, the initial particle weight computation is dominated by Cholesky factor computation and later the computation becomes dominated by likelihood estimation.

Time C to Time D

During this time period, the filter begins to reach steady state. We can see that the slope of the cumulative compute time graph roughly matches the rate sonar pings are being acquired. The variance of the particle cloud remains constrained, with the magnitude of the error staying within one or two standard deviations of the ground truth solution.

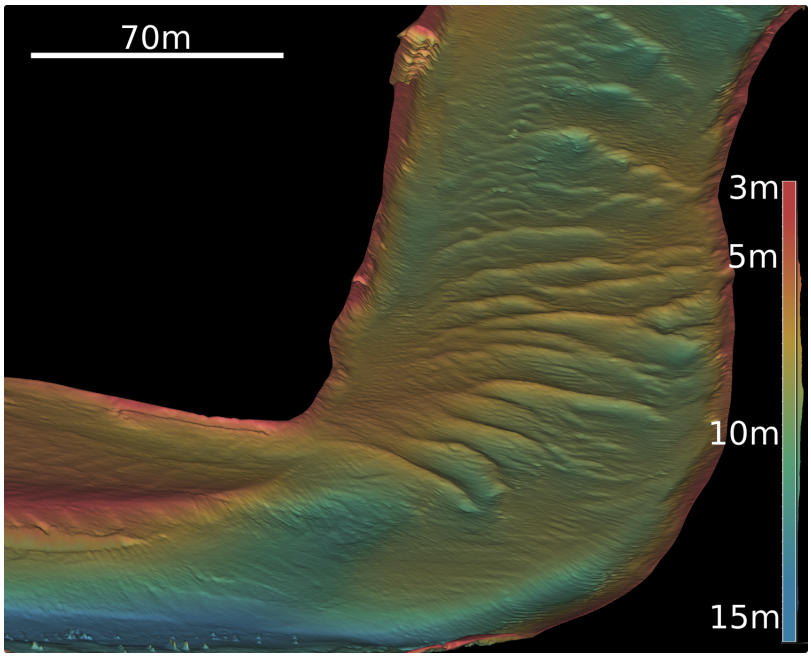


Figure 21. A finished map produced by the GP-BPSLAM filter.

Time D and Beyond

Figure 20 shows the same set of graphs for the entire duration of the survey. The magnitude of the error remains bounded below 5 m for most of the run. When the error sharply exceeds 5 m, it quickly converges again. The variance of the point cloud stays around ±1 m for the duration of the test. The compute time stays below the sonar acquisition time for the entire survey, demonstrating that in this configuration the filter is able to run in real time. The maximum GPU memory used at any point was less than 2.5 GB. The final map can be seen in Figure 21.

Testing shows that our GP-BPSLAM implementation produced more self-consistent maps than dead reckoning. Figure 22 shows the vertical standard deviation of our map calculated using a 30-cm grid. Over most of the survey area GP-BPSLAM produces a map that has lower standard deviation per cell than simple dead reckoning and slightly higher than the ground truth. Figure 23 shows a position error plot for a simple dead reckoning solution with consistently higher error.

6. Discussion

Although our implementation of BPSLAM was able to run in real time, it still requires significant computing power. For these experiments, an NVIDIA 2080ti was used. This type of device is not currently likely to be found on an AUV, but the capabilities of embedded GPUs are increasing at a rapid rate. However, remotely operated vehicles (ROVs) are often used in mapping work and still have a need for online navigation. A system with ample GPU power is realistic for an ROV topside system.

When considering GPU GP-BPSLAM for AUVs, there are a few factors to weigh. Although less powerful than desktop systems, embedded systems, such as the NVIDIA Jetson NX Xavier, are developing and have significantly lower power consumption and heat generation than desktop systems, making them viable on AUVs. On many AUVs, data are collected at a lower ping rate. For example, the Woods Hole Oceanographic Institution Sentry AUV collects multibeam data at 5 Hz as opposed to the 15–20 Hz on the POS. This significantly lower data rate makes embedded systems much more viable.

To further reduce compute times, several principled methods for approximated GPRs are available (Rasmussen and Williams, 2006). These methods could be applied to maintain real-time performance

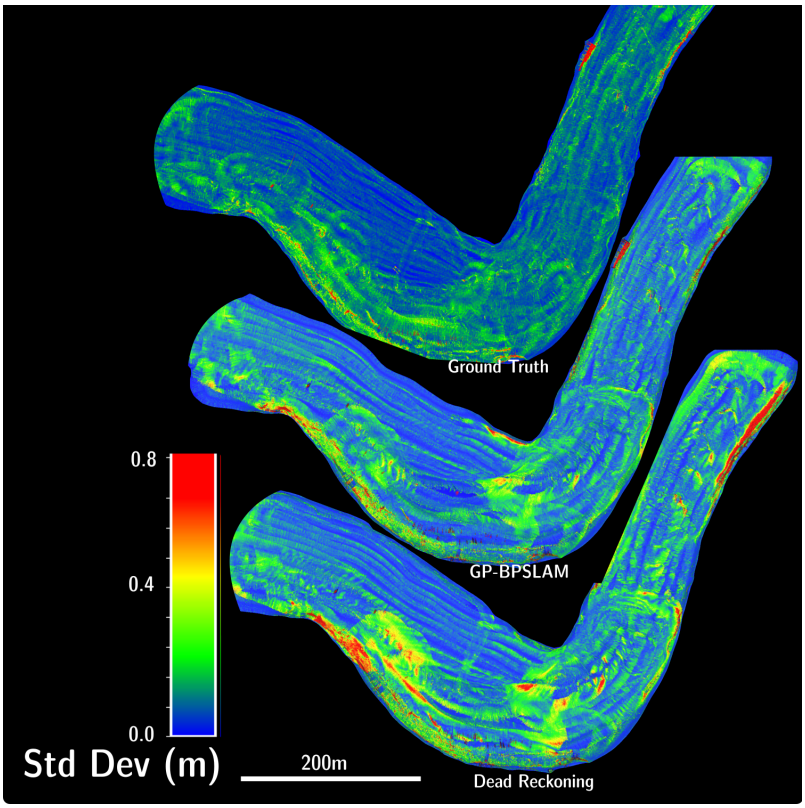


Figure 22. A comparison of the variation of measurements per cell between the ground truth, the GP-BPSLAM filter, and dead reckoning.

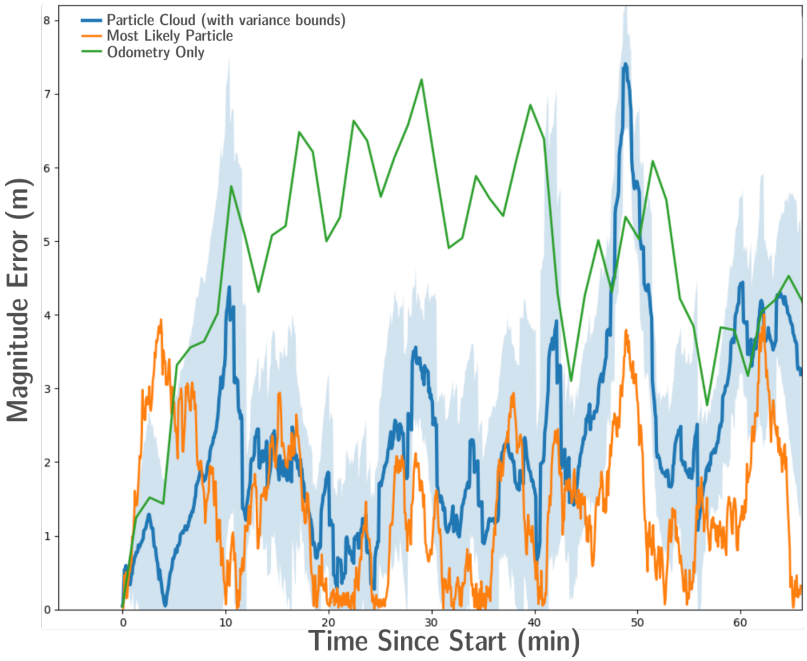


Figure 23. A plot comparing the position error of the particle cloud and the most likely particle trajectory over time.

on a wide range of hardware. Areas of very high point density are often the cause of sharp increases in compute times. These high point densities typically occur when the vehicle stops, makes a tight turn, or repeatedly crosses a previous path. In these cases, the seafloor is oversampled and it is possible to use approximate GPR computation in areas of high point density without significant loss of model quality. The sparse pseudo-input approximation (Snelson and Ghahramani, 2005) is compatible with the Cholesky decomposition method presented here and could place an upper bound on compute times.

Additional optimizations could also be made to our existing code that would increase GPU utilization and further increase performance when resampling. Since the Cholesky computation is shared between many leaf nodes in our recursive Cholesky factor, optimizations made to the prediction step of the GPR would be particularly helpful. It may also be possible to improve performance of the filter itself. The particle count has a significant impact on the accuracy and reproducibility of the navigation solution. In its current form, GP-BPSLAM's most expensive step is the GPR computation. This expensive computation requires a lower particle count to achieve real-time operation. It may be possible to allow each leaf particle to optimize its position (and associated trajectory) slightly based on the GPR model. In this way, it may be possible to achieve better self-consistency in the final map without increasing the particle count or significantly increasing computation time. This extension could hybridize the robustness of the Rao-Blackwellized particle filter and the optimally of submap SLAM.

7. Future Work

Future development of MP-GPR will likely include the approximations discussed above to increase the overall stability of the software and allow the method to more easily run on embedded GPU hardware such as an NVIDIA Jetson. The MP-GPR would also benefit from a nonstationary kernel implementation. This would allow a single model to adjust to highly variable seafloor topography.

Since GP-BPSLAM can currently maintain relatively few particle trajectories, we believe it would receive a major performance boost by hybridizing the particle filter approach with a submap SLAM approach. At each culling step, after unlikely trajectories are removed, the remaining trajectories would be allowed to optimize themselves slightly to minimize internal error. This optimization could be done without needing to recompute the GPR for each trajectory and thus could be done relatively efficiently. This would still allow for multiple hypotheses but could potentially yield a more accurate final product.

8. Conclusion

We have presented a novel way to compute a Gaussian process regression (GPR) for arbitrarily large data sets using massively parallel GPU programming. Since a GPR can predict model uncertainty (variance) over its domain, it has an advantage over traditional gridding or spline-based techniques for information driven path planning. To the best of our knowledge, this is the first time a GPR solver has been used to operate on a high resolution high rate bathymetric data in real time.

Our solver is able to process large data sets by using a sparsification technique compatible with massively parallel processing. The memory requirements are further reduced by only computing part of the solution at any given time. We also present a way to perform a Cholesky decomposition and online Cholesky update on the BSR matrices in an incremental way. We demonstrated that the solver was able to generate maps in real time using data acquired from our surface vessel.

We also showed that it is now possible to compute a BPSLAM navigation solution in real time using GPR trajectory maps. Our approach relies on efficient data management and formulating the GPR computations using a sparse approximation with a recursive Cholesky factor. Using RTK GPS ground truth on a surface vessel, we were able to show that our solution performed better than dead reckoning and had bounded position error. Additionally, our method was able to produce more self-consistent maps than simple dead reckoning.

Acknowledgments

This work was funded by the Office of Naval Research under award number N000141712467. The authors are grateful for the support of the Ocean Exploration Trust (OET), and the crew and operations team aboard E/V *Nautilus*.

ORCID

Kristopher E. Krasnosky https://orcid.org/0000-0002-2215-7792
Christopher Roman https://orcid.org/0000-0002-9185-4532

References

Balasuriya, B., Chathuranga, B., Jayasundara, B., Napagoda, N., Kumarawadu, S., Chandima, D., and Jayasekara, A. (2016). Outdoor robot navigation using Gmapping based SLAM algorithm. In *2016 Moratuwa Engineering Research Conference (MERCon)*, pages 403–408. IEEE.

Barkby, S., Williams, S. B., Pizarro, O., and Jakuba, M. V. (2011). Bathymetric SLAM with no map overlap using Gaussian processes. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1242–1248. IEEE.

Barkby, S., Williams, S. B., Pizarro, O., and Jakuba, M. V. (2012). Bathymetric particle filter SLAM using trajectory maps. *International Journal of Robotics Research*, 31(12):1409–1430.

Bichucher, V., Walls, J. M., Ozog, P., Skinner, K. A., and Eustice, R. M. (2015). Bathymetric factor graph SLAM with sparse point cloud alignment. In *OCEANS 2015-MTS/IEEE Washington*, pages 1–7. IEEE.

Bore, N., Torroba, I., and Folkesson, J. (2018). Sparse Gaussian process SLAM, storage and filtering for AUV multibeam bathymetry. In *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, pages 1–6. IEEE.

Calder, B. (2003). Automatic statistical processing of multibeam echosounder data. *International Hydrographic Review*, 4(1):53–68.

Cressie, N. (1990). The origins of kriging. *Mathematical Geology*, 22(3):239–252.

Franey, M., Ranjan, P., and Chipman, H. (2012). A short note on Gaussian process modeling for large datasets using graphics processing units. *arXiv preprint arXiv:1203.1269*.

Galceran, E., Campos, R., Palomeras, N., Ribas, D., Carreras, M., and Ridao, P. (2015). Coverage path planning with real-time replanning and surface reconstruction for inspection of three-dimensional underwater structures using autonomous underwater vehicles. *Journal of Field Robotics*, 32(7):952–983.

Galceran, E., Nagappa, S., Carreras, M., Ridao, P., and Palomer, A. (2013). Uncertainty-driven survey path planning for bathymetric mapping. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 6006–6012. IEEE.

Gramacy, R. B. and Apley, D. W. (2015). Local Gaussian process approximation for large computer experiments. *Journal of Computational and Graphical Statistics*, 24(2):561–578.

Gramacy, R. B., Niemi, J., and Weiss, R. M. (2014). Massively parallel approximate Gaussian process regression. *SIAM/ASA Journal on Uncertainty Quantification*, 2(1):564–584.

Hitchcox, T. and Forbes, J. R. (2020). A point cloud registration pipeline using Gaussian process regression for bathymetric SLAM. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4615–4622. IEEE.

Kohlbrecher, S., Meyer, J., von Stryk, O., and Klingauf, U. (2011). A flexible and scalable SLAM system with full 3D motion estimation. In *Proceedings of the IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE.

Krasnosky, K., Roman, C., and Casagrande, D. (2021). A bathymetric mapping and SLAM data set with high precision ground truth for marine robotics. *International Journal of Robotics Research*, 41(1):12–19.

Lang, T., Plagemann, C., and Burgard, W. (2007). Adaptive non-stationary kernel regression for terrain modeling. In *Robotics: Science and Systems*, volume 6.

Ma, T., Li, Y., Wang, R., Cong, Z., and Gong, Y. (2018). AUV robust bathymetric simultaneous localization and mapping. *Ocean Engineering*, 166:336–349.

Massot-Campos, M., Oliver, G., Bodenmann, A., and Thornton, B. (2016). Submap bathymetric SLAM using structured light in underwater environments. In *2016 IEEE/OES Autonomous Underwater Vehicles (AUV)*, pages 181–188. IEEE.

Melkumyan, A. and Ramos, F. (2009). A sparse covariance function for exact Gaussian process inference in large datasets. In *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI)*, volume 9, pages 1936–1942.

Moore, T. and Stouch, D. (2014). A generalized extended Kalman filter implementation for the robot operating system. In *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer.

Murphy, K. P. (1999). Bayesian map learning in dynamic environments. *Advances in Neural Information Processing Systems*, 12:1015–1021.

NVIDIA, Vingelmann, P., and Fitzek, F. H. (2020). CUDA, release: 2 October 1989.

Pairet, È., Hernández, J. D., Lahijanian, M., and Carreras, M. (2018). Uncertainty-based online mapping and motion planning for marine robotics guidance. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2367–2374. IEEE.

Palomer, A., Ridao, P., and Ribas, D. (2016). Multibeam 3D underwater SLAM with probabilistic registration. *Sensors*, 16(4):560.

Peng, D., Gao, J., and Zhou, T. (2019). Underwater terrain matching navigation based on Gaussian process regression with a multi-beam bathymetric sonar. *Journal of the Acoustical Society of America*, 146(4):3089–3089.

Qi, Y., Abdel-Gawad, A. H., and Minka, T. P. (2010). Sparse-posterior Gaussian processes for general likelihoods. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence*, pages 450–457. Citeseer.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press.

Roman, C. and Singh, H. (2005). Improved vehicle based multibeam bathymetry using sub-maps and SLAM. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3662–3669. IEEE.

Scheidegger, C. E., Fleishman, S., and Silva, C. T. (2005). Triangulating point set surfaces with bounded error. In *Symposium on Geometry Processing*, pages 63–72. Citeseer.

Shi, J. and Zhou, M. (2020). A data-driven intermittent online coverage path planning method for AUV-based bathymetric mapping. *Applied Sciences*, 10(19):6688.

Snelson, E. and Ghahramani, Z. (2005). Sparse Gaussian processes using pseudo-inputs. *Advances in Neural Information Processing Systems*, 18:1257–1264.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A. Y., et al. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan.

Torroba, I., Bore, N., and Folkesson, J. (2018). A comparison of submap methods for multibeam bathymetric mapping. In *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, pages 1–6.

Torroba, I., Bore, N., and Folkesson, J. (2019). Towards autonomous industrial-scale bathymetric surveying. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6377–6382.

VanMiddlesworth, M., Kaess, M., Hover, F., and Leonard, J. J. (2015). *Mapping 3D Underwater Environments with Smoothed Submaps*, pages 17–30. Springer International Publishing, Cham.

Vasudevan, S., Ramos, F., Nettleton, E., and Durrant-Whyte, H. (2009). Gaussian process modeling of large-scale terrain. *Journal of Field Robotics*, 26(10):812–840.

Vasudevan, S., Ramos, F., Nettleton, E., and Durrant-Whyte, H. (2011). Non-stationary dependent Gaussian processes for data fusion in large-scale terrain modeling. In *2011 IEEE International Conference on Robotics and Automation*, pages 1875–1882. IEEE.

Vaughn, J. I. (2015). *Factor Graphs and Submap Simultaneous Localization and Mapping for Micro-bathymetry*. PhD thesis, University of Rhode Island.

Wilson, T. and Williams, S. B. (2018). Adaptive path planning for depth-constrained bathymetric mapping with an autonomous surface vessel. *Journal of Field Robotics*, 35(3):345–358.

How to cite this article: Krasnosky, K. E., & Roman, C. (2022). A massively parallel implementation of gaussian process regression for real time bathymetric modeling and simultaneous localization and mapping. *Field Robotics, 2*, 940–970.

Appendix A: Detailed Methods

Online Cholesky Update

The online Cholesky update allows a GPR to be updated as new data become available without a complete recomputation. For the parallel implementation, it allows us to reduce the memory requirements needed to compute the complete Cholesky factorization and optimizes each step for computation in parallel.

Using a lower triangular Cholesky factor L_{11} and a covariance matrix $\mathbf{K}_{11}$, such that $\mathbf{K}_{11} = \mathbf{L}_{11}\mathbf{L}_{11}^{\intercal}$, the covariance matrix will become $\begin{bmatrix} \boldsymbol{K}_{11} & \boldsymbol{K}_{12} \\ \boldsymbol{K}_{12}^{\intercal} & \boldsymbol{K}_{22} \end{bmatrix}$ after new data are added. From this, we want to calculate the Cholesky factor matrix $\begin{bmatrix} \boldsymbol{S}_{11} & 0 \\ \boldsymbol{S}_{21} & \boldsymbol{S}_{22} \end{bmatrix}$. Because $\boldsymbol{L}_{11}$ is lower triangular, we can use substitution to solve for

$$\begin{aligned} \boldsymbol{S}_{11} &= \boldsymbol{L}_{11}, \\ \boldsymbol{S}_{21}^{\intercal} &= \boldsymbol{L}_{11}/K_{12}, \\ \boldsymbol{S}_{22} &= Chol(\boldsymbol{K}_{22} - \boldsymbol{S}_{21}\boldsymbol{S}_{21}^{\intercal}). \end{aligned} \tag{15}$$

Back Solving BSR Matrix

The BSR solution is organized as a triangular block matrix system

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & \boldsymbol{A}_{13} \\ \mathbf{0} & \boldsymbol{A}_{22} & \boldsymbol{A}_{23} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{A}_{33} \end{bmatrix} \begin{bmatrix} \boldsymbol{X}_1 \\ \boldsymbol{X}_2 \\ \boldsymbol{X}_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{B}_1 \\ \boldsymbol{B}_2 \\ \boldsymbol{B}_3 \end{bmatrix},$$

where $\boldsymbol{A}_{ij}$ are $n \times n$ dense matrices and $\boldsymbol{B}_i$ and $\boldsymbol{X}_i$ are $n \times m$ matrices. This system can be solved by

$$\begin{aligned} \boldsymbol{X}_3 &= \boldsymbol{B}_3/\boldsymbol{A}_{33}, \\ \boldsymbol{X}_2 &= [\boldsymbol{B}_2 - \boldsymbol{A}_{23}\boldsymbol{X}_3]/\boldsymbol{A}_{22}, \\ \boldsymbol{X}_1 &= [\boldsymbol{B}_1 - \boldsymbol{A}_{12}\boldsymbol{X}_2 - \boldsymbol{A}_{13}\boldsymbol{X}_3]/\boldsymbol{A}_{\mathbf{11}}, \end{aligned}$$

where $x = \boldsymbol{B}/\boldsymbol{A}$ represents the solution to the linear system $\boldsymbol{A}x = \boldsymbol{B}$.

The matrix multiplications needed to compute $\boldsymbol{X}_2$ and $\boldsymbol{X}_1$ can be done extremely quickly by using NVIDIA Tensor cores.

Hyperparameter Optimization

The marginal likelihood, $p(\boldsymbol{Y}|\boldsymbol{X})$, provides a metric to quantify the quality of a given hyperparameter selection. Usually represented as the log of the marginal likelihood (LML), it is defined as (Rasmussen and Williams, 2006)

$$\log \mathrm{P}(\boldsymbol{Y}|\boldsymbol{X}) = -\frac{1}{2}\boldsymbol{Y}^{\intercal}\boldsymbol{V}^{-1}\boldsymbol{Y} - \frac{1}{2}\log|\boldsymbol{V}| - \frac{M}{2}\log(2\pi). \tag{16}$$

In practice, $\boldsymbol{V}$ is represented as our Cholesky factor so the LML can be reformulated as

$$\log \mathrm{P}(\boldsymbol{Y}|\boldsymbol{X}) = -\frac{1}{2}\boldsymbol{Y}^{\intercal}\boldsymbol{\alpha} - \frac{1}{2}\log|\boldsymbol{L}\boldsymbol{L}^{\intercal}| - \frac{M}{2}\log(2\pi), \tag{17}$$

where $\boldsymbol{\alpha} = \boldsymbol{V}^{-1}\boldsymbol{Y} = CholeskySolve(\boldsymbol{L}, \boldsymbol{Y})$. The term $\log|\boldsymbol{L}\boldsymbol{L}^{\intercal}|$can be represented as $\log(|\boldsymbol{L}||\boldsymbol{L}^{\intercal}|)$. Noting that the determinant of a triangular matrix is simply the product of the diagonal elements, $|\boldsymbol{L}| = |\boldsymbol{L}^{\intercal}| = \prod_{i=0}^{n}\boldsymbol{L}_{i,i}$, $\log(|\boldsymbol{L}||\boldsymbol{L}^{\intercal}|)$ can be simplified as $\log((\prod_{i=1}^{M}\boldsymbol{L}_{i,i})^2)$ and finally

$2\log(\Sigma_{i=1}^{M}\boldsymbol{L}_{i,i})$. The LML can then be written as

$$\log p(\boldsymbol{Y}|\boldsymbol{X}) = -\frac{1}{2}\boldsymbol{Y}^{\intercal}\boldsymbol{\alpha} - \log(\Sigma_{i=1}^{M}\boldsymbol{L}_{i,i}) - \frac{M}{2}\log(2\pi), \tag{18}$$

where the Cholesky solve for $\boldsymbol{\alpha}$ can be done in parallel on a block matrix as described in this Appendix. The sum $\Sigma_{i=1}^{M}\boldsymbol{L}_{i,i}$ can be computed using an atomic add along the diagonal of each $\boldsymbol{L}$ matrix block.

In order to optimize the LML using gradient descent, it is useful to know its gradient explicitly. The gradient of the LML can be calculated as follows (Rasmussen and Williams, 2006):

$$\frac{\partial}{\partial\theta}\log(\boldsymbol{Y}|B) = \frac{1}{2}tr(\boldsymbol{\alpha}\boldsymbol{\alpha}^{\intercal} - \boldsymbol{V}^{-1}\frac{\partial\boldsymbol{V}}{\partial\theta}). \tag{19}$$

The partials $\frac{\partial\boldsymbol{V}}{\partial\theta}$ can usually be determined analytically from the kernel function. In the case of the exactly sparse square exponential, they are

$$\frac{\partial\boldsymbol{V}}{\partial l} = \sigma\left(\frac{2\pi d\left(1-\frac{d}{l}\right)\sin\left(\frac{2\pi d}{l}\right)}{3l^2} - \frac{d\cos\left(\frac{2\pi d}{l}\right)}{l^2} + \frac{d\left(\cos\left(\frac{2\pi d}{l}\right)+2\right)}{3l^2}\right), \tag{20}$$

$$\frac{\partial\boldsymbol{V}}{\partial\sigma} = \frac{\sin\left(\frac{2\pi d}{l}\right)}{2\pi} + \frac{1}{3}\left(1-\frac{d}{l}\right)\left(\cos\left(\frac{2\pi d}{l}\right)+2\right), \tag{21}$$

where $d = \sqrt{x_1^2+x_2^2}$. In our implementation of the Gaussian process regression, the covariance matrix $\boldsymbol{V}^{-1}$is never directly computed or stored. Instead, $\boldsymbol{V}^{-1}$ is computed by solving

$$\boldsymbol{V}^{-1} = CholeskySolve(\boldsymbol{L},\boldsymbol{I}), \tag{22}$$

and the gradient of the LML is then defined entirely in terms of L.

However, because of the need to compute V^{-1}directly from L, the memory requirements of the LML partial computation are significantly larger than those of the regression since L^{-1} needs to be stored in addition to L. For larger data sets, this can push the limits of modern GPUs. Fortunately, optimizing the hyperparameters can be done periodically using a smaller subset of data. This is usually sufficient for most survey applications where the vessel or vehicle configurations tend to remain consistent over time.